MACHINE LEARNING BASED MODELS FOR SIGNAL INTEGRITY ANALYSIS

BY

THONG NGUYEN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Jose Schutt-Aine, Chair
Professor Jennifer Bernhard
Professor Andreas Cangellaris
Professor Erhan Kudeki

ABSTRACT

With a short product cycle as we see today, fast and accurate modeling methods are becoming crucial for the development of new generations of electronic devices. Furthermore, increased complexity in circuitry and integration compounds design iteration and the associated, high-dimensional sensitivity analysis and performance optimization studies. Expedient design iteration, performance optimization, and design verification of state-of-the-art electronic devices and systems are hindered by the ever-increasing functionality integration. This thesis is meant to contribute a small part to the enormous amount of effort of the electronic design automation community in the quest for computationally efficient methods capable of handling the high-dimensional design space of such devices and systems using machine learning methods. This thesis focuses on applications related to high-speed channel and microwave circuit designs. It first explores the recurrent neural network for time-domain waveform prediction. Two different recurrent neural network architectures are distinguished, their advantages and disadvantages are pointed out. Different examples are used to demonstrate how each can be used to create macromodels of high-speed channel, speeding up signal integrity simulations. When combining with a feed-forward neural network, the recurrent neural network can be used as a parametrized model, creating a tunable equalization circuit.

In the weakly nonlinear system regime, Volterra representation is widely acceptable due to its familiarity and analogy to time-invariant linear system theory. Similar to IBIS in the data collection, but require less training data compared to recurrent neural network or even IBIS, Volterra-Laguerre theory is shown to be very effective in modeling I/O buffers. On top of that, using a simple multidimensional interpolant, a parametrized model can be created and verified to match very well with transistor level circuit simulations.

At the final stage of the design verification process, a system level inte-

gration and assessment is needed. Simulations of such complicated systems involve many tools at different levels of physics (die, package, board). At the end of the day, the ultimate goal is to judge whether the integration works using a handful number of figure of merit. Therefore, a surrogate model whose inputs are the design variables and outputs are the figures of merit is needed to replace expensive and lengthy simulation. For example, in high-speed link design, a model that receives the channel's geometry and equalization settings and return the eye width and eye height would be highly appreciated by the designer as it would dramatically reduce the verification time and make optimizations become convenient. The last part of this work introduces Gaussian Process for this purpose. Through its full Bayesian treatment, the Gaussian Process appears to be an excellent candidate for a black-box surrogate modeling method due to its data efficiency and fast convergence. Other machine learning methods are also considered in a comparative study in which Gaussian Process performs consistently well as expected.

Dâng mẹ, Bụt hiện tiền.
To mom.

ACKNOWLEDGMENTS

This thesis marks an important milestone in my life. The existence of this thesis would be impossible without all the help and support of many people with whom I have had the hornor to meet and know.

First, I am forever grateful for my advisor, Professor Jose Schutt-Aine, for all the trust and confidence he has in me, and the help he has been providing me during my entire time at ECE Illinois. He has given me all the freedom any graduate student could ever dream of to explore different possibilities and always tried to make time for me even when I showed up at his office's door without any warnings. All the knowledge and skills I acquired would not be possible without him.

My doctoral committee has been helpful offering feedback to my work and given me many questions that make me have a better view of the bigger picture of the problems I was trying to solve. I want to express my deepest gratitude to Professor Jennifer Bernhard, Professor Andreas Cangellaris and Professor Erhan Kudeki, prominent experts of their fields, busy with administrative tasks and loaded with packed schedules, yet they spent time to read my work, listen to my prestentation and give me feedback so I have the chance to better improve it. I deeply appreciate the time they spent for me.

Next, I would like to thank Karen Kuhns for all the help around the office, without her, my office life would be much harder and I would not be able to focus on what matters most: my research. I also cannot thank Jennifer Merry from the ECE Graduate Advising Office enough, she has been helping me with my on-campus employment and internship paperwork, answering my questions related to paperwork and offer advice for my next step. I am grateful for Casey Smith and Wally Smith and the staffs in the ECE electronics shop for all the help they have been constantly providing me.

I also would like to thank my lab/office mates: Dr. Xiao Ma, Dr. Xinying

Wang, Hanzhi Ma, Nancy Zhao, and Bobi Shi for all the help as friends and collaboration as colleagues we have had together. I will always remember Dr. Pallavi Sharma and Sakshi Srivastava patiently answering my questions related to antenna design, many of which are naively dumb and time consuming to explain, and the time I spent with them in ECE110. I also would like to thank Dr. Tianjian (TJ) Lu for being a huge inspiration for a part of my work. I am grateful to Dr. Fred German and Matthew Young from Ansys for providing me a unique opportunity learning and working on something outside of my comfort zone, which has enriched my knowledge about the RF world. I thank Dr. Mikhail Popovich and Dr. Juan Ochoa for giving me the first internship experience and being the best bosses any intern could possibly ask for. I also had the chance to work on an interesting big data application project at Hewlett Packard Enterprise under the supervision of Dr. Chris Cheng and Dr. YongJin Choi. I thank them for such valuable experience.

I also want to express my appreciation to the Center For Advanced Electronics Through Machine Learning (CAEML) and its inspiring industrial members for funding my research.

I would like to thank several excellent undergraduate students that I had a chance to mentor toward the end of my doctoral program: Mark Vlasaty, Daniel Abdoue, Kevin Kim, Kevin Yu, Benjamin Lam, Phuoc Nguyen, Matt Lian, and many others. They were not only helping me improve my mentoring skills but were my inspiration to be a better role model, a better student, and a better person in general.

My achievement today would not be possible without my friends, my family, and my extended family. I thank them for all the unconditional love they have for me. I am ready for the next chapter of my life knowing I have been loved in many ways by many people.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Problem statement

With the increasing complexity in modern designs of electronic systems, the two advantages of macro-modeling, namely, protecting intellectual property (IP) and reducing simulation time, become more comparative to conventional transistor-level simulation. In additon, system level integration is an important stage of electronics design cycle. This step involves putting all pieces of a design together and verifying their functionalities as a whole. In most cases, the need for system level integration is obvious because parts used in a design are supplied by different vendors whose IPs need protecting. The term *macromodeling* used in this thesis refers to black-box macromodeling, a modeling problem in which there is little to no information about the internal structure of the system of interest. Most information is collected via the terminals of the system, either in the time domain or frequency domain. Macromodeling problems in which the underlying structure is fully or partially known are called *white-box* or *gray-box* problems, and are not in the scope of this work.

Linear macromodeling techniques could also be applied to linear active devices (such as a linear power amplifier), but they mostly focus on simulating passive structures such as interconnects, power distribution networks, packages etc. Vector Fitting (VF) [2] is the most common algorithm and has been used in most commercial solvers. Loewner matrix framework [3] was introduced a decade after VF but only became popular in recent years and are drawing attentions due to its ease in implementation. The scope of this thesis, however, focuses on nonlinear macromodeling. This is especially important for applications involve time-intensive simulation such as high-speed link.

In high-speed link design, a passive channel is simulated along with a transmitter (TX) and a receiver (RX) for millions of bits of data to collect statistically sufficient data for the whole link performance. Simulating nonlinear and time-varying circuits in TX/RX architecture are challenging. The two most popular methods to model such circuits are I/O Buffer Information Specification (IBIS) [4] and Mπlog [5]. Over times, these methods have been adopted, enhanced, and improved by many researchers in the community to capture more signal and power integrity (SPI) phenomena. Similar to other nonlinear macromodeling techniques, IBIS-alike models use time-domain data for model extraction.

In recent years, machine learning (ML) methods have gained a lot of attention because they are robust, capable of efficiently handling a large amount of data. This thesis introduces some most prominent algorithms such as Neural Network (NN) and Gaussian Process (GP) to modeling problems, in particular, the modeling and analysis of high-speed links. The thesis is organized as follows: Chapter 2 presents input-output recurrent neural network (RNN) modeling in high-speed link, we will demonstrate different types of RNN, their advantages and disavantages, how to train and use them for modeling purposes. Eye diagrams can be efficiently constructed from RNN generated waveforms with accuracy as high as those constructed from other methods. Chapter 3 investigates the I/O buffer modeling problem using Volterra/Wiener theory. It discusses both time-domain and frequency-domain modeling techniques. X-parameters[1] are also reviewed and discussed as a data source for high-speed link model extraction. Finally, high-speed link performance black-box prediction and optimization are studied in Chapter 4. Variability analysis and uncertainty quantification are also carried out. The mathematical tool for these analyses is Gaussian Process due to its non-parametric nature and computational flexibility.

1.2 A brief review of machine learning/neural network methods

[1]X-parameters is a trademark of Keysight Technologies. However, the X-parameters format and underlying equations are open to public and are well-documented.

In this section, a concise introduction to machine learning (ML) methods is presented to equip the readers with basic understanding of them and their terminologies. An excellent introductory literature about the core understanding of ML methods is [6]. ML methods, particularly neural networks which are the focus of this thesis, are mostly fitting tools, like any other traditional interpolation methods, they seek for a relationship between some inputs $\boldsymbol{x} \in \mathbb{R}^n$ and some outputs $\boldsymbol{y} \in \mathbb{R}^m$:

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) \tag{1.1}$$

where $\boldsymbol{f}$ is a vector-valued function.

Take the linear regression method as an analogy, without loss of generality, we will assume $n = 1$ and $m = 1$, the mapping $\boldsymbol{f}$ becomes a scalar function. First, a hypothesis is proposed, that is, an assumption is made about what form $\boldsymbol{f}$ takes, what family functions $\boldsymbol{f}$ could be. For example, "$\boldsymbol{f}$ is a polynomial order M" is a hypothesis which assumes that the relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$ is an order M polynomial. The larger the hypothesis, the more expressive $\boldsymbol{f}$ is, which means $\boldsymbol{f}$ can represent a more complicated mapping.

Given a hypothesis, a set of parameters, frequently denoted as $\boldsymbol{\theta}$, (also refered to as *hyper-parameters*) is introduced. In our polynomial hypothesis example, the polynomial coefficients are the model's unknown parameters. Equation (1.1) becomes:

$$y = \sum_{i=0}^{M} \theta_i x^i \tag{1.2}$$

To find the hyper-parameters, a set of training data is needed. They are a set of N input - output pairs $\left\{\boldsymbol{x}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right\}$ $i = 1, 2, ..., N$, $\hat{\boldsymbol{y}}$ is used to denote the known data while $\boldsymbol{y}$ is used to denote the output generated by the model. For example, $\boldsymbol{x}$ could be 20 geometry related variables ($n = 20$) in a microstrip line filter design such as widths or lengths of the microstrip segments, $\boldsymbol{y}$ could be the operating frequency or the $-3dB$ bandwidth ($m = 1$), or both of them, in which case, $m = 2$. Using these N data points, the hyper-parameters can be extracted by constructing a least-square solution if $\boldsymbol{y}$ and $\boldsymbol{\theta}$ are linear with respect to each other. In our polynomial hypothesis example, it is completely feasible to extract $\boldsymbol{\theta}$ by doing so. The data can be used to set up a system

of equations:

$$\begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(N)} \end{bmatrix} = \begin{bmatrix} 1 & x^{(1)} & \left(x^{(1)}\right)^2 & \cdots & \left(x^{(1)}\right)^M \\ 1 & x^{(2)} & \left(x^{(2)}\right)^2 & \cdots & \left(x^{(2)}\right)^M \\ \vdots & & & & \\ 1 & x^{(N)} & \left(x^{(N)}\right)^2 & \cdots & \left(x^{(N)}\right)^M \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_2 \\ \vdots \\ \theta_M \end{bmatrix} \tag{1.3}$$

For $N \gg M$, Equation (1.3) is an over-determined system. $\boldsymbol{\theta}$ is the least square solution. This type of ML method is called the supervised learning method, named due to the fact that pairs of input - output are given to find the model's parameters. Unsupervised learning methods do not have pairs of input - output as data nor to find some mappings. An analogy of unsupervised learning is performing principle component analysis (PCA) on a data set to reduce the dimensionality. Both types of learning methods can be used together to achieve the best modeling solution. For example, not all 20 variables of the filter design may contribute significantly to its $-3dB$ bandwidth. To efficiently predict the $-3dB$ bandwidth of the filter, we can apply PCA on the input data to find, say, 8 most significant components in the input that strongly affect the bandwidth (unsupervised), then construct a regression model to learn the mapping between these 8 most significant components and the bandwidth (supervised). We will see this idea again in Chapter 4 with the Partial Least Square Regression method.

In some cases, the hypothesis is too complicated, a system of equations like Equation (1.3) is infeasble to obtain. The hyper-parameters have to be extracted by an iterative algorithm. At its core, solving for $\boldsymbol{\theta}$ is to solve the optimization problem:

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \, \mathcal{L}\left(\boldsymbol{y}, \hat{\boldsymbol{y}}\right) \tag{1.4}$$

where $\mathcal{L}\left(\boldsymbol{y}, \hat{\boldsymbol{y}}\right)$ is called a loss (cost) function, measured how well the model output matches the observed data. The most popular loss function is the mean-square error (MSE) between the two, defined by:

$$\mathrm{MSE}\left(\boldsymbol{y}, \hat{\boldsymbol{y}}\right) = \left\|\boldsymbol{y} - \hat{\boldsymbol{y}}\right\|_2^2 \tag{1.5}$$

where $\|\cdot\|_2$ is the 2-norm (Euclidean norm). Gradient descent [7] is a variety of iterative algorithms to solve for $\boldsymbol{\theta}$ using the simple idea: follow the oposite direction of the gradient to arrive at the minimum of a function. Gradient

descent optimization methods are popular because they only require the first order derivative of $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$. Their convergence rates are slower than those of other optimization methods that rely on higher order derivatives but they require less computations and storage. They start with a random initial value of $\boldsymbol{\theta}$ then iteratively update $\boldsymbol{\theta}$ as data is fed in. At iteration k, we have:

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \nabla_{\boldsymbol{\theta}^{(k-1)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{1.6}$$

where η is called the *learning rate*. The learning rate is one important parameter of the training process. Too large η will lead to the solution bouncing around the optimal value while too small η will unnecessarily prolong the training time. Modern gradient descent based algorithms use adaptive learning rate to speed up the training process when it is allegedly far away from the solution and slow down the steps when it appears to be close to a potential optimal solution. Due to the high complexity in the hypothesis, $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ could have multiple local, suboptimal solutions. Modern gradient descent based algorithms adjust the learning rate during the solving process in order to escape these local minimums and to have higher chances arriving at the global minimum. Note that because $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ could be non-convex, there could be more than one global minimum. For example, the bird function [8] shown in Figure 1.1 is given by:

$$f(x_1, x_2) = \sin(x_1)\, \mathrm{e}^{(1-\cos(x_2))^2} + \cos(x_2)\, \mathrm{e}^{(1-\sin(x_1))^2} + (x_1 + x_2)^2 \tag{1.7}$$

has two global minima at $(x_1, x_2) = (4.70104, 3.15294)$ and $(x_1, x_2) = (-1.58214, -3.13024)$ with the minimum function value of -106.764537.

Moreover, many of these algorithms recognize that feeding the whole dataset into the calculation of the gradient puts a burdern on the computation, they often split data into batches and feed the data into the algorithm one batch after another. These derive a whole family of algorithms called stochastic gradient descent. One round feeding the whole dataset to the optimization algorithm is called one epoch. Typically for the optimization to converge, multiple epochs are required. In-depth discussion and overview of different gradient descent optimization methods can be found in [7].

If the hypothesis is complex, Equation (1.4) represents a highly non-convex optimization problem. The solution can be highly unreliable. For example,

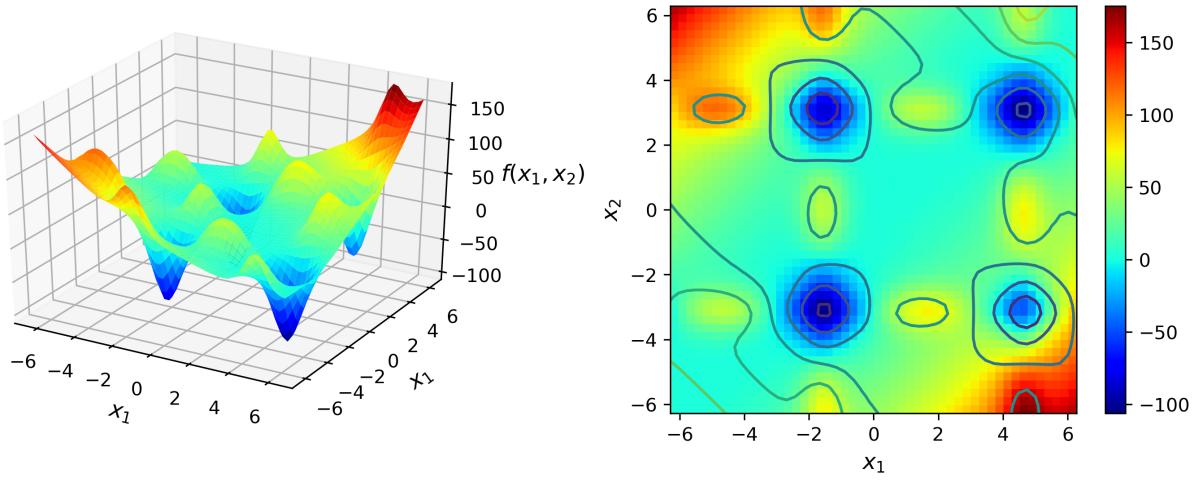


Figure 1.1: A non-convex function with more than one minimum.

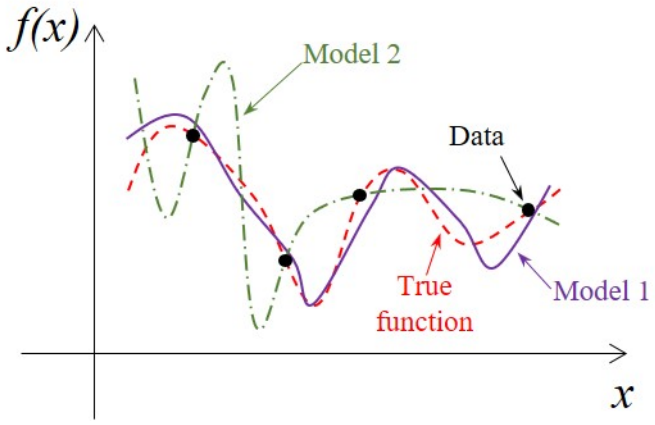
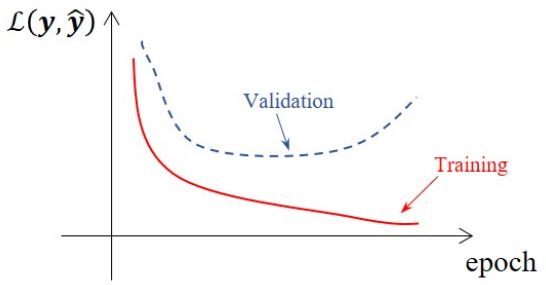
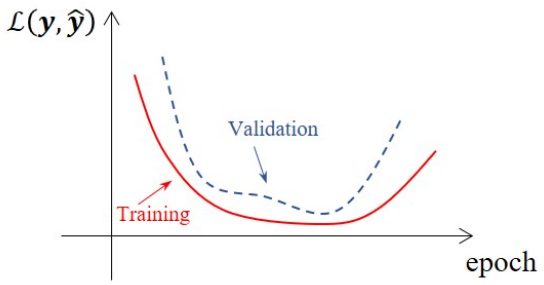


Figure 1.2: Overfiting happens to Model 2.

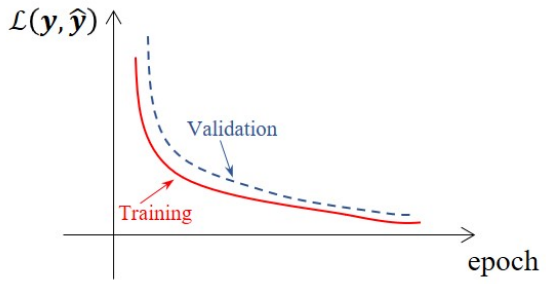
Figure 1.2 shows two different models and their fitting to the data compared to the true underlying function. Model 1 obviously matches the underlying function better than Model 2 even though Model 2 has lower fitting error. In fact, the fitting error of Model 2 is 0 because it goes through all the provided data points while that of Model 1 is small but not 0. This is called *overfitting*. We realize Model 2 in Figure 1.2 is a bad solution because we were able to observe its evaluations at other points than the provided data (black dots). This gives us a hint on how to properly manage the data to avoid overfitting. The observed/collected data is always split into two parts, typically 90% - 10% or 80% - 20%. The major part is called the training data and the minor part is called the validation data. The loss functions calculated using training data and validation data are called training loss and validation loss respectively. The training data is fed into the optimization algorithm to solve for $\boldsymbol{\theta}$. It is usually refered to as *the data seen by the model*. While the validation data is only used to calculate the validation loss at the end of each epoch, it is refered to as *the unseen data*. Figure 1.3 shows a few typical scenarios of what could happen with the training and validation loss and the insights they provide. As the epoch progresses (the optimization algorithm iteratively converges $\boldsymbol{\theta}$ to a solution), the relative position between training and validation loss could tell us how the training is and the possible problem (if any). For instance, Figure 1.3a shows what the losses would look like when we have overfitting. The losses match with what is shown in Figure 1.2. The model fits the seen data very well, the training loss is very small, but when the unseen data is fed into the model, its prediction is far off from the true value, resulting in very high validation loss. Figure 1.3b, on the other hand, presents a different problem. At first, the validation and training loss track each other very well and trendind down. This is a good sign that the model is converging and perform relatively well on unseen data. However, toward the end, suddenly they both get larger. This could very well be due to the learning rate was not properly adjusted. In particular, the learning rate was too large when $\boldsymbol{\theta}$ was approaching the right solution. Due to the large step, the algorithm missed the minimum and did not converge, moving away from the optimal solution, making the losses grow again. Finally, Figure 1.3c and Figure 1.3d show typical losses for a well converged model. The losses are trending down in both cases, they track each other well. The only difference is that in Figure 1.3d, the validation loss appear to be lower than the training



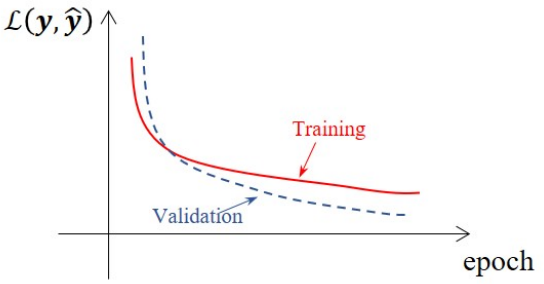
(a) Overfitting.



(b) Large learning rate toward the end of training.



(c) Good convergence.



(d) Good convergence. Testing loss being lower than training loss is typical result from using regularization during training.

Figure 1.3: Tracking of training and testing loss could provide more insight into the convergence of the model.

loss. If the model has some kinds of regularization, this is the expected behavior.

Regularization is a technique that modifies the loss function to keep the balance between the fitting ability and the model complexity. The loss function introduced above has a single term: a measure of how well the fit is (i.e. the MSE). But the regularized loss function has an additional term to enforce some conditions about the model's hyper-parameters. For example, the following loss function:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \|\hat{\boldsymbol{y}} - \boldsymbol{f}(\boldsymbol{x})\|_2^2 + \lambda \|\boldsymbol{\theta}\|_1 \tag{1.8}$$

where λ is a regularization coefficient, is called the Lasso regularized loss function [9]. Recall that $\boldsymbol{f}$ is parametrized by $\boldsymbol{\theta}$. Hence, in Equation (1.8), $\boldsymbol{\theta}$ appears in both the first term (2-norm) and the second term (1-norm). The first term simply measures how large the fitting error is, while the second term is large if $\boldsymbol{\theta}$ is dense (i.e. all hyper-parameters are non-zeros). To

minimize $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ in Equation (1.8), we are not looking for any solution that minimize the fitting error, we are actually looking for a sparse solution.

Advanced regularization methods for neural network such as dropout [10] randomly turn off some of the neurons in the network, forcing the neural network to make predictions without the contribution of those neurons to avoid overfitting. Training processes ultilize dropout as regularization will have slightly worse performance during training but better performance when validating, which explains the losses behavior in Figure 1.3d.

After the model is converged, a seperate set of data is used to test the model. The test dataset may appear to be same as the validation set, the process of feeding it through the model with fixed hyper-parameters also appears to be the same, but they are very different in nature. The validation dataset is used to test the model during training to check for convergence, the validation loss is used to compare against the training loss to give insight about what could be happening in the training process. For example, we could terminate the training early, or use the information about training and validation loss to appropriately adjust the learning rate or other hyper-parameters in the model such as the inital value of $\boldsymbol{\theta}$, the hypothesis etc. The test dataset is strictly used for testing purpose only. In practice, usually there is a single set of data. It should be shuffled and split into 2 sets: training and testing, the training set is then further split into actual training and validation as above. Shuffling the data is very important it ensures that the data is evenly distributed over the domain. If the model does not have any regularization but the losses behave like Figure 1.3d, it is an indicator that the data was not randomly shuffled. Too many *difficult* samples were in the training set while the test set has too *easy* samples. Figure 1.4 visualize this situation, because the data was not evenly distributed, the training data is located in the highly nonlinear region (*difficult* to learn) while the validation (or test) data is located in the fairly linear region (*easy* to learn), a model trained with this dataset will have training and validation loss exactly same as Figure 1.3d. In this case, because model was trained on difficult samples, it will perform well on the test set. However, if it happened to be the opposite, the performance of the model will be extremely poor on the test set.

In the case of neural network, a hypothesis is a composition of a series of weighted functions. For example, a 2-layer feed-forward neural network with

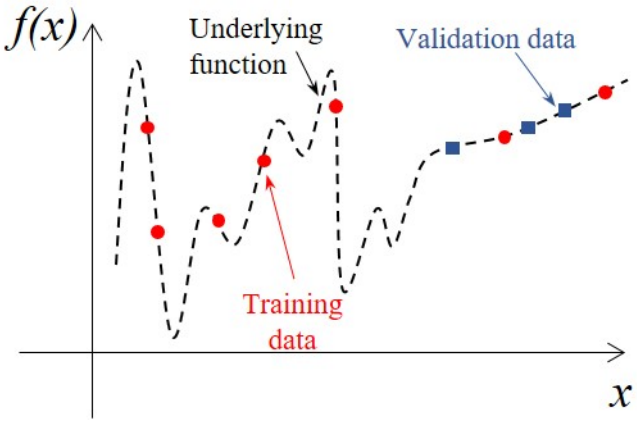


Figure 1.4: Unevenly distributed data.

tanh as the *activation function* is mathematically represented as:

$$y = \tanh\left[W_2 z\right] = \tanh\left[W_2 \tanh\left[W_1 x\right]\right] \tag{1.9}$$

where W_1, W_2 are the weights, z is an intermediate signal, output of the first layer and input of the second layer. To optimize the MSE loss, we need to be able to evaluate its gradient:

$$\begin{aligned} \nabla_W \text{MSE}\left(\boldsymbol{y}, \hat{\boldsymbol{y}}\right) &= \nabla_W \left\|\boldsymbol{y} - \hat{\boldsymbol{y}}\right\|_2^2 \\ &= \nabla_W \left(\tanh\left[W_2 \tanh\left[W_1 x\right]\right] - \hat{y}\right)^2 \end{aligned} \tag{1.10}$$

Manual calculations of the gradients are straightforward. However, automatic calculation is needed for robustness and flexibility in building different neural network architectures. When the activation function changes, there should be no extra effort to reimplement the gradient calculation. The automatic calculation of the gradient is called *autodifferentiation* and can be done thanks to chain rule. In machine learning terms, the function that implements a mapping between two variables with many intermediate variables in between is called a *computational graph*. Figure 1.5 shows a computational graph which happens to be as same as the feed-forward neural network. The *nodes* are nonlinear functions f_i and g while the *branches* are the input and output signals of each nodes. The *forward path* of the graph reads:

$$\mathcal{L} = g\left(y\right)$$

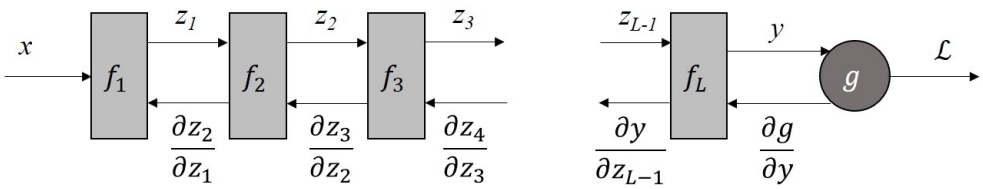


Figure 1.5: Forward and backward signal flows in a computational graph.

$$y = f_L(z_L)$$

$$z_L = f_{L-1}(z_{L-1})$$

$$\dots$$

$$z_1 = f_1(x)$$

Compactly, we could write:

$$\mathcal{L} = g \bullet f_L \bullet f_{L-1} \bullet \dots \bullet f_1 \bullet x$$

where $\bullet$ is the composition operator, that is, $g \bullet f \bullet x = g(f(x))$. Note that each $f_i,\ i = 1, 2, ..., L$ has hyper-parameters embedded in it. For example, in Equation (1.9), $f_1 \equiv f_2 \equiv \tanh$, the hyper-parameter of f_1 is W_1 and that of f_2 is W_2.

To take the derivative of $\mathcal{L}$ w.r.t the hyper-parameter, say W_i, we need to evaluate:

$$\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial g}{\partial y}\frac{\partial y}{\partial z_{L-1}}\frac{\partial z_{L-1}}{\partial z_{L-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i}\frac{\partial z_i}{\partial W_i} \tag{1.11}$$

Comparing the terms in Equation (1.11) with where they are in the graph in Figure 1.5, we realize that each term the signal flowing backward through each node. Any node in the graph receive some gradient information from the node after it (to the right in Figure 1.5), calculate the term it's responsible, then pass that information to the node after it (to the left in Figure 1.5). This process of having each node in the graph handle a part of the derivative that is corresponding to it and hand off the information to the next node that is closer to the target weight is called *backpropagation*. Implementation wise, if a node has an analytical expression for evaluating the forward pass ($z_i = f_i(z_{i-1})$), it should also has an analytical expression to perform the backward pass (i.e. calculate $\frac{\partial z_i}{\partial z_{i-1}}$).

Pytorch [11] is a Python framework that support autodifferentiation. Af-

ter declaring and registering a variable in Pytorch, it will keep track of all variables and all of the nodes they input to. The user just needs to define the forward pass using Pytorch's standard mathematical functions. We have to use Pytorch's mathematical functions because only then, Pytorch's autodifferentiation feature will be able to recognize them and use the corresponding derivatives of those functions when calculating the derivative. Backpropagation then can be performed to calculate the gradients needed for the optimizer. Pytorch uses *Tensor* datatype with the *grad* attribute to keep track of the computational graph and its nodes for gradient propagation. Data in this thesis is generated using Ansys Electronics Desktop and Keysight Advanced Design System, implementation of ML models and simulations is done in Python, Pytorch is the package of choice for autodifferentiation suport and neural network implementation.

Chapter 2

HIGH-SPEED LINK MODELING WITH RECURRENT NEURAL NETWORK (RNN)

2.1 Introduction

In the area of signal integrity, eye diagrams have become important metrics to assess the performance of a high-speed channel. In order to generate an eye diagram, transient waveforms are first obtained from a circuit simulator and then overlaid. Generating eye diagrams by using a circuit simulator can be very computationally intensive, especially in the presence of nonlinearities. As shown in Figure 2.1, there are often several Newton-like iterations involved at every time step when a SPICE-like circuit simulator handles a nonlinear system in the transient regime [1]. Given the size of a practical and large-scale circuit, the runtime of a circuit simulator on modern processors can be hours, days, or even weeks. There are many efforts in seeking novel numerical techniques to improve the computation efficiency of a circuit simulator. For example, hardware accelerators including FPGAs [1] and GPUs [12] are used to achieve the acceleration of matrix factorization in a circuit simulator. There is also work in efficiently generating eye diagrams, for example, using shorter bit patterns instead of the pseudo-random bit sequence as input sources to simulate the worst-case eye diagram [13]. In this work, we propose taking a different route and using machine learning methods, to be specific, recurrent neural network (RNN), to improve the efficiency of a transient simulation with nonlinear circuits, specifically, applied to high-speed link simulation.

Recently, many remarkable results are reported on modern time-series techniques by using RNN in the fields such as language modeling, machine translation, chatbot, and forecasting [14–18]. There are also a number of prior attempts in incorporating RNN into modeling and simulating electronic devices and systems. For example, researchers propose combining a NARX

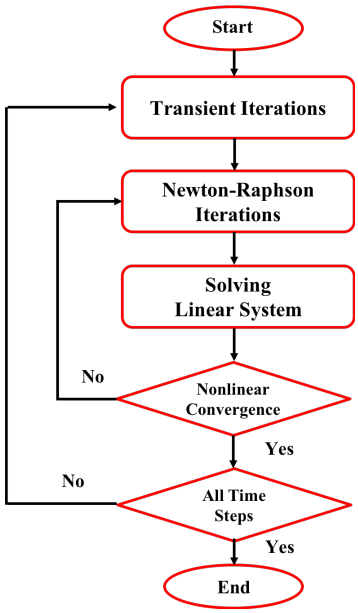


Figure 2.1: Flow chart of a circuit simulator [1].

(nonlinear auto-regressive network) topology with a feedforward neural network in modeling nonlinear RF devices [19], most recently, effort to model transmitter I/O buffers using this topology was also reported in [20] and references therein. A variant of RNN, known as Elman RNN (ERNN), is applied in simulating digital designs [21,22]. More recently, researchers present an ERNN-based model in simulating electrostatic discharge (ESD) [23]. The aforementioned two topologies, to be specific, NARX-RNN and ERNN, will be discussed in details in the following section. It is also worth mentioning that ML methods in general have been applied to many applications related to electronic designs such as modeling high-speed channels [24–26], replacing computationally expensive full-wave electromagnetic simulations [27,28], and building macro-model from S-parameters [29,30]. The difference between ERNN and NARX-based RNN will be discussed thoroughly.

Through the proposed approach, a modern ERNN is first trained and then validated on a relatively short sequence generated from a transient simulation. Once the training completes, the RNN can be used to make predictions on the remaining sequence in order to generate an eye diagram. The training cost can also be amortized when the trained RNN starts making predictions. As the time-domain waveforms are generated from RNN through inference instead of iterations of solving linear systems involved in a circuit

simulator, it significantly improves the computation efficiency. Besides, the proposed approach requires no complex circuit simulations nor substantial domain knowledge. We demonstrate through two high-speed link examples that the proposed approach can meet the accuracy of transistor-level simulation while the run time can be dramatically reduced.

We also investigate the performance of ERNNs built with different recurrent units, to be specific, vanilla recurrent neural network (VRNN) and long short-term memory (LSTM) unit in generating accurate eye diagrams. It is shown that the LSTM network outperforms VRNN in terms of both convergence and accuracy. The numerical issue of gradient vanishing or explosion during back propagation in the VRNN is also well resolved in the LSTM network. The activation function in this work is chosen as the rectified linear unit (ReLU) [31] as it enables better numerical stability and higher efficiency in training. *Adam* [32] optimizer is found to stand out among investigated optimizers such as Stochastic Gradient Descent (SGD) [33], Root Mean Square Propagation (RMSProp) [34] for fast convergence in training. It is also shown that with the training scheme proposed in this work, training sample length plays an important role in the convergence of the RNN.

2.2 Recurrent neural network

Understanding RNN cannot be separated from the feed-forward neural network (FNN), which consists of multiple layers of neurons. Unlike a FNN, in which the signal flows unidirectionally from the input to the output, an RNN has, in addition, a feedback loop from the output to the input. The FNN is a universal approximator, which can be written as the following:

$$y = f_L \circ f_{L-1} \circ ... \circ f_1 \circ x, \tag{2.1}$$

where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ represent the input and the output, respectively, f_l $(l = 1, 2..., L)$ is the weighted activation, and $\circ$ denotes the composition operation. As a comparison, the RNN can be understood as a universal Turing machine in the form of:

$$\begin{cases} h_t = g_h\left(x_t, h_{t-1}\right) \\ y_t = g_o\left(h_t\right) \end{cases} \tag{2.2}$$

where h_t and x_t are the hidden state and the input at time t, respectively, and g_h and g_o are weighted activations. y_t is the predictive output at time t. A *loss function* quantifying how *close in some sense* the prediction y_t is to the true value $\tilde{y}_t$ corresponding to the input x. Since both input and output are real values, it is most convenient to choose *mean-square error* (MSE) loss function. MSE loss is calculated as the square of 2-norm of the error vector.

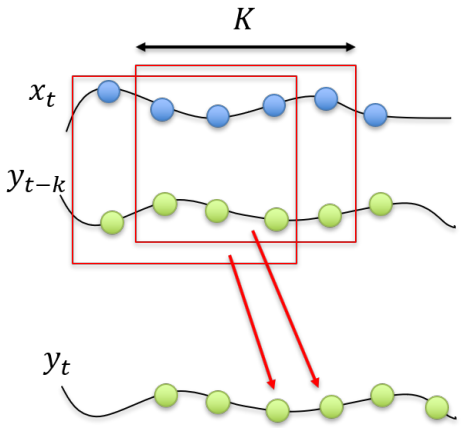
$$E_t\left(\tilde{y}_t, y_t\right) = \mathrm{MSE}\left(\tilde{y}_t, y_t\right) = \|\hat{y}_t - y_t\|_2^2 \tag{2.3}$$

Similar to that in a dynamical system, the concept of state is employed to describe the temporal evolution of a system, the power of an RNN in dealing with time-series tasks arises from the special variable, namely, the hidden (internal) state h_t. In system identification, the mappings including both g_h and g_o in Equation (2.2)) are learned via a least-square alike approximation process during which a set of pre-defined parameters are tuned. Similar models to the one described by Equation (2.2) can be found in the *autoregressive (AR)* family model, which are also very popular for time-series tasks. The models of an AR family can often be implemented with:

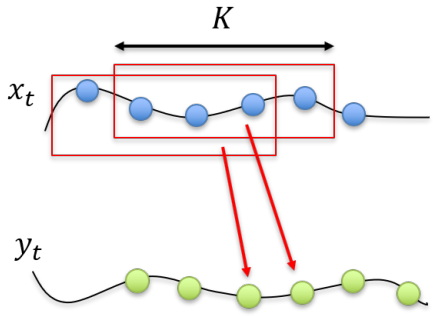
$$y_t = g\left(x_{t-i}, y_{t-j}\right) \quad 0 \leq i \leq K_x, 0 \leq j < K_y, \tag{2.4}$$

where K_x and K_y are known as the memory length of the input and output, respectively. It can be seen from Equation (2.4) that there is no explicit hidden state; instead, the feedback comes from the delayed versions of the output. In order to differentiate the mechanism described in Equation (2.4), the RNN with explicitly defined hidden states are often called the Elman RNN (ERNN) [35]. We use RNN to denote ERNN for simplicity. The term NARX-RNN and output-feedback RNN will be used interchangeably to refer to AR-based RNN which is described by Equation (2.4).

The difference between ERNN and NARX-RNN can be summarized by the visualization in Figure 2.2. It is obvious that ERNN structure detaches the dependency of the current time step output from the past time step ones. This is significantly speeding up the training and prediction process.



(a) NARX-RNN.



(b) ERNN.

Figure 2.2: Differences in input - output of NARX-RNN and ERNN.

It is often beneficial to unroll an RNN, which will ease the understanding for why the learning process of an RNN could be computationally intractable and how it is made tractable. As shown in Figure 2.3, the RNN is unrolled such that it can be fed with an input sequence of K time steps. The signal propagating through a unit in the unrolled RNN can thus be written as:

$$h_t = \phi_h \left(W_{ih} x_t + W_{hh} h_{t-1} \right) \tag{2.5}$$

and the output of the RNN unit is given by:

$$y_t = h_t, \tag{2.6}$$

where ϕ_h is the nonlinear activation function and W's are of appropriate dimensions and contain the tunable weights. It is worth mentioning that one can always add a fully connected layer to y_t in Equation (2.6) to transform

it into the desired form, which is also the reason why modern formulation of RNN takes the current state as the output.

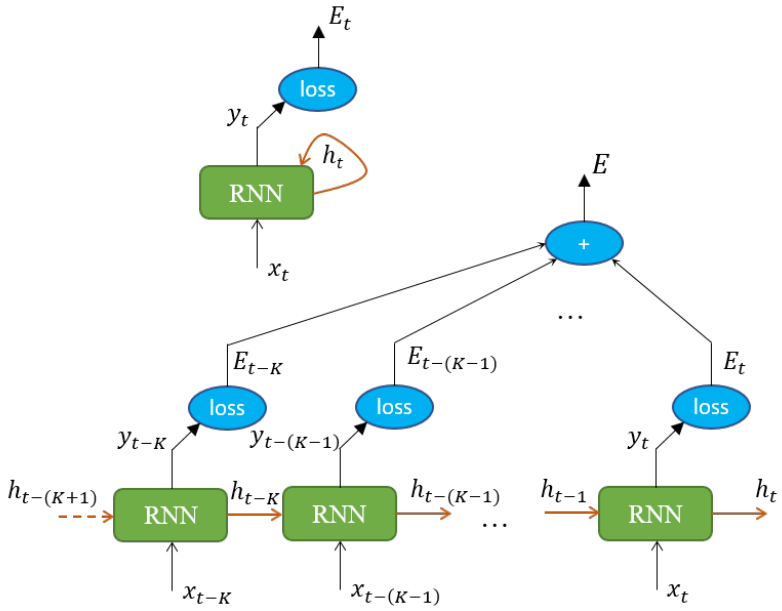


Figure 2.3: An unrolled RNN with input sequence of K steps with $\tilde{y}_\tau$ and E_τ representing the prediction and the corresponding loss (error) at time step t.

The unrolled RNN looks like a deep FNN (DNN), but the weights are shared across the units over time. It is an advantage of RNN over FNN as by unrolling the RNN, one obtains a DNN of the same number of layers but with much fewer parameters. Unfortunately, this also leads to disadvantages of RNN, which can be understood in the following. The gradient of the loss E at output with respect to a parameter θ can be written as:

$$\frac{\partial E}{\partial \theta} = \sum_{\tau=1}^{t} \frac{\partial E_\tau}{\partial \theta}, \tag{2.7}$$

where

$$\frac{\partial E_\tau}{\partial \theta} = \sum_{j=1}^{\tau} \frac{\partial E_\tau}{\partial y_\tau} \frac{\partial y_\tau}{\partial h_\tau} \frac{\partial h_\tau}{\partial h_j} \frac{\partial h_j}{\partial \theta} \tag{2.8}$$

The parameters in the RNN are updated through the back-propagation of the calculated gradients. After the k^{th} iteration the weight W (W denotes any of the weights in the neural network defined above) is updated by gradient

decsent:

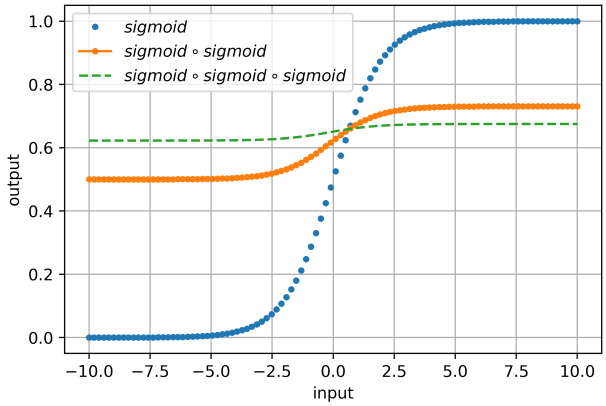
$$W^{(k+1)} = W^{(k)} - \eta_k \nabla_{W^{(k)}} E \tag{2.9}$$

where η_k is the learning rate at iteration k. The back-propagation of the gradients from time τ are done through all possible routes toward the past, which is also known as back-propagation through time (BPTT).

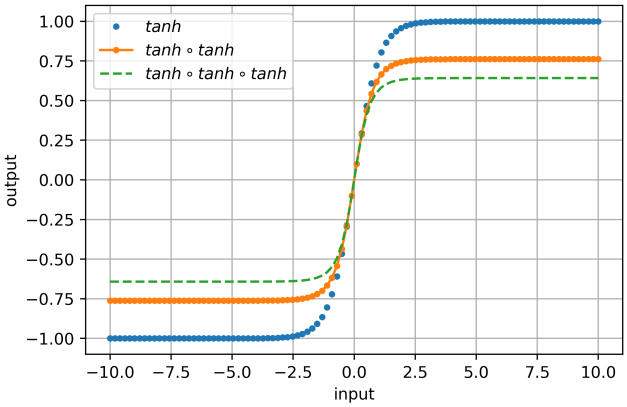
The disadvantages of BPTT is its computation inefficiency and numerical instability because at any time step τ, the calculation of the loss E_τ depends on all previous quantities in all previous time steps. It can be seen that with BPTT, the longer the sequence with which the RNN is trained, the more challenging the computation becomes considering both the degraded convergence rate and the increased demand in computing resources. Another numerical issue associated with the gradients with BPTT is that as the span of the temporal dependencies increases, the gradients tend to vanish or explode. The Jacobian term in the gradient of the loss function, $\frac{\partial h_\tau}{\partial h_j}$ in Equation (2.8) can be proved to be upper bounded by a geometric series [36]:

$$\left\| \frac{\partial h_\tau}{\partial h_j} \right\| < \gamma^{\tau - j}, \tag{2.10}$$

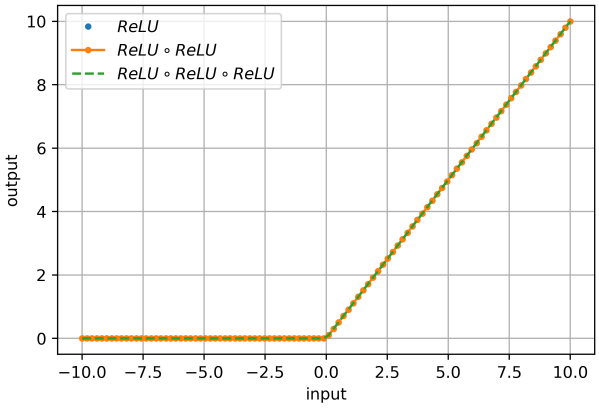
where γ is a constant decided by the norm of the nonlinearity in RNN. When hyperbolic tangent function, tanh, is chosen as the activation function, $\gamma = 1$ and for sigmoid, $\gamma = 0.25$ [36]. Therefore, the gradient either explodes or vanishes. We can use a simple numerical experiment to demonstrate the gradient vanishing and explosion. As shown in Figure 2.4, an input signal whose magnitude ranges from -10 to 10 is passed through various types of activation functions in multiple times. After the third time, the signal is flattened when sigmoid is taken as the activation function. Due to the vanishing of the gradients, sigmoid cannot be used as the activation function in a RNN unit. In contrast, as shown in Figure 2.4(c) when ReLU is taken as the activation function, the signal sustains its original shape after being passed through the unit for several iterations, which is also the reason why ReLU is very popular in modern RNN structures.



(a) Sigmoid.



(b) Tanh.



(c) ReLU.

Figure 2.4: Multiple passes through the same activation function.

One remedy to the problem of gradient vanishing or exploding is known as truncated back-propagation through time (TBPTT) [37, 38], which is a modified version of BPTT. The TBPTT used in this work assume that the signals only have memory length K. Equation (2.5) and Equation (2.6) can be explicitly written as:

$$\begin{cases} h_t = g_h\left(x_t, x_{t-1}, \ldots, x_{t-(K-1)}, h_{t-1}, h_{t-2}, \ldots, h_{t-(K-1)}\right) \\ y_t = g_o\left(h_t\right) \end{cases} \tag{2.11}$$

Another remedy utilizes a more sophisticated activation function with gating units to deal with problem of gradient vanishing or explosion, for example, the long short-term memory (LSTM) unit [39] and the gated recurrent unit (GRU) [40]. Both LSTM unit and GRU own *gates*, which allow the RNN cell to *forget*. The LSTM network implements the mapping $g_h(\cdot)$ with a more complicated mechanism than that of the vanilla RNN which is shown in Equation (2.5).

$$\begin{cases} i_t = \sigma\left(W_{ii}x_t + W_{hi}h_{t-1}\right) \\ f_t = \sigma\left(W_{if}x_t + W_{hf}h_{t-1}\right) \\ g_t = \tanh\left(W_{ig}x_t + W_{hg}h_{t-1}\right) \\ o_t = \sigma\left(W_{io}x_t + W_{ho}h_{t-1}\right) \\ c_t = f_t c_{t-1} + i_t g_t \\ h_t = o_t \tanh\left(c_t\right), \end{cases} \tag{2.12}$$

where h_t is hidden state at time t, c_t is called the cell state, and i_t, f_t, g_t and o_t are the input, forget, cell and output gates respectively. All of the W's are learnable weight matrices.

LSTM unit is chosen for this work due to its ablity to store and retrieve relevant information from the past time steps by using gating control signals, which resolves the issue of gradient vanishing or explosion [39]. The LSTM unit is used as the basic building block for the RNN structure. Figure 2.5 graphically shows the RNN unrolled K time steps in the horizontal direction, each square represents an LSTM unit. When processing an input sequence of length K, K units are cascaded such that h_{t-j} is the input to $h_{t-(j-1)}$ for $j = 0, 1, ..., K-1$. Each sequence of these LSTM units is then stacked on top of another to create stacked sequences which deepens the RNN. In

Figure 2.5, L stacks of sequence of length K are used. The output of the L^{th} layer, $\boldsymbol{h^{(L)}} = \left[\begin{array}{cccc} h_1^{(L)^T} & h_2^{(L)^T} & \cdots & h_K^{(L)^T} \end{array} \right]^T$, is then used to map to the output y_t.

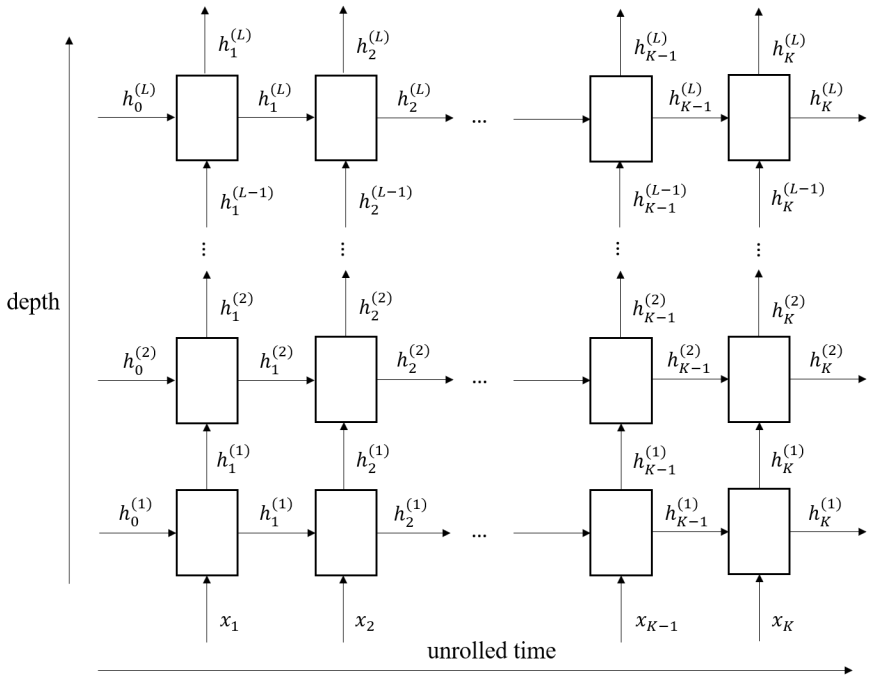


Figure 2.5: RNN cells are stacked up and concatenated to form a deep RNN.

Mathematically, the K time step truncated full stack RNN is now:

$$\begin{cases} \boldsymbol{h_t} = g_h^{(L)} \circ g_h^{(L-1)} \circ \cdots \circ g_h^{(2)} \circ g_h^{(1)} \left(\boldsymbol{x_t}, \boldsymbol{h_{t-1}} \right) \\ y_t = g_o \left(\boldsymbol{h_t} \right), \end{cases} \tag{2.13}$$

where

$$\boldsymbol{x_t} = \begin{bmatrix} x_{t-(K-1)} \\ x_{t-(K-2)} \\ \cdots \\ x_t \end{bmatrix} \in \mathbb{R}^{d_1 \times K}$$

$$\boldsymbol{h_t} = \begin{bmatrix} h_{t-(K-1)} \\ h_{t-(K-1)} \\ \cdots \\ h_t \end{bmatrix} \in \mathbb{R}^{l_h \times K}$$

and $h_t \in \mathbb{R}^{l_h}$ where l_h is the dimension of the hidden unit at time step t.

Following Equation (2.7), the gradient of the loss function of an L layer, K time step unrolled RNN at time t w.r.t. the parameter θ is:

$$\frac{\partial E}{\partial \theta} = \sum_{\tau=t-(K-1)}^{t} \frac{\partial E_\tau}{\partial \theta} \tag{2.14}$$

where

$$\frac{\partial E_\tau}{\partial \theta} = \sum_{j=t-(K-1)}^{\tau} \frac{\partial E_\tau}{\partial y_\tau} \frac{\partial y_\tau}{\partial \boldsymbol{h_\tau}} \frac{\partial \boldsymbol{h_\tau}}{\partial \boldsymbol{h_j}} \frac{\partial \boldsymbol{h_j}}{\partial \theta} \tag{2.15}$$

with $\boldsymbol{h_j}$, $\boldsymbol{h_\tau}$ are given by Equation (2.13).

The most trivial way to train an RNN is the *readout* training as shown in Figure 2.6. The *readout* training takes the output of the previous time step as the input. The ground-truth $\tilde{y}_k$ is only used in calculating loss with the corresponding prediction y_k. The RNN is fed with what it generated, which is also the reason it is called *readout*. *Readout* is mostly adopted in inference, i.e. when predictions are being made on the unseen data. Training in *readout* mode often takes longer time on convergence because the model has to make a lot of mistakes, being penalized for many times before it eventually learns to generate accurate predictions. Therefore, *teacher force* training is often preferred over *readout*. In *teacher force* training as illustrated in Figure 2.7, the ground-truth values are fed into an RNN as input.

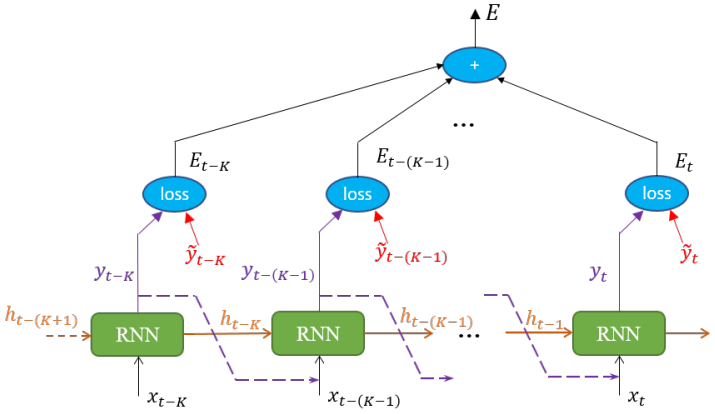


Figure 2.6: Readout training.

However, teacher forcing can only ensure an RNN to learn faster but not necessarily better. Similar to the mechanism behind overfitting, the underlying distribution of the input data in *teacher force* training may be very different from that during its *readout* mode inference. In that case, *teacher*

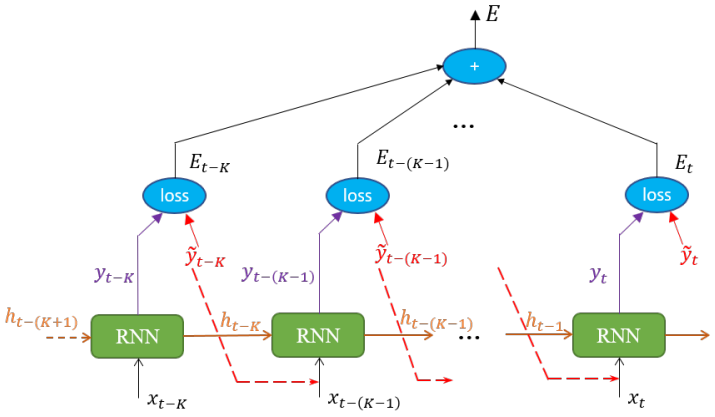


Figure 2.7: Teacher force training.

force training may have worse performance on unseen data comparing to its performance on the training set. To filter out the potential bias in training, we adopt a scheduling process in training as introduced by [41]. A good analogy of the scheduling process is the event of flipping a coin: we can imagine that a coin is flipped every time before the previous output is fed into an RNN as the input. The coin used in the scheduling process is biased: for the first few training epochs, the coin is biased towards the training data distribution such that the training is more into a *teacher force* mode; as the training evolves, the coin becomes biased towards the distribution of the predicted data, in other words, in a *readout* mode. In other words, at the begining of the training, the RNN is fed with true values of the output so it can converge faster, toward the end of the training, the RNN eventually gets fed with its prediction to improve its generalization ability. A simple implementation for such scheduling is using a Bernoulli random variable with a decaying sucess probability at epoch i $(i = 0, 1, ...)$ given by

$$p\left(i\right) = \frac{K}{K + \exp\left(\frac{i-1}{K}\right)} \tag{2.16}$$

Typical nonlinear circuits involved in a high-speed link are parameterized by some control variables. In particular, the equalization circuits often have so-called tap values that are decided by the link designers. Often times, the taps are set using some sort of optimization routines. In any cases, having a macro-model of the equalization circuit with the ability to tune the tap values without having to re-train the model is desirable. Noting that

the dynamical behavior of the RNN is controlled not only by the weights but also a hidden initial state which, in most cases, is set to zero when training an RNN model, we propose to make use of this hidden state to parametrize the RNN. An FNN is used to learn the mapping between the tap values and this latent variable which, in turn, controls the dynamical behavior of the RNN. This is feasible because for an equalization circuit, changing the tap values should not drammatically change its behavior. Figure 2.8 illustrates how FNN and RNN can be combined to create tunable models. The gradient is now back-propagated through RNN, then continue to back-propagate through the FNN, then weights of both networks are updated at the same time. Pytorch's auto-differentiation throughout the computational graph should be able to handle this configuration without much more effort when training RNN or FNN separately.

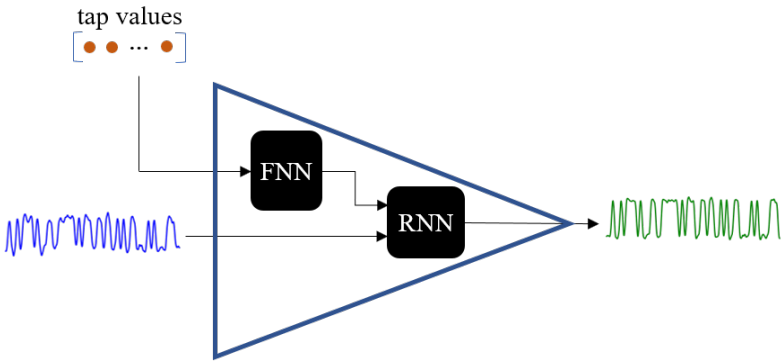


Figure 2.8: FRNN signal flow.

2.3 Numerical examples

The robustness of the proposed method using RNN to model high-speed channels will be illustrated through two different types of RNN presented in section 2.2 using a PAM2 and a PAM4 driver circuit. Different training conditions such as optimization method, memory length and recurrent cell topology etc. will be investigated. An example using FNN and RNN combined to model a receiver decision feedback equalizer is also presented. First, let us investigate a toy example: the Lorenz attractor.

2.3.1 Toy example: Lorenz system

The first example we will look into is the Lorenz system [42]. Lorenz model is a *simplified* model for atmospheric convection which describes the fluild flow between 2 rectangular plates. It is complicated enough to be a classical example in chaotic theory for chaos observation and modelling. The Lorenz attractor obeys the following equations:

$$\begin{cases} \dfrac{\mathrm{d}x}{\mathrm{d}t} &= \sigma\left(y-x\right) \\ \dfrac{\mathrm{d}y}{\mathrm{d}t} &= x\left(\rho-z\right)-y \\ \dfrac{\mathrm{d}z}{\mathrm{d}t} &= xy-\beta z \end{cases} \tag{2.17}$$

where σ, ρ, β are system parameters related to the physics of the fluid of interest and some other physical constants such as Rayleigh number. Note that x, y, z are not spatial coordinates, they are thermodynamics quantities. Lorenz system is famous because it illustrates the *butterfly effect*. A slight change in initial condition could lead to a totally different evolution of the system. Figure 2.9 shows a Lorenz attractor with many different initial conditions, indicated by the dots, for $\sigma = 10$, $\rho = 28$, $\beta = 2.667$. We collected the state evolution in 1,000 steps, 1ms each, for 100 different initial conditions. Due to fading memory, any past states values that are longer than a specific number, K, should not have any effects on the current states. Thus, we split the data into chunks of K values from $t-K$ to $t-1$ as inputs and use them to predict the outputs which are the states value at time t for all $t > 0$ in the limit of the generated data. The data is then splitted into train, validation and test set.

K is varied among 1, 2, 3, and 5 time steps to investigate the memory effect in Lorenz system. We kept the 10 trajectories away from training process for testing purposes. The rest of them are splitted into training and validation set. A few unseen trajectories prediction result are shown in Figure 2.10. It shows clearly that the memory length is an important factor to ensure the correct model is learned. The four models in Figure 2.10 have the same architecture: 4 layers of standard LSTM cells, each has 20 neural units, 0.3 dropout regularization. They were all trained using Adam optimizer with initial learning rate 0.001 and all reach convergence after more than 120 epoches. The shorter memory length models take a bit longer to train

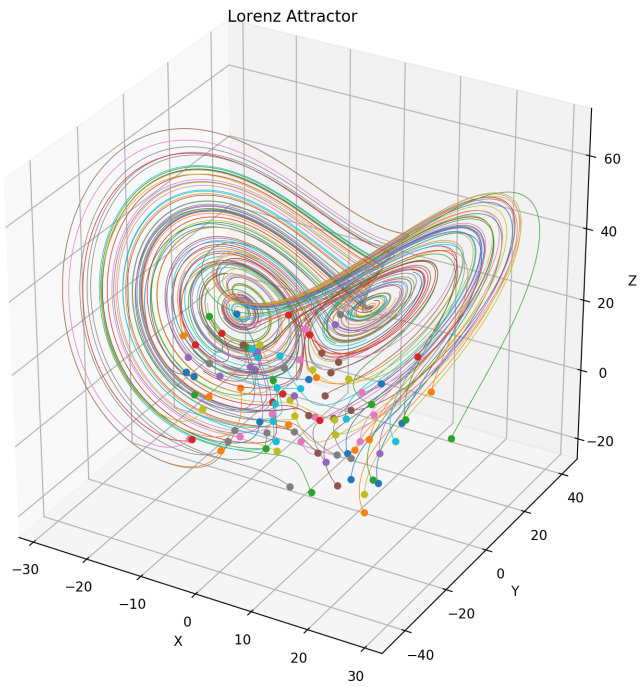


Figure 2.9: Lorenz attractor with different initial conditions (dots).

compared to the longer memory ones. However, when facing unseen data, the winning model is obvious. This experiment, conveniently, also shows that Lorenz system, though is chaotic, has short-term memory. Increasing the memory length in training doesn't improve the accuracy on test set, however, it helps training converge faster. Figure 2.11 shows convergence during training a memory length $K = 50$, while short memory length models in Figure 2.10 requires at least 120 epoches to acceptably converge, this long memory model only needs about 50 epoches. Figure 2.12 shows a comparison between ground truth and the model's prediction.

2.3.2 PAM2 channel simulation with output-feedback RNN (NARX-RNN)

In this section, we illustrate the training procedures of the RNN, with which the predictions can be made on the voltage waves arriving at the receiver of a high-speed channel using NARX-RNN. As demonstrated in Equation (2.4), the NARX-RNN does not have a hidden state explicitly defined in the model. The current output response is determined only using the current and past values of the input and the past values of the output. The

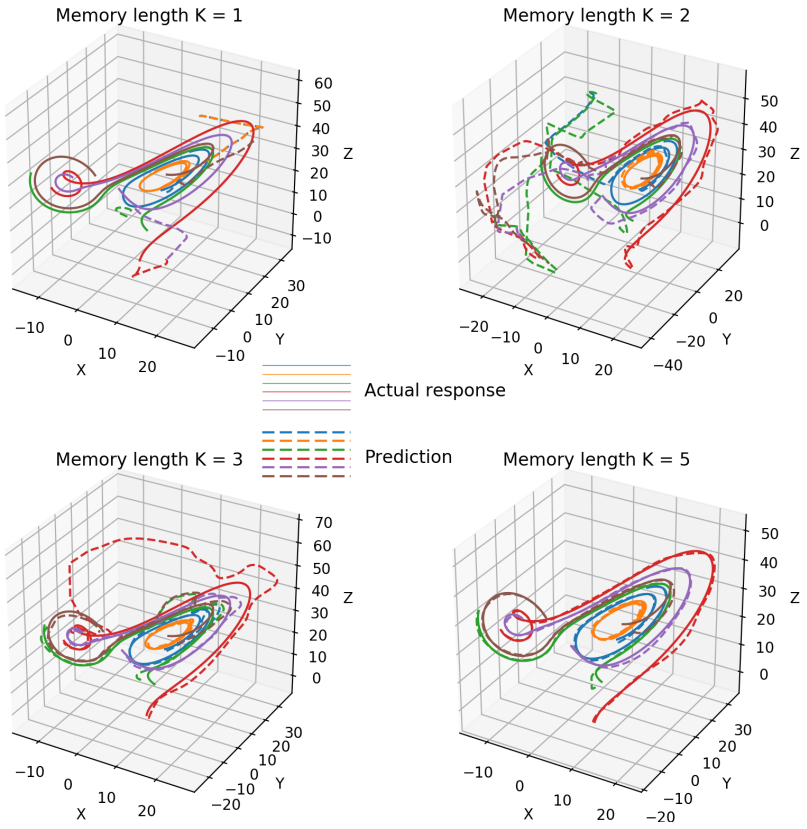


Figure 2.10: Prediction for Lorenz trajectories with different models trained on different memory lengths.

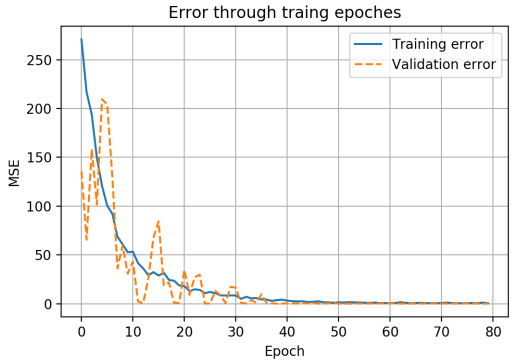


Figure 2.11: Training and validation error when training the Lorenz attractor.

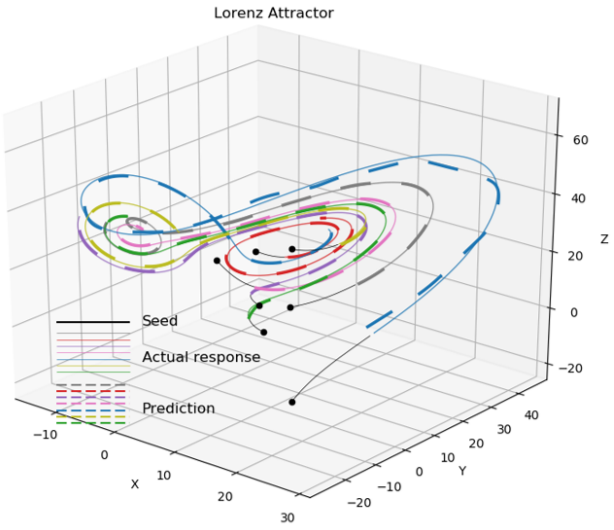


Figure 2.12: Prediction made on unseen trajectories by feeding a *seed* sequence of first 50 steps.

setup is shown in Figure 2.13: V_{TX0} is the output voltage of the transmitter (TX) when it is terminated with a 50 Ohm resistor; and V_{TX} and V_{RX} are voltages at the immediate output of TX and the input of RX in the presence of the channel. In this example, we use V_{TX} and V_{TX0} of the current time step and V_{RX} of the past to predict V_{RX} of the current time step.

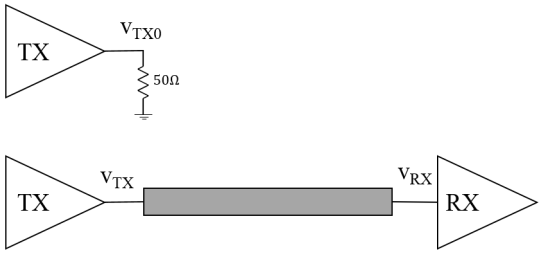


Figure 2.13: Simulation setup for data collection.

The data is normalized and segmented into sequences of length K. Sample sequences after normalization of all the signals of interest are depicted in Figure 2.14. This number K represents the memory dependency of the system. The larger the K is, the longer the memory the system keeps. A portion of the data (10%) is reserved for test. In this example, a stack of four LSTM cells of 20 hidden units is used. The optimization method used is Adam with 0.3 dropout regularization. Throughout our experiments, increasing K not only improves the convergence but also achieves higher accuracy. However, once K reaches the underlying memory length of the system under learn-

ing, a further increase does not offer better convergence nor higher accuracy. We use $K = 10$ in the following numerical experiments. The time steps for training is 11,000 and the model converges in about 48 epochs. Accurate predictions are achieved on unseen sequence as shown in Figure 2.15.

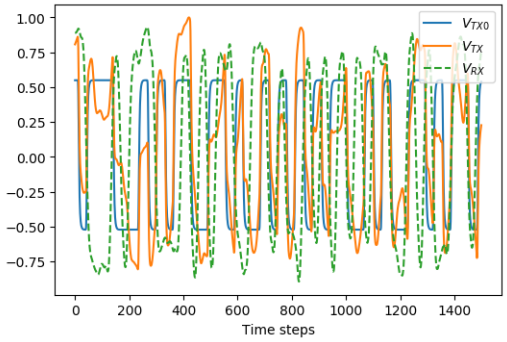


Figure 2.14: Training data collected with the setup shown in Figure 2.13.

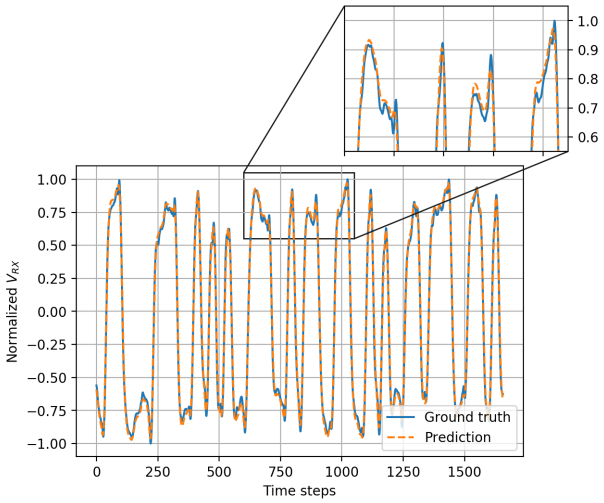


Figure 2.15: Predicted voltage at the receiver V_{RX} with a LSTM network.

Figure 2.16 and Figure 2.17 show the comparison between LSTM network and vanilla RNN in terms of their capability of handling the long-term memory. The same network architecture is adopted in this comparison including the number of layers, the layer width, and the regularization. It is shown that when the memory is relatively short with $K = 4$, the vanilla RNN cells fails to capture the signal evolution whereas the LSTM network makes quite accurate predictions. When the memory is sufficiently long, the vanilla RNN starts making comparably accurate predictions as the LSTM network does,

which is shown in Figure 2.17. From this comparison, it also reveals that training with Adam optimizer achieves better performance than the SGD optimizer regardless of the memory length.

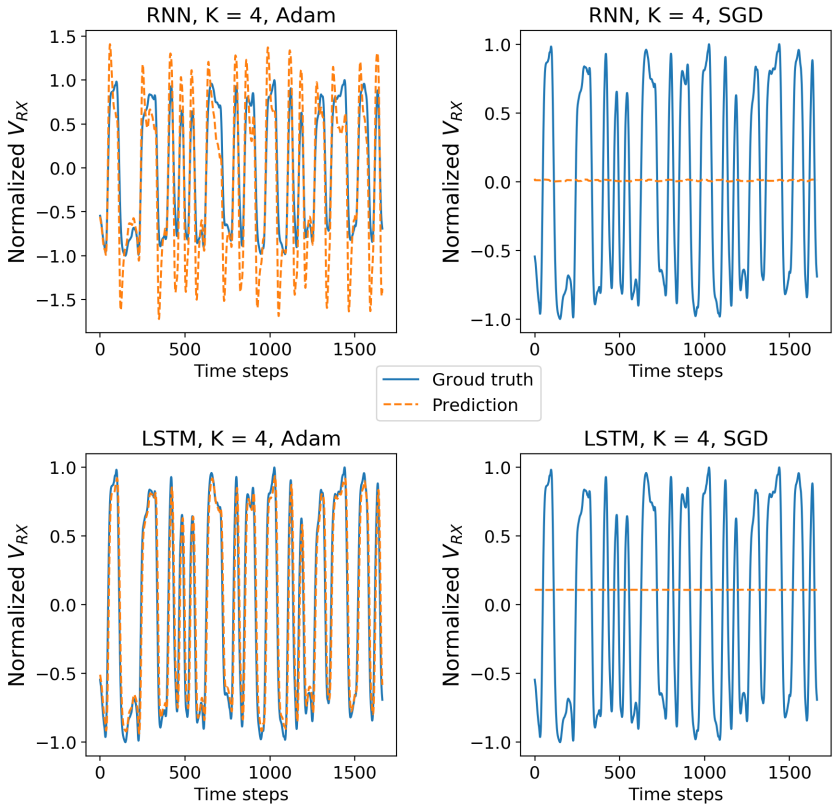


Figure 2.16: Comparison between vanilla RNN and LSTM network in handling relative short memory when the memory length K is chosen as 4.

It is worth mentioning that while using the LSTM and GRU networks, one needs to pay particular attention upon the selection of activation functions. For example, the gating signals f_t and i_t in Equation (2.12) controls the percentage of the memory passing through the gates, which ranges from 0 to 1. In this case, the activation function associated with f_t and i_t has to be the sigmoid function. Besides, the gating signal g_t in Equation (2.12) allows both addition and subtraction operations between the input and the forget gates and the hyperbolic tangent function is appropriate. As for a VRNN, the selection of activation functions is only based on the nonlinearities. As shown in Figure 2.18, using the hyperbolic tangent function as the activation

function in a vanilla RNN achieves more accurate predictions than that with ReLU.

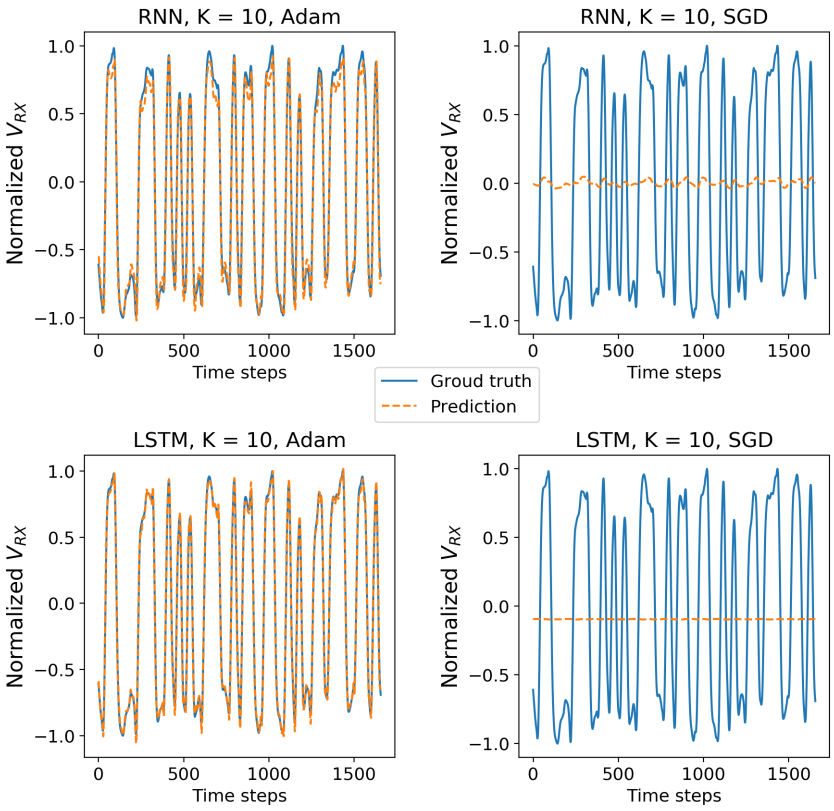


Figure 2.17: Comparison between vanilla RNN and LSTM network in handling sufficiently long memory when the memory length K is chosen as 10.

In addition, SGD optimizer does not work for the proposed RNN structure under the aforementioned settings for training. Adding momentum for SGD does not help the learning process either. However, Adam optimizer achieves accurate predictions. We also investigate RMSProp optimizer, which has been used for RNNs long before Adam is invented, to train the same architecture in terms of both VRNN and LSTM network. It is found that for short memory such as $K = 4$, using RMSProp optimizer does not achieve convergence; as the memory length K goes beyond 5, RMSProp optimizer performs as well as Adam. The result shown in Figure 2.19 confirms that for $K = 5$, networks trained by RMSProp make accurate predictions on the

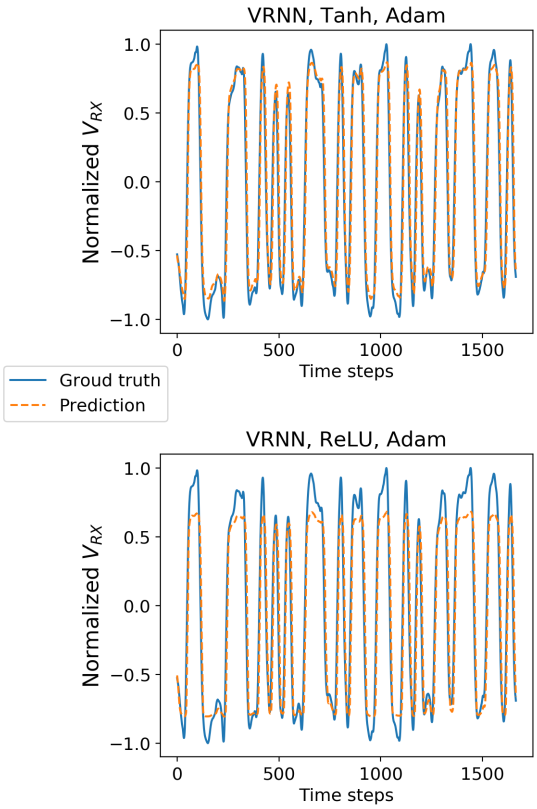


Figure 2.18: The impact from different types of activation functions on the prediction accuracy in a vanilla RNN.

output waveform. However, setting a high momentum to deploy adaptive learning rate degrades the performance of the network; as can be seen in Figure 2.19, the prediction accuracy becomes worse with momentum added in RMSProp.

2.3.3 PAM4 channel simulation with ERNN

The limitation of the output-feedback RNN used in the PAM2 example is that it strictly requires the output of the current time step before it can make predictions on one future time step, which can be seen from Equation (2.4). The neural network model, when being used in this way, cannot utilize batch inference. In this example, it is shown that using a deeper and wider network, an RNN-based model can be developed to utilize batch inference, which can dramatically reduce the run time for long transient simulation.

To prepare the training data, first, the transmitter output is measured

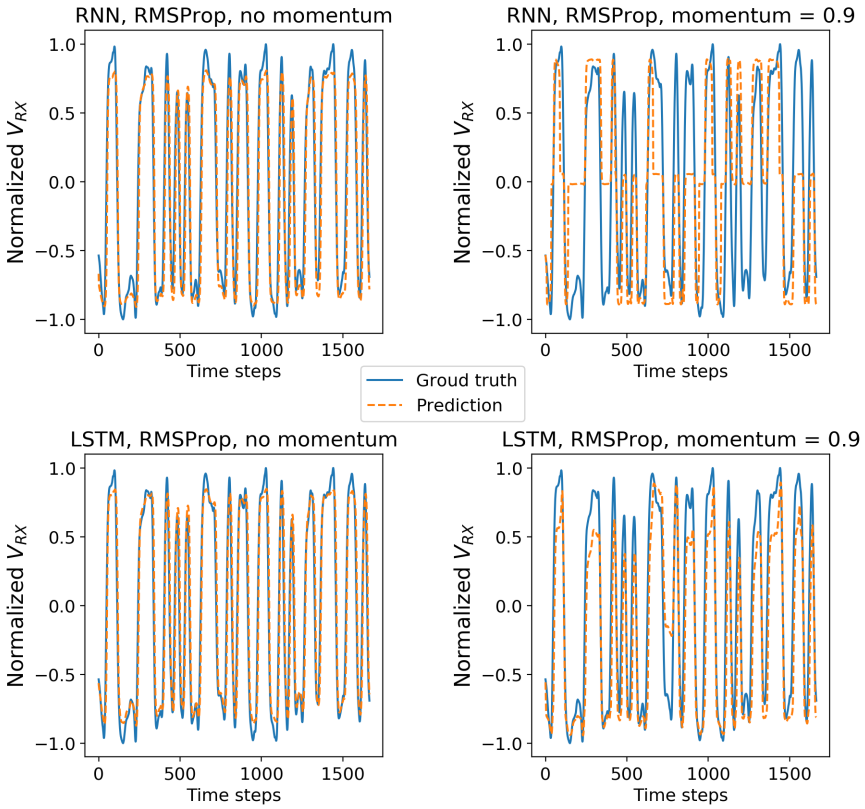


Figure 2.19: Performance of the same architecture using different RNN cells, trained by RMSProp when $K = 5$.

when it is opened, denoted as V_{TX0}. This signal is the Thevenin source to the combined "channel and receiver" system of interest. When the transmitter is connected to the channel and the receiver, the input to the channel from the transmitter V_{TX} and the input to the receiver after the channel V_{RX} are both collected for training purpose. Besides, the output voltage from the receiver V_{RO} is also captured and included in the training set. The setup for data collection is shown in Figure 2.20. The data in this example comes from a PAM4 transceiver circuit transmitting data at 28 Gbps. An LSTM network

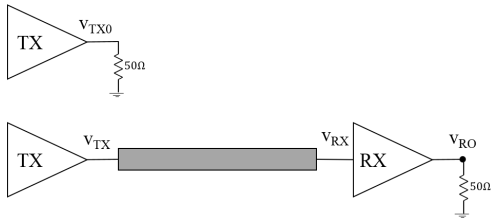


Figure 2.20: Setup to obtain training data for PAM4 example.

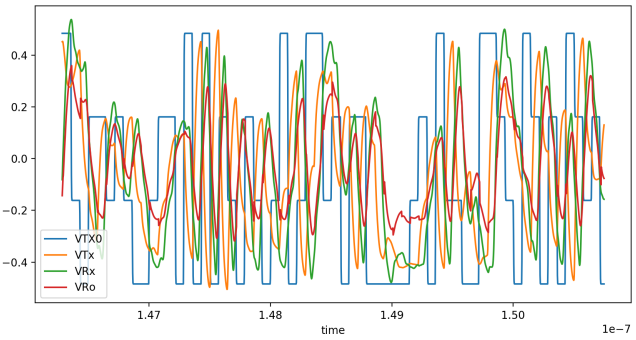


Figure 2.21: Voltages used to train ERNN in PAM4 example.

is trained on about $10,000$ time points of time domain response of as shown in Figure 2.21. A training waveform sample is shown in Figure 2.22.

We first investigate the impact from memory length on the training process. The memory length depends on not only the nonlinearity of the transmitter and receiver but also the delay of the channel. As for the training setup, Adam is used as the optimizer with initial learning rate of 0.001 and dropout regularization is fixed at 0.3. The LSTM network has six layers each with 30 hidden units. The memory length K is varied with everything else remaining the same. Figure 2.23 demonstrates the training performance under various memory lengths with the same network topology. By showing the results at different epochs, Figure 2.23 also reveals the fact that the learning ability of the LSTM network evolves as the training progresses. For example, at the 100^{th} epoch, the LSTM network learned the switching pattern of the waveforms; at the 1000^{th} epoch, the same network is able to make accurate predictions on V_{TX} in terms of both the pattern and the amplitude.

With the memory length chosen as $K = 50$ and at the 1000^{th} epoch, the predictions made with the LSTM network on V_{RO} are less accurate than those on V_{TX}, as shown in Figure 2.23. The reason that obtaining accurate predictions on V_{RO} is more challenging than that for V_{TX} is because the former requires a much better knowledge of the delay imposed by the channel. It seems the memory length set by $K = 50$ does not provide adequate data on the channel delay. After the memory length is increased to $K = 90$, the predictions on V_{RO} become much more accurate, as shown in 2.23c. A further increase of the memory length to $K = 100$ does not further improve the performance while demands more computation resources.

To further validate the model, we employ a PRBS much longer than the

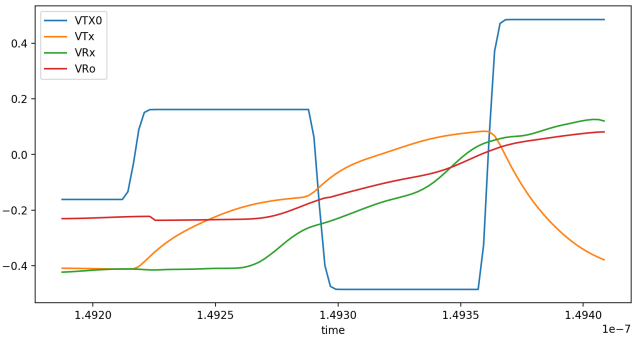
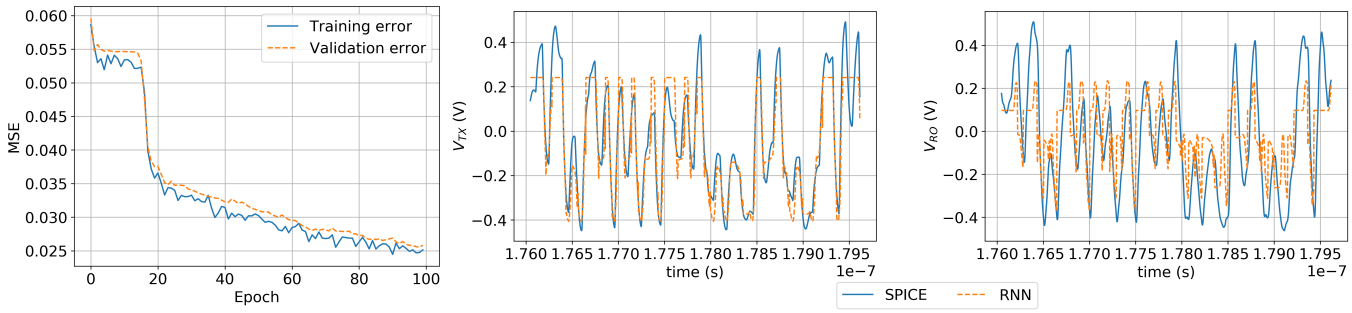


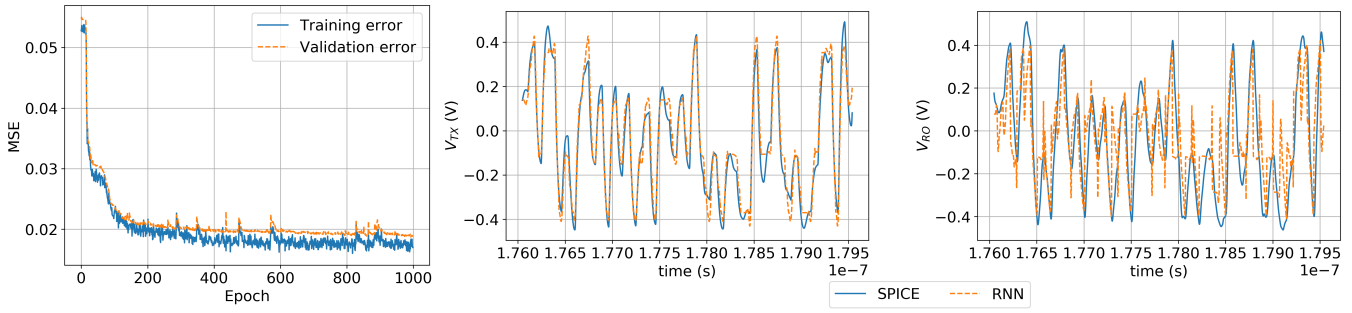
Figure 2.22: A training sample by windowing the training sequence with $K = 100$.

training one and generate eye diagrams. In Figure 2.24, it shows a very good agreement between the eye diagram generated from traditional SPICE-like simulation and the one from the proposed RNN-based model. For example, both eye diagrams point out that the optimal sampling point is about 14.662 μs.

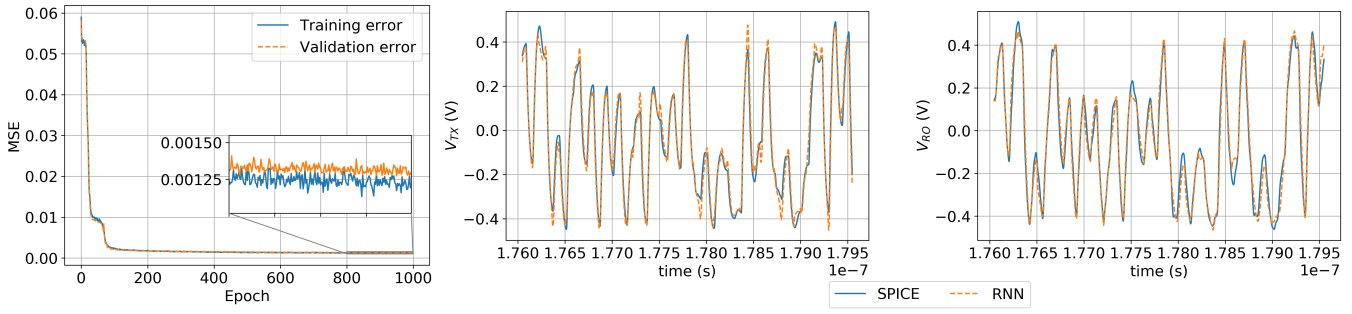
One limitation of the proposed method with RNN is the accumulation of numerical error. During the training process, TBPTT gives a noisy gradient information to the optimizer, which translates to the numerical error in the solution. This numerical error, though initially very small, gradually accumulates as the prediction goes on with the input sequence. Figure 2.25 shows the performance of the trained model in the previous section on a very long PRBS. Initially, the predicted results from the RNN model agree well with those obtained from the circuit simulation. However, as the prediction progresses, the numerical error due to TBPTT accumulates and degrades the performance of the RNN model. The accumulation of the numerical error is a well-known limitation of RNN trained by TBPTT, which at the same time leaves room for improvement in the future work on the proposed method with advanced techniques for sequence modeling such as attention mechanism [43]. However, it is worth noting that this accumulated error is naturally upper bounded by the complexity (also, partly by the truncated memory length) of the model. It can be seen that the performance of the RNN after hundreds of thousands of bits is still well acceptable.



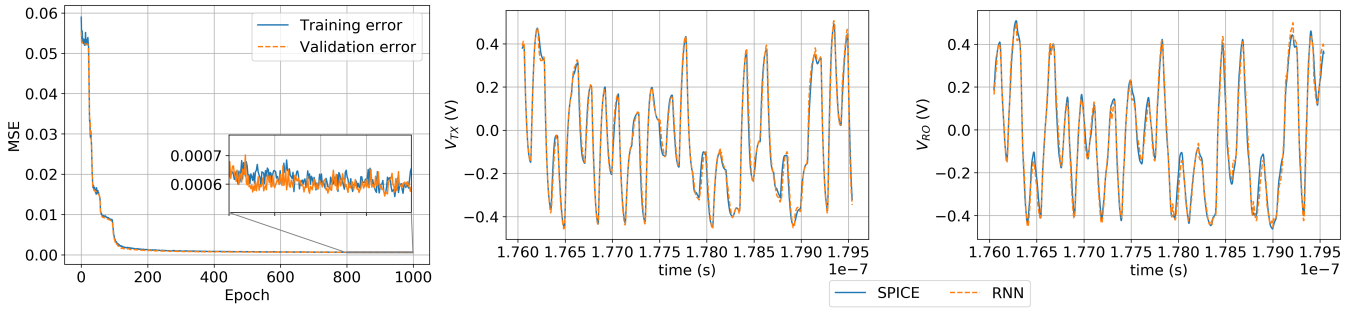
(a) When $K = 50$, trained in 100 epochs.



(b) When $K = 50$, trained in 1,000 epochs.

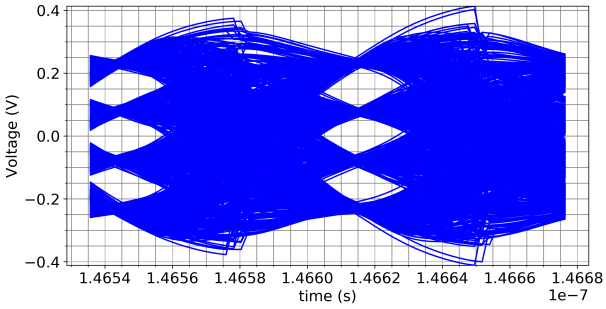


(c) When $K = 90$ trained in 1,000 epochs.

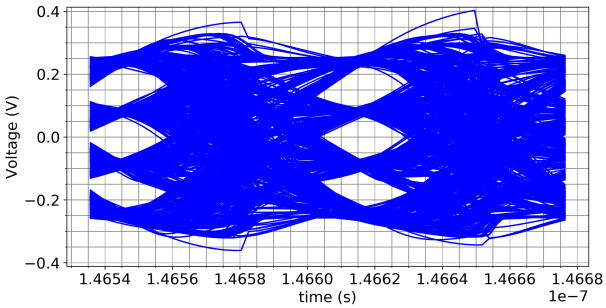


(d) When $K = 100$, trained in 1,000 epochs.

Figure 2.23: Training error (most left) and test performance of RNN model in PAM4 transceiver example.



(a) From SPICE



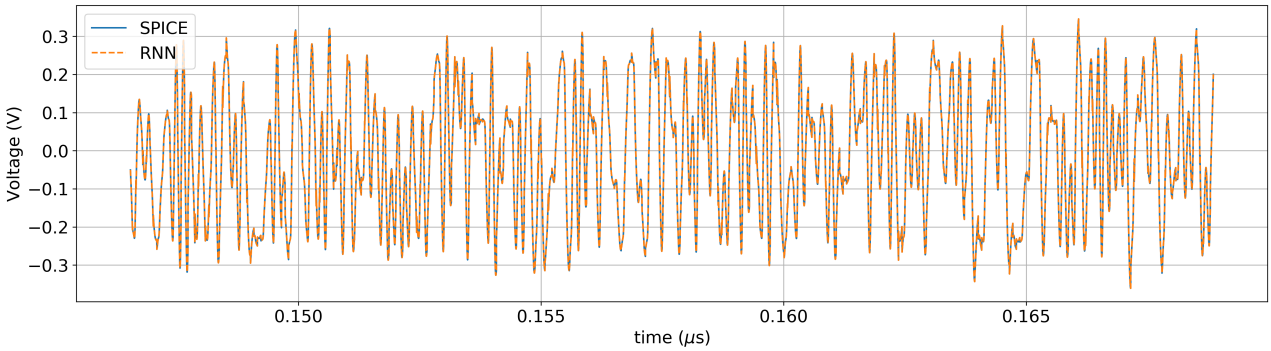
(b) From RNN

Figure 2.24: Eye diagram obtained in PAM4 transceiver example.

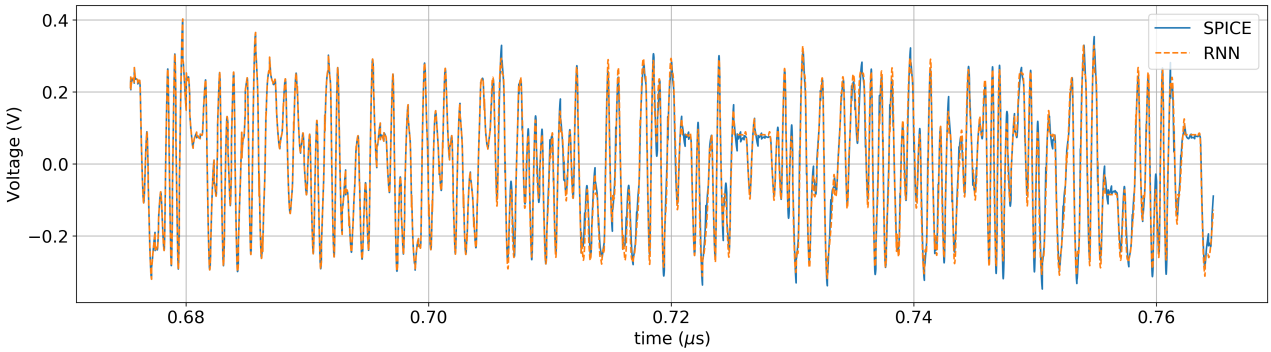
2.3.4 RX DFE circuit modeling with FNN and RNN combined

The example presented in this section is a 2-tap RX DFE circuit designed for a 32Gpbs link. The channel delay is about 1.7ns, as shown in Figure 2.26, there are 2 significant post-cursors that strongly contributes to intersymbol interference. The designed DFE is meant to cancel these 2 post-cursors. The channel pulse response and the equalized response are shown in Figure 2.26. The effect of DFE is reflected clearly on the equalized waveform which is cancelled out exactly at 2 most significant sampled post-cursor locations. The tap values are normalized so that they span from 0 to 1. Negative values of the taps will amplify the post-cursors instead of cancelling them, hence, are excluded from the study.

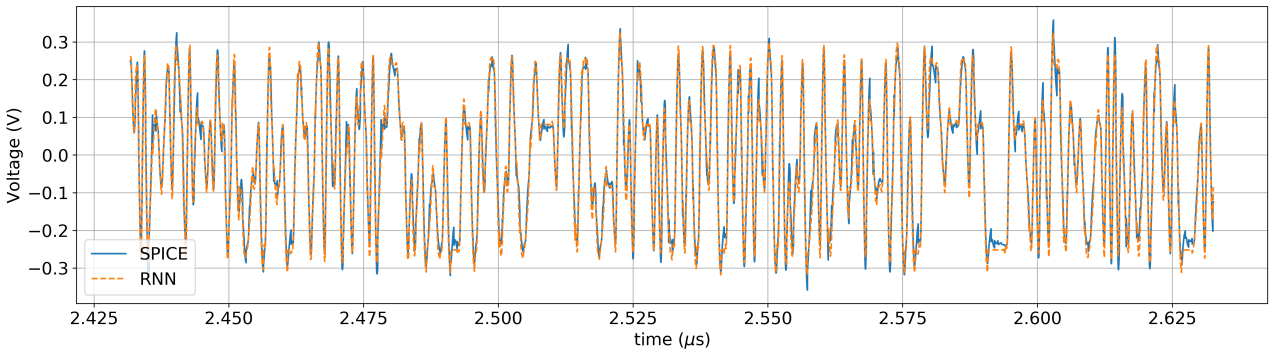
To prepare training data, many combinations of tap values are swept, the unequalized and equalized waveform is collected for each combination of tap values. In this example, we chose a 2-layer FNN which has 10 neurons and 20 neurons, respectively and a 6-layer, LSTM-cell RNN whose hidden state is in $\mathbb{R}^{30}$. Adam [32] is used for optimization with initial learning rate being 0.01. Also, RNN when trained was set to start out in *teacher-force* mode but



(a) Initially



(b) After tens of thousands of bits



(c) After hundreds of thousands of bits

Figure 2.25: Waveform comparison between SPICE simulation and RNN prediction on PAM4 example with another PRBS.

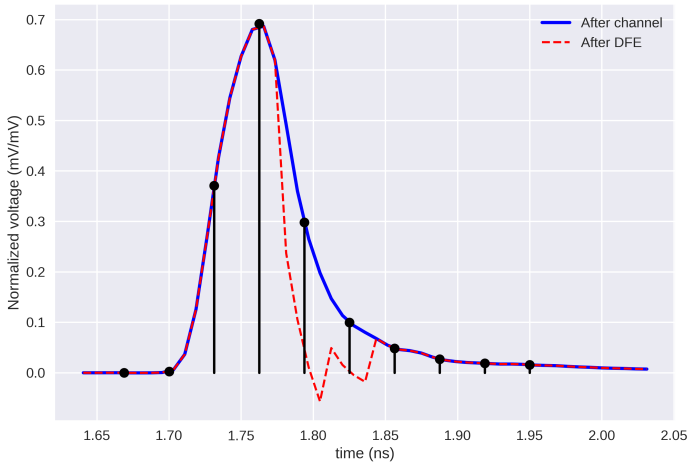


Figure 2.26: Single pulse response and DFE effect.

we used the scheduled training presented in Equation (2.16) to eventually switch the data fed to RNN during training to its own generated data. This will ensure the RNN to not overfit and have better generalization property.

After trained, the RNN is set to *read-out* mode and the model is tested with an unseen PRBS for various unseen tap values. The experiment result is shown in Figure 2.27, doted lines are results from a tradional SPICE-like simulator while dash lines are results from the FRNN model. Different colors are corresponding to different combinations of tap values. As shown, the predictions from FRNN model match the transient simulation very well, more noticeably, the FRNN model can be parameterized by tap values.

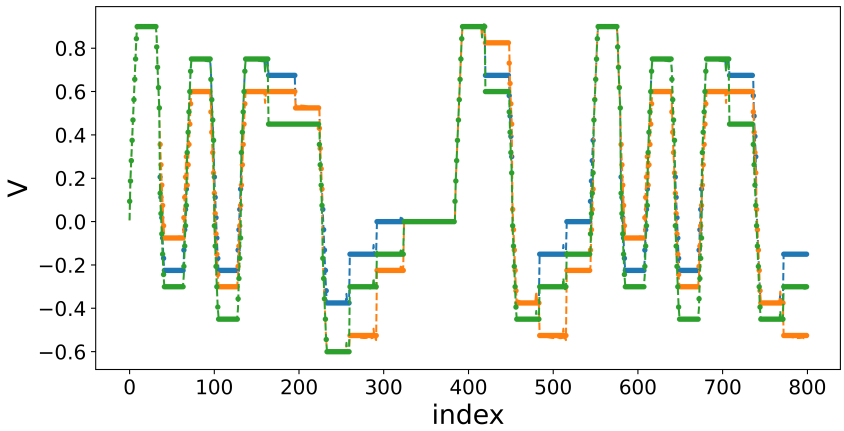


Figure 2.27: Performance of FRNN on unseen PRBS for unseen tap values.

Chapter 3

VOLTERRA MODELS FOR WEAKLY NONLINEAR CIRCUITS

3.1 Volterra Series

A general nonlinear dynamical system can be described by:

$$\begin{cases} \dot{x}(t) &= f(x(t), u(t)) \\ y(t) &= g(x(t), u(t)) \end{cases} \tag{3.1}$$

where $u(t) \in \mathbb{R}^p$, $x(t) \in \mathbb{R}^d$, $y(t) \in \mathbb{R}^q$ for an p-input, q-output system with d latent states, $f(\cdot)$ and $g(\cdot)$ are the nonlinear mapping from the input to the states amd between the states to the output. If they are affine mappings, Equation (3.1) becomes the well-known LTI statespace. For simplicity, we will consider the single-input, single-output (SISO) case ($p = d = q = 1$) in the following.

Given a weakly non-linear time-invariant (NLTI) system, also known as the Volterra system, memory effects can be well approximated by a truncated Volterra series [44]:

$$\begin{aligned} y(t) &= \sum_{n=1}^{N} y_n(t) \\ y_n(t) &= \frac{1}{n!} \int_{\mathbb{R}^n} h_n(\tau_1, \tau_2, ..., \tau_n) \prod_{i=1}^{n} u(t - \tau_i) \, \mathrm{d}\tau_i \end{aligned} \tag{3.2}$$

where $h_n(\tau_1, \tau_2, ..., \tau_n)$ is the n^{th} order Volterra kernel (VK). It can also be called multi-dimensional impulse response. An NLTI system is considered weakly non-linear if it possesses *fading memory* property. That is, the present output does not depend on the complete history [45]. Figure 3.1 shows a systematic decomposition of a weakly NLTI system superimposing contributions of VK's as described in Equation (3.2). Rigorous foundations and details about Volterra series can be found in [46, 47]. Volterra series can be

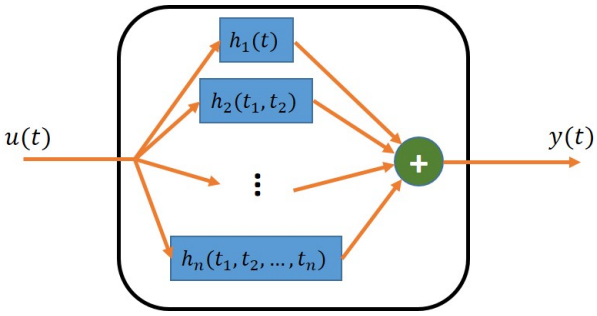


Figure 3.1: Volterra system decomposition.

thought of as the generalization from one to multi-dimensional representation of a dynamical system. It is associated with the class of weakly non-linear systems. The advantage of using Volterra series for modeling is that Fourier theory can be applied with a minor modification to reflect high-dimension functionals involved. Time- and frequency-domain responses are related by multi-dimensional Fourier transform. There have been efforts to estimate Volterra kernels from a truncated Volterra series using analytical derivations, time-domain measurements and even spectrum measurements [48–51]. Unlike the impulse response of an LTI system, a nonlinear system can be represented by different sets of VKs [47]. However, the symmetric VKs, which can be computed from all possible variable permutations of the same order non-symmetric kernels [47], are shown to be unique. The term Volterra kernels used in this work refers to the symmetric kernels.

A Volterra time-domain kernel can be transformed to its frequency-domain counter-part through a multi-dimensional Fourier transformation:

$$H_n(\omega_1, \omega_2, ..., \omega_n) = \int_{\mathbb{R}^n} h_n(\tau_1, \tau_2, ..., \tau_n) \exp\left(-j\sum_{k=1}^{n} \omega_k \tau_k\right) \prod_{i=1}^{n} \mathrm{d}\tau_i \tag{3.3}$$

Generally, Equation (3.3) is not straightforward to evaluate because it involves a high-dimensional integration. In practice, during the system identification, certain assumptions about the system of interest can be made. For example, there are no discontinuity jumps in the system response, or, the nonlinearity is polynomial. This results in some special forms which VKs can take, the evaluation of Equation (3.3) can make use of these special forms to simplify the calculation. Inversely, time-domain kernels can be recovered

using a multi-dimensional inverse Fourier transform:

$$h_n(t_1, t_2, ..., t_n) = \int_{\mathbb{R}^n} H_n(\omega_1, \omega_2, ..., \omega_n) \exp\left(j\sum_{k=1}^{n} \omega_k t_k\right) \prod_{i=1}^{n} \mathrm{d}\omega_i \quad (3.4)$$

As mentioned above, depending on the form of $f(\cdot)$ and $g(\cdot)$, Equation (3.1) can be reduced to a particular expression, the corresponding Volterra representation can be derived. An example of such specialization is the Wiener system. A Wiener model describes a nonlinear system by introducing a polynomial nonlinearity following an LTI system. Figure 3.2 shows a block diagram for 2^{nd} order Wiener system.

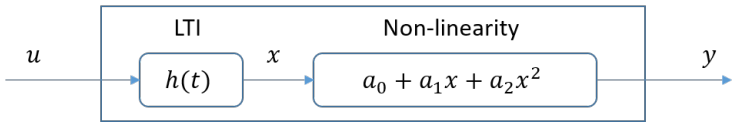


Figure 3.2: A 2^{nd} order Wiener system

A Wiener system has the n^{th} order Volterra kernel, which is *separable* [47], given by:

$$h_n(\tau_1, \tau_2, \ldots, \tau_n) = a_n \prod_{i=1}^{n} h_1(\tau_i) \quad (3.5)$$

A visualized example of 2nd order VK $H_2(f_1, f_2)$ which is the Fourier transform of $h_2(\tau_1, \tau_2)$ is shown in Figure 3.3.

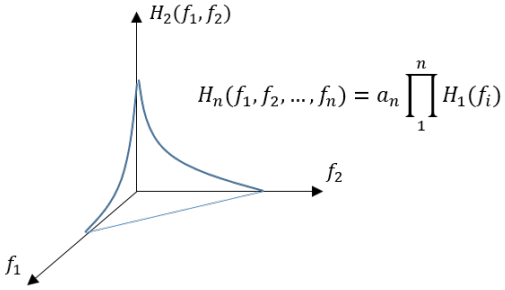


Figure 3.3: Second order Volterra kernel of a Wiener system.

For a Wiener system, Equation (3.1) simplifies to:

$$\begin{cases} \dot{x}(t) &= A_0 x(t) + B_0 u(t) \\ y(t) &= \sum_{i=0}^{N} a_i x^i(t) \end{cases} \quad (3.6)$$

with A_0, B_0 being size-appropriate state matrices.

If the VKs values are approximated by a sum of delayed delta pulses, i.e. sampled directly at discrete points τ_1, τ_2, ..., τ_n, Equation (3.2) is a non-parametric model. Whilst projecting VKs onto an orthogonal basis such as the Laguerre basis as presented in the coming section will result in a parametric model. The former is simple to model and straightforward to extract but requires a large number of coefficients to be determined while the latter requires more analytical work to be carried out before the extraction process but much fewer coefficients are needed. Also, by using a parametric model, additional assumption about the kernels, hence the system, is made. Laguerre basis is shown to be effective for nonlinear circuits in the scope of this thesis [52]. Therefore it is the choice of parametrization for the VKs in this thesis.

3.2 Time-domain non-parametric kernel estimation

In discrete time, indexed by k, Equation (3.2) becomes:

$$y_n(k) = \sum_{\tau_n=0}^{k} ... \sum_{\tau_1=0}^{k} h_n\left(\tau_1, \tau_2, ..., \tau_n\right) \prod_{i=1}^{n} Tu\left(k - \tau_i\right) \tag{3.7}$$

where T is the sampling interval when converting the continuous into a discrete time system, $t = kT$. $h_n\left(\tau_1, \tau_2, ..., \tau_n\right)$ is now simply the function evaluation of the continuous VK at discrete time indexed by $\tau_1, \tau_2, ..., \tau_n$.

Without loss of generality, let us walk through the details of extracting a k-time step VKs of a second order Volterra system. Explicitly writing out the 1^{st} and 2^{nd} term will allow us to see how non-parametric kernel estimation is feasible:

$$y_1(k) = T\sum_{\tau_1=0}^{k} h_1\left(\tau_1\right) u\left(k - \tau_1\right) \tag{3.8a}$$

$$y_2(k) = T^2\sum_{\tau_2=0}^{k}\sum_{\tau_1=0}^{k} h_2\left(\tau_1, \tau_2\right) u\left(k - \tau_1\right) u\left(k - \tau_2\right) \tag{3.8b}$$

It can be seen that $y_n\left(k\right)$ is linear in terms of $h_n\left(\tau_1, \tau_2, ..., \tau_n\right)$ with coefficient $T^n \prod_{i=1}^{n} u\left(k - \tau_i\right)$. Constructing a set of Equation (3.8)a for $k = 0, 1 \text{ and } 2$ we have:

$$\underbrace{\begin{bmatrix} y_1(0) \\ y_1(1) \\ y_1(2) \end{bmatrix}}_{\bar{y_1}} = T \underbrace{\begin{bmatrix} u(0) & & \\ u(1) & u(0) & \\ u(2) & u(1) & u(0) \end{bmatrix}}_{\mathbb{U}_1} \underbrace{\begin{bmatrix} h_1(0) \\ h_1(1) \\ h_1(2) \end{bmatrix}}_{\bar{h_1}} \tag{3.9}$$

Similarly, let

$$\bar{h_2}(k) = \begin{bmatrix} h_2(0,0) \\ h_2(0,1) \\ \vdots \\ h_2(0,k) \\ \hline h_2(1,0) \\ h_2(1,1) \\ \vdots \\ h_2(1,k) \\ \hline \\ \vdots \\ \\ \hline h_2(k,0) \\ h_2(k,1) \\ \vdots \\ h_2(k,k) \end{bmatrix} \tag{3.10a}$$

and

$$u_{0k} = \begin{bmatrix} u(k)\,u(k) & u(k)\,u(k-1) & u(k)\,u(k-2) & \cdots & u(0)\,u(0) \end{bmatrix} \tag{3.10b}$$

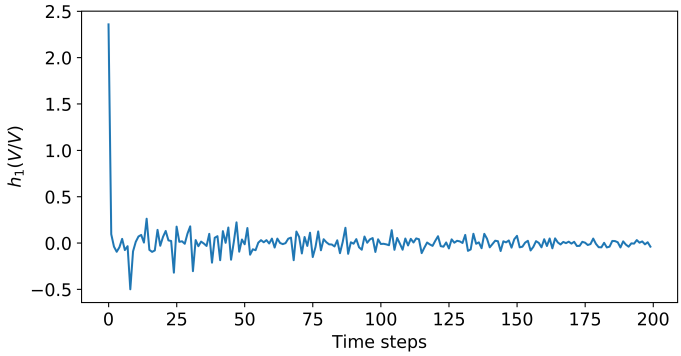
$$\mathbb{U}_2(k) = \begin{bmatrix} u_{0k} & u_{1k} & \cdots & u_{kk} \end{bmatrix} \tag{3.10c}$$

Equation (3.8)b can be written in the matrix form as:

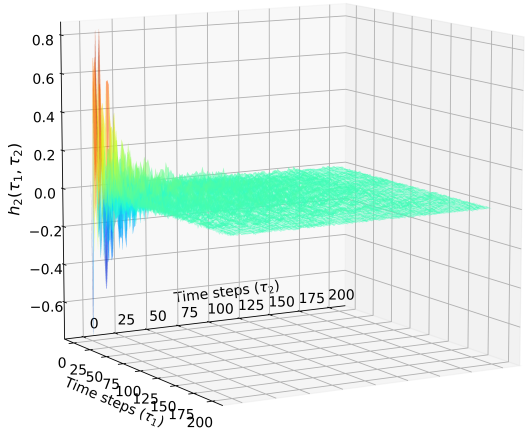
$$\bar{y_2}(k) = T^2 \mathbb{U}_2(k)\, \bar{h_2}(k) \tag{3.11}$$

in which $\mathbb{U}_2(k)$ is a very long row vector ($\mathbb{R}^{t^2 \times 1}$) and $H_2(k)$ is a very long column vector ($\mathbb{R}^{1 \times t^2}$). Finally,

$$\bar{y} = \bar{y_1} + \bar{y_2} = \begin{bmatrix} T\mathbb{U}_1 & T^2\mathbb{U}_2 \end{bmatrix} \begin{bmatrix} \bar{h_1} \\ \bar{h_2} \end{bmatrix} \tag{3.12}$$



(a) First order VK, $h_1(\tau)$.



(b) Second order VK, $h_2(\tau_1, \tau_2)$.

Figure 3.4: Non-parametric VKs extracted from CLTE circuit slightly driven nonlinear.

Note that the number of columns of $\mathbb{U}_2$, a.k.a the number of unknown in the 2^{nd} order kernel, or length of H_2, grows with t^2. If the training data is just a long sequence of input/output, Equation (3.12) is an under-determined system. Whilst there are enough multiple short input/output sequences, Equation (3.12) will be an over-determined system. Finding a minimum-norm least square in the former and a least-square solution in the latter case will ensure a unique solution to Equation (3.12). It is important to note that in this approach, the size of the kernels grows with the available data.

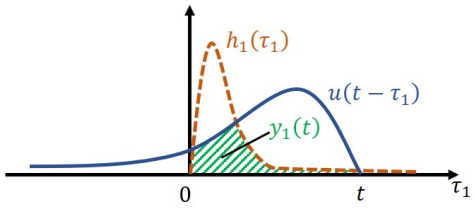
Figure 3.4 shows the first and second order VK extracted from an active continuous-time linear equalization (CTLE) circuit. Though CTLE was supposed to be linear, it was implemented with active devices, a slight increase in the voltage swing would purposely drive it into nonlinear region. The input and output voltage waveform were collected to construct a system similar to

Equation (3.12) up to order 3.

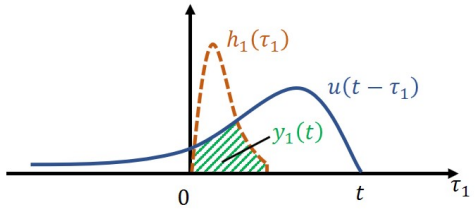
As can be seen in Figure 3.4, the kernel values die out quickly. This is practically reasonable for any real-world devices, the memory effect should be limitted to a few time steps back into the past. It is, therefore, more computationally suitable to truncate the impulse responses in Equation (3.7) to M time steps only. The n^{th} order response of a Volterra system now reads:

$$y_n(k) = \sum_{\tau_n=0}^{M} ... \sum_{\tau_1=0}^{M} h_n(\tau_1, \tau_2, ..., \tau_n) \prod_{i=1}^{n} Tu(k - \tau_i) \tag{3.13}$$

Equation (3.7) implements an inifite impulse response (IIR) alike system while Equation (3.13) implements a finite impulse response (FIR) alike system. The two are illustrated in Figure 3.5. Thanks to the fading memory property, the FIR alike implementation can be used without much of trading off accuracy for computational expense.



(a) IIR-like response by $h_1(\tau)$.



(b) FIR-like response by $h_1(\tau)$.

Figure 3.5: IIR versus FIR non-parametric first order Volterra impulse response.

3.3 Time-domain parametric kernel estimation

3.3.1 Volterra-Laguerre expansion

Non-parameteric modeling is straightforward as the model coefficients come directly from the data. However, these extracted kernels are not orthogonal. Most of the time, the order of the system of interest is unknown, the best can be done in practice is to iteratively increase the order until the fitting error, or some other figures of merit indicating the quality of the model, falls below a threshold. Without orthogonality, the identification process will have to solve for low order kernels all over again. Having a model in which the kernels are orthogonal by design will allow adaptive idenfication, elimilate the need to resolve the whole kernel set everytime the order is changed. In this section, we will see how the development of the Volterra model is carried out. Let us first discuss about functional orthogonality.

A set of real-value functions $\ell_i(t)$ is called orthonormal (loosely called orthogonal) over the interval (a, b) if:

$$\int_a^b \ell_i(t)\,\ell_j(t) = \delta_{ij}(t) = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases} \tag{3.14}$$

A function $f(t)$ is said to be expandable in terms of $\ell(t)$'s over the interval (a, b) when:

$$f(t) = \sum_{i=0}^{R} \theta_i \ell_i(t) \tag{3.15}$$

The expansion coefficients can then be given by:

$$\theta_i = \int_a^b f(t)\,\ell_i(t)\,\mathrm{d}t$$

Laguerre polynomials are naturally orthogonal and a complete set over $\mathcal{L}^2$ space. In addition to that, they are causal by definition. Expanding the Volterra kernels in terms of Laguerre polynomials is advantageous. They are, hence, chosen to be the projecting basis in this section. The Volterra-Laguerre (VL) expansion and its detailed implementation are discussed next.

For $t \geq 0$, let

$$\mathtt{L}_r(t) = \frac{1}{r!} \mathrm{e}^t \frac{\mathrm{d}^r}{\mathrm{d}t^r} \left(t^r \mathrm{e}^{-t} \right) = \sum_{i=0}^{r} \frac{(-1)^r}{r!} \begin{pmatrix} r \\ i \end{pmatrix} t^i \tag{3.16}$$

be the Rodrigues representation of the Laguerre polynomials, $\begin{pmatrix} r \\ i \end{pmatrix}$ is a binomial coefficient. Laguerre polynomials can also be computed recursively:

$$\begin{array}{rcl} \mathtt{L}_0(t) & = & 1 \\ \mathtt{L}_1(t) & = & 1 - t \\ \mathtt{L}_r(t) & = & \dfrac{2r - 1 - t}{r} L_{r-1}(t) - \dfrac{r-1}{r} L_{r-2}(t) \\ & & r = 2, 3, 4, \ldots \end{array} \tag{3.17}$$

The continuous time domain r^{th} Laguerre basis, $\ell_r(t)$, $r = 0, 1, 2...$, is built on top of these Laguerre polynomials with a decaying factor representing fading memory effect [53]:

$$l_r(t) = \sqrt{\sigma} \mathtt{L}_r(t) \, \mathrm{e}^{-\sigma t} = \sqrt{\sigma} \sum_{i=0}^{r} \frac{(-1)^i \, r! 2^{r-i}}{i! \left[(r-i)! \right]^2} (\sigma t)^{r-i} \, \mathrm{e}^{-\frac{\sigma}{2} t} \tag{3.18}$$

where $\sigma < 1$ is known as the time scale factor of the Laguerre basis, in the frequency domain, $-\dfrac{\sigma}{2}$ is the pole of the Laguerre basis .

Thorough details about the developement of continuous-time Laguerre functions in both time and frequency domain can be found in [44]. The algebraic form of the discrete Laguerre functions (DLF) is:

$$l_r(k) = \alpha^{\frac{k-r}{2}} (1-\alpha)^{\frac{1}{2}} \sum_{i=0}^{r} (-1)^i \begin{pmatrix} k \\ i \end{pmatrix} \begin{pmatrix} r \\ i \end{pmatrix} \alpha^{r-i} (1-\alpha)^i \tag{3.19}$$

where [54]:

$$a = \mathrm{e}^{-\frac{\sigma}{2} T} \tag{3.20}$$

$$a^2 = \alpha \tag{3.21}$$

Figure 3.6 shows the first 5 Laguerre functions given by Equation (3.19).

Before moving to applying the Laguerre expansion to the Volterra framework, let us now look at the evaluation of DLFs in a different perspective:

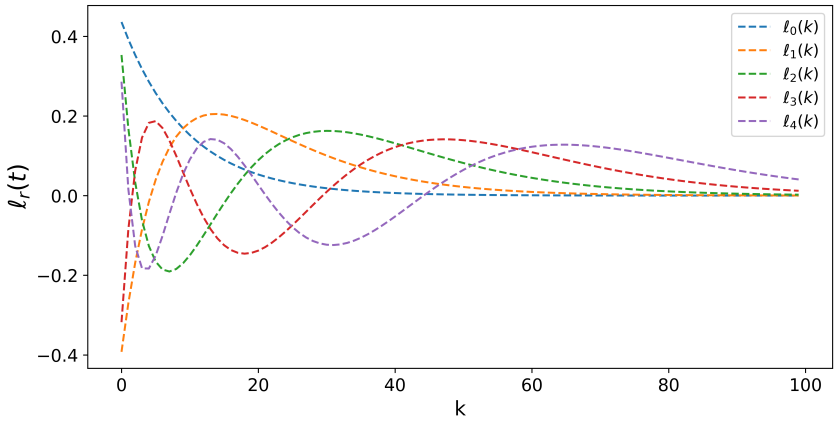


Figure 3.6: First few Laguerre functions.

the time evolution of Laguerre functions is obtained through a discrete linear system. The z-transform of DLFs is given as:

$$\begin{aligned} \Gamma_0(z) &= \frac{\sqrt{1-a^2}}{1-az^{-1}} \\ \Gamma_1(z) &= \Gamma_0(z)\frac{z^{-1}-a}{1-az^{-1}} \\ &\vdots \\ \Gamma_r(z) &= \Gamma_0(z)\left(\frac{z^{-1}-a}{1-az^{-1}}\right)^r \end{aligned} \tag{3.22}$$

or

$$\begin{aligned} \Gamma_0(z) &= \frac{\sqrt{1-a^2}}{1-az^{-1}} \\ \Gamma_r(z) &= \frac{z^{-1}-a}{1-az^{-1}}\Gamma_{r-1}(z) \quad r=1,2,3,\ldots \end{aligned} \tag{3.23}$$

with $|a|<1$ be the pole of the discrete time Laguerre network.

Let $R+1$ DLFs form a vector $L_R(k) = \begin{bmatrix} l_0(k) & l_1(k) & \cdots & l_R(k) \end{bmatrix}^T \in \mathbb{R}^{R+1}$, it sastifies the following difference equation [55]:

$$L_R(k+1) = A_R L_R(k) \tag{3.24}$$

with the initial condition:

$$L_R(0) = \sqrt{\beta} \begin{bmatrix} 1 \\ -a \\ a^2 \\ \vdots \\ (-1)^R a^R \end{bmatrix} \qquad \beta = 1 - a^2 \tag{3.25}$$

which was derived from Equation (3.19) by letting $k = 0$ for each DLF, and the transition matrix:

$$A_R = \begin{bmatrix} a & & & & & \\ \beta & a & & & & \\ -a\beta & \beta & a & & & \\ a^2\beta & \ddots & \beta & a & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ (-1)^{R-1} a^{R-1}\beta & \cdots & a^2\beta & -a\beta & \beta & a \end{bmatrix} \in \mathbb{R}^{(R+1)\times(R+1)} \tag{3.26}$$

If the Volterra kernels are projected to the space spanned by the first $R+1$ DLFs, we have:

$$h_n(\tau_1, \tau_2, \ldots, \tau_n) = \sum_{r_n=0}^{R} \cdots \sum_{r_1=0}^{R} \theta_{r_1,r_2,\ldots,r_n} \prod_{i=0}^{n} \ell_r(\tau_i) \tag{3.27}$$

where $\theta_{r_1,r_2,\ldots,r_n}$ is the Laguerre coefficients. It can be arranged into an n-dimensional hypercube length $R+1$ for better correlation to the original Volterra kernels.

Let, at time step k,

$$v_r(k) = \sum_{\tau=0}^{M} \ell_r(\tau)\, u(k-\tau) \tag{3.28}$$

be the response to the input $u(k)$ through the r^{th} Laguerre function, a linear combination of their products gives rise to the output. Equation (3.2), in discrete time, becomes [52]:

$$
\begin{aligned}
y(k) &= \sum_{n=0}^{N} y_n(k) \\
y_n(k) &= \sum_{r_n=0}^{R} \cdots \sum_{r_1=0}^{R} \theta_{r_1,r_2,\ldots,r_n} \prod_{i=0}^{n} v_r(k)
\end{aligned}
\tag{3.29}
$$

Essentially, one can find the final output by doing the following steps:

1. Generate the Laguerre responses, v_r, according to Equation (3.28). Figure 3.7 illustrates this step, the input u is passed through a bank of Laguerre filters, which result in v_r's.

2. Generate a product of any n Laguerre responses to generate one of the n^{th} order Laguerre responses.

3. Take the linear combination of all possible n^{th} order Laguerre responses to form the n^{th} order Volterra response, y_n. Once all n^{th} order Volterra responses are avalable, simply take the sum of them to obtain the final output.

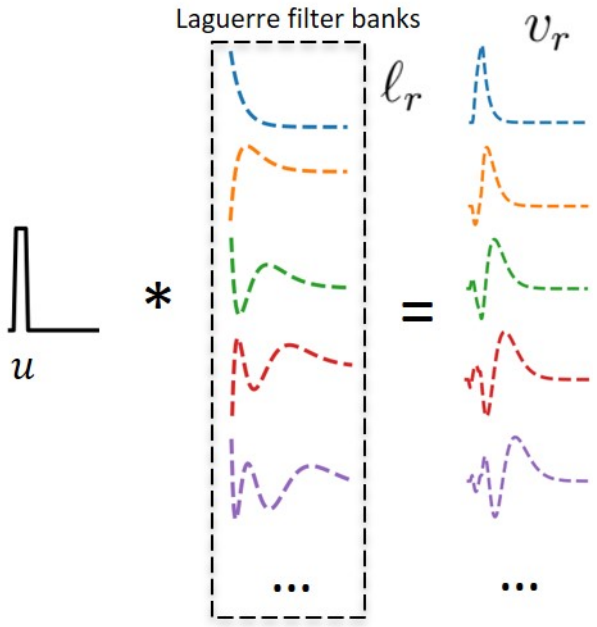


Figure 3.7: Generation of Laguerre responses.

Once again, we can see that y_n is nonlinear in v_r, consequently, in u, but is linear in $\theta_{r_1,r_2,\ldots,r_n}$. These Laguerre coefficients, therefore, can be extracted by

setting up a linear system. In fact, the same implementation in the previous section can be used to obtain the solution just by replacing u with v.

The dynamical point of view of the DLFs is important because it enables a one-step-back ability. To calculate the response at any time point, only the most recent time step information is needed. This is particularly important for this approach to be incorporated into existing solvers.

Figure 3.8 shows a comparison of the first order Volterra kernel extracted from the CTLE circuit using non-parametric and Laguerre discussed so far. It is quite obvious that Laguerre expansion based kernel is much smoother and flats out much quicker than non-parametric one.

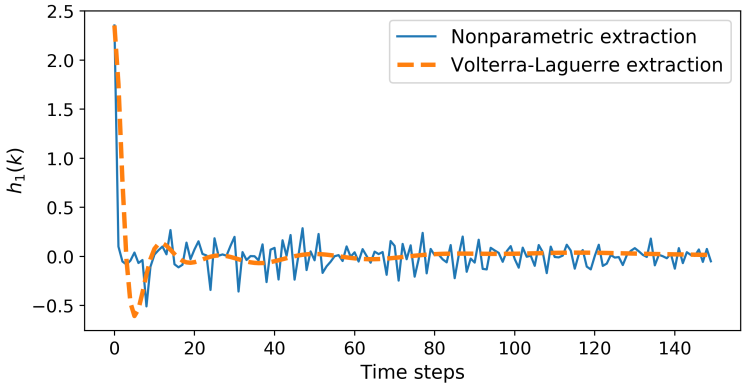


Figure 3.8: Extracted first order Volterra kernel: non-parametric vs. parametric.

In the next section, more examples will be presented to show the effectiveness of the proposed approach. Comparing to IBIS, the presented approach requires less data for training yet does not compromise its accuracy.

3.3.2 Parameterization of Volterra-Laguerre models

Using parametric VL models gives another advantage: the ability to parametrize the models. In the previous chapter, it was shown how the dynamical behavior of an RNN model can be adjusted using the initial hidden state which is mapped from the control inputs via an FNN. Figure 3.9 shows the signals flow in a paramterized model. In the previous chapter, f was represented by the FNN, g_ζ was the RNN while the latent variable z was the initial hidden state of the RNN. Using this parameterization model, one can always

parametrize a determinstic model to model stochastic effects caused by the control variables ζ without the knowledge of stochastic or statistical modeling

In this chapter, the dynamical response of a system is modeled with the VL representation. The models are actually just a handful of Laguerre coefficients. The parametrization, hence, can be done via an interpolation between ζ and $\theta_{r_1,r_2,\ldots,r_n}$'s. In the example shown in the next section, Gaussian Process which is presented in the next chapter was used as the interpolant. However, any interpolation method can be used to represent the mapping between ζ and the Laguerre coefficients.

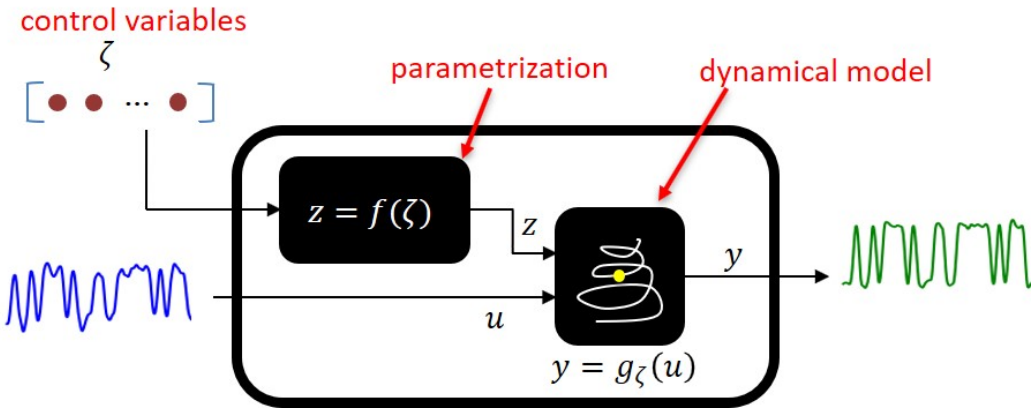


Figure 3.9: Parametrized nonlinear dynamical models.

3.4 Example

In this section, a 12V DC powered CMOS inverter is modeled by the presented framework. Figure 3.10 shows the training signals used to extract the Laguerre model for the inverter. Using the inverted signals of those shown in Figure 3.10 also lead to the same result as it has both rising edge and falling edge information.

The model is constructed with $N = 3$, $M = 200$, $R = 5$, $\alpha = 0.01$. The extracted model is then used to make predictions on a pseudorandom bit sequence (PRBS) test signal. Figure 3.11 shows the high agreement between the model and the transient simulation result.

Next, a decision feedback equalization (DFE) circuit is used to demonstrate the robustness of the presented framework. The DFE circuit must be tunable so that users can change the tap values to optimize the its performance

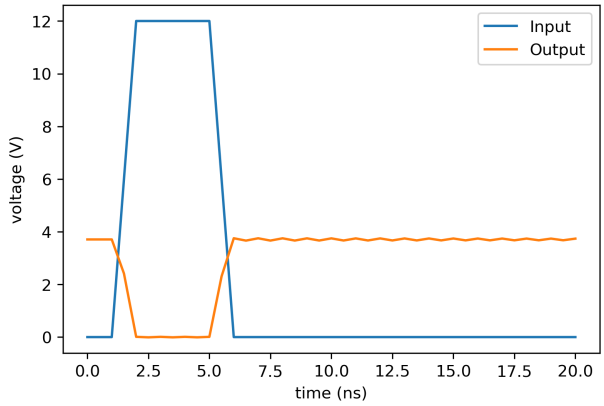


Figure 3.10: Training pulse to extract VL model for the inverter.

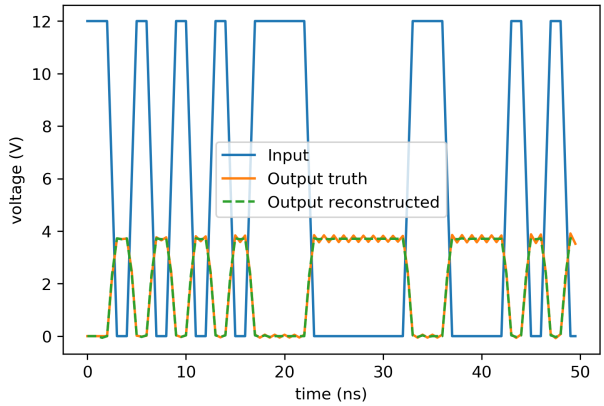


Figure 3.11: Test performance of extracted VL model the inverter with 0.5Gpbs, 1ns rise time PRBS input.

according to their specific design. In the previous chapter, it was shown how a combination of an FNN and an RNN can be used to parametrize a dynamical model of the DFE circuit. In this section, we show how to use the presented framework to achieve the same goal. As can be seen above, to sucessfully *train* a VL model, all needed was a pulse response which contains a rising and a falling edge. This would simplify the training process as much less data is needed for the training compared to using FRNN in the previous chapter. A 2-tap DFE circuit is used in this example for the ease of visualization. A uniform grid sweep of tap values is done, for each tap value, a pulse response similar to that in Figure 3.10 is collected for training. Then interpolants, particularly in this example, Gaussian Processes are used to represent the mapping between tap values and Laguerre coefficients, given a new tap value combination, the Laguerre coefficients that make up the

corresponding system are sampled from these interpolants and predictions can be made using Equation (3.28) and Equation (3.29).

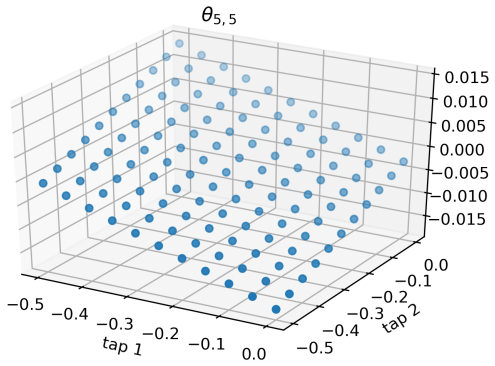


Figure 3.12: A second order Laguerre coefficient, $\theta_{5,5}$, vs. tap values.

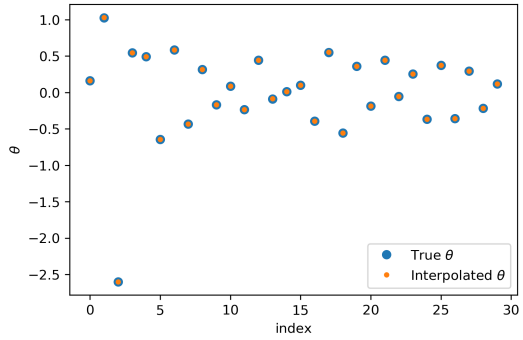


Figure 3.13: First 30 Laguerre coefficients for test case tap values of -0.3 and -0.35.

The parametrized model, at the end, consists of 1,111 interpolants whose input are the tap values and output are the Laguerre coefficients (total of 1 + 10 + 10*10 + 10*10*10). Figure 3.13 shows a good match between the interpolated Laguerre coefficients and direct extraction from known data (only the first 30 coefficients are shown). Figure 3.14 shows the performance of the resulted model for three different tap values. The output voltage predicted by the VL model is in high agreement with that from a transistor level simulator while that from IBIS has lower accuracy.

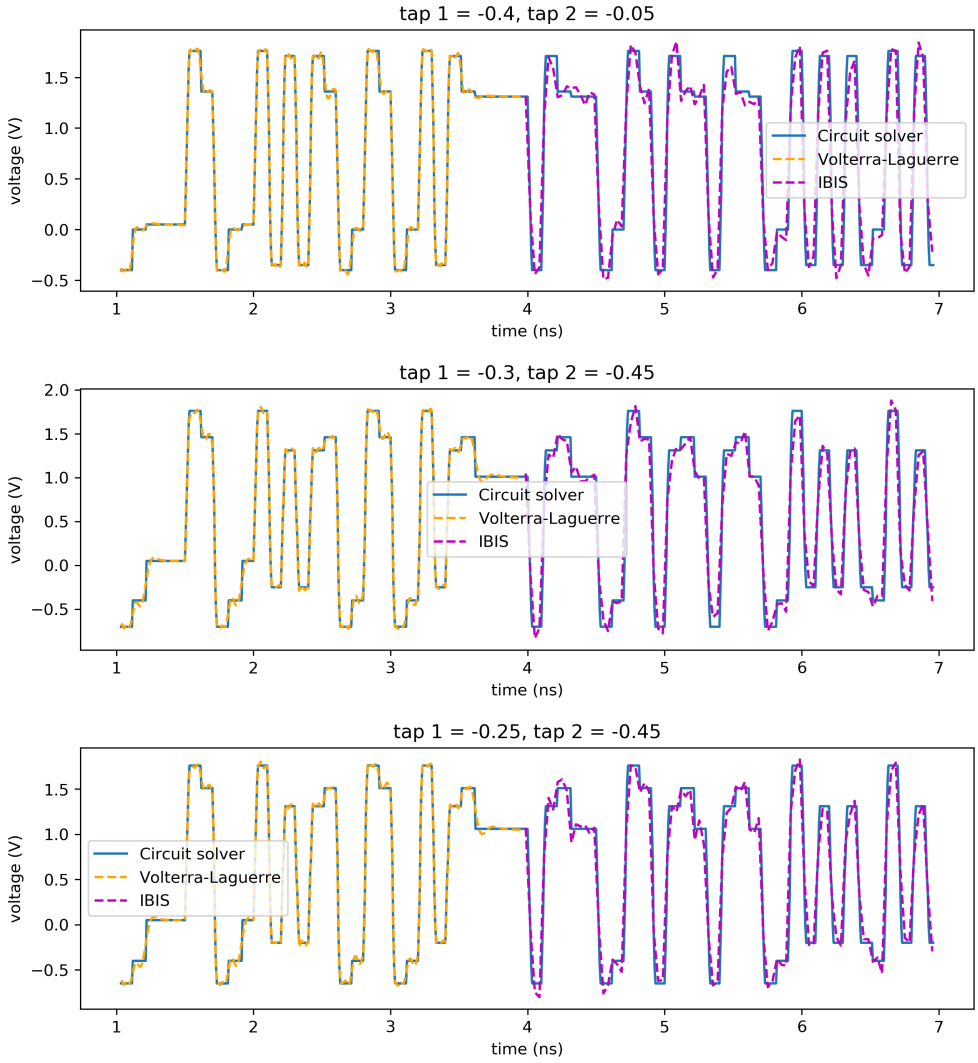


Figure 3.14: DFE circuit output for the same input PRBS, different tap values.

3.5 Volterra model extraction from frequency-domain data

As pointed out in [46, 47, 56, 57], Volterra model of a nonlinear system can also be extracted from frequency domain data. The most popular method is *harmonic probing*. The idea is to excite the system of interest with a number of sinusoidal tones. The response is charactersized by harmonics and intermodulations, the Fourier transform of the Volterra kernels can then be calculated. It was shown in [44] that the Fourier transform of the n^{th} order Volterra kernel can only be completely characterized by an $n-$tone

excitation. We could either use Harmonic Balance (HB) simulation [58] or multi- large signal operating point (LSOP) X-parameter [59] data to obtain the desired model. In this section, the discussion will be focused on using X-parameter data.

3.5.1 X-parameter overview

Figure 3.15 visually summarizes different types of X-parameters and how the signals flow in an X-parameter measurement (simulation) with one tone LSOP. Mathematical development and discussion on X-parameters can be found in [60, 61]. A typical device under test (DUT) has DC stimuli and RF stimuli, X-parameters characterize the DUT by observing the DC and RF response caused by both DC and RF stimuli, under LSOP excitations and under small-signal excitations on top of the LSOP.

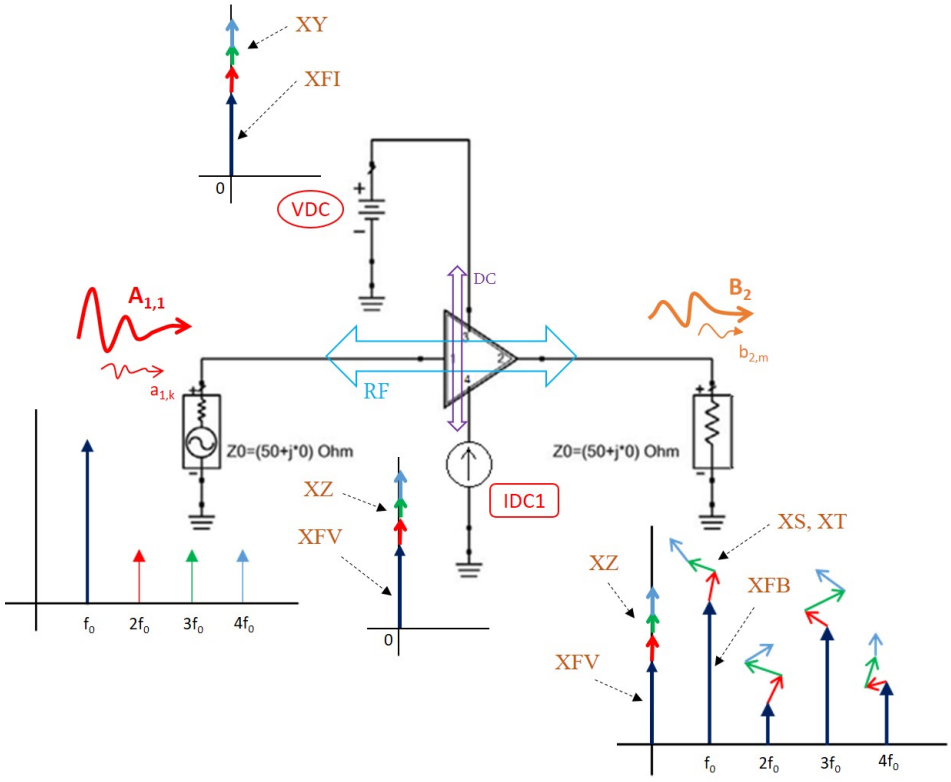


Figure 3.15: X-parameters concept.

For the response to LSOP excitations:

- RF responses to RF stimuli are B-type X-parameter (XB).

- DC responses to DC stimuli are V-type X-parameter (XV) (if the stimuli are DC current sources) or I-type X-parameter (XI) (if the stimili are DC voltage sources).

For the response to the small-signal excitation superimposed on top of the LSOP:

- RF responses to RF stimuli are S-type and T-type X-parameter (XS/XT).
- DC responses to DC stimuli are Z-type X-parameter (XZ) (if the stimuli are DC current sources) or Y-type X-parameter (XY) (if the stimuli are DC voltage sources).

Mathematically, the m^{th} harmonic output power wave at port i is given by:

$$b_{i,m} = \mathrm{XFB}_{i,m} P^m + \sum_{j,l \neq 1,1} \left(\mathrm{XS}_{i,m,j,l} P^{m-l} a_{j,l} + \mathrm{XT}_{i,m,j,l} P^{m+l} a^*_{j,l} \right) \tag{3.30}$$

whilst the DC response at port i is given by:

$$I_i = \mathrm{XI} + \sum_{j,l \neq 1,1} \mathcal{R}e \left\{ \mathrm{XY}_{i,j,l} a_{j,l} \right\} \tag{3.31}$$

$$V_i = \mathrm{XV} + \sum_{j,l \neq 1,1} \mathcal{R}e \left\{ \mathrm{XZ}_{i,j,l} a_{j,l} \right\} \tag{3.32}$$

where a_{11} is the incident large signal at port 1 at the fundamental frequency, corresponding to a certain power level and is used as reference to measure other signals at other ports and harmonics, $P = \angle a_{11}$ is the phase of a_{11}. The index i, m, j, l means the excitation was supplied to port j, harmonic l and the output was observed at port i, harmonic m. While for DC parameters, there is one less index because the output observation is a DC quantity. Index i, j, l means the DC output was observed at port i when the excitation was fed at port j, harmonic l.

3.5.2 The limitation of current NVNA

Consider the magnitude spectrum of a clock signal with peak-to-peak magnitude A and on-time τ and period T (for example, as shown in Figure 3.16).

It has an approximated spectrum bound shown in Figure 3.17 [62]. The spectrum contains several large tones up to at least $\frac{1}{\pi\tau}$. For PRBS signals, the number of tones as well as their magnitudes may reduce but the bound remains the same. When a PRBS-like signal excites a buffer circuit, multiple large tones arrive at the input at the same time. To describe the behavior of the circuit, a multi-tone LSOP X-parameter is required. By design, n-tone LSOP X-parameter provides the both harmonic and intermodulation information required by the harmonic probing setup. Unfortunately, measurement of X-parameter with more than 01 LSOP excitation is not yet available and all X-parameter measurement is limited to single LSOP.

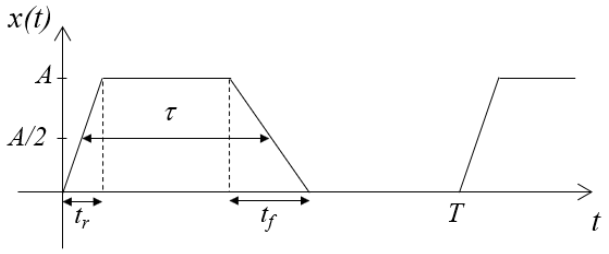


Figure 3.16: Clock signal.

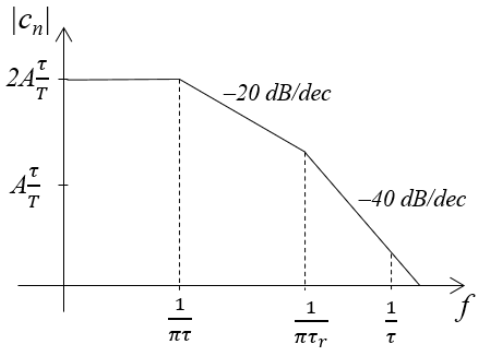


Figure 3.17: Envelope of a clock spectrum.

Chapter 4

HIGH-SPEED LINK DESIGN, OPTIMIZATION, AND VARIABILITY ANALYSIS WITH GAUSSIAN PROCESS

4.1 Introduction

Expedient design iteration and performance optimization and design verification of state-of-the-art electronic devices and systems are hindered by the ever-increasing functionality integration. In the quest for computationally efficient methods capable of handling the high-dimensional design space of such devices and systems, machine learning (ML) methods are being explored recently for modeling and design optimization applications. Surrogate modeling becomes a prominent need due to several reasons. First, for system level assessment, running a full simulation at different physical dimension levels (for example, the chip to chip communication link: IC - package - board - package - IC) is simply too expensive. A surrogate model would provide a tool to evaluate the output given an input almost instantly. This fast-to-evaluate property of a surrogate model proves to be extremely useful when a stochastic analysis such as variability or sensitivity analysis involves, direct Monte Carlo (MC) simulation, or millions and millions of evaluations can be performed to get the statistics or distribution of the variable of interest. Second, the surrogate model is an analytical expression of the mapping between inputs and outputs, hence opens the possibilities for direct design optimization. In the following paragraphs, we provide an overview of recent and on-going research pursuits in this direction that are most pertinent to the work reported in this chapter.

In the surrogate modeling area, the work in [24,25], which use feed-forward neural network (FNN) based models to solve the forward problem, is worth noting. A forward problem in this chapter refers to a problem where a set of design parameters is given and the electrical performance of a circuit is desired. While an inverse problem seeks for the inputs that yield a

desired output. For example, in a high-speed link, the channel geometry and equalization settings are inputs while the eye openings (eye width, eye height) could be the output. A comparative study between different ML methods such as Support Vector Regression (SVR) and FNN has been reported in [63–65] focusing on predicting performance of a high-speed link system. The work reported in [66] uses an SVR-based model to address the design process as an inverse problem instead of an optimization problem. In [67, 68], SVR and active subspace [69] are combined to perform reduced dimensionality regression. This results in a speed-up of the fitting process and facilitates the solution of the sensitivity analysis and design optimization problem. [70] uses Gaussian Process (GP) building a surrogate model to study the behavior of a bandpass filter under the variation of its design parameters. Also solving the forward problem, [71–74] use Partial Least-square Regression (PLS) and Least-square Support Vector Machine (LS-SVM) to perform not only predictions but sensitivity anlysis on design problems with as many as 30 design parameters. In addition, realizing that LS-SVM has a deterministic nature, [73] proposes to combine LS-SVM and GP to create a fully statistical model which, in addition to predictive modeling, provides a confidence measure for its predictions.

In the uncertainty quantification (UQ) regime, in general, we are interested in knowing how a variation in the input would affect the output. Robust, adaptive and computationally cheap methods for efficient stochastic analyses are favorable over naive MC (brute-force) analysis due to the prohibitive cost of MC especially for high dimensional problems. There are various studies and reports on this problem using polynomial chaos expansion (PCE) methods [75, 76]. PCE methods can be categorized into intrusive methods, which require the refomulation of the problem at hand to insert the randomness seeking for a PC representation of the solution [77, 78], and non-intrusive methods, which leave the deterministic model of interest untouched and use, instead, a sampling strategy to sample the data and fit the PCE [76, 79, 80]. Most surrogate modeling methods are suitable for non-intrusive uncertainty propagation, once the input - output mapping is learnt, propagating uncertainty from input to output can be obtained simply by running MC simulation on the surrogate model. Hence, the UQ problem can also be reduced to obtaining a very accurate and fast surrogate model.

Unlike the aforementioned methods, GP is stochastic in nature. A GP is

not just a possible mapping that explains the seen data but a distribution of possible such mappings. Full Bayesian treatment applied to GP parameters during training is what makes it stand out from other surrogate models. In the next sections, we review and compare different surrogate modeling methods, including GP, PLS, SVR and PC. Examples are presented to benchmark the performance of the models. As seen later, GP, more specifically, multi-output GP, overall performs consistently well in all experiments; therefore, it offers an attractive option for input-output black-box modeling, and for efficient uncertainty propagation and sensitivity analysis.

4.2 Gaussian Process Regression

The understanding of Gaussian Process Regression (GPR) cannot be separated from that of Bayesian regression. In the following, nonlinear regression will be revisited in the context of Bayesian learning. Then, the connection between Bayesian regression and GP is established.

4.2.1 Bayesian parametric regression

Consider the probabilistic view of a regression problem: given a set of data $\mathcal{D} = \left\{ \left(\boldsymbol{x}^{(i)}, y^{(i)} \right), i = 1, 2, ..., N \right\}$ of N pairs of $d-$dimensional vector-valued input $\boldsymbol{x}^{(i)} \in \mathcal{X} = \mathbb{R}^d$ and function-valued output $y^{(i)} \in \mathbb{R}$ such that:

$$y = f(\boldsymbol{x}) + \epsilon \tag{4.1}$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ being independent and identically distributed (i.i.d.) Gaussian noise, we seek for the conditional mean of the output, y_*, at test input $\boldsymbol{x}_*$, namely $E(y_* | \boldsymbol{x}_*, \mathcal{D}) = f(\boldsymbol{x}_*)$.

To better understand the probabilistic view of regression, consider the linear regression problem. Bayesian linear regression assumes a parametric form of f as $\exists \boldsymbol{\theta} \in \mathbb{R}^d$, such that:

$$f(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{\theta} = \sum_{k=1}^{d} x_k \theta_k \tag{4.2}$$

The least-square fitting problem minimizes the residual between the data and

the model, i.e. it seeks the solution of:

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{N} \left[\epsilon^{(i)}\right]^2 = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{N} \left\| y^{(i)} - f\left(\boldsymbol{x}^{(i)}\right) \right\|_2^2 \tag{4.3}$$

while the probabilistic view aims to explain the probability of *seeing* the data given the model, hence, it seeks the solution of:

$$\underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{N} p\left(y^{(i)} \middle| \boldsymbol{x}, \boldsymbol{\theta}\right) \tag{4.4}$$

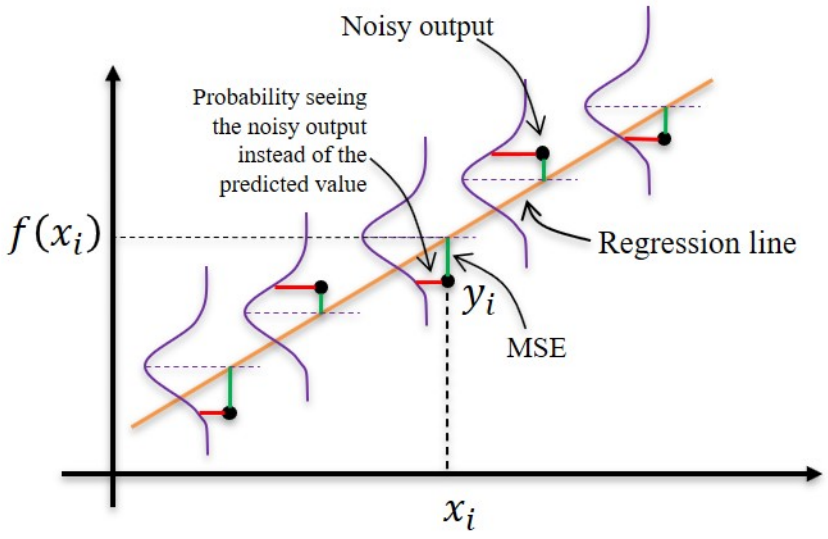


Figure 4.1: Linear regression: fitting point of view vs. probabilistic point of view.

Let us use a visual plot to understand how Equation (4.3) and Equation (4.4) are mathematically equivalent and are both result in the famous least-square solution. In Figure 4.1, black dots are the "noisy" output, the probabilistic model explains the residual as the following: when an input $\boldsymbol{x}$ is given, the regression model (the line) evaluates to $f(x)$. However, when collecting the data, a noise has disturbed this evaluation which results in the black dots as the observed data. The likelihood distribution (the purple bell-shape curve) tells us how likely a black dot is observed from its most likely value. In brief, the least-square solution minimizes the sum of vertical distances from the data to the regression line (the sum of green segments in Figure 4.1),while

the probabilistic model view maximizes the total likelihood of the observed data (the product of red segments). Since the noise is Gaussian, we have:

$$\begin{aligned} p\left(\mathcal{D}|\,\boldsymbol{\theta}\right) &= \prod_{i=1}^{N} p\left(y^{(i)}\big|\,\boldsymbol{x},\boldsymbol{\theta}\right) \\ &= \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}\left[y^{(i)} - \left[\boldsymbol{x}^{(i)}\right]^T\boldsymbol{\theta}\right]^2\right\} \\ &= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^N \exp\left\{-\frac{1}{2\sigma^2}\sum_{i=1}^{N}\left[y^{(i)} - \left[\boldsymbol{x}^{(i)}\right]^T\boldsymbol{\theta}\right]^2\right\} \\ &= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^N \exp\left\{-\frac{1}{2\sigma^2}\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)\right\} \end{aligned} \tag{4.5}$$

where $\boldsymbol{X} \in \mathbb{R}^{N\times d}$ is the input matrix whose rows are corresponding to different samples and columns are corresponding to different input components, $\boldsymbol{y} \in \mathbb{R}^N$ is the output vector.

Finding the maximum of $p\left(\mathcal{D}|\,\boldsymbol{\theta}\right)$ now is equivalent to finding the minimum of $\ell\left(\boldsymbol{\theta}\right) = \left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)$. Taking the derivatives of $\ell\left(\boldsymbol{\theta}\right)$ gives:

$$\frac{\partial \ell\left(\boldsymbol{\theta}\right)}{\partial \boldsymbol{\theta}} = -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} \tag{4.6}$$

Letting $\frac{\partial \ell(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0$, we obtain the least-square solution for $\boldsymbol{\theta}$ as:

$$\boldsymbol{\theta} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{4.7}$$

Recall that we have applied the following matrix calculus identities: if $\boldsymbol{A}$ is square and not a function of $\boldsymbol{x}$ and $\boldsymbol{u}$ is a vector and a function of $\boldsymbol{x}$, then:

$$\frac{\partial}{\partial \boldsymbol{x}}\boldsymbol{u}^T\boldsymbol{A}\boldsymbol{u} = \boldsymbol{u}^T\left(\boldsymbol{A}+\boldsymbol{A}^T\right)\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}} \tag{4.8}$$

and

$$\frac{\partial}{\partial \boldsymbol{x}}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T \tag{4.9}$$

In the above process, we did not assume any prior on $\boldsymbol{\theta}$. In other words, it was implied that $\boldsymbol{\theta}$ can be any real numbers (hence, some authors refer to this as the same as applying a uniform prior). The consequence of this

is that the solution we obtain is a *point estimate*, or, a single value. Indeed, it yields the solution given by Equation (4.7). In general, the probabilistic modeling process relies on Bayes' rule to solve for the parameters. Full Bayesian treatment is usually applied as it provides a convenient and powerful mean to insert domain knowledge into the model. The solution for $\boldsymbol{\theta}$ is not a single value anymore but a distribution. It tells the user how likely or unlikely a value $\boldsymbol{\theta}$ can take. A full Bayesian treatment to Bayes regression model will have a connection to GP.

When doing Bayesian regression, after imposing a prior on $\boldsymbol{\theta}$, in particular $\boldsymbol{\theta} \sim \mathcal{N}\left(0, \frac{1}{d}\Sigma_\theta\right)$, the parameter's ($\boldsymbol{\theta}$'s) posterior is given by Bayes' rule:

$$p\left(\boldsymbol{\theta}\middle|\,\mathcal{D}\right) = \frac{p\left(\mathcal{D}\middle|\,\boldsymbol{\theta}\right)p\left(\boldsymbol{\theta}\right)}{\int_\theta p\left(\mathcal{D}\middle|\,\boldsymbol{\theta}\right)p\left(\boldsymbol{\theta}\right)\mathrm{d}\boldsymbol{\theta}} \tag{4.10}$$

for a test (unseen) input $\boldsymbol{x_*}$, the predicted output y_* can be sampled from the posterior predictive distribution calculated by marginalizing $\boldsymbol{\theta}$ out,

$$p\left(y_*\middle|\,\boldsymbol{x_*}, \mathcal{D}\right) = \int_\theta p\left(y_*\middle|\,\boldsymbol{x_*}, \boldsymbol{\theta}\right)p\left(\boldsymbol{\theta}\middle|\,\mathcal{D}\right)\mathrm{d}\boldsymbol{\theta} \tag{4.11}$$

Usually, Equation (4.10) is intractable due to the integral in the denominator. However, thanks to $\boldsymbol{\theta}$'s prior and the noise being Gaussian, the posterior predictive distribution, Equation (4.11), is also Gaussian. Hence, what is left is to find its mean and variance, which is a straightforward process. A Bayesian nonlinear regressor introduces a nonlinear transformation, often called the feature map, to transform a $d-$dimensional vector-valued input $\boldsymbol{x}$ to a $d'-$dimensional vector-valued feature z, $\boldsymbol{\varphi} : \mathbb{R}^d \longmapsto \mathbb{R}^{d'}$; Equation (4.2) now becomes:

$$f\left(\boldsymbol{x}\right) = \boldsymbol{\varphi}\left(\boldsymbol{x}\right)^T\boldsymbol{\theta} \tag{4.12}$$

The posterior predictive distribution now involves the term $\boldsymbol{\varphi}\left(\boldsymbol{x}\right)^T\Sigma_\theta\boldsymbol{\varphi}\left(\boldsymbol{x'}\right)$ instead of just simply $\boldsymbol{x}^T\Sigma_\theta\boldsymbol{x'}$ where $\boldsymbol{x}$ and $\boldsymbol{x'}$ are 2 arbitrary inputs in either the training or the prediction points. Since $\Sigma_\theta \succ 0$, let

$$k\left(\boldsymbol{x}, \boldsymbol{x'}\right) = \frac{1}{d}\boldsymbol{\varphi}\left(\boldsymbol{x}\right)^T\Sigma_\theta\boldsymbol{\varphi}\left(\boldsymbol{x'}\right) \tag{4.13}$$

be the *covariance function*. It characterizes the similarity and correlation

between the features $\varphi(\boldsymbol{x})$ and $\varphi(\boldsymbol{x}')$, as we shall see in detail later, it gives rise to the conditional predictive distribution.

4.2.2 Non-parametric Gaussian Process

A Gaussian Process (GP) is a random process, i.e. a collection of random variables whose any finite set obeys a multivariate Gaussian distribution. That means a GP can be fully characterized by a mean function $m(\boldsymbol{x})$ and a covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$. We say that $f(\boldsymbol{x}) \sim \mathcal{GP}(m, k)$ iff for all $M \in \mathbb{N}$,

$$\mathbf{f} = \left[\begin{array}{cccc} f\left(\boldsymbol{x}^{(1)}\right) & f\left(\boldsymbol{x}^{(2)}\right) & \cdots & f\left(\boldsymbol{x}^{(M)}\right) \end{array} \right] \sim \mathcal{N}\left(\bar{\boldsymbol{f}}, \boldsymbol{K}\right) \tag{4.14}$$

where

$$\bar{\boldsymbol{f}}_i = m\left(\boldsymbol{x}^{(i)}\right) \tag{4.14a}$$

$m(\cdot)$ is, typically, without loss of generality, chosen to be $\mathbf{0}$ and:

$$\boldsymbol{K}_{\boldsymbol{ij}} = \mathrm{Cov}\left(f\left(\boldsymbol{x}^{(i)}\right), f\left(\boldsymbol{x}^{(j)}\right)\right) = k\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) \tag{4.14b}$$

which is also refered to as the Gram(ian) matrix.

A classical literature on this topic is [81]. GP is known as a probability distribution over a family of functions. A sample from a GP is a function, or more precisely, a finite set of (N) evaluations of a function. The kernel function gives rise to the covariance matrix of a multivariate Gaussian distribution which needs to be positive semi-definite (PSD). This puts a constraint on the type of kernel functions that are valid to describe a GP.

There are many popular kernel functions that can be used to specify a GP prior: squared-exponential, rational quadratic, Matern, RBF, periodic, etc. [81]. Kernel functions can also be combined together to represent a complex prior; the following kernel is the sum of q basic kernels:

$$k_a(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{q} k_i(\boldsymbol{x}, \boldsymbol{x}') \tag{4.15}$$

Since $k_i(\boldsymbol{x}, \boldsymbol{x}')$'s are PSD, their sum, $k_a(\boldsymbol{x}, \boldsymbol{x}')$, is also PSD.

Eventhough GP relies on Gaussian noise assumption to make use of its

analytical property, as pointed out in the previous section, it is rich enough to approximate any family of functions. For example, the RBF kernel is defined as:

$$k_{RBF}\left(\boldsymbol{x}, \boldsymbol{x'}\right) = h^2 \exp\left(-\frac{1}{2w^2}\left\|\boldsymbol{x} - \boldsymbol{x'}\right\|^2\right) \tag{4.16}$$

We will use a one-dimensional GP for ease of visualization. Figure 4.2 shows 10 different samples from a 1D GP built on the RBF kernel with $h = 1$ and $w = 1$. Each curve with different color in Figure 4.2 is a function sampled from the GP. Points on any curve in Figure 4.2 obey the multivariate Gaussian distribution whose covariance matrix is computed by Equation (4.16).

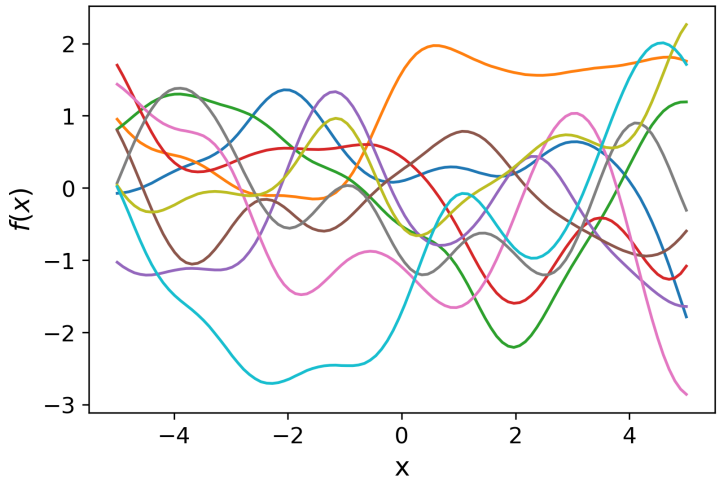


Figure 4.2: GP prior

Once some data points are given, there is no more uncertainty at the locations where data is available. Any functions that are sampled from the updated (posterior) GP are forced to go through the given data points. Figure 4.3 illustrates this phenomenon. Notice that all points on a function are correlated. Their correlation is given by the covariance matrix. Therefore, available data points only not enforce what the functions look like at the data location but also changes what they look like around those location. As we can see from Figure 4.3, 10 samples from the posterior GP look much more organized as if they were about to form a particular function that best explain the 5 given data points.

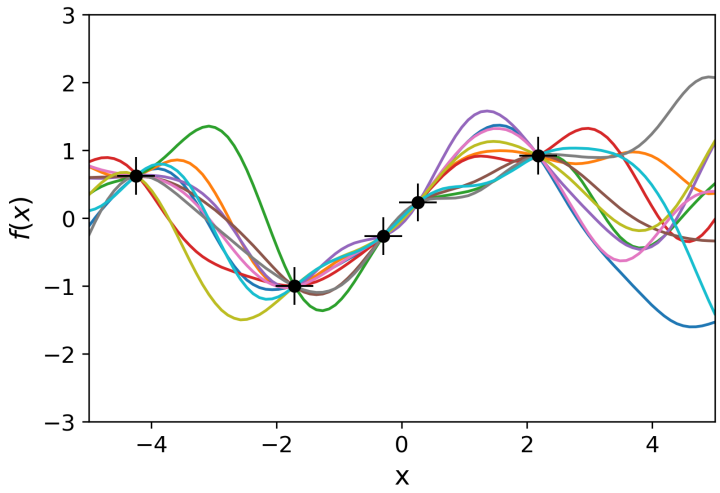


Figure 4.3: GP posterior

If we take all of the curves 50 examples of which were shown in Figure 4.4 and average them, we obtain the predictive mean. Figure 4.4 shows the true function and the GP predictive mean obtained from 5 data points above. The predictive mean can be interpreted as the prediction given by the trained GP model. The shaded area in Figure 4.4 can be interpreted as the 95% confidence interval of the GP prediction, in other words, 95% of the functions that go through the given data points will land in the shaded regions at locations where data is not given as shown.

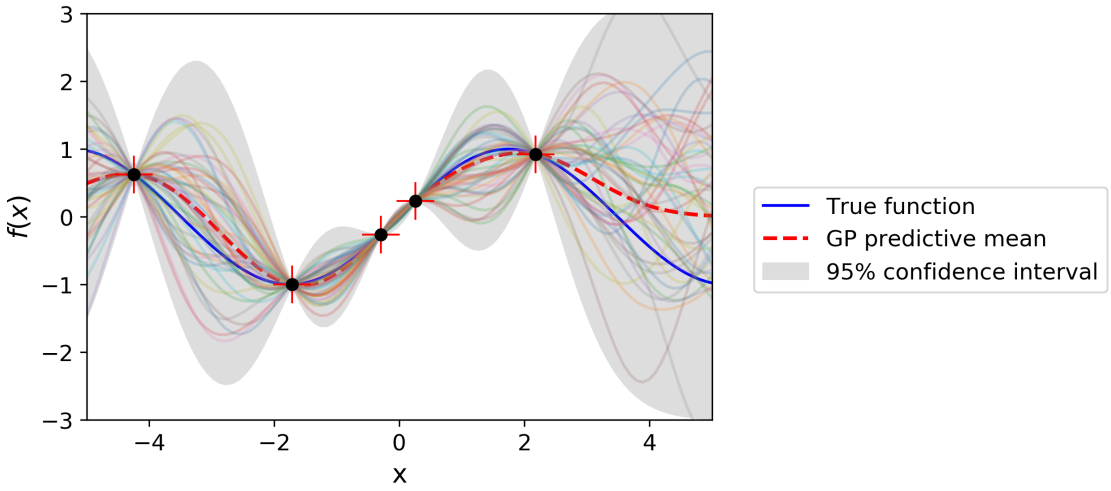


Figure 4.4: Predictive mean and uncertainty from the posterior GP

In addition, [82] proposes additive kernels that act differently on each di-

mension of the input $\boldsymbol{x}$ which enrich the class of functions that can be represented by GP even more. Furthermore, such an approach can also capture different degree of interactions between the input dimensions. The scope of this chapter will focus on basic kernels. Applications of GP involving additive kernels have been reported in [83, 84].

4.2.3 Relationship between GP and Bayesian Regression

Without loss of generality, we will use d as the dimension of the feature space in this section. In parametric Bayesian regression, Equation (4.12) assumes the model is a d-component basis expansion and could be rewritten as:

$$f(\boldsymbol{x}) = \sum_{i=1}^{d} \boldsymbol{\varphi_i}(\boldsymbol{x}) \theta_i \tag{4.17}$$

and the kernel function is written as:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \frac{\sigma^2}{d} \sum_{i=1}^{d} \boldsymbol{\varphi_i}(\boldsymbol{x}) \boldsymbol{\varphi_i}(\boldsymbol{x'}) \tag{4.18}$$

If we take $d \to \infty$ then Equation (4.18) becomes a Riemann sum and:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \int_{-\infty}^{\infty} \boldsymbol{\varphi_c}(\boldsymbol{x}) \boldsymbol{\varphi_c}(\boldsymbol{x'}) \, \mathrm{d}c \tag{4.19}$$

This is a remarkable result. First, it indicates that each feature map in Bayesian regression is corresponding to a kernel function in GP. For example, the squared exponential kernel function:

$$k_{SE}(x, x') = \sigma_{SE}^2 \exp\left[-\frac{(x - x')^2}{2\ell_{SE}^2}\right] \tag{4.20a}$$

was derived from the feature map:

$$\varphi_i(x) = \exp\left[-\frac{(x - c_i)^2}{2\ell^2}\right] \tag{4.20b}$$

and the prior imposed on the model coefficients:

$$\theta_i \sim \mathcal{N}\left(0, \frac{\sigma^2}{d}\right) \tag{4.20c}$$

with $\sigma_{SE}^2 = \sqrt{\pi}\ell\sigma^2$, $\ell_{SE} = \sqrt{2}\ell$.

Second, instead of choosing a set of finite feature map basis $\varphi_i(\boldsymbol{x})$ and perform regression task, an infinite order model ($d \to \infty$) can be formed by using GP. The model can learn as much as available data. Its complexity increases as the data provided increases and does not depend on the complexity of the hypothesis, i.e. the number of parameters. It is especially clear when the GP model making predictions at unseen points, the posterior predictive distribution requires Equation (4.11) to maginalize the model parameters $\boldsymbol{\theta}$ out. Third, a reproducing kernel Hilbert space (RKHS), $k(\boldsymbol{x}, \boldsymbol{x'})$, is all needed to build a powerful non-parametric model from the data. Topologically speaking, when using feature maps to convert the original input space ($\boldsymbol{x}$) to the feature space, $\varphi(\boldsymbol{x})$, we only need to be able to compute the dot product of the feature maps, i.e. evaluating $k(\boldsymbol{x}, \boldsymbol{x'})$ for any pair of input $\boldsymbol{x}, \boldsymbol{x'}$, the transformation itself, $\varphi(\cdot)$, is not needed [85, 86]. Any valid GP kernel function must have a corresponding feature map in Bayesian regression point of view. Thus, in doing regression by GP, instead of putting a prior on the expansion coefficients of a specific basis function, we put a GP prior on the mapping we are interested in and compute the posterior distribution as the data becomes available. A thorough treatment of Hilbert space and RKHS with more details can be found in [85].

It can be seen that for large d and small N, performing regression by GP is prefered over using a feature map while the reverse is true if the problem at hand has small d but N is large.

4.2.4 Inference in GP models

This section presents the steps needed to implement a GP regressor. First, a kernel is choosen, this is equivalent to choosing a nonlinear feature map in classical regression. Typically, depending on domain knowledge, an appropriate kernel can be chosen to reflect the property of the mapping of interest. However, there are certain kernels such as the Matern kernel are shown to

be empirically working for most applications as long as the mapping is continuous and differentiable (also known informally as *nicely behaved*). In this chapter, Matern-3/2 kernel [87] is chosen as the default kernel for all experiments shown later in the example section. The Matern-3/2 kernel is a particular case of a more general family of kernel called the Matern kernel, parametrized by a parameter ν. When $\nu = 1.5$, the Matern kernel is called Matern-3/2 and is given by:

$$k_{Matern-3/2}\left(\boldsymbol{x}, \boldsymbol{x}'\right) = \sigma^2 \left(1 + \sqrt{3}\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|}{\ell}\right) \exp\left(1 - \sqrt{3}\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|}{\ell}\right) \quad (4.21)$$

where σ and ℓ are the hyper-parameters.

Kernels always have some hyper-parameters that need to be picked by the users. However, thanks to the full Bayesian treatment, no cross-validation or grid search is needed to tune these hyper-parameters in GP. They can be learnt by minizing the negative log-likelihood (NLL) using a gradient based optimization method such as Adam optimizer [32]. The NLL can be analytically derived as [81]:

$$\mathrm{NLL}\left(\boldsymbol{\theta}\right) = \frac{N}{2}\log\left(2\pi\right) - \frac{1}{2}\log\left|\boldsymbol{K_{rr}}\right| - \frac{1}{2}\boldsymbol{y}^T\boldsymbol{K_{rr}^{-1}}\boldsymbol{y} \quad (4.22)$$

where $|\boldsymbol{K_{rr}}|$ is the determination of matrix $\boldsymbol{K_{rr}}$. The subscript r indicates that the Gram matrix $\boldsymbol{K_{rr}}$ is calculated from Equation (4.14)b using training data.

For example, if the training data is tabulated as in Figure 4.5, we have 16 inputs and 2 outputs (eye height and eye width), $\boldsymbol{x} \in \mathbb{R}^{16}$ and $\boldsymbol{y} \in \mathbb{R}^2$. Figure 4.5 shows 5 samples (indexed 0 to 4). Assuming that these 5 samples are all training data we have, in order to compute the NLL, we first need to calculate the covariance matrix $K_{rr} \in \mathbb{R}^{5\times5}$ the ij element of which is given by Equation (4.21):

$$K_{rr} = \begin{bmatrix} k_M\left(\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(0)}\right) & \cdots & k_M\left(\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(4)}\right) \\ k_M\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(0)}\right) & & k_M\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(4)}\right) \\ & \ddots & \\ \vdots & & \vdots \\ k_M\left(\boldsymbol{x}^{(4)}, \boldsymbol{x}^{(0)}\right) & \cdots & k_M\left(\boldsymbol{x}^{(4)}, \boldsymbol{x}^{(4)}\right) \end{bmatrix} \quad (4.23)$$

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16	Eye Height	Eye Width
0	5.1546	4.5378	8.3855	4.0612	9.8520	4.1919	4.6035	7.8138	8.2429	8.2456	238.95	4.6778	5.4746	7.4981	4.1951	7.2382	0.299	8.900000e-11
1	4.5298	5.3275	7.6279	3.8287	2.0269	4.9562	4.8207	7.4530	8.6283	8.2651	346.67	4.0702	4.5812	8.2825	3.7545	10.5290	0.172	7.400000e-11
2	5.1494	4.5283	7.5908	4.1035	5.9506	4.0397	5.2299	8.2270	7.9204	8.0055	238.79	3.8274	4.9280	7.7717	3.8238	12.9840	0.298	9.000000e-11
3	3.6663	4.8672	7.4646	3.8943	8.9856	4.3006	4.8659	7.2615	8.5871	7.2503	470.97	5.1414	4.5816	7.4883	4.1759	15.6720	0.052	4.750000e-11
4	3.9018	4.6748	8.1715	4.0805	15.0380	4.1089	5.0102	7.3070	8.4008	8.4214	370.58	4.5129	5.3528	7.8615	4.1090	8.3095	0.170	7.300000e-11

Figure 4.5: Excerpt of training data for the high-speed link example.

In this case, K_{rr} is a matrix function of σ and ℓ. So is the NLL.

To optimize the NLL, a gradient based optimizer is used. The gradient of the NLL can be calculated analytically to iteratively optimize the hyper-parameters:

$$\nabla_{\theta_j}\mathrm{NLL}\left(\theta\right) = -\boldsymbol{y}^T\overline{\boldsymbol{K}}^{-1}\frac{\partial\overline{\boldsymbol{K}}}{\partial\theta_i}\overline{\boldsymbol{K}}^{-1}\boldsymbol{y} - \mathrm{tr}\left(\overline{\boldsymbol{K}}^{-1}\frac{\partial\overline{\boldsymbol{K}}}{\partial\theta_i}\right) \tag{4.24}$$

where $\overline{\boldsymbol{K}} = \boldsymbol{K}_{rr} + \sigma^2\boldsymbol{I}$. This process of finding the optimal hyper-parameters for the kernel is refered to as the training process. Implementation wise, the NLL can be implemented in Pytorch the same way a neural network is and Pytorch's optimization methods can be made use of. The NLL is coded as a Pytorch function according to Equation (4.22) and registered to Pytorch as the cost (loss) function that needs to be minimized. Thanks to auto-differentiation, the gradient of NLL w.r.t. $\boldsymbol{\theta}$ can be automatically calculated, there is no need to compute $\nabla_{\theta_j}\mathrm{NLL}\left(\theta\right)$ manually. After looping through the training dataset, the hyper-parameters are updated the same way the weights in a neural network is updated.

Once the training completes, as testing data, $\boldsymbol{x}_*$, is fed into the model, prediction, $\boldsymbol{y}_*$, can be made by sampling from the posterior distribution:

$$p\left(\boldsymbol{y}_*|\,\boldsymbol{x}_*, \mathcal{D}\right) = \mathcal{N}\left(\boldsymbol{\mu}_*, \boldsymbol{K}_*\right) \tag{4.25}$$

where

$$\boldsymbol{\mu}_* = \boldsymbol{K}_{\boldsymbol{tr}}\overline{\boldsymbol{K}}^{-1}\boldsymbol{y} \tag{4.25a}$$

$$\boldsymbol{K}_* = \boldsymbol{K}_{\boldsymbol{tt}} - \boldsymbol{K}_{\boldsymbol{tr}}\overline{\boldsymbol{K}}^{-1}\boldsymbol{K}_{\boldsymbol{rt}} \tag{4.25b}$$

The subscript t stands for testing data. $\boldsymbol{K}_{\boldsymbol{tr}}$ is the covariance submatrix between test and train data, i.e. $\left(\boldsymbol{K}_{\boldsymbol{tr}}\right)_{ij} = k\left(\boldsymbol{x}_t^{(i)}, \boldsymbol{x}_r^{(j)}\right)$.

For a multi-output system, it is reasonable to expect a relationship, or

a correlation between different components of the outputs. As shown in the examples, for a multi-output system, learning all the output together exploits the correlation between them, makes the training converge faster and improves the prediction ability of the model. An m-component output GP, generally refered to as multi-output GP (MOGP) in this thesis, is constructed by creating a linear mixture of multiple single-output GPs. For example, if the output has two features, i.e. $m = 2$, the model can be constructed as:

$$\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} u^{(1)} \\ u^{(2)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{a_1} & \boldsymbol{a_2} \end{bmatrix} \begin{bmatrix} u^{(1)} \\ u^{(2)} \end{bmatrix} \tag{4.26}$$

where $u^{(1)}$ and $u^{(2)}$ are output of two latent single-output GP, $\boldsymbol{a_1} = \begin{bmatrix} a_{11} & a_{12} \end{bmatrix}^T$ and $\boldsymbol{a_2} = \begin{bmatrix} a_{21} & a_{22} \end{bmatrix}^T$. The covariance matrix of the MOGP relates to that of the latent GPs, $\boldsymbol{K_u}$, by

$$\operatorname{cov}\left(\boldsymbol{f}\left(\boldsymbol{x}\right), \boldsymbol{f}\left(\boldsymbol{x}'\right)\right) = \left(\sum_{i=1}^{m} \boldsymbol{a_i} \boldsymbol{a_i}^T\right) \boldsymbol{K_u} \tag{4.27}$$

The coefficient a_{ij}'s are also considered hyper-parameters, included in $\boldsymbol{\theta}$. There are many other methods to generate a multi-output kernel from single-output ones which are discussed in [87]. They are more complicated, more expensive to implement and more appropriate for big data applications than the scope of this thesis, hence, are not discussed here.

4.3 Other surrogate modeling methods

For comparison purposes, this section is devoted to review some most recent ML techniques for surrogate modeling in the literature including Partial Least-square Regression [71, 72], Support Vector Regression [63, 65, 67, 68, 88, 89] and Polynomial Chaos Regression [76, 79, 80, 90]. Polynomial Regression is also reviewed and used as the baseline to indicate the nonlinearity in the problem.

4.3.1 Partial Least-square (PLS) Regression

Partial Least-square (PLS) regression relies on the idea of principle component analysis. Principle component regression (PCR) involves the principle component analysis (PCA) in which the input space is reduced to the principle component space; then, an interpolation is carried out between a few significant principle components and the output. Assume a multi-input multi-output system, i.e. $y \in \mathbb{R}^q$. Let

$$\boldsymbol{X} = \boldsymbol{V}\boldsymbol{P}^{\boldsymbol{T}} \tag{4.28a}$$

$$\boldsymbol{Y} = \boldsymbol{U}\boldsymbol{Q}^{\boldsymbol{T}} \tag{4.28b}$$

be the principle decomposition of $\boldsymbol{X} \in \mathbb{R}^{N\times d}$ and $\boldsymbol{Y} \in \mathbb{R}^{N\times q}$, $\boldsymbol{V}, \boldsymbol{P}, \boldsymbol{U}$ and $\boldsymbol{Q}$ are of appropriate dimensions. PCR perform regression on V and $\boldsymbol{U}$. We can see that though $\boldsymbol{V}$ best describes inputs and $\boldsymbol{U}$ best describes outputs as PCA was applied to both input and output, it was applied separately. PLS fixes this limitation, it iteratively projects input and output onto the most significant components but the projection happens in a leapfrog scheme so that there is cross-information exchange between input and output while doing projections. Process details can be found in [71, 72]. After L projections, we obtain an L-component decomposition of $\boldsymbol{X}$ and $\boldsymbol{Y}$, $\boldsymbol{V}, \boldsymbol{U} \in \mathbb{R}^{N\times L}$ and $\boldsymbol{P} \in \mathbb{R}^{d\times L}$, $\boldsymbol{Q} \in \mathbb{R}^{q\times L}$. A regression model can be created using $\boldsymbol{U}$ and $\boldsymbol{V}$:

$$\boldsymbol{U} = \boldsymbol{V}\boldsymbol{\theta} \tag{4.29}$$

Predictions can be obtained by:

$$\boldsymbol{Y} = \boldsymbol{U}\boldsymbol{Q}^{\boldsymbol{T}} = \boldsymbol{V}\boldsymbol{\theta}\boldsymbol{Q}^{\boldsymbol{T}} = \boldsymbol{X}\boldsymbol{P}\boldsymbol{\theta}\boldsymbol{Q}^{\boldsymbol{T}} \tag{4.30}$$

For single-output case, the process finding $\boldsymbol{V}$ and $\boldsymbol{U}$ becomes one-step calculation while for multi-output case, it is iterative. In the experiments below, single-output and multi-output PLS are used as two different approaches and will also be benchmarked against each other. To find the optimal L, a cross-validation scheme is used, multiple PLS models are built as L varies and the optimal L is chosen when the corresponding model achieve lowest fitting error.

4.3.2 Support vector regression (SVR)

Support vector regression (SVR) [91], an important branch of support vector machine (SVM) [92], aims to solve the regression prediction problem by finding a regression plane to which all the data set are closest.

SVR method is utilized to find a plane expressed by $h(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\varphi(\boldsymbol{x}) + b$, such that $h(\boldsymbol{x})$ is as close as possible to y.

The SVR problem can be written as:

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\xi},\boldsymbol{\xi}'} & \frac{1}{2}\|\boldsymbol{w}\| + C\sum_{1\le i\le N}\left(\xi_i + \xi_i^{'}\right), \\ \text{s.t. } & h\left(\boldsymbol{x}^{(i)}\right) - y^{(i)} \le \varepsilon + \xi_i, \\ & y^{(i)} - h\left(\boldsymbol{x}^{(i)}\right) \le \varepsilon + \xi_i^{'}, \\ & \xi_i \ge 0,\ \xi_i^{'} \ge 0,\ i = 1, 2, \ldots, N \end{aligned} \tag{4.31}$$

where ξ_i and $\xi_i^{'}$ are slack variables, $\boldsymbol{w} = \begin{bmatrix} w_1 & \cdots & w_d \end{bmatrix}^T$ is a normal vector of hyperplane, C is a positive constant and SVR allows a margin of tolerance ε. The final SVR predictive function can be calculated via a Lagrange multiplier:

$$h(\boldsymbol{x}) = \sum_{1\le i\le N}\left(\alpha_i^{'} - \alpha_i\right)\kappa\left(\boldsymbol{x}, \boldsymbol{x}^{(i)}\right) + b \tag{4.32}$$

where $\alpha_i \ge 0$ and $\alpha_i^{'} \ge 0$ are introduced as Lagrange multipliers and Gaussian kernel was used for expriments presented in the example section [**?**, 68]:

$$\kappa\left(\boldsymbol{x}, \boldsymbol{x}^{(i)}\right) = \exp\left(\frac{-\left\|\boldsymbol{x} - \boldsymbol{x}^{(i)}\right\|^2}{2\sigma^2}\right) \tag{4.33}$$

where $\sigma > 0$ is the width of Gaussian kernel.

SVR algorithm is implemented by MATLAB Statistics and Machine Learning Toolbox, which hyperparameters, e.g. C, σ and ε, are calculated by Bayesian optimization method in order to minimize the cross-validation error and provide accuracy prediction results with robustness.

4.3.3 Polynomial Chaos (PC)

The Polynomial Chaos (PC) theory introduces a way to estimate an arbitrary random variable of interest as a function of another random variable

with a given distribution, and as a model of an orthonormal polynomial expansions. This method is known as its fast convergence and low computation cost than Monte Carlo (MC) analysis [93]. The statistic information such as mean and variance of the output is given at no cost with the process of solving PC model. The general form of multidimensional polynomials is estimated as [94]:

$$y \approx \sum_{i=0}^{P} c_i \Phi_i(\boldsymbol{x}) \tag{4.34}$$

where c_i denotes the unknown polynomial coefficients to be determined, $\Phi_i(\boldsymbol{x})$ represents multidimensional orthonormal polynomials, constructed using the product of the 1D orthonormal polynomials, via:

$$\boldsymbol{\Phi}_i = \prod_{k \in \mathcal{K}_i} \phi_k \tag{4.35}$$

where $\mathcal{K}_i$ is multi-index set for 1D orthonormal polynomials:

$$\mathcal{K}_i = \{k_{i1}, \cdots, k_{id}\}, \sum k_{ij} \leq m \tag{4.36}$$

The number of polynomial terms, $P = \dfrac{(m+d)!}{m!d!}$ where m is the polynomial order and d as the dimension of the input. Depending on the distribution of input variables $\boldsymbol{x}$, the polynomial basis $\phi(\cdot)$ is chosen accordingly to make the bases are orthogonal to each other. According to [94], Hermite polynomials can be used as basis functions to represent Gaussian random variables by a set of deterministic coefficients. In other case, Legendre polynomials can be used for uniform distributions. Rewriting Equation (4.34) in matrix form yields:

$$\underbrace{\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix}}_{y} = \begin{bmatrix} \Phi_0(\boldsymbol{x^{(1)}}) & \Phi_1(\boldsymbol{x^{(1)}}) & \cdots & \Phi_P(\boldsymbol{x^{(1)}}) \\ \Phi_0(\boldsymbol{x^{(2)}}) & \Phi_1(\boldsymbol{x^{(2)}}) & \cdots & \Phi_P(\boldsymbol{x^{(2)}}) \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_0(\boldsymbol{x^{(N)}}) & \Phi_1(\boldsymbol{x^{(N)}}) & \cdots & \Phi_P(\boldsymbol{x^{(N)}}) \end{bmatrix} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_P \end{bmatrix}}_{c} \tag{4.37}$$

The coefficients c_i's are linear w.r.t to y, therefore, given N sets of training samples, c_i's can easily be solved by linear regression method, given by:

$$c = (\boldsymbol{\phi}^T \boldsymbol{\phi})^{-1} \boldsymbol{\phi}^T y \tag{4.38}$$

4.3.4 Polynomial regression

Polynomial regression (PR) is the oldest regression method but appears to be very effective in modeling real world data. PR assumes the mapping between the input $\boldsymbol{x}$ and the output y is given by:

$$y = \sum_{i=0}^{P-1} \beta_i \prod_{j=1}^{d} x_j^{k_j} \tag{4.39}$$

where P is the number of terms, given by $P = \frac{(M+d)!}{M!d!}$, m is the polynomial order, x_j is the j^{th} component of $\boldsymbol{x} \in \mathbb{R}^d$, or the j^{th} input variable. The monomial power $k_j \geq 0$ must satisfy the condition:

$$\sum_{j=1}^{d} k_j \leq M \tag{4.40}$$

When $M = 1$, the regression model is linear, hence, refered to as linear regression (LR). For example, if $M = 3$ and $d = 2$, $P = 6$ and the PR model reads:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \beta_4 x_1^2 + \beta_5 x_2^2 \tag{4.41}$$

and the LR model only has linear terms:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \tag{4.42}$$

PR and LR are included in the study as the baseline to indicate how much nonlinearity exists in the problem. For a problem that LR performs well on, the input - output mapping must be linear and vice versa.

4.4 Examples

In this section, different examples in microwave circuit and high-speed designs will be used to investigate the performance of various surrogate models against GP. Each experiment is designed to build a predictive model starting with $N = 10$ randomly distributed training samples. Once a model is generated, it is validated by calculating the coefficient of determination, or R^2 score, between the true values $\hat{y}$ and the predicted values y, defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{N} \left\| y^{(i)} - \hat{y}^{(i)} \right\|^2}{\sum_{i=1}^{N} \left\| \hat{y}^{(i)} - \bar{y} \right\|^2} \tag{4.43}$$

where $\bar{y} = \frac{1}{N} \sum_{i=1}^{N} \hat{y}^{(i)}$.

An R^2 score of 1.0 means the predicted and the true values are in perfect agreement. If the validation R^2 score of a model reaches 0.99, the training for that particular model may be stopped, this will sometimes be refered to as the convergence of a model. If the model has not reached convergence, i.e. $R^2 < 0.99$, a number of training samples is added and the model is retrained using this new set of training data. It is worth noting that in cases where the output is multi-dimensional, when using single-output models such as PLS or SVR, we have to create multiple independent such models. Since MOPLS, MOGP and PC are multi-output models, only one model is needed. In the following, training R^2 scores for each output feature are reported separately as they are different for single-output models but one should expect a single R^2 score to report for multi-output models.

As demonstrated below, the PLS method, though fast, is inherently linear, hence, unable to capture nonlinear input-output mapping correctly. PC method, on the other hand, appears to over-handle the nonlinearity, hence, struggles to achieve a fast convergence rate when the underlying mapping is linear. For comparison convenience, a simple linear regression (LR) and third order polynomial regression (PR) model are also included.

Note that out of the three presented examples below, there are examples (the filter and low-noise amplifier) in which only a few numbers of samples were needed for the models to converge or nearly converge while in the other examples (the high-speed link), the models need much more training samples to do so. As discussed later, it is not due to the mapping being simple or

almost linear but rather due to the sensitivity of the output when varying the input. More discussions and arguments are presented along with the examples as that would highlight the pros and cons of using GP from a practical point of view.

4.4.1 Milimeter-wave filter

The first example is a 12GHz coupled-line bandpass filter [70]. There are twelve geometry-related design variables such as lengths, widths, and separations of the coupled lines, and stack-up features such as dielectric permittivity, loss tangent, etc. Figure 4.6 shows the geometry design and defined variables. In this experiment, it was assumed that there is perfect symmetry between the right and left half of the design. Which is why they share the same design variables. In reality, even if the nominal values are same on both sides, there is no reason for them to vary the same way when taking into account for stochastic effects. In this experiment, all 12 variables are varied up to 20% around their nominal values.

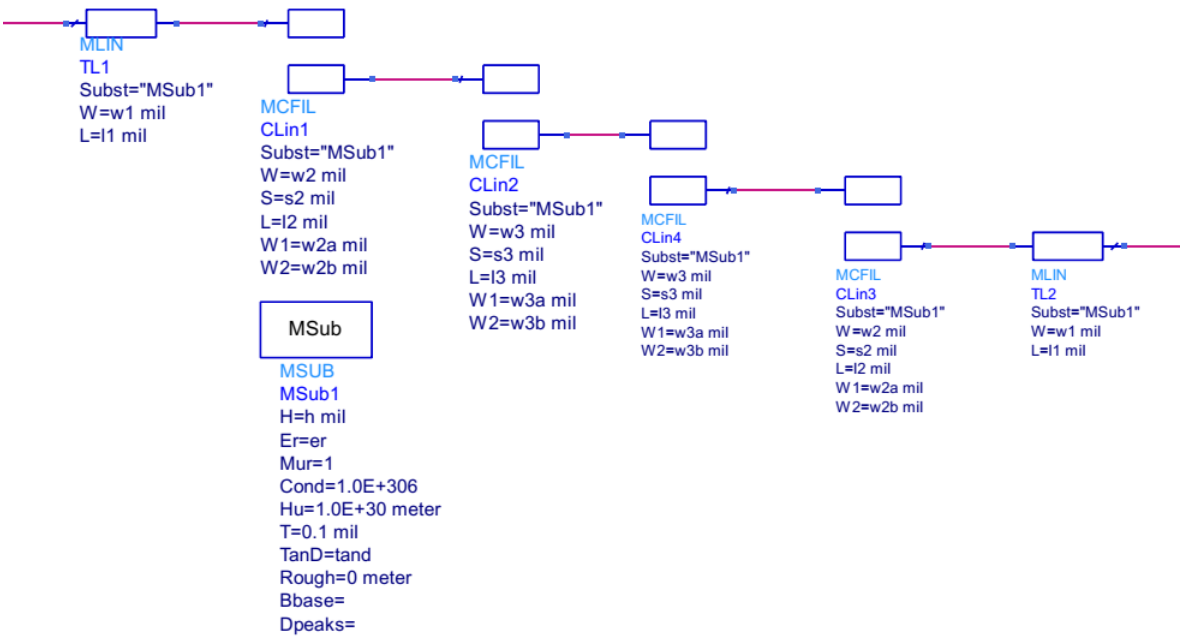


Figure 4.6: Filter design in Keysight ADS.

Figure 4.7 shows the insertion loss of the filter as design variables vary. To quantify the insertion loss of the filter, the center frequency (y_0), bandwidth

(y_1) and shape factor (y_2) are calculated. Surrogate models were created to predict these 3 figure of merits (FOM).

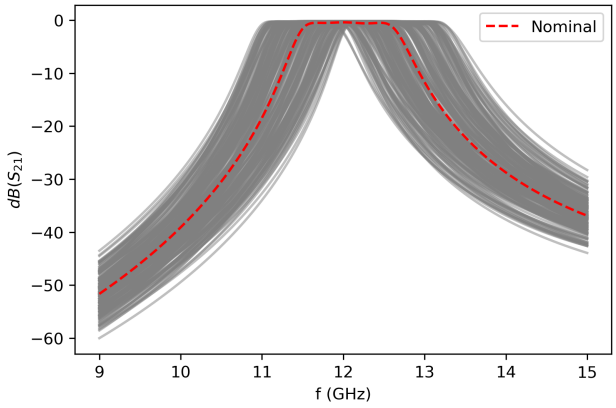
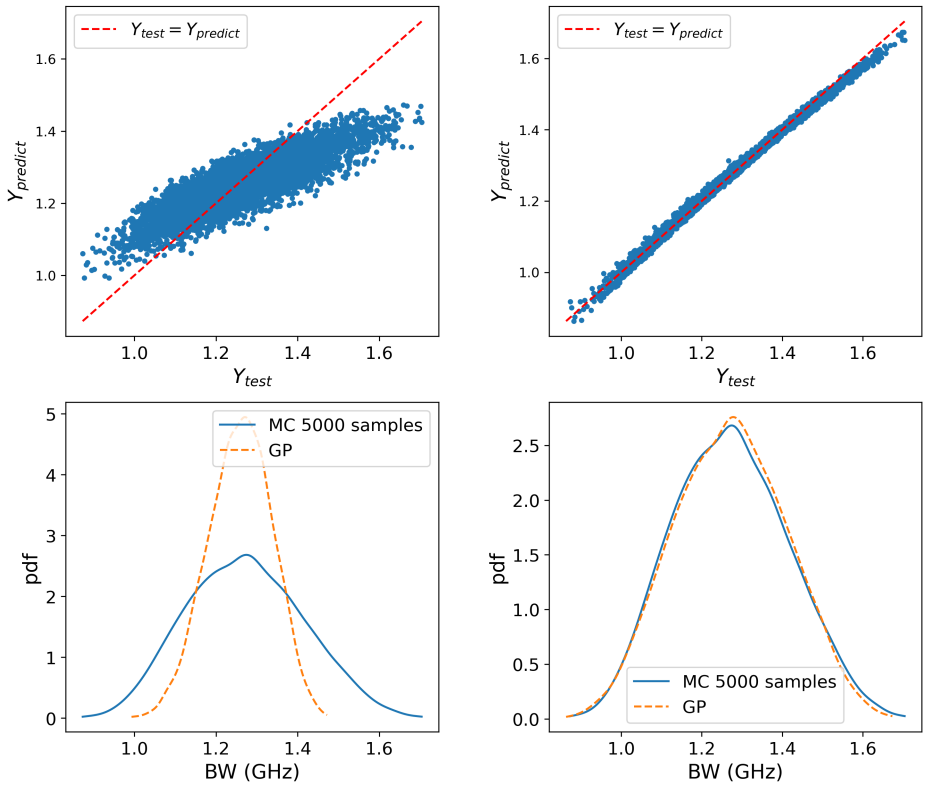


Figure 4.7: Filter insertion loss variations.

Three single-output GPs and one multi-output GP are trained on the same number, $N = 40$, of samples, then are tested on 5,000 test points. The GPs are constructed with zero mean function, Matern-3/2 kernel. Multi-output kernel is implemented using the linear model of coregionalization as explained in the previous section, all hyperparameters are learnt by optimizing the marginal likelihood with Adam optimizer. Since the size of our training samples are relatively small, direct inversion using the Cholesky decomposition is a reasonable choice for training, however, if the trained model is used to do inference on a larage number of test points, the cost will be significant, sparse GPs may be needed in such a case.

In general, multi-output GP converges much faster than single-output GPs, Figure 4.8 shows the result after 200 training epoch, the multi-output GP shows good agreement with 5,000 test point results while the single-output GP prediction is quite poor. However, if the single-output GP is trained upto 3,000 epoch, its performance is comparable to that of the multi-output GP.



(a) Single-output GP.

(b) Multi-output GP.

Figure 4.8: Filter bandwidth (BW) prediction when training with $N = 40$ samples.

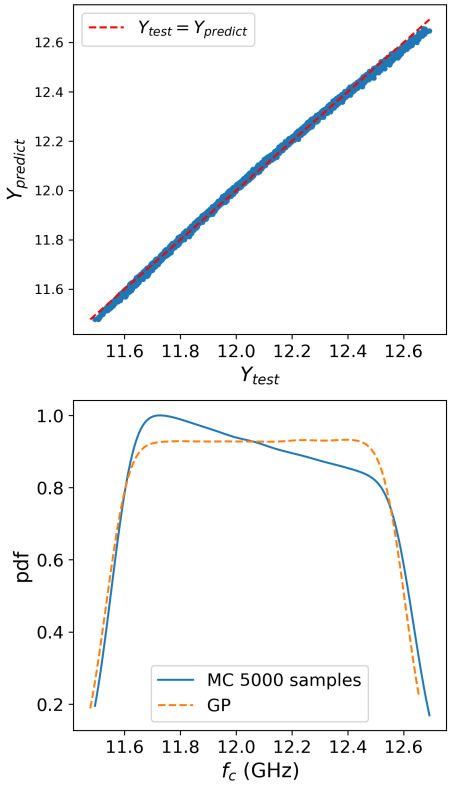
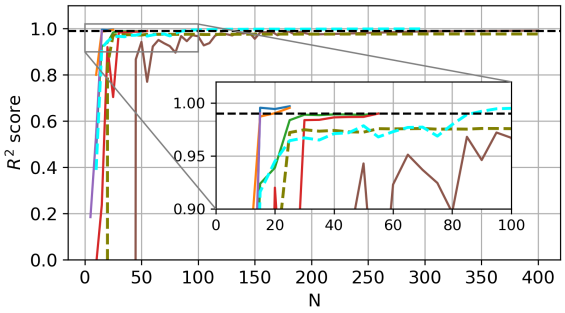


Figure 4.9: Center frequency (f_c) prediction when training with $N = 40$ samples.

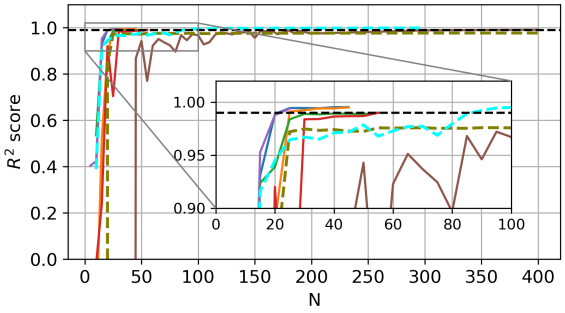
Figure 4.9 shows the prediction of the multi-output GP for f_c. Another single-output GP is trained to predict f_c, it reaches to similar performance after 3,000 training epoch.

Other surrogate models are now added for a comparative study against GP. Figure 4.10 shows the training process for each FOM. First, most models converge quite fast, only PC model requires a large number of samples to reach a validation R^2 score of 0.99. Second, single-out models have more difficulties learning the shape factor than the center frequency and the bandwidth as shown in Figure 4.10c, PLS and SVR models require more training samples to converge than MOPLS or MOGP. As mentioned before, MOPLS, MOGP and PC are multi-output models, there is a single R^2 score to determine their convergence. Table 4.1a shows the minimum training sample required for each model to reach 0.99 validation R^2 score. Though LR models did not reach 0.99 validation R^2 score even when all others have ($N = 400$), the fact that it was able to achieve a validation R^2 score higher than 0.96 for all 3 outputs indicates that eventhough this is a high dimensional prob-

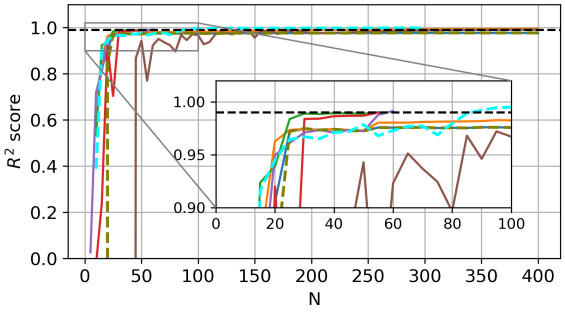
lem, the mapping between the design variables and the insertion loss FOMs are relatively linear. However, ML-based models are still more advantageous than the traditional linear model as they converge faster. In this particular example, SVR and PLS slightly outperforms GP models in terms of using the fewest training samples. An interesting observation is that single-output models appear to converge faster than multi-output models, indicating that there is little correlation between the outputs.



(a) Center frequency as output.



(b) Bandwidth as output.



(c) Shape factor as output.

Figure 4.10: Validation R^2 score during training with different numbers of training samples (N). The dash black line represents $R^2 = 0.99$.

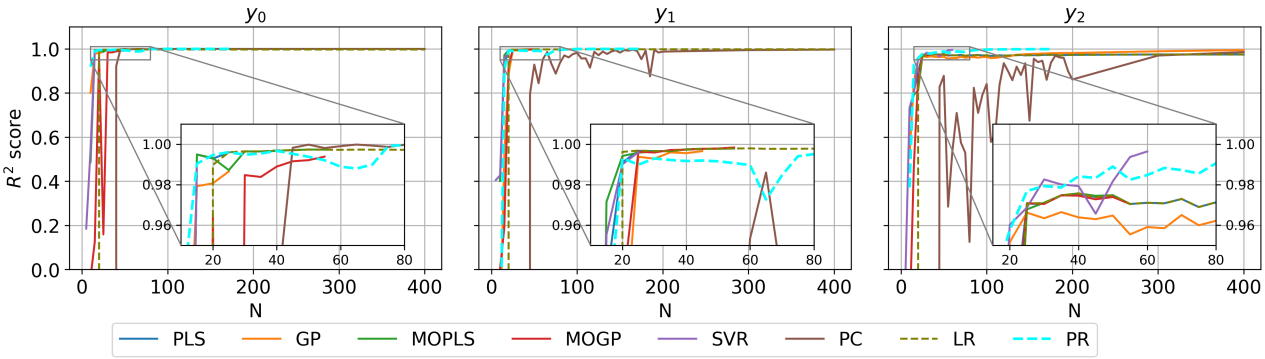


Figure 4.11: Test performance predicting center frequency (y_0), bandwidth (y_1) and shape factor (y_2) of a mm-wave bandpass filter, input variables are independent Gaussian distributed.

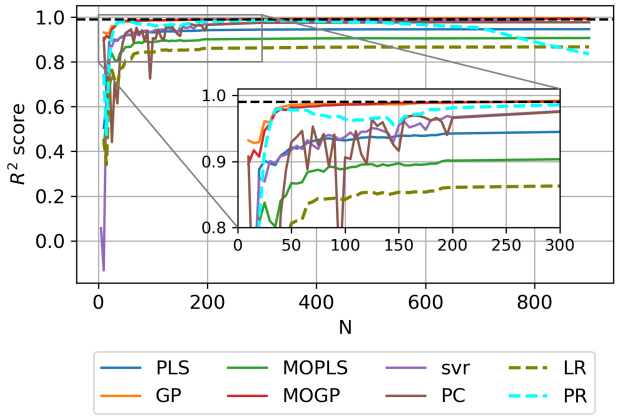
Figure 4.11 shows the test R^2 score for each output using different models trained by different N. The test performance of the models are consistent with their validation R^2 score, most models achieve test R^2 scores above 0.96.

4.4.2 High-Speed Link

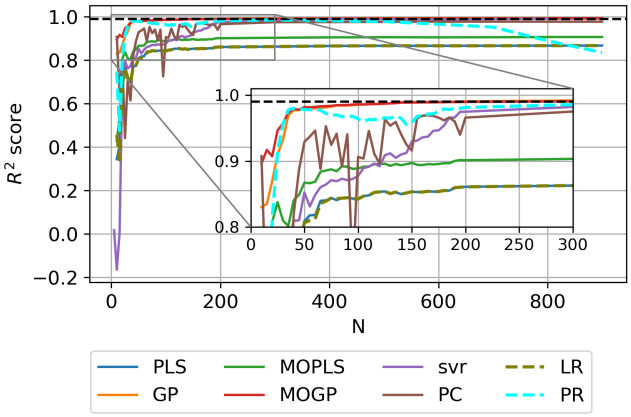
In this example, we consider a chip-to-chip, high-speed serial link model which involves 16 geometry-related design parameters associated with the stack-up and transmission lines [67, 68]. The link performance is quantified by looking at the eye opening at the receiver (RX), after equalization. These 16 parameters constitute an input parameter space of relatively high dimensionality such that a brute-force parameter sweep is intractable. A surrogate model that can quickly generate the eye openings from the geometric inputs is, therefore, imperative as it can be used for design optimization or uncertainty propagation.

Figure 4.12 shows the training result for eye width (y_0) and eye height (y_1). PLS (both single-output and multi-output) models never reaches validation R^2 score of 0.99. While GP-based models quickly approach the threshold, followed by SVR model, PC model has roughly the same convergence rate as SVR.

As shown in Table 4.1b, when N was varied up to 900, only GP models were able to consistently reach to 0.99 validation R^2 score for all outputs. Besides PLS models and LR, other models though did not reach the target value 0.99, their validation R^2 scores are all well above 0.96 for $N > 200$.

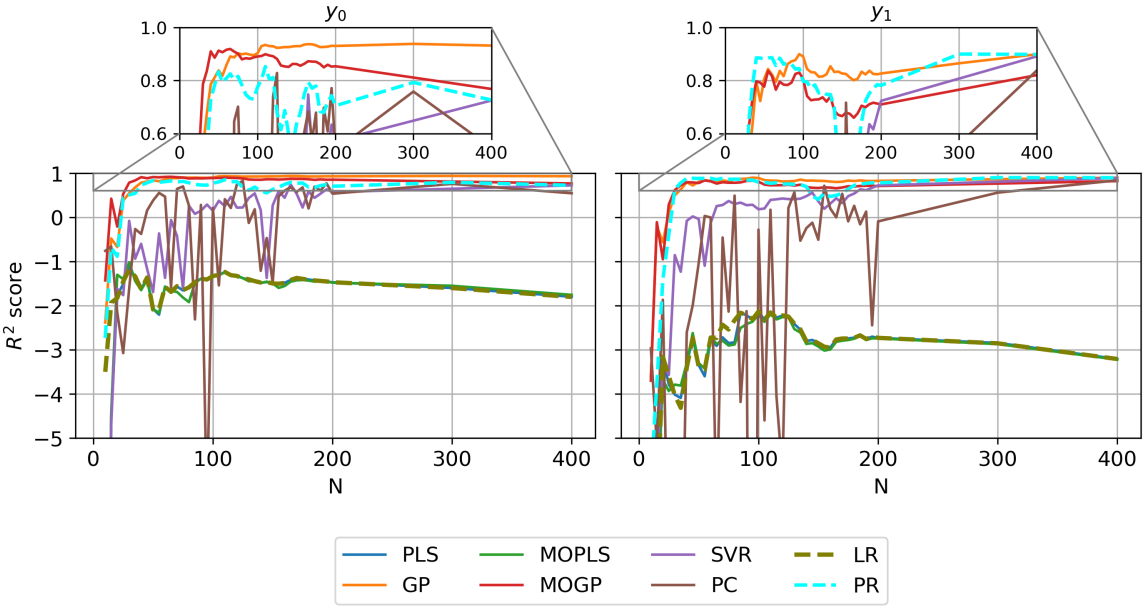


(a) Training result for eye height as output.

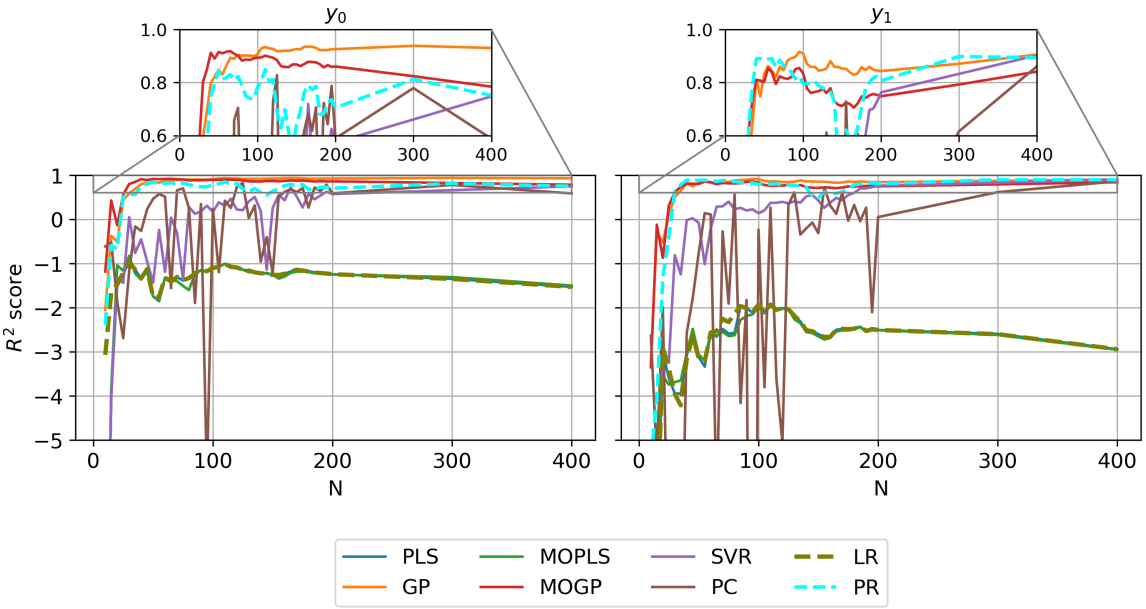


(b) Training result for eye width as output.

Figure 4.12: Validation R^2 score during training when varying N. Dash black line represents $R^2 = 0.99$

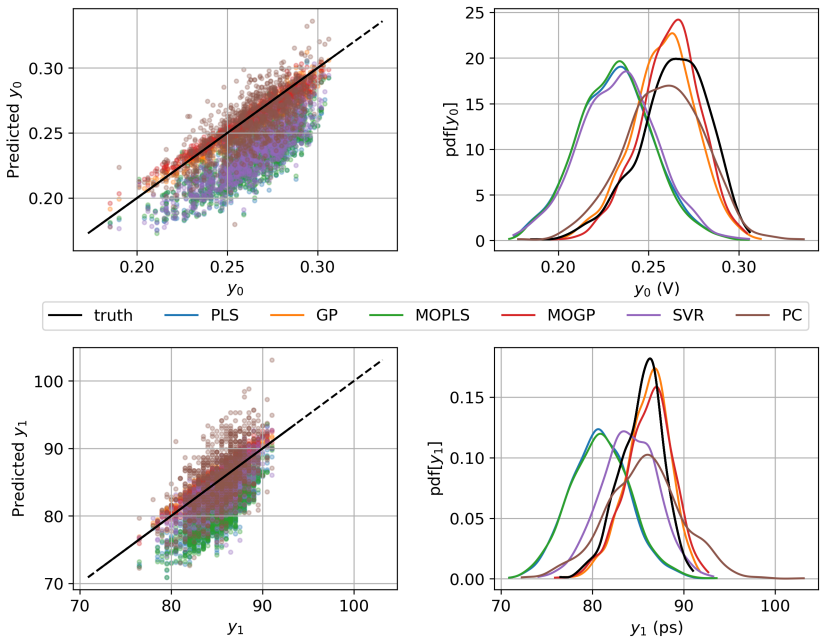


(a) Test performance with independent Gaussian distributed input variables

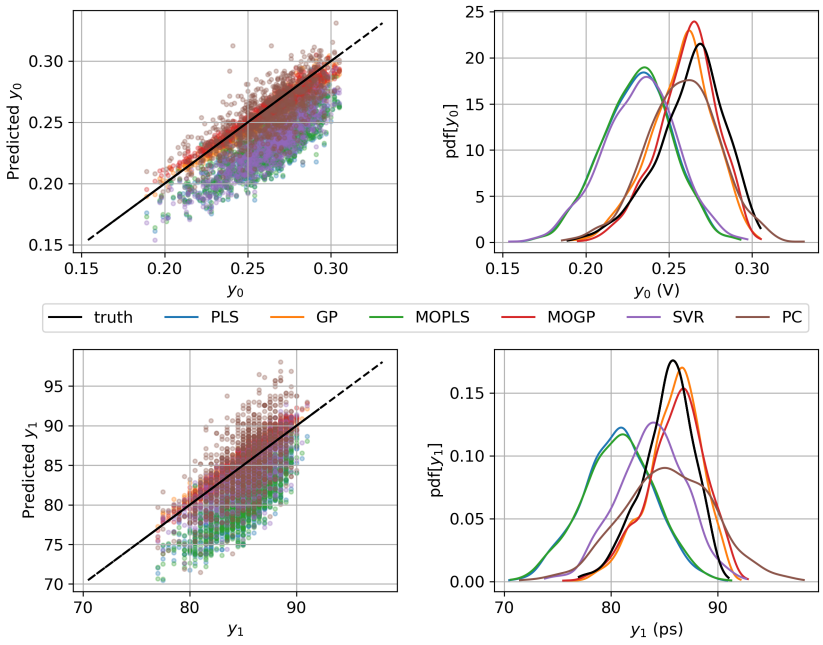


(b) Test performance with correlated Gaussian distributed input variables

Figure 4.13: Performance of trained models predicting eye height (y_0) and eye width (y_1)



(c) Predictive distribution by models trained with $N = 50$, input variables are independent Gaussian distributed.



(d) Predictive distribution by models trained with $N = 50$, input variables are correlated Gaussian distributed.

Figure 4.13: Performance of trained models predicting eye height (y_0) and eye width (y_1).

Table 4.1: Minimum training sample for each model to reach 0.99 validation R^2 score. Each row is for each output. N/A means the model did not reach 0.99 validation R^2 score within swept values of N.

(a) Filter example, N was varied up to $N_{max} = 400$.

	PLS	**GP**	**MOPLS**	**MOGP**	**SVR**	**PC**	**LR**	**PR**
y_0	15	20	55	55	15	400	N/A	90
y_1	25	20			25			
y_2	N/A	300			60			

(b) HSL example, N was varied up to $N_{max} = 900$.

	PLS	**GP**	**MOPLS**	**MOGP**	**SVR**	**PC**	**LR**	**PR**
y_0	N/A	300	N/A	300	N/A	N/A	N/A	N/A
y_1	N/A	185			800			

(c) LNA example, N was varied up to $N_{max} = 200$.

	PLS	**GP**	**MOPLS**	**MOGP**	**SVR**	**PC**	**LR**	**PR**
y_0	N/A	30	N/A	30	30	30	N/A	45
y_1	N/A	30			20			
y_2	N/A	30			30			

The trained models are then used to perform uncertainty propagation tests. A set of $1,000$ test samples are collected for each test. The inputs are sampled from two multivariate Gaussian distributions for two tests: one independent Gaussian distribution and the other one assuming correlations between the inputs. Figure 4.13 shows R^2 score for the two tests. PC models when trained with few samples appear to have generalization issue as its test R^2 score varies widely. Figure**??** and 4.13d depict the comparison of the predictive distribution of the outputs with the true distribution when all models are trained with $N = 50$ samples. The output distributions obtained from GP-based models prediction follow the true distribution quite well for both tests. As N increases, eventually that obtained from SVR and PC models also matches the true distribution well. Even with as many training points as $N = 1,000$, PLS models still could not reach to acceptable prediction accuracy as others.

Unlike the previous example, LR and PLS cannot learn the data well while PR can. This suggests that the underlying mapping is nonlinear, which explains why PLS is unable to reach higher R^2 score during training, thus failing to predict the output when the input is sampled from a distribution different

from the one that generated the training data. This example illustrates that with a good surrogate model, it is possible to perform millions of direct MC evaluations for uncertainty propagation analysis.

4.4.3 Low noise amplifier

Finally, a 2-stage low noise amplifier (LNA) designed for a carrier frequency of 8GHz is studied. The design comprises of two amplifier stages sandwiched by three matching networks. For this variability analysis study, one variable from the input matching network, three variables from the middle matching network and one variable from the second amplifying stage are varied. Three quantities of interest at operating frequency include total gain, output return loss and output noise figure are modeled.

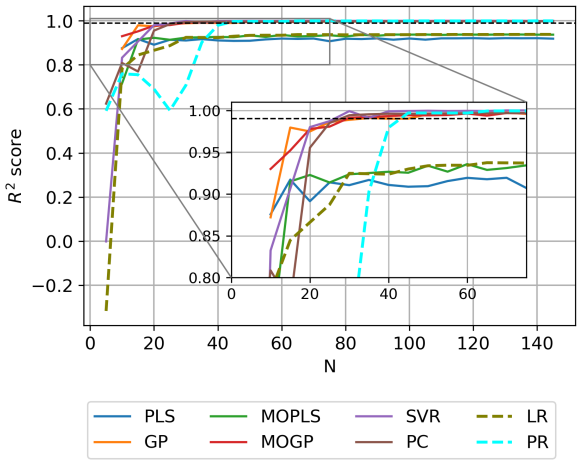


Figure 4.14: Validation R^2 score during training the LNA model when varying N. Dash black line represents $R^2 = 0.99$

The variability analysis was performed on the optimized design, hence, even 20% variation from their optimal values of the design variables does not yield large variations in output quantities (only about 5%). Therefore, similar to the filter example, the input - output mapping is relatively simple for all models to capture with a small number of samples as shown in Figure 4.14. This is well expected because the number of samples needed to generally sufficient cover the output space depends on the variance of the outputs (generally, to cover the larger space, we need more points). However, GP models are among models that reach convergence with the fewest number of

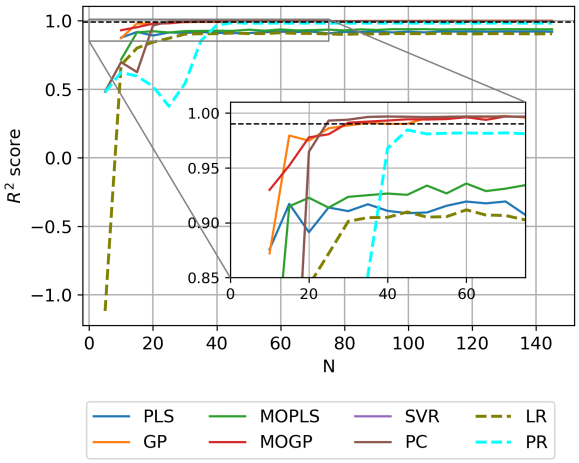


Figure 4.15: Testing R^2 score for the LNA model predicting the gain. Dash black line represents $R^2 = 0.99$

training samples compared to other traditional methods. It is worth noting that, inheritly linear models such as LR and PLS can only explain the linear part of the data and hence never reach the convergence defined by the 0.99 threshold.

Figure 4.15 shows the test performance of the trained models on more than 3,000 unseen samples. It is noticeable that PR model, though converged during training, slightly underperforms as its prediction accuracy was below 0.99 while all other ML methods not only converged with fewer number of samples but also yield higher accuracy in predictions.

4.5 Conclusion

In this chapter, GPR was introduced as a surrogate modeling method. The implementation of GP is simple though the concept behind it could potentially confusing. However, as explained above, GP is just a collection of points that obey the multivariate Gaussian distribution. Once the mean function and covariance matrix are known, getting the predictions from GP is no different than sampling values from a multidimensional Gaussian ditribution. The training of a GP can be simplified to the covariance matrix computation and optimizing the marginal likelihood for learning the kernel hyper-parameters. Both of which could potentially pose an expensive cost for big data applications. However, in applications where a single function

Table 4.2: Summary of reviewed surrogate models.

Method	**Advantages**	**Disadvantages**
PLS	Fast training, fast inference.	Only linear problems.
MOPLS		
GP	Can handle nonlinear problems, hyper-parameters learnt directly from data	Training time scales with $\mathcal{O}(N^3)$
MOGP		
SVR	Can handle nonlinear problems.	Require hyper-parameters tuning (cross-validation, Bayesian optimization etc.), training time scales with $\mathcal{O}(N^2)$ in the best case scenario
PC	Can handle nonlinear problems, straightforward implementation, no hyper-parameters	Input distribution dependence, training time scales with $\mathcal{O}(N^2)$

evaluation is highly expensive, the number of training samples needs to be as few as possible, this cost is acceptable. In the presented examples, data for training were randomly sampled. The results show that multi-output GP outperforms single-output GP for multiple output systems. The former converges 10 times faster than the latter, because correlations and dependence between outputs when learnt altogether could provide more information for the model to converge.

Table 4.2 summarizes the key advantages and drawbacks of the methods studied above. Via a high-speed channel, a microwave filter, and a low noise amplifier example, it was demonstrated that all of the reviewed methods show good agreement with the true distribution of the test data though some methods would require more training data than others. PLS models are extremely fast and straightforward to implement but they may fail to capture strongly nonlinear mappings. The takeaway key here is that MOGP is generally recommended to build surrogate models. MOGP though takes more time to train, consistently yields good results while using the fewest number of training samples. GP models mainly suffer from numerical inefficiency only when a large dataset involves, making it not favorable in big data applications but very suitable to model problems where only a small

number of training data is available due to the expensive cost for data collection such as signal integrity applications involving fullwave EM simulations as the scope of this thesis.

Training data is crucial to the convergence rate of surrogate models. Especially for GP models, training samples if chosen appropriately would quickly minimize the overall uncertainty in the model prediction. Potential extension of this work should seek to investigate which sampling method would provide the best convergence for GP models. Training speed of a GP could use some improvement as $\mathcal{O}(N^3)$ complexity for training process sometimes is unacceptable when the designers want to make use of the readily available data from past designs or from other analyses. There are reported works [95] on exploiting structural covariance matrices to approximate the calculation of the NLL in order to speed up the optimization process as well as the prediction step. These could be the next steps to further improve GP modeling for signal integrity analysis.

Chapter 5

CONCLUSION AND FUTURE WORK

Overall, machine learning methods are expressive and flexible, capable of handling different types of input - output mappings. This thesis explores the use of recurrent neural network to represent the nonlinear dynamical behavior high-speed channel circuits. Prior works focus on using the output-feedback RNN topology [19,96–98] which implements an explicit dependency of current timestep output on past timestep outputs. This explicit dependency is a severe drawback that forces the prediction to be done sequentially, which makes the models built with this topology incompatible with a channel simulator. In this thesis, the Elman RNN topology was deployed to remove that explicit dependency, allowing time parallelization when getting the output voltage, making the proposed RNN model highly suitable for high-speed channel simulation. If the computer memory is enough, response of millions of time steps can be obtained at once. In addition, for the first time, a combination of FNN and RNN was used to create a parameterized model. In particular, a model of the DFE circuit with variable tap values was created, completely protecting the IP of the IC vendor whilst giving maximum flexibility to the link designer to tweak the equalization settings when designing the high-speed link.

Alternatively to using a neural network, Volterra-Laguerre (VL) theory is shown in Chapter 3 to be as effective as RNN for the time-domain waveform prediction task. The framework has the advantage of sharing many similarities with the linear time-invariant system theory. In this work, the VL framework is implemented as a dynamical system. First, a filtering process is applied to the input via a bank of Laguerre filters to obtain first order Laguerre responses. Then high order of such responses are generated by taking high dimensional tensor product among them. A linear combination of these high order responses is the final response. The examples demonstrate the successful modeling of DFE circuits using this approach. The VL model

was compared against an IBIS model, the current technique widely used in industry, and shows a better agreement with transistor level simulations. Extraction of Laguerre coefficients has been reported in [52, 99]. By using a neural network to extract Laguerre coefficients, [52, 99] obtains a solution from a non-convex loss function, the effort to parametrize Laguerre coefficients obtained using this approach was not successful. The interpolation mapping between the control variables (specifically, the DFE taps) and the Laguerre coefficients did not converge. The model obtained from the interpolants for unseen tap values fail to generate the correct output waveform when compared to a transient simulation. The implementation in this thesis is an improvement over that in [52,99]. First, the proposed extraction process in this thesis allows multiple datasets from different time scales to be used for the model identification while the method in [52,99] requires unique, uniform timestep data. Second, by extracting the Laguerre coefficients directly using a least-square setup, the coefficients were obtained from minimizing a convex loss function. This is significant because the uniqueness of the solution is guaranteed thanks to convexity, which opens the success of parameterizing the Laguerre coefficients to create a tunable model just as an FNN - RNN combination could in Chapter 2.

Lastly, a non-parametric surrogate model using Gaussian Process is proposed in Chapter 4 to replace tedious simulations when the goal is to assess the performance of a complicated system via some figures of merit. Instead of tuning the hyper-parameter of the GP model by cross-validation or using educated guesses when training the GP model, a full Bayesian treatment of the hyper-parameters inside the model was implemented so that the GP model is robust and adaptive to different problems. It can learn to adjust its hyper-parameters using the available data fed to it, little to none domain expertise is required for a guaranteed convergence, though putting domain knowledge into the model through its prior will help speed up the training convergence. The computational cost for this full Bayesian treatment might be a concern for big data applications. However, this cost is justified and worthy for signal integrity and electromagnetics problems where a full-wave simulation usually involves.

X-parameter was reviewed and shown to have potential use in the Volterra-Laguerre framework, however, due to the limitation of existing nonlinear vector network analyzer, collecting multi-LSOP X-parameter for model con-

struction is prohibited. However, when the equipment is available, with the large amount of data associated with multi-LSOP X-parameters, machine learning methods will be the first candicate approach to extract useful information for model construction. Future extended work of this thesis should look into this direction for further improvement of I/O buffer modeling and the application of X-parameter in channel simulations.

Modeling is a never-ending task, the effort required to put into it will only grow over time as the complexity and the level of integration in modern electronics devices get higher and higher. With the advancement of machine learning and artificial intelligence, they are shown to be effective to help completing such an important and fascinating task.

REFERENCES

[1] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," in *2009 International Conference on Field-Programmable Technology*, Dec 2009, pp. 190–198.

[2] B. Gustavsen and A. Semlyen, "Rational approximation of frequency-domain responses by vector fitting," *IEEE Transactions on Power Delivery*, vol. 14, no. 3, pp. 1052–1061, July 1999.

[3] S. Lefteriu, "New approaches to modeling multi-port scattering parameters," M.S. thesis, Rice University, 2009.

[4] "IBIS Open Forum," Accessed: 2020-12-16. [Online]. Available: http://www.ibis.org/

[5] I. S. Stievano, I. A. Maio, and F. G. Canavero, "M/spl pi/log, macromodeling via parametric identification of logic gates," *IEEE Transactions on Advanced Packaging*, vol. 27, no. 1, pp. 15–23, 2004.

[6] Y. Abu-Mostafa, M. Magdon-Ismail, and H. Lin, *Learning from Data: A Short Course*. AMLBook.com, 2012. [Online]. Available: http://amlbook.com,http://work.caltech.edu/textbook.html,/bib/abu-mostafa/abu2012learning/Yaser%20S.%20Abu-Mostafa%2C%20Malik%20Magdon-Ismail%2C%20Hsuan-Tien%20Lin-Learning%20From%20Data_%20A%20short%20course-AMLBook.com%20%282012%29.pdf

[7] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[8] M. Jamil and X. Yang, "A literature survey of benchmark functions for global optimization problems," *CoRR*, vol. abs/1308.4008, 2013. [Online]. Available: http://arxiv.org/abs/1308.4008

[9] T. Hastie, R. Tibshirani, and M. Wainwright, *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman amp; Hall/CRC, 2015.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2627435.2670313

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[12] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 786–795, March 2015.

[13] W. D. Guo, J. H. Lin, C. M. Lin, T. W. Huang, and R. B. Wu, "Fast methodology for determining eye diagram characteristics of lossy transmission lines," *IEEE Transactions on Advanced Packaging*, vol. 32, no. 1, pp. 175–183, Feb 2009.

[14] L. Zhu and N. Laptev, "Deep and confident prediction for time series at uber," in *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, vol. 00, Nov. 2018. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ICDMW.2017.19 pp. 103–110.

[15] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[16] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969033.2969173 pp. 3104–3112.

[17] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple object recognition with visual attention," *CoRR*, vol. abs/1412.7755, 2014. [Online]. Available: http://arxiv.org/abs/1412.7755

[18] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[19] W. Liu, W. Na, L. Zhu, and Q. J. Zhang, "A review of neural network based techniques for nonlinear microwave device modeling," in *2016 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*, July 2016, pp. 1–2.

[20] H. Yu, T. Michalka, M. Larbi, and M. Swaminathan, "Behavioral modeling of tunable i/o drivers with preemphasis including power supply noise," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 233–242, 2020.

[21] A. Beg, P. W. C. Prasad, M. M. Arshad, and K. Hasnain, "Using recurrent neural networks for circuit complexity modeling," in *2006 IEEE International Multitopic Conference*, Dec 2006, pp. 194–197.

[22] W.-T. Hsieh, C.-C. Shiue, and C. N. J. Liu, "A novel approach for high-level power modeling of sequential circuits using recurrent neural networks," in *2005 IEEE International Symposium on Circuits and Systems*, May 2005, pp. 3591–3594 Vol. 4.

[23] Z. Chen, M. Raginsky, and E. Rosenbaum, "Verilog-A compatible recurrent neural network model for transient circuit simulation," in *2017 IEEE 26th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, Oct 2017, pp. 1–3.

[24] N. Ambasana, G. Anand, D. Gope, and B. Mutnury, "S-parameter and frequency identification method for ann-based eye-height/width prediction," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 7, no. 5, pp. 698–709, May 2017.

[25] C. Goay, P. Goh, N. Ahmad, and M. Ain, "Eye-height/width prediction using artificial neural networks from S-Parameters with vector fitting," *Journal of Engineering Science and Technology*, vol. 13, no. 3, pp. 625–639, Mar. 2018.

[26] T. Lu, J. Sun, K. Wu, and Z. Yang, "High-Speed Channel Modeling With Machine Learning Methods for Signal Integrity Analysis," *IEEE Transactions on Electromagnetic Compatibility*, pp. 1–8, 2018.

[27] G. Xue-lian, C. Zhen-nan, F. Nan, Z. Xiao-yu, and H. Jian-hong, "An artificial neural network model for s-parameter of microstrip line," in *2013 Asia-Pacific Symposium on Electromagnetic Compatibility (APEMC)*, May 2013, pp. 1–4.

[28] X. Zhang, Y. Cao, and Q. J. Zhang, "A combined transfer function and neural network method for modeling via in multilayer circuits," in *2008 51st Midwest Symposium on Circuits and Systems*, Aug 2008, pp. 73–76.

[29] T. Nguyen and J. E. Schutt-Aine, "A pseudo-supervised machine learning approach to broadband lti macro-modeling," in *2018 IEEE International Symposium on Electromagnetic Compatibility and 2018 IEEE Asia-Pacific Symposium on Electromagnetic Compatibility (EMC/APEMC)*, May 2018, pp. 1018–1021.

[30] F. Feng, C. Zhang, J. Ma, and Q. J. Zhang, "Parametric modeling of em behavior of microwave components using combined neural networks and pole-residue-based transfer functions," *IEEE Transactions on Microwave Theory and Techniques*, vol. 64, no. 1, pp. 60–77, Jan. 2016.

[31] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a.html pp. 315–323.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[33] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[34] T. Tieleman and G. Hinton, "Rmsprop gradient optimization," *URL http://www. cs. toronto. edu/tijmen/csc321/slides/lecture_ slides_ lec6. pdf*, 2014.

[35] J. L. Elman, "Finding structure in time," *COGNITIVE SCIENCE*, vol. 14, no. 2, pp. 179–211, 1990.

[36] R. Pascanu, T. Mikolov, and Y. Bengio, "On the Difficulty of Training Recurrent Neural Networks," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, pp. III–1310–III–1318.

[37] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, no. 4, pp. 490–501, Dec 1990.

[38] I. Sutskever, "Training recurrent neural networks," *University of Toronto, Toronto, Ont., Canada*, 2013.

[39] C. Olah, "Understanding LSTM Networks." [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[40] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[41] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," *CoRR*, vol. abs/1506.03099, 2015. [Online]. Available: http://arxiv.org/abs/1506.03099

[42] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of Atmospheric Sciences*, 1963.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[44] M. Schetzen, *The Volterra and Wiener Theories of Nonlinear Systems*. Melbourne, FL, USA: Krieger Publishing Co., Inc., 2006.

[45] N. Wiener, *Nonlinear problems in random theory*. MIT Press, Cambridge, MA, 1958.

[46] S. Boyd, L. O. Chua, and C. A. Desoer, "Analytical foundations of volterra series," *IMA J Math Control Info*, vol. 1, no. 3, pp. 243–282, jan.

[47] W. J. Rugh, *Nonlinear system theory: the Volterra/Wiener approach*. Johns Hopkins University Press, 1981.

[48] C. Xin, "Radio frequency circuits for wireless receiver front-ends," Ph.D. dissertation, Texas A&M University, 2004.

[49] L. Li and S. Billings, "Estimation of generalized frequency response functions for quadratically and cubically nonlinear systems," *Journal of Sound and Vibration*, vol. 330, no. 3, pp. 461 – 470, 2011.

[50] L. Li and S. Billings, "Volterra series truncation and reduction in the frequency domain for weakly nonlinear system," University of Sheffield, Tech. Rep., 2006.

[51] P. J. Lawrence, "Estimation of the volterra functional series of a nonlinear system using frequency-response data," *IEE Proceedings D - Control Theory and Applications*, vol. 128, no. 5, pp. 206–210, Sep. 1981.

[52] X. Wang, T. Nguyen, and J. E. Schutt-Aine, "Laguerre-volterra feed-forward neural network for modeling pam-4 high-speed links," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 10, no. 12, pp. 2061–2071, 2020.

[53] B. W. Israelsen and D. A. Smith, "Generalized laguerre reduction of the volterra kernel for practical identification of nonlinear dynamic systems," *CoRR*, vol. abs/1410.0741, 2014. [Online]. Available: http://arxiv.org/abs/1410.0741

[54] Mengtao Yuan, T. K. Sarkar, Baek Ho Jung, Zhong Ji, and M. Salazar-Palma, "Use of discrete laguerre sequences to extrapolate wide-band response from early-time and low-frequency data," *IEEE Transactions on Microwave Theory and Techniques*, vol. 52, no. 7, pp. 1740–1750, July 2004.

[55] *Discrete-time MPC Using Laguerre Functions*. London: Springer London, 2009, ch. 3, pp. 85–148. [Online]. Available: https://doi.org/10.1007/978-1-84882-331-0_3

[56] S. Boyd, Y. Tang, and L. Chua, "Measuring volterra kernels," *IEEE Transactions on Circuits and Systems*, vol. 30, no. 8, pp. 571–577, Aug 1983.

[57] P. Lawrence, "Estimation of the volterra functional series of a nonlinear system using frequency-response data," *IEE Proceedings D (Control Theory and Applications)*, vol. 128, pp. 206–210(4), September 1981.

[58] T. Nguyen, J. E. Schutt-Aine, and Y. Chen, "Volterra kernels extraction from frequency-domain data for weakly non-linear circuit time-domain simulation," in *2017 IEEE Radio and Antenna Days of the Indian Ocean (RADIO)*, Sept 2017, pp. 1–2.

[59] X. Y. Z. Xiong, L. J. Jiang, J. E. Schutt-Aine, and W. C. Chew, "Volterra series-based time-domain macromodeling of nonlinear circuits," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 7, no. 1, pp. 39–49, Jan 2017.

[60] D. E. Root, J. Verspecht, J. Horn, and M. Marcu, *X-Parameters: Characterization, Modeling, and Design of Nonlinear RF and Microwave Components*. Cambridge University Press, 2013.

[61] J. Verspecht and D. E. Root, "Polyharmonic distortion modeling," *IEEE Microwave Magazine*, vol. 7, no. 3, pp. 44–57, June 2006.

[62] C. R. Paul, *Introduction to Electromagnetic Compatibility (Wiley Series in Microwave and Optical Engineering)*. USA: Wiley-Interscience, 2006.

[63] H. Ma, E. Li, A. C. Cangellaris, and X. Chen, "Comparison of Machine Learning Techniques for Predictive Modeling of High-Speed Links," in *2019 IEEE 28th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, 2019, pp. 1–3.

[64] T. Lu, J. Sun, K. Wu, and Z. Yang, "High-Speed Channel Modeling With Machine Learning Methods for Signal Integrity Analysis," *IEEE Transactions on Electromagnetic Compatibility*, pp. 1–8, 2018.

[65] R. Trinchero and F. G. Canavero, "Modeling of eye diagram height in high-speed links via support vector machine," in *2018 IEEE 22nd Workshop on Signal and Power Integrity (SPI)*, 2018, pp. 1–4.

[66] R. Trinchero, M. A. Dolatsara, K. Roy, M. Swaminathan, and F. G. Canavero, "Design of High-Speed Links via a Machine Learning Surrogate Model for the Inverse Problem," in *2019 Electrical Design of Advanced Packaging and Systems (EDAPS)*, 2019, pp. 1–3.

[67] H. Ma, E.-P. Li, A. C. Cangellaris, and X. Chen, "High-speed link design optimization using machine learning SVR-AS method," in *2020 IEEE 28th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, Oct. 2020.

[68] H. Ma, E.-P. Li, A. C. Cangellaris, and X. Chen, "Support vector regression-based active subspace (SVR-AS) modeling of high-speed links for fast and accurate sensitivity analysis," *IEEE Access*, vol. 8, pp. 74 339–74 348, 2020.

[69] P. G. Constantine, *Active Subspaces: Emerging Ideas for Dimension Reduction in Parameter Studies*. USA: Society for Industrial and Applied Mathematics, 2015.

[70] T. Nguyen and J. Schutt-Aine, "Gaussian Process surrogate model for variability analysis of RF circuits," in *2020 IEEE Electrical Design of Advanced Packaging and Systems (EDAPS)*, 2020, pp. 1–3.

[71] M. Larbi, R. Trinchero, F. G. Canavero, P. Besnier, and M. Swaminathan, "Analysis of Parameter Variability in an Integrated Wireless Power Transfer System via Partial Least-Squares Regression," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 10, no. 11, pp. 1795–1802, 2020.

[72] M. Larbi, R. Trinchero, F. G. Canavero, P. Besnier, and M. Swaminathan, "Analysis of Parameter Variability in Integrated Devices by Partial Least Squares Regression," in *2020 IEEE 24th Workshop on Signal and Power Integrity (SPI)*, 2020, pp. 1–4.

[73] R. Trinchero and F. G. Canavero, "Combining LS-SVM and GP Regression for the Uncertainty Quantification of the EMI of Power Converters Affected by Several Uncertain Parameters," *IEEE Transactions on Electromagnetic Compatibility*, vol. 62, no. 5, pp. 1755–1762, 2020.

[74] R. Trinchero, M. Larbi, M. Swaminathan, and F. G. Canavero, "Statistical Analysis of the Efficiency of an Integrated Voltage Regulator by means of a Machine Learning Model Coupled with Kriging Regression," in *2019 IEEE 23rd Workshop on Signal and Power Integrity (SPI)*, 2019, pp. 1–4.

[75] J. Slim, F. Rathmann, A. Nass, H. Soltner, R. Gebel, J. Pretz, and D. Heberling, "Polynomial Chaos Expansion method as a tool to evaluate and quantify field homogeneities of a novel waveguide RF Wien filter," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 859, pp. 52–62, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168900217303807

[76] M. A. Dolatsara, A. Varma, K. Keshavan, and M. Swaminathan, "A Modified Polynomial Chaos Modeling Approach for Uncertainty Quantification," in *2019 International Applied Computational Electromagnetics Society Symposium (ACES)*, 2019, pp. 1–2.

[77] X. Chen, M. Qiu, J. E. Schutt-Aine, and A. C. Cangellaris, "Stochastic LIM for transient simulation of circuits with uncertainties," in *2014 IEEE 23rd Conference on Electrical Performance of Electronic Packaging and Systems*, 2014, pp. 29–32.

[78] Z. Zhang, T. A. El-Moselhy, I. M. Elfadel, and L. Daniel, "Stochastic Testing Method for Transistor-Level Uncertainty Quantification Based on Generalized Polynomial Chaos," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 10, pp. 1533–1545, 2013.

[79] C. Cui and Z. Zhang, "High-Dimensional Uncertainty Quantification of Electronic and Photonic IC with Non-Gaussian Correlated Process Variations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[80] C. Cui and Z. Zhang, "Stochastic Collocation With Non-Gaussian Correlated Process Variations: Theory, Algorithms, and Applications," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 9, no. 7, pp. 1362–1375, 2019.

[81] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[82] D. K. Duvenaud, H. Nickisch, and C. E. Rasmussen, "Additive Gaussian Processes," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 226–234. [Online]. Available: http://papers.nips.cc/paper/4221-additive-gaussian-processes.pdf

[83] H. M. Torun and M. Swaminathan, "Black-box optimization of 3d integrated systems using machine learning," in *2017 IEEE 26th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, Oct 2017, pp. 1–3.

[84] H. M. Torun, M. Swaminathan, A. K. Davis, and M. L. F. Bellaredj, "A global bayesian optimization algorithm and its application to integrated system design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 4, pp. 792–802, April 2018.

[85] V. I. Paulsen and M. Raghupathi, *An Introduction to the Theory of Reproducing Kernel Hilbert Spaces*, ser. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2016.

[86] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[87] M. A. Alvarez, L. Rosasco, and N. D. Lawrence, "Kernels for Vector-Valued Functions: A Review," *Foundations and Trends in Machine Learning*, vol. 4, no. 3, pp. 195–266, 2012. [Online]. Available: http://dx.doi.org/10.1561/2200000036

[88] G. Angiulli, M. Cacciola, and M. Versaci, "Microwave Devices and Antennas Modelling by Support Vector Regression Machines," *IEEE Transactions on Magnetics*, vol. 43, no. 4, pp. 1589–1592, April 2007.

[89] R. Trinchero, M. Larbi, H. M. Torun, F. G. Canavero, and M. Swaminathan, "Machine Learning and Uncertainty Quantification for Surrogate Models of Integrated Devices With a Large Number of Parameters," *IEEE Access*, vol. 7, pp. 4056–4066, 2019.

[90] A. K. Prasad, M. Ahadi, B. S. Thakur, and S. Roy, "Accurate polynomial chaos expansion for variability analysis using optimal design of experiments," in *2015 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*, 2015, pp. 1–4.

[91] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, Aug. 2004.

[92] V. N. Vapnik, *The nature of statistical learning theory*. Springer science & business media, 1995.

[93] D. Xiu, "Fast Numerical Methods for Stochastic Computations: A Review," in *Communications in computational physics*, vol. 5, no. 2-4, 2009, pp. 242–272.

[94] D. Xiu and G. E. Karniadakis, "The Wiener–Askey Polynomial Chaos for Stochastic Differential Equations," *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 619–644, 2002. [Online]. Available: https://doi.org/10.1137/S1064827501387826

[95] M. van der Wilk, "Sparse Gaussian Process Approximations and Applications," Ph.D. dissertation, University of Cambridge, 2019.

[96] Z. Aimin, Z. Hang, L. Hong, and C. Degui, "A recurrent neural networks based modeling approach for internal circuits of electronic devices," in *2009 20th International Zurich Symposium on Electromagnetic Compatibility*, Jan 2009, pp. 293–296.

[97] H. Yu, T. Michalka, M. Larbi, and M. Swaminathan, "Behavioral Modeling of Tunable I/O Drivers With Preemphasis Including Power Supply Noise," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 233–242, 2020.

[98] B. Li, B. Jiao, M. Huang, R. Mayder, and P. Franzon, "Improved System Identification Modeling for High-speed Receiver," in *2019 IEEE 28th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, 2019, pp. 1–3.

[99] T. Nguyen, X. Wang, X. Chen, and J. Schutt-Aine, "A deep learning approach for volterra kernel extraction for time domain simulation of weakly nonlinear circuits," in *2019 IEEE 69th Electronic Components and Technology Conference (ECTC)*, 2019, pp. 1889–1896.