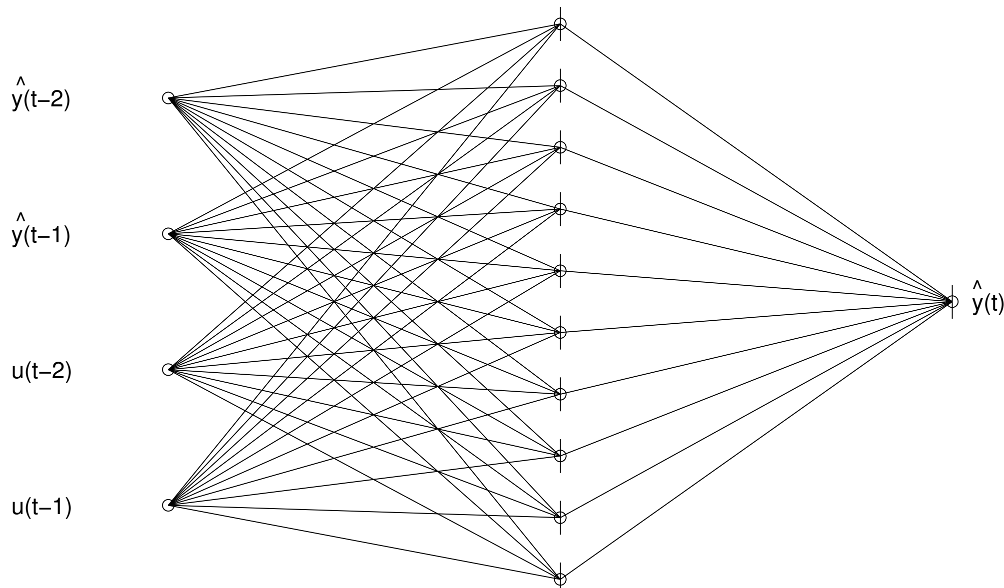
Neural Network Based System Identification

TOOLBOX

Version 2

For Use with MATLAB®



Magnus Nørgaard

Department of Automation
Department of Mathematical Modelling

Technical Report 00-E-891, Department of Automation

Technical University of Denmark

Release Notes

Neural Network Based
System Identification Toolbox

Version 2

Department of Automation, Technical University of Denmark, January 23, 2000

This note contains important information on how the present toolbox is to be installed and the conditions under which it may be used. Please read it carefully before use.

The note should be sufficient for being able to make the essential portion of the toolbox functions work properly. However, to enhance performance a number of functions have been rewritten in C and in order to compile these, it is necessary to read the information about CMEX files found in the MATLAB Application Program Interface Guide.

INSTALLING THE TOOLBOX

° Version 2.0 of the toolbox is developed for MATLAB 5.3 and higher. It has been tested under WINDOWS 98/NT, Linux, and HP9000/735. If you are running an older version of MATLAB you can run version 1.1 of the toolbox.

° All toolbox functions are implemented as plain m-files, but to enhance performance CMEX duplicates have been written for some of the most important functions. It is strongly recommended that the compilation be optimized with respect to execution speed as much as the compiler permits. Under MATLAB 5 it might be necessary to copy the file mexopts.sh to the working directory and modify it appropriately (ANSI C + max. optimization). To compile the MEX files under MATLAB 5 just type
>> *makemex*
in the MATLAB command window.

USING THE TOOLBOX

° The checks for incorrect functions calls are not very thorough and consequently MATLAB will often respond with quite incomprehensible error messages when a

function is passed the wrong arguments. When calling a CMEX-function, it may even cause MATLAB to crash. Hence, when using the CMEX functions it may be a good idea to make extra copies of the m-files they are replacing (do not just rename the m-files since they are still read by the “help” command). One can then start by calling the m-functions first to make sure the call is correct.

° The functions have been optimized with respect to speed rather than memory usage. For large network architectures and/or large data sets, memory problems may thus occur.

CONDITIONS/ DISCLAIMER

By using the toolbox the user agrees to all of the following.

° If one is going to publish any work where this toolbox has been used, please remember it was obtained free of charge and include a reference to this technical report (M. Nørgaard:”Neural Network Based System Identification Toolbox,” Tech. Report. 00-E-891, Department of Automation, Technical University of Denmark, 2000).

° Magnus Nørgaard and Department of Automation do <u>not</u> offer any support for this product <u>whatsoever</u>. The toolbox is offered free of charge - take it or leave it!

° The toolbox is copyrighted freeware by Magnus Nørgaard/Department of Automation, DTU. It may be distributed freely unmodified. It is, however, not permitted to utilize any part of the software in commercial products without prior written consent of Magnus Norgaard, The Department of Automation, DTU.

° THE TOOLBOX IS PROVIDED “AS-IS” WITHOUT WARRENTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRENTIES OR CONDITIONS OF MECHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MAGNUS NØRGAARD AND/OR THE DEPARTMENT OF AUTOMATION BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA, OR PROFITS, WHETHER OR NOT MN/IAU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND/OR ON ANY THEORY OF LIABILITY ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

MATLAB is a trademark of The MathWorks, Inc.
MS-Windows is a trademark of Microsoft Coporation.

Trademarks of other companies and/or organizations mentioned in this documentation appear for identification purposes only and are the property of their respective companies and/or organizations.

January 23, 2000

Magnus Nørgaard
Department of Automation, Building 326
Technical University of Denmark
2800 Lyngby
Denmark
e-mail: toolbox@magnusnorgaard.dk

1 Tutorial

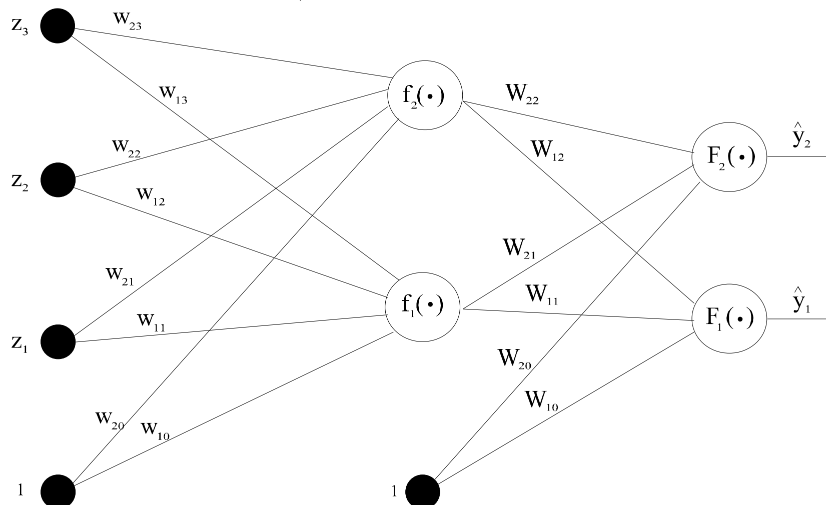
The present toolbox: “Neural Network Based System Identification Toolbox”, contains a large number of functions for training and evaluation of multilayer perceptron type neural networks. The main focus is on the use of neural networks as a generic model structure for the identification of nonlinear dynamic systems. The System Identification Toolbox provided by The MathWorks, Inc., has thus been a major source of inspiration in constructing this toolbox, but the functions work completely independent of the System Identification Toolbox as well as of the Neural Network Toolbox (also provoded by the MathWorks, Inc.). Although the use in system identification will be emphasized below, the tools made available here can also be used for time-series analysis or simply for training and evaluation of ordinary feed-forward networks (for example for curve fitting).

This chapter starts by giving a brief introduction to multilayer perceptron networks and how they may be trained. The rest of the tutorial will then address the nonlinear system identification problem, and a number of functions supporting this application are described. A small demonstration example, giving an idea of how the toolbox is used in practice concludes the chapter. A reference guide which details the use of the different functions is given in Chapter 2.

It should be emphasized that this is not a text book on how to use neural networks for system identification. A good understanding of system identification (see for example Ljung, 1987) and of neural networks (see Hertz et al., 1991 or Haykin, 1993), are important requirements for understanding this tutorial. Naturally, the textbook of Nørgaard et al. (2000) is recommended literature as it specifically covers the use of neural networks in system identification. The manual could have been written in more textbook like fashion, but it is the author’s conviction that it is better to leave elementary issues out to motivate the reader to obtain the necessary insight into identification and neural network theory before using the toolbox. Understanding is the best way to avoid that working with neural networks becomes a “fiddlers paradise”!

1 The Multilayer Perceptron

The Multilayer Perceptron (or MLP) network is probably the most-often considered member of the neural network family. The main reason for this is its ability to model simple as well as very complex functional relationships. This has been proven through a large number of practical applications (see Demuth & Beale, 1998).



A fully connected two layer feedforward MLP-network with 3 inputs, 2 hidden units (also called “nodes” or “neurons”), and 2 outputs units.

The class of MLP-networks considered here is furthermore confined to those having only one hidden layer and only hyperbolic tangent and linear activation functions (*f,F*):

$$\hat{y}_i(\mathbf{w},\mathbf{W}) = F_i\left(\sum_{j=1}^{q} W_{ij} h_j(\mathbf{w}) + W_{i0}\right) = F_i\left(\sum_{j=0}^{q} W_{ij} f_j\left(\sum_{l=1}^{m} w_{jl} z_l + w_{j0}\right) + W_{i0}\right)$$

The weights (specified by the vector θ, or alternatively by the matrices $\boldsymbol{w}$ and $\boldsymbol{W}$) are the adjustable parameters of the network, and they are determined from a set of *examples* through the process called *training*. The examples, or the training data as they are usually called, are a set of inputs, $u(t)$, and corresponding desired outputs, $y(t)$.

Specify the training set by:

$$Z^N = \{[u(t), y(t)] \mid t = 1, \ldots, N\}$$

The objective of training is then to determine a mapping from the set of training data to the set of possible weights:

$$Z^N \rightarrow \hat{\theta}$$

so that the network will produce predictions $\hat{y}(t)$, which in some sense are “close” to the true outputs $y(t)$.

The *prediction error approach*, which is the strategy applied here, is based on the introduction of a measure of closeness in terms of a mean square error criterion:

$$V_N(\theta, Z^N) = \frac{1}{2N}\sum_{t=1}^{N}\left[y(t) - \hat{y}(t|\theta)\right]^T\left[y(t) - \hat{y}(t|\theta)\right]$$

The weights are then found as:

$$\hat{\theta} = \arg\min_{\theta} V_N(\theta, Z^N)$$

by some kind of iterative minimization scheme:

$$\theta^{(i+1)} = \theta^{(i)} + \mu^{(i)} f^{(i)}$$

$\theta^{(i)}$ specifies the *current iterate* (number 'i'), $f^{(i)}$ is the *search direction*, and $\mu^{(i)}$ the *step size*.

A large number of training algorithms exist, each of which is characterized by the way in which search direction and step size are selected. The toolbox provides the following algorithms:

General Network Training Algorithms	
batbp	Batch version of the Back-propagation algorithm.
igls	Iterated generalize Least Squares training of networks with multiple outputs.
incbp	Recursive (/incremental) version of Back-propagation.
marq	Basic Levenberg-Marquardt method.
marqlm	Memory-saving implementation of the Levenberg-Marquardt method.
rpe	Recursive prediction error method.

All functions require the following six arguments when called:

NetDef : A “string-matrix” defining the network architecture:

```
NetDef=['HHHHHH'
        'LH----'];
```

specifies that the network has 6 tanh hidden units, 1 linear, and 1 tanh output unit.

w1, w2 : Matrices containing initial weights (optional. If passed as [] they are selected at random).

PHI : Matrix containing the inputs.

Y: Matrix containing the desired outputs.

trparms: Data structure containing parameters associated with the training algorithm. If it is left out or passed as [], default parameters will be used. Use the function SETTRAIN if you do not want to use the default values. More information is found in the reference guide (Chapter 2).

For example, the function call

>> [W1,W2,crit_vector,iter]=batbp(NetDef,w1,w2,PHI,Y)

will train a network with the well-known *back-propagation algorithm*. This is a gradient descent method taking advantage of the special strucure of the neural network in the way the computations are ordered. By adding the argument *trparms*

>> [W1,W2,crit_vector,iter]=batbp(NetDef,w1,w2,PHI,Y,trparms)

it is possible to change the default values for the training algorithm. The different fields of *trparms* can be set with the function *settrain*:

>> trparms = settrain;   % Set trparms to default
>> trparms = settrain(trparms,'maxiter',1000,'critmin',1e-4, 'eta',0.02);

The first command initializes the variable *trparms* to a data structure with the default parameters. The second command sets the maximum number of iterations to 1000, sepcifies that the training should stop if the value of the criterion function V_N gets below 10^{-4}, and finally sets the step size to 0.02. The *batbp* function will return the trained weights, (*W1,W2*), a vector containing the value of V_N for each iteration, (*crit_vector*), and the number of iterations actually executed, (*iter*). The algorithm is currently the most popular method for training networks, and it is described in most text books on neural networks (see for example Hertz et al., 1991). This popularity is however not due to its convergence properties, but mainly to the simplicity with which it can be implemented.

A Levenberg-Marquardt method is the standard method for minimization of mean-square error criteria, due to its rapid convergence properties and robustness. A version of the method, decribed in Fletcher (1987), has been implemented in the function *marq:*

>> [W1,W2,crit_vector,iter,lambda]=marq(NetDef,w1,w2,PHI,Y,trparms)

The difference between this method and the one described in Marquardt (1963) is that the size of the elements of the diagonal matrix added to the Gauss-Newton Hessian is adjusted according to the size of the ratio between actual decrease and predicted decrease:

$$r^{(i)} = \frac{V_N(\theta^{(i)}, Z^N) - V_N(\theta^{(i)} + f^{(i)}, Z^N)}{V_N(\theta^{(i)}, Z^N) - L^{(i)}(\theta^{(i)} + f^{(i)})}$$

where

$$\begin{aligned} L(\theta^{(i)} + f) &= \sum_{t=1}^{N} \left(y(t) - \hat{y}(t|\theta^{(i)}) - f^T \left[\left. \frac{\partial \hat{y}(t|\theta)}{\partial \theta} \right|_{\theta=\hat{\theta}} \right] \right)^2 \\ &= V_N(\theta^{(i)}, Z^N) + f^T G(\theta^{(i)}) + \frac{1}{2} f^T R(\theta^{(i)}) f \end{aligned}$$

G here denotes the gradient of the criterion with respect to the weights and R is the so-called Gauss-Newton approximation to the Hessian.

The algorithm is as follows:

1) Select an initial parameter vector $\theta^{(0)}$ and an initial value $\lambda^{(0)}$

2) Determine the search direction from $\left[R(\theta^{(i)}) + \lambda^{(i)} I\right] f^{(i)} = -G(\theta^{(i)})$, I being a unit matrix.

3) $r^{(i)} > 0.75 \Rightarrow \lambda^{(i)} = \lambda^{(i)}/2$ (If predicted decrease is close to actual decrease let the search direction approach the Gauss-Newton search direction while increasing the step size.)

4) $r^{(i)} < 0.25 \Rightarrow \lambda^{(i)} = 2\lambda^{(i)}$ (If predicted decrease is far from the actual decrease let the search direction approach the gradient direction while decreasing the step size.)

5) If $V_N(\theta^{(i)} + f^{(i)}, Z^N) < V_N(\theta^{(i)}, Z^N)$ then accept $\theta^{(i+1)} = \theta^{(i)} + f^{(i)}$ as a new iterate and let $\lambda^{(i+1)} = \lambda^{(i)}$ and $i=i+1$.

6) If the stopping criterion is not satisfied go to 2)

The call is the same as for the back-propagation function, but the data structure *trparms* can now control the training in several new ways. Most importantly, *trparms* can be modified to obtain a minimization of criteria augmented with a *regularization* term:

$$W_N(\theta, Z^N) = \frac{1}{2N} \sum_{t=1}^{N} (y(t) - \hat{y}(t|\theta))^T (y(t) - \hat{y}(t|\theta)) + \frac{1}{2N} \theta^T D \theta$$

The matrix D is a diagonal matrix, which is commonly selected to $D = \alpha I$. For a discussion of regularization by simple weight decay; see for example Larsen & Hansen (1994), Sjöberg & Ljung (1992). D is a field in *trparms*, and its default value is 0. The command

```
>> trparms = settrain(trparms,'D',1e-5);
```

modifies D so that $D=10^{-5}\times I$. The command

```
>> trparms = settrain(trparms,'D',[1e-5 1e-4]);
```

has the effect that a weight decay of 10^{-4} is used for the input-to-hidden layer weights, while 10^{-5} is used for the hidden-to-output layer weights.

settrain can also control the initial value of λ. This is not a particularly critical parameter since it is adjusted adaptively and thus will only influence the initial convergence rate: if it is too large, the algorithm will take small steps and if it is too small, the algorithm will have to increase it until small enough steps are taken.

batbp and *marq* both belong to the class of so-called *batch methods* (meaning "all-at-once") and hence they will occupy a large quantity of memory. An alternative strategy is to repeat a *recursive* (or *incremental*) algorithm over the training data a number of times. Two functions are available in the toolbox: *incbp* and *rpe.*

The call
>> [W1,W2,PI_vector,iter]=rpe(NetDef,w1,w2,PHI,Y,trparms)

trains a network using a recursive Gauss-Newton like method as described in Ljung (1987). Different updates of the covariance matrix have been implemented. The *method* field in *trparms* selects which of the three updating methods will be used. The default is exponential forgetting (*'ff'*)

Exponential forgetting (*method='ff'*):

$$K(t) = P(t-1)\psi(t)\left(\lambda I + \psi^T(t)P(t-1)\psi(t)\right)^{-1}$$

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\overline{P}(t) = \left(P(t-1) - K(t)\psi^T(t)P(t-1)\right) \Big/ \lambda$$

Constant-trace (*method='ct'*):

$$K(t) = P(t-1)\psi(t)\left(1 + \psi^T(t)P(t-1)\psi(t)\right)^{-1}$$

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\overline{P}(t) = P(t-1) - K(t)\psi^T(t)P(t-1)$$

$$P(t) = \frac{\alpha_{\max} - \alpha_{\min}}{tr(\overline{P}(t))}\overline{P}(t) - \alpha_{\min} I$$

Exponential forgetting and Resetting Algorithm (*method='efra'*):

$$K(t) = \alpha P(t-1)\psi(t)\left(1 + \psi^T(t)P(t-1)\psi(t)\right)^{-1}$$

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$P(t) = \frac{1}{\lambda}P(t-1) - K(t)\psi^T(t)P(t-1) + \beta I - \delta P^2(t-1)$$

For neural network training, exponential forgetting is typically the method giving the fastest convergence. However, due to the large number of weights usually present in a neural network, care should be taken when choosing the forgetting factor since it is difficult to ensure that all directions of the weight space will be properly excited. If λ is too small, certain eigenvalues of the covariance matrix, P, will increase uncontrollably. The constant trace and the *EFRA* method are constructed so that they bound the eigenvalues from above as well as from below to prevent covariance blow-up without loss of the tracking ability. Since it would be very time consuming to compute the eigenvalues of P after each update, the functions does not do this. However, *if* problems with the

algorithms occur, one can check the size of the eigenvaules by adding the command *eig(P)* to the end of the *rpe*-function before termination.

For the forgetting factor method the user must specify an initial "covariance matrix" $P(0)$. The common choice for this is $P(0)=c\times I$, c being a "large" number, say *10* to 10^4. The two other methods initialize *P(0)* as a diagonal matrix with the largest allowable eigenvalue as its diagonal elements. When using the constant trace method, the user specifies the maximum and the minimum eigenvalue (α_{min}, α_{max}) directly. The *EFRA*-algorithm requires four different parameters to be selected. Salgado et al. (1988) give valuable supplementary information on this.

For multivariable nonlinear regression problems it is useful to consider a weighted criterion like the following:

$$V_N(\theta, Z^N) = \frac{1}{2N}\sum_{t=1}^{N}(y(t)-\hat{y}(t|\theta))^T \Lambda^{-1}(y(t)-\hat{y}(t|\theta))$$

As explained previously all the training algorithms have been implemented to minimize the unweighted criterion (i.e., Λ is always the identity matrix). To minimize the weighted criterion one will therefore have to scale the observations before training. Factorize $\Lambda^{-1} = \Sigma^T\Sigma$ and scale the observations as $\bar{y}(t) = \Sigma y(t)$. If the network now is trained to minimize

$$V_N(\theta, Z^N) = \frac{1}{2N}\sum_{t=1}^{N}(\bar{y}(t)-\hat{\bar{y}}(t|\theta))^T (\bar{y}(t)-\hat{\bar{y}}(t|\theta))$$

the network output ($\hat{\bar{y}}(t)$) must subsequently be rescaled by the operation $\hat{y}(t) = \Sigma^{-1}\hat{\bar{y}}(t)$. If the network has linear output units the scaling can be build into the hidden-to-output layer matrix, *W*2, directly: $W = \Sigma^{-1}\overline{W}$.

Since the weighting matrix is easily factorized by using the MATLAB command *sqrtm* it is straightforward to train networks with multiple outputs:

```
>> [W1,W2,crit_vector,iter,lambda]=marq(NetDef,w1,w2,PHI,sqrtm(inv(Gamma))*Y,trparms);
>> W2=sqrtm(Gamma)*W2;
```

If the noise on the observations is white and Gassian distributed and the network architecture is *complete*, i.e., the architecture is large enough to describe the system underlying the data, the Maximum Likelihood estimate of the weights is obtained if Λ is selected as the noise covariance matrix. The covariance matrix is of course unknown in most cases and often it is therefore estimated. In the function *igls* an iterative procedure for network training and estimation of the covariance matrix has been implemented. The procedure is called *Iterative Generalized Least Squares*

```
>> [W1,W2]=marq(NetDef,W1,W2,PHI,Y,trparms);
>> [W1,W2,Gamma,lambda]=igls(NetDef,W1,W2,trparms,Gamma0,PHI,Y);
```

The function outputs the scaled weights and thus the network output (or the weights if the output units are linear) must be rescaled afterwards.

To summarize advantages and disadvantages of each algorithm, the following table grades the most important features on a scale from -- (worst) to ++ (best):

	Execution time	Robustness	Call	Memory
batbp	--	+	--	-
incbp	--	+	--	++
marq	++	++	++	--
marqlm	+	++	++	0
rpe	0	-	--	++

Apart from the functions mentioned above, the toolbox offers a number of functions for data scaling, for validation of trained networks and for detemination of optimal network architectures. These functions will be described in the following section along with their system identification counterparts. The section describes the real powerful portion of the toolbox, and it is essentially this portion that seperates this toolbox from most other neural network tools currently available.

2 System Identification

The procedure which must be executed when attempting to identify a dynamical system consists of four basic steps:

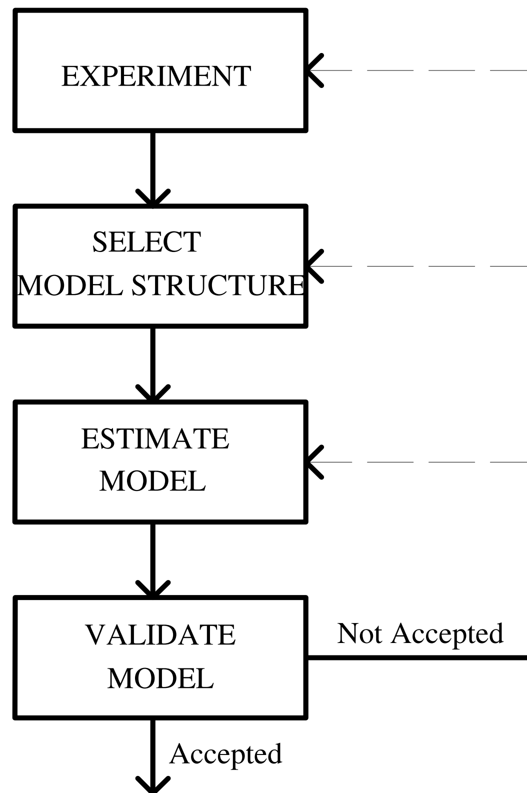


Figure 2.1 The system identification procedure.

Similarly to the System Identification Toolbox provided by The MathWorks, Inc., the experimental phase is not something which will be dealt with here. It is assumed that experimental data describing the underlying system in its entire operating region has been obtained beforehand with a proper choice of sampling frequency:

$$Z^N = \{[u(t), y(t)] | t = 1, \ldots, N\}$$

$\{u(t)\}$ is no longer just a set of inputs but it is now a *signal*, the *control signal*. Likewise $\{y(t)\}$ now represents the measured output signal. 't' specifies sampling instant number t. If the system under consideration has more than one input and/or output, $u(t)$ and $y(t)$ are simply vectors.

2.1 Select Model Structure

Assuming that a data set has been acquired, the next step is to select a model structure. Unfortunately, this issue is much more difficult in the nonlinear case than in the linear case. Not only is it necessary to choose a set of regressors but also a network architecture is required. The approach used here is described in Nørgaard et al. (2000). The idea is to select the regressors based

on inspiration from linear system identification and then determine the best possible network architecture with the given regressors as inputs.

The toolbox provides six different model structures, five of which are listed below. The sixth will not be discussed until in the next section since it has a form which is not motivated by equivalence to a linear model structure.

$\varphi(t)$ is a vector containing the regressors, θ is a vector containg the weights and g is the function realized by the neural network.

NNARX
Regression vector:
$$\varphi(t) = [y(t-1) \quad \dots \quad y(t-n_a) \quad u(t-n_k) \quad \dots \quad u(t-n_b-n_k+1)]^T$$

Predictor:
$$\hat{y}(t|\theta) = \hat{y}(t|t-1,\theta) = g(\varphi(t),\theta)$$

NNOE
Regression vector:
$$\varphi(t) = [\hat{y}(t-1|\theta) \quad \dots \quad \hat{y}(t-n_a|\theta) \quad u(t-n_k) \quad \dots \quad u(t-n_b-n_k+1)]^T$$

Predictor:
$$\hat{y}(t|\theta) = g(\varphi(t),\theta)$$

NNARMAX1
Regression vector:
$$\begin{aligned}\varphi(t) &= [y(t-1) \quad \dots \quad y(t-n_a) \quad u(t-n_k) \quad \dots \quad u(t-n_b-n_k+1) \quad \varepsilon(t-1) \quad \dots \quad \varepsilon(t-n_c)]^T \\ &= [\varphi_1^T(t) \quad \varepsilon(t-1) \quad \dots \quad \varepsilon(t-n_c)]^T\end{aligned}$$
where $\varepsilon(t)$ is the prediction error $\varepsilon(t) = y(t) - \hat{y}(t|\theta)$

Predictor:
$$\hat{y}(t|\theta) = g(\varphi_1(t),\theta) + (C(q^{-1})-1)\varepsilon(t)$$

where C is a polynomial in the backward shift operator $C(q^{-1}) = 1 + c_1 q^{-1} + \dots + c_{nc} q^{-nc}$

<u>NNARMAX2</u>
Regression vector:

$$\varphi(t) = [y(t-1) \quad \dots \quad y(t-n_a) \quad u(t-n_k) \quad \dots \quad u(t-n_b-n_k+1) \quad \varepsilon(t-1) \quad \dots \quad \varepsilon(t-n_c)]^T$$

Predictor:

$$\hat{y}(t|\theta) = g(\varphi(t),\theta)$$

<u>NNSSIF (state space innovations form)</u>
Regression vector:

$$\varphi(t) = [\hat{x}^T(t|\theta) \quad u^T(t) \quad \varepsilon^T(t|\theta)]^T$$

Predictor:

$$\hat{x}(t+1|\theta) = g(\varphi(t),\theta)$$

$$\hat{y}(t|\theta) = C(\theta)\hat{x}(t|\theta)$$

To obtain a so-called pseudo-observable form, a set of psudo-observability indices must be specified. For more detailed information see Chapter 4, Appendix A in Ljung (1987).

Only the NNARX model has a predictor without feedback. The remaining model types all have feedback through the choice of regressors, which in the neural network terminology means that the networks become *recurrent*: future network inputs will depend on present and past network outputs. This might lead to instability in certain regimes of the networks's operating range, and it can be very difficult to determine whether or not the predictor is stable. The NNARMAX1 structure was constructed to avoid this by using a linear MA-filter for filtering past residuals. The basic rule of thumb is, however, to use a NNARX type model whenever possible.

When a particular model structure has been selected, the next choice, which has to be made, is the number of past signals used as regressors, i.e., the *model order*. If no or only very little noise is present (in which case the *nnarx* function is *always* preferred), the toolbox provides a method for determining the so-called *lag space*:

>> *OrderIndexMat = lipschit(U,Y,nb,na)*

(See He & Asada, 1993). The function is not always equally succesful but sometimes reasonable performance is observed. However, it is **always** better to have enough physical insight into the system to be modelled to choose the model order properly.

2.2 Estimate Model

The toolbox contains the following functions for generating models from a specified model structure:

Nonlinear System Identification	
nnarmax1	Identify a neural network ARMAX (or ARMA) model (linear noise filter).
nnarmax2	Identify a neural network ARMAX (or ARMA) model.
nnarx	Identify a neural network ARX (or AR) model.
nnarxm	Identify a multi-output neural network ARX (or AR) model.
nnigls	Iterated generalized least squares training of multi-output NNARX models
nniol	Identify a neural network model suited for I-O linearization type control.
nnoe	Identify a neural network Output Error model.
nnssif	Identify a neural network state space innovations form model.
nnrarmx1	Recursive counterpart to NNARMAX1.
nnrarmx2	Recursive counterpart to NNARMAX2.
nnrarx	Recursive counterpart to NNARX.

The functions use either the Levenberg-Marquardt method or the recursive prediction error method and consequently they must be passed many of the same parameters as the functions *marq* or *rpe* when called:

>> [W1,W2, NSSEvec,iter,lambda]=nnarx(NetDef,NN,W1,W2,trparms,Y,U)

In order for the function to be able to determine the structure of the regression vector, the additional argument *NN* has to be passed (except for *nnssif*). In, for example, the NNARX case, $NN=[n_a\ n_b\ n_k]$. Except for *nnarxm* and *nnssif* none of the functions are able to handle multi-output systems. If one wishes to build a NNARMAX model of a system with more than one output it is thus necessary to build a seperate model for each output. However, all functions do handle multi-input systems, in which case n_b *and* n_k are specified as row vectors containing the proper specifications for each input.

The functions for building models based on recurrent networks can make use of the parameter *skip* in the data structure *trparms*. This parameter is used for reducing transient effects corrupting the training. These effects are due to the generally unknown initial conditions. The *skip* field in *trparms* is set as follows:

```
>> trparms=settrain;                          % Select default values
>> trparms=settrain(trparms,'skip',25);  % Set the skip field to 25
```

The default value of *skip* is 0. Selecting a value of, e.g., 25, means that the first 25 samples are not used for updating the weights.

The function *nniol* differs from the above mentioned functions in that it does not generate a model motivated by conventional linear system identification. The idea is to describe the system by:

$$\hat{y}(t|\theta) = f(y(t-1),\ldots, y(t-n_a), u(t-2),\ldots,u(t-n_b),\theta_f) + g(y(t-1),\ldots, y(t-n_a), u(t-2),\ldots,u(t-n_b),\theta_g)u(t-1)$$

where *f* and *g* are two seperate networks. This type of model is particularly interesting for control by so-called input-output linearization. Unless the intention is to use the model for that specific purpose, it is recommended to avoid this function since it includes two networks instead of just one.

2.3 Validate Model

When a network has been trained, the next step according the procedure, is to evaluate it.

Evaluation of Trained Networks	
fpe	Final Prediction Error estimate of generalization error for feed-forward nets.
ifvalid	Validation of models generated by NNSSIF.
ioleval	Validation of models generated by NNIOL.
loo	Leave-One-Out estimate of generalization error for feed-forward networks.
kpredict	*k*-step ahead prediction of network output.
nneval	Validation of feed-forward networks (trained by marq, batbp, incbp, or rpe).
nnfpe	FPE-estimate for I-O models of dynamic systems.
nnloo	Leave-One-Out estimate of the generalization error for NNARX models
nnsimul	Simulate identifyed model of dynamic system from sequence of controls.
nnvalid	Validation of I-O models of dynamic systems.
xcorrel	High-order cross-correlation functions.

The most common method of validation is to investigate the residuals (prediction errors) by cross-validation on a *test set*. The functions *nnvalid* and *ifvalid* perform a number of such tests, including autocorrelation function of the residuals and cross-correlation function between controls and residuals. These functions are displayed along with a histogram showing the distribution of the residuals. Moreover, a linear model is extracted from the network at each sampling instant by a so-called local instantanous linearization technique (see Nørgaard et al., 2000). The function *nnvalid*, which handles the validation for most of the model types, is called as follows if *nnarx* was used for generating the model:

```
>> [Yhat,NSSE]=nnvalid('nnarx',NetDef,NN,W1,W2,y,u)
```

u and *y* specify the test set control input and output signals. *Yhat* is the one-step-ahead predictions produced by the network while NSSE is the criterion evaluated on the test set (this is also called the

test error). With the function *xcorrel* it is possible to investigate a number of high-order cross-correlation functions (see Billings et al., 1992)

>> *xcorrel('nnarx',NetDef,NN,W1,W2,y,u)*

The test error is an important quantity since it can be viewed as an estimate of the generalization error. This should not be too large compared to training error, in which case one must suspect that the network is over-fitting the training data. If a test set is not available the average generalization error:

$$J(\mathbf{M}) \equiv E\left\{\overline{V}(\hat{\theta})\right\}, \overline{V}(\hat{\theta}) = \lim_{N\to\infty} E\left\{V_N(\hat{\theta}, N)\right\}$$

can be estimated from the training set alone by Akaike's final prediction error (FPE) estimate. Although a test set is available, the FPE estimate might still offer some valuable insights. For the basic unregularized criterion the estimate reads (see Ljung, 1987):

$$\hat{J}_{FPE}(\mathbf{M}) = \frac{N+d}{N-d} V_N(\hat{\theta}, Z^N)$$

d denoting the total number of weights in the network. When the regularized criterion is used, the expression gets somewhat more complex (see Larsen & Hansen, 1994):

$$\hat{J}_{FPE}(\mathbf{M}) = \frac{N+\gamma_1}{\left(N+\gamma_1-2\gamma_2\right)} V_N(\hat{\theta}, Z^N)$$

where

$$\gamma_1 = tr\left[R(\hat{\theta})(R(\hat{\theta})+\tfrac{1}{N}D)^{-1}R(\hat{\theta})(R(\hat{\theta})+\tfrac{1}{N}D)^{-1}\right]$$

and

$$\gamma_2 = tr\left[R(\hat{\theta})(R(\hat{\theta})+\tfrac{1}{N}D)^{-1}\right]$$

$\gamma_1 \approx \gamma_2$ specifies the so-called *effective* number of parameters in the network. It seems as if the estimate often becomes more reliable in the regularized case, which probably has to do with the regularization having a smoothing effect on the criterion. A more smooth criterion function means that the assumptions on which the FPE was derived are more likely to be valid. The function *nnfpe* computes the FPE estimate for all the input-output models.

>> *[FPE,deff]=nnfpe('nnarx',NetDef, W1,W2,u,y,NN,trparms);*

For models trained with *nnarx* the leave-one-out cross-validation scheme provides an estimate of the generalization error which in general will be more accurate than the FPE estimate. The leave-one-out estimate is defined by

$$\hat{J}_{LOO}(\mathbf{M}) = \frac{1}{2N}\sum_{t=1}^{N}\left(y(t)-\hat{y}(t\,|\,\hat{\theta}_t)\right)^2$$

where

$$\hat{\theta}_t = \underset{\theta}{\arg\min} W_{N-1}\left(\theta, Z^N \setminus \{u(t), y(t)\}\right)$$

The function is called as follows

>> *Eloo =nnloo(NetDef,NN,W1,W2,trparms,u,y)*

If the *maxiter* field in *trparms*, which specifies the maximum number of iterations, is set to 0, an approximation called *linear unlearning* is employed to avoid having to to train *N* networks. The idea is that $\hat{\theta}_t$ is approximated from a series expansion around $\hat{\theta}$. This leads to the expression

$$\hat{J}_{LOO}(\mathbf{M}) = \frac{1}{2N}\sum_{t=1}^{N}\left(y(t) - \hat{y}(t|\hat{\theta})\right)^2 \frac{N + \psi^T(t,\hat{\theta})H^{-1}(\hat{\theta})\psi(t,\hat{\theta})}{N - \psi^T(t,\hat{\theta})H^{-1}(\hat{\theta})\psi(t,\hat{\theta})}$$

where

$$\psi(t,\hat{\theta}) = \left.\frac{\partial \hat{y}(t|\theta)}{\partial \theta}\right|_{\theta=\hat{\theta}}$$

See Hansen & Larsen (1995) for a derivation. If the network is trained to minimize the unregularized criterion, the inverse Hessian is approximated by the recursive approximation (also used by the function *rpe)*:

$$P(t) = \left(P(t-1) - P(t-1)\psi(t)\left(I + \psi^T(t)P(t-1)\psi(t)\right)^{-1}\psi^T(t)P(t-1)\right)$$

Visual inspection of the plot comparing predictions to actual measurements is probably the most important validation tool. However, one has to be careful with one-step ahead predictions. Often they may look very accurate, even though the estimated model is quite far from what it should be. This is particularly true when the system is sampled rapidly compared to its (fastest) dynamics. An additional check which Ljung (1987) has recommended, is to perform a pure simulation of the model. This can be done using the function *nnsimul*.

>> *Ysim=nnsimul('nnarx',NetDef,NN,W1,W2,y,u)*

One can also investigate the *k*-step ahead predictions with the function *kpredict*

>> *Ypred = kpredict('nnarx',NetDef,NN,k,W1,W2,y,u);*

For feedforward networks there exist a few functions performing more or less the same tests as the functions discussed above for models of dynamic systems. The function *nneval* is used for investigating the residuals while the functions *fpe* and *loo* computes the FPE-estimate and LOO estimate, respectively.

2.4 Going Backwards in the Procedure
The figure illustrating the identification procedure shows some ”feedback” paths from validation to the previous blocks. The path from validation to training is due to the criterion having local minima. Since it is very likely that one ends up in a “bad” local minimum, the network should be trained a couple of times, starting from different initial weights. Regularization has a tremendous smoothing effect on the criterion, and several of the local minima are hence often removed by this. Local minima do, however, remain one of the major problems for nonlinear regressions, and there is no simple way of avoiding them.

Another path in figure 2.1 leads back to the model structure selection block. Because of the way the model structure selection problem has been divided into two seperate subproblems, this can mean two things, namely: “try another regressor structure” or “try another network architecture.” While the regressor structure typically has to be chosen on a trial-and-error basis, it is to some extent possible to automate the network architecture selection. For this purpose the toolbox provides the functions listed in the table below:

Determination of Optimal Network Architecture	
netstruc	Extract weight matrices from matrix of parameter vectors.
nnprune	Prune models of dynamic systems with Optimal Brain Surgeon (OBS).
obdprune	Prune feed-forward networks with Optimal Brain Damage (OBD).
obsprune	Prune feed-forward networks with Optimal Brain Surgeon (OBS).

The so-called *Optimal Brain Surgeon (OBS)* is the most important strategy, and it is consequently the only method which has been implemented for models of dynamic systems. The method was originally proposed by Hassibi & Stork (1993), but Hansen & Pedersen (1994) have later derived a modification of the method so that it can handle networks trained according to the regularized criterion.

Hansen & Pedersen (1994) define a saliency as the estimated increase of the *unregularized* criterion when a weight is eliminated. Because of this definition, a new expression for the saliences is obtained. The saliency for weight ‘*j*’ is defined by:

$$\zeta_j = \frac{\lambda}{N} e_j^T H^{-1}(\theta^*) D\theta^* + \tfrac{1}{2}\lambda_j^2 e_j^T H^{-1}(\theta^*) R(\theta^*) H^{-1}(\theta^*) e_j$$

where θ^* specifies the minimum and $H(\theta^*)$ the Gauss-Newton Hessian of the regularized criterion:

$$H(\theta^*) = R(\theta^*) + \tfrac{1}{N} D$$

e_j is the jth unit vector and λ_j is the Lagrange multiplier, which is determined by:

$$\lambda_j = \frac{e_j^T \theta^*}{e_j^T H^{-1}(\theta^*) e_j} = \frac{\theta_j^*}{H_{j,j}^{-1}(\theta^*)}$$

The constrained minimum (the minimum when weight '*j*' is 0) is then found from

$$\delta\theta = \theta^* - \theta = -\lambda_j H^{-1}(\theta^*) e_j$$

Notice that for the unregularized criterion, where $R(\theta^*) = H(\theta^*)$, the above algorithm will degenerate to scheme of Hassibi & Stork (1993).

The problem with the basic OBS-scheme is that it does not take into account that it should not be possible to have networks where a hidden unit has lost all the weights leading to it, while there still are weights connecting it to the output layer, or vice versa. For this reason the implementation in this toolbox goes one step further. In the beginning the saliences are calculated and the weights are pruned as described above. But when a situation occurs where a unit has only one weight leading to it or one weight leading from it, the saliency for removing the entire unit is calculated instead. This gives rise to some minor extensions to the algorithm (the saliency for the unit is calculated by setting all weights connected to the unit to 0. Although this is not the optimal strategy it appears to be working alright in practice. See also (Pedersen et al., 1995)):

Define J as the set of indices to the weights leading to and from the unit in question. Furthermore, let E_J be a matrix of unit vectors corresponding to each element of the set J. In order to calculate the saliency for the entire unit, the above expressions are then be modified to:

$$\zeta_J = \lambda_J^T E_J^T H^{-1}(\theta^*) \tfrac{1}{N} D\theta^* + \tfrac{1}{2} \lambda_J^T E_J^T H^{-1}(\theta^*) R(\theta^*) H^{-1}(\theta^*) E_J \lambda_J$$

$$\lambda_J = \left[E_J^T H^{-1}(\theta^*) E_J \right]^{-1} E_J^T \theta^*$$

$$\delta\theta = \theta^* - \theta = -H^{-1}(\theta^*) E_J \lambda_J$$

When a weight (or unit) has been removed, it is necessary to obtain the new inverse Hessian before proceeding to eliminate new weights. If the network has been retrained it is of course necessary to construct and invert the Hessian once more. If, however, the network is not retrained, a simple trick from inversion of partitioned matrices can be used for approximating the inverse Hessian of the reduced network (Pedersen et al., 1995).

Assume that the pruned weight(s) is(are) located at the end of the parameter vector, θ. The Hessian is then partioned as follows:

$$H = \begin{bmatrix} \tilde{H} & h_J \\ h_J^T & h_{JJ} \end{bmatrix}$$

$\tilde{H}$ is the 'new' Hessian, which is to be inverted. Partitioning the inverse Hessian in the same way yields

$$H^{-1} = \begin{bmatrix} \tilde{P} & p_{JJ} \\ p_J^T & p_{JJ} \end{bmatrix}$$

The new inverse Hessian is then determined as the *Schur Complement* of $\tilde{P}$:

$$\tilde{H}^{-1} = \tilde{P} - p_J p_{JJ}^{-1} p_J^T$$

It is difficult to decide how often the network should be retrained during the pruning session. Svarer et al. (1993) suggest a scheme for retraining each time 5% of the weights have been eliminated. The safest, but also the most time consuming strategy, is of course to retrain the network each time a weight has been eliminated.

If a model has been generated by *nnarx*, the OBS function is called as:

```
>> [thd,tr_error,FPE,te_error,d_eff,pvec] = ...
            nnprune('nnarx',NetDef,W1,W2,U,Y,NN,trparms,prparms,U2,Y2);
```

prparms specifies how often the network is retrained. If one wants to run a maximum of 50 iterations each time 5% of the weights has been eliminated, set *prparms* to *prparms=[50 5]*.

thd is a matrix containing the parameter vectors θ after each weight elimination. The last column of *thd* contains the weights for the initial network. The next-to-last column contains the weights for the network appearing after eliminating one weight, and so forth. To extract the weight matrices from *thd*, the function *netstuc* has been implemented. If, for example, the network containing 25 weights is the optimal, the weights are retrieved by:

```
>> [W1,W2] = netstruc(NetDef,thd,25);
```

2.5 Time-series Analysis

Almost all functions for supporting the system identification process have been written so that they can handle time-series as well. Check the reference in Chapter 2 or the online help facility on how this is can be accomplished.

2.6 Important Issues

- The criterion function will typically have a number of local minima and there is no way to determine whether a given minimum is global or not. Although regularization has a smoothing effect on the criterion and often will remove a number of the minima, it is still recommended to always train the network a couple of times assuming different initial weights.

- It has been shown that by stopping the training process before a minimum has been reached, an effect similar to that of regularization is obtained (Sjöberg & Ljung, 1992). If the network architecture is selected larger than necessary, it is therefore advantagous to use "early stopping." One may find this strategy appealing, when the alternative is a tedious process of finding the optimal weight decay and/or prune the network. However, one should be aware that when using early stopping, the FPE and LOO estimates of the generalization error will not work reliably since the expressions were derived under the assumption that the network is trained to a minimum. An important validation tool is thereby lost. Also the pruning algorithms assume that the network has been trained to a minimum. It this is not the case, it is definitely not recommended to prune the network either. In this context it is also important to note that in

practice it is difficult to reach a minimum with a recursive algorithm. It is thus recommended to use batch algorithms whenever possible.

- Not all m-functions have been written in equally vectorized code. For example, it is not possible to implement recursive training algorithms and training algorithms for model structures based on recurrent networks in quite as "MATLAB-efficient" code as for batch algorithms to train plain feedforward networks. Whereas no significant increase in speed is obtained by porting the functions *batbp*, *marq* (,and *nnarx*) to C, a serious performance improvement results if the recursive algorithms and the algorithms for training models based on recurrent networks are written in C. So far, four CMEX-programs have been implemented: marq.c, nnarmax2.c, nnoe.c, nnssif.c. If a C-compiler is available on the system these can be compiled with the command *makemex*.

- Many functions require a lot of memory since the code has been optimized with respect to execution speed rather than memory usage.

- One should be careful not to select the forgetting factor too small when training networks by the recursive pediction error method. A small forgetting factor results in very rapid covariance blow-up.

- When the predictor contains feedback, stability problems may occur during training as well as afterwards, when the network is used. One should thus be very careful with models based on recurrent networks.

- Initialization of recurrent networks. If *nnoe*, *nnarmax1* and *nnarmax2* are called with empty matrices, [], for the weights, the functions try to initialize the weights to make a stable initial model. Although it is not always a problem that the initial model is unstable, the instability will often result in severe numerical problems. When pruning recurrent networks unstable networks often occur during the process.

- It might be that there are certain features in the training data which one is not interested in capturing with the model. This could be regular outliers, but it could also be some undesired physical effects. Consequently one is sometimes interested in removing certain points and time-intervals from the training set. Unfortunately this is not possible unless the predictor has no feedback (as in NNARX models), since removal of data points otherwise will give rise to transients.

3 Example

A simple demonstration example giving a walk through of several of the toolbox functions is presented below. The subject of the example investigation is a set of data obtained by simulation of an open-loop stable, nonlinear, continuous system. The data set can be found along with the toolbox functions in a file called *spmdata.mat*. Since the data has been generated by simulation, it is known that it is possible to describe it by a nonlinear output error model. In general, information like this is clearly not available, and hence it will in practice be necessary to test a number of different model structures before picking the best one. The working procedure is typically the same that outlined below.

Many of the functions produce several plots, but some of these are sacrificed in the example in order to give the reader a clear overview. If the reader wants to run the example himself, he should be aware that the results need not be exactly as shown since they depends heavily on the initial weights chosen by random in the function *nnoe*.

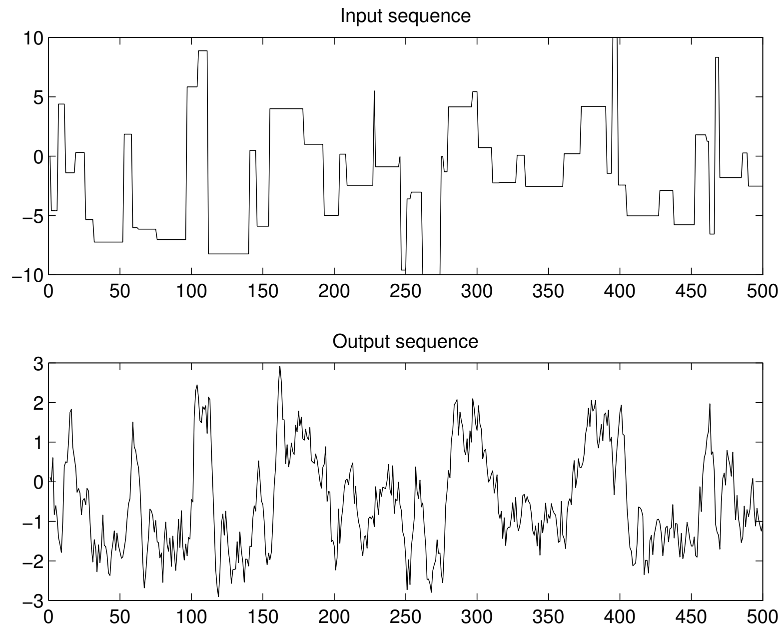
Load data obtained by experiment
The data is found in the file *spmdata.mat*, which is found along with all the toolbox functions. Load the file and use the *whos* command to view the contents:

```
>> load spmdata
>> whos
        Name     Size     Elements   Bytes   Density   Complex

          u1     1 by 500     500     4000     Full      No
          u2     1 by 500     500     4000     Full      No
          y1     1 by 500     500     4000     Full      No
          y2     1 by 500     500     4000     Full      No

Grand total is 2000 elements using 16000 bytes
```

u1 and y1 will be used for training, while u2 and y2 will be used for validation purposes only.

Display the training data in a MATLAB figure by typing the following sequence of commands:

```
>> subplot(211), plot(u1)
>> title('Input sequence')
>> subplot(212), plot(y1)
>> title('Output sequence')
>> subplot(111)
```



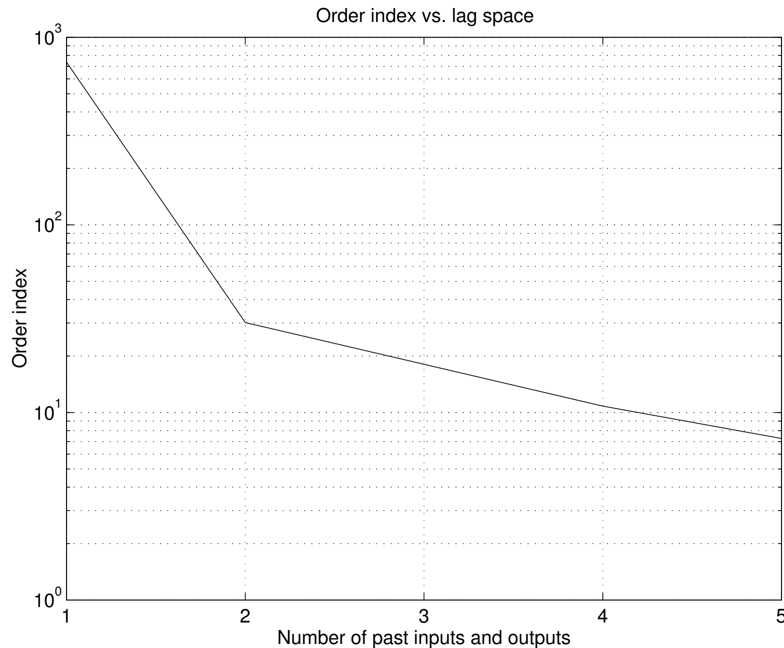
First the training set is scaled to zero mean and variance one and then the test set is scaled with the same constants:

```
>> [u1s,uscales] = dscale(u1);
>> [y1s,yscales] = dscale(y1);
>> u2s = dscale(u2,uscales);
>> y2s = dscale(y2,yscales);
```

Determine regressors

The function *lipschit* is employed to determine the order of the system. For example, the indices for system orders from 1 to 5 are investigated (be aware that it takes some time!):

```
>> OrderIndices = lipschit(u1s,y1s,1:5,1:5);
```



It is difficult to conclude anything certain from this plot, which probably has to do with the noise corrupting the measurements. It is, however, not unreasonable to assume that the system can be modeled by a second order model since the slope of the curve is decreases for model orders >= 2.

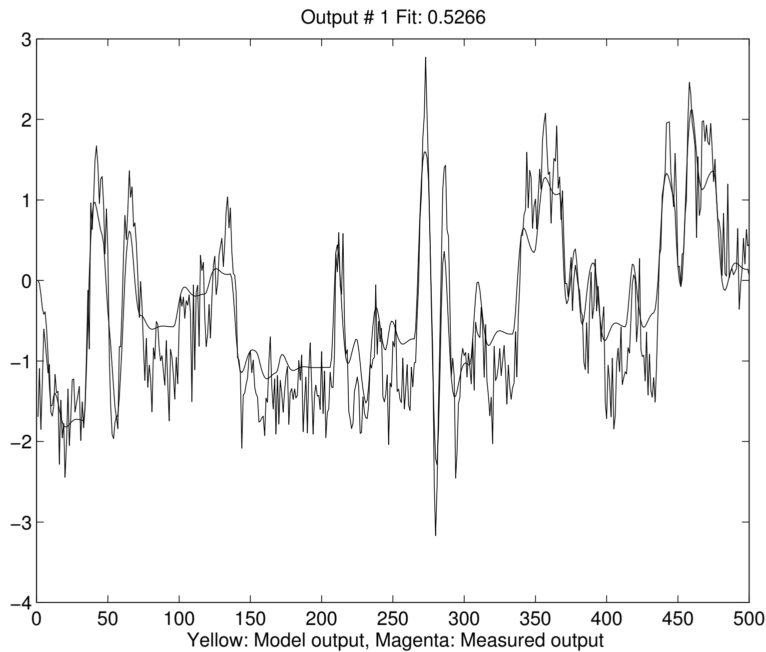
Fit a Linear Model

The golden rule in identification (and in most other matters) is to try simple things first. If a linear model does a decent job, one should not bother wasting time on fancy neural network based model structures. To identify a linear OE-model, the System Identification Toolbox from The MathWorks, Inc. (Ljung, 1995) is employed:

```
>> th = oe([y1' u1'],[2 2 1]);
>> present(th);
This matrix was created by the command OE     on 2/1 2000 at 20:2
Loss fcn: 0.2814   Akaike`s FPE: 0.28594 Sampling interval 1
The polynomial coefficients and their standard deviations are

B =

         0    0.0153    0.0275
         0    0.0055    0.0076


F =

    1.0000   -1.5135    0.6811
         0    0.0286    0.0209
```

The function *compare* is called to compare measurements and predictions:

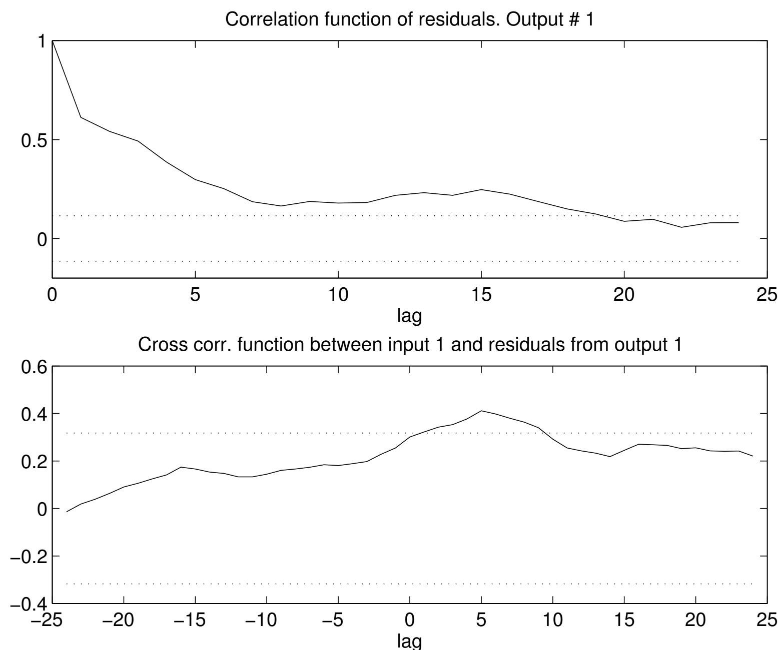
>> figure(1), compare([y2' u2'],th,1);



The prediction is the smooth curve while the more noisy one represents the measurements.

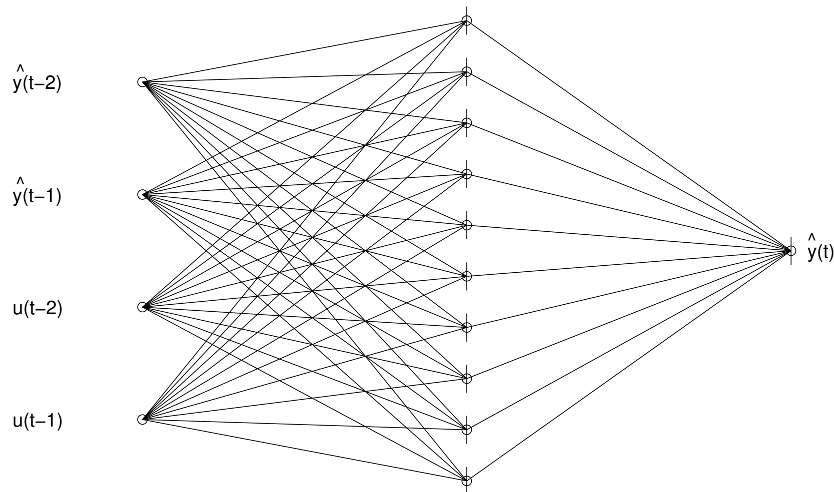
One of the most common ways of validating an estimated linear model, is to display the auto correlation function of the residuals and the cross correlation function between input and residuals. This can be done with the function *resid*:

>> figure(2), resid([y2' u2'],th);



From the comparison plot as well as from the correlation functions it clearly appears that the linear model has severe problems, especially for large magnitudes. It is thus concluded that this is due to the underlying system being nonlinear.

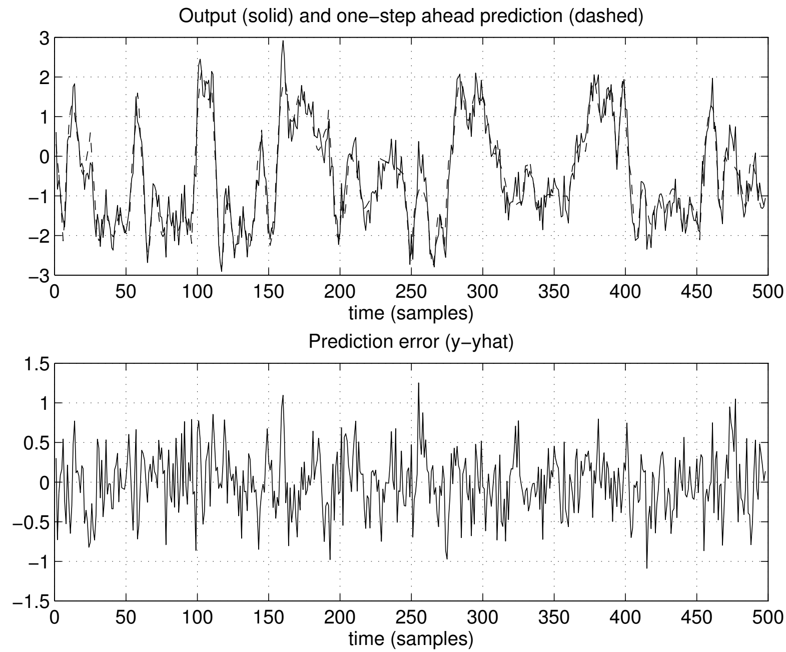
Choose a Nonlinear Model Structure
for comparison a NNOE model structure is attempted. Initially a fully connected network architecture with 10 hidden hyperbolic tangent units is selected:



Define the model structure: second order model and 10 tanh units in hidden layer:
>> *NetDef = ['HHHHHHHHHH'; 'L---------'];*
>> *NN = [2 2 1];*

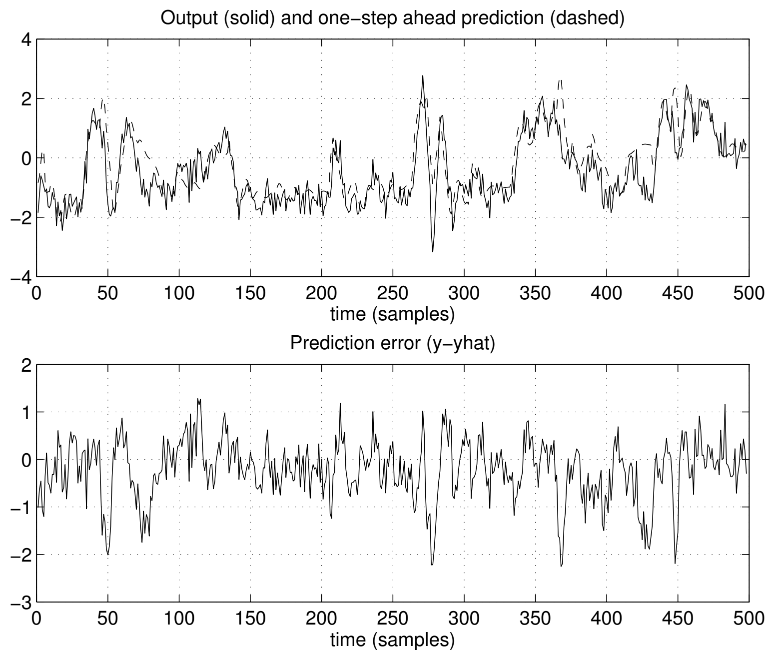
The network now has to be trained. Let the function initialize the weights, try max. 300 iterations and use a small weight decay. Select the field *skip* to 10 to reduce the influence of the unknown initial conditions:
>> *trparms = settrain;*
>> *trparms = settrain(trparms,'maxiter',300,'D',1e-3,'skip',10);*
>> *[W1,W2,NSSEvec]=nnoe(NetDef,NN,[],[],trparms,y1s,u1s);*

Validation of the trained network
The function *nnvalid* is called to validate the generated NNOE-model. First the validation is performed on the training set (only the comparison plot is shown here):
>> *[w1,w2] = wrescale('nnoe',W1,W2,uscales,yscales,NN);*   *% Rescale the weights*
>> *[yhat,NSSE] = nnvalid('nnoe',NetDef,NN,w1,w2,y1,u1);*

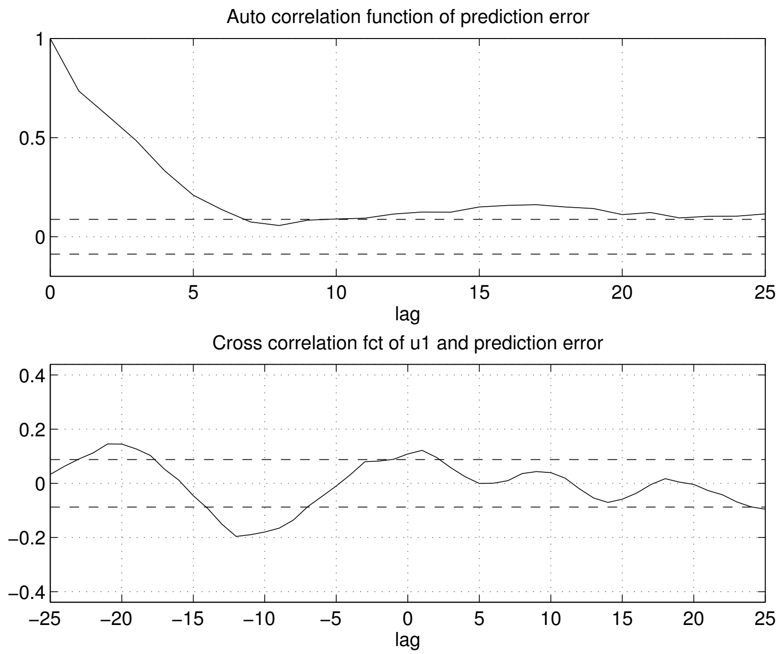


This looks quite satisfactory, and it is definitely better than the linear model. However, it is never a problem to fit the training data accurately. Redoing the validation on the test set clearly gives a less flattering result:

>> [yhat,NSSE] = nnvalid('nnoe',NetDef,NN,w1,w2,y2,u2);



>> figure(2)

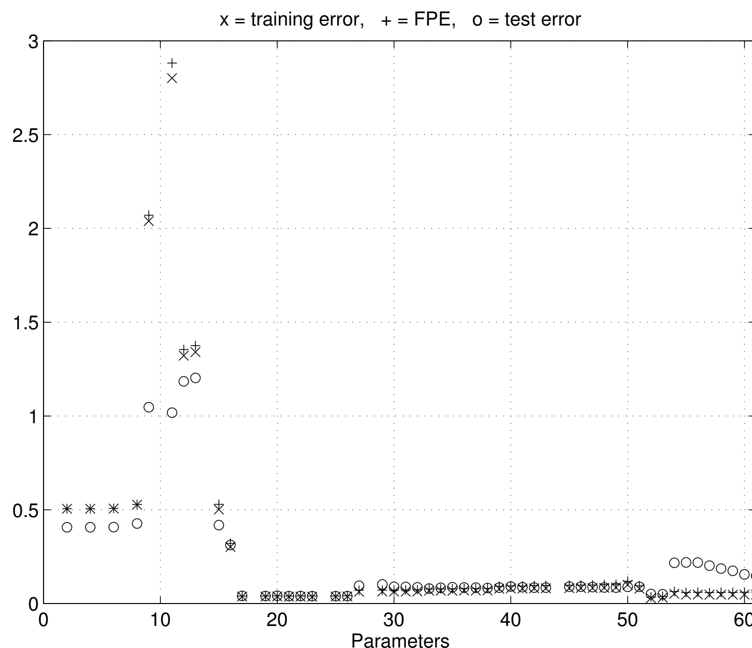


Both plots more than just indicate that not everything is as it perherps could be. By comparing the plots for training and test set, it is quite obvious that the network is overfitting the data. It is concluded therefore that the model structure selected contains too many weights.

Improve Performance by Pruning
In order to remove the superfluous weights from the network, the function *nnprune* is called. This function determines the optimal network architecture by pruning the network according to the Optimal Brain Surgeon strategy.

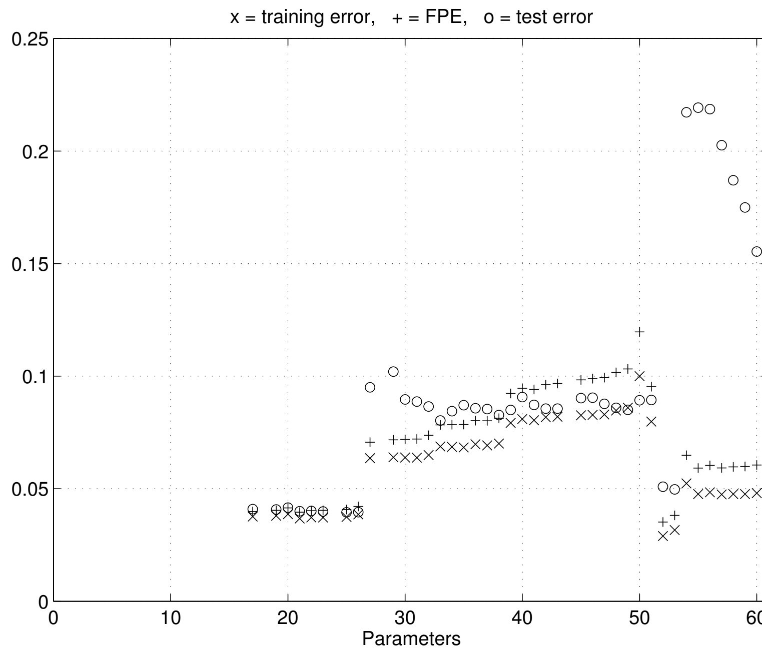
Run the function so that the network is retrained after each weight elimination (this is the slowest but safest strategy). Do a maximum of 50 iterations when retraining the network. Pass the test data to the function so that the test error later can be used for pointing out the optimal network. The test error is the most reliable estimate of the generalization error and it is thus reasonable to select the network with the smallest test error as final one. The pruning function is called by using the commands:

```
>> prparms = [50 0];
>> [thd,trv,fpev,tev,deff,pv] = ...
        nnprune('nnoe',NetDef,W1,W2,u1s,y1s,NN,trparms,prparms,u2s,y2s);
```



The above plot, which is produced by the *nnprune* function, should be read from right to left. The plot displays training error, test error, and FPE estimate of the generalization error of each of the intermidiate networks. To determine where the test error reaches its minimum, it is necessary choose a new scale for the y-axis to magnify the interesting area:

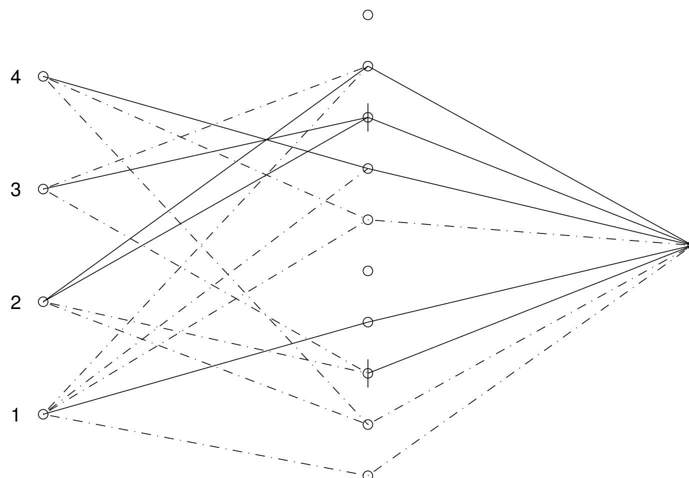
```
>> figure(1), set(gca,'Ylim',[0 0.25]);
```



This plot clearly reveals that the minimum of the test error occurs when there are only 25 weights left in the network. This could also have determined by using the commands:
>> [mintev,index] = min(tev(pv));
>> index=pv(index)

The outcome of the pruning session may seem somewhat peculiar. Ideally one would expect the training error to increase monotonically as the weights are pruned, while FPE estimate and test error should decrease until a certain minimum, and the start increasing as well. However, it is not uncommon that the results look this strange, particularly when the network is recurrent. Also the results depend heavily on from which local minimum one starts out.

The optimal network weights are extracted from the matrix *thd* by using the function *netstruc*, which also displays the network:
>> [W1,W2] = netstruc(NetDef,thd,index);



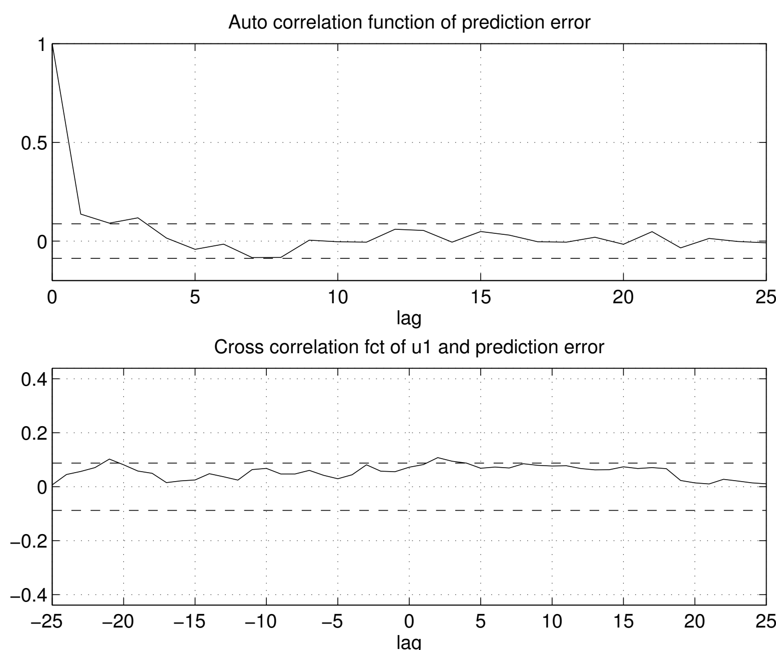
Experience has shown that regularization is helpful when pruning neural networks. However, when the optimal network architecture is found, the network should be retrained without weight decay. This is done by issuing the commands:
>> *trparms = settrain(trparms,'D',0);*
>> *[W1,W2,NSSEvec]=nnoe(NetDef,NN,W1,W2,trparms,y1s,u1s);*

Validation of the Final Network
Start by rescaling the weights so that the validation can be performed on unscaled data (this is just to being able to see the differences to analysis of the estimated linear model):
>> *[w1,w2] = wrescale('nnoe',W1,W2,uscales,yscales,NN);*

Notice (for example by calling *drawnet*) that the biases eliminated during pruning have been reintroduced by the rescaling function.

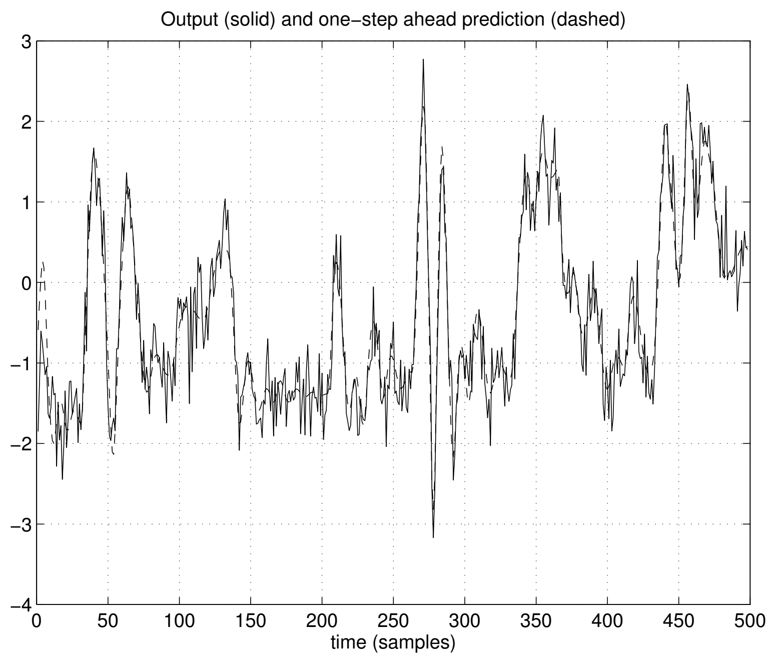
Validate the final model:
>> *[yhat,NSSE] = nnvalid('nnoe',NetDef,NN,w1,w2,y2,u2);*

The correlation coefficients almost stay within thier standard deviations now and thus look far better than those shown previously:
>> figure(2)



The predictions are also much closer to the observed output (see the first subplot in figure(1) produced by *nnvalid* or issue the following sequence of commands):
>> *plot(y1(3:500)), hold on*
>> *plot(yhat,'m--'), hold off*
>> *title('Output (solid) and one-step ahead prediction (dashed)')*

>> xlabel('time (samples)')



4 References

S.A. Billings, H.B., Jamaluddin, S. Chen (1992): *"Properties of Neural Networks With Applications to Modelling non-linear Dynamical Systems"*, Int. J. Control, Vol. 55, No 1, pp. 193-224, 1992.

H. Demuth & M. Beale: *"Neural Network Toolbox: User's Guide, Version 3.0"*, The MathWorks Inc., Natick, MA, 1998.

R. Fletcher (1987):*"Practical Methods of Optimization"*, Wiley, 1987

L.K. Hansen and J. Larsen (1996): *"Linear Unlearning for Cross-Validation",* Advances in Computational Mathematics, Vol. 5, pp. 269-280, 1996.

L.K. Hansen & M. W. Pedersen (1994):*"Controlled Growth of Cascade Correlation Nets"*, Proc. ICANN '94, Sorrento, Italy, 1994, Eds. M. Marinaro & P.G. Morasso, pp. 797-800.

B. Hassibi, D.G. Stork (1993): *"Second Order Derivatives for Network Pruning: Optimal Brain Surgeon"*, NIPS 5, Eds. S.J. Hanson et al., 164, San Mateo, Morgan Kaufmann, 1993.

S. Haykin (1993): *"Neural Networks, A Comprehensive Foundation,"* IEEE Press, 1993.
X. He & H. Asada (1993): *"A New Method for Identifying Orders of Input-Output Models for Nonlinear Dynamic Systems"*, Proc. of the American Control Conf., S.F., California, 1993.

J. Hertz, A. Krogh & R.G. Palmer (1991): "*Introduction to the Theory of Neural Computation"*, Addison-Wesley, 1991.

Y. Le Cun, J:S. Denker, S.A Solla (1989): *"Optimal Brain Damage"*, Advances in Neural Information Processing Systems, Denver 1989, ed. D. Touretzsky, Morgan Kaufmann, pp. 598-605.

Y. Le Cun, I. Kanter, S.A. Solla (1991):*"Eigenvalues of Covariance Matrices: Application to Neural-Network Learning"*, Physical Review Letters, Vol 66, No. 18, pp. 2396-2399, 1991.

L. Ljung (1987):*"System Identification - Theory for the User",* Prentice-Hall, 1987.

L. Ljung (1995): *"System Identification Toolbox User's Guide"*, The MathWorks Inc., 1995

J. Larsen & L.K. Hansen (1994):*"Generalization Performance of Regularized Neural Network Models"*, Proc. of the IEEE Workshop on Neural networks for Signal Proc. IV, Piscataway, New Jersey, pp.42-51, 1994.

K. Madsen (1991):*"Optimering"*, (in danish). Haefte 38, IMM, DTU, 1991

D. Marquardt (1963): *"An Algorithm for Least-Squares Estimation of Nonlinear Parameters,"* SIAM J. Appl. Math. 11, pp. 164-168.

M. Nørgaard, O. Ravn, N. K. Poulsen, L. K. Hansen (2000): *"Neural networks for Modelling and Control of Dynamic Systems,"* Springer-Verlag, London, UK, 2000.

J.E. Parkum (1992): *"Recursive Identification of Time-Varying Systems"*, Ph.D. thesis, IMM, Technical University of Denmark, 1992.

M.W. Pedersen, L.K. Hansen, J. Larsen (1995): *"Pruning With Generalization Based Weight Saliences: γOBD, γOBS"*, Proc. of the Neural Information Processing Systems 8.

M.E. Salgado, G. Goodwin, R.H. Middleton (1988): *"Modified Least Squares Algorithm Incorporating Exponential Forgetting And Resetting"*, Int. J. Control, 47, pp. 477-491.

J. Sjöberg, H. Hjalmerson, L. Ljung (1994):*"Neural Networks in System Identification"*, Preprints 10th IFAC symposium on SYSID, Copenhagen, Denmark. Vol.2, pp. 49-71, 1994.

J. Sjöberg & L. Ljung (1992): *Overtraining, Regularization, and Searching for Minimum in Neural Networks"*, Preprint IFAC Symp. on Adaptive Systems in Control and Signal Processing, Grenoble, France. pp. 669-674.

C. Svarer, L.K. Hansen, J. Larsen (1993): *"On Design and Evaluation of Tapped-Delay Neural Network Architectures"*, The 1993 IEEE Int. Conf. on Neural networks, San Francisco, Eds. H.R. Berenji et al., pp. 45-51.