Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 53-2 (2020) 1175–1181

Deep Learning and System Identification ⋆

Lennart Ljung * **Carl Andersson** ** **Koen Tiels** ***
Thomas B. Schön **

* *Division of Automatic Control, Linköping University, Linköping, Sweden (e-mail: Lennart.Ljung@liu.se)*
** *Dept of Information Technology, Uppsala University, 75105 Uppsala, Sweden (e-mail {Carl.Andersson, Thomas.Schon}@it.uu.se)*
*** *Was with the dept in Uppsala. Now at Dept of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands. (e-mail: k.tiels@tue.nl)*

Abstract: Deep learning is a topic of considerable interest today. Since it deals with estimating – or learning – models, there are connections to the area of System Identification developed in the Automatic Control community. Such connections are explored and exploited in this contribution. It is stressed that common deep nets such as feedforward and cascadeforward nets are nonlinear ARX (NARX) models, and can thus be easily incorporated in System Identification code and practice. The case of LSTM nets is an example of NonLinear State-Space (NLSS) models.

Keywords: Model structure, Bias/Variance Trade-off, Model Validation, LSTM, Cascadeforwardnet, Deep nets

1. INTRODUCTION

Deep Learning is a recent and most active and present topic in today's research on modeling and estimation. Tremendous success has been reached in diverse and important application areas, like computer vision, Krizhevsky et al. [2012], and natural language processing, Mikolov et al. [2013], just to mention a few.

System identification is a classical topic in Automatic Control, since 1956, Zadeh [1956], and is well established with many textbooks, e.g. Ljung [1999], software packages and numerous reported applications.

Both topics deal with the same underlying problem: *to infer models from observed data*. Therefore, there are several links between the two topics and essential considerations when using them. It is the purpose of this contribution to illuminate these connections. This allows the two areas to exchange useful practices, like considering bias-variance trade-offs and residual analysis. It will also be very important to evaluate various deep networks for system identification applications and thus to provide further important model structures and tools for this area.

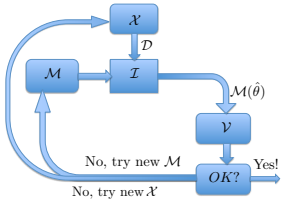
2. THE SYSTEM IDENTIFICATION LOOP

System identification is concerned with building mathematical models of dynamical systems based on measurements of input and output signals.

There are four essential decisions in a system identification problem:

(1) Design an experiment and collect the *data*.
(2) Decide upon a *model structure*.
(3) *Estimate the parameters* of the model structure by adjusting it to the data.
(4) *Validate* the model.

Typically some of the choices – especially the choice of model structure in conjunction with model validation – may have to be revised in the course of solving the problem. Therefore the system identification process in practice is a "loop", see e.g. the figure below or Fig 17.1 in Ljung [1999].



$\mathcal{X}$: The Experiment
$\mathcal{D}$: The Measured Data
$\mathcal{M}$: The Model Set
$\mathcal{I}$: The Identification Method
$\mathcal{V}$: The Validation Procedure

2.1 Model Structures

The key item in any modeling task is to decide on the set of models $\mathcal{M}$ where to seek the model. Many model structures for dynamical systems have been suggested. A generic way to think about a model is to see it as a *predictor* for the next output based on previous input-output observations. It can be denoted by $\hat{y}(t|\theta)$, where θ is a parameter ranging over a given parameter set. Especially for nonlinear systems, many different structures have been suggested, see Schoukens and Ljung [2019] for a recent survey. Two specific structures will be singled out for the purpose of the present contribution:

⋆ This research was financially supported by the Swedish Foundation for Strategic Research (SSF) via the project *ASSEMBLE* (contract number: RIT15-0012) and by the Swedish Research Council via the projects *Learning flexible models for nonlinear dynamics* (contract number: 2017-03807) and *NewLEADS - New Directions in Learning Dynamical Systems* (contract number: 621-2016-06079). Ljung's work was supported by Vinnova's Competence Center LinkSic.

10.1016/j.ifacol.2020.12.1329

NARX Models The model uses a (static) nonlinear function of a regression vector $\varphi(t)$ formed from past data, typically a finite collection of past input and output values:

$$\hat{y}(t|\theta) = f(\varphi(t), \theta) \tag{1}$$

NLSS Models A general NonLinear State-Space model is formed as

$$x(t+1) = f(x(t), y(t), u(t), \theta) \tag{2a}$$
$$\hat{y}(t|\theta) = h(x(t), \theta) \tag{2b}$$

where f and h can be parametrized by θ in many different ways. Often state-space models are discussed without letting y direcly affect the state. However for a complete prediction model it is necessary to allow f to depend on y. Also grey-box modeling of mechanical systems requires this, e.g. Noël and J.Schoukens [2017]

2.2 Parameter Estimation

The estimation of the parameters is essentially done by minimizing the fit between observed outputs and predicted model outputs. Conceptually

$$\min_{\theta} \sum_{t=1}^{N} \|y(t) - \hat{y}(t|\theta)\|^2 \tag{3}$$

where $\hat{y}(t|\theta)$ is the predicted output according to the model $\mathcal{M}(\theta)$.

This is a simple and natural choice, that corresponds to the Maximum Likelihood estimate under mild conditions, Ljung [1999]. It is common practice to append a *regularization term* $\rho(\theta)$ to (3):

$$\min_{\theta} \sum_{t=1}^{N} \|y(t) - \hat{y}(t|\theta)\|^2 + \lambda\rho(\theta); \quad \text{e.g. } \rho(\theta) = \theta^T R\theta \tag{4}$$

to curb the flexibility of the model structure. Another common technique for curbing the flexibilty is to apply *early stopping*, i.e. to stop the search for the minimum, before the lowest value (for the estimation data) has been found, e.g. Sjöberg and Ljung [1995]

2.3 Bias/Variance Trade-off

The error in the estimated model has two components: *the bias*, which is the distance between the true system and the best model available in the model set, and *the variance* which is the distance between the best model available and the obtained estimate, due to noise and imperfections in the data. As the complexity/flexibility of the model set increases, the bias will decrease and the variance will increase.

This means that to have a small total error, we must seek a model of sufficient flexibility with as few free parameters as possible. This favors so called *grey-box models*, where physical insight is used for determining the model structure, but it also means that it is useful to employ parameters that can be used to model various parts of the system.

2.4 Model Validation

Model validation means that we should gain confidence that the estimated model is capable of covering essential parts of the system behavior - not only for the estimation data used for its estimation.

A simple and common way to do this is *Cross Validation*: Collect a validation data set, that is different from the estimation data set. Simulate the model for the validation input and compare that model output with the measured validation output. Make the comparison by eye inspection or compute a numerical measure of the fit.

2.5 Residual Analysis

A further validation test is to compute the residuals, "the leftovers" in the model $\varepsilon(t)$, the part of the output that the model could not explain or reproduce. These should be independent or at least uncorrelated with information that was available when forming the model output.

Residual analysis means that typically the autocorrelation function for the residuals is computed along with the cross correlation function between residuals and past inputs. It is then evaluated that these functions do not deviate too much from zero.

3. LEARNING AND DEEP LEARNING

The core of all learning can be said to be *function estimation*, i.e. measure noisy observations of a function between two spaces $\mathcal{X}$ to $\mathcal{Y}$

$$y_t = \mathcal{F}(x_t) + \text{ noise}, \qquad t = 1, \dots, N \tag{5}$$

and infer the function $\mathcal{F}$, where $x_t \in \mathcal{X}$ denotes the input and $y_t \in \mathcal{Y}$ denotes the output. This is like a classical curve-fitting problem, but the spaces may be of very general nature: $\mathcal{X}$ could be signals, images, texts, etc., and $\mathcal{Y}$ could be numerical values or classifications. It is the broad range of possible spaces that allows the problem formulation to be applied to a wide variety of tasks, which is the reason for the current intense interest in the field. But for standard numerical spaces $\mathcal{X}$ and $\mathcal{Y}$, the formulation (5) remains an example of standard statistical *nonlinear regression*. Most applications of machine learning have a *black-box* view of $\mathcal{F}$, i.e. no physical knowledge about the function is employed, but only flexible mappings are used when estimating it.

A typical example of flexible mappings used in machine learning are *neural networks*, e.g. Kung [1993]. See also (6) below. They have the capability of allowing arbitrarily accurate approximation of any reasonable function, and have been successfully used in many applications. Recently it has been suggested to use further expansions of the neural network concept, so called *deep networks* for function estimation in (5). This is known as *deep learning*, and has proved to be overwhelmingly successful in many applications.

4. DEEP LEARNING MODEL STRUCTURES

We can think of the general function approximators for $\mathcal{F}$ in (5) as a sequential construction of several generalized linear regressions, i.e. repetitive use of linear regression and static nonlinearities. This is, loosely, the idea behind neural networks.

4.1 Single hidden layer Feedforward Neural Networks

A neural network is a function mapping from a d_0-dimensional vector x to a (scalar) y that is often written as (assuming a linear output layer)

$$y = \sum_{\ell=1}^{d_1} \gamma_\ell \kappa(\alpha_\ell(x + \beta_\ell)) \tag{6}$$

Here κ is known as the *activation function* and is a nonlinear function from $\mathcal{R}$ to $\mathcal{R}$. The parameters α and β are quantities of sizes adjusted to the dimension of x. The numbers $\theta = \{\alpha_\ell, \beta_\ell, \gamma_\ell\}_{\ell=1}^{d_1}$ are the parameters of the model. A classical choice of κ is the sigmoid

$$\kappa(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

which gives a *sigmoidal network*. A very common choice today is the rectified linear unit (ReLU) function

$$\kappa(z) = z_+ = \max(z, 0) \tag{8}$$

which makes the sum (6) piece-wise linear with breakpoints in $-\beta_\ell$ for $\ell = 1, \ldots, d_1$. In the remainder of this contribution we restrict ourselves to RELU activation functions, since it is a common choice, and allows more compact calculations and expressions.

The mapping (6) is the classical *one-hidden layer feedforward neural network*. The terms $\kappa(\alpha_\ell(x + \beta_\ell))$ are the *hidden units* and together they form the hidden layer.

There are many ways to extend this idea to more sophisticated nets, see e.g. Goodfellow et al. [2016], Pascanu et al. [2014].

4.2 Multilayer Feedforward Neural Networks

To elaborate on the one-hidden layer network we can cascade it with similar structures as further layers. With notation inspired by Gilbert Strang, Strang [2018], the ReLU net can be described as follows: Let the nonlinear function $\mathbf{F}_1$ for given A_1, b_1 be defined as

$$\mathbf{F}_1(x) = (A_1 x + b_1)_+ \tag{9}$$

where (8) is interpreted element-wise. If A_1 $(d_1 \times d_0)$ is built up from α_ℓ and $b_1(d_1 \times 1)$ is formed from β_ℓ and α_ℓ, it is easy to see that $\mathbf{F}_1(x)$ are exactly the hidden units in (6) with the ReLU activation function. So the net (6) is simply $C\mathbf{F}_1(x)$ with C defined from γ_ℓ. Seeing the hidden units as new "regressors" (corresponding to x), applying the ReLU mapping to them and adding the obtained outputs linearly with C we get a cascaded mapping, defined as

$$C\mathbf{F}_2(\mathbf{F}_1(x)) \tag{10}$$

and we have a net with two hidden layers with d_1, and d_2 hidden units, respectively. Here

$$\mathbf{F}_2(x) = (A_2 x + b_2)_+ \quad (A_2 \text{ is } d_2 \times d_1 \text{and } b_2 \text{ is } d_2 \times 1) \tag{11}$$

Repeating, a net with L hidden layers is obtained by

$$y = C\mathbf{F}_L(\mathbf{F}_{L-1}(...\mathbf{F}_2(\mathbf{F}_1(x)))) \tag{12}$$

This is what is called `feedforwardnet` in the MATLAB Deep Learning Toolbox. This network corresponds to the so called *TCN (Temporal Convolutional Network)* in Andersson et al. [2019]. The TCN however uses a convolutional neural network instead of a feedforward neural network (see Goodfellow et al. [2016] for an introduction to the convolutional neural network).

A variant is called `cascadeforwardnet` in the Deep Learning Toolbox. It corresponds to the case where all previous hidden units are used as regressors in the next layer, not only the ones from the previous layer. That is, in (10) $\mathbf{F}_2$ will be a function of x, $\mathbf{F}_1(x)$, etc.

It is important to note, that all the deep networks described in this section are NARX models (1).

The output from the net, i.e. $y = \hat{y}(t|\theta)$ is a static mapping from $x = \varphi(t)$. The parameter θ collects all the elements in the matrices C, A_i, b_i $i = 1, ..., L$, $\hat{y}(t|\theta)$ is the predicted model output at time t and $\varphi(t)$ is the *regression vector*, typically containing past inputs and outputs.

5. RECURRENT AND LSTM NEURAL NETWORKS

5.1 Recurrent neural network (RNN)

The feedforward nets, the NARX models, can only convey the information to the next time step $t + 1$ from the finite number of past inputs and outputs in the regression vector $\varphi(t)$. To let past information linger longer, it is possible to extend the regression vector by introducing components formed from the hidden units in the previous time step. That creates a "feedback loop" around the net (see e.g. fig 5.3 in Ljung [1999]) and makes the calculations *recurrent* over time, That means that hidden units will be an unobserved recurrent state, $x(t)$, that is capable of conveying information from a more distant past. The same idea can of course be used also for multilayer feedforward nets. The resulting *Recurrent Neural Network, (RNN*) will have the format of an NLSS model (2) with information in the state that can potentially linger infinitely long, due to the feedback (which also may create instability). It will also allow the modeling of so called Nonlinear Output Error (NOE) models, see e.g. Ljung [1999], Section 5.4. The prediction $\hat{y}(t|\theta)$ is thus a nonlinear mapping of this state and can depend on more distant information than it would with a NARX model with regression vector $\varphi(t)$.

5.2 LSTM Network

A feature of an RNN is that at each time step, as the state is updated it will be modified with new information, and old information may decay. An idea to secure old information for a longer time is the Long Short-Term Memory (LSTM) network, Hochtreiter and Schmidhuber [1997]. It has two kinds of states, a *cell state* $c_t^{(\ell)}$, for long term memory and a *hidden state* $h_t^{(\ell)}$ for short term memory. It also has *gates* for controlling how information is added and removed from the cell state.

The resulting net consists of the cascade of one or more LSTM layers and a fully connected linear layer. An LSTM layer is more complicated than the classical one-hidden layer network in (6), but is still a nonlinear mapping from its input to its output.

A single LSTM layer consists of four so-called gates and two states. In particular, the ℓth layer has input gate $i_t^{(\ell)}$, forget gate $f_t^{(\ell)}$, cell gate $g_t^{(\ell)}$, output gate $o_t^{(\ell)}$, cell state

$c_t^{(\ell)}$, and hidden state $h_t^{(\ell)}$. The mapping from the input $\tilde{u}_t^{(\ell)}$ to the output $\tilde{y}_t^{(\ell)} = h_t^{(\ell)}$ of the ℓth layer is as follows:

$$i_t^{(\ell)} = \sigma\left(W_{\text{ii}}^{(\ell)}\tilde{u}_t^{(\ell)} + b_{\text{ii}}^{(\ell)} + W_{\text{hi}}^{(\ell)}h_{t-1}^{(\ell)} + b_{\text{hi}}^{(\ell)}\right) \quad (13a)$$

$$f_t^{(\ell)} = \sigma\left(W_{\text{if}}^{(\ell)}\tilde{u}_t^{(\ell)} + b_{\text{if}}^{(\ell)} + W_{\text{hf}}^{(\ell)}h_{t-1}^{(\ell)} + b_{\text{hf}}^{(\ell)}\right) \quad (13b)$$

$$g_t^{(\ell)} = \tanh\left(W_{\text{ig}}^{(\ell)}\tilde{u}_t^{(\ell)} + b_{\text{ig}}^{(\ell)} + W_{\text{hg}}^{(\ell)}h_{t-1}^{(\ell)} + b_{\text{hg}}^{(\ell)}\right) \quad (13c)$$

$$o_t^{(\ell)} = \sigma\left(W_{\text{io}}^{(\ell)}\tilde{u}_t^{(\ell)} + b_{\text{io}}^{(\ell)} + W_{\text{ho}}^{(\ell)}h_{t-1}^{(\ell)} + b_{\text{ho}}^{(\ell)}\right) \quad (13d)$$

$$c_t^{(\ell)} = f_t^{(\ell)} * c_{t-1}^{(\ell)} + i_t^{(\ell)} * g_t^{(\ell)} \quad (13e)$$

$$h_t^{(\ell)} = o_t^{(\ell)} * \tanh\left(c_t^{(\ell)}\right) \quad (13f)$$

where $\sigma(\cdot)$ is a sigmoid function (see (7)) operating element-wise and $*$ is the Hadamard (element-wise) product. The weight matrices W and the bias vectors b contain the parameters of the LSTM layers. The *Number of Units* in an LSTM layer is the dimension of the hidden and the cell states. That also defines the dimensions of the matrices W and b. A useful tutorial for the LSTM layer is provided by Olah [2015].

The LSTM layers are connected by taking the output of a layer as the input to the following layer: $\tilde{u}_t^{(\ell)} = h_t^{(\ell-1)}$ for $\ell = 2, \ldots, L$. The output of the last LSTM layer is the input to a fully connected linear layer: $y_t = W_{\text{fc}}h_t^{(L)} + b_{\text{fc}}$.

It is important to note that an LSTM network is an NLSS model, (2). Indeed, when considering a state vector $x(t)$ that collects the cell and hidden states in all layers ($x_t^{(\ell)} = \left[(c_t^{(\ell)})^\mathsf{T}\ (h_t^{(\ell)})^\mathsf{T}\right]^\mathsf{T}$ and $x(t) = \left[(x_t^{(1)})^\mathsf{T}\ \ldots\ (x_t^{(L)})^\mathsf{T}\right]^\mathsf{T}$), input $u(t) = \tilde{u}_{t+1}^{(1)}$, output $\hat{y}(t|\theta) = y_t$, and a parameter vector θ that collects all the elements in the weight matrices W and the bias vectors b, the LSTM network is an NLSS model as in (2). More explicitly, the output y_t is a function of $h_t^{(L)}$, which is part of the state vector $x(t)$, so of the form (2b). For the state equation (2a), observe from (13) that $x_{t+1}^{(\ell)}$ depends on the states in the same layer one time step earlier ($x_t^{(\ell)}$) and on the input $\tilde{u}_{t+1}^{(\ell)}$ to the layer. For $\ell = L, L-1, \ldots, 2$, this input $\tilde{u}_{t+1}^{(\ell)} = h_{t+1}^{(\ell-1)}$ is part of the state of the previous layer ($x_{t+1}^{(\ell-1)}$), which in turn depends on the states of that layer one time step earlier ($x_t^{(\ell-1)}$), and on the input $\tilde{u}_{t+1}^{(\ell-1)}$. Eventually, $x_{t+1}^{(\ell)}$ depends on the states of all the previous layers one time step earlier ($x_t^{(\ell-1)}, \ldots, x_t^{(1)}$), which is part of $x(t)$, and on the input $\tilde{u}_{t+1}^{(1)}$ to the first layer, which is $u(t)$.

6. BIAS/VARIANCE TRADE-OFF FOR THE CASCADE FORWARD NET

It is easy to realize that (12) with the ReLU activation function still is a piece-wise linear mapping from x to y: it is formed from linear operations mixed with the max function. The number of parameters in this net is

$$d_L + \sum_{k=1}^{L} d_k \cdot d_{k-1} + d_k \quad (14)$$

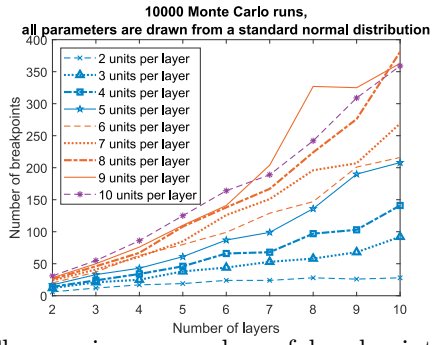


Fig. 1. The maximum number of breakpoints obtained with a feedforward ReLU net for different numbers of layers and units per layer.

If all hidden layers have the same length (d) this amounts to $(L+1)d + (L-1)d^2 + d \cdot \dim x$ parameters. Hence, the number of parameters increases with the number of layers and hidden units per layers in the order of magnitude Ld^2.

What is the *"expressive power"* or flexibility of this net? Since it is a piece-wise linear mapping the flexibility can be measured by how many breakpoints are allowed in the curve from a scalar input x to the scalar y. This has been studied in e.g. Daubechies et al. [2019], Mehrabi et al. [2018], Montúfar et al. [2014].

These papers state that the maximum number of breakpoints scales as d^L. This number is attainable, but for very special choices. [In Daubechies et al. [2019] it is stated that ".. can create roughly d^L breakpoints for very special choices of weights and biases. By choosing to use n parameters in a deep rather than shallow network, one can thus produce functions with many more breakpoints than parameters, albeit these functions have a very special structure."]

To check the flexibility of randomly generated nets, we have simulated and counted breakpoints for deep nets with random matrices. Fig. 1 shows an empirical result on how the maximum number of breakpoints obtained with a feedforward ReLU net expressed in (12) (without skip connections) scales with the number of layers (L) and the number of hidden units per layer (d). For each combination of L and d (both varying from 2 to 10), 10000 Monte Carlo simulations are performed in which the elements in A_i, b_i, and C are drawn from a standard normal distribution. With these random initializations, the maximum number of breakpoints is much lower than d^L.

When plotting these results relative to the number of parameters (see Fig. 2), it is interesting to see that increasing the number of layers is more effective than increasing the number of units per layer. Also, the number of breakpoints per parameter goes well above 0.5 in some cases, so the ReLU implementation is more efficient in generating breakpoints than a standard piece-wise linear function implementation that uses $2n + 2$ parameters for n breakpoints ($2n$ parameters for the x and y coordinates of the breakpoints and 2 parameters for the slopes of the first and the last segment).

So, even if randomly generated nets on average have less flexibility, *the point is that while the maximum number of breakpoints may increase drastically with the depth of*

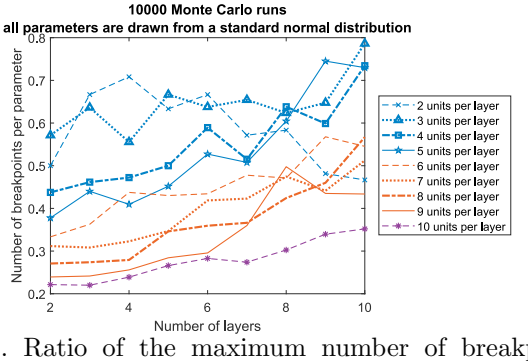


Fig. 2. Ratio of the maximum number of breakpoints obtained with a feedforward ReLU net for different numbers of layers and units per layer to the number of parameters.

the net, the number of parameters may not increase at the same rate in a deep network. Consequently, there is a potential for a better bias/variance trade-off. This of course means that there will be hidden dependencies between the breakpoints. The parameterization is so to speak less generous than the possible model flexibility. This requires the necessity to form "subroutines" in the parameterization to deal with patterns that reoccur in different parts of the function. But this also allows for the possibility to form these automatically as part of the parameter fitting process. This makes it possible to handle the bias/variance trade-off more efficiently.

7. DEEP LEARNING AS A SYSTEM IDENTIFICATION TASK

All this means that deep learning is a task that fits very well into the system identification framework as depicted in the figure in Section 2. What is special is that the choice of model structure $\mathcal{M}$ is done within the family of deep networks. But it is still an (statistical) estimation problem, and all the aspects of validation, bias/variance trade-off discussed in Section 2 are still relevant. An early contribution to bias/variance trade-off in learning problems is given in Geman et al. [1992], and a more recent one is Belkin et al. [2019].

Once the deep learning model structures $\hat{y}(t|\theta)$ are incorporated in the model objects of system identification, the work process becomes very similar.

For example, MATLAB's System identification toolbox, Ljung [2018] allows any `network` object in MATLAB's Deep Learning toolbox to be translated to an `idnlarx` model by the command

```
m=neuralnet(network);
```

`idnlarx` is an object that handles general *Nonlinear ARX* models (1). After that, `m` can be estimated, validated, used for simulation and prediction like any `idnlarx` object. It can also be used in the GUI for estimating and manipulating nonlinear models.

8. EXAMPLE: THE SILVERBOX DATA

A data set that has been often used as a nonlinear benchmark case is the so-called "Silverbox Example". This is a forced Duffing oscillator which is an electronic circuit that mimics a mechanical system with

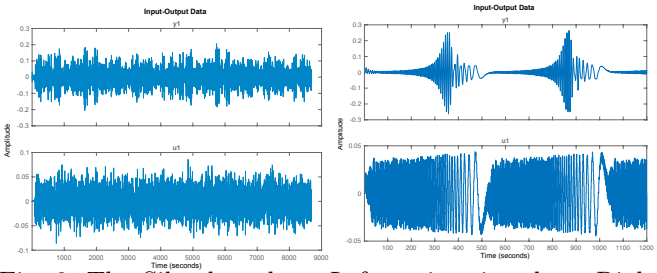


Fig. 3. The Silverbox data. Left: estimation data, Right: validation data

a cubic hardening spring. The experiment is described in Schoukens et al. [2016] and the data is available on www.nonlinearbenchmark.org/#Silverbox. Two sets of data were collected with different excitations, depicted in Fig. 3. Such different data sets for estimation and validation are quite demanding for the model quality. It is not so common in tests, but were used e.g. in Schoukens and Ljung [2019], Fig S21. The estimation data are used to fit a model, and the validity of the model is checked by how well it reproduces the validation data. That is, the model is simulated using the validation data input and the discrepancy between model output and measured validation data output is determined. This is a standard system identification task.

To load the estimation and validation data the following MATLAB sequence can be used

```
load SNLS80mV.mat
fs=1e7/2^14;
edat=iddata((V2(30550:38500)',...
        V1(30550:38500)',1/fs);
load Schroeder80mV
ySchroeder=V2(10585:10585+1023);
uSchroeder=V1(10585:10585+1023);
vdat=iddata(ySchroeder',uSchroeder',1/fs);
```

8.1 Linear Model

A common linear model of the Box-Jenkins type (see e.g. Chapter 4 in Ljung [1999]) with 4 poles, 4 zeros and a second order noise transfer function, is estimated by

```
mbj=bj(edat,[4 4 2 2 0]);
```

The accuracy of this model is computed by

```
[ys,Fit]=compare(vdat,mbj)
Fit=29.71 % (percent of the output variation ...
            reproduced by the model)
```

The validation output and the error between this and the simulated model output is shown in Fig. 4. It can be seen that the simulation is not very good. The error is at times as big as the signal itself. The residual analysis for this model is obtained by

```
resid(vdata,mbj)
```

and is shown in Fig. 5. There is quite significant correlation left in the residuals, suggesting that the model should be improved by a more flexible model structure.

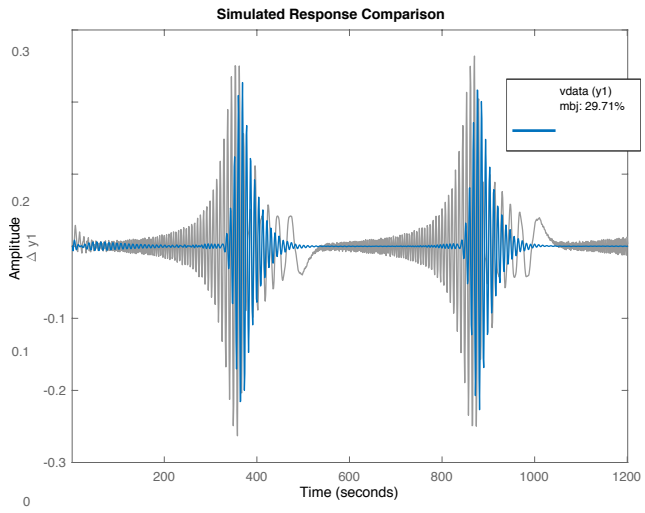


Fig. 4. The validation output (grey) and the error between model and simulated output (blue) for the Box-Jenkins model. The fit is **29.7** %

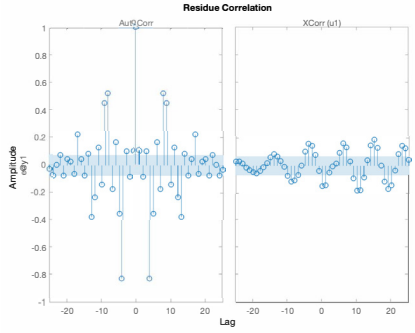


Fig. 5. Residual analysis for the validation data and the Box-Jenkins model. Left: the autocorrelations for the residuals. Right: Cross correlation between input and residuals

8.2 Deep Cascaded Network

A deep learning model (12) with 6 layers of cascaded feedforward nets with 6 nodes each is estimated and validated by

```
net=cascadeforwardnet([6,6,6,6,6,6]);
N2=neuralnet(net);
mN2=nlarx(edat,[4 4 0],N2);
[Ys,Fit]=compare(vdat,mN2)
Fit=99.18
```

The validation output and the error between this and the simulated model output is shown in Fig. 6. The simulation is remarkably good: the error is hardly visible. Recall that the simulation is done without any access to the measured validation output, and note that the character of the validation data is quite different from that of the estimation data.

The residual analysis (`resid(vdata,mN2)`) for this model is shown in Fig. 7. The correlation between the residuals and past inputs (lag larger than 0) is quite insignificant. There is some correlation left among the residuals themselves, but no attempt has been made to build a model for the color of the additive disturbances. Anyway the size of the residuals is very small.

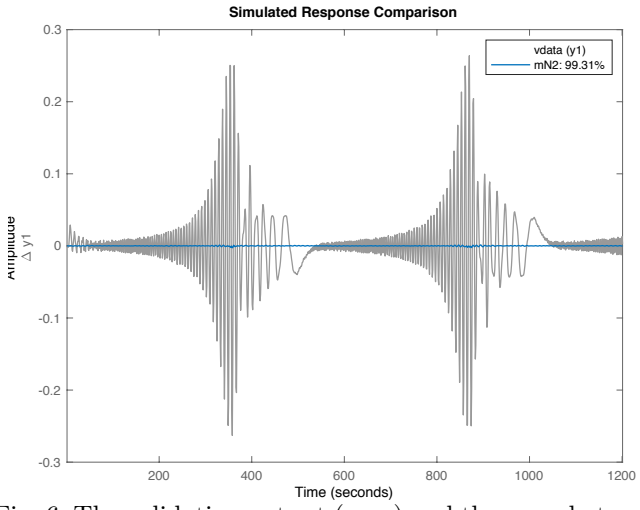


Fig. 6. The validation output (grey) and the error between model and simulated output (blue) for the deep cascadeforwardnet model. The fit is **99.2** %

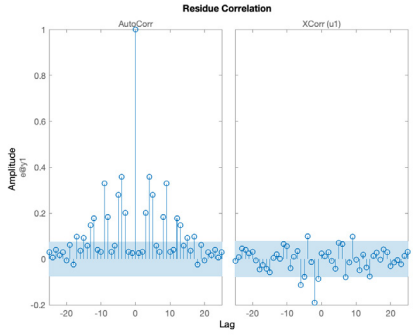


Fig. 7. Residual analysis for the validation data and the deep net model. Left: the autocorrelations for the residuals. Right: Cross correlation between input and residuals

8.3 LSTM Network

An LSTM network with 8 hidden units can be trained in MATLAB with the deep learning toolbox. In the simplest case, this network can be implemented as,

```
numHiddenUnits=8;
featureDimension=1; % u(t)
layers=[ ...
  sequenceInputLayer(featureDimension), ...
  lstmLayer(numHiddenUnits, ...
            'OutputMode','sequence'), ...
  fullyConnectedLayer(1), ...
  regressionLayer];
options=trainingOptions('adam', ...
  'MaxEpochs',1000, ...
  'InitialLearnRate',0.01, ...
  'Plots','training-progress');
net=trainNetwork(XTrain,YTrain,layers,options);
Fit=88.9
```

In this case we transform the data as defined in the previous section to a format for the deep learning model

```
XTrain=cell(1,1);
YTrain=cell(1,1);
XTrain{1}=edat.u';
```

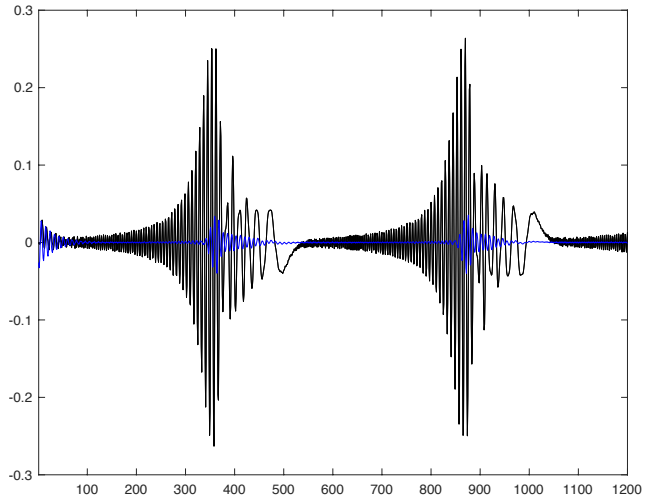


Fig. 8. The validation output (grey) and the error between model and simulated output (blue) for the LSTM model. The fit is **88.9 %**

```
YTrain{1}=edat.y';
```

The validation output, in Fig. 8, shows that the LSTM model performs worse when the input data has higher amplitude than seen in the training data. This is related to the internal LSTM body that is built up of sigmoid functions that saturate when extrapolating. One should note that the LSTM model is not tailored for this type of problem but rather for problems with significantly longer autocorrelations.

A deep LSTM network is obtained simply by repeating the line `lstmLayer(numHiddenUnits,... 'OutputMode',... 'sequence'), ...` a few times in the definition of `layers`. The number of hidden units may differ in the different layers. For the tested data, no improvement was found for deeper LSTM models.

9. CONCLUSIONS

The use of deep learning – in particular for estimating dynamical systems – has strong links to system identification practice. A few things have been stressed in this contribution:

- Classical statistical tools such as cross validation, bias/variance trade-off, residual analysis are essential also for deep learning
- The workflow in deep learning and system identification has many similarities
- The power of deep nets also for standard system identification has been demonstrated.
  The `cascadeforwardnet` obtains models for the standard Silverbox benchmark case that easily match the best results obtained with other sophisticated solutions, cf. Schoukens and Ljung [2019], sidebar "The forced Duffing Oscillator".

REFERENCES

C. Andersson, A. H. Ribeiro, K. Tiels, N. Wahlström, and T. B. Schön. Deep convolutional networks in system identification. In *Proceedings of the IEEE 58th IEEE Conference on Decision and Control (CDC)*, Nice, France, 2019.

M. Belkin, D. Hsu, S. Ma, and S. Mandal. Reconciling modern machine-learning practice and classical bias-variance trade-off. *Proc. National Academy of Sciences, PNAS*, 116(32):15849–15854, august 2019.

I. Daubechies, R. DeVore, S. Foucart, B. Hanin, and G. Petrova. Nonlinear approximation and (deep) ReLU networks. *ArXiv*, (1905.02199), 2019.

S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, January 1992.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.

S. Hochtreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems NIPS*, pages 1097–1105, 2012.

S. Kung. *Digital Neural Networks*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

L. Ljung. *System identification, Theory for the user*. System sciences series. Prentice Hall, Upper Saddle River, NJ, USA, second edition, 1999.

L. Ljung. *The System Identification Toolbox: The Manual*. The MathWorks Inc. 1st edition 1986, 9th edition 2018, Natick, MA, USA, 2018.

M. Mehrabi, A. Tchamkerten, and M. I. Yousefi. Bounds on the approximation power of feedforward neural networks. *ArXiv*, (1806.11416), 2018.

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. Technical report, arXiv:1301.3781, 2013.

G. Montúfar, R. Pascanu, K. Chu, and Y. Bengio. On the number of linear regions of deep neural networks. *ArXiv*, (1402.1869), 2014.

J.-P. Noël and J.Schoukens. Grey-box state-space identification of nonlinear mechanical vibrations. *Int. J. Control*, 91:1118–1139, 2017.

C. Olah. Understanding LSTM networks – colah's blog. *https://colah.github.io/posts/2015-08-understanding-lstms/*, 2015.

R. Pascanu, C Gulcehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. In *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*, 2014.

J. Schoukens and L. Ljung. Nonlinear system identification – a user-oriented roadmap. *IEEE Control Systems Magazine*, 39(6):28 –99, December 2019.

J. Schoukens, M. Vaes, and R. Pintelon. Linear system identification in a nonlinear setting: Nonparametric analysis of the nonlinear distortion and their impact on the best linear approximation. *IEEE Control Systems Magazine*, 36:38–69, 2016.

J. Sjöberg and L. Ljung. Overtraining, regularization and searching for minimum with application to neural nets. *Int. Journal of Control*, 62(6):1391–1407, Dec 1995.

G. Strang. The functions of deep learning. https://sinews.siam.org/Details-Page/the-functions-of-deep-learning, 2018.

L. A. Zadeh. On the identification problem. *IRE Transactions on Circuit Theory*, 3:277–281, 1956.