Generating Comprehensible Equations from Unknown Discrete Dynamical Systems Using Neural Networks

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

John M. Maroli, M.S.

Graduate Program in Department of Electrical and Computer Engineering

The Ohio State University

2019

Dissertation Committee:

Ümit Özgüner, Advisor

Keith Redmill, Co-Advisor

Yingbin Liang

Abstract

This research presents a novel framework for generating system equations from the input-output data of unknown discrete dynamical systems. The two-step process consists of system identification followed by a black box input-output analysis in the likes of Monte Carlo sample-based global sensitivity analysis. System identification is performed using temporal convolutional networks trained with only input-output data. A structured approach to network design and training is detailed for yielding accurate identification models and a benchmark is performed using a publicly available dataset known as the Silverbox. The trained identification model serves as a system emulator that can be excited at low computational cost, allowing for detailed sample-based sensitivity analysis. The key to the analysis is an imagined decomposition of the the model into a sum of less complex constituent functions of all input combinations. A method for sampling the constituent functions is presented to not only determine relevant constituents, but to estimate them as well. The sum of relevant constituent functions is equal to the original model, which parallels the source system of the original input-output data. The imagined decomposition of the identification model allows for a potentially complex estimation problem to be broken down into many smaller and less complex problems.

The analysis resultant is a human comprehensible mathematical model for the discrete dynamical system, where comprehensibility implies that the equation gives insight into system operation. The presented framework helps to shed light on black box identification

models, and the system equation extracted from the identification model can be used as a transparent replacement for the original model. This aids in a myriad of practical applications such as control, stability analysis, and software verification. The framework is fully implemented and made publicly available. A number of synthetic examples are presented along with data-driven analysis of the Silverbox dataset, vehicle dynamics data, and simulated motor cooling data.

to my wife, Kimberly . . .

to our baby in heaven.

"Sometimes the smallest things take up the most room in your heart."

- Winnie the Pooh

Acknowledgments

I thank my wife, Kimberly, for her unconditional love and support throughout my graduate education. I thank my parents, John and Patricia, for raising me to be who I am today and for their constant love and support in all my endeavors. I thank my extended family and friends for their loving encouragement and support throughout my life, especially during these past few years. Most of all, I thank God for providing me with the passion and perseverance to complete my Ph.D.

I thank my advisor, Professor Ümit Özgüner, and co-advisor, Professor Keith Redmill, for their guidance in my education and research. I thank Professor Lisa Fiorentini for her willing assistance in helping me find my research direction and her passion for teaching others. I thank the many other instructors I had at The Ohio State University over my 8 and a half years of undergraduate and graduate education, of whom there are too many to name.

Vita

May 2015	B.S. Electrical and Computer Engineering, The Ohio State University
May 2019	M.S. Electrical and Computer Engineering, The Ohio State University
August 2015 - May 2019	Graduate Research Associate, The Ohio State University
May 2013 - Present	Engineering Student Trainee, NASA Glenn Research Center

Publications

[1] J. Jing, J. M. Maroli, Y. B. Salamah, M. Hejase, L. Fiorentini, and Ü. Özgüner. "Control Method Designs and Comparisons for Tractor-Trailer Vehicle Backward Path Tracking". In: *2019 American Control Conference (ACC)*. July 2019, pp. 5531–5537.

[2] J. M. Maroli, Ü. Özgüner, and K. Redmill. "A Framework for Automated Collaborative Fault Detection in Large-Scale Vehicle Networks". In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. June 2019, pp. 1923–1927. DOI: 10.1109/IVS.2019.8814176.

[3] M. Hejase, J. Jing, J. M. Maroli, Y. Bin Salamah, L. Fiorentini, and Ü. Özgüner. "Constrained Backward Path Tracking Control using a Plug-in Jackknife Prevention System for Autonomous Tractor-Trailers". In: *Proceedings of the 2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. Nov. 2018, pp. 2012–2017. DOI: 10.1109/ITSC.2018.8569262.

[4] D. Yang, J. M. Maroli, L. Li, M. El-Shaer, B. A. Jabr, K. Redmill, F. Özgüner, and Ü. Özgüner. "Crowd Motion Detection and Prediction for Transportation Efficiency in Shared Spaces". In: *Proceedings of the 2018 IEEE International Science of Smart City Operations and Platforms Engineering in Partnership with Global City Teams Challenge (SCOPE-GCTC)*. Apr. 2018, pp. 1–6. DOI: 10.1109/SCOPE-GCTC.2018.00007.

[5] J. M. Maroli, Ü. Özgüner, K. Redmill, and A. Kurt. "Automated rotational calibration of multiple 3D LIDAR units for intelligent vehicles". In: *Proceedings of the 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. Oct. 2017, pp. 1–6. DOI: 10.1109/ITSC.2017.8317598.

[6] G. Ozbilgin, Ü. Özgüner, O. Altintas, H. Kremo, and J. Maroli. "Evaluating the requirements of communicating vehicles in collaborative automated driving". In: *Proceedings of the 2016 IEEE Intelligent Vehicles Symposium (IV)*. June 2016, pp. 1066–1071. DOI: 10.1109/IVS.2016.7535521.

Fields of Study

Major Field: Electrical and Computer Engineering

Table of Contents

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

The rise of machine learning has yielded substantial advancements to system identification and modelling at the cost of transparency and human comprehension. Neural networks models have generality and accuracy that is difficult to outperform without domain specific knowledge, however their structure makes it challenging to gain insight into systems that they represent. As such, neural network identification models are commonly likened to opaque black boxes. This obscurity hinders human comprehension and weakens trustworthiness. Additionally, many classical control and stability analysis techniques cannot be applied to black box models and software verification becomes computationally expensive if not impossible to perform. For these reasons, a general methodology for generating human comprehensible equations of unknown discrete dynamical systems typically represented by black box models is needed.

1.1 Motivation and Problem Statement

Neural networks are well known for their ability to approximate generic functions, being declared universal function approximators [1]. They have also been demonstrated as effective for both identification and control of nonlinear dynamical systems [2][3]. Neural network models have been shown to provide accuracy at least as good as conventional system modelling methods [4]. Recurrent neural networks (RNN) are no more than dynamical

systems themselves, while feedforward neural networks (FNN) are static systems. With a series of input and/or output delays, FNNs can also be trained to represent dynamical systems when the delays are exposed to the network input.

Identification of dynamical systems using both feedforward and recurrent neural network architectures is formally presented in the seminal work of Narendra et. al. [2]. Neural networks can be used to model complex and highly nonlinear systems for which no classical models exist. While neural networks perform well for modelling systems, their trained structure generally gives little information about the systems they represent. In the case where a neural network is used to model an unknown system characterized only by input-output data, it is as if we have a black box model of a black box. This compounding obscurity weakens the human trustworthiness of the system model as well as the whole of neural network identification models. It is becoming increasingly important to explain black box models for not only human trustworthiness, but for legal reasons as well [5]. The General Data Protection Regulation (GDPR), adopted by the European Parliament and turned into law in 2018, requires the "meaningful explanations of the logic involved" in automated decision making. There are a myriad of other reasons as to why black box system models are undesirable. From a control perspective, many classical control and stability analysis techniques cannot be applied to these systems. Software verification becomes a challenge as well, with no succinct way to verify the models [6].

In this work, we perform identification of discrete dynamical systems using a neural network variant and then introduce a method for analyzing trained neural networks to generate approximate equations for the systems they represent. In this way, we use a neural network as a tool to extract a system equation from a system [7]. Neural networks are primarily chosen for both their identification performance and their low computational cost

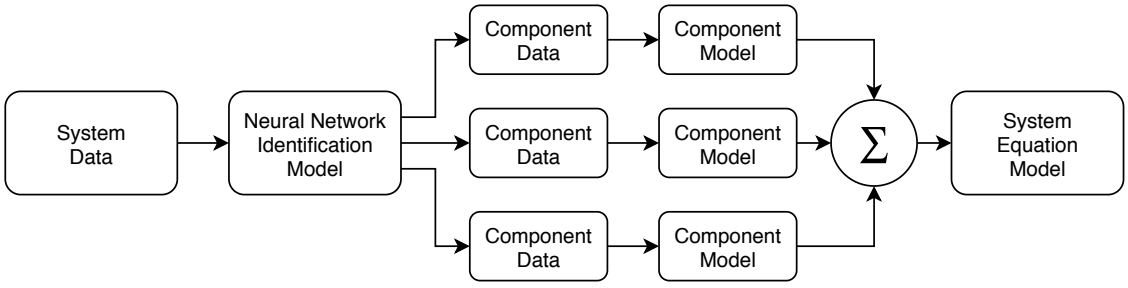


Figure 1.1: The process flow from input-output data to system equation through decomposition of a virtual system identification model.

for forward propagation of input samples. They are also some of the biggest offenders when it comes to having a lack of model transparency and are a prime example of an opaque system. Our analysis sheds light on the black box model that neural networks are often viewed as when used for system identification, and by proxy exposes the black box system modelled by the neural network. We refer to the analysis process as system equation extraction, which draws concepts from the established fields of neural network rule extraction and sensitivity analysis.

The identification and analysis procedures are combined to create a framework for modelling generic systems. This combined framework parses input-output data to generate a mathematical model of the underlying system in the form of a discrete dynamical system equation. The identification process creates a virtual model of the system that is then computationally analyzed. The analysis builds a system equation based on a decomposition of the identification model, breaking down the complexity of generating the equation into less complex fragments. This eases the process of creating mathematical system models and allows for the modelling of systems with compounding complexity. The process flow is presented in Fig. 1.1 and is regarded as the main contribution of this work.

1.2 Related Work

There are three primary areas of literature from which the presented work draws inspiration. The ultimate goal of this analysis is the generation of a mathematical model of a system using input-output data. In essence, this goal is system identification. The neural network identification methods used in this work accomplish the task of creating an accurate mathematical model with high generalization to generic systems, however they lack transparency. Other identification methods that may have transparency generally lack accuracy or generality. In attempt to obtain the accuracy and generality of neural networks while keeping model transparency, neural network rule extraction was examined. In the spirit of generating a system equation, the extraction of human understandable regression rules from neural networks was desired. This leads into sensitivity analysis, which is utilized in this work to generate a transparent system equation with the neural network properties of accuracy and generality.

1.2.1 System Identification

The primary goal of system identification is to create a mathematical model of a dynamical system using measured data. The problem is ultimately finding a model amongst a set of candidates that fits the data [8]. Rather than finding a single candidate model to fit the original data in this work, multiple candidate models are found to fit less complex fragments of the decomposed model. The summand of these fragments is then the overall model fitting the original data.

This work contains two levels of system identification as seen in Fig. 1.1. The initial identification model is created from the model class of neural networks, chosen for the ability to yield high accuracy and applicability to generic systems. The low computational cost of

propagating an input through a neural network model is also advantageous for performing sensitivity analysis, where propagation of many inputs may be computationally expensive. For this reason, machine learning is a popular choice for the construction of meta-models to be paired with sensitivity analysis [9]. The second level of system identification operates on a decomposition of the first level to fit basic equation models to the decomposed fragments. This process can use any standard curve-fitting techniques [10].

The foundations of neural network system identification are presented in the seminal work of Narendra and Parthasarathy [2]. Neural network based control is also presented to compliment identification. The work is extended in [3] by Levin and Narendra with the additional study of observability. Another foundational work for neural network system identification shows the building of generic nonlinear autoregressive moving average with exogenous inputs (NARMAX) [11] models using neural networks [12]. Similar work pertaining to nonlinear autoregressive with exogenous inputs (NARX) models is presented in [13]. These works provide a basis for neural network system identification that has since grown exponentially, particularly with the rise of machine learning in the past decade. The discussion of neural network system identification is continued in Chapter 2.

1.2.2 Rule Extraction

Rule extraction aims to extract human understandable rule sets from a neural network that provide similar (ideally identical) function to the original network. An early survey of rule extraction techniques by Andrews, Diederich, and Tickle [14] provides a conglomerate of methods as well as a widely used taxonomy for classifying them. Their work is extended and consolidated in [15][16][17]. Three main categories for neural network rule extraction techniques arise: decompositional, eclectic, and pedagogical. Decompositional techniques

rely on the analysis of the neural network itself (e.g. individual network weights), while pedagogical techniques extract rules from input-output relationships. Eclectic techniques are a hybrid of the aforementioned methods, with no clear belonging in one group or the other. A fourth group, deemed compositional, is brought up to address rule extraction from ensembles of neurons rather than individual neurons as in the decompositional approach.

We view our goal of system equation extraction as a rule extraction problem where we extract a single rule (an equation) from a neural network trained for regression (the neural network identification model). A number of existing works present methods for extracting rules from neural networks trained for regression. The work of Setiono et. al. [18][19] holds notable significance, extracting linear rules in the form of *if condition, then consequence*. While this method does not provide the generic system equation we desire, it does extract a number of linear equations providing a close approximation to the original neural network. Saito and Nakano [7] proposed another method for extracting regression rules from neural networks around the same time, where the consequential portions of the *if...then...* rules are expressed as polynomial equations. The methods proposed in these prior works are decompositional in nature, and rely on knowledge of the underlying neural network. To perform system equation extraction in the most general form (from a black box model) and avoid limiting ourselves to particular neural network architectures, we look into purely pedagogical approaches.

1.2.3 Sensitivity Analysis

Neural network sensitivity analysis can be used in a strict pedagogical fashion to perform rule extraction from neural networks. The early work of Ruck et. al. [20] looks at examining the sensitivity of a network's outputs to its inputs in order to determine the usefulness of

each input feature. While their work uses weight analysis to do this and would fall on the decompositional end of the rule extraction spectrum, their concept of sensitivity analysis to determine network input importance is paramount in the field of rule extraction. The work of Kewley et. al. [21] is a seminal work using neural network sensitivity analysis in a purely pedagogical manner. All input variables but one at a time are held at their average values while the one input is varied over its entire range. The variability is measured at the network outputs, capturing the sensitivity of the output to each input. The most important inputs are deduced to be the most sensitive, and insensitive inputs are found to be removable without significant pitfalls. The input-output relationships gathered help further understanding of the neural network model. An extension to the work is made by Embrechts et. al. [22] that likens [21] to one dimensional sensitivity analysis (1-D SA) and introduces two dimensional sensitivity analysis (2-D SA). In 2-D SA, all combinations of two input variables are varied over their allowable ranges in an attempt to understand how pairs of inputs effect the output. To reduce computational burden, 1-D SA is used to identify and eliminate insignificant inputs prior to the use of 2-D SA. The possibility for 3-D SA is mentioned, implying a more general n-D SA. Cortez et. al. [23] later use the n-D SA implication as part of a basis to introduce a novel sensitivity analysis visualization approach that they call Global Sensitivity Analysis (GSA) for determining overall neural network input importance (global sensitivity analysis is also a much broader category of sensitivity analysis discussed later). They later extend their work in [24] and introduce three new sensitivity analysis methods, comparing them to 1-D SA and GSA. Notably, they emphasize that the sensitivity analysis methods examined could be used on almost any black box model, making them all purely pedagogical.

One of the earliest works focusing on pedagogical rule extraction comes from Saito and Nakano [25]. They extract symbolic knowledge from a neural network trained to diagnose diseases when given symptoms. For their analysis, the inputs to the network are set so that a single symptom is exhibited, and the resultant disease diagnosis is observed. This is performed for all symptoms individually, and then the effect of having no symptoms on the disease prediction is subtracted from each network output. The resultant is the effect of a single symptom (input) on a disease (output). Despite their work being one of the earliest in the field of neural network rule extraction, it draws significant relation to this research. They are essentially isolating the contributions of the one-combinations of inputs (individual inputs), while our analysis is expanded to isolate all combinations. This concept of combinatoric analysis can be seen in the work of Vahed and Omlin [26][27]. They observe the effects of all possible combinations of input strings on recurrent neural networks, detailing the first purely pedagogical rule extraction approach for recurrent networks.

The previous works detailed in this section pertain to sensitivity analysis in the scope of neural networks. A broader class of sensitivity analysis methods for examining generic models is known as global sensitivity analysis (not to be confused with the GSA method of [23]). These methods attempt to quantify output uncertainty due to the uncertainty in the input variables and combinations of input variables. This usually equates to determining which inputs and input combinations effect the output. One of the most famous global sensitivity analysis methods comes from I. M. Sobol. In his seminal work [28], it is shown that an integral function can be decomposed into summands of different dimensions, and likewise the variance of the function can be decomposed into the variances of the summands. This concept is used to establish which inputs or groups of inputs contribute to the output variance of a model and provides an estimate of model sensitivity. The most notable

contribution is the introduction of Sobol indices, which represent the contributed portion of variance of each summand to the total variance of the model. The use of variance to compute sensitivity is commonly referred to as variance-based sensitivity analysis. An important expansion on the work of Sobol was the introduction of a "total effect" parameter index to measure the contribution of each input parameter to the model variance as opposed to the contribution of each summand [29]. Further work by Sobol on estimating the influence of individual inputs or groups of inputs on the model output is presented in [30].

It is important to note that the decomposition of a function is a recurring theme in sensitivity analysis that existed before the work of Sobol. The Analysis of Variance (ANOVA) decomposition breaks up the variance of a model output into terms of increasing dimensionality. In this way, the function decomposition that empowers the creation of Sobol indices also empowers ANOVA. A number of other techniques using similar function decomposition are surveyed in [31]. The function decomposition of a generic function $f(x)$ with input $x = (x_1, ..., x_n)$ as noted in [30] is

$$f(x) = f_0 + \sum_i f_i(x_i) + \sum_{i<j} f_{ij}(x_i, x_j) + \cdots + f_{12..n}(x_1, x_2, \ldots, x_n). \tag{1.1}$$

This effectively breaks $f(x)$ into a summand of 2^n functions covering all unique combinations of input variables. The analysis technique presented in Chapter 3 is driven by this same decomposition.

1.3 Overview and Outline

The predominant contribution of this work is a new framework to generate discrete dynamical system equations from input-output data. System identification is first performed on the data and then the identification model is analyzed using a sample-based sensitivity analysis. Constructing an accurate system identification model is a key prerequisite

to the analysis. System identification is examined in Chapter 2, where an accurate and generalizable identification method is presented. The model analysis framework is then presented in Chapter 3. The framework is an automated and iterative process that generates a comprehensible discrete dynamical system equation. Chapter 4 presents an in-depth analysis of the framework pertaining to practical use with a full implementation in Python. A multitude of synthetic examples are examined and real-world data is analyzed pertaining to vehicle systems.

Chapter 2: System Identification using Neural Networks

This work pertains to identification of observable systems described by the state equations

$$\begin{array}{rcl} x[k+1] & = & f(x[k],u[k]) \\ y[k] & = & h(x[k]) \end{array} \tag{2.1}$$

where $f(\cdot)$ and $h(\cdot)$ represent generic functions, $x[k] \in \mathbb{R}^n$ is the system state at time k, $y[k] \in \mathbb{R}^p$ is the system output, and $u[k] \in \mathbb{R}^m$ is the system input. Systems without measurable states are the focus in this work, so an input-output model is desired. The output equation for $y[k]$ can be expanded using the state equation as follows:

$$\begin{aligned} y[k] &= h[x(k)] \\ &= h[f[x(k-1),u(k-1)]] \\ &= h[f[f[x(k-2),u(k-2)],u(k-1)]] \\ &= h[f[f[f[x(k-3),u(k-3)],u(k-2)],u(k-1)]] \\ &= h[f[f[f[f[...f[x(0),u(0)]...],u(k-3)],u(k-2)],u(k-1)]]. \end{aligned}$$

It becomes easy to see that $y[k]$ is a function of the initial state $x[0]$ and the input trajectory $u[k-1],...,u[0]$. Since the system is assumed to be observable, the states can be determined using the outputs. With this information, it is now apparent that $y[k]$ can be determined strictly from past outputs and past inputs.

The work of [3] provides analytical justification for representing (2.1) in the form of the input-output model

$$y[k] = F\left(y[k-1], ..., y[k-n], u[k-1], ..., u[k-n]\right). \tag{2.2}$$

In this model, the system output is calculated as a function of n past outputs along with n past inputs. The goal of identification is to determine F, or approximate it as best as possible. The identification model approximation of F is referred to as $\hat{F}$. A more generic form of (2.2) is used in this work to account for an input delay $d \geq 0$:

$$y[k] = F\left(y[k-1], ..., y[k-n], u[k-d], ..., u[k-d-n+1]\right). \tag{2.3}$$

Note that (2.2) is the case of (2.3) when $d = 1$, allowing for control of the system through $u[k]$ to be computed using the output feedback of $y[k]$. Uncontrolled systems can be modelled using $d = 0$.

2.1 Choosing an Identification Model

Setting up a suitable identification model forms a basis for system identification [2]. Two major identification models are used in this work: the series-parallel model and the parallel model. A neural network variant is chosen to implement both identification models. The goal of identification is to approximate F with $\hat{F}$, a suitable task for neural networks since they are universal function approximators. While both models are examined, the series-parallel model is preferred in this work since it is essentially the input-output system model of (2.2).

The series-parallel identification model, shown in Fig. 2.1, produces an estimate of the system output $\hat{y}[k] \in \mathbb{R}^p$ using inputs from $u[k-d], ..., u[0]$ and outputs from $y[k-1], ..., y[0]$.

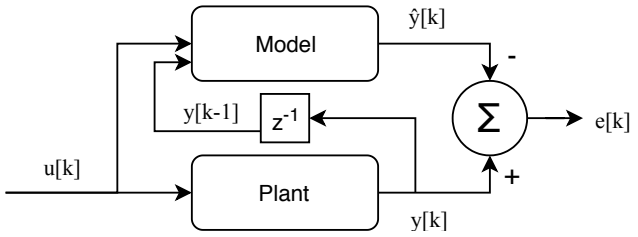


Figure 2.1: The series-parallel identification model.

The series-parallel model mimics the system input-output realization of (2.2) and is represented by

$$\hat{y}[k] = \hat{F}\left(y[k-1], ..., y[k-l_y], u[k-d], ..., u[k-d-l_u+1]\right). \tag{2.4}$$

The number of states n is assumed to be unknown in the identification task, so l_y past outputs and l_u past inputs are used. Both l_y and l_u must be set sufficiently large so that they are greater than n for successful identification. After training, the series-parallel model is simply a static mapping of past inputs and outputs to the estimated output.

The parallel identification model is shown in Fig. 2.2. It is similar to the series-parallel model, but with one key difference: it uses past output estimates $\hat{y}[k-1], ..., \hat{y}[0]$ in place of actual past outputs. The parallel model is represented by

$$\hat{y}[k] = \hat{F}\left(\hat{y}[k-1], ..., \hat{y}[k-l_y], u[k-d], ..., u[k-d-l_u+1]\right). \tag{2.5}$$

The parallel model is recurrent in nature, making neural network representation more complicated. The parallel model encounters stability problems during training, with no guarantee of parameter convergence and no guarantee that the output error will tend towards zero [2]. A simplified version of the parallel identification model is obtained by ignoring past output estimates:

$$\hat{y}[k] = \hat{F}\left(u[k-d], ..., u[k-d-l_u+1]\right). \tag{2.6}$$

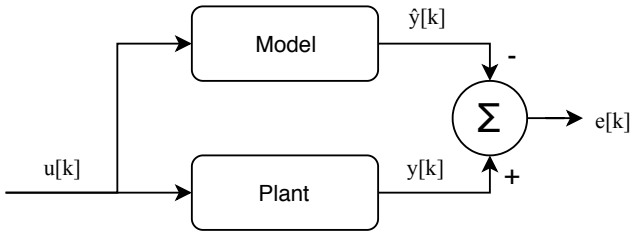


Figure 2.2: The parallel identification model.

This simplified model is a static mapping, eliminating the problem of recurrence. The model analysis methodology presented in Chapter 3 relies on a static map model, so the series-parallel model and simplified parallel model are the focus of this work.

A full parallel model can still be implemented using a neural network without the training issues that occur because of recurrence, so long as the network is trained in a series-parallel fashion and then used as a parallel model. This often works well enough when the series-parallel model is sufficiently accurate [2]. The past system outputs in the series-parallel model are simply replaced with past estimated outputs when switching from training to operation. Operation of a series-parallel model as a parallel model in this fashion can be used to simulate a system, estimating the future output trajectory using only the input trajectory. While this can be done with the simplified parallel model, a larger number of past inputs are generally needed if the system in question is a function of past outputs.

Having an accurate simulation model of a system is beneficial as it allows the system output to be estimated without actual measurement. This enables the creation of computer models and system observers for example. Simulation models in this work are trained as series-parallel models and converted to parallel models. The series-parallel models themselves perform one-step-ahead prediction, which is also a valuable prediction metric that can be extended to generic n-step-ahead prediction.

Identification models other than the two described can be obtained by expanding the model input. Measured system states, for example, could be included in the identification model at the input or output side. The neural network used as the identification model is a static function map trained to yield a specific $\hat{y}[k]$ for each set of arguments to $\hat{F}$. As such, $\hat{y}[k]$ and the arguments of $\hat{F}$ can be adjusted to contain any relevant model variables. The focus of this work is restricted to the series-parallel and simplified parallel input-output models since we intend to examine black box systems.

2.2 Neural Network Model Implementation

Neural networks are selected to represent $\hat{F}$ since they are universal function approximators and work in a diverse variety of applications. As a preliminary disclaimer to avoid notation confusion, we emphasize that x is used to depict a neural network input for the remainder of this work, and is representative of a system's state only where explicitly stated. The variable x is commonly used in both control and machine learning literature.

An overview of neural network system identification is provided in this section with notation adapted to compliment discrete dynamical systems. The network's goal is to generate an estimate of the system output $\hat{y}[k]$ belonging to an estimated output sequence $\hat{y}[k],...,\hat{y}[0]$ using an input sequence $x[k],...,x[0]$, where $x[k],...,x[0] \in \mathbb{R}^q$ are inputs to the neural network identification model $\hat{F}$. The network output $\hat{y}[k]$ is a prediction of the real system output $y[k]$ from the sequence of observed outputs $y[k],...,y[0]$.

In practice, the entire input sequence from time 0 to k is either not needed or not available for use in predicting $y[k]$. The length l of the input sequence from time k and prior that is used to predict the output is referred to as the effective history of the network in this work.

These input values are gathered to create an input matrix $\bar{x}[k]$, defined as

$$\bar{x}[k] = \begin{bmatrix} x[k] & x[k-1] & \cdots & x[k-l+1] \end{bmatrix}. \tag{2.7}$$

These select values of the input sequence directly correspond to inputs of the neural network. Constraining the input sequence in this manner sets the neural network input dimension to lq. Since the effective history of the network will likely not cover all past inputs, the network can be functionally represented by the input-output relationship

$$\hat{y}[k] = \hat{F}(\bar{x}[k]). \tag{2.8}$$

The neural network input-output relationship is easily related to the series-parallel identification model by defining the input at each time k as

$$x[k] = \begin{bmatrix} y[k-1] \\ u[k-d] \end{bmatrix}. \tag{2.9}$$

If $l = n$, then the functional relationship of the neural network model in (2.8) mimics the generic input-output model of (2.3). As mentioned before, n is assumed to be unknown, so setting l to a sufficiently large value such that $l \geq n$ is needed for proper identification. Setting l for the neural network input in this manner is effectively equivalent to setting $l_y = l_u$ in the identification model, and is done for simplicity. The simplified parallel identification model is represented similarly by defining the input at each time k as

$$x[k] = u[k-d]. \tag{2.10}$$

For the most generality in representing discrete dynamical systems of the form (2.1), the series-parallel model is recommended. The simplified parallel model is used when it is known that past outputs are not needed to estimate the current system output. Estimating a system that is a function of past outputs using the parallel model will require a longer history

of past inputs than needed due to the recursive calculation of the output. In this case, using past outputs leads to a simpler model with fewer overall arguments. A long input history requirement for a parallel model indicates that a series-parallel model should be tried.

Once an identification model has been chosen, it must be set up for training. Training data is obtained by recording sequences of input-output data from the plant, this is true regardless of whether the series-parallel or parallel identification models are used. Input-output data can be gathered as a system runs under normal operation, or it can be gathered when the system is purposefully excited. A sufficient and diverse amount of data must be collected for proper training, however this is entirely application dependent. More complex systems will generally require more data for training. Training requires input-output data pairs from the system in the form of $(\bar{x}[k], y[k])$. The error between the system output and the network predicted output $e[k] = y[k] - \hat{y}[k]$ is used to train the network. Any standard neural network optimization method can be used, however the Adam [32] optimizer is used for all training in the context of this work. When sufficiently trained, the network output $\hat{y}[k]$ serves as an estimate for the sequence output $y[k]$ given input $\bar{x}[k]$.

Before training can begin, the input data and output data should be normalized to be of zero mean and unit covariance. This is accomplished by subtracting the mean of all input samples $\mu^{(x)}$ from each individual input sample and then dividing each resultant by the standard deviation of all input samples $\sigma^{(x)}$. The output training data is normalized likewise for training with mean $\mu^{(y)}$ and covariance $\sigma^{(y)}$. The model is then trained to minimize $e_i[k]$. After training, all inputs to the network must be normalized using $\mu^{(x)}$ and $\sigma^{(x)}$, and all outputs must be reconstituted through multiplying by $\sigma^{(y)}$ and then adding $\mu^{(y)}$. The input-output relationship of (2.8) is modified to yield (2.11):

$$\hat{y}[k] = \sigma_y \hat{F}\left(\frac{\bar{x}[k] - \mu_x}{\sigma_x}\right) + \mu_y. \tag{2.11}$$

2.3 Temporal Convolutional Networks

The neural network variant used in this work as the identification model is known as the temporal convolutional network (TCN). This variant is beneficial for system identification, although it is not a requirement for the equation generation methodology presented in Chapter 3. The only requirement of the analysis is that a static system model exists with the specified input-output structure (a static model always yields the same output given the same input sequence, in opposition to dynamic models). For identification, the TCN is trained using corresponding input-output data pairs from a system in the same manner as a standard FNN.

The TCN is described and evaluated for sequence modelling tasks in [33]. It is shown to yield better overall performance than RNNs for sequence modelling despite RNNs having a theoretical ability to capture infinitely long sequence history. As was demonstrated with FNNs in early system identification work [2], convolutional neural networks (CNNs) can still represent dynamical systems using delays despite lacking recurrent connections. The TCN likewise does not require recurrent connections. Recent work on sequence modelling proves that stable recurrent neural networks are well approximated by FNNs, and demonstrates that often recurrent models can and should be replaced by non-recurrent models for sequence modelling tasks [34]. Other recent advances in sequence modelling are gravitating towards convolutional architectures as opposed to recurrent architectures as well, as is the case with WaveNet [35]. WaveNet is a precursor to the TCN which introduced a new architecture that relies on dilated causal convolutions, a predominant aspect of TCNs. The TCN demonstrates

benefits over standard CNNs for sequence modelling, most notably an increased sequence history. Identification of dynamical systems using convolutional neural networks is presented in [36]. The performance of CNNs is compared to FNNs in regards to dynamical system identification, where the CNN is found to outperform the FNN in the presence of noise. Performance without noise is similar, although the CNN has more hidden layers and less hidden nodes than FNNs used for system modelling. The promising performance of CNNs for dynamical system identification along with the demonstrated superiority of TCNs for sequence modelling tasks inspired the investigation of TCNs for system identification in this work.

As mentioned with generic neural network identification model implementation, the TCN's goal is to again generate an estimate of the system output $\hat{y}[k]$ belonging to an estimated output sequence $\hat{y}[k],...,\hat{y}[0]$ using an input sequence $x[k],...,x[0]$. The TCN architecture is deliberately kept simple, with two key attributes: the output must be kept to the same length as the input, and there can be no information leakage from the future to the past. To keep the output and input lengths equal, a 1D fully-convolutional network is used with each hidden layer equal in size to the input layer. Zero padding is added to keep layer sizes equal. The number of time steps exposed to the network defines the input and output length, not the dimensionality of either. The network uses only causal convolutions to prevent information leakage.

To expand the effective history l of the network, dilated causal convolutions are used to yield a history that grows exponentially with network depth. A dilation factor D introduces a fixed buffer of size $D-1$ between taps of the convolution filter of size K. Rather than using a simple convolutional layer for each level, a generic residual module is employed to help stabilize deeper and larger TCNs. The dilation factor for each residual block is determined

as $D = 2^i$, where i is the level of the residual block starting from 0. Each residual module includes two layers of dilated causal convolution of dilation factor D and filter size K with weight normalization, nonlinearity, and dropout. An optional 1x1 convolution is added from the input to the output of each residual to account for inputs and outputs of different widths. Examining the network structure, the effective history of the network can be determined as

$$l = (K-1)(2^{L+1}-1) - K + 2. \tag{2.12}$$

The TCN architecture must be specified in order to train a successful system identification model. In designing the network and determining the filter size K and number of levels L, the effective history l of the network must be taken into account. The values of K and L must be selected such that l is greater than or equal to the input $x[k-n]$ needed to determine the output $y[k]$. Since the system is assumed to be unknown, K and L must be selected to be sufficiently large.

An example TCN with $L = 2$ levels and filter size $K = 3$ is shown in Fig. 2.3. Each level is represented by a residual block consisting of an input layer, hidden layer, and output layer. When connecting residuals, the result of the output layer of one residual becomes the input layer of the next residual. This connection is simplified in the middle layer of Fig. 2.3 to showcase the network effective history calculation. It can be seen that the effective history of the network is $l = 13$, so we would represent this network as $\hat{y}[k] = \hat{F}\left(\begin{bmatrix} x[k] & \cdots & x[k-12] \end{bmatrix}\right)$.

Every kernel leg in the network contains hidden neurons that are independent from the network effective history and are not visible in Fig. 2.3. The number of these hidden neurons effects the network's ability to learn functions. More hidden neurons in each leg equates to better function approximation since a neural network with a single hidden layer containing a large number of hidden nodes can approximate any continuous function [1]. Increasing the

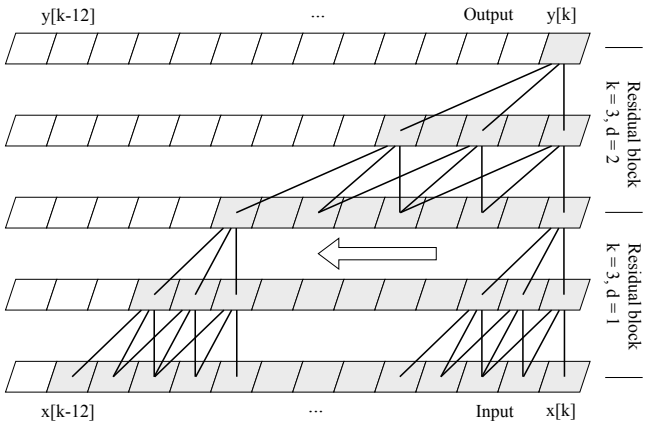


Figure 2.3: Example TCN with $L = 2$ levels and filter size $K = 3$. The full effective history of the network can be seen, extending from $x[k]$ to $x[k-12]$.

hidden neuron count, however, increases required training effort. In this work, sufficiently large values were experimentally chosen for each trained network. If computational burden is not an issue, a high number of neurons per layer can be selected initially, yielding good approximation potential.

After the TCN structure is setup, training can begin as with the generic neural network formulation. Input-output data pairs from the system in the form of $(\bar{x}[k], y[k])$ are gathered with regard to the format of $x[k]$ for the chosen identification model. It should be noted that while a standard FNN would have lq individual inputs without regards to order or position, the TCN has l inputs of dimension q with a significant order.

2.4 The Silverbox Benchmark

The identification capabilities of TCNs and standard feedforward neural networks (FNNs) are benchmarked and compared using the Silverbox dataset: a publicly available dataset from a circuit equivalent to a nonlinear spring-mass damper. The TCN is found to have superior performance in simulation of the test portion of the dataset. The goal of

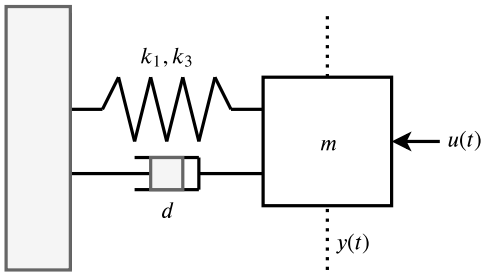


Figure 2.4: The nonlinear spring-mass damper system with identical dynamics to the Silverbox electrical circuit.

the benchmark study is to establish the TCN as a suitable identification method to use in the analysis methodology of Chapter 3. Published benchmark results are surveyed and compared to the TCN results. Analysis of existing results reveals testing variances that effect model performance, so guidelines for fair comparison of models on the Silverbox benchmark are also presented.

The Silverbox is a nonlinear electrical circuit governed by the following second-order differential equation

$$m\ddot{y}(t) + d\dot{y}(t) + k_1 y(t) + k_3 y^3(t) = u(t). \tag{2.13}$$

It is predominantly linear with a cubic nonlinearity. The dynamics are identical to that of the nonlinear spring-mass damper system shown in Fig. 2.4. In the spring-mass damper, a known force $u(t)$ is applied to mass m and displacement $y(t)$ is measured. Constants k_1 and k_3 describe the nonlinear spring and d describes the damper. Like the spring-mass damper, the Silverbox system has a single input and single output. The input $u(t)$ and output $y(t)$ are each represented by a voltage measurement since the Silverbox is an electrical circuit. Noise is inherent in the system and the measurements. The dataset, formally introduced by [37], consists of 131072 samples taken at a frequency of $10^7/2^{14}$ Hz. The input was generated

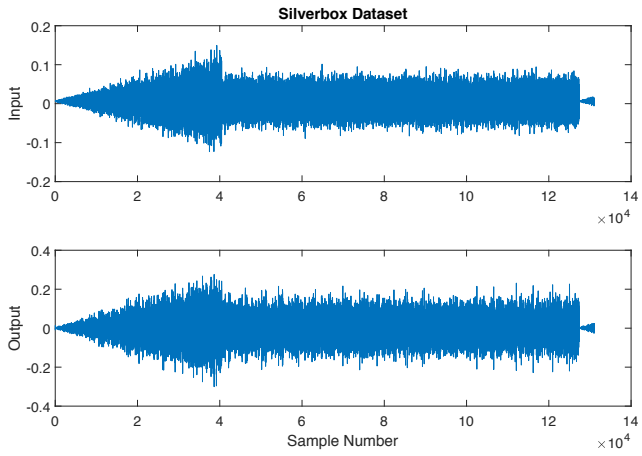


Figure 2.5: Input and output data from the Silverbox.

from a discrete time reference signal converted into an analog signal using zero-order hold reconstruction. The reconstructed signal was then passed through an analog low-pass filter. The reference signal consists of two parts that divide the dataset. The first part is white Gaussian noise with amplitude varied linearly from zero to its maximum value and filtered through a 9th order discrete 200 Hz Butterworth filter. Per the dataset description, the reference for this signal consists of 40000 samples. The second part consists of 10 realizations of a random odd multisine, each separated by 100 zeros. The dataset is shown in Fig. 2.5. Benchmark performance on the dataset is typically obtained by first breaking the dataset into two portions based on how it was created. The dataset resembles an arrow, and the head of the arrow containing the white Gaussian noise is used as the test set for testing identification performance. The body of the arrow containing the multisines is used for estimation of the identification model and is referred to as the training set. The input reaches 149 mV in the test set and 101 mV in the training set, while the output reaches 300 mV and 231 mV respectively. Because of this, extrapolation must be performed to accurately identify the system.

The identification problem is formulated using the series-parallel model without input delay as follows:

$$\hat{y}[k] = \hat{F}\left(\begin{bmatrix} y[k-1] & \dots & y[k-l] \\ u[k] & \dots & u[k-l+1] \end{bmatrix}\right). \tag{2.14}$$

The TCN is trained with the training set to perform one-step-ahead (OSA) prediction using the l most recent output measurements and inputs. The error metric of interest in the Silverbox benchmark is the RMSE performance of the identification model on the test set. Measuring OSA performance involves computing the RMSE between all calculated $\hat{y}[k]$ and their corresponding ground truth values $y[k]$. While the series-parallel identification model is trained to perform OSA prediction, the RMSE of the test set in free simulation is the metric of focus for the Silverbox dataset. Free simulation begins with the OSA formulation of (2.14) and iteratively replaces past output measurements with past output estimates at each step. After l steps, the simulation values are calculated as

$$\hat{y}[k] = \hat{F}\left(\begin{bmatrix} \hat{y}[k-1] & \dots & \hat{y}[k-l] \\ u[k] & \dots & u[k-l+1] \end{bmatrix}\right). \tag{2.15}$$

This is essentially the full parallel model, however it is important to note that the TCN was trained as a series-parallel model due to the issues mentioned in Section 2.1 with the recursive nature of the parallel model. When all $\hat{y}[k]$ are computed using simulation, the RMSE is calculated as done with the OSA prediction.

2.4.1 Existing Benchmark Results

A diverse assortment of nonlinear system identification methods have been applied to the Silverbox dataset with published benchmark results. Works reporting simulation results on the test set are compiled in Table 2.1, while works reporting OSA results are shown in Table 2.2. Values marked with a * were not explicitly stated, but derived from plots. It should be emphasized that most of the works do not present the goal of obtaining the best

benchmark result. The dataset is commonly used to compare variants of similar models under the same scope, as well as to show the effectiveness of parameter reduction. This results in some variance among the published works that prohibits fair comparison based on simulation or OSA RMSE alone. Terms for establishing a fair identification performance benchmark on the Silverbox dataset are presented later in Section 2.4.2.

The first simulation benchmark results come from six works at a special session of the 2004 IFAC Symposium on Nonlinear Control Systems (NOLCOS 2004) focused on the Silverbox. In [38], a plethora of grey box and black box identification techniques were applied using a MATLAB toolbox with the best result obtained using a neural network having a single hidden layer of 30 neurons, a sigmoid activation function, the 10 most recent output measurements, the 11 most recent input measurements, and a custom cubic regressor to match the cubic nonlinearity of the system. Least-squares support-vector machine (LS-SVM) variants are compared in the work of [39], with the best result coming from a fixed-size LS-SVM in the primal space with partial least squares (FS-PLS). The model is split into linear and nonlinear blocks by [40] where fast estimation of the linear portion is performed followed by the nonlinear portion. Another block structure separating linear and nonlinear portions is proposed in [41]. A weighted combination of local linear state-space models is used by [42]. In [43], different dynamic neural network architectures are applied, with a multilayer perceptron having special weight initialization achieving the best result.

Many simulation benchmark results have come from the Silverbox dataset following NOLCOS 2004. The work of [39] is further developed in [44] and [45], where a partially linear LS-SVM (PL-LSSVM) is proposed to improve the performance of existing black-box models when it is evident that linear regressors are present. The best reported simulation

benchmark of 0.26 mV is presented in [46] using a polynomial nonlinear state-space (PNLSS) model. Identification of the system in another block-like form referred to as a linear fractional representation (LFR) is performed in [47]. The linear portion is estimated using standard identification techniques while the static nonlinearity is modelled using piecewise affine (PWA) identification techniques. Simulation metrics are reported as a measure of model fit, but the RMSE is not reported. An initialization scheme for nonlinear state-space models is applied in [48]. Nonlinear dynamic system identification is transformed into an approximate static problem, where system dynamics and nonlinear terms are identified separately. In [49], the effective number of parameters in nonlinear identification benchmarks is studied for fair comparison of models from different classes. In their work, a neural network based nonlinear state-space (NN-NLSS) model was applied to the Silverbox. The work of [50] reduces PNLSS to a simpler LFR general block structure with an assumed polynomial static nonlinearity, dubbed Poly-LFR. In [51], an identification model based on a nonlinear autoregressive exogenous (NARX) model with filtered regressors is presented where the nonlinear regression is implemented with sparse Gaussian processes (GP-FNARX). Two schemes for reducing the effective number of parameters are presented in [52] based on fixed-size LS-SVM (FS-LSSVM): fixed-size ordinary least squares (FS-OLS) and fixed-size ridge regression (FS-RR) with truncation through singular value decomposition (SVD). In [53], unprecisiated fuzzy logic (FLu) is added to fuzzy logic to yield extended fuzzy logic (FLe) for addressing open-world problems (such as system identification). The work of [54] introduces a sub-class of recursive linear-in-the-parameters nonlinear filters known as recursive functional link polynomial filters (RFLiP). These filters are a type of universal approximator and are tested in simulation on the Silverbox, however the performance metric given is normalized mean square error (NMSE) instead of RMSE. In [55], online

system identification is performed by extending randomized single-hidden layer feedforward networks (SLFNs). A type of randomized SLFN known as a random vector functional link network (RVFL) is combined with the robust learning rule named normalized least mean M-estimate (NLMM). In [56], robust variants of fixed-size least squares support vector regression (FS-LSSVR) models are introduced to handle non-Gaussian noise and outliers.

Table 2.1: Silverbox benchmark results for test set simulation.

Author	Ref	Method	RMSE	Simulation	Training	Validation
Ljung	[38]	Neural network + cubic regressor	0.30	1 - 40495	40586 - 127410	N/A
Paduart	[40]	Physical block-oriented	0.71	Head	Body	N/A
Hjalmarsson	[41]	Physical block-oriented	0.96	1 - 40000	40001 - Body	N/A
Verdault	[42]	Weighted local linear state-space	1.3	1 - 40495	40585 - 49192	N/A
Sragner	[43]	Special multilayer perceptron	7.8	1 - 40000	40001 - 125000	N/A
Espinoza	[39]	FS-PLS	0.318	1 - 40000	40001 - 85000	85001 - End
Espinoza	[45]	PL-LSSVM	0.271	1 - 40000	40001 - 85000	85001 - End
Paduart	[46]	PNLSS	0.26	1 - 40700	40701 - Body	N/A
Pepona	[47]	PWA-LFR	N/A	1 - 40000	50001 - 52000	N/A
Marconato	[48]	Nonlinear state-space	0.34	1 - 40000	40001 - 80000	85001 - 125000
Marconato	[49]	NN-NLSS	0.33*	Unclear	Unclear	N/A
Van Mulders	[50]	Poly-LFR	0.35	Head	Body	N/A
Sabahi	[53]	Extended Fuzzy Logic (FLe)	9.10	Unclear	Unclear	N/A
Carini	[54]	RFLiP	N/A	Unclear	Unclear	N/A
Mattos	[55]	RVFL-NLMM	10*	1 - 40000	40001 - End	N/A
Santos	[56]	FS-LSSVR	0.76*	1 - 40000	40001 - 85000	85001 - End
Maroli	—	TCN (fair)	2.18	1 - 40585	40586 - 85000	85001 - 127500
Maroli	—	TCN (1-40000)	1.74	1 - 40000	40001 - 127500	N/A

Table 2.2: Silverbox benchmark results for test set one-step-ahead prediction.

Author	Ref	Method	RMSE	Test	Training	Validation
Sragner	[43]	Special multilayer perceptron	0.765	1 - 40000	40001 - 125000	N/A
Espinoza	[45]	PL-LSSVM	0.057	1 - 40000	40001 - 85000	85001 - End
Frigola	[51]	GP-FNARX	0.07*	Unclear	Unclear	N/A
Castro	[52]	FS-OLS and FS-RR	0.054	1 - 39000*	39001 - 91400*	91401 - End*
Mattos	[55]	RVFL-NLMM	3*	1 - 40000	40001 - End	N/A
Maroli	—	TCN (fair)	0.214	1 - 40585	40586 - 85000	85001 - 127500
Maroli	—	TCN (1-40000)	0.227	1 - 40000	40001 - 127500	N/A

2.4.2 Establishing a Fair Benchmark

Among published literature, there is disagreement on the subset of samples used for the test set as well as model selection methods. Information on the test, training, and validation subsets of the dataset is included in Table 2.1 and Table 2.2. Some ranges are based on assumptions left to the reader. The majority of works use samples 1-40000 to test their models and use samples 40001 onward for training. This is presumably because the introduction of the dataset details that the first part of the reference signal used to create the dataset consists of 40000 samples. A problem arises, however, when using samples 40001 to 40585 for model training: extrapolation performance is no longer fully tested. Careful examination of the dataset in Fig. 2.6 indicates that the white Gaussian noise of increasing amplitude continues approximately through sample 40585. Horizonal lines in the plot indicate the range of the test data; it can be seen that samples between 40001 and 40585 violate these bounds. Including them in the training set would limit the extent of extrapolation tested, although some extrapolation would still be tested since the limits of both the input and output data in samples 1 to 40000 exceed the limits in samples 40001 to 40585.

The last portion of the dataset, referred to as the tail of the arrow, is examined in similar detail in Fig. 2.7. This section includes samples 127500 to 131072 and is excluded in part or in whole from the majority of the examined works. It is not described in the dataset introduction but resembles the Gaussian noise of increasing amplitude from the test signal. For this reason, it should be left out of the training and validation sets to increase the identification challenge.

The work of [57] examines the Silverbox dataset before publication and uses samples 1 to 40700 for model testing. It is mentioned that models are extrapolated when the test set

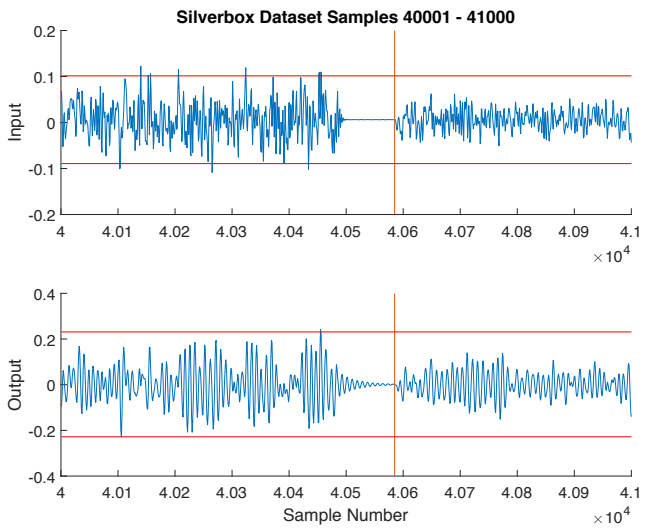


Figure 2.6: A detailed view of samples 40001 to 41000. Horizontal lines mark the range of the training data and a vertical line marks the proposed test/training division.

amplitude reaches the training set maximum of 0.1V. This sets precedence for use of the dataset in a way that tests extrapolation performance, which can only be done when using samples after 40586 for training. The works of [57], [38], [42], [46], and [47] are therefore the only ones examined that explicitly test their model extrapolation performance. Both [50] and [49] state that extrapolation is tested, but do not explicitly state the testing and training sample subsets. [48] states that extrapolation is tested as well, but is not explicit with the testing and training sample subsets. They leave the assumption that samples 1-40000 are for testing, which would not fully test extrapolation.

Model extrapolation makes the identification problem more difficult, so comparison of models performing extrapolation to non-extrapolating models is unfair. There is need for a consensus on the Silverbox dataset so that models can be compared for both their identification and extrapolation performance. For this reason, we set forth the following guidelines for benchmarking system identification methods using the Silverbox dataset:

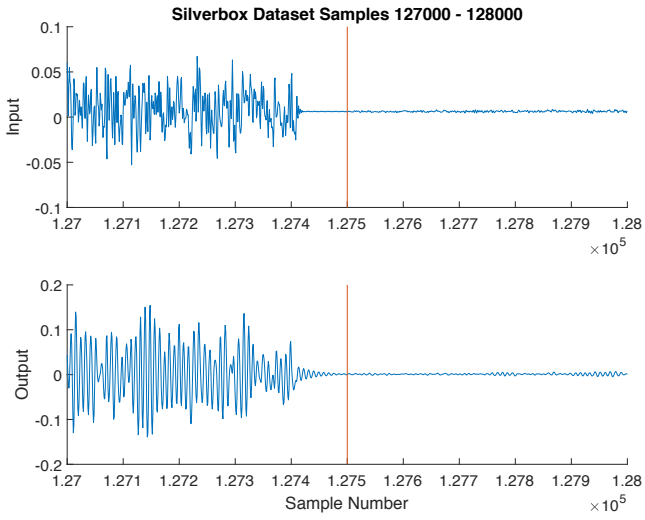


Figure 2.7: A detailed view of samples 127001 to 128000. Horizontal lines mark the range of the training data and a vertical line marks the proposed datset cutoff.

1. Model training/estimation should be performed on a subset A of samples 40586 to 127500. This is referred to as the training set.

2. Selection of the final model should be based on model performance over the samples of A^c (samples not in A), known as the validation set.

3. Testing should be performed on samples 1 to 40585, known as the test set. Both simulation and one-step-ahead RMSE metrics should be reported. This tests model identification with extrapolation.

4. Testing should be performed on samples 1 to 32473 and both simulation and one-step-ahead RMSE metrics should be reported. This strictly tests model identification performance without extrapolation.

2.4.3 TCN Identification Performance

The TCN and FNN identification models were trained and evaluated adhering to the guidelines established in Section 2.4.2. All models were implemented using PyTorch [58] in Python with the TCN implemented as in [33]. The model formulations both contain many high level parameters, some of which were fixed to limit the search space of available models while keeping the comparison between the TCN and FNN fair. The number of input and output measurements available to the models was limited to 10 each ($n = 10$). Additionally, the Adam optimizer was used for training both models with ReLU activation functions and batch sizes of 512.

The following TCN model configurations were trained with a learning rate of 0.002 and with varying hidden node counts: L=1 K=6, L=2 K=3, and L=3 K=2. The configuration with L=2 and K=3 yielded the best performance and was chosen for reporting. Error metrics for 25 models of this configuration with varying numbers of hidden nodes in each kernel leg are shown in Fig. 2.8. Each model was trained for 1500 epochs, at which point further training was not beneficial. The epoch yielding the lowest validation RMSE for each model was selected to present error metrics for that model. The TCN with 22 hidden nodes yielded the best validation performance compared to other models, with its metrics shown in Table 2.3. Training, validation, and test metrics are reported based on OSA prediction. A trailing "E" represents model metrics with extrapolation, while "NE" represents metrics without extrapolation.

FNN models were trained with a learning rate of 0.001 for 7500 epochs, at which point further training was not beneficial. Network configurations having 1, 2, and 3 hidden layers with varying node counts were trained with no significant difference between configurations. As a result, metrics for an FNN with a single hidden layer are reported. Fig. 2.8 shows

metrics for all tested FNN models with a single hidden layer. The best results were obtained with 40 hidden nodes.

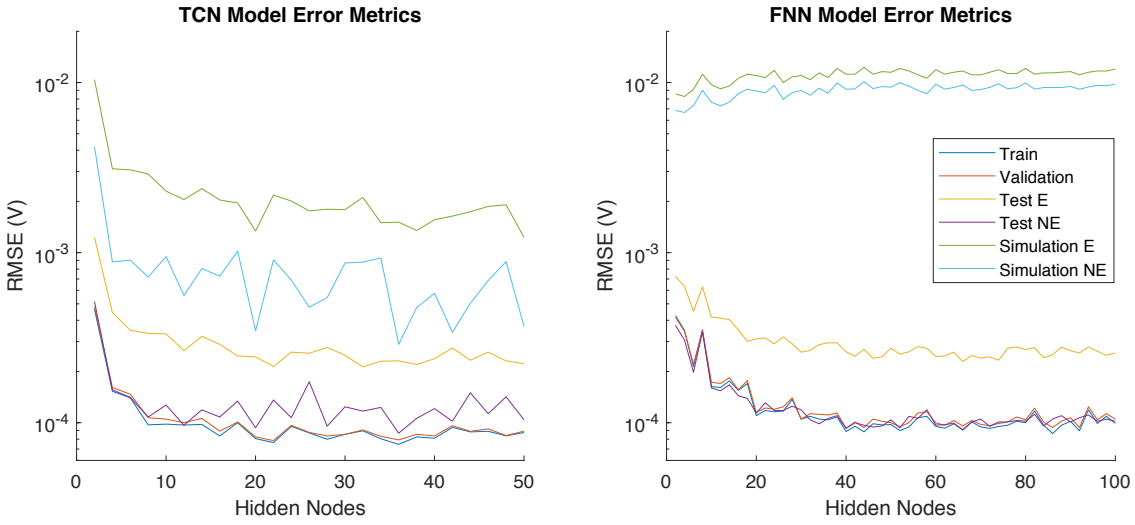


Figure 2.8: Error metrics for TCN models (left) with $K = 3$, $L = 2$ and FNN models (right) trained on samples 1-40585 having a single hidden layer of varying hidden node counts.

Table 2.3: Top performing model metrics.

Metric	TCN RMSE mV	FNN RMSE mV
Train	0.0764	0.0667
Validation	0.0785	0.0927
Test E	0.214	0.261
Test NE	0.136	0.0926
Simulation E	2.18	11.2
Simulation NE	0.907	9.13

Comparing the best FNN and TCN models, it is evident that the TCN holds a significant advantage over the FNN in simulation performance. The TCN model obtains a simulation

RMSE of 2.18 mV, while the FNN model only achieves an RMSE of 11.2 mV. The performance gap is similar when simulation without extrapolation is examined, with the TCN achieving 0.907 mV and the FNN 9.13 mV. Interestingly, training error, validation error, and test error are similar. This is true even with models having different hidden node counts as seen in the plots of Fig. 2.8. The plots also highlight the disparity in simulation performance between the TCN and FNN, as well as show trends evident when increasing model complexity. The simulation performance of the FNN is best with low hidden node counts, getting worse and then plateauing above 40 hidden nodes. The TCN enjoys the opposite effect, continually improving as the number of hidden nodes increases and eventually plateauing.

It should be noted that the FNN has a lower number of parameters given the same hidden node count. The top performing FNN with 40 hidden nodes has 881 parameters, while the top TCN with 22 hidden nodes has 4753 parameters. The TCN model, however, still outperforms the best FNN model in simulation performance when parameter counts are considered. The tested TCN model with 8 hidden nodes has 721 parameters and yielded an RMSE of 0.0971 mV for training, 0.107 mV for validation, 0.335 mV for testing, 0.108 mV for testing without extrapolation, 2.90 mV for simulation, and 0.719 mV for simulation without extrapolation.

Comparing results among literature is not straightforward due to the inconsistencies in model creation and testing. The TCN results presented in Table 2.3 can be compared fairly to the results of [38], [42], and [46], which achieve top simulation values of 0.30 mV, 1.3 mV, and 0.26 mV respectively. All of these models outperform the selected TCN on the full simulation with an RMSE of 2.18 mV, however the TCN's simulation without extrapolation RMSE of 0.907 mV is more competitive. This shows that the extrapolation performance of the TCN is poor in comparison to it's identification performance. The TCN

has the advantage of generalizing to any nonlinear system since it is a black box model. The results presented by [38] are for a grey box model, meaning that some information about the system was leveraged to improve the presented black box neural network model. In their case, a cubic regressor was used knowing that the system contained a cubic nonlinearity. The results presented by [42] are for a black-box model, which also presents a simulation without extrapolation RMSE on samples 1-25000 of 0.7 mV. The best available benchmark results found in [46] are from a black box PNLSS model, however it is noted that the superior performance is due to the correspondence between the PNLSS model and the structure of the silverbox with a cubic feedback in the output.

A subset of models examined in [38] best match the FNN models in this work and so effort was made to reproduce them. The best black box model from their work using a nonlinear autoregressive exogenous (NARX) structure and no custom regressor was a sigmoidal neural network with a single hidden layer of 75 hidden units. This model is most similar to the FNN in this work and achieved a simulation RMSE of 0.52 mV when trained in MATLAB. Our FNN implementation using PyTorch in Python was not able to attain these results, bringing about questions on differences in either the model platforms or test setup. We obtained a simulation RMSE of 10 mV while trying to reproduce their work, using the same training and test sets, ReLU activation (which outperformed sigmoidal activation), a batch size of 512, 7500 epochs, the Adam optimizer with a learning rate of 0.001, 75 hidden units, and the same regressor of $y[k-1], y[k-2], u[k], u[k-1], u[k-2]$.

The results of the TCN created under the established guidelines are presented in Table 2.1 and Table 2.2. For a more direct comparison to other literature, TCN and FNN models tested on samples 1-40000 and trained on samples 40001-127500 are also presented. Table 2.4 shows a direct comparison of these TCN and FNN models to the models created and tested

under the established fair guidelines. It is apparent that the fair guidelines create a more challenging simulation task. The presented models tested on samples 1-40000 were chosen using the validation RMSE in the same manner as the models tested under fair guidelines. Interestingly, both models had different hidden node counts than their fair counterparts: the TCN had 20 and the FNN had 82.

Table 2.4: Top model performance under different test sets.

Metric	Fair RMSE mV	1-40k RMSE mV
TCN Test E	0.214	0.227
TCN Test NE	0.136	0.109
TCN Simulation E	2.18	1.74
TCN Simulation NE	0.907	0.689
FNN Test E	0.261	0.249
FNN Test NE	0.0926	0.0807
FNN Simulation E	11.2	10.6
FNN Simulation NE	9.13	8.7

It is important to note that the models chosen based on validation data were not always the best performing models. The TCN with 50 hidden nodes had the best simulation with extrapolation performance of 1.23 mV while the TCN with 36 hidden nodes had the best simulation without extrapolation performance of 0.289 mV. This is approaching the noise level of 0.25 mV mentioned by [46]. This performance cannot be fairly reported since model selection would rely on the test data being known, however more diverse validation data better resembling the test data may lead to selection of these superior models. The validation data used for model selection was dissimilar from the test data since it was generated from multisines as opposed to Gaussian noise.

2.5 Summary and Conclusions

In this chapter, identification of nonlinear systems using neural networks was investigated. The use of TCNs to implement identification models was presented and then tested on the Silverbox dataset. Benchmark results were compared between TCN and standard FNN architectures, and both were compared to published results. Analysis of the dataset and published works yielded inconsistencies among testing methods, so guidelines for fair benchmarking on the Silverbox dataset were proposed. The selected TCN configuration yielded a simulation RMSE of 2.18 mV and a simulation without extrapolation RMSE of 0.907 mV on the Silverbox dataset using the established testing guidelines. The simulation results show that the TCN performs poorly at extrapolation compared to existing methods, however it is more competitive with strict simulation. One-step-ahead results of the TCN were comparative to the standard FNN, however the TCN presents as far superior to the FNN with simulation. Analysis of all TCN models showed that better simulation performance was achieved by models not selected based on validation performance. This indicates that more diverse validation data would lead to selection of a TCN model yielding top ranking simulation performance. In conclusion, the TCN is a powerful nonlinear system identification method useful for modelling black-box systems with competitive one-step-ahead prediction and high-ranking simulation performance.

Chapter 3: Equation Generation Methodology

The primary contribution of this work is a model analysis technique built into a structured framework for generating comprehensible equations describing discrete dynamical systems. The input to the framework is input-output data from a system, and the result is a mathematical model representing the system in the form of an equation. The resultant requirements are as follows: first and foremost, the equation must accurately represent the relationship of the input-output data. The equation must also be comprehensible, meaning it is concise, yields insight into the system operation, and takes user inclination into account. A concise system equation includes only relevant inputs and includes the minimum number of terms needed to meet accuracy requirements. The inputs and functional relationships included in the system equation should give insight into the dynamics of the system and help relate the system to other like systems. Anyone using the framework should also have some degree of control over the system equation format so that they can impart prior knowledge of the system on the process.

At a high level, the framework accomplishes two main tasks. First, a model of the input-output data is created in the form of a TCN. Second, the model is analyzed through a form of sample-based sensitivity analysis to build a discrete dynamical system equation. The initial system equation model is the result of an in-depth sensitivity analysis of the TCN identification model. The TCN presented in Chapter 2 is trained on input-output data

pairs to create the identification model in this work, although any model can be used in the analysis process so long as it is accurate and represents a static mapping of inputs to outputs. This chapter presents the analysis technique, which is based on a decomposition of the initial identification model into less complex identification problems. The identification and analysis procedures are integrated into an automated framework for reliably generating equations from input-output data.

The overall framework is presented in Fig. 3.1. It begins with input-output data and ends in a refined system equation describing the functional relationship of the data. The central branch of the framework is comprised of four main processes. The first process is training of the identification model, followed by model excitation and then response analysis. The response analysis yields an initial equation which is refined in the final genetic algorithm tuning process. The key to the analysis is the interpretation of the identification model. The model is re-imagined as a sum of non-additively separable "product functions" in the spirit of the function decomposition (1.1). The term "product functions" is not standardized but rather liberally defined in this work. It is a key aspect in the analysis technique described in Section 3.1. Sample points from each of these product functions are generated through specific model excitations propagated through the model input. The sample points are analyzed to determine which product functions contribute to the model response and then curve fitting is used to estimate the contributing product functions. The sum of product functions becomes the initial estimated system equation. The first three processes of the framework's central branch encompass the sample-based sensitivity analysis and form the backbone of the framework. The remainder of the framework exists to supplement the analysis and add robustness.

After generating an initial system equation, the relevant inputs from the input-output data are known. The inputs that don't contribute to the output are removed from the identification model training data and it is retrained in attempt to improve accuracy by eliminating irrelevant information. After retraining, excitation and analysis are repeated to yield a new estimated system equation. The accuracy of the equation is checked against the original input-output data at this point. If accuracy is not satisfactory then it is possible that the model excitation was not exhaustive enough to illicit a useful response for the analysis. The number of excitation instances is expanded in this case and model analysis is repeated until the estimated system equation fits the input-output data. If successive analysis attempts fail to yield an accurate estimated equation, then the analysis ends. If the data is determined to fit the estimated system equation, then genetic algorithm tuning commences. Genetic algorithm tuning is the fourth step in the central branch of the framework and the final step of the analysis. The goal of the tuning process is to adjust equation parameters and coefficients to better model the original input-output data.

The result of the framework is ultimately a refined system equation that describes the discrete dynamical system from which the input-output data originated. The different processes in the framework are described in this chapter. Section 3.1 details the model interpretation that enables the analysis, specifically the re-imagining of the model as a sum of modified product functions. Identification model training is done as was described in Chapter 2. Section 3.2 describes the process for exciting the model while the significance of the excitations and response analysis process are detailed in Section 3.3. The genetic algorithm parameter tuning process is described in Section 3.4 and conclusions are presented in Section 3.5.

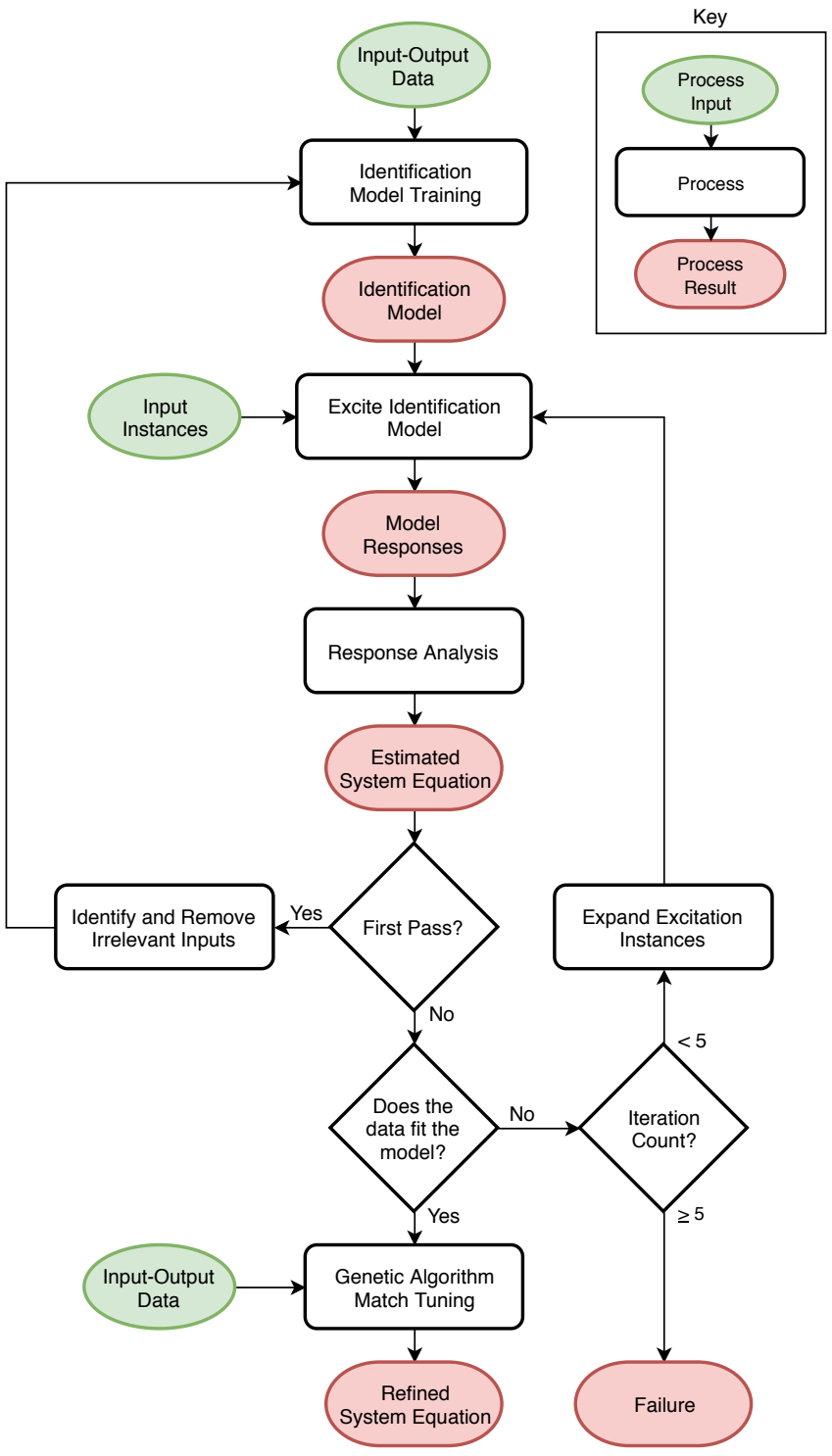


Figure 3.1: The overall framework of the equation generation methodology. The primary goal is converting input-output data into a comprehensible system equation.

3.1 Model Interpretation

The key to the presented methodology is the interpretation of the identification model. As discussed in Chapter 2, the model $\hat{F}$ produces an estimated system output $\hat{y}[k]$ using the model input $\bar{x}[k]$. Again, $\bar{x}[k]$ can be defined for a series-parallel model using (2.9), a modified parallel model using (2.10), or a custom identification model. To set up the analysis, a more in-depth view of the models is presented. The series-parallel model predicts an estimate $\hat{y}[k]$ of the current system output $y[k]$ using the l input values $u[k],...,u[k-l+1]$ and l output values $y[k-1],...,y[k-l]$. An input delay of $d=0$ is assumed for simplicity of exposition. Expanding the model input based on the dimension of u and y yields the following structure:

$$\bar{x}[k] = \begin{bmatrix} u_1[k] & \cdots & u_1[k-l+1] \\ \vdots & \ddots & \vdots \\ u_m[k] & \cdots & u_m[k-l+1] \\ y_1[k-1] & \cdots & y_1[k-l] \\ \vdots & \ddots & \vdots \\ y_p[k-1] & \cdots & y_p[k-l] \end{bmatrix}. \tag{3.1}$$

The parallel model predicts an estimate $\hat{y}[k]$ of the current system output $y[k]$ using the l input values $u[k],...,u[k-l+1]$. Expanding the model input in a similar fashion yields

$$\bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & \cdots & u_1[k-l+1] \\ u_2[k] & u_2[k-1] & \cdots & u_2[k-l+1] \\ \vdots & \vdots & \ddots & \vdots \\ u_m[k] & u_m[k-1] & \cdots & u_m[k-l+1] \end{bmatrix}. \tag{3.2}$$

For analysis of the identification model, the input is viewed in the generic form

$$\bar{x}[k] = \begin{bmatrix} x_1[k] & x_1[k-1] & \cdots & x_1[k-l+1] \\ x_2[k] & x_2[k-1] & \cdots & x_2[k-l+1] \\ \vdots & \vdots & \ddots & \vdots \\ x_q[k] & x_q[k-1] & \cdots & x_q[k-l+1] \end{bmatrix} \tag{3.3}$$

where $q=m+p$ for the series-parallel model and $q=m$ for the parallel model. It then follows that the input matrix $\bar{x}[k]$ contains ql elements.

The identification model is re-imagined as a sum of modified product functions. In the context of this work, a product function is defined such that it is not additively separable into functions of its arguments [59]. The product functions must contain unique sets of arguments and may only be constant in the special case where there are no arguments, $f(\emptyset) = c \neq 0$. Examples are $f(x_1, x_2) = x_1 \cdot x_2$ and $\sin(x_1) \cdot {x_2}^2$ since we cannot write these functions as $g(x_1, x_2) + h(x_1) + c$. Another example of a non-additively separable product function is $f(x_1) = x_1 + x_1^2$ since we cannot separate out a function of another variable or a non-zero constant. Product functions themselves are constituted from product functions multiplied together as well. It is trivial to show that any function can be written equivalently by a sum of product functions as described. A function of three elements, for example, can be broken down as follows:

$$F(x_1, x_2, x_3) = f_1(x_1, x_2, x_3) + f_2(x_1, x_2) + f_3(x_1, x_3) + f_4(x_2, x_3) + f_5(x_1) + f_6(x_2) + f_7(x_3) + f_8(\emptyset).$$

Note that a product function exists for each subset of the power set of the input arguments.

The product functions in this work are modified such that their value at an all-zero input is equal to zero. This prevents them from being true product functions as previously defined but is a key aspect of the model analysis to be examined further in Section 3.3.1. An example of a product function that needs modification to fit this requirement is e^x since $e^x = 1$ at $x = 0$. The modified product function form becomes $e^x - 1$. As another example, the modified product function form of $sin(x + c)$ is simply $sin(x + c) - sin(c)$.

All uses of product functions in this work refer to the modified variety. Formally, the system is re-imagined as the sum of all possible modified product functions

$$F(\bar{x}[k]) = \sum_{\bar{X}_i \in P(\bar{x}[k])}^{|P(\bar{x}[k])|} f_i(\bar{X}_i) \tag{3.4}$$

where all ql elements of the input $\bar{x}[k]$ are considered function arguments. The sum of functions now contains a generic function f_i corresponding to each unique combination of input arguments $\bar{X}_i$, yielding a total of 2^{ql} product functions. As noted before, the unique combinations of arguments of $\bar{x}[k]$ are obtained from its power set. The identification model $\hat{F}$ approximates F as a sum of $\hat{f}$ approximating f. Many of the product functions will be zero in practice, so identifying the non-zero product functions will give an accurate estimate of the system function. Through targeted excitation of the identification model and analysis of the responses, each non-zero product function is determined to generate an equivalent function to the model.

3.2 Model Excitation

The model is purposefully excited in such a way as to extract information about the constituent product functions. The model is viewed as a static mapping from l inputs belonging to $\mathbb{R}^q$ to a single output $\hat{y}[k]$. Let $\bar{x}_i$ be a unique instance i of $\bar{x}[k]$ with elements x_s from $s = 1, ..., ql$

$$\bar{x}_i = \begin{bmatrix} x_1 & x_2 & \cdots & x_l \\ x_{l+1} & x_{l+2} & \cdots & x_{2l} \\ \cdots & \cdots & \ddots & \cdots \\ x_{(q-1)l+1} & x_{(q-1)l+2} & \cdots & x_{ql} \end{bmatrix} \tag{3.5}$$

where each element x_s can be either 0 or non-zero. The non-zero elements hold a value of 1 in this work, and $\bar{x}_i$ is used as an element-wise multiplicative mask to nullify particular model inputs. Since there are ql elements in $\bar{x}_i$, it follows that there exist 2^{ql} unique combinations for $\bar{x}_i$. Each excitation mask $\bar{x}_i$ corresponds to a $f_i(\bar{X}_i)$ in (3.4) where the non-zero elements of $\bar{x}_i$ correspond to the members of $\bar{X}_i$. For example, the $\bar{x}_i$ with non-zero inputs $x_1, x_2, ...$ corresponds to the $\bar{X}_i$ containing $x_1[k], x_1[k-1], ...$ Each mask $\bar{x}_i$ is used in the determination of each f_i.

Let the output of the model corresponding to input i be calculated as

$$\hat{y}_i = \hat{F}(\bar{x}_i). \tag{3.6}$$

Let X be the set containing all combinations of $\bar{x}_i$, and let $\hat{Y}$ be the set containing the corresponding outputs as follows

$$\begin{aligned} X &= \left\{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{2^{ql}-1}\right\} \\ \hat{Y} &= \left\{\hat{y}_0, \hat{y}_1, \ldots, \hat{y}_{2^{ql}-1}\right\}. \end{aligned} \tag{3.7}$$

The model output $\hat{y}_i$ is easy to calculate since it is simply the result given input $\bar{x}_i$. Analysis of the model response is detailed in Section 3.3 as this section is focused on the model excitation preceding analysis, however a brief overview of analysis objectives is needed. The goal of decomposing the identification model into product functions is to create several easy identification problems rather than trying to solve a very large and potentially difficult identification problem. The overall model function $\hat{F}$ is a sum of all $\hat{f}_i(\bar{X}_i)$. The excitations are created to sample the product functions so that they can be estimated individually as smaller identification problems.

For ease of exposition, $\hat{f}_i$ will represent a numeric resultant of the function $\hat{f}_i(\bar{X}_i)$. The product function values $\hat{f}_i$ are determined through analysis of the model response. One of the primary analysis tasks is to obtain point pairs from each product function in the form $\{\bar{x}_i, \hat{f}_i\}$ for fitting a function to $\hat{f}_i(\bar{X}_i)$. The model excitation described up to this point is sufficient for yielding a single numeric value per product function for each $\bar{x}_i$. More data points per product function are needed for analysis, which is accomplished through multiples of each excitation mask $\bar{x}_i$. Let $\gamma^{(v)}$ be the v^{th} multiplier for the input, defined as a matrix of equal dimension to $\bar{x}_i$ whose elements are individually drawn from a uniform distribution over the model input range. Let Γ be referred to as the sweep set, containing r unique multipliers

$$\Gamma = \left\{\gamma^{(1)}, \ldots, \gamma^{(r)}\right\}. \tag{3.8}$$

Let $\hat{y}_i^{(v)}$ represent the model output for the input multiplied by $\gamma^{(v)}$ using the Hadamard product (element-wise matrix multiplication) as shown:

$$\hat{y}_i^{(v)} = \hat{F}\left(\gamma^{(v)} \circ \bar{x}_i\right). \tag{3.9}$$

These input-output point pairs can be grouped into sets in a similar fashion to X and $\hat{Y}$. Let $X^{(v)}$ contain all $\gamma^{(v)} \circ \bar{x}_i$ and let $\hat{Y}^{(v)}$ contain all corresponding $\hat{y}_i^{(v)}$ for multiplier v:

$$\begin{aligned} X^{(v)} &= \left\{\gamma^{(v)} \circ \bar{x}_0, \gamma^{(v)} \circ \bar{x}_1, \ldots, \gamma^{(v)} \circ \bar{x}_{2^{ql}-1}\right\} \\ \hat{Y}^{(v)} &= \left\{\hat{y}_0^{(v)}, \hat{y}_1^{(v)}, \ldots, \hat{y}_{2^{ql}-1}^{(v)}\right\} \end{aligned}. \tag{3.10}$$

These input-output values are used in the analysis to obtain a point pair $\{\gamma^{(v)} \circ \bar{x}_i, \hat{f}_i^{(v)}\}$ for each product function. It follows that using Γ, r corresponding input-output sets are obtained that provide enough information to extract r points per product function.

3.3 Response Analysis

After the identification model is trained and excited, the responses are analyzed to estimate the equation of the system that it represents. The analysis method is based on examining the effect of the carefully designed input instances on the output of the model. The input instances allow sampling of the model in a decomposed form, aiding in determining which product functions are present in the overall model function. After isolating the most significant product functions, their equations are estimated using standard curve fitting techniques. The analysis yields an equation that is approximate to the model, but in the form of (2.3). This sheds light on the black box identification model and can provide valuable information for system analysis, control, software verification, and a breadth of related fields. The analysis methodology is presented in Section 3.3.1 with a step-by-step example walkthrough in Section 3.3.2.

3.3.1 Methodology

The first step in the analysis is to determine the structure of $\hat{F}$ by first determining which product functions are significant. Significant product functions are later estimated. By definition of the product functions, the significant inputs to the model are also revealed. It is recommended to obtain input-output set pairs $\{X^{(v)}, \hat{Y}^{(v)}\}$ using an initial sweep set with $r \geq 25$ for determining which product functions are significant. The analysis is detailed using the default set pairs $\{X, \hat{Y}\}$ and is to be repeated for every additional pair. As mentioned earlier, each input-output set pair yields a single point $\{\bar{x}_i, \hat{f}_i\}$ for each product function. Analysis begins with examination of $\hat{Y}$, which contains responses of the overall model. The overall model may be complex and consist of many different functional relationships between any number of inputs. The concept of product functions is used to decompose the model into less complex fragments in order to more easily identify it. Starting with the base case, if $\hat{F}$ itself was equivalent to a single product function then all elements of $\hat{Y}$ would be identical except for one $\hat{y}_i$. It would follow that $\hat{F} = \hat{f}_i(\bar{X}_i)$ for the corresponding i. For the breadth of scenarios where the model is not represented by a single product function, each $\hat{y}_i$ may be the equivalent resultant of a sum of product functions. In this case, each $\hat{y}_i$ may be "contaminated" with additive elements that must be removed to isolate the contribution of each product function to $\hat{F}$. This is a concept found in [25] that we extend.

To determine what must be removed from each $\hat{y}_i$, we carefully examine its makeup. While $\hat{y}_i$ is numerically determined using (3.6), it is structurally defined as a sum of non-additively separable product functions with unique argument combinations as seen in (3.4). The input instances $\bar{x}_i$ are designed so that product function responses $\hat{f}_i$ can be isolated. Each $\bar{x}_i$ is used to determine if a product function of it's non-zero elements $\hat{f}_i(\bar{X}_i)$ makes a significant contribution to the model output. Since we are examining all combinations

of $\bar{x}_i$, we are in turn examining product functions of all input combinations. This method of excitation is meaningful because of the product function modification enforcing a zero output with a zero input, allowing each excitation to target a specific product function. Every zero input nullifies the response of all product functions containing that input. This concept can be used to isolate individual product functions.

Recall the breakdown of a generic function of three inputs from Section 3.1. There exist $2^3 = 8$ combinations of $\bar{x}_i$, which evaluate as

$$\begin{aligned}
F(0,0,0) &= f_0(\emptyset) \\
F(x_1,0,0) &= f_3(x_1) + f_0(\emptyset) \\
F(0,x_2,0) &= f_2(x_2) + f_0(\emptyset) \\
F(0,0,x_3) &= f_1(x_3) + f_0(\emptyset) \\
F(x_1,x_2,0) &= f_6(x_1,x_2) + f_3(x_1) + f_2(x_2) + f_0(\emptyset) \\
F(x_1,0,x_3) &= f_5(x_1,x_3) + f_3(x_1) + f_1(x_3) + f_0(\emptyset) \\
F(0,x_2,x_3) &= f_4(x_2,x_3) + f_2(x_2) + f_1(x_3) + f_0(\emptyset) \\
F(x_1,x_2,x_3) &= f_7(x_1,x_2,x_3) + f_6(x_1,x_2) + f_5(x_1,x_3) + f_4(x_2,x_3) \\
&\quad + f_3(x_1) + f_2(x_2) + f_1(x_3) + f_0(\emptyset).
\end{aligned}$$

It can be seen that any product function having a single zero input is equal to zero (note that $f(\emptyset)$ technically does not have any zero inputs). With this constraint, the value of the function with an all zero input becomes $f(\emptyset)$. The value of the function with a single non-zero input becomes the sum of $f(\emptyset)$ and the product function containing the non-zero input. Since $f(\emptyset)$ is known, the product function containing the non-zero input can be

calculated by subtraction. All product functions can be calculated in this manner:

$$
\begin{aligned}
f_0(\emptyset) &= F(0,0,0) \\
f_1(x_3) &= F(0,0,x_3) - f_0(\emptyset) \\
f_2(x_2) &= F(0,x_2,0) - f_0(\emptyset) \\
f_3(x_1) &= F(x_1,0,0) - f_0(\emptyset) \\
f_4(x_2,x_3) &= F(0,x_2,x_3) - f_2(x_2) - f_1(x_3) - f_0(\emptyset) \\
f_5(x_1,x_3) &= F(x_1,0,x_3) - f_3(x_1) - f_1(x_3) - f_0(\emptyset) \\
f_6(x_1,x_2) &= F(x_1,x_2,0) - f_3(x_1) - f_2(x_2) - f_0(\emptyset) \\
f_7(x_1,x_2,x_3) &= F(x_1,x_2,x_3) - f_6(x_1,x_2) - f_5(x_1,x_3) - f_4(x_2,x_3) \\
&\quad - f_3(x_1) - f_2(x_2) - f_0(\emptyset).
\end{aligned}
$$

The key observations from this example are that 1) a point value for each product function can be determined given any input $\bar{x}_i$ and that 2) the calculation is recursive in nature. A point value is obtained for each product function since $F(\cdot)$ evaluates to a real number and the recursive nature allows all product function values to be computed starting from f_0. It can be seen in the example that given any input to F, values for $f_0, ..., f_7$ can be calculated. Extending this to the 2^{ql} model inputs $\bar{x}_i$, numeric values $\hat{f}_i$ for all product functions can be calculated using the model responses $\hat{y}_i$. This pattern is formally represented in the next paragraph.

Each product function is defined based on the following recursive relationship

$$\hat{f}_i(\bar{X}_i) = \hat{y}_i - g(P(\bar{X}_i) - \{\bar{X}_i\}) \tag{3.11}$$

where g is a helper function operating on a set of sets to isolate each $\hat{f}_i(\bar{X}_i)$ by removing the contribution of other product functions:

$$g(A) = \sum_{\alpha=1}^{|A|} \left(\hat{f}_i(\bar{X}_i) \,|\bar{X}_i \in A\right). \tag{3.12}$$

The value of g is the sum of the product functions of argument combinations in the power set of $\bar{X}_i$ except for $\bar{X}_i$ itself since $\hat{f}_i(\bar{X}_i)$ is being determined. Essentially, $\hat{f}_i(\bar{X}_i)$ is the output of the model for the input $\bar{x}_i$ minus the "contaminant" product functions. This is recursive in nature since $\hat{f}_i(\bar{X}_i)$ depends on other product functions that have not yet been assigned a numeric value. For each $\hat{f}_i(\bar{X}_i)$ to be computed numerically, the product functions for all proper subsets of $\bar{X}_i$ must be computed first. The value of all $\hat{f}_i(\bar{X}_i)$, can be calculated by starting with the base case where $\bar{X}_i$ is the empty set. This special case where all elements of $\bar{x}_i$ are equal to 0 is assigned to $i = 0$. We define the corresponding $\hat{y}_0$ as the network bias, which numerically represents $\hat{f}_0(\bar{X}_0)$ where $\bar{X}_0 = \emptyset$. Using $\hat{f}_0(\emptyset)$, we can calculate all $\hat{f}_i(\bar{X}_i)$ for $\bar{X}_i$ containing one element as $f_i(\bar{X}_i) = \hat{y}_i - \hat{f}_0(\emptyset)$. Extending this pattern and following (3.11), we obtain the value of each product function $\hat{f}_i$ for the input $\bar{x}_i$.

Since multiple point values from each product function must be obtained, the set of multipliers Γ is used to perform a sweep over the input and this process is repeated. Recall that model excitation using Γ yields $X^{(v)}$ and $\hat{Y}^{(v)}$ for each element $\boldsymbol{\gamma}^{(v)}$ in Γ. The r values in Γ yield r points per product function in the form of $\{(\boldsymbol{\gamma}^{(v)} \circ \bar{x}_i, \hat{f}_i^{(v)}\}$ after the recursive analysis. The sweep length must therefore be selected to provide sufficient coverage of the input space while considering computational burden, since complexity grows linearly with r. The point pairs will later be used to estimate each product function through curve fitting, however it must be emphasized that not all product functions need to be estimated.

Since the process of extracting a single point pair per product function is computationally burdensome, a smaller number of points are first obtained for each product function to determine which ones contribute significantly to the model. This is accomplished using a smaller sweep set referred to as the initial sweep set. Product functions that contribute significantly are later sampled in detail using a much larger sweep set referred to as the

detailed sweep set. Product functions that are deemed insignificant are discarded to save computational time. Let $z_i^{(v)}$ be defined as the magnitude of the isolated contribution

$$z_i^{(v)} = |f_i^{(v)}|. \tag{3.13}$$

The average magnitude of each product function across the r samples is simply

$$z_i = \sum_{v=1}^{r} \frac{z_i^{(v)}}{r}. \tag{3.14}$$

The set Z contains all z_i analogous to $\bar{X}$ and $\hat{Y}$

$$Z = \{z_0, \ldots, z_{2^q-1}\}. \tag{3.15}$$

The magnitude contribution set Z is used to help identify significant product functions. To aid in this task, another set S is introduced in the spirit of Sobol indices [28], containing the variances s_i across the samples of each product function:

$$S = \{s_0, \ldots, s_{2^q-1}\}. \tag{3.16}$$

The identification model at this point has been reimagined as a sum of product functions, although realistically most of those product functions will not contribute to the overall model. To simplify the extracted equation, insignificant product functions are discarded and $\hat{F}$ is represented as a sum of significant product functions. The sets Z and S respectively hold information on the average magnitude contribution and the variance of each product function. Information from both are used since they hold advantages in different scenarios. Large constants or functions with large responses in a narrow range will have low variance but high magnitude and so are better detected by examination of Z. On the other end of the spectrum, functions with small responses in a narrow range will have both low magnitude and variance. These functions are in practice better identified by examination of S.

Searching through Z and S to find the indices with large magnitudes and/or variances will yield the significant product functions. Product functions corresponding to $z_i > \varepsilon_z$ and/or $s_i > \varepsilon_s$ are determined to be significant, where $\varepsilon_z, \varepsilon_s > 0$ are the thresholds for significance. The product functions that are summed to yield $\hat{F}$ are defined based on the level of accuracy desired in the approximation, which can be adjusted with $\varepsilon_z, \varepsilon_s$. The analysis as implemented in this work selects a product function if either $z_i > \varepsilon_z$ or $s_i > \varepsilon_s$. The approximation can be represented by

$$\hat{F} \approx \sum_{i=0}^{2^{ql}-1} \left(\hat{f}_i(\bar{X}_i) \,|\, z_i > \varepsilon_z \vee s_i > \varepsilon_s \right). \tag{3.17}$$

Average magnitude and variance contribution were chosen as selection criteria in this work, however other metrics can be substituted or added based on the nature of the product functions in question. A function with a relatively low magnitude response at most input values with an extremely large response in a narrow input range may better be detected with *max*. The estimated area under the curve formed from the input-output point pairs can also be used as a measure of product function contribution, or weight can be assigned to product function response based on the input distribution if it is heavily biased. Although ε_z and ε_s are simply constants, they are defined strategically to aid in understanding of the analysis. ε_z is defined as a fraction of the total average magnitude minus the constant magnitude $\sum Z - z_0$ and ε_s is likewise defined as a fraction of the total variance $\sum S$. In this way, the desired percent contribution of each product function to the total average magnitude or total variance can be used as a threshold. The constant magnitude z_0 is subtracted from the sum of magnitudes so that a large constant doesn't inhibit product function detection.

The generic product functions $\hat{f}_i(\bar{X}_i)$ contributing most to $\hat{F}$ are known at this point along with their corresponding arguments and samples, though the functions themselves are unknown. The product functions are estimated by curve fitting, however the r samples

from each product function used for determining the overall structure of $\hat{F}$ are likely not numerous enough. This is due to a few key reasons. The number of samples needed to determine if a product function is present is far less than the samples needed to fit a curve to the function, especially for higher dimensional data. The model excitation and response analysis is computationally expensive, so it is desirable to use the minimum number of excitations possible to determine the product functions in the model. Potential product functions that do not exist in the model have no need to undergo curve fitting, and so their sample points are not used in this process. To decrease overall computational time while avoiding wasteful computations, the minimal r samples are used to initially detect product functions. Product functions determined to be present are then targeted to produce additional response samples for curve fitting using a larger detailed sweep set.

The relevant inputs to $\hat{F}$ are obtained to produce targeted excitations that yield samples for curve fitting. Each $\bar{X}_i$ where $z_i > \varepsilon_z$ or $s_i > \varepsilon_s$ represents a set of relevant inputs. Obtaining samples from each product function is again a recursive process. For product function arguments $\bar{X}_i$, multiples of the input instance $\bar{x}_i$ and multiples of all input instances pertaining to subsets of the power set of $\bar{X}_i$ must be generated. The responses are analyzed using (3.11) to yield samples for curve fitting. At this point, the samples belong to product functions of the decomposed model, which will generally be much simpler functions than the original model. Because of this, standard curve fitting techniques can be used [10] to estimate each product function. For this work, the `curve_fit()` method of SciPy in Python is used to perform curve fitting [60].

At this point, we have determined an equation approximate to the model as a sum of product functions. For a well trained identification model with a response that closely matches its corresponding dynamical system, the approximated function can be assumed to

represent the system equation. The last step in our analysis is to relate the approximation that we obtained in (3.17) to the system input-output model in (2.2). To do this, we replace $\hat{F}$ and all elements of $\bar{x}_i$ in (3.17) with their analogous counterparts from (2.2); $y[k]$ replaces $\hat{F}$ and elements of $\bar{x}[k]$ replace elements of $\bar{x}_i$. We now have a discrete dynamical system input-output model that is approximate to our identification model.

3.3.2 Example Walkthrough

The example system in this section is intentionally simple to aid in explanation of the analysis methodology. Assume that the following discrete dynamical system is to be identified using only input and output measurements:

$$y[k] = -0.5u[k-1] + 0.5y[k-2]^2 + 0.5u[k]y[k-1].$$

For simplicity of the example, it is known that the system equation is of the form

$$y[k] = F(u[k], u[k-1], y[k-1], y[k-2]).$$

A series-parallel identification model is selected and represented by the TCN as follows:

$$\hat{y}[k] = \hat{F}(\bar{x}[k])$$

where

$$\bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] \\ y_1[k-1] & y_1[k-2] \end{bmatrix}.$$

The TCN is typically trained at this point using the input-output data as described in Chapter 2, however the actual system equation is used as the model to illustrate the analysis methodology. This is an ideal scenario where the model perfectly matches the system.

Analysis begins with creation of the input instances $X = \{\bar{x}_0, \ldots, \bar{x}_{15}\}$ where the unique instances $\bar{x}_i$ of $\bar{x}[k]$ are of the form

$$\bar{x}_i = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}.$$

The model response $\hat{Y} = \{\hat{y}_0, \ldots, \hat{y}_{15}\}$ is obtained. Each input instance is shown in Table 3.2 with its corresponding index and the model response. The analysis seeks to represent the model $\hat{F}$ as a sum of estimated product functions, whose general form is determined by (3.4) as

$$\hat{F} = \hat{f}_0(\bar{X}_0) + \cdots + \hat{f}_{15}(\bar{X}_{15}).$$

A sample for each product function is obtained using $\hat{Y}$ along with the recursive relationship of (3.11) and (3.12) as follows:

$$\begin{aligned}
\hat{f}_0(\bar{X}_0) &= \hat{y}_0 \\
\hat{f}_1(\bar{X}_1) &= \hat{y}_1 - \hat{f}_0(\bar{X}_0) \\
\hat{f}_2(\bar{X}_2) &= \hat{y}_2 - \hat{f}_0(\bar{X}_0) \\
\hat{f}_3(\bar{X}_3) &= \hat{y}_3 - \hat{f}_0(\bar{X}_0) \\
\hat{f}_4(\bar{X}_4) &= \hat{y}_4 - \hat{f}_0(\bar{X}_0) \\
\hat{f}_5(\bar{X}_5) &= \hat{y}_5 - \hat{f}_1(\bar{X}_1) - \hat{f}_2(\bar{X}_2) - \hat{f}_0(\bar{X}_0) \\
\hat{f}_6(\bar{X}_6) &= \hat{y}_6 - \hat{f}_1(\bar{X}_1) - \hat{f}_3(\bar{X}_3) - \hat{f}_0(\bar{X}_0) \\
&\cdots \\
\hat{f}_{15}(\bar{X}_{15}) &= \hat{y}_{15} - \left(\sum_{\alpha=0}^{14} \hat{f}_\alpha(\bar{X}_\alpha) \right).
\end{aligned}$$

At this point, the product functions are usually sampled many times to determine which ones contribute significantly to the model output. The analysis calls for the use of an initial sweep set Γ, however the simplicity of this example only requires a single sample obtained using X and $\hat{Y}$. The average magnitude contribution z_i is simply $|\hat{f}_i|$ with only one sample, however Z would normally be computed using (3.13), (3.14), and (3.15). The variance s_i is not used since a single sample has no variance. All $\hat{f}_i$ and z_i are shown in Table 3.2.

It is evident in Table 3.2 that z_2, z_4, and z_6 are non-zero and therefore represent significant product functions. The insignificant z_i will likely be slightly above zero when the model is not ideal, so significance is established by exceeding the threshold ε_z. Using (3.17), the general structure of the estimated equation is determined

$$\hat{F} = \hat{f}_2(\bar{X}_2) + \hat{f}_4(\bar{X}_4) + \hat{f}_6(\bar{X}_6).$$

This is further broken down based on the definition of $\bar{X}_i$:

$$\hat{F} = \hat{f}_2(x_2) + \hat{f}_4(x_4) + \hat{f}_6(x_1, x_3).$$

This is then related back to the system input-output model in (2.2) to give the general form of the estimated system equation as a sum of product functions:

$$\hat{y}[k] = \hat{f}_2(u_1[k-1]) + \hat{f}_4(y_1[k-2]) + \hat{f}_6(u_1[k], y_1[k-1]).$$

At this point, the model would normally be retrained after removing irrelevant inputs, however it was assumed that irrelevant inputs were not visible to the network. All that remains is to estimate $\hat{f}_2$, $\hat{f}_4$, and $\hat{f}_6$. Typically, a detailed sweep set is created with a large number of multiplicands to compute the input-output point pairs using (3.9). For this example, we assume that $\Gamma = \left\{\gamma^{(1)}, \gamma^{(2)}, \gamma^{(3)}, \gamma^{(4)}, \gamma^{(5)}\right\}$ contains 5 values as follows:

$$\Gamma = \left\{ \begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} -0.5 & -0.5 \\ -0.5 & -0.5 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right\}.$$

In practice, Γ should contain many more values and the elements within each $\gamma^{(v)}$ should be different. When multiplied by the input instances in X, 5 sets of inputs $X^{(1)}, ..., X^{(5)}$ are created that yield 5 sets of outputs $\hat{Y}^{(1)}, ..., \hat{Y}^{(5)}$. Using the recursive relationship, 5 samples of each product function are obtained and shown in Table 3.1.

It is evident that $\hat{f}_2(x_2) = -0.5x_2$ and $\hat{f}_4(x_4) = 0.5x_4^2$ even with a low number of points, however a larger number of points is needed to properly fit the functions with confidence.

Table 3.1: Example walkthrough sample points.

i	$\hat{f}^{(1)}$	$\hat{f}^{(2)}$	$\hat{f}^{(3)}$	$\hat{f}^{(4)}$	$\hat{f}^{(5)}$
2	0.5	0.25	0	-0.25	-0.5
4	0.5	0.125	0	0.125	0.5
6	0.5	0.125	0	0.125	0.5

The elements of each $\gamma^{(v)}$ must also be different as evidenced by the example samples for $\hat{f}_6$. Without different elements, the samples for $\hat{f}_4$ and $\hat{f}_6$ would be identical and the curve fitting process would fail to differentiate them. With sufficient sampling, the product functions are properly identified to yield the equation of the original system.

Table 3.2: Analysis methodology example walkthrough.

i	$\bar{x}_i$	$\bar{X}_i$	$\hat{y}_i$	$\hat{f}_i$	z_i	$> \varepsilon_z$?
0	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$	$\emptyset$	0	0	0	
1	$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$	$\{x_1\}$	0	0	0	
2	$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$	$\{x_2\}$	**-0.5**	**-0.5**	**0.5**	**X**
3	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$	$\{x_3\}$	0	0	0	
4	$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$	$\{x_4\}$	**0.5**	**0.5**	**0.5**	**X**
5	$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$	$\{x_1, x_2\}$	-0.5	0	0	
6	$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$	$\{x_1, x_3\}$	**0.5**	**0.5**	**0.5**	**X**
7	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\{x_1, x_4\}$	0.5	0	0	
8	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\{x_2, x_3\}$	-0.5	0	0	
9	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$	$\{x_2, x_4\}$	0	0	0	
10	$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$	$\{x_3, x_4\}$	0.5	0	0	
11	$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$	$\{x_1, x_2, x_3\}$	0	0	0	
12	$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$	$\{x_1, x_2, x_4\}$	0	0	0	
13	$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$	$\{x_1, x_3, x_4\}$	1	0	0	
14	$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$	$\{x_2, x_3, x_4\}$	0	0	0	
15	$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$	$\{x_1, x_2, x_3, x_4\}$	0.5	0	0	

3.4 Genetic Algorithm Tuning

Once the iterative analysis process has yielded an estimated system equation that fits the original input-output data, genetic algorithm tuning is used to refine the estimate. The estimated system equation serves as a model in itself with various coefficients and parameters. At this step, the equation is a sum of product functions that have been individually determined via curve fitting. It is a good approximate of the input-output data relationship and so the equation parameters should be close to optimal. The genetic algorithm is used to make slight adjustments to the parameters of the equation as a whole since they were initially determined on a product function by product function basis.

Genetic algorithms (GA) are a class of optimization techniques inspired by the biological concept of evolution. Their goal is typically to optimize parameters in order to maximize a heuristic function using operators such as crossover and mutation. Operators are applied to a group of individuals known as a population to create a new population. Each successive population group is referred to as a generation. The work in [61] gives an overview of genetic algorithms and explains 'why' and 'when' they should be used as an optimization tool.

A brief overview of genetic algorithms is given in the context of this work. The genetic algorithm problem formulation begins with a definition of the individuals in a population. Each individual is represented as an array of equation parameters represented by floating point numbers. In genetic algorithm terminology, this array is the chromosome representing the individual. An example estimated system equation is shown as follows with the corresponding chromosome representation in Fig. 3.2.

$$F(x_1, x_2, x_3) = 0.50{x_1}^2 + 0.25x_2 - 1.00\sin(2.00x_3) + 0.75$$

0.50	0.25	-1.00	2.00	0.75

Figure 3.2: An example chromosome representing an estimated system equation.

The initial population is created from a set number of individuals with variations between them. These variations introduce population diversity, which is necessary for the genetic algorithm to explore a variety of potential solutions. In this work, the assumption is made that the parameters are near their optimal values since the estimated system equation has been verified against the original system input-output data. As a result, the parameters of individuals in the initial population are drawn from a distribution centered around the original estimate. A single distribution is defined for each parameter. A triangular distribution [62] is used in this work since it is conceptually simple and the equation parameters have user-defined minimum and maximum values. The triangular distribution has a lower limit a, upper limit b, and mode c. Let α be an original parameter estimate, α_{min} be the user-defined minimum value, and α_{max} be the user-defined maximum value. Each parameter is drawn from its corresponding distribution with $c = \alpha$, $a \geq \alpha_{min}$, and $b \leq \alpha_{max}$. The upper and lower distribution limits in this work are defined by

$$a = c - \frac{(c - \alpha_{min})}{3} \tag{3.18a}$$

$$b = c + \frac{(\alpha_{max} - c)}{3}. \tag{3.18b}$$

The divisor of 3 moves a and b closer to the mode and can be modified as desired. This is not necessary but does help to tighten the distribution. Another benefit of the triangular distribution is that it allows skew towards either a or b. This can help to introduce diversity

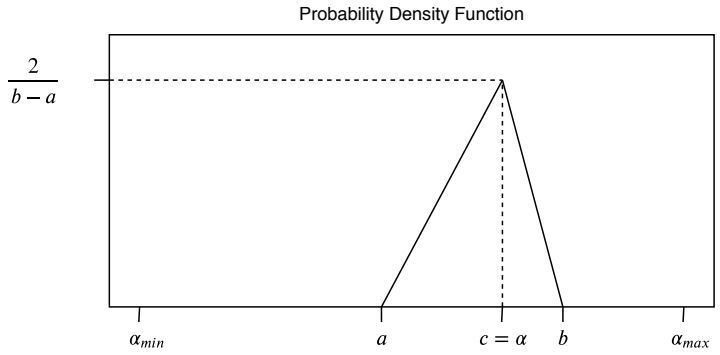


Figure 3.3: The probability density function of the triangular distribution.

to the population when c is biased heavily towards the minimum or maximum parameter values. The triangular distribution probability density function is shown in Fig 3.3.

The initial population of individuals is referred to as the first generation. After every population is created, it must be evaluated using a heuristic function designed to drive the behavior of individuals in the population. The heuristic function used in this work is the mean squared error (MSE) between the estimated function using an individual's parameters and the input-output data, with the goal being to minimize the MSE. The genetic operators of crossover and mutation are applied on the top performing members of each generation to yield an identically sized population of new individuals constituting the new generation. The crossover operation creates a new individual using a combination of the chromosomes from two individuals of the previous generation. The mutation operation makes a random change to the resultant of the crossover operation. Both the crossover and mutation operations are shown in Fig 3.4.

A cloning operation is included in the genetic algorithm tuning to preserve the best performing parameter set in future generations. The parameter set having the lowest MSE is directly copied to the next generation before the crossover and mutation operations begin.

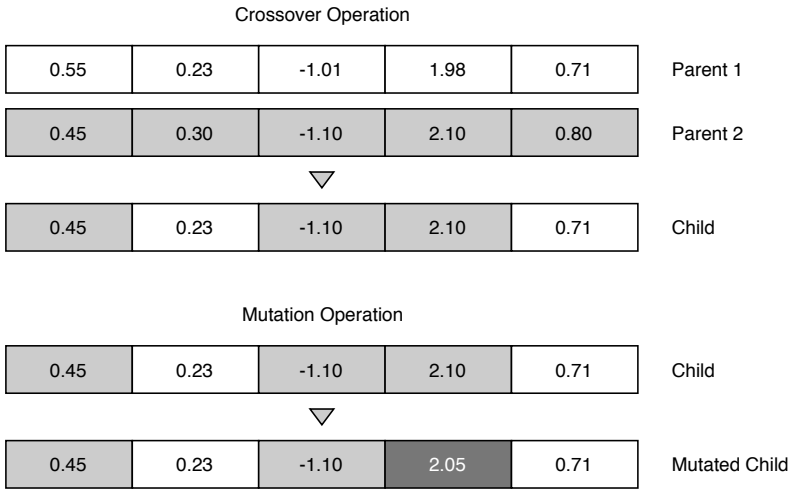


Figure 3.4: The crossover and mutation operations used to create members in successive generations.

When creating the initial population, the parameters of the original equation estimate are introduced as a population member. The top performing member of each generation is guaranteed to have an MSE that is less than or equal to the original individual with this formulation. The genetic algorithm tuning process is run for a set number of generations and then the best member of the final generation is used as the parameter set for the final refined system equation.

3.5 Summary and Conclusions

In this chapter, a comprehensive framework for generating discrete dynamical system equations from input-output data was introduced. The TCN identification model from Chapter 2 was combined with a new variant of sample-based sensitivity analysis. The analysis procedure operates on an imagined function decomposition of the model to break down the identification problem into multiple smaller problems. The smaller identification problems are less complex and easier to solve than the full identification problem, and the

sum of the smaller models yields the full model. The smaller models are estimated using standard curve fitting techniques and summed to yield a full equation of the original model.

The analysis yields an equation equivalent to black box identification models like the TCN, however it can operate on any input-output model. This chapter presented the theoretical basis of the analysis and built a framework around it to convert input-output data into a discrete dynamical system equation. The main contribution of this work is the analysis, with the full framework existing to facilitate practical application and to increase the accuracy of generated equations.

Chapter 4: Application Analysis and Examples

The system identification and sensitivity analysis techniques developed in this work are implemented for practical application in this chapter with comprehensive operational study and examples. The framework presented in Chapter 3 is implemented in Python and made available both in Appendix A and publicly at `https://github.com/Jmmaroli/eqn-gen`. The implementation follows the process diagram of Fig. 3.1 using 4 key functions. The first function implements the TCN system identification of Chapter 2 and the second function implements the sensitivity analysis from Chapter 3. The third function evaluates equation estimates produced by the analysis and the fourth function performs genetic algorithm tuning. The implementation is further detailed in Section 4.1, with Section 4.2 presenting computed results for the example walkthrough of Section 3.3.2. Section 4.3 presents a myriad of synthetic experiments designed to highlight important aspects of the analysis and Section 4.4 shows application of the framework to real systems. The implementation work is summarized and conclusions are drawn in Section 4.5.

4.1 Implementation

The Python implementation shown in Appendix A is used for all analyses in this section. The code structure reflects the framework structure as shown in Fig. 3.1, with 4 core functions created to represent framework processes. Identification model training is performed by the

function

```
model_dictionary = create_model( model_parameters,
                                 input_data,
                                 output_data,
                                 input_mask )
```

The argument `model_parameters` is a dictionary of parameters pertaining to the TCN model, `input_data` and `output_data` are arrays containing the input and output data respectively, and `input_mask` is the binary mask over $\bar{x}[k]$ used to remove insignificant inputs during model training. The input-output data arrays take the form

$$\texttt{input\_data} = \begin{bmatrix} x_1[k-N] & \cdots & x_q[k-N] \\ \vdots & \ddots & \vdots \\ x_1[k] & \cdots & x_q[k] \end{bmatrix} \tag{4.1a}$$

$$\texttt{output\_data} = \begin{bmatrix} y_1[k-N] & \cdots & y_q[k-N] \\ \vdots & \ddots & \vdots \\ y_1[k] & \cdots & y_q[k] \end{bmatrix} \tag{4.1b}$$

where N is the number of data points observed. This form facilitates application since data is often recorded at discrete time steps from the start of a file to the end. The function creates a TCN based on information defined in `model_parameters` and then formats the input-output data for training. After training is performed, the identification model is returned along with various metrics, all contained in `model_dictionary`.

The trained model is excited and then analyzed to yield an estimated system equation using the function

```
model_function, output_mask = analyze_model( analysis_parameters,
                                              model_dictionary,
                                              input_data,
                                              output_data,
                                              input_mask )
```

The argument `analysis_parameters` is a dictionary of parameters and settings specific to the analysis and `model_dictionary` is passed in from the output of `create_model()`. The `input_data` and `output_data` arrays are used as a reference for verifying that potential product functions improve the MAE of the estimate and `input_mask` serves the same purpose as it does for `create_model()`. The function returns a representation of the estimated equation, `model_function`, and a binary mask identifying important inputs to the model, `output_mask`. The output mask is used as the input mask for model retraining after an initial analysis is performed.

Determination of fit for estimated equations is handled by the function

```
metrics = evaluate_function( model_function,
                             input_data,
                             output_data )
```

where `model_function` is passed in from the output of `create_model()` and the `input_data` and `output_data` arrays are the same as before. The function returns `metrics`, which holds the MAE and RMSE of the estimated equation in comparison to the original input and

output data for each dimension of the output. The maximum and minimum response values are also returned.

The final core function performs genetic algorithm tuning of an estimated system equation to produce a refined system equation

```
model_function_tuned = tune_model( tuning_parameters,
                                   model_function,
                                   input_data,
                                   output_data )
```

The arguments are identical to `evaluate_function()` with the addition of `tuning_parameters` to specify the GA population size and number of generations. The output `model_function_tuned` represents a refined system equation with tuned parameters.

The 4 core functions are used to implement the overall framework, which is represented by the function `estimate_equation` described in Algorithm 1. This single function fulfills the framework that was laid out in Fig. 3.1. The initial model is trained and called `mdl_v1`, which is analyzed to yield an equation estimate `fcn_v1` and a mask of significant inputs to be used for model retraining: `msk_v2`. The retrained model is called `mdl_v2`, which is analyzed to yield `fcn_v2`. The estimates `fcn_v1` and `fcn_v2` are evaluated based on the original data to yield metrics `met_v1` and `met_v2`. The first two equation estimates are combined on a channel by channel basis to yield `fcn_v3`. If the fit of `fcn_v3` is poor, then it is recreated using a larger initial sweep set until the fit is deemed to be good. At this point, genetic algorithm tuning is performed on `fcn_v3` to yield the final generated equation `fcn_v4`.

Algorithm 1 Overall framework implementation

Input: model_parameters, analysis_parameters, input_data, output_data
Output: printed system equation
Function `estimate_equation()`**:**

```
mdl_v1 = create_model()
fcn_v1, msk_v2 = analyze_model( mdl_v1 )
met_v1 = evaluate_function( fcn_v1 )

mdl_v2 = create_model( msk_v2 )
fcn_v2 = analyze_model( mdl_v2 )
met_v2 = evaluate_function( fcn_v2 )

foreach output channel, c do
    if met_v2[c] better than met_v1[c] then
        fcn_v3[c] = fcn_v2[c]
        met_v3[c] = met_v2[c]
    else
        fcn_v3[c] = fcn_v1[c]
        met_v3[c] = met_v1[c]
    end
end

if fcn_v3 fit is poor then
    while fcn_v3 fit is poor do
        if iteration > limit then
            exit
        end
        // increase excitation instances in model_parameters
        // recreate fcn_v3 as done above without recreating mdl_v1

        iteration++
    end
end

fcn_v4 = tune_model( fcn_v3 )
print( fcn_v4 )
```

4.1.1 Model Creation

The function `create_model()` initializes and trains the model based on the parameters in `model_parameters`. The parameters most relevant to the structure of the TCN and the exposition of this work are described in Table 4.1; all other parameters are detailed in the

code. The TCN is created with L residual blocks set by `levels`, a kernel size of K set by `ksize`, and `nhid` units per kernel leg. A total of l samples are exposed to the network, set by `history`, and the number of training epochs is set by `epochs`.

Table 4.1: Relevant parameters in `model_parameters`.

Name	Description
`epochs`	The number of training epochs
`ksize`	Kernel size
`levels`	Number of residual blocks
`nhid`	Hidden units in each kernel leg
`history`	Samples exposed to the network

4.1.2 Model Analysis

The function `analyze_model()` accepts the model defined in `model_dictionary` and performs analysis based on the parameters in `analysis_parameters`. The parameters most relevant to the analysis process and the exposition of this work are described in Table 4.2; all other parameters are detailed in the code. The product function candidates are contained in a single object: `functions` (see `fitting_functions()` in Appendix A for the exact definition). The number of samples obtained for initial detection of product functions is set by `sweep_initial`, while the number obtained for detailed sampling and curve fitting of product functions is set by `sweep_detailed`.

If the MAE of the overall equation is not improved as a result of adding a product function and the contribution of the product function as measured by z_i and s_i is low, then the product function is omitted from the equation. The percent contribution threshold for potential inclusion is defined by `contrib_thresh`, while `contrib_thresh_omit` sets

the limit for conditional inclusion based on MAE such that `contrib_thresh_omit` > `contrib_thresh`.

Table 4.2: Relevant parameters in `analysis_parameters`.

Name	Description
`functions`	The product function candidates
`sweep_initial`	Number of initial samples
`sweep_detailed`	Number of samples for curve fitting
`contrib_thresh`	Percent contribution for inclusion
`contrib_thresh_omit`	Percent contribution for conditional inclusion

4.1.3 Template Functions

The model analysis fits function templates to the samples from each product function. These functions are defined in the `functions` argument of `analyze_model()`. For each detected product function, all template functions of the same dimension as the product function are fit to the samples for that product function. Metrics of fit for each template function are calculated and used to select the one that best resembles the product function. The metric used for evaluating product functions in this work is magnitude average error (MAE), which is conceptually simple. Popular metrics of fit such as R^2 cannot be used since the majority of template functions are nonlinear. Other metrics such as MSE or RMSE can be substituted for MAE if desired, however they are not as easily interpreted. The template functions included in the implementation of this work are shown in Table 4.3, however any number of template functions could be included.

In the most straightforward case, the template function yielding the best MAE is chosen to represent the corresponding product function. This generally yields a more accurate, albeit complex, overall system equation. Just as the TCN model can suffer from overfitting,

Table 4.3: List of template functions in the implementation.

Functions of 1 Variable
ax_1
$ax_1{}^2 + bx_1$
$ax_1{}^3 + bx_1{}^2 + cx_1$
$ax_1{}^4 + bx_1{}^3 + cx_1{}^2 + dx_1$
$ax_1{}^5 + bx_1{}^4 + cx_1{}^3 + dx_1{}^2 + ex_1$
$ax_1{}^2$
$ax_1{}^3$
$a\left(e^{bx_1} - 1\right)$
$a\sin(bx_1 + c) - \sin(c)$
$a\tanh(bx_1)$
$a\tanh(b(x_1 + c)) + d(x_1 + c) - (a\tanh(bc) + dc)$
Functions of 2 Variables
ax_1x_2
$ax_1x_2 + bx_1{}^2x_2 + cx_1x_2{}^2$
$ax_1x_2 + bx_1{}^2x_2 + cx_1x_2{}^2 + dx_1{}^3x_2 + ex_1{}^2x_2{}^2 + fx_1x_2{}^3$
$ax_1x_2 + bx_1{}^2x_2 + cx_1x_2{}^2 + dx_1{}^3x_2 + ex_1{}^2x_2{}^2 + fx_1x_2{}^3 + gx_1{}^4x_2 + hx_1{}^3x_2{}^2 + ix_1{}^2x_2{}^3 + jx_1x_1{}^4$
$ax_1\left(e^{x_2} - 1\right)$
$ax_2\left(e^{x_1} - 1\right)$
$a\tanh(bx_1 - c)\tanh(dx_2 - e) - a\tanh(-c)\tanh(-e)$
$a\tanh(bx_2 - c)\tanh(dx_1 - e) - a\tanh(-c)\tanh(-e)$
Functions of 3+ Variables
$ax_1x_2x_3$
$ax_1x_2x_3x_4$
$ax_1x_2x_3x_4x_5$

product function template selection must overcome overfitting as well. For example, samples of a product function that is clearly linear by inspection may yield a lower MAE when a higher order polynomial function is fit to them. To overcome this problem, a weight is assigned to each product function to give advantage to ones that are preferred. These are generally simpler functions, although the weights could be used to steer product function selection based on user preference (e.g. if the user prefers a linear equation unless accuracy is significantly effected).

To limit the space of function parameters and coefficients $a, b, c, ...$, reasonable bounds are specified for each parameter of each template function. The upper and lower bounds

of each parameter are arguments to the `curve_fit()` function of SciPy and greatly aid in quick parameter estimation. In the function $a\sin(bx_1+c)-\sin(c)$ for example, c can be bounded to $[-\pi,\pi]$ since sine is periodic. Other functions can be bounded based on reasonable expectation, such as limiting quartic or quintic polynomial coefficients to $[-1,1]$ if large output values are not expected.

It is important to note that the template functions do not absolutely have to meet the multiplicative requirement of product functions, although they must give a 0 output for a 0 input. For example, $f(x_1,x_2)=ax_1+bx_2$ could be used as an approximation of the true product function for $f(x_1,x_2)$. This greatly increases the flexibility of the framework by allowing many types of numerical approximations to be introduced. Of notable significance, finite differences can be used as template functions, allowing for the approximation of derivatives. Finite differences are mathematical expressions of the form

$$f(x+b)-f(x+a).$$

The first derivative can be approximated using finite differences as

$$f'(x)\approx\frac{f(x+h)-f(x)}{h},$$

and the second derivative can be approximated in similar fashion as

$$f''(x)\approx\frac{f(x+h)-2f(x)+f(x-h)}{h^2}.$$

Template functions can be constructed based on derivative approximations or any other numerical approximation to fit the samples of each product function. While the actual product functions are multiplicative in nature, curve fitting is an approximation itself and so multiplicity is not required after sampling the product functions.

4.2 Verbose Example

The example system in this section is the same as the system from Section 3.3.2. The purpose of this section is to illustrate the operation of the implemented framework while drawing parallels to the analysis theory. Recall that the following discrete dynamical system is to be identified using only input and output measurements:

$$y[k] = -0.5u[k-1] + 0.5y[k-2]^2 + 0.5u[k]y[k-1].$$

Again for simplicity of the example, it is known that the system equation is of the form

$$y[k] = F(u[k], u[k-1], y[k-1], y[k-2]).$$

A TCN is setup to be trained using input-output data from the system. The TCN is designed with L=1 and K=2 so that its effective history is greater than 2. Arguments outside the minimum required effective history are ignored since it is known they are not in the system equation. A series-parallel identification model is selected and represented by the TCN as follows:

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] \\ y_1[k-1] & y_1[k-2] \end{bmatrix}.$$

A large sequence of system input-output data is recorded where the system input is drawn from $U(-1,1)$. From this system input-output data, 25000 pairs of training data are gathered for the TCN. Each pair consists of a network input from the recorded sequence in the form of $\bar{x}[k]$, and the corresponding system output $y_1[k]$. The TCN is trained until the response is satisfactorily close to the modeled system. For this example, training was performed over 100 epochs using an initial learning rate of 0.002 with the Adam optimizer. The MAE of the trained network in reference to the input-output data is 3.638e-03.

Analysis begins with the creation of the excitation instances. The unique instances $\bar{x}_i$ of $\bar{x}[k]$ are of the form

$$\bar{x}_i = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$$

and it follows that there are 16 of them: $X = \{\bar{x}_0, \ldots, \bar{x}_{15}\}$. Table 4.4 shows the input instances and corresponding analysis information for each index in similar fashion to Table 3.2 of the example walkthrough in Section 3.3.2.

The excitation instances in X are used in conjunction with an initial sweep set Γ containing $r = 25$ multipliers to yield input sets $X^{(1)}, ..., X^{(25)}$. The inputs are propagated through the TCN model to yield output sets $\hat{Y}^{(1)}, ..., \hat{Y}^{(25)}$. The recursive relationship of (3.11) and (3.12) is used to obtain 25 samples $\hat{f}_i^{(1)}, ..., \hat{f}_i^{(25)}$ of each product function. These samples are used to compute the average magnitude contribution of each product function $Z = \{z_0, ..., z_{15}\}$ from (3.13), (3.14), and (3.15). The sample variances $S = \{s_0, ...s_{15}\}$ are computed as well for each product function in accordance with (3.16). The product functions where either $z_i > \varepsilon_z$ or $s_i > \varepsilon_s$ are used to estimate the overall model $\hat{F}$ as in (3.17). The analysis implementation dynamically defines ε_z and ε_s so that product functions contributing at least 5% to the total average magnitude or sum of variances are included in the model (`contrib_thresh=5%`).

As in Section 3.3.2, examination of Z and S show that the model takes the form

$$\hat{y}[k] = \hat{f}_2(u_1[k-1]) + \hat{f}_4(y_1[k-2]) + \hat{f}_6(u_1[k], y_1[k-1]).$$

At this point, the model would normally be retrained after removing irrelevant inputs, however it was assumed that irrelevant inputs were not visible to the network. All that remains is to estimate $\hat{f}_2$, $\hat{f}_4$, and $\hat{f}_6$. For this, a detailed sweep set Γ is created with $r = 1000$ multipliers to compute the input-output point pairs of each product function. The point

pairs are plotted in Fig. 4.1 for visual illustration. A curve is fit to the set of points for each function using SciPy to yield

$$\hat{f}_2\left(u_1[k-1]\right) = -0.497u_1[k-1]$$

$$\hat{f}_4\left(y_1[k-2]\right) = 0.498y_1[k-2]^2$$

$$\hat{f}_6\left(u_1[k], y_1[k-1]\right) = 0.464u_1[k]y_1[k-1].$$

The first round of the analysis is now complete and the resultant estimate of the system equation learned by the TCN is

$$\hat{y}[k] = -0.497u_1[k-1] + 0.498y_1[k-2]^2 + 0.464u_1[k]\,y_1[k-1].$$

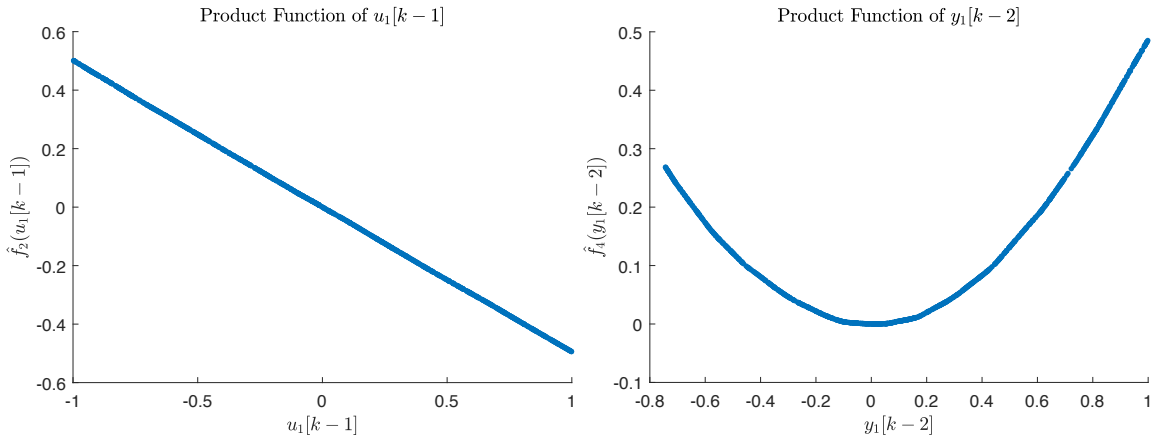
The equation is compared to the original input-output data and found to have a MAE of 5.098e-03. This is close to the MAE of 3.638e-03 for the TCN. Since the estimated equation is determined to satisfactorily fit the original system data, genetic algorithm tuning now begins. The refined system equation after 25 generations of tuning is

$$\hat{y}[k] = -0.500u_1[k-1] + 0.500y_1[k-2]^2 + 0.500u_1[k]\,y_1[k-1].$$

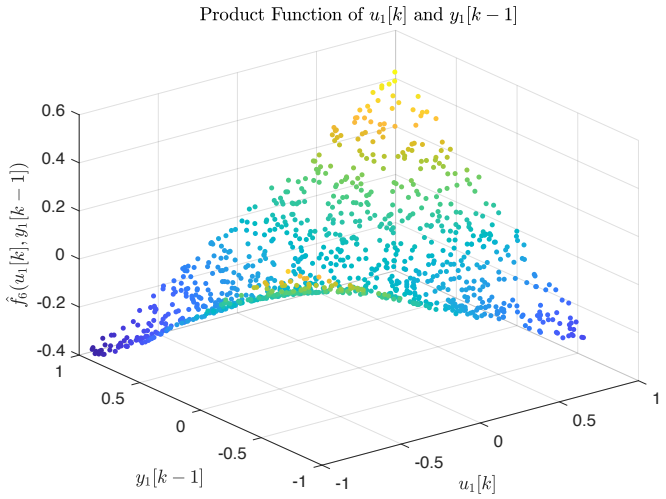
The parameters are truncated for brevity although evaluation is performed using the 64 bit floating point parameter values. The MAE is now only 5.204e-05, which is better than the original estimate and the TCN. The refined system equation is close to the original example equation, however it was extracted entirely offline from the trained TCN without the presence of the original system.

Table 4.4: Example construction of input instances and analysis values.

i	$\bar{x}_i$	$\bar{X}_i$	z_i	s_i	$> \varepsilon_z, \varepsilon_s$?
0	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$	$\{\emptyset\}$	0.0034	0.0000	
1	$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$	$\{x_1\}$	0.0087	0.0001	
2	$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$	$\{x_2\}$	**0.6821**	**0.3555**	**X**
3	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$	$\{x_3\}$	0.0122	0.0001	
4	$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$	$\{x_4\}$	**0.5875**	**0.2070**	**X**
5	$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$	$\{x_1, x_2\}$	0.0127	0.0002	
6	$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$	$\{x_1, x_3\}$	**0.3712**	**0.2202**	**X**
7	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\{x_1, x_4\}$	0.0154	0.0006	
8	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\{x_2, x_3\}$	0.0118	0.0001	
9	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$	$\{x_2, x_4\}$	0.0441	0.0055	
10	$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$	$\{x_3, x_4\}$	0.0342	0.0022	
11	$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$	$\{x_1, x_2, x_3\}$	0.0176	0.0008	
12	$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$	$\{x_1, x_2, x_4\}$	0.0122	0.0003	
13	$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$	$\{x_1, x_3, x_4\}$	0.0379	0.0055	
14	$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$	$\{x_2, x_3, x_4\}$	0.0126	0.0003	
15	$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$	$\{x_1, x_2, x_3, x_4\}$	0.0198	0.0009	



(a) (b)



(c)

Figure 4.1: Plots showing 1000 samples for each product function of significance as extracted from the TCN model: (a) $\hat{f}_2(u_1[k-1])$; (b) $\hat{f}_4(y_1[k-2])$; (c) $\hat{f}_6(u_1[k], y_1[k-1])$

4.3 Synthetic Experiments

The purpose of this section is to provide synthetic experiments that demonstrate the use of the equation generation framework; showing its robustness as well as its limitations. The experiments all include a defined system that is excited with 25000 samples of uniform random noise to yield input-output data. The framework then operates on the input-output data without knowledge of the system, starting with TCN system identification and continuing to analysis as in Fig. 3.1. The framework produces MAE results for 1) the initial TCN identification model, 2) the initial estimated equation, 3) the retrained TCN identification model, 4) the second estimated equation, and 5) the final GA tuned system equation.

4.3.1 Verbose Example Extension

The example of Section 4.2 is repeated, dropping assumptions about the system format to demonstrate typical use of the equation generation implementation. A longer effective history is desirable when analyzing an unknown system for the initial analysis so that all significant system inputs are detected. The effective history is therefore increased from 2 to 5. The input-output data is organized in the fashion of the series-parallel model and used to train a TCN in the format

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & u_1[k-2] & u_1[k-3] & u_1[k-4] \\ y_1[k-1] & y_1[k-2] & y_1[k-3] & y_1[k-4] & y_1[k-5] \end{bmatrix}.$$

The MAE of the TCN model is 6.081e-3. Analysis is performed with an initial sweep set of size 25 and a detailed sweep set of size 1000 to yield the initial estimated system equation

$$\hat{y}[k] = -0.497u_1[k-1] + 0.442y_1[k-2]^2 + 0.459u_1[k]\,y_1[k-1].$$

The initial equation has an MAE of 8.503e-3. The corresponding input mask used to nullify insignificant inputs for retraining is simply

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

The TCN is retrained without exposure to insignificant inputs, yielding an MAE of 2.833e-03. Analysis is performed again with the same sweep set sizes to yield

$$\hat{y}[k] = -0.507u_1[k-1] + 0.502y_1[k-2]^2 + 0.485u_1[k]\,y_1[k-1].$$

The second equation has a corresponding MAE of 3.893e-3, which is then tuned by genetic algorithm over 25 generations to yield the final refined system equation

$$\hat{y}[k] = -0.500u_1[k-1] + 0.500y_1[k-2]^2 + 0.501u_1[k]\,y_1[k-1].$$

This equation is the third and final equation produced by the analysis, and yields an MAE better than that of both TCN models and both estimated equations: 1.014e-4.

4.3.2 Noise Resilience

The discrete dynamical system state equations (2.1) can be extended to include process noise v and measurement noise w in their formulation:

$$\begin{array}{rcl} x[k+1] & = & f(x[k], u[k], v[k]) \\ y[k] & = & h(x[k], w[k]) \end{array}. \tag{4.2}$$

Both noise sources are assumed to be white Gaussian. Process noise is inherent in the evolution of the system state and effects the determination of future states. Measurement noise does not effect the system state but effects the output measurement $y[k]$. The effect of these noises on the equation generation framework is examined by extension of the system in Section 4.3.1.

4.3.2.1 Process Noise

Process noise v is introduced into the system additively as follows

$$y[k] = -0.5u[k-1] + 0.5y[k-2]^2 + 0.5u[k]y[k-1] + v[k].$$

Increasing levels of noise drawn from $\mathcal{N}(\mu, \sigma^2)$ are added to examine the effect of different levels of process noise on the system. Let $v[k] \sim \mathcal{N}(0, \sigma^2)$ where σ is varied between 0 and 0.16 in increments of 0.02. A significant amount of process noise will make the system itself unstable, which occurs when $\sigma > 0.16$. The MAE at each σ is reported in Fig. 4.2 for the 2 TCN models and 3 equation models generated by the framework, along with the magnitude average of the noise signal. Examination of the plot shows that the MAE values for all models in the framework lie just above the noise floor. Exact relationships are difficult to derive from the plot since the values are so close. The magnitude average of the noise signal is subtracted from the MAE of each model to yield the MAE above the noise floor, presented in Fig. 4.3. The TCN models generally have a similar or worse MAE to their extracted equations, while the GA tuned equation is superior to all models.

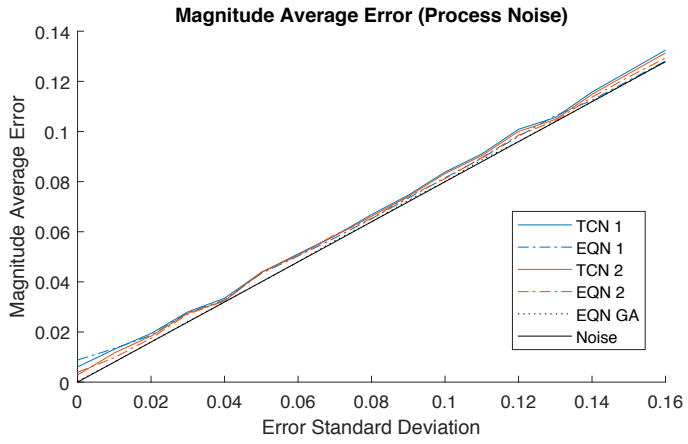


Figure 4.2: Effect of varied process error on the example system.

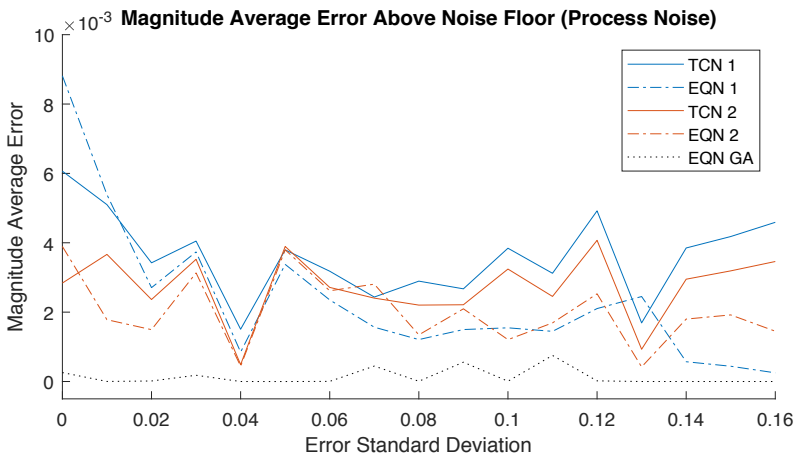


Figure 4.3: Effect of varied process error above the noise floor.

The framework demonstrates resilience to process noise, which can be attributed to the initial TCN model. Neural networks themselves are good at ignoring noise while learning relationships within data. The extracted equation is effectively generated after having much of the noise removed by the TCN, explaining its close performance to the TCN.

4.3.2.2 Measurement Noise

Measurement noise w does not effect the evolution of system states, but only the measurement itself. Let $\tilde{y}$ represent the recorded measurements of the system output used to build the identification model. The measurement is the result of additive measurement noise as follows

$$y[k] = -0.5u[k-1] + 0.5y[k-2]^2 + 0.5u[k]y[k-1]$$

$$\tilde{y}[k] = y[k] + w[k].$$

The TCN model is trained on pairs of $\bar{x}[k]$ and $\tilde{y}[k]$. Let $w[k] \sim \mathcal{N}(0, \sigma^2)$ where σ is varied between 0 and 2.20 in increments of 0.02. This represents a ratio of the magnitude average noise to the magnitude average measurement from 0% to 99%. The MAE at each σ is

reported in Fig. 4.4 for the 2 TCN models and 3 equation models, along with the magnitude average of the noise signal. The MAE of all models is around the noise level of the system, indicating strong resistance to measurement noise. A closer examination of the information is provided by subtracting the magnitude average of the noise from the MAE of each model as seen in Fig. 4.5. Green vertical lines on the plot represent 50%, 75%, 90%, and 95% measurement noise ratios. Below 50% measurement noise, the 5 models follow the trend of the process noise with the TCN models having similar or better performance to their extracted equations and the GA tuned model having the best performance. The area between 50% and 95% is rather convoluted, with no clear distinction in model performance. Above 95%, the TCN models generally outperform the equation models, although performance below the noise floor is based on chance.

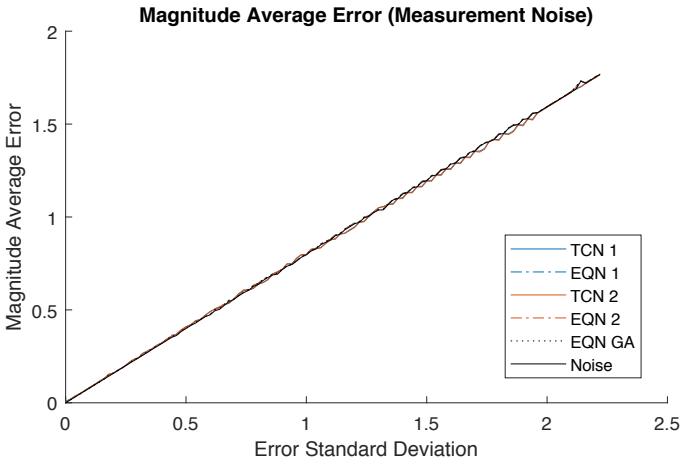


Figure 4.4: Effect of varied measurement error on the example system.

The plots show that the framework demonstrates significant resilience to measurement noise. This can again be attributed to the initial TCN model and the general ability of neural networks to ignore noise while learning underlying relationships. Noise resilience is

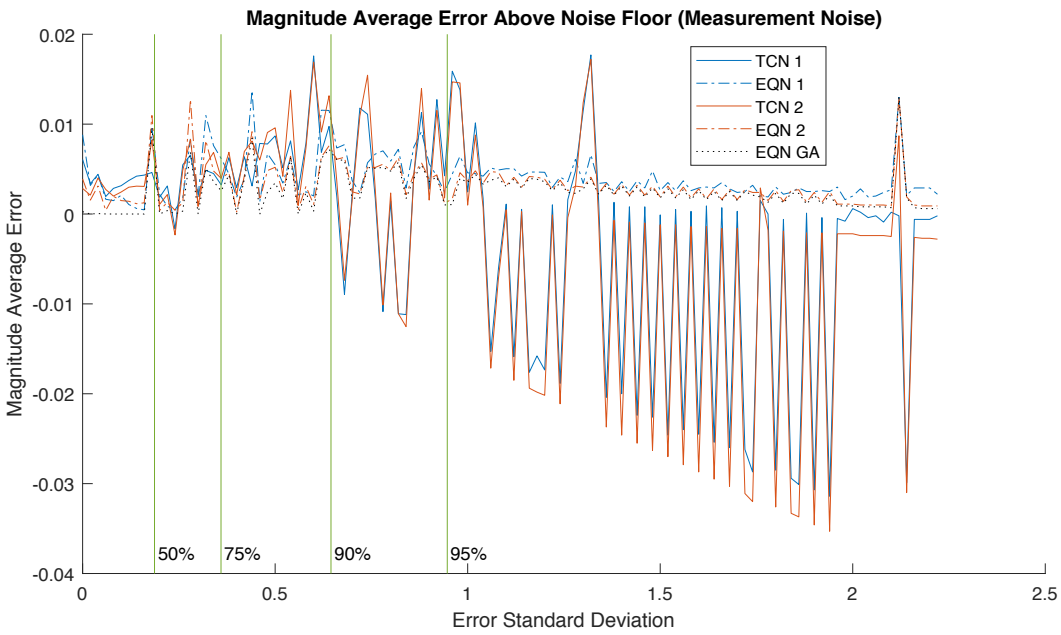


Figure 4.5: Effect of varied measurement error above the noise floor.

therefore another justification for the choice of neural network identification models in the framework.

4.3.3 Periodic Functions

Periodic functions can be a source of trouble for the framework. Notably, insufficient initial sampling can lead to missed detection of periodic functions since there may be multiple points at which they equal zero. While improbable with a larger number of samples, a small initial sampling may not detect the product function by only sampling the zero points. Take the simple example system

$$y[k] = \sin(4\pi u[k]) + u[k-1]. \tag{4.3}$$

Assume that no initial sweep set is used, and rather just a single point per product function is extracted from the default X and Y. It follows that $f(u[k]) = \sin(4\pi u[k])$ cannot be

detected since $f(0) = f(1) = 0$. Even with an initial sweep set, the periodicity of sine could potentially cause missed detection.

The system is excited with input drawn from $U[-1,1]$ and then the input-output data is analyzed using the framework. The TCN is formulated as

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & u_1[k-2] \end{bmatrix}$$

and has a corresponding MAE of 2.588e-2. An initial sweep set is not used for analysis, which yields the estimated equation

$$y_1[k] = 1.035u_1[k-1]$$

with a corresponding MAE of 3.193e-1. The TCN is retrained based on the input mask $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ to yield a corresponding MAE of 3.198e-1. Analysis of the retrained TCN model yields

$$y[k] = 10\left(e^{0.104u_1[k-1]} - 1\right),$$

having a corresponding MAE of 3.201e-1. Both equation estimates and the retrained TCN are an order of magnitude worse than the original TCN, triggering the framework to increase the initial and detailed sweep sizes and perform analysis again. Analysis with an initial sweep set of just 5 elements produces the equation

$$y[k] = 0.496\sin(12.593u_1[k-1] - 0.005) - \sin(-0.005) + 1.035u_1[k-1].$$

which has a much improved MAE of 1.854e-2. A new input mask $\begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$ is created for retraining, yielding a TCN with a corresponding MAE of 2.364e-2. Analysis yields the equation

$$y[k] = 0.489\sin(12.631u_1[k-1] + 0.008) - \sin(0.008) + 0.992u_1[k-1]$$

with a corresponding MAE of 1.486e-2. The GA tuning step produces the final equation

$$y[k] = 0.501\sin(12.568u_1[k-1] - 0.002) - \sin(-0.002) + 1.001u_1[k-1]$$

with a corresponding MAE of 2.367e-3, outperforming the TCN models and prior equation estimates. The resultant equation appears slightly different than the original system equation due to the requirement that modified product functions must be 0 with a 0 input. This is explained further in Section 4.3.4.

4.3.4 Modified Product Functions

Recall that product functions in this work are defined as non-additively separable functions of their input arguments. The definition is modified to allow for additive separability in the case of functions where a 0 input does not yield a 0 output: a necessary condition to derive the recursive relationship in (3.11) and (3.12). A prime example of a product function that must be modified is e^x. The need for product function modification is demonstrated through analysis of the example system

$$y[k] = e^{u_1[k]}.$$

The functional decomposition of this system with input instance $\bar{x}_i = [x_1]$ follows

$$F(x_1) = e^{x_1} = f_1(x_1) + f_0(\emptyset)$$

where the individual product functions are calculated by

$$f_0(\emptyset) = F(0)$$

$$f_1(x_1) = F(x_1) - f_0(\emptyset).$$

From the analysis theory, the all 0 input $\bar{x}_0$ is used to determine the model bias. This is the constant contribution of the product function $f_0(\emptyset)$. By examining the example system, it

is clear that there is no constant term. In fact, the only product function is $f_1(x_1)$. Without modification of the product function e^x, the analysis would indicate that both $f_1(x_1)$ and $f_0(\emptyset)$ are present in the system. It would assert that $f_0(\emptyset) = 1$ and then try to fit template product functions to $F(x_1) - f_0(\emptyset)$, which is $e^x - 1$. Without modification, there would not exist a product function matching $e^x - 1$ since it is additively separable. This founds the case for product function modification. By allowing $e^x - 1$ to be classified as a product function, the analysis still asserts that $f_0(\emptyset) = 1$ exists in the model, however a template function now exists for $F(x_1) - f_0(\emptyset)$. The resultant model is constructed as the sum of product functions

$$F(x_1) = (e^{x_1} - 1) + 1,$$

which is easily seen to be equivalent to e^{x_1}. In similar fashion, all product functions where an all 0 input does not yield a 0 output are modified. This makes the resultant equation slightly more complex, but it is a necessary cost associated with the analysis.

Brief results from the framework implementation are reported for the example system. The system is excited with input drawn from $U[-1,1]$ and then the input-output data is analyzed using the framework with the TCN formulation

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & u_1[k-2] \end{bmatrix}.$$

Initial TCN training and analysis yields the correct input mask $\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$. Retraining, analysis, and GA tuning are performed to yield the final equation

$$y[k] = 1.000\left(e^{0.999u_1[k]} - 1\right) + 1.001.$$

The MAE values at each step in the analysis process are presented as follows:

TCN 1	EQN 1	TCN 2	EQN 2	EQN GA
3.111e-3	2.410e-3	9.776e-4	8.490e-4	6.213e-4

4.3.5 Taylor Series Expansions

The implemented framework's use of curve fitting means that it is susceptible to many of the same issues associated with curve fitting. A notable issue is that the Taylor Series expansion of a function can be fit to samples describing the function instead of the function itself. This can be seen looking again at the simple example system

$$y[k] = e^{u_1[k]}.$$

The Taylor Series expansion of e^x is denoted as

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots.$$

This example further examines the results from Section 4.3.4. For both the initial and retrained model analysis, the MAE of each fit template function is calculated for each product function and examined. The MAE value for the modified product function template $e^x - 1$ is presented with the MAE values for the polynomial product function templates representing the Taylor Series expansion in Table 4.5. The values are based on analysis of the retrained TCN model.

Table 4.5: Product function Taylor Series expansion.

Product Function Template	MAE
$a\left(e^{bx_1} - 1\right)$	4.979e-4
ax_1	1.758e-1
$ax_1{}^2 + bx_1$	2.291e-2
$ax_1{}^3 + bx_1{}^2 + cx_1$	3.206e-3
$ax_1{}^4 + bx_1{}^3 + cx_1{}^2 + dx_1$	5.791e-4
$ax_1{}^5 + bx_1{}^4 + cx_1{}^3 + dx_1{}^2 + ex_1$	3.857e-4

It is evident that the MAE of the Taylor Series expansion of $e^x - 1$ corresponding to the 5th order polynomial is in fact lower than the MAE of $e^x - 1$ itself. Choosing the 5th order polynomial leads to a much more complex system equation though, which is undesirable when the goal of the framework is to generate comprehensible equations. The weight associated with each template product function is used to help remedy this problem, penalizing the high order polynomial and giving selection advantage to the exponential function. Adjusting the weights is a matter of user preference, effecting automatic selection of product functions. Equal weights leads to an accurate and complex function that is often overfit, so weighting should be done heuristically to prefer simpler functions.

4.3.6 Missing Template Functions

When a template function is missing that would otherwise best fit the product function samples, it is simply approximated using existing template functions. This is the general case with any curve fitting problem. The template functions shown in Section 4.1.3 are used to analyze the following system

$$y[k] = \frac{1}{u_1[k] + 1}.$$

The system is excited with input drawn from $U[-0.75, 0.75]$ and then the input-output data is analyzed using the framework with the TCN formulation

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & u_1[k-2] \end{bmatrix}.$$

Initial training and analysis are performed, yielding the correct input mask $\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$. Retraining, analysis, and GA tuning are then performed to yield the system equation estimate

$$y[k] = 0.373\left(e^{-2.738u_1[k]} - 1\right) + 0.969.$$

The MAE that is calculated at each step of the framework is reported as follows:

TCN 1	EQN 1	TCN 2	EQN 2	EQN GA
3.640e-2	4.425e-2	5.423e-3	5.919e-2	4.276e-2

The MAE of the final GA tuned equation is worse than both TCN models, which is a result of the equation structure not resembling the original system. The estimated equation is still a good representation of the modeled system, as evidenced by the response plot of Fig. 4.6.

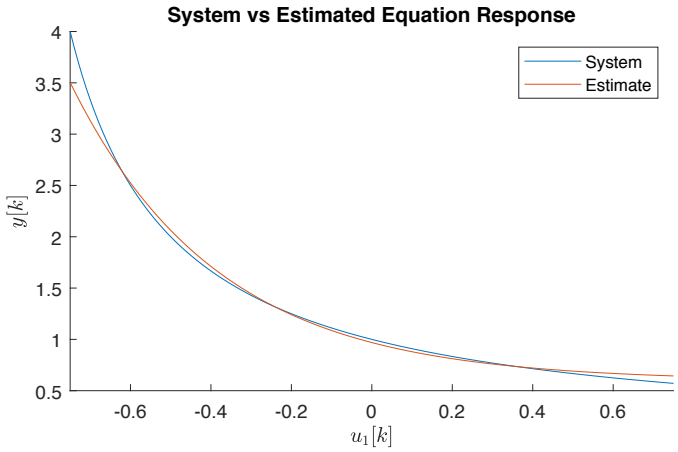


Figure 4.6: System vs. estimated equation response with a missing template function.

4.3.7 Models Undefined at or around 0

The framework requires product functions to have a 0 output for a 0 input, but what if a system is not defined at or around the 0 input? This is permissible in the framework provided all the template product functions are defined at the 0 input. The input data is simply mean-centered and then analysis is performed. After analysis, the shift needed to center the input data is used to modify the final function. This yields a more complex product

function that requires some simplification, but is equally effective in producing accurate estimated equations.

To illustrate the operation of the framework on models undefined at or around a 0 input, the following example system is analyzed

$$y[k] = u_1[k]^2.$$

The input is drawn from $U[1,2]$ and then the input-output data is analyzed using the framework with the TCN formulation

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & u_1[k-2] \end{bmatrix}.$$

Initial TCN training and analysis yields the correct input mask $\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$. Retraining, analysis, and GA tuning are performed to yield the final equation

$$y[k] = 1.002(u_1[k] - 1.500)^2 + 3.000(u_1[k] - 1.500) + 2.250.$$

Examination of the equation shows that in a similar fashion to product functions needing modification, models undefined at or around zero are determined to contain product functions not in the original system. The product functions that don't exist in the original system, however, will cancel out to yield the original system equation. The MAE of the final GA tuned equation is better than both TCN models and their corresponding equation estimates, in line with accuracy trends seen in models defined at 0. The MAE at each step of the framework is reported as follows:

TCN 1	EQN 1	TCN 2	EQN 2	EQN GA
2.867e-3	3.835e-3	1.728e-3	1.526e-3	2.060e-4

4.3.8 Higher Dimensional Product Functions

Product functions with more than 2 arguments cannot easily be visualized, and so manually examining their samples is not feasible. Since the framework operates without human intervention, this is not an issue provided template functions are defined for product functions containing higher numbers of arguments. To illustrate the operation of the framework on models containing product functions with more than 2 arguments, the following example system is analyzed

$$y[k] = u_1[k]\,u_1[k-2]\,u_1[k-4]\,u_1[k-6]\,u_1[k-8].$$

The input is drawn from $U[-1,1]$ and then the input-output data is analyzed using the framework with the TCN formulation

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & \cdots & u_1[k-9] \end{bmatrix}.$$

Initial TCN training and analysis yields the correct input mask, however retraining and secondary analysis produce a worse estimate. The initial equation estimate is GA tuned to yield the final equation

$$y[k] = 1.000u_1[k]\,u_1[k-2]\,u_1[k-4]\,u_1[k-6]\,u_1[k-8].$$

Examination of the results alone does not indicate any significant difference from usage of the framework on systems containing product functions of 1 or 2 variables, however key model and analysis parameter changes are needed to obtain results for systems containing higher order product functions. Notably, more TCN training epochs were needed in this example for the error to reach a satisfactory level (100 epochs vs. 10-30 epochs in all other synthetic experiments). Emphasis must be placed on training the TCN network sufficiently for the framework to succeed since the equation estimate of the sensitivity analysis is based

solely on the TCN. For the analysis, larger initial and detailed sweep sets were required (initial and detailed sweep sets of size 250 and 5000 used vs. 25 and 1000 in all other synthetic experiments). Since the accuracy of the framework improves with larger sweep sets, it is recommended to use the largest sweep set possible considering computational expense to increase successful detection and identification of higher dimensional product functions. The MAE at each step of the framework is reported as follows:

TCN 1	EQN 1	TCN 2	EQN 2	EQN GA
4.250e-3	1.027e-3	6.046e-3	1.540e-2	1.134e-7

4.4 Data-Driven Experiments

This section demonstrates application of the implemented framework to real-world systems. We assume no prior knowledge of the systems examined before processing the input-output data. The framework is often run multiple times and in multiple configurations to gather information about the systems in question.

4.4.1 Silverbox Analysis

The Silverbox dataset used for identification benchmarking in Section 2.4 is analyzed using the implemented equation generation framework. Recall that the Silverbox is a circuit equivalent of a nonlinear spring-mass damper system in the likes of an automotive suspension, described by the continuous equation

$$m\ddot{y}(t) + d\dot{y}(t) + k_1 y(t) + k_3 y^3(t) = u(t).$$

It has a single input $u(t)$ and single output $y(t)$ that are represented in discrete time by $u[k]$ and $y[k]$. The input represents applied force and the output represents displacement, however

both are voltages in the dataset since the Silverbox is a circuit. For further detail, refer back to Section 2.4.

A TCN model is formulated with K=3, L=2, and 22 hidden units per kernel leg. This is chosen based on the most successful simulation model reported in Section 2.4.3, although a smaller history and fewer training epochs are used due to the diminishing returns on accuracy for increased computational effort. There is also a point where improving the accuracy of the TCN model does not effect the selection and fitting of product functions. The TCN is formulated as a series-parallel model as follows:

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & u_1[k-2] & u_1[k-3] & u_1[k-4] \\ y_1[k-1] & y_1[k-2] & y_1[k-3] & y_1[k-4] & y_1[k-5] \end{bmatrix}.$$

Training is performed over 100 epochs using samples 40585-85000. Error metrics are reported as MAE and RMSE to be both consistent with the analysis work and in consensus with Silverbox literature. The training one-step-ahead prediction MAE is 0.503 mV and the RMSE is 0.642 mV. The validation error MAE as reported on samples 85001-127500 is 0.497 mV and the RMSE is 0.643 mV.

The framework is used to analyze the system twice; once with weight towards linear product functions, and once with no product function weighting. The initial TCN model is identical for both analyses, as is the initial magnitude and variance contributions. An initial sweep set with 250 elements is used to yield 250 points per product function for determining the average magnitude contribution and variance contribution of each product function. The identified product functions and their corresponding magnitude and variance contributions based on a 2% contribution cutoff are shown in Table 4.6. This corresponds to an input mask of $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix}$ for later retraining. The product functions determined to be significant are sampled using a detailed sweep set containing 1000 elements. Plots of the resultant samples for each product are shown in Fig 4.7.

Table 4.6: Product function contributions for the Silverbox.

Product Function	$z_i/\sum(Z)$	$s_i/\sum(S)$
$f(y_1[k-1])$	4.6%	24.8%
$f(y_1[k-3])$	4.1%	24.2%
$f(y_1[k-4])$	2.2%	6.5%
$f(u_1[k])$	1.9%	4.4%

Examination of the sampled product functions reveals that the Silverbox system is predominantly linear, which is consistent with the findings of other Silverbox literature. Manual examination of the product function plots of one or two variables proves to be a power visual tool for learning about systems. Product functions of higher orders cannot be easily visualized and examined, leaving their estimation solely to the framework. The framework process up to this point is identical regardless of the template product function weights, and the analysis is now split based on weighting schemes as previously mentioned. Since the system is predominantly linear, analysis is first performed with higher weight on the linear template product function. The system equation is initially estimated as

$$y[k] = 0.484y[k-1] - 0.418y[k-3] - 0.231y[k-4] + 0.480u[k],$$

having a one-step-ahead prediction MAE of 12.48 mV and an RMSE of 15.53 mV. When analysis is performed without any weight assigned to template product functions, the system equation is initially estimated as

$$\begin{aligned} y[k] = {} & 40.0y[k-1]^5 - 3.37y[k-1]^4 - 3.38y[k-1]^3 + 28.6y[k-1]^2 + 0.543y[k-1] \\ & + 7.87y[k-3]^5 - 1.79y[k-3]^4 - 1.07y[k-3]^3 + 0.144y[k-3]^2 - 0.395y[k-3] \\ & - 4.99y[k-4]^5 + 0.207y[k-4]^4 + 0.396y[k-4]^3 - 0.0525y[k-4]^2 - 0.237y[k-4] \\ & + 1.00\tanh(1.38(u[k]-0.0558)) - 0.891(u[k]-0.0558) + 0.0272, \end{aligned}$$

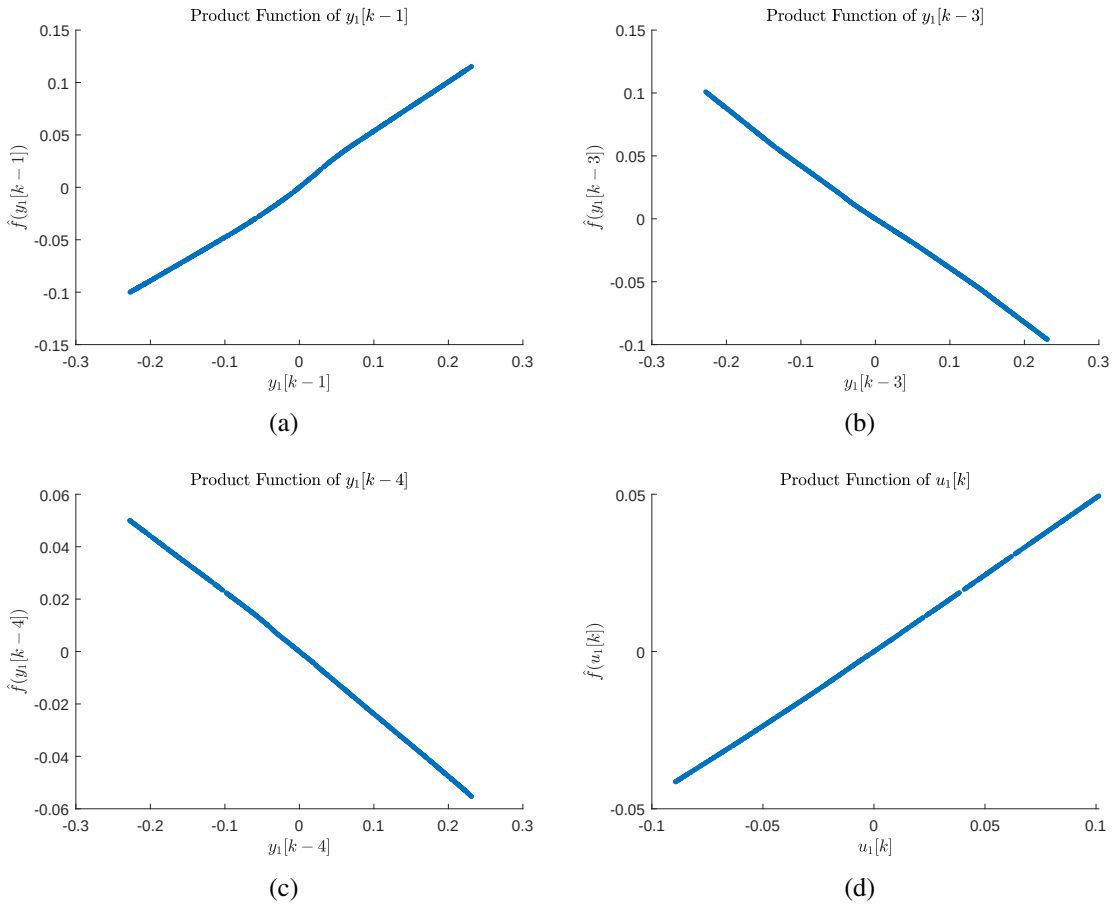


Figure 4.7: Plots showing 1000 samples for each product function of significance as extracted from the TCN trained on the Silverbox dataset: (a) $\hat{f}(y_1[k-1])$; (b) $\hat{f}(y_1[k-3])$; (c) $\hat{f}(y_1[k-4])$; (d) $\hat{f}(u_1[k])$

having a one-step-ahead prediction MAE of 11.02 mV and an RMSE of 14.01 mV. While the accuracy of the estimate has improved, the equation is much more complex. The framework generally suffers from overfitting without weighting the template product functions. As is the general case with system identification, there is an accuracy versus complexity trade-off.

Both weighted and non-weighted analyses are performed again after retraining of the TCN model using the input mask. The same retrained TCN is used for both analyses

since the input mask is based on the contribution of product functions, and is therefore identical. The retrained TCN model has a corresponding training one-step-ahead prediction MAE of 1.880 mV and a RMSE of 2.369 mV. The validation MAE is 1.887 mV and the RMSE is 2.381 mV. The model is excited with an initial sweep set of 250 elements and the corresponding product functions are identified based on a 2% contribution cutoff and shown in Table 4.7. Of particular interest is the fact that a product function not present in the initial model analysis is present in the retrained model analysis. As with the initial model, a detailed sweep set of 1000 elements is used to sample the product functions of the retrained model. The plots are shown in Fig. 4.8.

Table 4.7: Product function contributions for the Silverbox after retraining.

Product Function	$z_i/\sum(Z)$	$s_i/\sum(S)$
$f(y_1[k-1])$	38.9%	51.7%
$f(y_1[k-3])$	30.0%	38.0%
$f(y_1[k-4])$	12.9%	6.5%
$f(u_1[k])$	9.0%	2.8%
$f(y_1[k-1], y_1[k-3])$	2.4%	0.4%

Analysis continues with curve fitting for the samples of each product function. Linear functions are again fit to the most significant 4 product functions with weighting, and nonlinear functions are fit without weighting. In both analysis cases, the least significant product function $\hat{f}(y_1[k-1], y_1[k-3])$ cannot be fit using template functions in a way that improves the MAE of the overall equation estimate. Since it is of low significance and does not improve the estimate, it is dropped. The sum of product function estimates with and without template product function weighting is finally tuned by GA. The estimated equation with weighting favoring linear functions is

$$y[k] = 1.057y[k-1] - 0.896y[k-3] + 0.334y[k-4] + 0.476u[k],$$

having a one-step-ahead prediction MAE of 2.993 mV and RMSE of 3.637 mV. The MAE before GA tuning is 4.914 mV and the RMSE 6.075 mV. Without weighting, the estimated equation is

$$\begin{aligned} y[k] &= -12.3y[k-1]^5 + 4.06y[k-1]^4 - 0.646y[k-1]^3 - 0.221y[k-1]^2 + 1.05y[k-1] \\ &+ 33.0y[k-3]^5 - 0.291y[k-3]^4 - 3.30y[k-3]^3 - 0.114y[k-3]^2 - 0.850y[k-3] \\ &- 0.00287\tanh(34.6(y[k-4] - 1.76e-5)) + 0.346(y[k-4] - 1.76e-5) - 6.14e-4 \\ &+ 99.4u[k]^5 + 26.9u[k]^4 - 4.49u[k]^3 - 0.469u[k]^2 + 0.518u[k], \end{aligned}$$

having a one-step-ahead prediction MAE of 2.359 mV and RMSE of 2.958 mV. The MAE before GA tuning is 3.769 mV and the RMSE 4.778 mV. A third analysis was performed afterwards using a heuristically tuned balanced weighting scheme to yield the final estimated equation with linear and nonlinear terms

$$\begin{aligned} y[k] &= 0.981\tanh(1.50(y[k-1] + 0.0421)) - 0.415(y[k-1] + 0.0421) - 0.0455 \\ &- 0.865y[k-3] \\ &- 0.00386\tanh(34.4(y[k-4] + 1.76e-5)) + 0.358(y[k-4] + 1.76e-5) - 6.13e-4 \\ &+ 0.518u[k]. \end{aligned}$$

The one-step-ahead error metrics for the linear, nonlinear, and mixed equations are summarized for easy comparison in Table 4.8.

Table 4.8: One-step-ahead RMSE of Silverbox models in mV.

Model	TCN 1	EQN 1	TCN 2	EQN 2	EQN GA
Linear	0.643	15.53	2.381	6.075	3.637
Nonlinear	0.643	14.01	2.381	4.778	2.958
Mixed	0.643	15.46	2.381	3.816	2.777

There are a few key observations to be made by examining the error metrics of the three equations. First and foremost, there is an increase in error between the initial TCN and retrained TCN. This indicates that there is likely useful information in some of the inputs deemed to be insignificant, which can be caused by the 2% contribution cutoff being too high. A second indicator of this is that the initial equation estimate has a much higher error than the initial TCN it was extracted from, which could also be caused and/or exacerbated by a lack of well-fitting product function templates. Lowering the cutoff threshold may give more accurate results, however the equation complexity may grow. Therefore, the analysis should be performed with a desired acceptable level of error predefined. The metrics show that equation generated using heuristically tuned balanced weighting outperforms the equations with no weighting and with weighting heavily biased toward the linear product function. This shows that carefully tuning the weights can prevent overfitting.

The analysis was performed again with a 1% contribution cutoff using the balanced product function weighting, however the final estimated equation was more complex and not as accurate. The error metrics are reported as follows:

TCN 1	EQN 1	TCN 2	EQN 2	EQN GA
0.643	10.65	0.705	5.970	3.552

The initial equation contained 6 product functions of a single argument and 2 of two arguments. The second and final equations contained 6 product functions of a single argument and 2 of three arguments. Comparison of the initial and retrained TCN models indicates that the significant inputs were properly identified for retraining. This points to a lack of proper template functions as the reason for increased error in the equation estimates. To increase the framework's accuracy in generating an equation, the product

function samples should be inspected to come up with better template functions. Ideally, if enough template functions exist, then manual inspection is not needed.

To compare the estimated equations to the Silverbox identification methods surveyed in Section 2.4, the equation models are run in free simulation. The linear equation achieves a simulation RMSE of 15.380 mV, which is competitive amongst linear models for the Silverbox. An RMSE of 14.9 mV is achieved for a linear model in [42], while a slightly better linear model with an RMSE of 14.4 mV is obtained in [38]. Similar results of 14.8 mV and 14.9 mV are obtained for linear models in [40]. A linear model obtains an RMSE of 9.90 mV in [43], and another linear model obtains an RMSE of 13.7 mV in [46]. The nonlinear equation achieves a simulation RMSE of 18.590 mV, further supporting the suspicion that the model suffers from overfitting. The mixed equation created using balanced template product function weights achieves a simulation RMSE of 12.563 mV. While the performance of the mixed equation surpasses the performance of the strict linear equation, it is not very competitive against the top nonlinear identification models for the Silverbox. It does, however, offer full transparency and simplicity. The performance of the framework on the Silverbox dataset could be improved with further tuning and the addition of more template product functions. The Silverbox is used to demonstrate the use of the framework on real systems, so top tier equation results are not pursued.

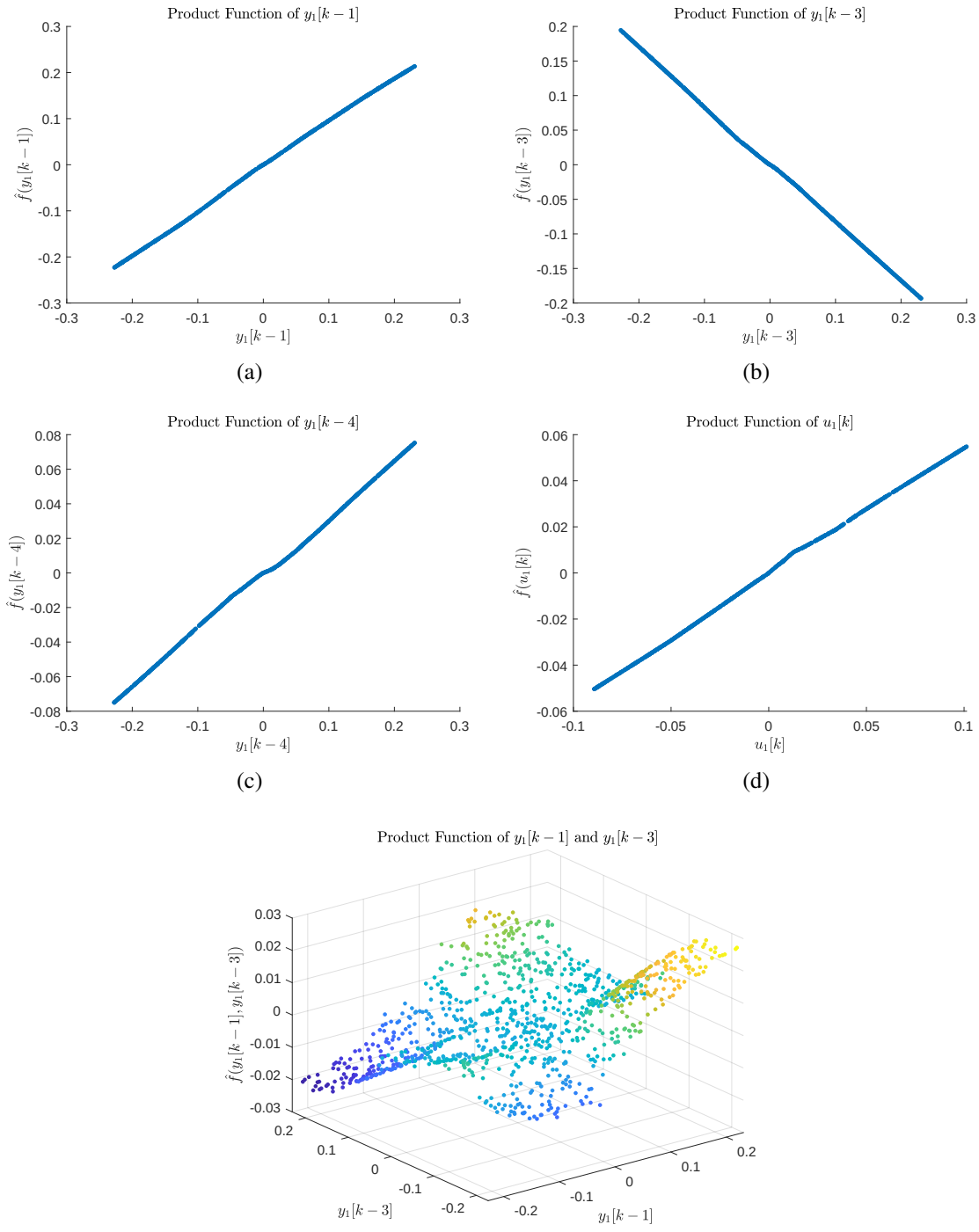
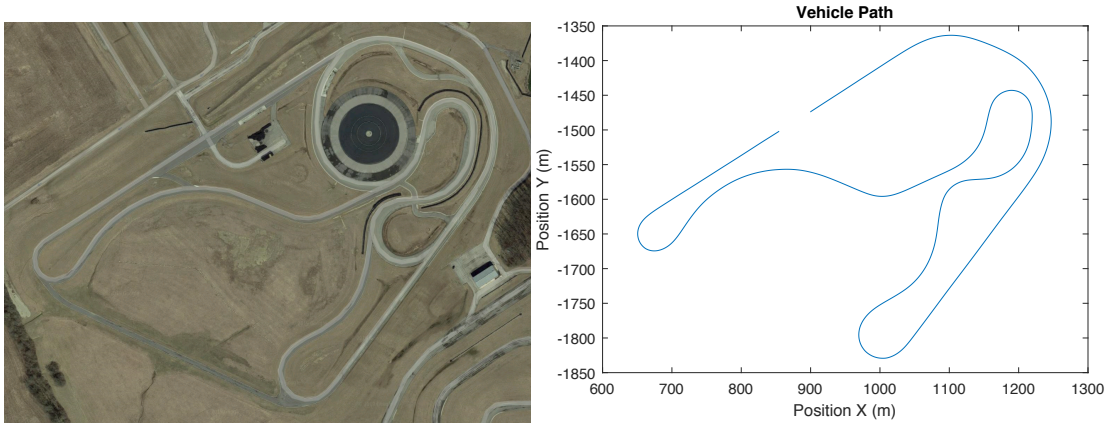


Figure 4.8: Plots showing 1000 samples for each product function of significance as extracted from the retrained TCN for the Silverbox dataset: (a) $\hat{f}(y_1[k-1])$; (b) $\hat{f}(y_1[k-3])$; (c) $\hat{f}(y_1[k-4])$; (d) $\hat{f}(u_1[k])$; (e) $\hat{f}(y_1[k-1], y_1[k-3])$

4.4.2 Vehicle Roll Analysis

The framework is used to generate a discrete dynamical system equation for the roll of a vehicle. The vehicle, a 2017 Lincoln MKZ, was equipped with a Dataspeed Inc. ADAS (Advanced Driver Assistance Systems) Kit and run autonomously with path and velocity profiling controllers on the Winding Road Course at Transportation Research Center Inc. (TRC) in East Liberty, Ohio. An aerial view of the test track is provided in Fig 4.9 along with the vehicle path on the track. Data from three full laps and one partial lap around the track is used for the analysis, with the variables of interest being velocity, steering angle, and vehicle roll. Data from the first lap is shown in Fig. 4.10. The sample period is 0.05 second.



(a) (b)

Figure 4.9: The test track used for vehicle data collection: (a) Aerial view; (b) Vehicle path.

The analysis begins with formulation of the identification problem and the TCN. A dynamics equation describing vehicle roll is desired. Velocity and steering angle are major factors in vehicle roll, so they are chosen as the inputs to the system. Let y represent vehicle

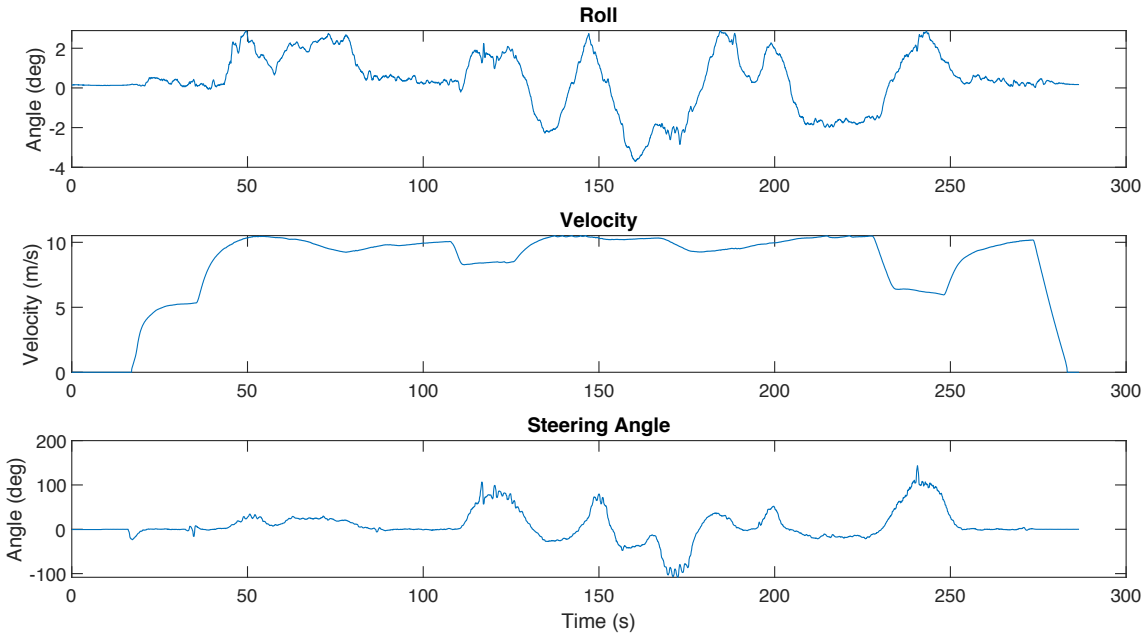


Figure 4.10: Vehicle data from a lap of the Winding Road Course.

roll, and u_1, u_2 represent vehicle velocity and steering angle respectively. The TCN is formulated as follows to predict the current roll using past roll, velocity, and steering angle

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & u_1[k-2] \\ u_2[k] & u_2[k-1] & u_2[k-2] \\ y[k-1] & y[k-2] & y[k-3] \end{bmatrix}.$$

A kernels size of 2 is used with 2 residual blocks and 32 hidden nodes per kernel leg. A dropout rate of 10% is used on hidden nodes to assist with training and prevent overfitting. Dropout is needed in this problem to prevent the TCN from learning to predict roll using only past roll values. The TCN is trained for 150 epochs on 90% of the data, leaving 10% of the data for validation. A training MAE of 2.344e-02 degrees and a validation MAE of 4.446e-02 degrees are obtained. A cutoff of 0.5% magnitude and variance contribution is used to determine which product functions are significant.

An initial equation is generated and the TCN is retrained by masking off the irrelevant inputs, however this leads to worse performance with a training MAE of 4.203e-02 degrees and a validation MAE of 5.434e-02 degrees. As a result, the equation extracted from the initial TCN is tuned by GA to yield

$$y[k] = 9.93e-1y[k-1] + 4.81e-4u_1[k]y[k-1] + 1.95e-4u_2[k].$$

The final equation has a corresponding MAE of 2.116e-02 degrees, outperforming the initial TCN model. This is an optimal success case for the framework. As formulated, the equation and TCN produce a one-step-ahead prediction of the roll angle. The one-step-ahead and simulation roll angle predictions estimated using the equation are compared to the original angle data in Fig. 4.11. One-step-ahead performance is excellent, however simulation performance does not see the same success. The simulated system equation is stable across the test data and follows general trends in the original roll angle data, but the MAE is 1.190 degrees as opposed to the OSA MAE of 0.021 degrees. As a result, the estimated equation is well suited for N-step-ahead prediction but is not suitable as a system observer or full simulation model.

The simulation performance of the equation is directly related to the performance of the TCN it was extracted from. Since the equation outperforms the TCN, better performance cannot be expected from the analysis. The limiting factor is the trained TCN, which indicates that more information is needed during training to improve the analysis resultant equation. Notably, road grade is not included in the TCN model although it effects the roll angle. The course is relatively flat, but such low roll angles in the training data are susceptible to corruption by slight road grades. Completing the course at higher speeds would also illicit higher roll angles, helping with system excitation for identification. Training data diversity on different courses would also help with increasing the TCN accuracy.

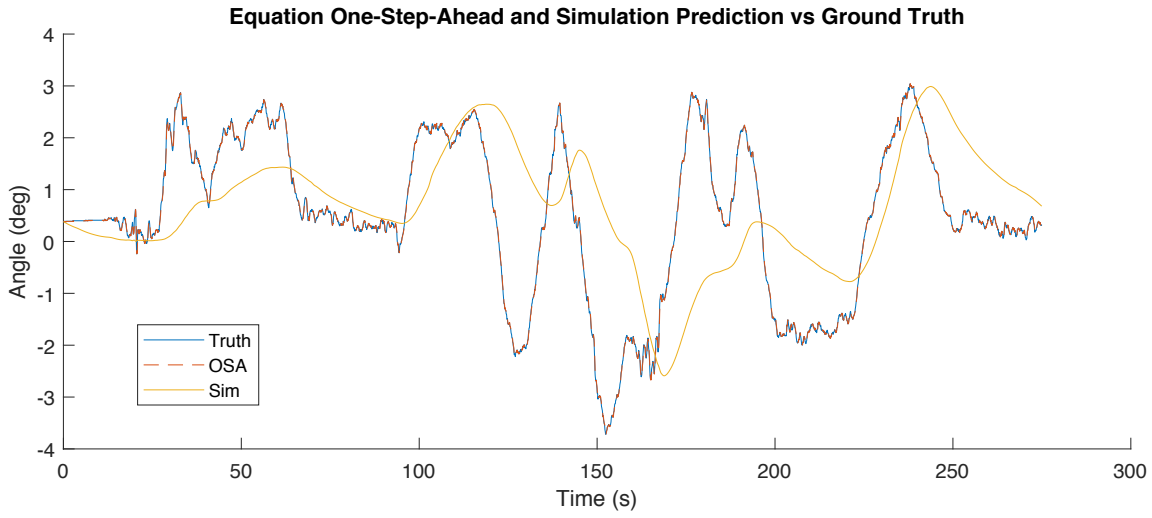


Figure 4.11: One-step-ahead and simulation prediction using the estimated equation.

The equation performance is already better than the TCN performance, but it can still be improved. The sampled product functions are shown in Fig. 4.12 with their magnitude and variance contributions, yielding insight into the equation generation process. The equation accurately represents the two most dominant product functions, however representation of the least dominant product function $\hat{f}(u_1[k], y[k-1])$ is poor. While it is best approximated with the template function $au_1[k]y[k-1]$, it would be better fit with a piecewise function of two planes. Expanding the template product functions may likely increase the performance of the equation model in this scenario.

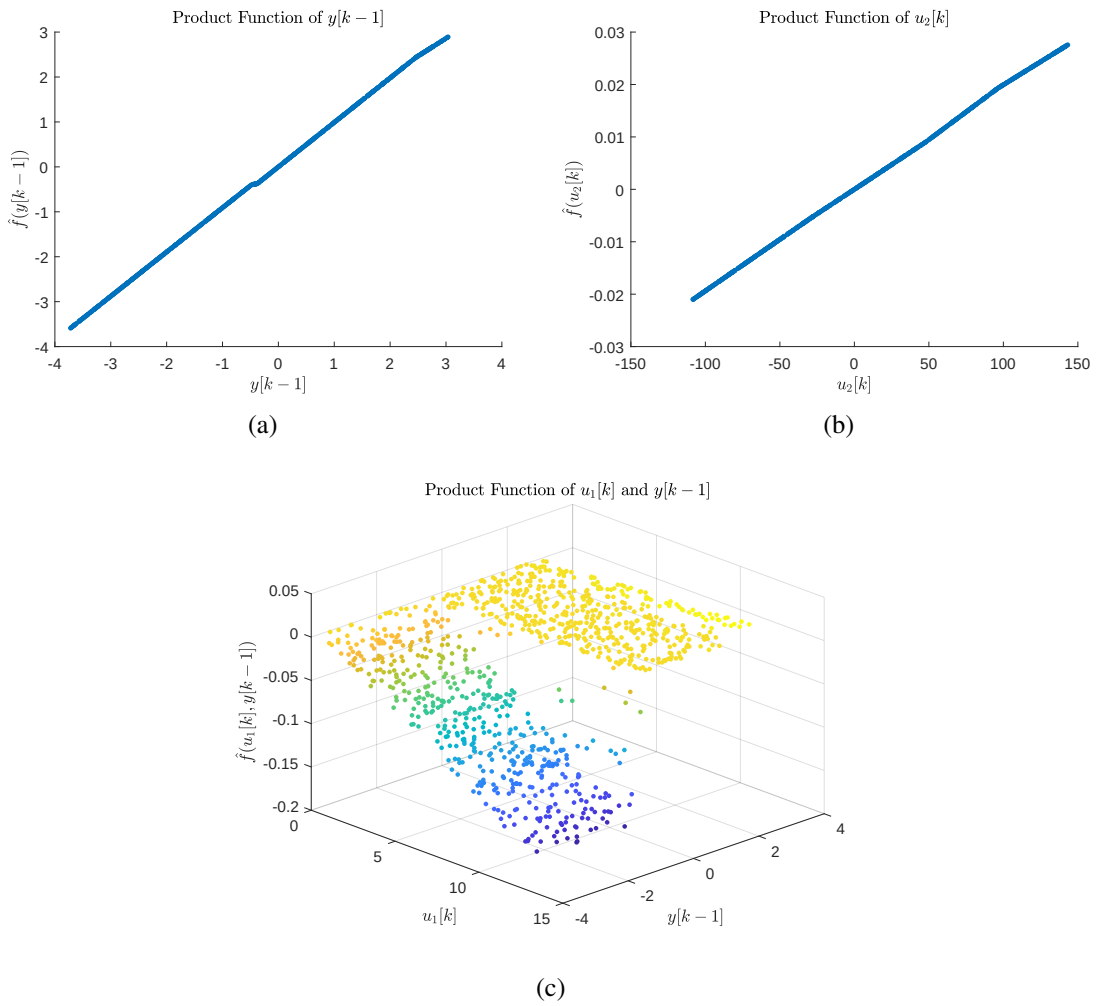


Figure 4.12: The sampled product functions from the initial TCN of the vehicle roll data: (a) $\hat{f}(y[k-1])$, 92.9% magnitude, 99.9% variance; (b) $\hat{f}(u_2[k])$, 1.0% magnitude, 0.0% variance; (c) $\hat{f}(u_1[k], y[k-1])$, 0.6% magnitude, 0.0% variance.

4.4.3 Motor Cooling Analysis

A simulated electric vehicle motor cooling system is analyzed using the framework to model the relationship between the motor temperature and the cooling system inlet and outlet temperatures. The goal is to create a predictor of the motor temperature using only the inlet and outlet temperatures. The cooling system is part of the electric vehicle model from the

Simulink example "Electric Vehicle Configured for HIL" [63] and is shown in Fig. 4.13. Let y represent motor temperature and let u_1 and u_2 represent the inlet and outlet temperatures respectively. Data is collected with a period of 2 seconds over a simulation duration of 20000 seconds. The vehicle throttle and the scenario road grade are varied randomly every 30 seconds between $0-50\%$ and $0-5^\circ$ respectively. The TCN is formulated as

$$\hat{y}[k] = \hat{F}(\bar{x}[k]), \quad \bar{x}[k] = \begin{bmatrix} u_1[k] & u_1[k-1] & \cdots & u_1[k-6] \\ u_2[k] & u_2[k-1] & \cdots & u_2[k-6] \end{bmatrix}$$

and trained on data from $5000-18500$ seconds (data prior to 5000 seconds is discarded as the motor had not yet reached a stable temperature). A total of 100 training epochs are used with a kernel size of 3, 2 residual blocks, 48 hidden nodes in the kernel, and a dropout rate of 5%. A cutoff of 1.0% magnitude and variance contribution is used to determine significant product functions, which are shown in Fig. 4.15. The GA tuned final equation is

$$\begin{aligned} \hat{y}[k] = & -0.149(u_1[k]-69.4)^2 + 6.71(u_1[k]-69.4) \\ & - 30.3\tanh(0.116((u_2[k]-68.2)+18.2)) - 5.50((u_2[k]-68.2)+18.2) + 130 \\ & + 0.133(u_1[k]-69.4)(u_2[k]-68.2) \\ & + 76.047. \end{aligned}$$

Metrics at each step of the analysis process are reported as follows:

TCN 1	EQN 1	TCN 2	EQN 2	EQN GA
7.994e-1	4.363e+0	1.003e+0	3.341e+0	9.348e-1

Data from $18500-20000$ seconds is used for validation and testing. Since the model takes the form of a simplified parallel model, it is always running in a similar fashion to simulation. Simulation of the test section is shown in Fig. 4.14 along with the ground truth data. The simulation MAE over the test set is 1.001 °C. Overall, simulation performance

follows performance of the actual system well. Since the GA tuned equation error is similar to the error of the best TCN model, the framework was successful at generating an equation. If further improvement is desired, then the TCN model will likely need to be changed with the addition of more variables.

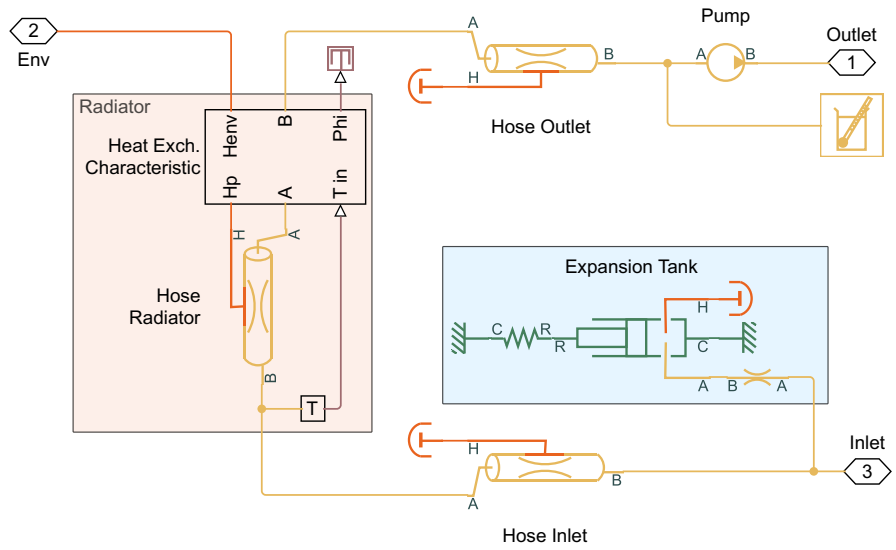


Figure 4.13: The electric vehicle cooling system.

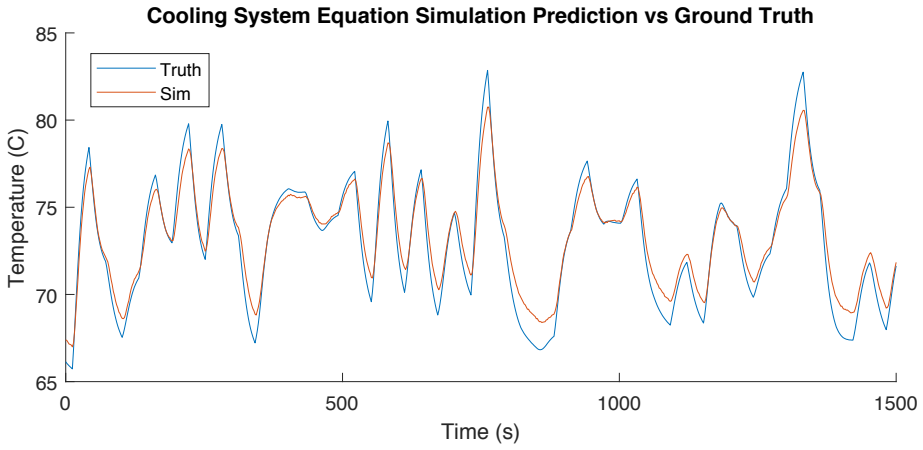


Figure 4.14: Simulation of the electric vehicle motor temperature.

One of the key benefits of this framework is the flexibility of the model structure. Many combinations of input and output variables can be tested to generate predictors. The framework will automatically identify the most important inputs and input relationships. In this way, creative predictors can be generated with little effort.

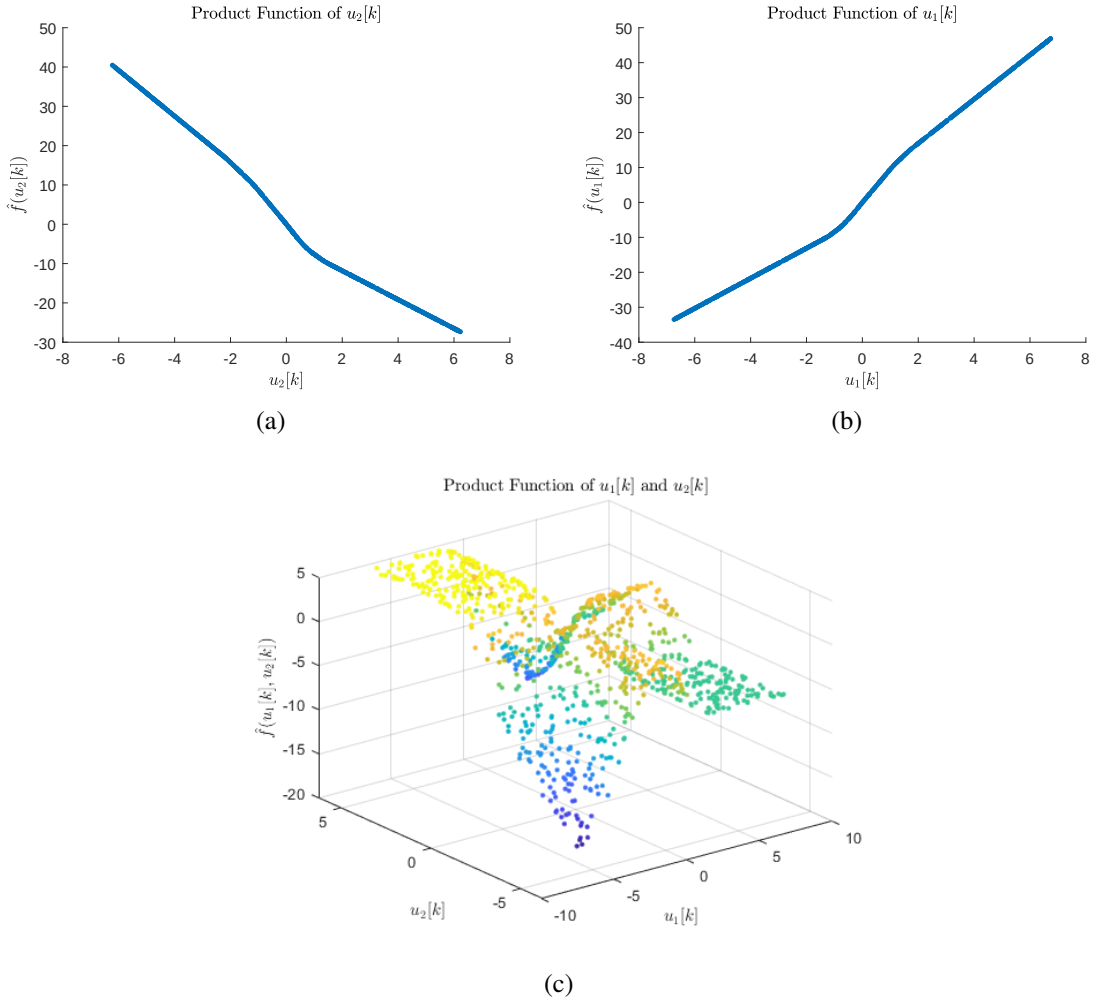


Figure 4.15: The sampled product functions from the retrained TCN of the motor cooling data (constant not shown): (a) $\hat{f}(u_2[k])$, 48.1% magnitude, 49.3% variance; (b) $\hat{f}(u_1[k])$, 42.6% magnitude, 47.3% variance; (c) $\hat{f}(u_1[k], u_2[k])$, 9.3% magnitude, 3.4% variance.

4.5 Summary and Conclusions

An implementation of the framework established in Chapter 3 was provided in this chapter for generating discrete dynamical system equations from input-output data. The strengths and weaknesses of the framework were analyzed and the implementation was used to examine synthetic and real-world systems. The framework was demonstrated to be successful at decomposing identification problems into a series of less complex curve fitting problems. It was often observed that the resultant equation yielded better accuracy than the neural network model it was derived from. This indicates that some portion of underlying system dynamics were discovered by the framework, or at least approximated well. While neural network models are generic and generally very accurate, they are often much more complex than needed and have many parameters. The equation model generated by the framework is far simpler than the neural network model and attempts to determine underlying system structure as a means to reduce model complexity. Once the system structure is estimated, then the genetic algorithm tuning adjusts the equation parameters to better match the original input-output data. The framework proves to be a powerful and generic analysis tool in examination of discrete dynamical systems.

Chapter 5: Conclusion

In this work, neural network based system identification was combined with a form of sensitivity analysis to yield a framework for generating comprehensible discrete dynamical system equations from input-output data. The framework was implemented and the implementation evaluated for performance on both synthetic and real systems. The framework was demonstrated to successfully determine or approximate system structure and yield accurate system equations.

5.1 Contributions

The fundamental contribution from this body of work is an advancement in the state of systems analysis. The presented analysis framework is a pragmatic tool for not only generating comprehensible system equations for input-output data, but for understanding the underlying dynamics of a system as well. The framework sheds light on black box systems and black box models alike by either exposing their structure or providing a clear approximation. The framework also breaks down models, allowing for complex identification problems to be reduced to a series of much simpler curve fitting problems.

The analysis framework is the resultant of combining neural network based system identification with a form of sensitivity analysis. A neural network variant, known as the temporal convolutional network, was investigated in this work for identification capabilities

and shown to yield benefits over traditional feedforward neural networks in free running simulation performance. This indicates that it is preferred for use as an observer or parallel system identification model and has superior N-step-ahead prediction capabilities. For this reason, along with the generalization capacity of neural networks, the temporal convolutional network was chosen for the identification portion of the framework.

The identification model created using a temporal convolutional network is analyzed with a form of sensitivity analysis introduced in this work. The analysis is applicable to generic models, so its contribution is not limited to the presented framework. The analysis process is powered by a unique interpretation of a functional decomposition of the model, providing a way to sample functional fragments of the model. This decomposition simplifies the creation of the equation model by breaking the overall identification problem into a series of smaller and less complex curve fitting problems. The sum of functional fragments is then equivalent to the full equation model. This process allows for analysis of increasingly complex systems.

The system identification and sensitivity analysis methods were combined and augmented to yield the presented system equation generation framework. Iterative processes and conditional operations were integrated into the framework to increase the success rate of identifying a system's structure and increase the accuracy of the resultant equation. The final analysis framework is a robust and powerful system analysis tool.

5.2 Future Work

Future work can be categorized under either system identification, model analysis, or the framework implementation. The temporal convolutional network used for system identification has many high level parameters within itself, such as number of residual blocks and

kernel size. The relationship between these parameters and the structure of systems should be identified. For example, does a kernel size of 2 lead to better identification of product functions of 2 arguments? Beyond temporal convolutional networks, a multitude of other neural network variants exist that could be examined. The temporal convolutional network was chosen in this work due to the potential for success exhibited by other convolutional architectures on sequence modelling problems and the gap in existing literature studying its identification capabilities.

The model analysis may show improvement with better sampling techniques. Currently, random uniform sampling is used to obtain product function samples. More intelligent sampling procedures typically seen in sensitivity analysis such as Latin hypercube sampling [64] may likely improve the analysis. Determining the optimal sensitivity measure (or combination thereof) will also likely improve the analysis by increasing the accuracy of detecting product functions present in the model. A comparison of sensitivity measures should be conducted in future work.

The framework implementation has much room for expansion in future work. The list of template product functions included in the presented implementation can be added to based on real use of the framework to create a robust library of functions. Ideally, templates could be shared between those using the framework to accomplish this task. Template functions for product functions of more than three arguments must be defined as well. Among the template functions, better selection methods should be examined. Template functions are currently chosen based on heuristic weighting, which allows for user preference but hinders the level of autonomy achieved by the framework. Among both the system identification and template function selection, issues with overfitting must be better mitigated. The function

weighting helps reduce overfitting of template functions while dropout reduces overfitting during temporal convolutional network identification.

5.3 Concluding Remarks

This work presents a framework for generating equations of discrete dynamical systems using input-output data and provides a publicly available implementation - a tool to be used for system analysis. The framework sheds light on black box systems and affords a new level of transparency in system identification. The resultant equation models produced by the framework can serve as substitutes for opaque models where transparency is desired or required. This allows for new modelling options in fields where black box models are undesirable, such as control, stability analysis, and software verification. In summary, the presented framework is a new instrument in the domain of systems analysis that generates comprehensible discrete dynamical system equations from input-output data using a methodical and transparent process.

Appendix A: Python Code

Listing A.1: Example usage of the implemented framework.

```
import numpy as np

from lib.fitting_functions import fitting_functions
from estimate_equation import estimate_equation

if __name__ == "__main__":

    model_parameters = {
        "epochs": 20,               # upper epoch limit
        "cuda": False,              # use the GPU
        "dropout": 0.0,             # dropout applied to layers
        "clip": -1,                 # -1 means no clip
        "ksize": 3,                 # kernel size
        "levels": 2,                # number of levels
        "nhid": 64,                 # number of hidden units per layer
        "batch_size_train": 32,     # batch size for training
        "train_loss_full": 0,       # [1] full loss, [0] avg over batches
        "lr": 0.002,                # learning rate
        "lr_grad_period": 20,       # decay period for learning rate
        "lr_grad_rate": 0.5,        # decay multiplier after decay period
        "optimizer": 'Adam',        # optimizer to use
        "history": 5,               # number of time sample inputs to the network
        "test_data": 0.2,           # test data proportion
        "seed": 1111,               # random seed
        "visual": False,            # plot training metrics
        "save_visual": True,        # save training metric plot
    }

    analysis_parameters = {
        "functions": fitting_functions(),   # an object containing template product functions
        "sweep_initial": 25,                # elements in initial sweep set for significance detection
        "sweep_detailed": 1000,             # elements in detailed sweep set for curve fitting
        "contrib_thresh": 0.05,             # minimum product function significance for inclusion [0.00-1.00]
        "contrib_thresh_omit": 0.10,        # threshold for conditional product function inclusion
        "use_f_weight": True,               # use product function weighting
        "seed": 1111,                       # analysis rng seed for reproducability
        "verbose": True,                    # print details of analysis
        "visual": False,                    # show plots of available 2D and 3D product function samples
        "save_visual": True,                # save plots of available 2D and 3D product function samples
        "GA_population": 250,               # population size for GA tuning
        "GA_generations": 100               # number of generations in GA tuning
    }

    # Verbose example extension.
    # y[k] = -0.5*u[k-1] + 0.5*y[k-2]^2 + 0.5*u[k]y[k-1]
    input_data = 2*(np.random.rand(25000, 2)-0.5)
    output_data = np.zeros([25000, 1])

    for i in range(2, 25000):
        u_k0 = input_data[i, 0]
        u_k1 = input_data[i-1, 0]
        y_k1 = output_data[i-1, 0]
        y_k2 = output_data[i-2, 0]
        input_data[i, 1] = y_k1
        input_data[i-1, 1] = y_k2

        output_data[i, 0] = -0.5*u_k1 + 0.5*pow(y_k2,2) + 0.5*u_k0*y_k1

    # Estimate the equation using input-output data.
    estimate_equation(model_parameters, analysis_parameters, input_data, output_data)
```

Listing A.2: The `estimate_equation()` function which executes the overall framework.

```
# Estimate a discrete dynamical system equation for input_data and output_data.
# Author: John M. Maroli

import copy

from lib.create_model import create_model
from lib.analyze_model import analyze_model
from lib.evaluate_function import evaluate_function
from lib.tune_model import tune_model

FORMAT = '%.3e'

def estimate_equation(model_parameters, analysis_parameters, input_data, output_data):

    sweep_initial = analysis_parameters["sweep_initial"]
    sweep_detailed = analysis_parameters["sweep_detailed"]
    input_mask = 1

    # Initial round of training.
    print("Initial training and analysis")
    print("============================================================")
    model_dictionary_v1 = create_model(model_parameters, input_data, output_data, input_mask)

    model_function_v1, new_mask = analyze_model(analysis_parameters, model_dictionary_v1,
                                                input_data, output_data, input_mask)
    metrics_v1 = evaluate_function(model_function_v1, input_data, output_data)

    for channel_id, channel_metrics in enumerate(metrics_v1):
        print("Channel y" + str(channel_id+1) + " metrics")
        print("MAE  : " + str(FORMAT%channel_metrics["MAE"]))
        print("RMSE : " + str(FORMAT%channel_metrics["RMSE"]))
        print("MAX  : " + str(FORMAT%channel_metrics["MAX"]))
        print("MIN  : " + str(FORMAT%channel_metrics["MIN"]))
        print()

    # Model retraining after first analysis.
    print("Masked retraining and analysis")
    print("============================================================")
    print("New mask")
    print(new_mask)
    print()
    model_dictionary_v2 = create_model(model_parameters, input_data, output_data, new_mask)
    model_function_v2, _ = analyze_model(analysis_parameters, model_dictionary_v2,
                                         input_data, output_data, new_mask)
    metrics_v2 = evaluate_function(model_function_v2, input_data, output_data)

    for channel_id, channel_metrics in enumerate(metrics_v2):
        print("Channel y" + str(channel_id+1) + " metrics")
        print("MAE  : " + str(FORMAT%channel_metrics["MAE"]))
        print("RMSE : " + str(FORMAT%channel_metrics["RMSE"]))
        print("MAX  : " + str(FORMAT%channel_metrics["MAX"]))
        print("MIN  : " + str(FORMAT%channel_metrics["MIN"]))
        print()

    model_function_v3 = []
    metrics_v3 = []
    for c in range(0, len(metrics_v2)):
        if metrics_v2[c]["MAE"] < metrics_v1[c]["MAE"]:
            print("Channel y" + str(c+1) + " improved")
            model_function_v3.append(model_function_v2[c])
            metrics_v3.append(metrics_v2[c])
        else:
            print("Channel y" + str(c+1) + " did not improve")
            model_function_v3.append(model_function_v1[c])
            metrics_v3.append(metrics_v1[c])
        print("MAE  : " + str(FORMAT%metrics_v1[c]["MAE"]) + " -> " + str(FORMAT%metrics_v2[c]["MAE"]))
        print("RMSE : " + str(FORMAT%metrics_v1[c]["RMSE"]) + " -> " + str(FORMAT%metrics_v2[c]["RMSE"]))
    print()

    # Check if data fits the model and re-analyze if needed.
    mae_percent_range = []
    for c in range(0, len(metrics_v3)):
        mae_percent_range.append(metrics_v3[c]["MAE"]/(metrics_v3[c]["MAX"]-metrics_v3[c]["MIN"]))
    # Threshold value for deeper analysis as MAE percent of total range.
    if any(value > 0.10 for value in mae_percent_range):
        hf_loop_limit = 5
        hf_loop_count = 1
        while hf_loop_count <= hf_loop_limit:
            hf_sweep_initial = sweep_initial*(5*hf_loop_count)
            hf_sweep_detailed = sweep_detailed*(int(0.5*hf_loop_count)+1)

            hf_analysis_parameters = copy.deepcopy(analysis_parameters)
            hf_analysis_parameters["sweep_initial"] = hf_sweep_initial
            hf_analysis_parameters["sweep_detailed"] = hf_sweep_detailed

            # Perform higher fidelity analysis on initial model.
```

```
            print("############################################################")
            print("Poor fit: higher fidelity analysis required")
            print("Increasing initial sweep multiples from {:d} to {:d}".format(sweep_initial, hf_sweep_initial))
            print("############################################################")
            print()
            print("High fidelity analysis iteration " + str(hf_loop_count))
            print("============================================================")
            model_function_v1, new_mask = analyze_model(hf_analysis_parameters, model_dictionary_v1,
                                                        input_data, output_data, input_mask)
            metrics_v1 = evaluate_function(model_function_v1, input_data, output_data)

            for channel_id, channel_metrics in enumerate(metrics_v1):
                print("Channel y" + str(channel_id+1) + " metrics")
                print("MAE  : " + str(FORMAT%channel_metrics["MAE"]))
                print("RMSE : " + str(FORMAT%channel_metrics["RMSE"]))
                print("MAX  : " + str(FORMAT%channel_metrics["MAX"]))
                print("MIN  : " + str(FORMAT%channel_metrics["MIN"]))
                print()

            # Perform higher fidelity analysis on retrained model.
            print("Masked retraining and analysis")
            print("============================================================")
            print("New mask")
            print(new_mask)
            print()
            model_dictionary_v2 = create_model(model_parameters, input_data, output_data, new_mask)
            model_function_v2, _ = analyze_model(hf_analysis_parameters, model_dictionary_v2,
                                                 input_data, output_data, new_mask)
            metrics_v2 = evaluate_function(model_function_v2, input_data, output_data)

            for channel_id, channel_metrics in enumerate(metrics_v2):
                print("Channel y" + str(channel_id+1) + " metrics")
                print("MAE  : " + str(FORMAT%channel_metrics["MAE"]))
                print("RMSE : " + str(FORMAT%channel_metrics["RMSE"]))
                print("MAX  : " + str(FORMAT%channel_metrics["MAX"]))
                print("MIN  : " + str(FORMAT%channel_metrics["MIN"]))
                print()

            # Combined analysis evaluation.
            model_function_v3 = []
            metrics_v3 = []
            for c in range(0, len(metrics_v2)):
                if metrics_v2[c]["MAE"] < metrics_v1[c]["MAE"]:
                    # Retrained model results are better.
                    print("Channel y" + str(c+1) + " improved")
                    model_function_v3.append(model_function_v2[c])
                    metrics_v3.append(metrics_v2[c])
                else:
                    # Initial model results are better.
                    print("Channel y" + str(c+1) + " did not improve")
                    model_function_v3.append(model_function_v1[c])
                    metrics_v3.append(metrics_v1[c])
                print("MAE  : " + str(FORMAT%metrics_v1[c]["MAE"]) + " -> " + str(FORMAT%metrics_v2[c]["MAE"]))
                print("RMSE : " + str(FORMAT%metrics_v1[c]["RMSE"]) + " -> " + str(FORMAT%metrics_v2[c]["RMSE"]))
            print()

            # Check if the data fits the model.
            mae_percent_range = []
            for c in range(0, len(metrics_v3)):
                mae_percent_range.append(metrics_v3[c]["MAE"]/(metrics_v3[c]["MAX"]-metrics_v3[c]["MIN"]))
            if all(value < 0.10 for value in mae_percent_range):
                print("Higher fidelity analysis was successful")
                break

            # Set a limit on depth of analysis.
            hf_loop_count = hf_loop_count + 1
            if hf_loop_count == hf_loop_limit:
                print("WARNING: Higher fidelity analysis failed to fit data,")
                print("         parameter tuning will still be attempted")
                break
        print()

    print("Genetic algorithm tuning")
    print("============================================================")
    tuning_parameters = {"GA_population": analysis_parameters["GA_population"],
                         "GA_generations": analysis_parameters["GA_generations"],
                         "visual": analysis_parameters["visual"],
                         "save_visual": analysis_parameters["save_visual"]}
    model_function_v4 = tune_model(tuning_parameters, model_function_v3, input_data, output_data)
    metrics_v4 = evaluate_function(model_function_v4, input_data, output_data)

    model_function_v5 = []
    metrics_v5 = []
    for c in range(0,len(metrics_v4)):
        if metrics_v4[c]["MAE"] < metrics_v3[c]["MAE"]:
            print("Channel y" + str(c+1) + " improved")
            model_function_v5.append(model_function_v4[c])
            metrics_v5.append(metrics_v4[c])
```

```
        else:
            print("Channel y" + str(c+1) + " did not improve")
            model_function_v5.append(model_function_v3[c])
            metrics_v5.append(metrics_v3[c])
        print("MAE  : " + str(FORMAT%metrics_v3[c]["MAE"]) + " -> " + str(FORMAT%metrics_v4[c]["MAE"]))
        print("RMSE : " + str(FORMAT%metrics_v3[c]["RMSE"]) + " -> " + str(FORMAT%metrics_v4[c]["RMSE"]))

    print()

    print("Final estimation")
    print("============================================================")
    # Print GA tuned equations.
    for idc, channel_function in enumerate(model_function_v5):
        y_str = "y" + str(idc+1) + "[k] = "
        for idf, product_function in enumerate(channel_function):
            if product_function["estimate_string"] != None:
                y_str = y_str + product_function["estimate_string"]
                if idf < len(channel_function) - 1:
                    y_str = y_str + " + "
        print(y_str)
        print()
```

Listing A.3: The `create_model()` function which creates and trains a TCN model.

```
# Generate a TCN model from input output data.
#
# Input and output are time series, each row is a time entry and each column
# is a dimension of the data. Most recent data is the last column.
#
# Example input:                Example output:
# x1[k-n] x2[k-n] x3[k-n]       y1[k-n] y2[k-n]
# ...     ...     ...           ...     ...
# x1[k-2] x2[k-2] x3[k-2]       y1[k-2] y2[k-2]
# x1[k-1] x2[k-1] x3[k-1]       y1[k-1] y2[k-1]
# x1[k]   x2[k]   x3[k]         y1[k]   y2[k]
#
# Training data is rearanged into input 'X_train' and output 'Y_train'.
# Each row of 'X_train' is a sample of length 'history' and dimension
# 'input_channels' in the following format (ex: history = 4):
#
# x1[k-3] x1[k-2] x1[k-1] x1[k]
# x2[k-3] x2[k-2] x2[k-1] x2[k]
# x3[k-3] x3[k-2] x3[k-1] x3[k]
#
# Note that this matrix is flipped right-left in the dissertation.
#
# Each row of 'Y_train' is a sample of length 1 and dimension
# 'output_channels' in the following format:
# y1[k]
# y2[k]
#
# The goal of the TCN is to predict y[k] given x[k] .. x[k-n]
#
# The mask is in the same format as 'X_train' and determines which inputs
# are hidden from the TCN. Element-wise multiplication is used to set input
# elements to 0 with the mask.

import torch
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
import time
import os
import matplotlib.pyplot as plt
from lib.model import TCN
import pyprind # Progress bar

FORMAT = '%.3e'

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

def create_model(model_parameters, inputData, outputData, inputMask=1):

    epochs = model_parameters["epochs"]                     # upper epoch limit
    cuda = model_parameters["cuda"]                         # use the GPU
    dropout = model_parameters["dropout"]                   # dropout applied to layers
    clip = model_parameters["clip"]                         # -1 means no clip
    ksize = model_parameters["ksize"]                       # kernel size
    levels = model_parameters["levels"]                     # number of levels
    nhid = model_parameters["nhid"]                         # number of hidden units per layer
    batch_size_train = model_parameters["batch_size_train"] # batch size for training
    train_loss_full = model_parameters["train_loss_full"]   # [1] full loss, [0] avg over batches
    lr = model_parameters["lr"]                             # learning rate
    lr_grad_period = model_parameters["lr_grad_period"]     # decay period for learning rate
    lr_grad_rate = model_parameters["lr_grad_rate"]         # decay multiplier after decay period
```

```
    optimizer = model_parameters["optimizer"]                # optimizer to use
    history = model_parameters["history"]                    # number of time sample inputs to the network
    test_data = model_parameters["test_data"]                # test data proportion
    seed = model_parameters["seed"]                          # random seed
    visual = model_parameters["visual"]                      # plot training metrics
    save_visual = model_parameters["save_visual"]            # save training metric plot

    torch.manual_seed(seed)

    # Get the current data output folder if saving data and plots.
    if save_visual:
        if not os.path.exists('./output'):
            os.mkdir('./output')
        model_dir_count = 1
        while os.path.exists('./output/model_{}'.format(model_dir_count)):
            model_dir_count = model_dir_count + 1
        os.mkdir('./output/model_{}'.format(model_dir_count))

    # Create the TCN network.
    print("Initializing TCN model...")
    history_eff = (ksize-1)*(pow(2,levels+1)-1)-(ksize-1) + 1
    input_length = inputData.shape[0]
    output_length = outputData.shape[0]
    if input_length != output_length:
        print("Input and output data must be the same length.")
        exit
    input_channels = inputData.shape[1]     # dimension of x[k]
    output_channels = outputData.shape[1]   # dimension of y[k]
    channel_sizes = [nhid]*levels
    model = TCN(input_channels, output_channels, channel_sizes, ksize, dropout=dropout)
    print("The network has ", levels, " hidden levels with a kernel size of ", ksize, sep='')
    print("The oldest input that can be seen is x[k-", history_eff-1, "]", sep='')
    if history_eff > history: print("WARNING: Effective history exceeds input history,")
    if history_eff > history: print("         inputs beyond x[k-" + str(history-1) + "] are not visible")
    print("The network contains " + str(count_parameters(model)) + " parameters")
    print()

    # Format the training and test data.
    samples = input_length - history + 1
    train_samples = int(np.round((1-test_data)*samples))
    test_samples = samples-train_samples
    X_train = np.zeros([train_samples, input_channels, history])
    Y_train = np.zeros([train_samples, output_channels])
    X_test = np.zeros([test_samples, input_channels, history])
    Y_test = np.zeros([test_samples, output_channels])
    input_range = []
    input_shift = []
    shiftedInputData = np.copy(inputData)
    for i in range(0,input_channels):
        minRange = min(inputData[:,i])
        maxRange = max(inputData[:,i])
        if minRange <= 0 and maxRange >= 0:
            # No shifting required, the zero point is in the input range.
            input_range.append([minRange, maxRange])
            input_shift.append(0)
        else:
            # Adjust the input channel to center it about zero.
            shift = (minRange + maxRange)/2
            shiftedInputData[:,i] = shiftedInputData[:,i] - shift
            input_shift.append(shift)
            input_range.append([minRange-shift, maxRange-shift])
    for i in range(0,train_samples):
        X_train[i,:,:] = shiftedInputData[i:(i+history)].T
        Y_train[i,:] = outputData[i+history-1].T
    for i in range(train_samples,samples):
        X_test[i-train_samples,:,:] = shiftedInputData[i:(i+history)].T
        Y_test[i-train_samples,:] = outputData[i+history-1].T

    # Normalize data to be mean centered with unit covariance.
    # Apply input mask.
    mu_x = np.mean(X_train,axis=0)
    sig_x = np.var(X_train,axis=0)
    mu_y = np.mean(Y_train,axis=0)
    sig_y = np.var(Y_train,axis=0)
    mu_y_t = torch.tensor(mu_y, dtype=torch.float)
    sig_y_t = torch.tensor(sig_y, dtype=torch.float)
    X_train = torch.tensor(((X_train-mu_x)/sig_x)*inputMask,dtype=torch.float)
    Y_train = torch.tensor((Y_train-mu_y)/sig_y,dtype=torch.float)
    X_test = torch.tensor(((X_test-mu_x)/sig_x)*inputMask,dtype=torch.float)
    Y_test = torch.tensor((Y_test-mu_y)/sig_y,dtype=torch.float)

    # Move data to the GPU if one is present.
    if cuda:
        model.cuda()
        X_train = X_train.cuda()
        Y_train = Y_train.cuda()
        X_test = X_test.cuda()
        Y_test = Y_test.cuda()
```

```
        mu_y_t = mu_y_t.cuda()
        sig_y_t = sig_y_t.cuda()

    # Define the training function.
    optimizer = getattr(optim, optimizer)(model.parameters(), lr=lr)
    def train(epoch):
        model.train()
        batch_idx = 1
        epoch_loss = 0
        for i in range(0, X_train.size()[0], batch_size_train):
            if i + batch_size_train > X_train.size()[0]:
                x, y = X_train[i:], Y_train[i:]
            else:
                x, y = X_train[i:(i+batch_size_train)], Y_train[i:(i+batch_size_train)]
            optimizer.zero_grad()
            output = model(x)
            loss = F.mse_loss(output*sig_y_t+mu_y_t, y*sig_y_t+mu_y_t)
            loss.backward()
            if clip > 0:
                torch.nn.utils.clip_grad_norm(model.parameters(), clip)
            optimizer.step()
            batch_idx += 1
            epoch_loss += loss.data.item()
        return epoch_loss / (batch_idx - 1)

    # Define test loss function.
    def evaluateTestLoss():
        model.eval()
        output = model(X_test)
        test_loss = F.mse_loss(output*sig_y_t+mu_y_t, Y_test*sig_y_t+mu_y_t)
        return test_loss.data.item()

    # Train the network
    print("Training TCN model...")
    progress_bar = pyprind.ProgBar(2*epochs, monitor=True)
    testLossHistory = list()
    trainLossHistory = list()
    for ep in range(1, epochs+1):
        trainloss = train(ep)
        testloss = evaluateTestLoss()
        if train_loss_full == 1:
            # Calculate training loss over all training data. More accurate but uses more RAM.
            if cuda : model.cpu()
            trainloss = F.mse_loss(model(X_train.cpu())*sig_y_t+mu_y_t, Y_train.cpu()*sig_y_t+mu_y_t)
            if cuda : model.cuda()
        testLossHistory.append(testloss)
        trainLossHistory.append(trainloss)
        if ep % lr_grad_period == 0:
            lr = lr*lr_grad_rate
        progress_bar.update()
        progress_bar.update()
    time.sleep(0.5)
    print()

    # Plot training and test loss.
    if save_visual == True or visual == True:
        plt.figure()
        ax = plt.subplot(1,1,1)
        plt.plot(trainLossHistory)
        plt.plot(testLossHistory)
        plt.legend(['Training Loss','Test Loss'])
        plt.title('Training and Test Loss')
        plt.xlabel('Epoch')
        plt.ylabel('MSE Loss')
        ax.set_yscale("log", nonposy='clip')
        if save_visual == True: plt.savefig('./output/model_{}/loss.pdf'.format(model_dir_count))
        if visual == True: plt.show()
    print("Min train: epoch " + str(np.argmin(trainLossHistory)+1))
    print("Min test: epoch " + str(np.argmin(testLossHistory)+1))
    print()

    # Calculate data fit metrics on the CPU.
    model.cpu()
    X_data = torch.cat([X_train.cpu(), X_test.cpu()], dim=0)
    Y_data = torch.cat([Y_train.cpu(), Y_test.cpu()], dim=0)
    Y_pred = np.zeros([samples, output_channels])
    Y_pred = torch.tensor(Y_pred, dtype=torch.float)
    batch_size_test = 128
    for i in range(0, samples, batch_size_test):
        if i + batch_size_test > samples:
            Y_pred[i:] = model(X_data[i:])
        else:
            Y_pred[i:(i+batch_size_test)] = model(X_data[i:(i+batch_size_test)])
    Y_err = (Y_data.detach().cpu().numpy()*sig_y+mu_y) - (Y_pred.detach().cpu().numpy()*sig_y+mu_y)
    Y_err_train = Y_err[0:train_samples]
    Y_err_test = Y_err[train_samples:]
    mae = []
    mse = []
```

```
    rmse = []
    for i in range(0, output_channels):
        mae.append({})
        mse.append({})
        rmse.append({})
        mae[i]["total"] = (np.mean(abs(Y_err[:, i])))
        mae[i]["train"] = (np.mean(abs(Y_err_train[:, i])))
        mae[i]["test"] = (np.mean(abs(Y_err_test[:, i])))
        mse[i]["total"] = (sum(pow(Y_err[:, i], 2))/len(Y_err[:, i]))
        mse[i]["train"] = (sum(pow(Y_err_train[:, i], 2))/len(Y_err_train[:, i]))
        mse[i]["test"] = (sum(pow(Y_err_test[:, i], 2))/len(Y_err_test[:, i]))
        rmse[i]["total"] = np.sqrt(mse[i]["total"])
        rmse[i]["train"] = np.sqrt(mse[i]["train"])
        rmse[i]["test"] = np.sqrt(mse[i]["test"])
        print("TCN channel " + str(i+1) + " metrics")
        print("Total MAE: " + str(FORMAT%mae[i]["total"]) + " MSE: " + \
              str(FORMAT%mse[i]["total"]) + " RMSE: " + str(FORMAT%rmse[i]["total"]))
        print("Train MAE: " + str(FORMAT%mae[i]["train"]) + " MSE: " + \
              str(FORMAT%mse[i]["train"]) + " RMSE: " + str(FORMAT%rmse[i]["train"]))
        print("Test  MAE: " + str(FORMAT%mae[i]["test"]) + " MSE: " + \
              str(FORMAT%mse[i]["test"]) + " RMSE: " + str(FORMAT%rmse[i]["test"]))
    print()

    # Assemble the function output.
    modelDictionary = {
        "model": model,
        "model_parameters": model_parameters,
        "history_eff": history_eff,
        "mu_x": mu_x,
        "sig_x": sig_x,
        "mu_y": mu_y,
        "sig_y": sig_y,
        "input_channels": input_channels,
        "output_channels": output_channels,
        "input_range": input_range,
        "mae": mae,
        "mse": mse,
        "rmse": rmse,
        "input_shift": input_shift
    }

    return modelDictionary
```

Listing A.4: The `analyze_model()` function which analyzes a trained TCN model.

```
# Analyze a model to generate an equation.
#
# Input is the model, template fitting functions, and the sweep set.
# The sweep set is an array of multiples to use in the fitting process.

import time
import itertools
import multiprocessing
import numpy as np
from joblib import Parallel, delayed
from scipy.optimize import curve_fit
import pyprind # Progress bar
import torch
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import mat4py
import os

from lib.evaluate_function import evaluate_function

# Format of function parameters.
FORMAT = '%.3e'

def analyze_model(analysis_parameters, model_dictionary, input_data, output_data, input_mask=1):

    functions = analysis_parameters["functions"]
    sweep_initial = analysis_parameters["sweep_initial"]
    sweep_detailed = analysis_parameters["sweep_detailed"]
    contrib_thresh = analysis_parameters["contrib_thresh"]
    contrib_thresh_omit = analysis_parameters["contrib_thresh_omit"]
    use_f_weight = analysis_parameters["use_f_weight"]
    seed = analysis_parameters["seed"]
    np.random.seed(seed)
    verbose = analysis_parameters["verbose"]
    visual = analysis_parameters["visual"]
    save_visual = analysis_parameters["save_visual"]

    # Check inputs for validity.
    if sweep_initial < 1:
        print("ERROR: analyze_model parameter sweep_initial must be >= 1")
        return None, None
```

```
    if sweep_detailed < 100:
        print("ERROR: analyze_model parameter sweep_detailed must be >= 100")
        return None, None

    # Function for indexing the large impulse array.
    def array_to_int(num_list):          # [1,2,3]
        str_list = map(str, num_list)    # ['1','2','3']
        num_str = ''.join(str_list)      # '123'
        num = int(num_str, 2)            # 123
        return num

    model = model_dictionary["model"]
    history = model_dictionary["model_parameters"]["history"]
    history_eff = model_dictionary["history_eff"]
    mu_x = model_dictionary["mu_x"]
    sig_x = model_dictionary["sig_x"]
    mu_y = model_dictionary["mu_y"]
    sig_y = model_dictionary["sig_y"]
    input_channels = model_dictionary["input_channels"]
    output_channels = model_dictionary["output_channels"]
    input_range = model_dictionary["input_range"]
    input_shift = model_dictionary["input_shift"]

    # Establish tensor types of certain variables for computation.
    mu_y_t = torch.tensor(mu_y, dtype=torch.float)
    sig_y_t = torch.tensor(sig_y, dtype=torch.float)

    # Get the current data output folder if saving data and plots.
    if save_visual == True:
        if not os.path.exists('./output'):
            os.mkdir('./output')
        analysis_dir_count = 1
        while os.path.exists('./output/analysis_{}'.format(analysis_dir_count)):
            analysis_dir_count = analysis_dir_count + 1
        os.mkdir('./output/analysis_{}'.format(analysis_dir_count))

    # Generate every possible combination of impulses.
    if history < history_eff:
        history_eff = history
    combination_count = pow(2, input_channels*(history_eff))
    combinations = [x for x in range(0, input_channels*(history_eff))]
    impulse_array = np.zeros([combination_count, input_channels, history])
    # Loop through every combination of subsets of constituants
    for combination_id in range(0, len(combinations)+1):
        for subset in itertools.combinations(combinations, combination_id):
            impulse = np.zeros([1, 1, input_channels*(history_eff)])
            for element in subset:
                impulse[0, 0, input_channels*(history_eff)-1-element] = 1
            index = array_to_int(impulse[0, 0, :].astype(int))
            impulse_shaped = np.reshape(impulse, [input_channels, history_eff])
            # Add buffer elements to account for a history longer than scope.
            impulse_array[index, :, (history-history_eff):history] = impulse_shaped

    # Generate the impulse sweep set for creating multiples of impulses.
    if sweep_initial != 1:
        impulse_sweep_set = 2*np.random.rand(sweep_initial, input_channels, history)-1
        # Bound sweep set to be within range of the original input data.
        for i in range(0, input_channels):
            min_value = input_range[i][0]
            max_value = input_range[i][1]
            impulse_sweep_set[:, i, :] = impulse_sweep_set[:, i, :]*(max_value-min_value)+min_value

    # Obtain the output for input impulses.
    print("Exciting model...")
    model.cpu()
    if sweep_initial != 1:
        impulse_response = np.zeros([combination_count, output_channels, sweep_initial])
    else:
        impulse_response = np.zeros([combination_count, output_channels, 1])
    batch_idx = 1
    batch_size_analyze = 256
    progress_bar = pyprind.ProgBar(len(range(0, combination_count, batch_size_analyze)), monitor=True)
    # Calculate the bias at the zero point.
    model_input = np.copy(impulse_array[0:1, :, :])
    bias = model(torch.tensor((model_input-mu_x)/sig_x, dtype=torch.float))*sig_y_t+mu_y_t

    # Calculate the response from all impulse combinations.
    for i in range(0, combination_count, batch_size_analyze):
        if i + batch_size_analyze > combination_count:
            # Handle the last batch.
            impulse = impulse_array[i:]
            if sweep_initial > 1:
                for j in range(0, sweep_initial):
                    mult = impulse_sweep_set[j, :, :]
                    model_input = mult*impulse*input_mask
                    output = (model(torch.tensor((model_input-mu_x)/sig_x, dtype=torch.float))*sig_y_t+mu_y_t).
 detach().cpu().numpy()
                    impulse_response[i:, :, j] = output
```

```
            else:
                model_input = impulse*input_mask
                output = (model(torch.tensor((model_input-mu_x)/sig_x, dtype=torch.float))*sig_y_t+mu_y_t).detach
    ().cpu().numpy()
                impulse_response[i:, :, 0] = output
        else:
            # Handle a standard size batch.
            impulse = impulse_array[i:(i+batch_size_analyze)]
            if sweep_initial > 1:
                for j in range(0, sweep_initial):
                    mult = impulse_sweep_set[j, :, :]
                    model_input = mult*impulse*input_mask
                    output = (model(torch.tensor((model_input-mu_x)/sig_x, dtype=torch.float))*sig_y_t+mu_y_t).
    detach().cpu().numpy()
                    impulse_response[i:(i+batch_size_analyze), :, j] = output
            else:
                model_input = impulse*input_mask
                output = (model(torch.tensor((model_input-mu_x)/sig_x, dtype=torch.float))*sig_y_t+mu_y_t).detach
    ().cpu().numpy()
                impulse_response[i:(i+batch_size_analyze), :, 0] = output
        batch_idx += 1
        progress_bar.update()
    #impulse_response = impulse_response.detach().numpy()
    time.sleep(0.5) # Allows progress bar to finish printing elapsed time.
    print()

    def process_subcombination(subcombination):
        sub_impulse = np.zeros([input_channels*history])
        # Determine index of combination in impulse_response
        for element in subcombination:
            sub_impulse[input_channels*history-1-element] = 1
        sub_index = array_to_int(sub_impulse.astype(int))
        # Loop through all subcombinations
        subsub_indices = []
        for l in range(0, len(subcombination)+1):
            for subsubcombination in itertools.combinations(subcombination, l):
                if subcombination != subsubcombination:
                    subsub_impulse = np.zeros([input_channels*history])
                    # Determine index of subcombination in impulse_response
                    for element in subsubcombination:
                        subsub_impulse[input_channels*history-1-element] = 1
                    subsub_index = array_to_int(subsub_impulse.astype(int))
                    subsub_indices.append(subsub_index)
        return sub_index, subsub_indices

    # Analyze responses (note: progress bar is not linear with computation time)
    print("Analyzing responses...")
    progress_bar = pyprind.ProgBar(combination_count, monitor=True)
    num_cores = multiprocessing.cpu_count()
    for combination_id in range(0, len(combinations)+1):
        # Loop all combinations
        results = Parallel(n_jobs=num_cores)(delayed(process_subcombination)(subcombination) \
                          for subcombination in itertools.combinations(combinations, combination_id))
        for each in results:
            sub_index = each[0]
            subsub_indices = each[1]
            for subsub_index in subsub_indices:
                impulse_response[sub_index, :, :] = impulse_response[sub_index, :, :] - \
                                                    impulse_response[subsub_index, :, :]
            progress_bar.update()
    time.sleep(0.5) # Allows progress bar to finish printing elapsed time.
    print()

    # Examine the impulse response for all combinations and generate a function.
    print("Estimating system equation...")
    # Create a mask of relevant inputs for later model retraining.
    new_mask = np.zeros([input_channels, history])
    # Create a sweep set for curve fitting.
    fit_sweep_set = np.random.rand(sweep_detailed, input_channels, history)
    for i in range(0, input_channels):
        min_value = input_range[i][0]
        max_value = input_range[i][1]
        fit_sweep_set[:, i, :] = fit_sweep_set[:, i, :]*(max_value-min_value)+min_value
    model_function = []
    for channel_id in range(0, output_channels):
        # Function for the output channel is a sum of product functions.
        channel_function = []
        # Get the magnitude average point value of each product function contribution.
        Z = np.sum(abs(impulse_response[:, channel_id, :]), 1)/sweep_initial
        # Get the variance of each product function.
        S = np.var(impulse_response[:, channel_id, :], 1)
        total_variance = sum(S)
        # Get indices of responses from largest to smallest.
        response_indices = np.flip(np.argsort(Z), 0)
        # Get indices of variances from largest to smallest.
        variance_indices = np.flip(np.argsort(S), 0)

        # Identify the top responses.
```

```
        if verbose:
            print("############################################################")
            print("Estimate of channel " + str(channel_id+1))
            print("############################################################")
        candidate_limit = min(25, len(response_indices))
        sig_indexes = []
        for k in range(0, candidate_limit):
            sig_index = response_indices[k]
            sig_response = Z[sig_index]
            z_sorted = np.flip(np.sort(Z[1:], 0), 0)
            contribution_magnitude = sig_response/sum(z_sorted)
            if contribution_magnitude > contrib_thresh:
                sig_indexes.append(sig_index)
        for k in range(0, candidate_limit):
            sig_index = variance_indices[k]
            sig_variance = S[sig_index]
            contribution_variance = sig_variance/total_variance
            if contribution_variance > contrib_thresh and sig_index not in sig_indexes:
                sig_indexes.append(sig_index)

        # Estimate equations for top responses.
        for sig_index in sig_indexes:
            sig_response = Z[sig_index]
            sig_variance = S[sig_index]
            sig_impulse = impulse_array[sig_index:sig_index+1, :, :]
            if verbose: print("Response ID " + str(sig_index) + " contribution:")

            # Process a product function if the response is significant.
            # Significance is % contribution to total magnitude or variance.
            # Bias is not included in magnitude significance.
            z_sorted = np.flip(np.sort(Z[1:], 0), 0)
            contribution_magnitude = sig_response/sum(z_sorted)
            contribution_variance = sig_variance/total_variance
            if sig_index is not 0:
                if verbose: print("Magnitude : " + str('%.1f'%(contribution_magnitude*100)) + "%")
                if verbose: print("Variance  : " + str('%.1f'%(contribution_variance*100)) + "%")
            else:
                if verbose: print("Bias contribution omitted from calculation.")
            if verbose: print("============================================================")
            if contribution_magnitude > contrib_thresh or contribution_variance > contrib_thresh:

                # Determine the arguments of the product function.
                arg_list = []
                for input_id in range(0, input_channels):
                    for element_id, element in enumerate(sig_impulse[0, input_id, :].astype(int)):
                        if element == 1:
                            delay = history - 1 - element_id
                            arg_list.append({"input_channel": input_id, "delay": delay})
                            new_mask[input_id, element_id] = 1

                # Create the product function template string.
                f_list = []
                f_str = "f("
                for _, arg in enumerate(arg_list):
                    f_list.append("x" + str(arg["input_channel"]+1) + "(k-" + str(arg["delay"]) + ")")
                for arg_num, arg_str in enumerate(f_list):
                    f_str = f_str + arg_str
                    if arg_num < len(f_list)-1:
                        f_str = f_str + ","
                if len(arg_list) == 0:
                    f_str = f_str + "0"
                f_str = f_str + ")"

                # Estimate the product function.
                def fcn_empty(_):
                    return 0
                def txt_empty(_):
                    return ""
                dct_empty = {
                    "txt": "?",
                    "txt_fcn": txt_empty,
                    "fcn": fcn_empty,
                    "upper": [],
                    "lower": [],
                    "weight": 1.0
                }
                product_function = {
                    "arg_list": arg_list,
                    "template_string": f_str,
                    "estimate_string": "f(?)",
                    "parameters": [],
                    "function": dct_empty,
                    "shift": []
                }
                if len(arg_list) > 0:
                    # Obtain sample points for curve fitting.
                    x_data = np.zeros([sweep_detailed, input_channels, history])
                    y_data = np.zeros([sweep_detailed, output_channels])
```

```
                    for idx in range(0, sweep_detailed):
                        mult = fit_sweep_set[idx, :, :]
                        model_input = mult*sig_impulse*input_mask
                        x_data[idx, :, :] = model_input
                        y_data[idx, :] = (model(torch.tensor((model_input-mu_x)/sig_x, dtype=torch.float))).
detach().numpy()*sig_y+mu_y
                    # Recursively subtract contributions from product functions of arguments.
                    contribution_list = []
                    for idf in range(0, len(arg_list)):
                        new_contributions = []
                        for arg_combination in itertools.combinations(arg_list, idf):
                            arg_impulse = np.zeros([sweep_detailed, input_channels, history])
                            for arg in arg_combination:
                                arg_impulse[:, arg["input_channel"], history-1-arg["delay"]] = 1
                            model_input = arg_impulse * fit_sweep_set
                            output = (model(torch.tensor((model_input-mu_x)/sig_x, dtype=torch.float))).detach().
numpy()*sig_y+mu_y
                            for contribution in contribution_list:
                                output = output - contribution
                            new_contributions.append(output)
                        contribution_list[0:0] = new_contributions
                    for contribution in contribution_list:
                        y_data = y_data - contribution

                    # Format data for curve fitting
                    arg_count = len(arg_list)
                    x_data_fit = np.zeros([arg_count, sweep_detailed])
                    y_data_fit = np.zeros([sweep_detailed])
                    arg = 0
                    for i in range(0, input_channels):
                        for j in range(0, history):
                            if sig_impulse[0, i, j] == 1:
                                x_data_fit[arg, :] = x_data[:, i, j]
                                y_data_fit[:] = y_data[:, channel_id]
                                product_function["shift"].append(input_shift[i])
                                arg = arg + 1

                    # Plot 2D and 3D data for visual inspection.
                    if save_visual == True or visual == True:
                        if arg_count == 1:
                            plt.figure()
                            plt.scatter(x_data_fit[0], y_data_fit, marker='.')
                            plt.title(product_function["template_string"])
                            plt.xlabel(f_list[0])
                            if save_visual == True:
                                plt.savefig('./output/analysis_{}/{}.pdf'.format(analysis_dir_count, \
                                            product_function["template_string"]))
                                pltDict = {"x": x_data_fit[0].tolist(),
                                       "y": y_data_fit.tolist()}
                                mat4py.savemat('./output/analysis_{}/{}.mat'.format(analysis_dir_count, \
                                               product_function["template_string"]), pltDict)
                            if visual == True: plt.show()
                        if arg_count == 2:
                            plt.figure()
                            ax = plt.axes(projection='3d')
                            ax.scatter3D(x_data_fit[0], x_data_fit[1], y_data_fit, c=y_data_fit, marker='o')
                            ax.set_title(product_function["template_string"])
                            ax.set_xlabel(f_list[0])
                            ax.set_ylabel(f_list[1])
                            if save_visual == True:
                                plt.savefig('./output/analysis_{}/{}.pdf'.format(analysis_dir_count, \
                                            product_function["template_string"]))
                                pltDict = {"x": x_data_fit[0].tolist(),
                                       "y": x_data_fit[1].tolist(),
                                       "z": y_data_fit.tolist()}
                                mat4py.savemat('./output/analysis_{}/{}.mat'.format(analysis_dir_count, \
                                               product_function["template_string"]), pltDict)
                            if visual == True: plt.show()

                    # Estimate the product function using curve fitting.
                    if arg_count in functions:
                        candidate_functions = functions[arg_count]
                    else:
                        candidate_functions = []
                        product_function["estimate_string"] = product_function["template_string"]
                    best_fit = 100
                    for f in candidate_functions:
                        try:
                            popt, pcov = curve_fit(f["fcn"],
                                                   x_data_fit,
                                                   y_data_fit,
                                                   bounds=(f["lower"], f["upper"]),
                                                   maxfev=250000)
                            pcount = len(popt)
                            err = y_data_fit-f["fcn"](x_data_fit, *popt)
                            # Compute root mean squared error.
                            rmse = np.sqrt(sum(pow(err, 2))/sweep_detailed)
                            # Compute mean average error.
```

```
                            mae = np.mean(abs(err))
                            # Compute one standard deviation errors (just the normal std).
                            #std = np.sqrt(np.diag(pcov))
                            if verbose:
                                print("Fit for " + f["txt_fcn"](arg_list, product_function["shift"], *popt))
                                print("MAE  : " + str(FORMAT%mae))
                                print("RMSE : " + str(FORMAT%rmse))
                                #print("STD  : " + str(std))
                            f_weight = 1.0
                            if use_f_weight == True: f_weight = f["weight"]
                            if mae/f_weight < best_fit:
                                best_fit = mae/f_weight
                                product_function["parameters"] = popt
                                product_function["function"] = f
                                product_function["estimate_string"] = f["txt_fcn"](arg_list,
                                                                                   product_function["shift"],
                                                                                   *popt)
                                if verbose: print("Current best fit for Response " + str(sig_index))
                            if verbose: print()
                            # Perform curve fitting with different parameter initializations in attempt to
    improve fit.
                            fit_iterations = 5*pcount
                            for _ in range(1, fit_iterations):
                                pinit = np.random.rand(pcount)*(np.array(f["upper"])-np.array(f["lower"])) \
                                        + np.array(f["lower"])
                                popt_new, pcov = curve_fit(f["fcn"],
                                                           x_data_fit,
                                                           y_data_fit,
                                                           bounds=(f["lower"], f["upper"]),
                                                           p0=pinit,
                                                           maxfev=10000)
                                err = y_data_fit-f["fcn"](x_data_fit, *popt_new)
                                # Compute root mean squared error.
                                rmse = np.sqrt(sum(pow(err, 2))/sweep_detailed)
                                # Compute mean average error.
                                mae = np.mean(abs(err))
                                if mae/f_weight < 0.999*best_fit:
                                    best_fit = mae/f_weight
                                    product_function["parameters"] = popt_new
                                    product_function["function"] = f
                                    product_function["estimate_string"] = f["txt_fcn"](arg_list,
                                                                                       product_function["shift"],
                                                                                       *popt_new)
                                    if verbose:
                                        print("Revised fit for " + f["txt_fcn"](arg_list,
                                                                                product_function["shift"],
                                                                                *popt_new))
                                        print("MAE  : " + str(FORMAT%mae))
                                        print("RMSE : " + str(FORMAT%rmse))
                                        print("Current best fit for Response " + str(sig_index))
                                        print()
                        except Exception as e:
                            if best_fit == 100:
                                product_function["estimate_string"] = product_function["template_string"]
                            if verbose:
                                print("Warning: Fit could not be estimated for " + f["txt"] + ",")
                                print("         " + str(e))
                                print("")
                    else:
                        # Handle constant bias at the zero point.
                        channel_bias = bias[0, channel_id].detach().numpy()
                        channel_bias_str = str('%.3f'%channel_bias)
                        product_function["parameters"] = [channel_bias]
                        def fcn_bias(x, a):
                            return a
                        def txt_bias(argList, argShift, a):
                            return str('%.3f'%a)
                        dct_bias = {
                            "txt": "a",
                            "fcn": fcn_bias,
                            "txt_fcn": txt_bias,
                            "upper": [2*channel_bias],
                            "lower": [0],
                            "weight": 1.0
                        }
                        product_function["function"] = dct_bias
                        product_function["estimate_string"] = channel_bias_str
                        if verbose:
                            print("Constant " + channel_bias_str)
                            print()

                    # Check if the candidate product function improves the accuracy of the model.
                    if sig_index > 0:
                        current_function = [channel_function]
                        candidate_function = [channel_function + [product_function]]
                        current_metrics = evaluate_function(current_function,
                                                            input_data,
                                                            output_data[:, channel_id:channel_id+1])
```

```
                    candidate_metrics = evaluate_function(candidate_function,
                                                          input_data,
                                                          output_data[:, channel_id:channel_id+1])

                    # Include product functions that are above a threshold and improve the overall MAE.
                    if candidate_metrics[0]["MAE"] > current_metrics[0]["MAE"]:
                        if verbose:
                            print("Warning: Candidate product function worsens overall MAE.")
                            print("         MAE increases from " + str(FORMAT%current_metrics[0]["MAE"])+\
                                  " to " + str(FORMAT%candidate_metrics[0]["MAE"]) + ".")
                        if contribution_magnitude < contrib_thresh_omit \
                        and contribution_variance < contrib_thresh_omit:
                            if verbose: print("         Candidate product function omitted.")
                        else:
                            channel_function.append(product_function)
                            if verbose: print("         Candidate product function added.")
                    else:
                        if verbose: print("Overall MAE declines from " + str(FORMAT%current_metrics[0]["MAE"]) \
                                          + " to " + str(FORMAT%candidate_metrics[0]["MAE"]) + ".")
                        channel_function.append(product_function)
                else:
                    channel_function.append(product_function)
            else:
                # Stop building the channel equation.
                if verbose:
                    print("Insignificant product function response.")
                    print()
                    print("############################################################")
                    print("Channel " + str(channel_id+1) + " function completed.")
                    print("############################################################")
                break
            if verbose: print()

        # Print the completed equation for the current output channel.
        if verbose: print("System equation")
        if verbose: print("============================================================")
        # Print the function template for the current output channel.
        y_str = "y" + str(channel_id+1) + "[k] = "
        for idf, product_function in enumerate(channel_function):
            y_str = y_str + product_function["template_string"]
            if idf < len(channel_function) - 1:
                y_str = y_str + " + "
        print(y_str)
        y_str = "y" + str(channel_id+1) + "[k] = "
        for idf, product_function in enumerate(channel_function):
            if product_function["estimate_string"] != None:
                y_str = y_str + product_function["estimate_string"]
                if idf < len(channel_function) - 1:
                    y_str = y_str + " + "
        print(y_str)
        print()

        model_function.append(channel_function)

    return model_function, new_mask

# Future work: Use better fit metric than weighted MAE.
# https://autarkaw.org/2008/07/05/finding-the-optimum-polynomial-order-to-use-for-regression/
```

Listing A.5: `evaluate_function()` evaluates performance of equation estimates.

```
# Evaluate the function generated by model analysis.

import numpy as np

def evaluate_function(model_function, input_data, output_data):

    metrics = []

    data_length = input_data.shape[0]
    output_channels = output_data.shape[1]

    #for channel in range(0, len(model_function)):
    for channel_id, channel_function in enumerate(model_function):

        channel_y = np.zeros([data_length, output_channels])
        max_delay_overall = 0

        for _, product_function in enumerate(channel_function):

            arg_list = product_function["arg_list"]
            arg_count = len(arg_list)
            arg_shift = product_function["shift"]
            if arg_count > 0:

                max_delay = max([arg["delay"] for arg in arg_list])
```

```
                if max_delay > max_delay_overall:
                    max_delay_overall = max_delay

                # Arrange input data to be fed into product function.
                x = np.zeros([arg_count, data_length-max_delay])
                for i, arg in enumerate(arg_list):
                    input_channel = arg["input_channel"]
                    delay = arg["delay"]
                    if delay > 0:
                        x[i] = input_data[(max_delay-delay):-delay, input_channel]-arg_shift[i]
                    else:
                        x[i] = input_data[(max_delay-delay):, input_channel]-arg_shift[i]

                # Feed input data to product function.
                params = product_function["parameters"]
                y_est = product_function["function"]["fcn"](x, *params)
                channel_y[max_delay:, channel_id] = channel_y[max_delay:, channel_id] + y_est
            else:
                # No arguments = bias term.
                params = product_function["parameters"]
                bias = product_function["function"]["fcn"](_, *params)
                channel_y[:, channel_id] = channel_y[:, channel_id] + bias

        # Calculate the error.
        y_err = output_data[max_delay_overall:, channel_id] - channel_y[max_delay_overall:, channel_id]
        y_err = y_err[max_delay_overall+1:]
        mae = np.mean(abs(y_err))
        rmse = np.sqrt(sum(pow(y_err, 2))/len(y_err))
        ymax = np.max(output_data[max_delay_overall:, channel_id])
        ymin = np.min(output_data[max_delay_overall:, channel_id])

        channel_metrics = {
            "MAE": mae,
            "RMSE": rmse,
            "MAX": ymax,
            "MIN": ymin,
        }
        metrics.append(channel_metrics)

    return metrics

# Future work: examine MAE vs RMSE
# https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d
```

Listing A.6: The `tune_model()` function which performs genetic algorithm optimization.

```
# Tune model parameters using a genetic algorithm.

import random
import copy
import time
import os
import numpy as np
import pyprind
import matplotlib.pyplot as plt

from lib.evaluate_function import evaluate_function

def tune_model(tuning_parameters, model_function, input_data, output_data):

    population_size = tuning_parameters["GA_population"]
    generation_count = tuning_parameters["GA_generations"]
    visual = tuning_parameters["visual"]
    save_visual = tuning_parameters["save_visual"]

    if save_visual == True:
        # Setup the most recent analysis directory to store GA tuning metrics.
        if not os.path.exists('./output'):
            os.mkdir('./output')
        analysis_dir_count = 1
        while os.path.exists('./output/analysis_{}'.format(analysis_dir_count)):
            analysis_dir_count = analysis_dir_count + 1
        analysis_dir_count = analysis_dir_count - 1
        if not os.path.exists('./output/analysis_{}'.format(analysis_dir_count)):
            os.mkdir('./output/analysis_{}'.format(analysis_dir_count))

    # Tune each channel individually.
    model_function_tuned = copy.deepcopy(model_function)
    for channel_id, channel_function in enumerate(model_function_tuned):

        # Get population details.
        parameter_count = 0
        upper_bounds = []
        lower_bounds = []
        parameters = []
        for fcn_id, product_function in enumerate(channel_function):
```

```
            parameter_count = parameter_count + len(product_function["parameters"])
            upper_bounds.extend(product_function["function"]["upper"])
            lower_bounds.extend(product_function["function"]["lower"])
            parameters.extend(product_function["parameters"])

        # Create the initial population of parameters.
        population = np.random.rand(population_size, parameter_count)
        # Member of initial estimate.
        for member_id in range(0, 1):
            population[member_id, :] = parameters
        # Members near initial estimate.
        for member_id in range(1, population_size):
            lower_bounds_dist = np.array(parameters) - (np.array(parameters) - np.array(lower_bounds))/3
            upper_bounds_dist = np.array(parameters) + (np.array(upper_bounds) - np.array(parameters))/3
            # Triangular distribution chosen because it is bounded.
            population[member_id, :] = np.random.triangular(lower_bounds_dist,parameters,upper_bounds_dist)

        # Execute genetic algorithm.
        print("Tuning channel " + str(channel_id+1) + "...")
        top_heuristic = np.zeros(generation_count)
        progress_bar = pyprind.ProgBar(generation_count, monitor=True)
        for generation_id in range(0, generation_count):
            # Evaluate generation.
            heuristic = np.zeros(population_size)
            for member_id in range(0, population_size):
                # Substitute the product function parameters
                parameter_index = 0
                for fcn_id, product_function in enumerate(channel_function):
                    product_function["parameters"] = list(
                            population[member_id, parameter_index:parameter_index+len(product_function["
    parameters"])])
                    parameter_index = parameter_index + len(product_function["parameters"])
                # Evaluate the new channel function
                metrics = evaluate_function([channel_function],
                                            input_data,
                                            output_data[:, channel_id:channel_id+1])
                heuristic[member_id]  = metrics[0]["MAE"]

            # Perform crossover of best members, clone best member.
            # Rank from smallest to largest MAE.
            member_rank = np.argsort(heuristic)
            upper_rank = member_rank[0:int(len(member_rank)/2)]
            population[0, :] = population[member_rank[0]]
            top_heuristic[generation_id] = heuristic[member_rank[0]]
            for member_id in range(1, population_size):
                parents = random.sample(list(upper_rank), k=2)
                crossover_point = random.randint(0,parameter_count)
                child = np.concatenate((population[parents[0], :crossover_point], population[parents[1],
    crossover_point:]))
                population[member_id, :] = child

            # Perform mutations of new members.
            for member_id in range(1, population_size):
                if np.random.rand() < 0.25:
                    mutation_mask = np.random.randint(2, size=parameter_count)
                    mutation_degree = 0.1*2*(np.random.rand(parameter_count)-0.5)
                    mutation = mutation_mask*mutation_degree
                    population[member_id, :] = population[member_id, :] + mutation
            progress_bar.update()
        time.sleep(0.5) # Allows progress bar to finish printing elapsed time.

        # Assign new parameters to product function.
        parameter_index = 0
        for fcn_id, product_function in enumerate(channel_function):
            product_function["parameters"] = list(
                    population[0, parameter_index:parameter_index+len(product_function["parameters"])])
            if len(product_function["parameters"]) > 0:
                parameter_index = parameter_index + len(product_function["parameters"])
                product_function["estimate_string"] = product_function["function"]["txt_fcn"](
                        product_function["arg_list"],
                        product_function["shift"],
                        *product_function["parameters"])
        print()

        # Plot GA tuning metrics.
        if save_visual == True or visual == True:
            plt.figure()
            plt.plot(top_heuristic)
            plt.title('Top MAE vs Generation')
            plt.xlabel('Generation')
            plt.ylabel('MAE')
            if save_visual == True: plt.savefig('./output/analysis_{}/ga_mae.pdf'.format(analysis_dir_count))
            if visual == True: plt.show()

    return model_function_tuned

# Future: In GA tuning, use a parameter confidence interval to limit the search space.
# http://kitchingroup.cheme.cmu.edu/blog/2013/02/12/Nonlinear-curve-fitting-with-parameter-confidence-intervals/
```

Listing A.7: `fitting_functions()` returns an object with the template product functions.

```
# Define the template product functions.

import numpy as np

# Polynomials.
#=============================================================================#
def fcn_poly1_1(x,a):
    return a*x[0]
def txt_poly_1(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_poly2_1(x,a,b):
    return a*pow(x[0],2) \
         + b*x[0]
def txt_poly2_1(argList,argShift,a,b):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                b,argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_squared_1(x,a):
    return a*pow(x[0],2)
def txt_squared_1(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]^2".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_poly3_1(x,a,b,c):
    return a*pow(x[0],3) \
         + b*pow(x[0],2) \
         + c*x[0]
def txt_poly3_1(argList,argShift,a,b,c):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]^3 + " \
                "{:.2e}*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],
                c,argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^3 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_cubed_1(x,a):
    return a*pow(x[0],3)
def txt_cubed_1(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]^3".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^3".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_poly4_1(x,a,b,c,d):
    return a*pow(x[0],4) \
         + b*pow(x[0],3) \
         + c*pow(x[0],2) \
         + d*x[0]
def txt_poly4_1(argList,argShift,a,b,c,d):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]^4 + " \
                "{:.2e}*x{:d}[k-{:d}]^3 + " \
                "{:.2e}*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],
                d,argList[0]["input_channel"]+1,argList[0]["delay"])
```

```
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^4 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^3 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                d,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_poly5_1(x,a,b,c,d,e):
    return a*pow(x[0],5) \
         + b*pow(x[0],4) \
         + c*pow(x[0],3) \
         + d*pow(x[0],2) \
         + e*x[0]
def txt_poly5_1(argList,argShift,a,b,c,d,e):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]^5 + " \
                "{:.2e}*x{:d}[k-{:d}]^4 + " \
                "{:.2e}*x{:d}[k-{:d}]^3 + " \
                "{:.2e}*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],
                d,argList[0]["input_channel"]+1,argList[0]["delay"],
                e,argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^5 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^4 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^3 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                d,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                e,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_poly22_2(x,a):
    return a*x[0]*x[1]
def txt_poly22_2(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1])

def fcn_poly33_2(x,a,b,c):
    return a*pow(x[0],1)*pow(x[1],1) \
         + b*pow(x[0],2)*pow(x[1],1) \
         + c*pow(x[0],1)*pow(x[1],2)
def txt_poly33_2(argList,argShift,a,b,c):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]^2*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]^2".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})^2".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1])

def fcn_poly44_2(x,a,b,c,d,e,f):
    return a*pow(x[0],1)*pow(x[1],1) \
         + b*pow(x[0],2)*pow(x[1],1) \
         + c*pow(x[0],1)*pow(x[1],2) \
         + d*pow(x[0],3)*pow(x[1],1) \
         + e*pow(x[0],2)*pow(x[1],2) \
         + f*pow(x[0],1)*pow(x[1],3)
def txt_poly44_2(argList,argShift,a,b,c,d,e,f):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}] + " \
```

```
                "{:.2e}*x{:d}[k-{:d}]^2*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]^3*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]^2*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]^3".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                d,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                e,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                f,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^3*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})^3".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                d,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                e,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                f,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1])

def fcn_poly55_2(x,a,b,c,d,e,f,g,h,i,j):
    return a*pow(x[0],1)*pow(x[1],1) \
         + b*pow(x[0],2)*pow(x[1],1) \
         + c*pow(x[0],1)*pow(x[1],2) \
         + d*pow(x[0],3)*pow(x[1],1) \
         + e*pow(x[0],2)*pow(x[1],2) \
         + f*pow(x[0],1)*pow(x[1],3) \
         + g*pow(x[0],4)*pow(x[1],1) \
         + h*pow(x[0],3)*pow(x[1],2) \
         + i*pow(x[0],2)*pow(x[1],3) \
         + j*pow(x[0],1)*pow(x[1],4)
def txt_poly55_2(argList,argShift,a,b,c,d,e,f,g,h,i,j):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]^2*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]^3*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]^2*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]^3 + " \
                "{:.2e}*x{:d}[k-{:d}]^4*x{:d}[k-{:d}] + " \
                "{:.2e}*x{:d}[k-{:d}]^3*x{:d}[k-{:d}]^2 + " \
                "{:.2e}*x{:d}[k-{:d}]^2*x{:d}[k-{:d}]^3 + " \
                "{:.2e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]^4".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                d,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                e,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                f,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                g,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                h,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                i,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                j,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^3*(x{:d}[k-{:d}]-{:.2e}) + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})^3 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^4*(x{:d}[k-{:d}]-{:.2e}) + " \
```

```
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^3*(x{:d}[k-{:d}]-{:.2e})^2 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})^2*(x{:d}[k-{:d}]-{:.2e})^3 + " \
                "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})^4".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                c,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                d,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                e,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                f,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                g,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                h,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                i,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                j,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1])

def fcn_linear_3(x,a):
    return a*x[0]*x[1]*x[2]
def txt_linear_3(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.3e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                argList[2]["input_channel"]+1,argList[2]["delay"])
    else:
        return "{:.3e}*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})*(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                argList[2]["input_channel"]+1,argList[2]["delay"],argShift[2])

def fcn_linear_4(x,a):
    return a*x[0]*x[1]*x[2]*x[3]
def txt_linear_4(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.3e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]*x{:d}[k-{:d}]*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                argList[2]["input_channel"]+1,argList[2]["delay"],
                argList[3]["input_channel"]+1,argList[3]["delay"])
    else:
        return "{:.3e}*(x{:d}[k-{:d}]-{:.2e})*" \
               "(x{:d}[k-{:d}]-{:.2e})*" \
               "(x{:d}[k-{:d}]-{:.2e})*" \
               "(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                argList[2]["input_channel"]+1,argList[2]["delay"],argShift[2],
                argList[3]["input_channel"]+1,argList[3]["delay"],argShift[3])

def fcn_linear_5(x,a):
    return a*x[0]*x[1]*x[2]*x[3]*x[4]
def txt_linear_5(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.3e}*x{:d}[k-{:d}]*x{:d}[k-{:d}]*x{:d}[k-{:d}]*x{:d}[k-{:d}]*x{:d}[k-{:d}]".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"],
                argList[2]["input_channel"]+1,argList[2]["delay"],
                argList[3]["input_channel"]+1,argList[3]["delay"],
                argList[4]["input_channel"]+1,argList[4]["delay"])
    else:
        return "{:.3e}*(x{:d}[k-{:d}]-{:.2e})*" \
               "(x{:d}[k-{:d}]-{:.2e})*" \
               "(x{:d}[k-{:d}]-{:.2e})*" \
               "(x{:d}[k-{:d}]-{:.2e})".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                argList[2]["input_channel"]+1,argList[2]["delay"],argShift[2],
                argList[3]["input_channel"]+1,argList[3]["delay"],argShift[3],
                argList[4]["input_channel"]+1,argList[4]["delay"],argShift[4])
#==============================================================================#

# Exponentials.
#==============================================================================#
def fcn_exp_1(x,a,b):
    return a*(np.exp(b*x[0])-1)
def txt_exp_1(argList,argShift,a,b):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*(e^({:.2e}*x{:d}[k-{:d}])-1)".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"])
```

```
    else:
        return "{:.2e}*(e^({:.2e}*(x{:d}[k-{:d}]-{:.2e}))-1)".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_exp_lin12_2(x,a):
    return a*x[0]*(np.exp(x[1])-1)
def txt_exp_lin12_2(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]*(e^(x{:d}[k-{:d}])-1)".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],
                argList[1]["input_channel"]+1,argList[1]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(e^(x{:d}[k-{:d}]-{:.2e})-1)".format(
                a,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],
                argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1])

def fcn_exp_lin21_2(x,a):
    return a*x[1]*(np.exp(x[0])-1)
def txt_exp_lin21_2(argList,argShift,a):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*x{:d}[k-{:d}]*(e^(x{:d}[k-{:d}])-1)".format(
                a,argList[1]["input_channel"]+1,argList[1]["delay"],
                argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*(x{:d}[k-{:d}]-{:.2e})*(e^(x{:d}[k-{:d}]-{:.2e})-1)".format(
                a,argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],
                argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])
#============================================================================#

# Sinusoidal functions.
#============================================================================#
# This function is hard to fit, so forms with less parameters are included.
def fcn_sin_1(x,a,b,c):
    return a*np.sin(b*x[0]+c)-a*np.sin(c)
def txt_sin_1(argList,argShift,a,b,c):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*sin({:.2e}*x{:d}[k-{:d}]+{:.2e})-{:.2e}".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"],c,(a*np.sin(c)))
    else:
        return "{:.2e}*sin({:.2e}*(x{:d}[k-{:d}]-{:.2e})+{:.2e})-{:.2e}".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],c,(a*np.sin(c)))

def fcn_tanh_1(x,a,b):
    return a*np.tanh(b*x[0])
def txt_tanh_1(argList,argShift,a,b):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*tanh({:.2e}*x{:d}[k-{:d}])".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"])
    else:
        return "{:.2e}*tanh({:.2e}*(x{:d}[k-{:d}]-{:.2e}))".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0])

def fcn_tanhx_1(x,a,b,c,d):
    return a*np.tanh(b*(x[0]+c)) + d*(x[0]+c) - (a*np.tanh(b*c)+d*c)
def txt_tanhx_1(argList,argShift,a,b,c,d):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*tanh({:.2e}*(x{:d}[k-{:d}]+{:.2e})) + " \
                "{:.2e}*(x{:d}[k-{:d}]+{:.2e}) - " \
                "{:.2e}".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"],c,
                d,argList[0]["input_channel"]+1,argList[0]["delay"],c,
                (a*np.tanh(b*c)+d*c))
    else:
        return "{:.2e}*tanh({:.2e}*((x{:d}[k-{:d}]-{:.2e})+{:.2e})) + " \
                "{:.2e}*((x{:d}[k-{:d}]-{:.2e})+{:.2e}) - " \
                "{:.2e}".format(
                a,b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],c,
                d,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],c,
                (a*np.tanh(b*c)+d*c))

def fcn_tanh12_2(x,a,b,c,d,e):
    return a*np.tanh(b*x[0]-c)*np.tanh(d*x[1]-e) - a*np.tanh(-c)*np.tanh(-e)
def txt_tanh12_2(argList,argShift,a,b,c,d,e):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*tanh({:.2e}*x{:d}[k-{:d}]-{:.2e})*" \
                "tanh({:.2e}*x{:d}[k-{:d}]-{:.2e}) - " \
                "{:.2e}".format(
                        a,b,argList[0]["input_channel"]+1,argList[0]["delay"],c,
                        d,argList[1]["input_channel"]+1,argList[1]["delay"],e,
                        (a*np.tanh(-c)*np.tanh(-e)))
    else:
        return "{:.2e}*tanh({:.2e}*(x{:d}[k-{:d}]-{:.2e})-{:.2e})*" \
                "tanh({:.2e}*(x{:d}[k-{:d}]-{:.2e})-{:.2e}) - " \
                "{:.2e}".format(
                        a,b,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[0],c,
                        d,argList[1]["input_channel"]+1,argList[1]["delay"],argShift[1],e,
                        (a*np.tanh(-c)*np.tanh(-e)))

```

```
def fcn_tanh21_2(x,a,b,c,d,e):
    return a*np.tanh(b*x[1]-c)*np.tanh(d*x[0]-e) - a*np.tanh(-c)*np.tanh(-e)
def txt_tanh21_2(argList,argShift,a,b,c,d,e):
    if all(shift == 0 for shift in argShift):
        return "{:.2e}*tanh({:.2e}*x{:d}[k-{:d}]-{:.2e})*" \
                "tanh({:.2e}*x{:d}[k-{:d}]-{:.2e}) - " \
                "{:.2e}".format(
                        a,b,argList[1]["input_channel"]+1,argList[1]["delay"],c,
                        d,argList[0]["input_channel"]+1,argList[0]["delay"],e,
                        (a*np.tanh(-c)*np.tanh(-e)))
    else:
        return "{:.2e}*tanh({:.2e}*(x{:d}[k-{:d}]-{:.2e})-{:.2e})*" \
                "tanh({:.2e}*(x{:d}[k-{:d}]-{:.2e})-{:.2e}) - " \
                "{:.2e}".format(
                        a,b,argList[1]["input_channel"]+1,argList[1]["delay"],argShift[0],c,
                        d,argList[0]["input_channel"]+1,argList[0]["delay"],argShift[1],e,
                        (a*np.tanh(-c)*np.tanh(-e)))
#============================================================================#

# Return a list containing template functions for argCount number of arguments.
def fitting_functions():

    #============================================================================#
    # Functions of 1 variable.
    #============================================================================#
    dct_poly1_1 = {
        "txt": "a*x1",
        "txt_fcn": txt_poly_1,
        "fcn": fcn_poly1_1,
        "upper": [10],
        "lower": [-10],
        "weight": 1.0
    }
    dct_poly2_1 = {
        "txt": "a*x1^2 + b*x1",
        "txt_fcn": txt_poly2_1,
        "fcn": fcn_poly2_1,
        "upper": [10, 10],
        "lower": [-10, -10],
        "weight": 0.55
    }
    dct_squared_1 = {
        "txt": "a*x1^2",
        "txt_fcn": txt_squared_1,
        "fcn": fcn_squared_1,
        "upper": [10],
        "lower": [-10],
        "weight": 1.0
    }
    dct_poly3_1 = {
        "txt": "a*x1^3 + b*x1^2 + c*x1",
        "txt_fcn": txt_poly3_1,
        "fcn": fcn_poly3_1,
        "upper": [10, 10, 10],
        "lower": [-10, -10, -10],
        "weight": 0.20
    }
    dct_cubed_1 = {
        "txt": "a*x1^3",
        "txt_fcn": txt_cubed_1,
        "fcn": fcn_cubed_1,
        "upper": [10],
        "lower": [-10],
        "weight": 1.0
    }
    dct_poly4_1 = {
        "txt": "a*x1^4 + b*x1^3 + b*x1^2 + c*x1",
        "txt_fcn": txt_poly4_1,
        "fcn": fcn_poly4_1,
        "upper": [100, 100, 10, 10],
        "lower": [-100, -100, -10, -10],
        "weight": 0.15
    }
    dct_poly5_1 = {
        "txt": "a*x1^5 + b*x1^4 + b*x1^3 + b*x1^2 + c*x1",
        "txt_fcn": txt_poly5_1,
        "fcn": fcn_poly5_1,
        "upper": [100, 100, 10, 10, 10],
        "lower": [-100, -100, -10, -10, -10],
        "weight": 0.10
    }
    #============================================================================#
    dct_exp_1 = {
        "txt": "a*(e^(b*x1)-1)",
        "txt_fcn": txt_exp_1,
        "fcn": fcn_exp_1,
        "upper": [10, 5],
        "lower": [-10, -5],
```

```
        "weight": 0.25
    }
    #========================================================================#
    dct_sin_1 = {
        "txt": "a*sin(b*x1+c) - a*sin(c)",
        "txt_fcn": txt_sin_1,
        "fcn": fcn_sin_1,
        "upper": [10, 10*3.14159, 3.14159],
        "lower": [0, 0, -3.14159],
        "weight": 0.25
    }
    dct_tanh_1 = {
        "txt": "a*tanh(b*x1)",
        "txt_fcn": txt_tanh_1,
        "fcn": fcn_tanh_1,
        "upper": [1000, 10],
        "lower": [-1000, 0.001],
        "weight": 0.25
    }
    dct_tanhx_1 = {
        "txt": "a*tanh(b*(x1+c)) + d*(x1+c) - (a*tanh(b*c)+d*c)",
        "txt_fcn": txt_tanhx_1,
        "fcn": fcn_tanhx_1,
        "upper": [1000, 100, 100, 100],
        "lower": [-1000, 0.001, -100, -100],
        "weight": 0.25
    }
    #========================================================================#

    #========================================================================#
    # Functions of 2 variables.
    #========================================================================#
    dct_poly22_2 = {
        "txt": "a*x1*x2",
        "txt_fcn": txt_poly22_2,
        "fcn": fcn_poly22_2,
        "upper": [10],
        "lower": [-10],
        "weight": 1.0
    }
    dct_poly33_2 = {
        "txt": "a*x1*x2 + b*x1^2*x2 + c*x1*x2^2",
        "txt_fcn": txt_poly33_2,
        "fcn": fcn_poly33_2,
        "upper": [10, 10, 10],
        "lower": [-10, -10, -10],
        "weight": 0.5
    }
    dct_poly44_2 = {
        "txt": "a*x1*x2 + b*x1^2*x2 + c*x1*x2^2 + d*x1^3*x2 + e*x1^2*x2^2 + f*x1*x2^3",
        "txt_fcn": txt_poly44_2,
        "fcn": fcn_poly44_2,
        "upper": [10, 10, 10, 10, 10, 10],
        "lower": [-10, -10, -10, -10, -10, -10],
        "weight": 0.4
    }
    dct_poly55_2 = {
        "txt": "a*x1*x2 + b*x1^2*x2 + c*x1*x2^2 + d*x1^3*x2 + e*x1^2*x2^2 + " \
               "f*x1*x2^3 + g*x1^4*x2 + h*x1^3*x2^2 + i*x1^2*x2^3 + j*x1*x2^4",
        "txt_fcn": txt_poly55_2,
        "fcn": fcn_poly55_2,
        "upper": [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
        "lower": [-10, -10, -10, -10, -10, -10, -10, -10, -10, -10],
        "weight": 0.3
    }
    #========================================================================#
    dct_exp_lin12_2 = {
        "txt": "a*x1*(e^(x2)-1)",
        "txt_fcn": txt_exp_lin12_2,
        "fcn": fcn_exp_lin12_2,
        "upper": [10],
        "lower": [-10],
        "weight": 0.5
    }
    dct_exp_lin21_2 = {
        "txt": "a*x2*(e^(x1)-1)",
        "txt_fcn": txt_exp_lin21_2,
        "fcn": fcn_exp_lin21_2,
        "upper": [10],
        "lower": [-10],
        "weight": 0.5
    }
    #========================================================================#
    dct_tanh12_2 = {
        "txt": "a*tanh(b*x1-c)*tanh(d*x2-e) - a*tanh(-c)*tanh(-e)",
        "txt_fcn": txt_tanh12_2,
        "fcn": fcn_tanh12_2,
        "upper": [1000, 1, 100, 1, 100],
```

```
        "lower": [-1000, 0.001, -100, 0.001, -100],
        "weight": 0.5
    }
    dct_tanh21_2 = {
        "txt": "a*tanh(b*x2-c)*tanh(d*x1-e) - a*tanh(-c)*tanh(-e)",
        "txt_fcn": txt_tanh21_2,
        "fcn": fcn_tanh21_2,
        "upper": [1000, 1, 100, 1, 100],
        "lower": [-1000, 0.001, -100, 0.001, -100],
        "weight": 0.5
    }
    #============================================================================#

    #============================================================================#
    # Functions of 3+ variables.
    #============================================================================#
    dct_linear_3 = {
        "txt": "a*x1*x2*x3",
        "txt_fcn": txt_linear_3,
        "fcn": fcn_linear_3,
        "upper": [10],
        "lower": [-10],
        "weight": 1.0
    }

    dct_linear_4 = {
        "txt": "a*x1*x2*x3*x4",
        "txt_fcn": txt_linear_4,
        "fcn": fcn_linear_4,
        "upper": [10],
        "lower": [-10],
        "weight": 1.0
    }

    dct_linear_5 = {
        "txt": "a*x1*x2*x3*x4*x5",
        "txt_fcn": txt_linear_5,
        "fcn": fcn_linear_5,
        "upper": [10],
        "lower": [-10],
        "weight": 1.0
    }
    #============================================================================#

    input_1_list = [dct_poly1_1,
                    dct_poly2_1,
                    dct_poly3_1,
                    dct_poly4_1,
                    dct_poly5_1,
                    dct_squared_1,
                    dct_cubed_1,
                    dct_exp_1,
                    dct_sin_1,
                    dct_tanh_1,
                    dct_tanhx_1,
                    ]

    input_2_list = [dct_poly22_2,
                    dct_poly33_2,
                    dct_poly44_2,
                    dct_poly55_2,
                    dct_exp_lin12_2,
                    dct_exp_lin21_2,
                    dct_tanh12_2,
                    dct_tanh21_2,
                    ]

    input_3_list = [dct_linear_3]

    input_4_list = [dct_linear_4]

    input_5_list = [dct_linear_5]

    functionDictionary = {
        1: input_1_list,
        2: input_2_list,
        3: input_3_list,
        4: input_4_list,
        5: input_5_list
    }

    return functionDictionary
```

Bibliography

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366.

[2] K. S. Narendra and K. Parthasarathy. "Identification and control of dynamical systems using neural networks". In: *IEEE Transactions on Neural Networks* 1.1 (Mar. 1990), pp. 4–27.

[3] A. U. Levin and K. S. Narendra. "Control of nonlinear dynamical systems using neural networks. II. Observability, identification, and control". In: *IEEE Transactions on Neural Networks* 7.1 (Jan. 1996), pp. 30–42.

[4] J. D. Donne and U. Ozguner. "A comparative study of neural vs. conventional methods for modeling and prediction". In: *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*. IEEE, Aug. 1992, pp. 548–553. DOI: 10.1109/ISIC.1992.225047.

[5] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. "A survey of methods for explaining black box models". In: *ACM computing surveys (CSUR)* 51.5 (2018), p. 93.

[6] Henrik Jacobsson. "Rule extraction from recurrent neural networks: A taxonomy and review". In: *Neural Computation* 17.6 (2005), pp. 1223–1263.

[7] Kazumi Saito and Ryohei Nakano. "Extracting regression rules from neural networks". In: *Neural Networks* 15.10 (2002), pp. 1279–1288.

[8] Lennart Ljung. "System identification". In: *Wiley Encyclopedia of Electrical and Electronics Engineering* (2001).

[9] Paul Duffy. "Machine Learning for Sensitivity Analysis of Probabilistic Environmental Models". In: 2015.

[10] Peter Lancaster and Kestutis Salkauskas. *Curve and surface fitting: an introduction*. Academic press, 1986.

[11] Sheng Chen and Steve A Billings. "Representations of non-linear systems: the NARMAX model". In: *International Journal of Control* 49.3 (1989), pp. 1013–1032.

[12] Sheng Chen, SA Billings, and PM Grant. "Non-linear system identification using neural networks". In: *International journal of control* 51.6 (1990), pp. 1191–1214.

[13] José Maria P Menezes Jr and Guilherme A Barreto. "Long-term time series prediction with the NARX network: An empirical evaluation". In: *Neurocomputing* 71.16-18 (2008), pp. 3335–3343.

[14] Robert Andrews, Joachim Diederich, and Alan B Tickle. "Survey and critique of techniques for extracting rules from trained artificial neural networks". In: *Knowledge-Based Systems* 8.6 (1995), pp. 373–389.

[15] Alan B Tickle, Mostefa Golea, Ross Hayward, and Joachim Diederich. "The truth is in there: current issues in extracting rules from trained feedforward artificial neural networks". In: *Proceedings of International Conference on Neural Networks (ICNN'97)*. Vol. 4. IEEE, 1997, pp. 2530–2534.

[16] Alan Tickle, Robert Andrews, Mostefa Golea, and Joachim Diederich. "The Truth is in There: Directions and Challenges in Extracting Rules From Trained Artificial Neural Networks". In: *IEEE Transactions on Neural Networks* 9 (Feb. 2000).

[17] A. B. Tickle, R. Andrews, M. Golea, and J. Diederich. "The truth will come to light: directions and challenges in extracting the knowledge embedded within trained artificial neural networks". In: *IEEE Transactions on Neural Networks* 9.6 (Nov. 1998), pp. 1057–1068. ISSN: 1045-9227. DOI: 10.1109/72.728352.

[18] Rudy Setiono, Wee Kheng Leow, and Jacek M Zurada. "Extraction of rules from artificial neural networks for nonlinear regression". In: *IEEE transactions on neural networks* 13.3 (2002), pp. 564–577.

[19] RUDY Setiono. "Techniques for extracting classification and regression rules from artificial neural networks". In: *Computational intelligence: The experts speak* (2003), pp. 99–114.

[20] Dennis W Ruck, Steven K Rogers, and Matthew Kabrisky. "Feature selection using a multilayer perceptron". In: *Journal of Neural Network Computing* 2.2 (1990), pp. 40–48.

[21] R. H. Kewley, M. J. Embrechts, and C. Breneman. "Data strip mining for the virtual design of pharmaceuticals with neural networks". In: *IEEE Transactions on Neural Networks* 11.3 (May 2000), pp. 668–679. ISSN: 1045-9227. DOI: 10.1109/72.846738.

[22] Mark J Embrechts, Fabio A Arciniegas, Muhsin Ozdemir, and Robert H Kewley. "Data mining for molecules with 2-D neural network sensitivity analysis". In: *International Journal of smart engineering system design* 5.4 (2003), pp. 225–239.

[23] P. Cortez and M. J. Embrechts. "Opening black box Data Mining models using Sensitivity Analysis". In: *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. Apr. 2011, pp. 341–348. DOI: 10.1109/CIDM.2011.5949423.

[24] Paulo Cortez and Mark J Embrechts. "Using sensitivity analysis and visualization techniques to open black box data mining models". In: *Information Sciences* 225 (Mar. 2013), pp. 1–17.

[25] Kazumi Saito and Ryohei Nakano. “Medical diagnostic expert system based on PDP model”. In: *IEEE 1988 International Conference on Neural Networks*. Vol. 1. July 1988, pp. 255–262. DOI: 10.1109/ICNN.1988.23855.

[26] A. Vahed and C. W. Omlin. “Rule extraction from recurrent neural networks using a symbolic machine learning algorithm”. In: *ICONIP’99. ANZIIS’99 ANNES’99 ACNN’99. 6th International Conference on Neural Information Processing. Proceedings (Cat. No.99EX378)*. Vol. 2. IEEE, Nov. 1999, pp. 712–717. DOI: 10.1109/ICONIP.1999.845683.

[27] A Vahed and Christian W Omlin. “A machine learning method for extracting symbolic knowledge from recurrent neural networks”. In: *Neural Computation* 16.1 (Jan. 2004), pp. 59–71.

[28] Ilya M Sobol. “Sensitivity estimates for nonlinear mathematical models”. In: *Mathematical modelling and computational experiments* 1.4 (1993), pp. 407–414.

[29] Toshimitsu Homma and Andrea Saltelli. “Importance measures in global sensitivity analysis of nonlinear models”. In: *Reliability Engineering & System Safety* 52.1 (1996), pp. 1–17.

[30] Ilya M Sobol. “Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates”. In: *Mathematics and computers in simulation* 55.1-3 (2001), pp. 271–280.

[31] GEB Archer, Andrea Saltelli, and IM Sobol. “Sensitivity measures, ANOVA-like techniques and the use of bootstrap”. In: *Journal of Statistical Computation and Simulation* 58.2 (1997), pp. 99–120.

[32] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[33] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. “An empirical evaluation of generic convolutional and recurrent networks for sequence modeling”. In: *arXiv preprint arXiv:1803.01271* (2018).

[34] John Miller and Moritz Hardt. “When recurrent models don’t need to be recurrent”. In: *arXiv preprint arXiv:1805.10369* 4 (2018).

[35] Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. “Wavenet: A generative model for raw audio”. In: *CoRR abs/1609.03499* (2016).

[36] M. Lopez and W. Yu. “Nonlinear system modeling using convolutional neural networks”. In: *2017 14th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*. IEEE, Oct. 2017, pp. 1–5. DOI: 10.1109/ICEEE.2017.8108894.

[37] T. Wigren and J. Schoukens. “Three free data sets for development and benchmarking in nonlinear system identification”. In: *2013 European Control Conference (ECC)*. July 2013, pp. 2933–2938.

[38] Lennart Ljung, Qinghua Zhang, Peter Lindskog, and Anatoli Juditski. “Estimation of grey box and black box models for non-linear circuit data”. In: *IFAC Proceedings Volumes* 37.13 (2004). 6th IFAC Symposium on Nonlinear Control Systems 2004 (NOLCOS 2004), Stuttgart, Germany, 1-3 September, 2004, pp. 399–404. ISSN: 1474-6670.

[39] Marcelo Espinoza, Kristiaan Pelckmans, Luc Hoegaerts, Johan A.K. Suykens, and Bart De Moor. “A comparative study of ls-svm’s applied to the silver box identification problem”. In: *IFAC Proceedings Volumes* 37.13 (2004). 6th IFAC Symposium on Nonlinear Control Systems 2004 (NOLCOS 2004), Stuttgart, Germany, 1-3 September, 2004, pp. 369–374. ISSN: 1474-6670.

[40] J. Paduart, G. Horvath, and J. Schoukens. “Fast identification of systems with nonlinear feedback”. In: *IFAC Proceedings Volumes* 37.13 (2004). 6th IFAC Symposium on Nonlinear Control Systems 2004 (NOLCOS 2004), Stuttgart, Germany, 1-3 September, 2004, pp. 381–385. ISSN: 1474-6670.

[41] Håkan Hjalmarsson and Johan Schoukens. “On Direct Identification of Physical Parameters in Non-Linear Models”. In: *IFAC Proceedings Volumes* 37.13 (2004). 6th IFAC Symposium on Nonlinear Control Systems 2004 (NOLCOS 2004), Stuttgart, Germany, 1-3 September, 2004, pp. 375–380. ISSN: 1474-6670.

[42] Vincent Verdult. “Identification of Local Linear State-Space Models: The Silver-Box Case Study”. In: *IFAC Proceedings Volumes* 37.13 (2004). 6th IFAC Symposium on Nonlinear Control Systems 2004 (NOLCOS 2004), Stuttgart, Germany, 1-3 September, 2004, pp. 393–398. ISSN: 1474-6670.

[43] László Sragner, Johan Schoukens, and Gábor Horváth. “Modelling of a slightly nonlinear system: a neural network approach”. In: *IFAC Proceedings Volumes* 37.13 (2004). 6th IFAC Symposium on Nonlinear Control Systems 2004 (NOLCOS 2004), Stuttgart, Germany, 1-3 September, 2004, pp. 387–392. ISSN: 1474-6670.

[44] M. Espinoza, J. A. K. Suykens, and Bart De Moor. “Kernel based partially linear models and nonlinear identification”. In: *IEEE Transactions on Automatic Control* 50.10 (Oct. 2005), pp. 1602–1606. ISSN: 0018-9286.

[45] Marcelo Espinoza Tapia. “Structured Kernel Based Modeling and its Application to Electric Load Forecasting”. In: (2006).

[46] Johan Paduart, Lieve Lauwers, Jan Swevers, Kris Smolders, Johan Schoukens, and Rik Pintelon. “Identification of nonlinear systems using Polynomial Nonlinear State Space models”. In: *Automatica* 46.4 (2010), pp. 647–656. ISSN: 0005-1098.

[47] E. Pepona, S. Paoletti, A. Garulli, and P. Date. “Identification of Piecewise Affine LFR Models of Interconnected Systems”. In: *IEEE Transactions on Control Systems Technology* 19.1 (Jan. 2011), pp. 148–155. ISSN: 1063-6536.

[48] Anna Marconato, Jonas Sjöberg, Johan Suykens, and Johan Schoukens. “Identification of the Silverbox Benchmark Using Nonlinear State-Space Models”. In: *IFAC Proceedings Volumes* 45.16 (2012). 16th IFAC Symposium on System Identification, pp. 632–637. ISSN: 1474-6670.

[49] A. Marconato, M. Schoukens, Y. Rolain, and J. Schoukens. “Study of the effective number of parameters in nonlinear identification benchmarks”. In: *52nd IEEE Conference on Decision and Control*. Dec. 2013, pp. 4308–4313.

[50] Anne Van Mulders, Johan Schoukens, and Laurent Vanbeylen. “Identification of systems with localised nonlinearity: From state-space to block-structured models”. In: *Automatica* 49.5 (2013), pp. 1392–1396. ISSN: 0005-1098.

[51] R. Frigola and C. E. Rasmussen. “Integrated pre-processing for Bayesian nonlinear system identification with Gaussian processes”. In: *52nd IEEE Conference on Decision and Control*. Dec. 2013, pp. 5371–5376.

[52] R. Castro, S. Mehrkanoon, A. Marconato, J. Schoukens, and J. A. K. Suykens. “SVD truncation schemes for fixed-size kernel models”. In: *2014 International Joint Conference on Neural Networks (IJCNN)*. July 2014, pp. 3922–3929.

[53] F. Sabahi and M. R. Akbarzadeh-T. “Extended Fuzzy Logic: Sets and Systems”. In: *IEEE Transactions on Fuzzy Systems* 24.3 (June 2016), pp. 530–543. ISSN: 1063-6706.

[54] A. Carini and G. L. Sicuranza. “Recursive functional link polynomial filters: An introduction”. In: *2016 24th European Signal Processing Conference (EUSIPCO)*. Aug. 2016, pp. 2335–2339.

[55] César Lincoln C Mattos, Guilherme A Barreto, and Gonzalo Acuña. “Randomized Neural Networks for Recursive System Identification in the Presence of Outliers: A Performance Comparison”. In: *International Work-Conference on Artificial Neural Networks*. Springer. 2017, pp. 603–615.

[56] José Daniel A. Santos and Guilherme A. Barreto. “Novel sparse LSSVR models in primal weight space for robust system identification with outliers”. In: *Journal of Process Control* 67 (2018). Big Data: Data Science for Process Control and Operations, pp. 129–140. ISSN: 0959-1524.

[57] J. Schoukens, J.G. Nemeth, P. Crama, Y. Rolain, and R. Pintelon. “Fast approximate identification of nonlinear systems”. In: *Automatica* 39.7 (2003), pp. 1267–1274. ISSN: 0005-1098.

[58] Wenhao Yu, Jie Tan, C Karen Liu, and Greg Turk. “Preparing for the unknown: Learning a universal policy with online system identification”. In: *arXiv preprint arXiv:1702.02453* (2017).

[59] Wassily Leontief. “A note on the interrelation of subsets of independent variables of a continuous function with continuous first derivatives”. In: *Bulletin of the American mathematical Society* 53.4 (Apr. 1947), pp. 343–350.

[60] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed January 3, 2019]. 2001–.

[61] Kim-Fung Man, Kit-Sang Tang, and Sam Kwong. "Genetic algorithms: concepts and applications [in engineering design]". In: *IEEE transactions on Industrial Electronics* 43.5 (1996), pp. 519–534.

[62] Catherine Forbes, Merran Evans, Nicholas Hastings, and Brian Peacock. *Statistical distributions*. John Wiley & Sons, 2011.

[63] MathWorks. *Electric Vehicle Configured for HIL*. `https://www.mathworks.com/help/physmod/sps/examples/electric-vehicle-configured-for-hil.html`. Accessed: 2018-01-26.

[64] Jon C Helton and Freddie Joe Davis. "Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems". In: *Reliability Engineering & System Safety* 81.1 (2003), pp. 23–69.