

Neural Networks.

Lab 3: Multi layer perceptrons. Nonlinear regression and prediction.

1. Defining multi layer perceptrons.

A multi layer perceptron (i.e. feedforward neural networks with hidden layers) contains at least two layers of functional units. This means that at least one layer contains hidden units, which do not communicate with the environment. If the number of hidden units is appropriately chosen multi layer perceptrons are universal approximators, i.e. they can solve, at least theoretically, any association problem (nonlinearly separable classification, nonlinear regression and prediction problems).

The function used in Matlab to define a multi layer perceptron is `newff`. In the case of a network having `m` layers of functional units (i.e., `m-1` hidden layers) the syntax of the function is:

```
ffnet = newff(inputData, desiredOutputs, {k1,k2,...,km-1}, {f1,f2,...,fm}, '<training algorithm>')
```

The significance of the parameters involved in `newff` is:

inputData: a matrix containing on its columns the input data in the training set. Based on this, `newff` establishes the number of input units.

desiredOutputs: a matrix containing on its columns the correct answers in the training set. Based on this, `newff` establishes the number of output units.

{k₁,k₂,...,k_{m-1}}: the number of units on each hidden layer

{f₁,f₂,...,f_m}: the activation functions for all functional layers. Possible values are: 'logsig', 'tansig' and 'purelin'. The hidden layers should have nonlinear activation functions ('logsig' or 'tansig') while the output layer can have also linear activation function.

'<training algorithm>': is a parameter specifying the tip of the learning algorithm. There are two main training variants: incremental (the training process is initiated by the function `adapt`) and batch (the training process is initiated by the function `train`). When the function `adapt` is used the possible values of this parameter are: 'learngd' (it corresponds to the serial standard BackPropagation derived using the gradient descent method), 'learngdm' (it corresponds to the momentum variant of BackPropagation), 'learngda' (variant with adaptive learning rate). When the function `train` is used the possible values of this parameter are: 'traingd' (it corresponds to the classical batch BackPropagation derived using the gradient descent method), 'traingdm' (it corresponds to the momentum variant of BackPropagation), 'trainlm' (variant based on the Levenberg Marquardt minimization method).

Remark 1: the learning function can be different for different layers. For instance by `ffnet.inputWeights{1,1}.learnFcn='learngd'` one can specify which is the learning algorithm used for the weights between the input layer and the first layer of functional units.

Exemple 1. Definition of a network used to represent the XOR function:

```
ffnet=newff([0 1 0 1;0 0 1 1],[0 1 1 0],[5,1],{'logsig','logsig'},'traingd')
```

2. Training multi layer perceptrons.

There are two training functions: `adapt` (incremental learning) and `train` (batch learning).

2.1. Adapt.

The syntax of this function is:

```
[ffTrained,y,err]=adapt(ffnet, inputData, desiredOutputs)
```

There several parameters which should be set before starting the learning process:

- Learning rate - lr: the implicit value is 0.01
- Number of epochs - passes (number of passes through the training set): the implicit value is 1
- Momentum coefficient – mc (when 'learngdm' is used): the implicit value is 0.9

Example: `ffnet.adaptParam.passes=10; ffnet.adaptParam.lr=0.1;`

2.1. Train.

The syntax of this function is:

```
[ffTrained,y,err]=train(ffnet, inputData, desiredOutputs)
```

There several parameters which should be set before starting the learning process:

- Learning rate - lr: the implicit value is 0.01
- Number of epochs - epochs (number of passes through the training set): the implicit value is 1
- Maximal value of the error: goal
- Momentum coefficient – mc (when 'learngdm' is used): the implicit value is 0.9

Example: `ffnet.trainParam.passes=1000; ffnet.trainParam.lr=0.1; ffnet.trainParam.goal=0.001;`

3. Simulating multi layer perceptrons.

In order to compute the output of a network one can use the function `sim`, having the syntax:

```
sim(ffTrained, testData)
```

4. Applications.

4.1. Representation of XOR.

Variant 1: Let us consider that the function is defined on $\{-1,1\}$ and also takes values in $\{-1,1\}$.

```

in=[-1 -1 1 1;-1 1 -1 1]; d=[-1 1 1 -1];
nnXOR=newff(in,d,5,{'tansig','tansig'},'traingd');
nnXOR.trainParam.epochs=1000; nnXOR.trainParam.goal=0.01;
nnXOR.trainParam.lr=0.5;
nnXORtrained=train(nnXOR,in,d);
res=sim(nnXORtrained,in);
disp('Result:'); disp(res);

```

Exercises:

1. Try different training algorithms: 'traingdm', 'trainlm'
2. Replace train with adapt (change accordingly the parameters).
2. Modify the above script for the variant when XOR is defined on {0,1} and takes values in {0,1}.

4.2. Linear and nonlinear regression

Let us consider a set of bidimensional data represented as points in plane (the coordinates are specified by using the function `ginput`). Find a linear or a nonlinear function which approximates the data (by minimizing the sum of squared distances between the points and the graph of the function).

```

function [in,d]=regression(layers,hiddenUnits,training,cycles)
% if layers = 1 then a single layer linear neural network will be
created
% if layers = 2 then a network with one hidden layer will be created
and
%             hiddenUnits denote the number of units on the hidden
layer
% training = 'adapt' or 'train'
% cycles = the number of passes (for adapt) and epochs (for train)

% read the data
clf
axis([0 1 0 1])
hold on
in = [];
d = [];
n = 0;
b = 1;
while b == 1
[xi,yi,b] = ginput(1);
plot(xi,yi,'r*');
n = n+1;
in(1,n) = xi;
d(1,n) = yi;
end
inf=min(in); sup=max(in);
% define the network
if (layers==1)
    reta=newlind(in,d); % linear network designed starting from input
data
else
    ret=newff(in,d,hiddenUnits,{'tansig','purelin'},'traingd');

```

```

if(training == 'adapt')
    % setting the parameters
    ret.adaptParam.passes=cycles;
    ret.adaptParam.lr=0.05;
    % network training
    reta=adapt(ret,in,d);
else
    ret.trainParam.epochs=cycles;
    ret.trainParam.lr=0.05;
    ret.trainParam.goal=0.001;
    % network training
    reta=train(ret,in,d);
end
end
% graphical representation of the approximation
x=inf:(sup-inf)/100.:sup;
y=sim(reta,x);
plot(in,d,'b*',x,y,'r-');
end

```

Exercises:

1. Test the ability of function `regression` to approximate different types of data.
Hint: for linear regression call the function as: `regression(1,0,'adapt', 1)` (the last two parameters are ignored)
for nonlinear regression the call should be: `regression(2,5,'adapt', 5000)`
2. In the case of nonlinear regression analyze the influence of the number of hidden units. Values to test: 5 (as in the previous exercise), 10 and 20. In order to test the behavior of different architectures on the same set of data save the data specified in the first call and define a new function which does not read the data but receive them as parameters.

4.3. Prediction

Let us consider a sequence of data (a time series): x_1, x_2, \dots, x_n which can be interpreted as values recorded at successive moments of time. The goal is to predict the value corresponding to moment $(n+1)$. The main idea is to suppose that a current value x_i depends on N previous values: $x_{i-1}, x_{i-2}, \dots, x_{i-N}$. Based on this hypothesis we can design a neural network which is trained to extract the association between any subsequence of L and the next value in the series.

Therefore, the neural network will have N input units, a given number of hidden units and 1 output unit. The training set will have $n-N$ pairs of (input data, correct output):

$$\{(x_1, x_2, \dots, x_N), x_{N+1}, ((x_2, x_3, \dots, x_{N+1}), x_{N+2}), \dots, ((x_{n-N}, x_{n-N+1}, \dots, x_{n-1}), x_n)\}.$$

A solution in Matlab is:

```

function [in,d]=prediction(data,inputUnits,hiddenUnits,training,epochs)
% data : the series of data (one row matrix)
% inputUnits : number of previous data which influences the current one
% hiddenUnits : number of hidden units
% traininb= 'adapt' sau 'train'
% epochs = number of epochs
L=size(data,2);
in=zeros(inputUnits,L-inputUnits);
d=zeros(1,L-inputUnits);
for i=1:L-inputUnits
    in(:,i)=data(1,i:i+inputUnits-1);
    d(i)=data(1,i+inputUnits);
end
ret=newff(in,d,hiddenUnits,{'tansig','purelin'},'traingd');
if(training == 'adapt')
    ret.adaptParam.passes=epochs;
    ret.adaptParam.lr=0.05;
    reta=adapt(ret,in,d);
else
    ret.trainParam.epochs=epochs;
    ret.trainParam.lr=0.05;
    ret.trainParam.goal=0.001;
    reta=train(ret,in,d);
end
% graphical plot
x=1:L-inputUnits;
y=sim(reta,in);
inTest=zeros(inputUnits,1);
inTest=data(1,L-inputUnits+1:L)';
rezTest=sim(reta,inTest);
disp('Predicted value:'); disp(rezTest);
plot(x,d,'b-',x,y,'r-',L+1,rezTest,'k*');
end

```

Example: prediction(5,10,'adapt',2000)

Exercise.

1. Analyse the influence of the number of input units on the ability of the network to make prediction (Hint: try the following values: 2, 5, 10, 15)
2. Analyse the influence of the number of hidden units on the ability of the network to make prediction (Hint: the number of hidden units should be at least as large as the number of input units)
3. Analyse the influence of training algorithm on the ability of the network to make prediction