Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Levenberg-Marquardt-based OBS Algorithm using Adaptive Pruning Interval for System Identification with Dynamic Neural Networks

Christian Endisch, Peter Stolze, Peter Endisch, Christoph Hackl and Ralph Kennel
Institute for Electrical Drive Systems and Power Electronics
Technische Universität München
80333 München, Germany

***Abstract*—This paper presents a pruning algorithm using adaptive pruning interval for system identification with general dynamic neural networks (GDNN). GDNNs are artificial neural networks with internal dynamics. All layers have feedback connections with time delays to the same and to all other layers. The parameters are trained with the Levenberg-Marquardt (LM) optimization algorithm. Therefore the Jacobian matrix is required. The Jacobian is calculated by real time recurrent learning (RTRL). As both LM and OBS need Hessian information, computing time can be saved, if OBS uses the scaled inverse Hessian already calculated for the LM algorithm. This paper discusses the effect of using the scaled Hessian instead of the real Hessian in the OBS pruning approach. In addition to that an adaptive pruning interval is introduced. Due to pruning the structure of the identification model is changed drastically. So the parameter optimization task between the pruning steps becomes more or less complex. To guarantee that the parameter optimization algorithm has enough time to cope with the structural changes in the GDNN-model, it is suggested to adapt the pruning interval during the identification process. The proposed algorithm is verified simulatively for two standard identification examples.**

***Index Terms*—System identification, dynamic neural network, recurrent neural network, GDNN, optimization, Levenberg-Marquardt, real time recurrent learning, network pruning, OBS.**

I. Introduction

A crucial task in system identification with recurrent neural networks is the selection of an appropriate network architecture. It is very difficult to decide a priori which architecture and size are adequate for an identification task. There are no general rules for choosing the structure before identification. A common approach is to try different configurations until one is found that works well. This approach can be very time consuming if many topologies have to be tested. In this paper we start with an oversized general dynamic neural network (GDNN, see Fig. 1) and remove unnecessary parts. In GDNN all layers have feedback connections with many time delays. The output depends not only on the current input, but also on previous inputs and previous states of the network. During the identification process the network architecture is reduced to find a model for the plant as simple as possible. For architecture reduction in static neural networks several pruning algorithms are known [1],[6],[11],[16]. Two well known methods in feedforward neural networks are optimal brain damage (OBD) [11] and optimal brain surgeon (OBS) [6]. Both methods are based on weight ranking due to the saliency, which is defined as the change in the output error using Hessian information. The OBD method calculates the saliency only with the pivot elements of the Hessian without retraining after the pruning step. The OBS uses the complete Hessian information to calculate the saliency, which is regarded as a continuation of the OBD method. Contrary to OBD the OBS algorithm includes a retraining part for the remaining weights. In [3] it is shown that network pruning with OBS can also be applied to dynamic neural networks. In OBS pruning the calculation of the inverse Hessian causes a great amount of computation. To overcome this disadvantage of OBS it is suggested to use the OBS pruning algorithm in conjunction with the LM training algorithm [3],[12]. Thus OBS gets the inverse Hessian for free. The structure of the GDNN is changed drastically during the pruning and identification process. Because of this the influence of one pruning step on the identification system can be quite different. The deletion of one weight in a very large neural network usually does not influence the identification system very much and the error after pruning can be reduced quite fast, whereas the deletion of one weight in a small neural network can influence the system extremely leading to huge model errors. In the latter case the identification system needs more time to cope with the structural changes in the GDNN. In this paper the time between two pruning steps, the so-called pruning interval, is adapted online. The interval adaption makes use of a scaling algorithm by evaluating the mean error between successive pruning steps. The next section presents the recurrent neural network used in this paper. Administration matrices are introduced to manage the pruning process. Section III deals with the parameter optimization method used throughout this paper. In section IV the LM-based OBS algorithm is discussed and the adaptive pruning interval approach is presented. Identification examples are shown in section V. Finally, in section VI we summarize the results.

II. General Dynamic Neural Network (GDNN)

De Jesus described in his doctoral thesis [9] a broad class of dynamic networks, he called the framework layered digital

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

dynamic network (LDDN). The sophisticated formulations and notations of the LDDN allow an efficient computation of the Jacobian matrix using RTRL [4]. Therefore we follow these conventions suggested by De Jesus. In [7]-[10] the optimal network topology is assumed to be known. In this paper the network topology is unknown and so we choose an oversized network for identification. In GDNN all feedback connections exist with a complete tapped delay line (from a first-order time delay element z^{-1} up to the maximum order time delay element $z^{-d_{max}}$). The output of a tapped delay line (TDL) is a vector containing delayed values of the TDL input. Also the network inputs have a TDL. Fig. 1 shows a three-layer GDNN. The simulation equation for layer m is

$$\begin{aligned}\underline{n}^m(t) = &\sum_{l\in L_m^f}\sum_{d\in DL^{m,l}} \underbrace{\underline{LW}^{m,l}(d)}\cdot \underline{a}^l(t-d)+\\ &\sum_{l\in I_m}\sum_{d\in DI^{m,l}} \underbrace{\underline{IW}^{m,l}(d)}\cdot \underline{p}^l(t-d)+\underline{b}^m\end{aligned} \tag{1}$$

$\underline{n}^m(t)$ is the summation output of layer m, $\underline{p}^l(t)$ is the l-th input to the network, $\underline{IW}^{m,l}$ is the input weight matrix between input l and layer m, $\underline{LW}^{m,l}$ is the layer weight matrix between layer l and layer m, $\underline{b}^m$ is the bias vector of layer m, $DL^{m,l}$ is the set of all delays in the tapped delay line between layer l and layer m, $DI^{m,l}$ is the set of all input delays in the tapped delay line between input l and layer m, I_m is the set of indices of input vectors that connect to layer m, L_m^f is the set of indices of layers that directly connect forward to layer m. The output of layer m is

$$\underline{a}^m(t) = \underline{f}^m(\underline{n}^m(t)) \tag{2}$$

where $\underline{f}^m(\cdot)$ are nonlinear activation functions. In this paper we use $tanh$-functions in the hidden layers and linear activation functions in the output layer. At each point of time the equations (1) and (2) are iterated forward through the layers. Time is incremented from $t = 1$ to $t = Q$. (See [9] for a full description of the notation used here). In Fig. 1 below the matrix-boxes and below the arrows the dimensions are shown. R^m and S^m respectively indicate the dimension of the input and the number of neurons in layer m. $\underline{\hat{y}}$ is the output of the GDNN. During the identification process the optimal network architecture should be found. Administration matrices show which weights are valid or not.

A. Administration Matrices

The layer weight administration matrices $\underline{\mathcal{AL}}^{m,l}(d)$ have the same dimensions as the layer weight matrices $\underline{LW}^{m,l}(d)$ of the GDNN, the input weight administration matrices $\underline{\mathcal{AI}}^{m,l}(d)$ have the same dimensions as the input weight matrices $\underline{IW}^{m,l}(d)$ and the bias weight administration vectors $\underline{\mathcal{A}b}^m$ have the same dimensions as the bias weight vectors $\underline{b}^m$. The elements of the administration matrices can have the boolean values 0 or 1, indicating if a weight is valid or not. If e.g. the layer weight $lw_{k,i}^{m,l}(d) = [\underline{LW}^{m,l}(d)]_{k,i}$ from neuron i of layer l to neuron k of layer m with a dth-order time-delay is valid, then $\left[\underline{\mathcal{AL}}^{m,l}(d)\right]_{k,i} = \alpha l_{k,i}^{m,l}(d) = 1$. If the element in the administration matrix equals to zero the corresponding weight has no influence on the GDNN. With these definitions the kth output of layer m can be computed by

$$\begin{aligned}n_k^m(t) = &\sum_{l\in L_m^f}\sum_{d\in DL^{m,l}}\left(\sum_{i=1}^{S^l} lw_{k,i}^{m,l}(d)\cdot\alpha l_{k,i}^{m,l}(d)\cdot a_i^l(t-d)\right)\\ &+\sum_{l\in I_m}\sum_{d\in DI^{m,l}}\left(\sum_{i=1}^{R^l} iw_{k,i}^{m,l}(d)\cdot\alpha i_{k,i}^{m,l}(d)\cdot p_i^l(t-d)\right)\\ &+b_k^m\cdot\alpha b_k^m\\ a_k^m(t) = &f_k^m(n_k^m(t))\end{aligned} \tag{3}$$

where S^l is the number of neurons in layer l and R^l is the dimension of the lth input. Table I shows an example of administration matrices for a three-layer GDNN with 3 neurons in each hidden layer and $d_{max} = 2$ at the beginning of a pruning process. Only the weight $l_{1,3}^{2,2}(1)$ from neuron 3 of layer 2 to neuron 1 of the same layer with first-order time-delay is deleted, because $\alpha l_{1,3}^{2,2}(1) = 0$. All other weights are valid.

TABLE I
EXAMPLE OF ADMINISTRATION MATRICES FOR A THREE-LAYER GDNN, 3 NEURONS IN THE HIDDEN LAYERS AND $d_{max} = 2$

Layer	Administration Matrices							
1	$\underline{\mathcal{AI}}^{1,1}(1)$	$\underline{\mathcal{AI}}^{1,1}(2)$						
	1	1						
	1	1						
	1	1						
	$\underline{\mathcal{AL}}^{1,1}(1)$	$\underline{\mathcal{AL}}^{1,1}(2)$	$\underline{\mathcal{AL}}^{1,2}(1)$	$\underline{\mathcal{AL}}^{1,2}(2)$	$\underline{\mathcal{AL}}^{1,3}(1)$	$\underline{\mathcal{AL}}^{1,3}(2)$	$\underline{\mathcal{A}b}^1$	
	1 1 1	1 1 1	1 1 1	1 1 1	1	1	1	
	1 1 1	1 1 1	1 1 1	1 1 1	1	1	1	
	1 1 1	1 1 1	1 1 1	1 1 1	1	1	1	
2	$\underline{\mathcal{AL}}^{2,1}(0)$		$\underline{\mathcal{AL}}^{2,2}(1)$	$\underline{\mathcal{AL}}^{2,2}(2)$	$\underline{\mathcal{AL}}^{2,3}(1)$	$\underline{\mathcal{AL}}^{2,3}(1)$	$\underline{\mathcal{A}b}^2$	
	1 1 1		1 1 **0**	1 1 1	1	1	1	
	1 1 1		1 1 1	1 1 1	1	1	1	
	1 1 1		1 1 1	1 1 1	1	1	1	
3			$\underline{\mathcal{AL}}^{3,2}(0)$		$\underline{\mathcal{AL}}^{3,3}(1)$	$\underline{\mathcal{AL}}^{3,3}(2)$	$\underline{\mathcal{A}b}^3$	
			1 1 1		1	1	1	

B. Implementation

For the simulations throughout this paper the graphical programming language *Simulink (Matlab)* was used. GDNN, Jacobian calculation, optimization algorithm and pruning were implemented as S-function in *C*.

III. PARAMETER OPTIMIZATION

First of all a quantitative measure of the network performance has to be defined. In the following we use the squared error

$$\begin{aligned}E(\underline{w}_k) &= \frac{1}{2}\cdot\sum_{q=1}^{Q}(\underline{y}_q - \underline{\hat{y}}_q(\underline{w}_k))^T\cdot(\underline{y}_q - \underline{\hat{y}}_q(\underline{w}_k))\\ &= \frac{1}{2}\cdot\sum_{q=1}^{Q}\underline{e}_q^T(\underline{w}_k)\cdot\underline{e}_q(\underline{w}_k)\end{aligned} \tag{4}$$

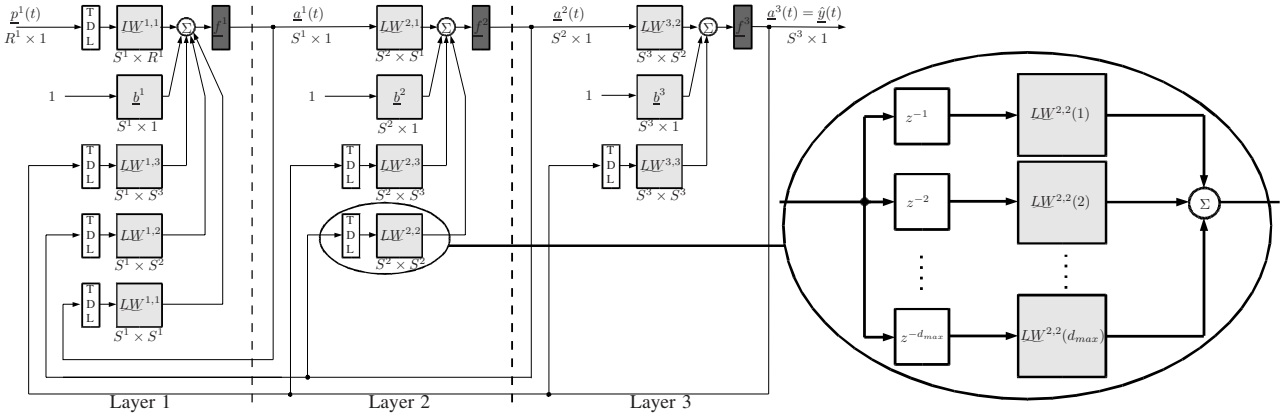

Fig. 1. Three-layer GDNN (two hidden layers)

where q denotes one pattern in the training set, $\underline{y}_q$ and $\underline{\hat{y}}_q(\underline{w}_k)$ are, respectively, the desired target and actual model output of the q-th pattern. The vector $\underline{w}_k$ is composed of all weights in the GDNN. The cost function $E(\underline{w}_k)$ is small if the training (and pruning) process performs well and large if the it performs poorly. The cost function forms an error surface in a $(n+1)$-dimensional space, where n is equal to the number of weights in the GDNN. In the next step this space has to be searched in order to reduce the cost function.

A. Levenberg-Marquardt Algorithm

All Newton methods are based on the second-order Taylor series expansion about the old weight vector $\underline{w}_k$:

$$\begin{aligned} E(\underline{w}_{k+1}) &= E(\underline{w}_k + \Delta\underline{w}_k) \\ &= E(\underline{w}_k) + \underline{g}_k^T \cdot \Delta\underline{w}_k + \frac{1}{2} \cdot \Delta\underline{w}_k^T \cdot \underline{\underline{H}}_k \cdot \Delta\underline{w}_k \end{aligned} \tag{5}$$

If a minimum on the error surface is found, the gradient of the expansion (5) with respect to $\Delta\underline{w}_k$ is zero:

$$\nabla E(\underline{w}_{k+1}) = \underline{g}_k + \underline{\underline{H}}_k \cdot \Delta\underline{w}_k = 0 \tag{6}$$

Solving (6) for $\Delta\underline{w}_k$ gives the Newton method

$$\begin{aligned} \Delta\underline{w}_k &= -\underline{\underline{H}}_k^{-1} \cdot \underline{g}_k^T \\ \underline{w}_{k+1} &= \underline{w}_k - \underline{\underline{H}}_k^{-1} \cdot \underline{g}_k \end{aligned} \tag{7}$$

The vector $-\underline{\underline{H}}_k^{-1} \cdot \underline{g}_k^T$ is known as the Newton direction, which is a descent direction, if the Hessian matrix $\underline{\underline{H}}_k$ is positive definite. There are several difficulties with the direct application of Newton's method. One problem is that the optimization step may move to a maximum or saddle point if the Hessian is not positive definite and the algorithm could become unstable. There are two possibilities to solve this problem. Either the algorithm uses a line search routine (e.g. Quasi-Newton) or the algorithm uses a scaling factor (e.g. LM). The direct evaluation of the Hessian matrix is computationally demanding. Hence the Quasi-Newton approach (e.g. BFGS formula) builds up an increasingly accurate term for the inverse Hessian matrix iteratively, using first derivatives of the cost function only. The Gauss-Newton and LM approach approximate the Hessian matrix by [5]

$$\underline{\underline{H}}_k \approx \underline{\underline{J}}^T(\underline{w}_k) \cdot \underline{\underline{J}}(\underline{w}_k) \tag{8}$$

and it can be shown that

$$\underline{g}_k = \underline{\underline{J}}^T(\underline{w}_k) \cdot \underline{e}(\underline{w}_k) \tag{9}$$

where $\underline{\underline{J}}(\underline{w}_k)$ is the Jacobian matrix

$$\underline{\underline{J}}(\underline{w}_k) = \begin{bmatrix} \frac{\partial e_1(\underline{w}_k)}{\partial w_1} & \frac{\partial e_1(\underline{w}_k)}{\partial w_2} & \cdots & \frac{\partial e_1(\underline{w}_k)}{\partial w_n} \\ \frac{\partial e_2(\underline{w}_k)}{\partial w_1} & \frac{\partial e_2(\underline{w}_k)}{\partial w_2} & \cdots & \frac{\partial e_2(\underline{w}_k)}{\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_Q(\underline{w}_k)}{\partial w_1} & \frac{\partial e_Q(\underline{w}_k)}{\partial w_2} & \cdots & \frac{\partial e_Q(\underline{w}_k)}{\partial w_n} \end{bmatrix} \tag{10}$$

which includes first derivatives only. n is the number of all weights in the neural network and Q is the number of time steps evaluated. With (7), (8) and (9) the Gauss-Newton method can be written as

$$\underline{w}_{k+1} = \underline{w}_k - \left[\underline{\underline{J}}^T(\underline{w}_k) \cdot \underline{\underline{J}}(\underline{w}_k)\right]^{-1} \cdot \underline{\underline{J}}^T(\underline{w}_k) \cdot \underline{e}(\underline{w}_k) \tag{11}$$

Now the LM method can be expressed with the scaling factor μ_k

$$\underline{w}_{k+1} = \underline{w}_k - \left[\underline{\underline{J}}^T(\underline{w}_k) \cdot \underline{\underline{J}}(\underline{w}_k) + \mu_k \cdot \underline{\underline{I}}\right]^{-1} \cdot \underline{\underline{J}}^T(\underline{w}_k) \cdot \underline{e}(\underline{w}_k) \tag{12}$$

where $\underline{\underline{I}}$ is the identity matrix. As the LM algorithm is the best optimization method for small and moderate networks (up to a few hundred weights), this algorithm is used for all simulations in this paper. LM optimization normally is offline. This means that all training patterns have to be available before training is started. In order to get an online training algorithm we use a sliding time window that includes the information of the last Q time steps. With the last Q errors the Jacobian matrix $\underline{\underline{J}}(\underline{w}_k)$ from equation (10) is calculated quasi-online. In every time step the oldest training pattern drops out of the time window and a new one (from the current time step) is added — just like a first in first out (FIFO) buffer. If the time window is large enough it can be assumed that the information content of the training data is constant. With this simple method we are able

to implement the LM algorithm online. As this quasi-online optimization method works surprisingly well it is not necessary to use a recurrent approach like [15]. For the simulations in this paper the window size is set to $Q = 500$ (using a sampling time of 0.01 sec.).

B. Jacobian Calculations

To create the Jacobian matrix, the derivatives of the errors have to be computed, see (10). The GDNN has feedback elements and internal delays, so that the Jacobian cannot be calculated by the standard backpropagation algorithm. There are two general approaches to calculate the Jacobian matrix for dynamic systems: By backpropagation through time (BPTT) [18] or by real time recurrent learning (RTRL) [19]. For Jacobian calculations the RTRL algorithm is more efficient than the BPTT algorithm [10]. According to this the RTRL algorithm is used in this paper. Therefore we make use of the developed formulas of the layered digital dynamic network. The interested reader is referred to [7]-[10],[3] for further details.

IV. OBS Pruning in GDNNs

The goal of OBS pruning is to set one of the GDNN-weights to zero while the cost function given in (4) is minimized. This particular weight is denoted as $w_{k,z}$. In the original OBS algorithm proposed by Hassibi [6] all weights in the model are evaluated by the saliency

$$S_{k,z} = \frac{1}{2} \cdot \frac{w_{k,z}^2}{\left[H_k^{-1}\right]_{z,z}} \tag{13}$$

The weight with the smallest saliency is deleted. The contributed weight update can be calculated by

$$\Delta \underline{w}_k = -\frac{w_{k,z}}{\left[H_k^{-1}\right]_{z,z}} \cdot H_k^{-1} \cdot \underline{i}_z \tag{14}$$

where $\underline{i}_z$ is the unit vector with only the $z-$th element equal to 1.

A. LM-based OBS

The OBS in (13) and (14) requires the complex calculation of the inverse Hessian H_k^{-1}. In subsection III-A the LM optimization method made an approximation of this calculation extended with the LM-scaling factor μ_k

$${}^{*}H_k^{-1} = \left(J(\underline{w}_k)^T \cdot J(\underline{w}_k) + \mu_k \cdot I\right)^{-1} \tag{15}$$

Using this scaled and approximated inverse Hessian matrix ${}^{*}H_k^{-1}$ the OBS algorithm according to (13) and (14) can be rewritten as

$${}^{*}S_{k,z} = \frac{1}{2} \cdot \frac{w_{k,z}^2}{[(J^T(\underline{w}_k) \cdot J(\underline{w}_k) + \mu_k \cdot I)^{-1}]_{z,z}} \tag{16}$$

with the optimum change of the weights

$${}^{*}\Delta \underline{w}_k = -\frac{w_{k,z} \cdot (J^T(\underline{w}_k) \cdot J(\underline{w}_k) + \mu_k \cdot I)^{-1} \cdot \underline{i}_z}{[(J^T(\underline{w}_k) \cdot J(\underline{w}_k) + \mu_k \cdot I)^{-1}]_{z,z}} \tag{17}$$

Thus the OBS approach is obtained with low computational cost.

B. Effect of the scaled Hessian in OBS

Now the question is: How does the LM-scaling factor μ_k in (16) and (17) influence the OBS algorithm? First of all, let us have a look at the trusted region method governed by the LM-scaling factor μ_k. For small values of μ_k the LM optimization (12) works as Newton method (7). In this case the second order Taylor series expansion (5) (with $\underline{g}_k$ and ${}^{*}H_k$) gives a good approximation for the real cost function (4). By contrast, large values of μ_k lead to simple Gradient Descent (GD) optimization with step length μ_k^{-1}, see (12). If so, the second order series expansion (with $\underline{g}_k$ and ${}^{*}H_k$) is not suitable to approximate the real cost function. In LM the Taylor series expansion provides a model for the real cost function, but the model is only trusted within some region around the current search point $\underline{w}_k$ governed by the LM-scaling factor μ_k.

Next we consider the LM-based pruning approach defined by (16) and (17). For small values of μ_k the expression $J(\underline{w}_k)^T \cdot J(\underline{w}_k) + \mu_k \cdot I$ gives a good approximation for the real Hessian matrix and we recover the original OBS-formulas:

$$\begin{aligned} \mu_k \ll 1: \qquad {}^{*}S_{k,z} &\approx S_{k,z} \\ {}^{*}\Delta \underline{w}_k &\approx \Delta \underline{w}_k \end{aligned} \tag{18}$$

More interesting is the case for large values of μ_k. Then the second order series expansion (5) (with $\underline{g}_k$ and ${}^{*}H_k$) is not suitable to approximate the real cost function (4). Thus the expression $J(\underline{w}_k)^T \cdot J(\underline{w}_k) + \mu_k \cdot I$ does not represent the real Hessian and is not suitable for OBS-calculation. For large values of μ_k the LM-based OBS given by (16) and (17) can be reduced to the form:

$$\mu_k \gg 1: \qquad {}^{*}S_{k,z} \approx \frac{\mu_k}{2} \cdot w_{k,z}^2 \tag{19}$$

$${}^{*}\Delta \underline{w}_k \approx -w_{k,z} \cdot \underline{i}_z \tag{20}$$

(19) and (20) comply with a very simple pruning concept (in the following referred to as Magnitude Pruning) [2]: It is supposed that small weights in the neural network are less important than large weights. The squared magnitude of the weight value is used as saliency, see (19). This pruning approach is not the best (even small weights can play an important role in a neural network). However, it is a good alternative pruning method, because for large values of μ_k we do not have Hessian information in order to apply the much more efficient OBS pruning. With (20) only the particular weight with the smallest magnitude is set to zero, no additional weight update is done, the remaining weights are kept constant. This feature is also desirable, since we do not have Hessian information for adequate weight adaption.

The effect of using the scaled Hessian in OBS can be summarized as follows: For small values of μ_k the Hessian information is accurate and the standard OBS is applied. For large LM-scaling factors no suitable Hessian information is available and Magnitude Pruning is obtained as an alternative to OBS. The LM-based OBS switches smoothly between the two pruning methods OBS and Magnitude Pruning.

C. Control on Pruning Success

Every ΔT_p time steps one pruning step is executed. The time constant ΔT_p is called pruning interval. Every pruning step starts with the evaluation of the last pruning step. Therefore the cost function value E_{0p} before pruning is stored in order to judge the success of the pruning operation. The parameter ΔE_{max} defines the maximum relative error increase. This user defined parameter limits the error increase during the pruning process. The last pruning step is canceled if the current cost function value $E(\underline{w}_k)$ is too high compared to the value before pruning (see Fig. 2):

$$\begin{aligned} &\texttt{if } E(\underline{w}_k) > \Delta E_{max} \cdot E_{0p} = \Delta E_{max} \cdot E(\underline{w}_{k-\Delta T_p}) \\ &\quad \texttt{undo last pruning step} \end{aligned} \tag{21}$$

Therefore also the GDNN weights have to be stored before pruning.

D. Adaptive Pruning Interval

Usually the cost function increases after a pruning step.The parameter optimization procedure between the pruning steps tries to reduce the error. It can be observed that the deletion of one weight in an oversized GDNN does not have too much influence on the identification process. There are many redundant weights available. By contrast, the smaller the network, the more complex the retraining process after a pruning step. For this reason the pruning operation can be improved, if the pruning interval ΔT_p is adapted online. It must be assured that the optimization algorithm is able to cope with the structural changes in the GDNN-model within the pruning interval ΔT_p. First of all we have to find a measure if the pruning interval ΔT_p is accurately chosen. Therefore we recommend to average over the last ΔT_p cost function values between the pruning steps (see Fig. 2 and 3):

$$E_{mean} = \frac{1}{\Delta T_p} \cdot \big(E(w_k) + E(w_{k-1}) + \ldots + E(w_{k-\Delta T_p+1}) \big) \tag{22}$$

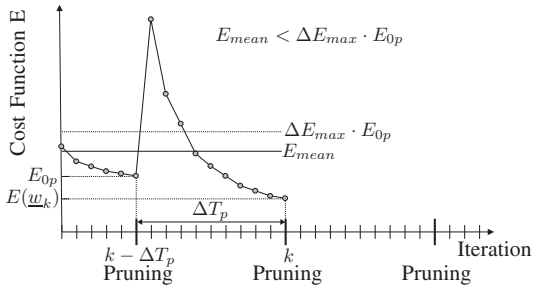

Fig. 2. Example of an adequate pruning interval with $\Delta T_p = 10$.

This mean cost function indicates if the current pruning interval ΔT_p is suitable. If the mean error E_{mean} is high, i. e. the optimization algorithm cannot cope with the new optimization task within the time ΔT_p, the pruning interval should be increased, see Fig. 3. If the mean error E_{mean} is small, the pruning process could be sped up by decreasing the pruning interval. Fig. 2 depicts an example for an adequately chosen pruning interval. The pruning interval ΔT_p is adapted by the following scaling algorithm:

$$\begin{aligned} &\texttt{if} \quad E_{mean} > \Delta E_{max} \cdot E_{0p} \\ &\qquad \Delta T_p = \vartheta \cdot \Delta T_p \\ &\texttt{else} \\ &\qquad \Delta T_p = \frac{1}{\vartheta} \cdot \Delta T_p \end{aligned} \tag{23}$$

with the user defined scaling factor $\vartheta > 1$. The following identification examples in section V show the adaption of the pruning interval ΔT_p during the pruning and parameter optimization process.

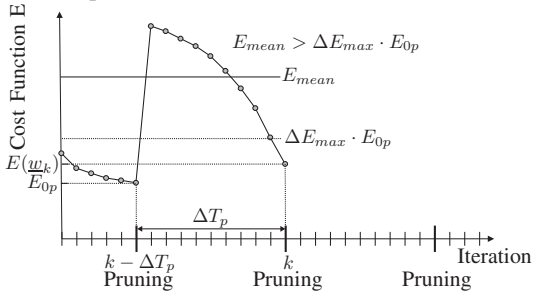

Fig. 3. Pruning interval is too short

V. Identification

In this section two identification examples are presented using the LM-based pruning algorithm suggested in (16) and (17) with adaptive pruning interval proposed in (22) and (23), Jacobian calculation with RTRL and the LM algorithm (12). The first example to be identified is a simple linear PT_2 system. This example is chosen to test the architecture selection ability of the suggested pruning approach. This test is possible for linear systems because we know the required difference equation. If the pruning algorithm with adaptive pruning interval succeeds in deleting the right weights, the final GDNN-model has to consist of the parameters of the difference equation. The second example is a more complex one. A nonlinear dynamic system has to be identified.

A. Excitation Signal

In system identification it is a very important task to use an appropriate excitation signal. A nonlinear dynamic system requires that all combinations of frequencies and amplitudes (in the system's operating range) are represented in the signal. In this paper an APRBS-signal (Amplitude Modulated Pseudo Random Binary Sequence) is applied, uniformly distributed from -1 to +1 in amplitudes and from $10ms$ to $250ms$ in hold time. For more information the reader is referred to [14].

B. Identification of PT_2 plant

The simple PT_2-plant $\frac{10}{0.1 \cdot s^2 + s + 100}$ is identified by a three-layer GDNN (with three neurons in the hidden layers and $d_{max} = 2 \Rightarrow n = 93$). This network architecture is obviously larger than necessary. If the continuous-time PT_2-plant is

discretized using zero-order hold [17] with a sample time of $10ms$ the discrete model can be expressed as

$$\begin{aligned} y(k) &= 0.0048 \cdot u(k-1) + 0.0046 \cdot u(k-2) \\ &\quad +1.8100 \cdot y(k-1) - 0.9048 \cdot y(k-2) \end{aligned} \quad (24)$$

The initial pruning interval is set to $\Delta T_p = 5$, the maximum error increase is $\Delta E_{max} = 10$ and the scaling factor ϑ is defined to 1.1. The pruning process starts after iteration $Q = 500$, when the sliding time window is filled up. Fig.

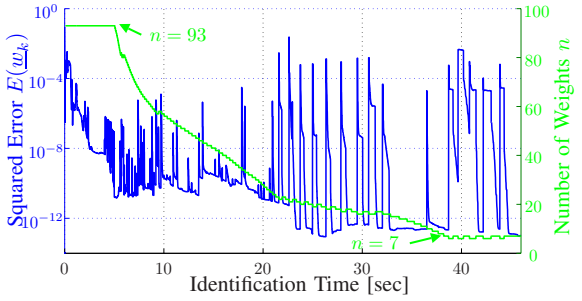

Fig. 4. Squared Error $E(\underline{w}_k)$ and number of weights n for PT_2-identification example (at the beginning: $n = 93$, at the end: $n = 7$)

4 shows the weight reduction from $n = 93$ to $n = 7$ and the identification error, which is also reduced although the model is becoming smaller. After about 20 seconds a lot of pruning steps are canceled due to (21). Table II summarizes

TABLE II
EXAMPLE OF ADMINISTRATION MATRICES FOR A THREE-LAYER GDNN, 3 NEURONS IN THE HIDDEN LAYERS AND $d_{max} = 2$

Layer	Administration Matrices						
1	$\mathcal{AI}^{1,1}(1)$	$\mathcal{AI}^{1,1}(2)$					
	1	0					
	0	0					
	0	0					
	$\mathcal{AL}^{1,1}(1)$	$\mathcal{AL}^{1,1}(2)$	$\mathcal{AL}^{1,2}(1)$	$\mathcal{AL}^{1,2}(2)$	$\mathcal{AL}^{1,3}(1)$	$\mathcal{AL}^{1,3}(2)$	$\mathcal{Ab}^1$
	0 0 0	0 0 0	0 0 0	0 0 0	0	0	0
	0 0 0	0 0 0	0 0 0	0 0 0	0	0	0
	0 0 0	0 0 0	0 0 0	0 0 0	0	0	0
2	$\mathcal{AL}^{2,1}(0)$		$\mathcal{AL}^{2,2}(1)$	$\mathcal{AL}^{2,2}(2)$	$\mathcal{AL}^{2,3}(1)$	$\mathcal{AL}^{2,3}(1)$	$\mathcal{Ab}^2$
	1 0 0		0 0 0	0 0 0	0	0	0
	1 0 0		1 0 0	0 0 0	0	0	0
	0 0 0		0 0 0	0 0 0	0	0	0
3			$\mathcal{AL}^{3,2}(0)$		$\mathcal{AL}^{3,3}(1)$	$\mathcal{AL}^{3,3}(2)$	$\mathcal{Ab}^3$
			0 1 0		1	1	0

the administration matrices after the identification and pruning process. Most of the weights are deleted. Along with the weight vector

$$\begin{aligned} \underline{w}_{k=4600} &= [\text{-0.0027} \quad \text{-2.6959} \quad \text{-0.6168} \quad 0.2213 \\ &\quad 2.8668 \quad 1.8105 \quad \text{-0.9048}]^T, \end{aligned} \quad (25)$$

we are able to draw the final GDNN-model, see Fig. 5. Each circle in the signal flow graph includes a summing junction and an activation function. The input and the output of the model have two time delays. Due to the very small input weight -0.0027 all the tanh-activation functions can be neglected (only the linear region near the origin with slope 1 is used) and

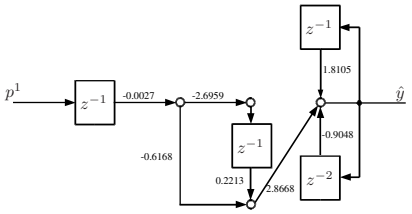

Fig. 5. Identification result: GDNN-model of PT_2 the example

we get the following difference equation for the final GDNN-model:

$$\begin{aligned} \hat{y}(k) &= 0.0048 \cdot u(k-1) + 0.0046 \cdot u(k-2) \\ &\quad +1.8105 \cdot y(k-1) - 0.9048 \cdot y(k-2) \end{aligned} \quad (26)$$

This equation looks similar to the real difference equation of the system (24). Fig. 6 depicts another PT_2-identification using the same initial pruning interval $\Delta T_p = 5$ but no adaption is conducted. The optimization algorithm cannot cope with the structural changes and the identification error increases. Fig. 7

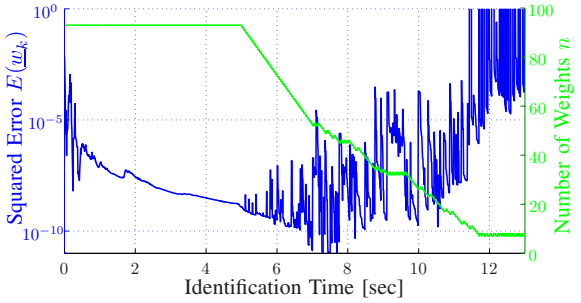

Fig. 6. Constant pruning interval is too small: The Squared Error $E(\underline{w}_k)$ increases.

shows the output of the PT_2-plant y (solid gray line) and the output of the GDNN-model $\hat{y}$ (thick dashed black line) for the first 3 seconds (sample time: $10ms$).

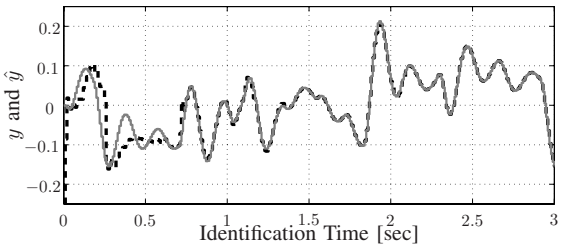

Fig. 7. Outputs of the PT_2-plant y (solid gray line) and of the GDNN-model $\hat{y}$ (thick dashed black line) for the first 3 seconds.

C. Identification of a nonlinear plant

The first example is chosen to underpin the weight reduction ability of the proposed pruning algorithm. In this subsection

a complex nonlinear dynamic system presented by Narendra [13] is considered:

$$y(k) = \frac{y(k-1) \cdot y(k-2) \cdot y(k-3) \cdot u(k-2) \cdot [y(k-3)-1] + u(k-1)}{1 + y^2(k-2) + y^2(k-3)}$$

For this identification example a three-layer GDNN (with four neurons in the hidden layers and $d_{max} = 3 \Rightarrow n = 212$) is used. The initial pruning interval is $\Delta T_p = 100$, the maximum error increase is set to $\Delta E_{max} = 20$ and the scaling factor ϑ is defined to 1.1. Fig. 8 depicts the weight reduction from $n =$

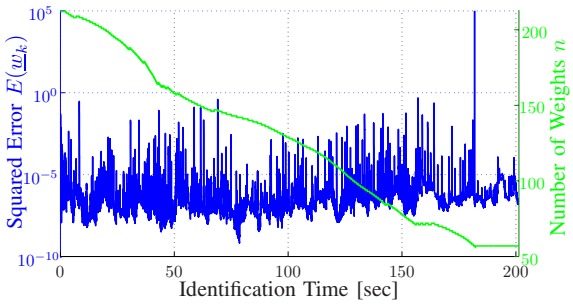

Fig. 8. Squared Error $E(\underline{w}_k)$ and number of weights n for nonlinear identification example (at the beginning: $n = 212$, at the end: $n = 57$)

212 to $n = 57$ and the identification error for 200 seconds. The pruning process is stopped after 182 seconds due to the abrupt rise of the cost function. The last pruning step is undone. Fig.

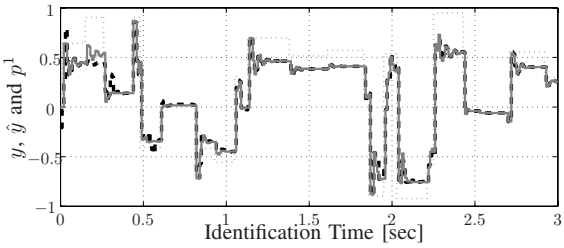

Fig. 9. Output of the nonlinear plant y (solid gray line), output of GDNN-model $\hat{y}$ (thick dashed black line) and APRBS excitation signal p^1 (thin dashed black line) for the first 3 seconds.

9 shows the output of the nonlinear plant y (solid gray line), the output of GDNN-model $\hat{y}$ (thick dashed black line) and the APRBS excitation signal (thin dashed black line) during the first 3 seconds (sample time: $10ms$).

VI. Conclusion

This paper uses the LM-based OBS approach for system identification with GDNNs. The advantage of this pruning method is that the OBS-method exploits Hessian information already calculated for LM parameter optimization. The LM-based OBS switches smoothly between the two pruning methods OBS and Magnitude Pruning governed by the LM-scaling factor. To manage the pruning process in GDNN administration matrices are introduced. These matrices indicate which weights still exist and which are already deleted. In every pruning step the success of the last pruning step is checked. If the increase in the error is too high, the last pruning step is canceled. For this revision the weights have to be stored. The structure of the GDNN is changed drastically during the pruning and identification process. So the influence of one pruning step to the identification system can be quite different. To guarantee that the parameter optimization algorithm has enough time to cope with the structural changes in the GDNN-model, this paper suggests an adaptive pruning interval. We defined the mean cost function value to measure the success of the current pruning interval. Depending on this value the pruning interval is adapted by a scaling algorithm. The proposed pruning algorithm is verified by one simple linear and one complex nonlinear dynamic system identification example. Network pruning does not only work for static neural networks. These structure finding algorithms offer a very interesting application in system identification with dynamic neural networks.

References

[1] M. Attik, L. Bougrain, F. Alexandre, "Optimal Brain Surgeon Variants For Feature Selection," in IEEE Proc. of the Intern. Joint Conf. on Neural Networks, pp. 1371–1374, 2004.

[2] C. M. Bishop, Neural Networks for Pattern Recognition. Oxford University Press, 1995.

[3] C. Endisch, C. Hackl and D. Schröder, "Optimal Brain Surgeon For General Dynamic Neural Networks," in Lecture Notes in Artificial Intelligence (LNAI) 4874, Springer-Verlag Berlin, pp. 15-28, 2007.

[4] C. Endisch, P. Stolze, C. Hackl and D. Schröder, "Comments on Backpropagation Algorithms for a Broad Class of Dynamic Networks," IEEE Trans. on Neural Networks, vol. 20, no. 3, pp. 540–541, 2009.

[5] M. Hagan, B. M. Mohammed, "Training Feedforward Networks with the Marquardt Algorithm," IEEE Trans. on Neural Networks, vol. 5, no. 6, pp. 989–993, 1994.

[6] B. Hassibi, D. G. Stork, G. J. Wolff, "Optimal Brain Surgeon and General Network Pruning," IEEE Intern. Conf. on Neural Networks, vol. 1, pp. 293–299, 1993.

[7] O. De Jesús, M. Hagan, "Backpropagation Algorithms Through Time for a General Class of Recurrent Network," IEEE Int. Joint Conf. Neural Network, Washington, pp. 2638–2643, 2001.

[8] O. De Jesús, M. Hagan, "Forward Perturbation Algorithm For a General Class of Recurrent Network," IEEE Int. Joint Conf. Neural Network, Washington, pp. 2626–2631, 2001.

[9] O. De Jesús, "Training General Dynamic Neural Networks," Ph.D. dissertation, Oklahoma State University, Stillwater, OK, 2002.

[10] O. De Jesús, M. Hagan, "Backpropagation Algorithms for a Broad Class of Dynamic Networks," IEEE Trans. on Neural Networks, vol. 18, no. 1, pp. 14–27, 2007.

[11] Y. Le Cun, J. S. Denker, S. A. Solla, "Optimal Brain Damage," in D. S. Touretzky, "Advances in Neural Information Processing Systems," Morgan Kaufmann, pp. 598–605, 1990.

[12] J. Meng, "Penalty OBS Scheme for Feedforward Neural Network," Proceedings of the 17th IEEE Intern. Conf. on Tools with Artificial Intelligence (ICTAI'05), pp. 479–483, 2005.

[13] K. S. Narendra, K. Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks," IEEE Trans. on Neural Networks, vol. 1, no. 1, pp. 4–27, 1990.

[14] O. Nelles, Nonlinear System Identification. Berlin Heidelberg New York: Springer-Verlag, 2001.

[15] L.S.H. Ngia, J. Sjöberg, "Efficient Training of Neural Nets for Nonlinear Adaptive Filtering Using a Recursive Levenberg-Marquardt Algorithm," IEEE Trans. on Signal Processing, vol. 48, no. 7, pp. 1915–1927, 2000.

[16] R. Reed, "Pruning Algorithms – A Survey," IEEE Trans. on Neural Networks, vol. 4, no. 5, pp. 740–747, 1993.

[17] D. Schröder, Elektrische Antriebe - Regelung von Antriebssystemen. 2nd edn., Berlin Heidelberg New York: Springer-Verlag, 2001.

[18] P.J. Werbos, "Backpropagation Through Time: What it is and how to do it," Proc. IEEE, vol. 78, no. 10, pp. 1550-1560, 1990.

[19] R.J. Williams, D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," Neural Computing, vol. 1, pp. 270–280, 1989.