Surrogate and Hybrid Models for Control

Bernt Lie

University of South-Eastern Norway, Porsgrunn, Norway, Bernt.Lie@usn.no

Abstract

With access to fast computers and efficient machine learning tools, it is of interest to use machine learning to develop surrogate models from complex physics-based models. Next, a hybrid model is a combination model where a data driven model is built to describe the difference between an imperfect physics-based/surrogate model and experimental data. Availability of Big Data makes it possible to gradually improve on a hybrid model as more data become available. In this paper, an overview is given of relevant ideas from model approximation/data driven models for dynamic systems, and machine learning via artificial neural networks. To illustrate how the ideas can be implemented in practice, a simple introduction to package Flux for language Julia is given. Several types of surrogate models are developed for a simple, illustrative system. Finally, the development of a hybrid model is illustrated. Emphasis is put on ideas related to Digital Twins for control.

Keywords: Digital Twin, surrogate models, hybrid models, dynamic systems, control

1 Introduction

Digital Twin (El Saddik, 2018) is a key concept in Industrie 4.0 (Hermann et al., 2016), and is related to Internet of Things and Big Data. Here, a Digital Twin involves static and dynamic models for various uses such as analysis, optimization, control design, operator training, etc.; a modeling languages such as Modelica (Fritzson, 2015) is suitable for describing such systems. Detailed models may be too complex for optimization/control, and need to be simplified, (Benner et al., 2017). A *surrogate model*[1] is a simplified model that allows for fast solution, derived from a more complex model. Surrogate models can be developed directly from physics, e.g., using weighted residual methods (Finlayson, 2014), or by fitting a mathematical structure to simulation data, (Luo and Chen, 2018), (Cueto et al., 2016), (Bishop, 2011), (Murphy, 2012).

A mechanistic/surrogate model allows for analysis prior to building the system, while a data driven model uses data from the real systems for training. A *hybrid* model, in this context, is an imperfect mechanistic based model where the deviation between model prediction and experimental data is described by a data driven model, (Farrell and Polycarpou, 2006). A hybrid model allows for analysis prior to building the system, with gradual improvement of the model as more data becomes available.

Machine learning (ML) for control involves dynamics, and is more demanding than ML for static systems, (Farrell and Polycarpou, 2006). Today, a multitude of tools for machine learning are available, e.g., TensorFlow[2] with interfaces to many scientific computing tools. Languages MATLAB[3], Python[4], Julia[5] (Bezanson et al., 2017), etc., have their own ML packages, e.g., Flux for Julia (Innes, 2018). Tools for solving mechanistic models include OpenModelica[6], the DifferentialEquations package for Julia (Rackauckas and Nie, 2017), etc. OMJulia allows for integrating Modelica code with Julia, (Lie et al., 2019). For data driven models, regularization and validation is important (Boyd and Vandenberghe, 2018).

This paper gives an overview of traditional data driven modeling principles, with relations to ML ideas, and give examples of how to use a free ML package (Flux for Julia). Surrogate and hybrid models are illustrated through an academic example. This way, core ideas of machine learning for surrogate and hybrid models are introduced with practical demonstrations of how to use simple, free tools.

In Section 2, an overview of classical data driven modeling principles is given. In Section 3, artificial neural networks are related to the classical principles. In Section 4, the ideas of surrogate models and hybrid models are illustrated on a concrete example. A brief discussion with conclusions is given in Section 5. Practicalities such as validation, pre-processing of data, etc., are very important for data driven models; due to space limitations, these topics are not discussed thoroughly here.

2 Data Driven Modeling Principles

2.1 Data and Input-output Mapping

We consider a mapping from an input vector (input node) $x \in \mathscr{X} \subseteq \mathbb{R}^{n_x}$ to an output vector (node) $y \in \mathscr{Y} \subseteq \mathbb{R}^{n_y}$. With N_d available data pairs (x_i, y_i) $x_i \in \mathscr{X}$ and $y_i \in \mathscr{Y}$, we organize these data in two matrices $X_\mathrm{d} \in \mathbb{R}^{n_x \times N_\mathrm{d}}$ and $Y_\mathrm{d} \in \mathbb{R}^{n_y \times N_\mathrm{d}}$. We then seek a mapping $y = f(x)$ from these data.

The input data vector x is often extended into a *feature* vector $\phi(x) \in \mathbb{R}^{n_\phi}$; $\phi(x)$ is also known as a *regressor* vec-

[1]Surrogate: from Latin *subrogatus* = substitute

[2]https://www.tensorflow.org/

[3]https://www.mathworks.com/

[4]https://www.anaconda.com/

[5]https://julialang.org/

[6]www.openmodelica.org

tor. A common strategy is to postulate that $f(x)$ can be expressed as a linear combination of the feature vector,

$$y = f(x) = \theta\phi(x) + e, \tag{1}$$

where e is some error. Here, $\theta \in \mathbb{R}^{n_y \times n_\phi}$ is a matrix; in other models, θ may be a vector or an ordered collection of objects; the shape of θ will be clear from the context. A predictor $\hat{y}$ for y is $\hat{y} = \theta\phi(x)$, where it is assumed that e does not contain valuable information.

Some examples of feature vectors: (i) the identity function, $\phi_i(x) = x_i$ with $n_\phi = n_x$, (ii) polynomials, for $n_x = 1$: $\phi_i(x) = x^{i-1}$, (iii) Fourier series, with $n_x = 1$: $\phi_1(x) = 1$, $i > 1 : \phi_{2i}(x) = \sin\left(\frac{2\pi x}{i}\right)$, $\phi_{2i+1} = \cos\left(\frac{2\pi x}{i}\right)$, (iv) splines: zero order splines consist of a sequence of non-overlapping, adjacent, single pulses. Higher order splines are found recursively by integration.

With the given data set X_d, we form $\Phi_\mathrm{d} \triangleq \Phi(X_\mathrm{d}) \in \mathbb{R}^{n_\phi \times N_\mathrm{d}}$. Combined with 1, this leads to

$$Y_\mathrm{d} = \theta\Phi_\mathrm{d} + E_\mathrm{d}. \tag{2}$$

A common strategy is to find an estimate $\hat{\theta}$ such that the squared errors $E_\mathrm{d} = Y_\mathrm{d} - \theta\Phi_\mathrm{d}$ are minimized, e.g., that *loss*[7] function $\mathscr{L}_1$ is minimized:

$$\hat{\theta} = \arg\min_{\theta} \mathscr{L}_1 \tag{3}$$

$$\text{where} : \mathscr{L}_1 = \|E_\mathrm{d}\|_\mathrm{F}^2 \tag{4}$$

and $\|\cdot\|_\mathrm{F}$ is the Frobenius norm. Then $\hat{E}_\mathrm{d}$ is orthogonal to regressor Φ_d,

$$\hat{E}_\mathrm{d} \perp \Phi_\mathrm{d} \Leftrightarrow \hat{E}_\mathrm{d}\Phi_\mathrm{d}^T = 0 \Leftrightarrow \left(Y_\mathrm{d} - \hat{\theta}\Phi_\mathrm{d}\right)\Phi_\mathrm{d}^T = 0 \tag{5}$$

which is known as the *normal equation*.

The normal equation is linear in the unknown $\hat{\theta}$, and can easily be solved using linear algebra solvers. Computer tools such as MATLAB, Julia, etc. find $\hat{\theta}$ by using the *slash operator*, $\hat{\theta} = Y_\mathrm{d}/\Phi_\mathrm{d}$. The slash operator applies the Moore-Penrose pseudo inverse of Φ_d if Φ_d has a nontrivial nullspace.

2.2 Case study: B-spline regressors

Consider the random data set $(X_\mathrm{d}, Y_\mathrm{d})$ of $N_\mathrm{d} = 10$ points in the upper panel of Figure 1, with corresponding zero order and first order splines.

The kth order mappings $f_k(x)$ are given as $f_k(x) = \sum_{j=1}^{n_\phi} y_{\mathrm{d},j} \cdot \phi_j^{(k)}(x)$ where $\phi_j^{(k)}$ is the $k$thorder spline at position x_j. Interpolation is achieved by setting $n_\phi = N_\mathrm{d}$. Regularly spaced N_d data points X_d corresponding to the center of the splines results in interpolants as in the lower panel of Figure 1. With noisy data, a least squares solution with $n_\phi < N_\mathrm{d}$ is used instead of interpolation.

[7] *Loss* of model accuracy; alternatively denoted *cost*.

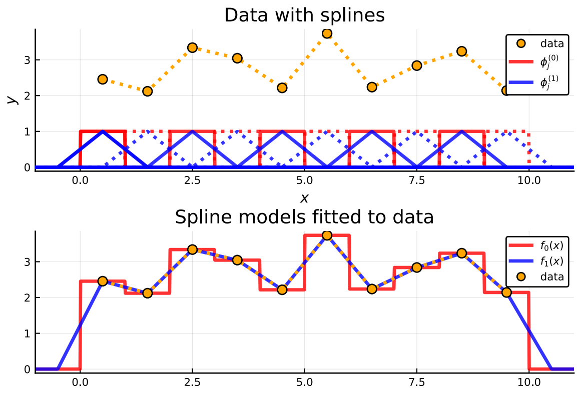

Figure 1. *Upper panel*: Data set X_d vs. Y_d (orange, circle + dotted line), with chosen set of zero order splines $\phi_j^{(0)}(x)$ (alternating red, solid and dotted square pulses) and first order splines $\phi_j^{(1)}(x)$ (alternating blue, solid and dotted saw tooth shapes). *Lower panel*: Data set X_d vs. Y_d (orange, circle + dotted line), fitted interpolation functions $f_0(x)$ (red line) and $f_1(x)$ (blue line).

Remark 1. Observe that the $n_\phi = 10$ zero order splines $\phi_j^{(0)}(x)$, $j \in \{1, \ldots, 10\}$ given by the square pulses in Figure 1, one for each $N_\mathrm{d} = 10$ of the data points in $(X_\mathrm{d}, Y_\mathrm{d})$, can be expressed by $N_\mathrm{d} + 1$ Heaviside functions (unit step functions) $\mathbb{H}_\xi(x)$ as

$$\phi_j^{(0)}(x) = \mathbb{H}_{x_{\mathrm{d},j} - \frac{1}{2}}(x) - \mathbb{H}_{x_{\mathrm{d},j} + \frac{1}{2}}(x)$$

where

$$\mathbb{H}_\xi(x) = \begin{cases} 0, & x < \xi \\ 1, & x \geq \xi \end{cases}.$$

The implication of this is that any scalar data set can be expressed by N_d zero order splines, and can equally be expressed by $N_\mathrm{d} + 1$ Heaviside (unit step) functions. The idea generalizes to $n_x > 1$ and $n_y > 1$. ▲

2.3 Regularization

Depending on the choice of regressors $\phi_j(x)$, the resulting regressor matrix Φ_d may be well conditioned or ill-conditioned. Choosing a high order polynomial mapping often leads to an ill-conditioned problem. Ill-conditioned problems can be handled by introducing additional information; this is known as *regularization*. In regularization, additional *hyper parameters* are introduced, and selected by the user through *validation*

A common method to regularize the problem is by Principal Component Regression (PCR), where the hyper parameter is the "practical rank" r. A related approach is Partial Least Squares (PLS).

Another type of regularization method is based on *multi objective optimization*, (Boyd and Vandenberghe, 2018), with primary loss function $\mathscr{L}_1$ from Equation 4, and a secondary loss $\mathscr{L}_2$. Examples of secondary loss functions are: (i) $\mathscr{L}_2 = \|\theta\|_\mathrm{F}^2$ (Tikhonov/ridge regularization, weight decay), (ii) $\mathscr{L}_2 = \|\theta - \theta^{\circ}\|_{\mathrm{F}}^2$ where θ° is some prior estimate (similar to Bayes statistics), (iii) various measures of the model solution such as $\mathscr{L}_2 = \|\hat{Y} - \tilde{Y}\|_{\mathrm{F}}^2$ (closeness to a pre-specified solution $\tilde{y}$), constraint in model slope $\mathscr{L}_2 = \left\|\frac{\partial y}{\partial x}\right\|_{\mathrm{F}}^2$, model curvature $\mathscr{L}_2 = \left\|\frac{\partial^2 y}{\partial x^2}\right\|_{\mathrm{F}}^2$, etc. We then choose some positive weight λ (hyper parameter), and create a modified loss function

$$\mathscr{L} = \mathscr{L}_1 + \lambda \mathscr{L}_2, \tag{6}$$

where $\mathscr{L}$ is minimized for a number of hyper parameter values λ. Finally, λ is chosen via validation.

We could also add constraints on the model as part of the regularization, e.g., require that $\theta \subseteq \Theta$, $\hat{Y} \subseteq \mathscr{Y}$, etc.

2.4 Validation and generalization

It is common to fit/train multiple models from data, based on a series of user selectable hyper parameters, Section 2.3. Additional hyper parameters could be model order, number of iterations in model fitting, etc. Choosing hyper parameters is an optimization problem in itself; the various hyper parameters are assessed by comparing their primary loss function from training data $(X_{\mathrm{t}}, Y_{\mathrm{t}})$ vs. from fresh, validation data $(X_{\mathrm{v}}, Y_{\mathrm{v}})$, (Boyd and Vandenberghe, 2018). When the model has been validated and hyper parameters have been chosen, a final *grading* of how well the model generalizes is made from other fresh, grading data $(X_{\mathrm{g}}, Y_{\mathrm{g}})$.[8]

For numeric reasons, all data sets should be preprocessed by removing their mean. Furthermore, the data should be scaled to have either

1. unit standard deviation (standardized data), or

2. variation in the region $[-1,1]$ or $[0,1]$ (normalized data).

Missing data and outliers should be removed, replaced by interpolation, etc.

2.5 Nonlinear mappings

As a slight generalization of the linear mapping with basis functions, we can define predictors

$$\hat{y} = \theta \Phi(x; \omega)$$

where θ is a linear model parameter and ω is a nonlinear model parameter. Examples could be spline functions with parametric location and width, Gaussian functions with parameterized mean value (location) and standard deviation (width), etc. Mappings that are nonlinear in the parameters require iteration to find the parameters minimizing the loss function, and may exhibit multiple (local) minima.

[8]"Grading" data are sometimes denoted "test" data.

2.6 Time series modeling

Continuous time: Assume that the data are generated from a *continuous* time model, say,

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,u)$$

and that we know $\left(\frac{\mathrm{d}x}{\mathrm{d}t}, x, u\right)$ at a number N_{d} of data points. We can then arrange the data into data structures

$$X_{\mathrm{d}} = \begin{pmatrix} x_1 & x_2 & \cdots & x_{N_{\mathrm{d}}} \\ u_1 & u_2 & \cdots & u_{N_{\mathrm{d}}} \end{pmatrix}$$
$$Y_{\mathrm{d}} = \begin{pmatrix} \left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_1 & \left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_2 & \cdots & \left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_{N_{\mathrm{d}}} \end{pmatrix},$$

and fit a model to the data set. For experimental data, this may not be a realistic approach since noise makes it hard to find $\left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_j$. With data generated from a simulation model, this approach may work: differential algebraic equation (DAE) solvers often make $\left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_j$ available.

Discrete time: A common approach to time series modeling is to postulate a *discrete* time model — either an input-output model of form

$$y_t = f\left(y_{t-1}, \ldots, y_{t-n_y}, u_t, \ldots, u_{t-n_u}; \theta\right) \tag{7}$$

or a state space model of form

$$\begin{aligned} x_t &= f(x_{t-1}, u_{t-1}; \theta) \\ y_t &= g(x_t, u_t; \theta). \end{aligned}$$

For a state space model, the initial value x_1 is unknown and must be included in model parameter θ.

Predictors and data structures: The loss function $\mathscr{L}$ for data fitting can either be based on minimizing the one step-ahead error $\sum_i \left(y_i - \hat{y}_{i|i}\right)^2$ (prediction error, PE) or multi step-ahead error $\sum_i \left(y_i - \hat{y}_{i|1}\right)^2$ (output error/shooting error, SE). The resulting PE predictor has form

$$\hat{y}_{t|t-1} = f_{\mathrm{PE}}\left(y_{t-1}, \cdots, y_{t-n_y}, u_t, \cdots, u_{t-n_u}; \hat{\theta}_{\mathrm{PE}}\right),$$

while the resulting SE predictor has form

$$\hat{y}_{t|1} = f_{\mathrm{SE}}\left(\hat{y}_{t-1|1}, \cdots, \hat{y}_{t-n_{y|1}}, u_t, \cdots, u_{t-n_u}; \hat{\theta}_{\mathrm{SE}}\right).$$

Normally, $\hat{\theta}_{\mathrm{PE}} \neq \hat{\theta}_{\mathrm{SE}}$, and $f_{\mathrm{PE}}(\cdot) \neq f_{\mathrm{SE}}(\cdot)$. Here, it should be observed:

1. The PE predictor constitutes a static mapping from known data to the one step-ahead prediction. As such, the predictor is guaranteed to be stable. The SE predictor, on the other hand, constitutes a dynamic/difference equation predictor which may or may not be stable.

2. It is simpler to find the PE estimate $\hat{\theta}_{\text{PE}}$ than the SE estimate $\hat{\theta}_{\text{SE}}$ because (i) the PE problem has a simpler loss function/data structure, and (ii) the PE problem is relatively linear with few (local) minima while the SE problem may be highly nonlinear with multiple (local) minima.

3. Because the PE predictor is one step-ahead, it is not suitable for, e.g., Model Predictive Control where long-term predictions are used.

4. Creating a dynamic predictor suitable for long-term prediction from $\hat{y}_{t|1} = f_{\text{PE}}\left(\hat{y}_{t-1|1}, \cdots, \hat{y}_{t-n_y|1}, u_t, \cdots, u_{t-n_u}; \hat{\theta}_{\text{PE}}\right)$ is not guaranteed to work, and may fail miserably.

For the model in 7 (assume $n_u = n_y$), the data are arranged as

$$X_{\text{d}} = \begin{pmatrix} y_{n_y-1} & y_{n_y} & \cdots & y_{N_{\text{d}}-1} \\ y_{n_y-2} & y_{n_y-1} & \cdots & y_{N_{\text{d}}-2} \\ \vdots & \vdots & \ddots & \vdots \\ y_1 & y_2 & \cdots & y_{N_{\text{d}}-n_y} \\ u_{n_y} & u_{n_y+1} & \cdots & u_{N_{\text{d}}} \\ \vdots & \vdots & \ddots & \vdots \\ u_1 & u_2 & \cdots & u_{N_{\text{d}}-n_y} \end{pmatrix}$$
$$Y_{\text{d}} = \begin{pmatrix} y_{n_y} & y_{n_y+1} & \cdots & y_{N_{\text{d}}} \end{pmatrix}.$$

are suitable for fitting a PE model.

A state space model use data arranged as

$$X_{\text{d}} = \begin{pmatrix} u_1 & u_2 & \cdots & u_{N_{\text{d}}} \end{pmatrix}$$
$$Y_{\text{d}} = \begin{pmatrix} y_1 & y_d & \cdots & y_{N_{\text{d}}} \end{pmatrix},$$

and will by the very nature of an unknown initial state $x_1 \in \theta$ lead to a shooting error estimator.

Model fitting based on one step-ahead prediction error allows for random permutation of the columns in $(X_{\text{d}}, Y_{\text{d}})$. Model fitting based on the shooting error, however, requires that the data are kept in the correct order.

3 Artificial Neural Networks

3.1 Neural Networks

Artificial neural networks (ANNs) were originally inspired by some hypotheses on how the brain works, but have been generalized into mathematical structures which have direct resemblance with approximation theory. Essentially, ANNs attempt to describe arbitrary mappings from input vectors $x \in \mathbb{R}^{n_x}$ to output vectors $y \in \mathbb{R}^{n_y}$,

$$y = f(x; \theta) + e$$

where θ is some model parameter and e is some model error. In ANNs, this mapping consists of chaining a set of parameterized layer mappings and then tuning the parameters to achieve good description of available data. Here, we will consider the classical Feed forward Neural Network (FNN).

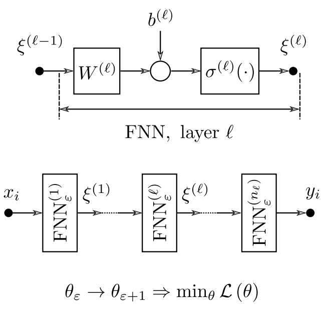
$b^{(\ell)}$
$\xi^{(\ell-1)}$
$W^{(\ell)}$
$\sigma^{(\ell)}(\cdot)$
$\xi^{(\ell)}$
FNN, layer ℓ
x_i
$\text{FNN}_\varepsilon^{(1)}$
$\xi^{(1)}$
$\text{FNN}_\varepsilon^{(\ell)}$
$\xi^{(\ell)}$
$\text{FNN}_\varepsilon^{(n_\ell)}$
y_i
$\theta_\varepsilon \to \theta_{\varepsilon+1} \Rightarrow \min_\theta \mathcal{L}(\theta)$

Figure 2. Structure of layer in Feed forward Neural Network (FNN; upper diagram), chained FNN layers (lower diagram), with update of parameter estimate $\theta_\varepsilon = \left(\ldots, W^{(\ell)}, b^{(\ell)}, \ldots\right)$ at *epoch* ε which reduces the loss function $\mathscr{L}(\theta_\varepsilon)$.

3.2 Feed forward Neural Network

A Feed forward Neural Net (FNN) is composed of layer mappings from input x to output y. The layers have the structure as in the upper diagram of Figure 2.

For an FNN of n_ℓ layers, layer ℓ has form

$$\xi^{(\ell)} = \sigma^{(\ell)}\left(W^{(\ell)}\xi^{(\ell-1)} + b^{(\ell)}\right)$$

with *weight* matrix $W^{(\ell)} \in \mathbb{R}^{n_\xi^{(\ell-1)} \times n_\xi^{(\ell)}}$, *bias* vector $b^{(\ell)} \in \mathbb{R}^{n_\xi^{(\ell)}}$, and *activation* function $\sigma(\cdot)$ mapped on each individual element in the vector argument. The boundary conditions are

$$\xi^{(0)} \equiv x$$
$$y \equiv \xi^{(n_\ell)},$$

while $\xi^{(\ell)} \in \mathbb{R}^{n_\xi^{(\ell)}}$ is denoted a *hidden* variable for $\ell \in \{1, \ldots, n_\ell - 1\}$. The number of hidden variables (nodes) $n_\xi^{(\ell)}$ may vary from layer to layer, and is a design choice.

The (non-)linear *activation* function $\sigma^{(\ell)}(\cdot)$ may differ from layer to layer; often a (strictly) increasing, sigmoid shaped activation function is used. Typical choices of the activation function are given in Table 1.

For the first layer,

$$\xi^{(1)} = \sigma^{(1)}\left(W^{(1)}x + b^{(1)}\right).$$

The classical activation function was the *binary step* function, which is identical to a *Heaviside* function. By also shifting the function argument via biases, we can create a sequence of $n_\xi^{(1)}$ equally spaced Heaviside functions. Combining this with a second, identity activation output layer, this allows us to create $n_\xi^{(1)} - 1$ zero order splines,

Remark 1. With arbitrary large $n_\xi^{(1)}$, we can then create arbitrarily many zero order splines. In summary, a FNN is strongly related to classical spline regression.

By changing the activation function of the hidden layer to another, sigmoid shaped function, this is akin to the use

Table 1. Common ANN activation functions $\sigma(x)$.

Name	Function $\sigma(x)$	Julia name
Identity	x	`identity`
Binary step, $\mathbb{H}_0(x)$	$\begin{cases} 0, & x<0 \\ 1, & x\geq 0 \end{cases}$	
Logistic/sigmoid, $\sigma(x)$	$\frac{1}{1+\exp(-x)}$	`sigma`
Hyperbolic tangent	$\tanh(x)$	`tanh`
Rectified Linear Unit function	$\max(0,x)$	`relu`

of higher order splines, or other basis functions with local or exponential support. Because of this, FNNs for regression problems often contain a single hidden layer, and an output layer with identity activation.

3.3 Measures of model fit

In regression problems, the tuning parameter θ of the network,

$$\theta = \left\{W^{(1)}, b^{(1)}, \ldots, W^{(n_\ell)}, b^{(n_\ell)}\right\},$$

is sought such that it minimizes some loss function measuring the deviation between observations $y \in Y_\mathrm{d}$ and predictions $\hat{y} = f(x)$ where $x \in X_\mathrm{d}$. The chosen loss function is typically the *Residual Sum of Squares* (RSS),

$$\mathscr{L}_\mathrm{RSS} \triangleq \sum_{i=1}^{N_\mathrm{d}} \|y_i - \hat{y}_i\|_2^2,$$

or the *Mean Squared Error* (MSE) $\mathscr{L}_\mathrm{MSE} \triangleq \frac{1}{N_\mathrm{d}}\mathscr{L}_\mathrm{RSS}$. Sometimes, the *Root Mean Squared Error* (RMSE) $\mathscr{L}_\mathrm{RMSE} \triangleq \sqrt{\mathscr{L}_\mathrm{MSE}}$ is used. All three loss functions have identical minimizing parameter θ. Sometimes, regularization is used, Section 2.3.

3.4 Training of the Neural Network

Training of a network implies tuning parameters θ such that loss function $\mathscr{L}$ is minimized. Standard methods for doing this is using some gradient or Newton method. Gradient methods do iterations in the gradient direction $\frac{\partial \mathscr{L}}{\partial \theta}$,

$$\Delta\theta_\varepsilon = \theta_{\varepsilon+1} - \theta_\varepsilon = -\eta \left.\frac{\partial \mathscr{L}}{\partial \theta}\right|_\varepsilon; \quad (8)$$

η is the "learning rate", typically $\eta = 0.01$, and ε is the major iteration number known as the *epoch*. Alternatively, η can be found via line search.

The chaining of several layers increases the nonlinear relationship in $\mathscr{L}(\theta)$, leading to local minima. Computation of the gradient involves the recursive *back propagation* algorithm to propagate gradients through the layers. Modern tools often use Automatic Differentiation to compute gradients for each layer.

For "Big data" that do not fit into local memory, calculating the full gradient is challenging. One solution is to sample a random subset from the data, compute the "stochastic" gradient from this smaller data set, and do many stochastic gradient descents. This tends to give similar/better results than using the full data set gradient.

Each major iteration ε in the parameter tuning is known as an *epoch*; an epoch consists of a number of minor iterations such as back-propagation updating, line searches, stochastic gradient descent steps, etc.

Newton methods are known to be more efficient than gradient methods. Because neural networks may have a large number of parameters, computation and storing of the Hessian $\frac{\partial^2 \mathscr{L}}{\partial \theta^2}$ is very demanding. Some algorithms use various types of quasi Newton methods in combination with line search, (Bishop, 1994). Still, the most common tuning algorithms are based on gradient descent. Table 2 gives an overview of some gradient descent based methods.

3.5 FNN in Julia Flux

Package Flux (Innes, 2018) in the modern computer language Julia[9] provides easy access to machine learning algorithms, and combines well with the OMJulia interface between OpenModelica and Julia. After importing Flux by `julia> using Flux`, an FNN layer is created by command `julia> L = Dense(n_lm1,n_l,sigma )` where `L` is our chosen name of the layer, while `n_lm1` $= n_\xi^{(\ell-1)}$, `n_l`$= n_\xi^{(\ell)}$, and the activation function is `sigma` $= \sigma$. If input argument `sigma` is skipped, the activation function defaults to identity mapping. The values of weight matrix W and bias vector b can be inspected by attributes `julia> L.W.data` and `julia> L.b.data`. In general, typing the object name (`L`) followed by period, `L.`, and then hitting the tabulator key gives a pop-up menu with possible attributes.

By default, instantiating the layer with command `Dense(n_lm1,n_l,sigma)` populates matrix W and bias vector b using a type `Float32` random number generator — this choice is made to ease optional training of the neural network on GPUs[10]. Other data types can be specified at *instantiation* of the layer.

Layers can be *chained* together with command `mod = Chain(L1,L2,...)`. To "unchain" the model into separate layers, command `mod.layers` results in a tuple of the layers, thus `mod.layers[2].W.data` produces the W matrix of layer `L2`.

[9]`www.julialang.org`
[10]GPU = graphical processing unit

Table 2. Common gradient descent methods in ANN training.

Name	Julia name
Classical gradient descent, learning rate η	`Descent(eta)`
Momentum gradient descent, learning rate η, momentum ρ	`Momentum(params,eta=0.01;rho=0.9)`
Nesterov gradient descent, learning rate η, Nesterov momentum ρ	`Nesterov(eta;rho=0.9)`
Stochastic gradient descent, learning rate η, moment estimate exponential decay rates β	`ADAM(eta=0.001,beta=(0.9,0.999))`

Assuming we have created a Feed forward Neural Network *model* `mod` by chaining several FNN layers, we need to specify the *loss* function `loss`, define the *parameters* `par` to tune, specify the *optimization* algorithm `opt`, and prepare the *data* set `data` consisting of $X_d \in \mathbb{R}^{n_x \times N_d}$ and $Y_d \in \mathbb{R}^{n_y \times N_d}$.

```
loss(x, y) = mean((mod(x).-y).^2)       #
    Statistics package
par = params(mod)                 # Flux
    command
opt = ADAM(eta=0.001,beta=(0.9,0.999)) # See
    Table 2
data = [(Xd,Yd)]
```

We are now ready to *train* the network, which implies taking a major iteration step to adjust `par` such that the value of `loss` is reduced, e.g., 8. Command `julia> train!(loss,par,data,opt)` carries out one training epoch, where exclamation mark `!` in `train!` indicates that the model parameters are changed *in place*. We can thus train the network over `nE` epochs with a **for** loop:

```
nE = 1_000 # number of epochs
for i in 1:nE
   train!(loss,par,data,opt)
end
```

(Bishop, 1994) tests how well an FNN with 1 hidden layer of 5 elements/nodes, $\sigma = \tanh(x)$, and identity output can describe 4 different scalar functions. Using Julia Flux and 3000 epochs, this can be done as follows (for function $y = x^2$):

```
# Generating 50 data points
x_d = reshape(range(-1,1,length=50),1,50)
y_d = x_d.^2
D = [(x_d,y_d)]
#
# Model
mod = Chain(Dense(1,5,tanh),Dense(5,1))
# Loss/cost function
loss(x, y) = mean((mod(x).-y).^2)
# Optimization algorithm
opt = ADAM(0.002, (0.99, 0.999))
# Parameters of the model
par = params(mod);
# Running 3000 epochs
for i in 1:3000
    Flux.train!(loss,par,D,opt)
end
```

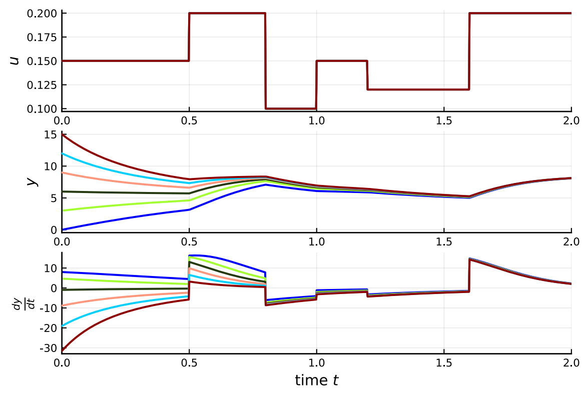

Figure 3. Experimental data of model in 9.

```
# Generating model output
y_m = Tracker.data(mod(x_d));
```

Bishop carries out similar training for functions $|x|$, $\sin 2(\pi x)$, and $2\mathbb{H}_0(x) - 1$; the results from using Julia are similar to those presented in (Bishop, 1994). However, Bishop achieved far superior fitting with 1000 "cycles", presumably because he used a BFGS quasi-Newton method instead of Stochastic Gradient Descent.

4 Case: first order system with input

4.1 Model

For illustrating key ideas, a simple model is chosen which can be visualized in 2D or 3D. A scalar differential equation $\frac{dy}{dt} = f(y,u)$ allows to plot $\frac{dy}{dt}$ as a function of y and u. Consider the model

$$\frac{dy}{dt} = 2(4-y) - 10^2 \exp\left(-\frac{1}{u}\right) y^2 + 10^3 \exp\left(-\frac{1}{u}\right) y, \tag{9}$$

where y can, e.g., be a concentration c_A and u can, e.g., be scaled absolute temperature T.

4.2 Experiments

Figure 3 indicates possible experimental data generated by the model in Equation 9 using OMJulia.

In Figure 4, the experimental simulation data are overlaid on the surface plot of the model given by Equation 9.

For experiments on real systems, the initial state $y(0)$ can not be chosen, but has to be accepted as is. Because of this, real world data typically only cover a fraction of

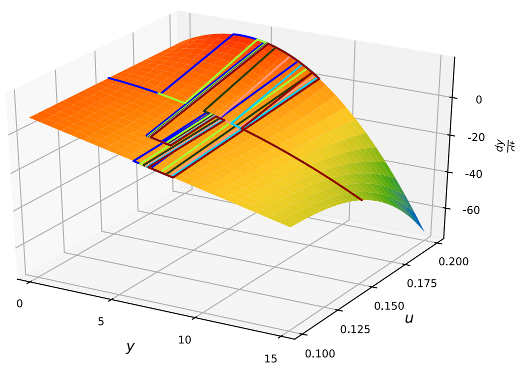

Figure 4. Experimental data from a range of initial states $y(0)$ over a control variation $u(t)$. Comparison with true vector field surface.

the vector field surface. On the other hand, if the purpose of model fitting is to develop a surrogate (or: simplified) model of a more complex model, we have much more freedom in choosing $y(0)$.

4.3 FNN based surrogate models

Building a continuous model based on $(y,u) \rightarrow \frac{dy}{dt}$ as in Section 2.6, and based on the data in Section 4.2 leads to the results in Figure 5.

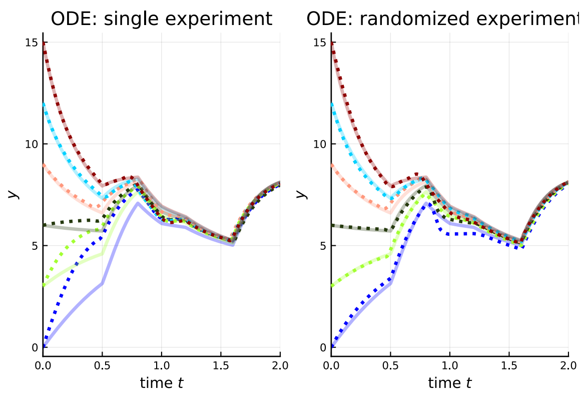

Figure 5. Solid lines: experimental data. Dotted lines: fitted FNN model $\frac{dy}{dt} = \text{FNN}_c(y,u)$. Left panel: using only data from experiment with $y(0) = 15$. Right panel: using all available data.

Observe from the left panel of Figure 5 that models fitted to data from a particular operating regime, may be poor for other regimes.

Building a discrete time model based on $(y_{t-1}, u_t, u_{t-1}) \rightarrow y_t$ as in Section 2.6, and based on the data in Section 4.2 leads to the results in Figure 6.

Observe that the left panel results in Figure 5, are predictions based on a "moving average" of the data,

$$\hat{y}_t = \text{FNN}_{\text{d,PE}}(y_{t-1}, u_t, u_{t-1});$$

with the given prediction error (PE) loss function, this is the correct predictor formulation. The right panel results in Figure 5 are based on a recursive predictor

$$\hat{y}_t = \text{FNN}_{\text{d,PE}}(\hat{y}_{t-1}, u_t, u_{t-1})$$

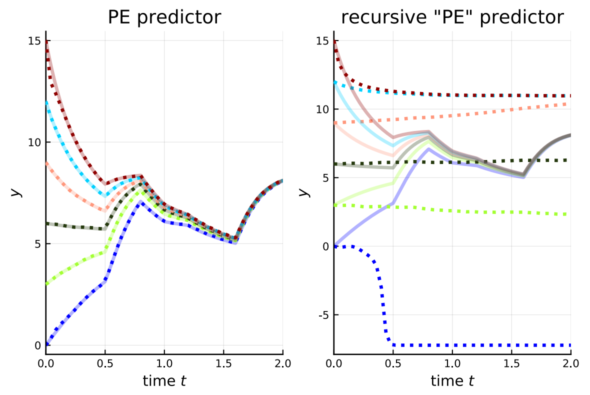

Figure 6. Solid lines: experimental data. Dotted lines: fitted FNN model $y_t = \text{FNN}_d(y_{t-1}, u_t, u_{t-1})$ using all available data. Left panel: PE predictor, Right panel: ad hoc recursive use of PE predictor.

which is somewhat ad hoc, Section 2.6. From Figure 6, the ad hoc recursive model is not suitable for long range predictions.

Next, we consider building a discrete time model based on $(y_{t-1}, \ldots, y_{t-5}, u_t, \ldots, u_{t-5}) \rightarrow y_t$ as in Section 2.6, and based on the data in Section 4.2 leads to the results in Figure 7.

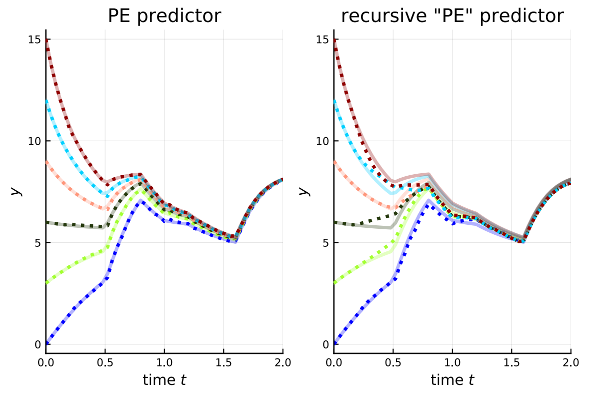

Figure 7. Solid lines: experimental data. Dotted lines: fitted FNN model $y_t = \text{FNN}_d(y_{t-1}, \ldots, y_{t-5}, u_t, \ldots, u_{t-5})$ using all available data. Left panel: PE predictor, Right panel: ad hoc recursive use of PE predictor.

With this more complex model, both the "moving average" predictor

$$\hat{y}_t = \text{FNN}_{\text{d,PE}}(y_{t-1}, \ldots, y_{t-5}, u_t, \ldots, u_{t-5});$$

with the given prediction error (PE) loss function, the recursive ad hoc PE predictor

$$\hat{y}_t = \text{FNN}_{\text{d,PE}}(\hat{y}_{t-1}, \ldots, \hat{y}_{t-5}, u_t, \ldots, u_{t-5}) \qquad (10)$$

give decent predictions — even with the use of $n_{\xi}^{(1)} = 3$ nodes in the hidden layer.

4.4 Hybrid model

Often, a simplified model is available which does not represent the system well over an extended operating regime.

In Figure 8, the simplified model is a linear approximation at $u = 0.15$ (left panel).

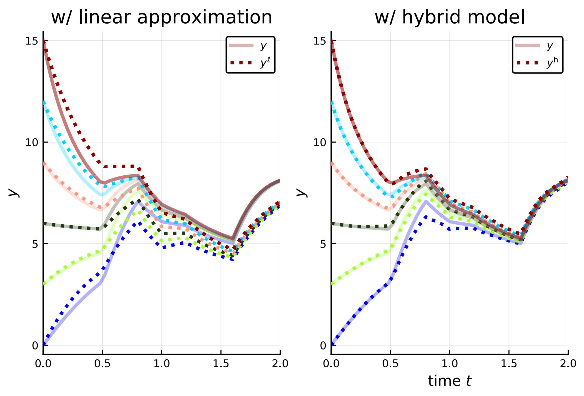

Figure 8. Solid lines: experimental data. Dotted lines: approximate models. Left panel: linear approximation, Right panel: hybrid model with ad hoc recursive PE predictor as correction to linear approximation.

With available experimental data, a model using 5 past deviations $\Delta y = y - y^{\ell}$ as in 10 was trained, and used to correct the linear approximation in a hybrid model $y_t^{\mathrm{h}} = y_t^{\ell} + \mathrm{FNN}_{\mathrm{d,PE}}(\ldots, \hat{y}_{t-j}, \ldots, u_{t-j}, \ldots)$ with results as in the right panel of Figure 8.

5 Discussion and Conclusions

This paper has aimed at illustrating some important concepts related to digital twins for control relevant models. As Figure 4 indicates, physics-based models give much more information than experimental data do — under normal operation, experimental data will only cover a small fraction of the space of inputs, states, and derivatives. However, physics-based models typically involve some approximations that makes it impossible with perfect fit to experimental data. Data driven methods normally do not impose such restrictions.

A sub goal has been to inspire to experiment with freely available computing tools for machine learning. Examples of building both continuous time and discrete time surrogate models are given. Continuous time model building requires access to some state derivative, and give decent prediction capabilities — for the simple case where all states are available. When using Feed forward Neural Nets, the resulting model is normally a prediction error (PE) model, which in principle only offer one step-ahead prediction with limited suitability for, e.g., Model Predictive Control. By making a PE model recursive, some success may be achieved. An interesting alternative is to use Recurrent Neural Nets (RNN), which gives a state description. Due to space limitations, RNN is not treated here.

References

Peter Benner, Mario Ohlberger, Albert Cohen, and Karen E. Wilcox. *Model Reduction and Approximation: Theory and Algorithms*. SIAM, Philadelphia, USA, July 2017. ISBN 978-1611974812.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Sha. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 49(1):65–98, 2017. doi:10.1137/14100067.

Christopher M. Bishop. Neural networks and their applications. *Rev. Sci. Instrum.*, 65(6):1803–1832, June 1994.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, April 2011. ISBN 978-0387310732.

Stephen Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Algebra. Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018. ISBN 978-1316518960.

Elías Cueto, David González, and Icíar Alfaro. *Proper Generalized Decompositions: An Introduction to Computer Implementation with Matlab*. Springer, March 2016. ISBN 978-3319299938.

Abdulmotaleb El Saddik. Digital Twins: The Convergence of Multimedia Technologies. *IEEE Multimedia*, 25(2):97–92, 2018. ISSN Print ISSN: 1070-986X Electronic ISSN: 1941-0166. doi:10.1109/MMUL.2018.023121167.

Jay A. Farrell and Marios M. Polycarpou. *Adaptive Approximation Based Control. Unifying Neural, Fuzzy and Traditional Adaptive Approximation Approaches*. Wiley-Interscience, Hoboken, New Jersey, 2006.

Bruce A. Finlayson. *The Method of Weighted Residuals and Variational Principles*. Classics in Applied Mathematics. SIAM, Philadelphia, 2014. ISBN 978-1611973235.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, Piscataway, NJ, second edition, 2015. ISBN 978-1-118-85912-4.

Mario Hermann, Tobias Pentek, and Boris Otto. Design Principles for Industrie 4.0 Scenarios. In *Proceedings, 2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE, January 2016. doi:10.1109/HICSS.2016.488.

Mike Innes. Flux: Elegant Machine Learning with Julia. *Journal of Open Source Software*, 2018. doi:10.21105/joss.00602.

Bernt Lie, Arunkumar Palanisamy, Alachew Mengist, Lena Buffoni, Martin Sjölund, Adeel Asghar, Adrian Pop, and Peter Fritzson. OMJulia: An OpenModelica API for Julia-Modelica Interaction. In *Proceedings of the 13th International Modelica Conference*, pages 699–708, February 2019. doi:10.3384/ecp19157. Regensburg, Germany, March 4–6, 2019.

Zhendong Luo and Goong Chen. *Proper Orthogonal Decomposition Methods for Partial Differential Equations*. Academic Press, 2018. ISBN 978-0128167984.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning. The MIT Press, August 2012. ISBN 978-0262018029.

Christopher Rackauckas and Qing Nie. DifferentialEquations.jl — A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software*, 5(15), 2017. doi:10.5334/jors.151.