MACHINE LEARNING APPROACH FOR CASCADE-ABLE NONLINEAR TRANSCEIVER MODELING AND HIGH SPEED LINK SIMULATION

BY

YIXUAN ZHAO

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, 2022

Urbana, Illinois

ABSTRACT

With the rapid developments in integrated circuit technology, the data rates of chip-to-chip communication are fast approaching several tens of Gb/s. While the desire for massive data-exchange is satisfied as a result of transceiver links operating at high frequency, signal integrity (SI) issues emerge due to short switching times. To identify and resolve these problems early in the production cycle, SI simulations such as time-domain transient analysis are incorporated in pre- and post-layout design stages. For efficiency concern, it is often desired to use accurate and efficient black-box macromodels of components on board instead of their SPICE-like representations. The motivation rests in the nonlinear nature of the transceivers, which oftentimes requires multiple Newton-Raphson iterations before convergence can be achieved. This thesis is meant to contribute a small part to the enormous amount of effort of the behavior modeling community in the quest for computationally efficient methods capable of handling high speed link (HSL) simulation of nonlinear devices and systems using machine learning methods. Specifically, this work reports a feed forward-neural network (FNN) approach with finite memory neurons to model nonlinear transistor level buffers. After proper training, the FNN models can be cascaded with various channels characterized by either their geometrical or scattering parameters. At each cascading node, a FNN model is applied to predict the corresponding voltage waveform and forward that prediction along the link as input for the next available model. Compared to the industrial standard models like SPICE and IBIS, HSL simulation done through FNN models does not involve complicated converging iterations nor does it requires substantial domain knowledge. Furthermore, we demonstrated that by overlaying the high-correlation output responses from the FNN models, eye digram analysis can now be performed in a much faster manner as opposed to the conventional SPICE circuit solvers.

To Mom and Dad,
who always believe in me
and encourage me to go on every adventure.
Here I am :)

ACKNOWLEDGEMENT

This dissertation marks an important milestone in my life. I could not have imagined, on the day of 2012 when I joined University of Illinois at Urbana-Champaign as an undergrad, that I would be able to accomplish this much to earn the Doctoral degree. This long journey of academic pursuit and self-reflection would not been possible without the guidance from my dear advisor Professor José Schutt-Ainé, who introduced me to his group during my sophomore year. For ten years I have seen his students come and go, but what remains unchanged is his great passion towards scientific discovery as well as his continuous encouragement to all the group members. I am truly thankful for his patience and support throughout my course of study, especially during the time when I was in self-doubt.

Besides my advisor, I would like to express my deepest gratitude to the rest of my dissertation committee: Professor Jennifer Bernhard, Professor Erhan Kudeki and Professor Peter Dragic. I highly appreciate their valuable feedback to my preliminary exam presentation so that I was able to better improve my work before the final defense. It is my honor to have them witness my progress from an ignorant ECE freshman all the way till here.

Next, I would like to thank my colleagues who generously share their knowledge to lead me through the obstacles I encountered during my research. Dr. Thong Nguyen, whom I am forever grateful to, always patiently addresses my concerns even when they are naively dumb and time consuming to explain. I respect his attitudes towards academic-related matters: relentlessly searching for new break-through possibilities while selflessly devoting time to mentor students like me who are struggling. I wish to thank Dr. Hanzhi Ma as well for all the advice as a friend and collaboration as a researcher. It is frankly rare to meet a fellow female member in this field, yet

I am so lucky to have her as both my roommate and co-author. I want to thank my group members: PhD candidate Bobi Shi, PhD candidate Juhitha Konduru, Dr. Xiao Ma, Dr. Xinying Wang, Dr. Xiou Ge, Gene Shiue and many who graduated before me. I dearly enjoy our talks and I hope for a bright future for all of you.

Also, I wisth to thank Karen Kuhns, Victor Shangguan, Dr. Hongliang Li, Dr. Shuo Liu and the rest of the residents on the fifth floor of ECEB. You constantly remind me that I am never fighting alone. I would also like to offer special thanks to Yuhe Cao, who, although no longer with us, continues to inspire me working towards this grand finale. I hope I had carried on your dream and may your soul rest in peace.

In addition, I want to express appreciation to my closest friends who motivate me through the unseen challenges. Many thanks to Siyan Guo, my lovely roommate for the past five years. We met during our freshman year and since then she has been looking out for me like a big sister. Although we parted after her graduation, our bound is stronger than ever when we encourage each other to purse our dreams. I would also like to thank my gaming buddies, Dr. Qing Ding and Jingchao Zhou. Time I spent in this little town is no longer tedious when we hop on discord and game for hours. This means a lot to me. I cannot imagine how stressful a PhD life can be without our happy hours in Overwatch. I thank Shuchen Song and G-Tay for their companions throughout my grad school journey. There were times when I deviated from the path, but they pulled me back and guided me all way till here.

Finally, my achievement today would not be possible without the support from my parents, Feng Zhao and Yan Liao. I deeply appreciate their unconditional love and encouragement, especially of their scarifies to send me eight thousands miles from home for better eduction. It is been too long since we last hold hands. I love you mom and dad. Although you will not be reading this dissertation, I still sincerely hope my work had done your proud.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Due to the ever increasing demand of parallel processing and multi-level cache memory, integrated circuit (IC) designers nowdays are fitting billions of 7 nm technology transistors on a fingernail-size chip [1]. While the transistor count keeps growing at the rate of Moore's law, on-chip signal integrity (SI) analysis has never been more challenging given the accumulating IC complexity. The burden not only lies in the number of the gates, but also the shrank timing and noise margin that are outcomes of the faster transition times [2]. Conventionally, SI engineers use Simulation Program with Integrated Circuit Emphasis (SPICE) for time domain simulation by solving for node voltage and branch current at each time step. Nevertheless, this process becomes extremely computationally expensive when it comes to transient simulation, where nonlinearities are taken into account. As shown in Fig. 1.1, multiple Newton-Raphson iterations are required before convergence is achieved at every time step for a nonlinear system. Given the super-linearity of the computation size, it is impractical to apply SPICE for large scale SI validation if structural models of transistors are used.

To improve the analysis efficiency, behavioral models are introduced as black-box alternatives to their equivalent SPICE-like circuits. Instead of accessing the transistor level design, the model yields the output in response to the excitation based on either previously simulated data or bench measurements. One of the most well-known standards is the I/O Buffer Information Specification (IBIS) model, where the I/O's voltage and current data are saved as tables categorized by various usage scenarios [4]. With a slight decrease in accuracy, much faster simulation speed is attained in the absence

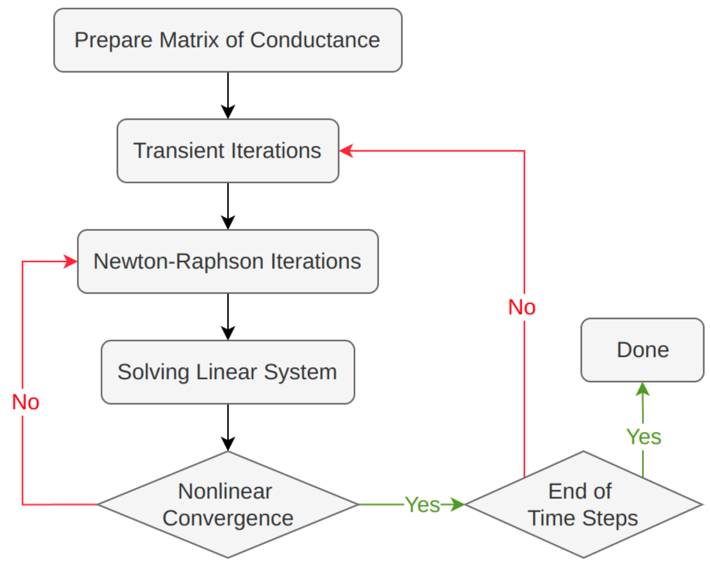


Figure 1.1: Flowchart of a transient SPICE simulation [3].

of nonlinear solving iterations. While IBIS models are supported by the vast majority of IC vendors because of their intellectual property (IP) protection nature, end users are assumed to have substantial domain knowledge in order to use the models properly in addition to producing them. As the IBIS standard evolves over time to include more complicated features like interconnect and algorithmic calculation [5], the learning curve faced by a novice user is becoming excessively steep.

In the effort of simplifying the modeling process, attempts were made deriving behavior models with machine learning (ML) methods. Rather than physically mimicking the SPICE circuit with current sources and passive components [6], the ML approach approximates the nonlinear function represented by the I/O data through training and stores the vector of kernels as the model. In recent years, many remarkable results are reported on modeling highly nonlinear electronic devices and systems with supervised learning algorithm, to be specific, the neural network (NN) framework. For instance, several research efforts successfully model broadband power amplifiers with variants of NN, including convolutional neural network (CNN) [7], deep neural network (DNN) [8] and memory polynomial neural network (MPNN) [9]. There are also articles that focus on modeling sequential circuits for digital designs simulation with recurrent neural network (RNN) [10, 11, 12].

Nevertheless, very few NN based behavior models were developed that are suitable for the high speed link (HSL) simulation (see Chapter 1.2.2). To perform such analysis, a passive channel is placed in-between the nonlinear transceiver models and millions of bits of data is collected at the cascading nodes to construct eye diagrams which are used to assess the link performance. The major issue with the reported NN frameworks is the ambiguous statement on the cascade-ability of the transmitter(TX) and the receiver (RX) blocks. In other words, the existing modeling techniques treated a particular configuration of HSL as an entity with an underlaying assumption that the transceiver model needs to be re-trained every time when a new channel is presented. Since the required training time of the NN models is as much, if not more, than performing the SPICE nonlinear iterations, the purpose of adopting behavior models in HSL simulation is defeated unless the cascade-ability of the transceiver models is properly addressed.

To resolve the aforementioned issue, this dissertation proposed a feed-forward neural network (FNN) based modeling technique for generating cascade-able transceiver blocks that are dedicated for HSL analysis. Different protocols are assigned to the TX and RX models to ensure the trained blocks could function independently regardless of changes in channel. In terms of cascading, the FNN RX model always takes the voltage predictions from the previous block as input and then pass on the calculated responses to the next available block along the link. It is also worth noting that when applying the FNN models, computation efficiency and convergence are guaranteed because the HSL simulation is now merely done by matrix multiplication between the voltage excitations and the kernels stored within the TX/RX models. The performance of this new method is examined by comparing the runtime and waveforms between the FNN model and the SPICE circuit for many distinct test cases, namely the NRZ excitation, PAM-4 excitation, DFE equalization and differential signaling.

This dissertation is structured as follows: In Chapter 1, the motivation of this project and a review on the related previous works are presented. Chapter 2 explained the TX/RX modeling protocols and the structure of FNN. Then in Chapter 3, training procedure of the two transistor-level gate examples, NAND and CMOS, are demonstrated and the trained blocks are cascaded with testing channels. The accuracy of the modeling results are evaluated against their references from a commercial circuit solver as well in this chapter. The proposed method's limitation and alternative implementation are discussed in Chapter 4. Chapter 5 summarizes the contribution of the proposed approach and provides an outlook on future developments. At last, there are two appendixes that explain the training data preparation and hardware acceleration with CUDA in detail.

1.2 Review of Previous Works

This section discusses the currently available behavioral modeling techniques for HSL simulation. The first subsection focus on the traditional IBIS approach, which is the golden standard adopted by many EDA tool. The second subsection focus on the more recent NN based methods, of which the modeling accuracy is justified but are greatly limited by their incapability of functioning in a cascade-able manner.

1.2.1 IBIS and IBIS-AMI Models

As a global industry standard, IBIS and IBIS-AMI models offer good accuracy in nonlinear system simulation as well as protecting vendors intellectual property. Begin from early 1990s, semiconductor companies started supplying IBIS models to their end users to simulate SI at chip and board level. This modeling technique is particularly well suited for large scale analysis because the device's I/O relation is pre-solved on the vendor side under typical, maximum and minimum operation conditions. The enhanced version of IBIS, called the IBIS-AMI with BIRD flow, was first released in 2012 aiming to specify the interface between the TX/RX and the channel simulator. In this subsection, a brief introduction of both models is given so the readers understand how IBIS functions as a cascade-able unit in HSL simulation.

The IBIS model is a human-readable text file (*.ibis) that records the buffer's I-V (current versus voltage) and switching characteristics V-T (output voltage versus time) in a tabular format [14]. In addition, it also allows

Table 1.1: Simulated or measured elements required by the IBIS model

Keyword	Type	I/O buffer	Input buffer	Output buffer
[GND Clamp]	I-V	Yes	Yes	No
[POWER clamp]	I-V	Yes	Yes	No
[Pullup]	I-V	Yes	No	Yes
[Pulldown]	I-V	Yes	No	Yes
[Rising Waveform]	V-T	Yes	No	Yes
[Falling Waveform]	V-T	Yes	No	Yes

the simulator to take in account the information of the packaging around the buffer such as the parasitic RLC, which are stored as optional lines in electrical parameter keywords. In its simplest format, IBIS model characterize the buffers with six sets of I-V and V-T data as shown in Table 1.1. Each of the keywords is a look-up table simulated or measured with the steps described in Table 1.2. Generally, the min-corner case is generated with the weakest driver strength and/or slowest bit rate while the max-corner case takes the reverse. The exact values of V_{SS} and V_{DD} are determined by the realistic use cases of the buffer. For example, a CMOS can be modeled with V_{SS} equals to ground and V_{DD} equals to its biasing voltage. For more complicate designs like differential buffer, additional keywords such as differential capacitance are required to fully characterize the device. After generating an IBIS model, one must manually validate that the extracted I-V and V-T tables are in accordance with the SPICE simulation result as well as their compliance with the IBIS syntax. In the context of a transient simulation, the solver first calculates the dependent multipliers $Ku(t)$ and $Kd(t)$ from the V-T tables and then perform a summation of the currents gathered at the output node as

$$-Iout(t) = Ku(t) \cdot Ipu(V) + Kd(t) \cdot Ipd(V) + Ipc(V) + Igc(V), \qquad (1.1)$$

where the $Ix(V)$ values are determined according to the IV tables at each time point using piecewise linear interpolation [15]. Overall, as long as the range of IV/VT tables covers the voltage excitation used in the transient simulation, IBIS model is theoretically capable of a yielding accurate results for a cascading chain of any length.

Table 1.2: Simulation or measurement environment of IBIS keywords [13]

Keywords	Data extraction flow (typical and min/max-corners)
[GND Clamp]	Disable the output pin and connect it to a ground reference or pull down reference $-V_{SS}$. Insert a dc voltage source V_{pin} in-between the connection and perform a sweep from $-(2 \times V_{SS} + V_{DD})$ to $(2 \times V_{DD} + V_{SS})$. Measure the current I_{pin} at the output. Gather values of I_{pin} and V_{pin} to make the I-V table.
[POWER clamp]	Disable the output pin and connect it to a power/pull-up reference V_{DD}. Insert a dc voltage source V_{pin} in-between the connection and perform a sweep from $-(2 \times V_{SS} + V_{DD})$ to $(2 \times V_{DD} + V_{SS})$. Measure the current I_{pin} at the output. Gather values of I_{pin} and V_{pin} to make the I-V table.
[Pullup]	Enable the output pin and set it to logic high. Connect it to a power/pull-up reference V_{DD}. Insert a dc voltage source V_{pin} in-between the connection and perform a sweep from $-(2 \times V_{SS} + V_{DD})$ to $(2 \times V_{DD} + V_{SS})$. Measure the current I_{pin} at the output. Gather values of I_{pin} and V_{pin} to make the I-V table.
[Pulldown]	Enable the output pin and set it to logic low. Connect it to a ground reference or pull down reference $-V_{SS}$. Insert a dc voltage source V_{pin} in-between the connection and perform a sweep from $-(2 \times V_{SS} + V_{DD})$ to $(2 \times V_{DD} + V_{SS})$. Measure the current I_{pin} at the output. Gather values of I_{pin} and V_{pin} to make the I-V table.
[Rising Waveform]	Enable the output pin and apply appropriate input voltage pulse so its logic state switches from low to high. Insert a fixture of r Ω between the pin and the power/pull-up reference V_{DD}. Measure the rising slew rate dV/dt and record it as the V-T table.
[Falling Waveform]	Enable the output pin and apply appropriate input voltage pulse so its logic state switches from high to low. Insert a fixture of r Ω between the pin and the pull down reference $-V_{SS}$. Measure the falling slew rate dV/dt and record it as the V-T table.

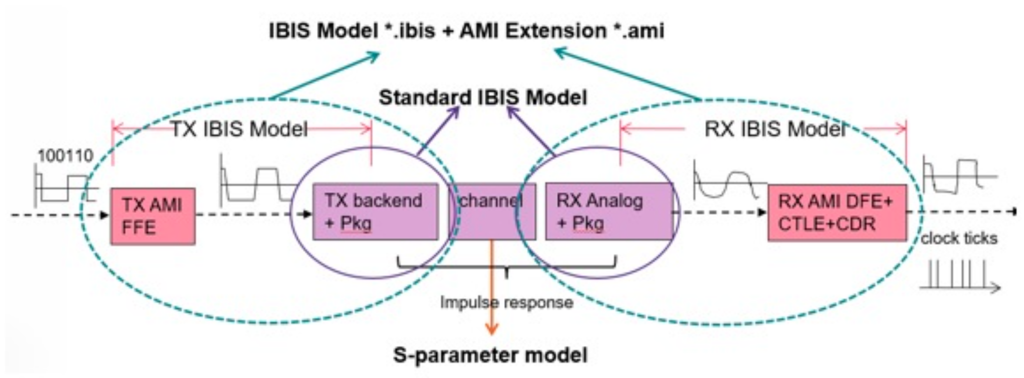


Figure 1.2: Flowchart of a HSL simulation with IBIS-AMI models [16].

The IBIS-AMI model is invented to standardize the HSL simulation with IBIS specifications. The model is divided into two files: the analog IBIS part (*.ibis) and the executable algorithmic AMI part (*.dll). As shown in Figure 1.2, the equalization settings such as FFE in TX and DFE in RX are included in the AMI extension file. The HSL transient simulation with IBIS-AMI models begins with the calculation of impulse response, which is a combined force of Equation (1.1) along with applying the inverse Fourier transform on the S-parameter characterized channel. Then, the AMI portion acts as a DSP block which takes the calculated impulse response and produces a modified output based on the equalization presets. It is worth mentioning that if the TX/RX IBIS-AMI models in the HSL support the same link training protocol, the new versions of AMI could perform dynamic equalization, meaning that the tap values in DFE are adaptive to the signature of the channel. Note that if IBIS-AMI models are used, the HSL is strictly defined as one TX cascading with one channel and one RX, which is not as robust as the previously mentioned IBIS model.

1.2.2 Neural Network Models

To facilitate the process of generating and utilizing customizable behavior models in HSL simulation, NN based modeling techniques are being explored recently, taking advantage of the hardware acceleration for faster convergence during the training phase. For instance, Chu *et al.* [17] introduced a back propagation based NN (BNN) model that predicts the output after the receiver (RX) given the excitation before the transmitter (TX). This work

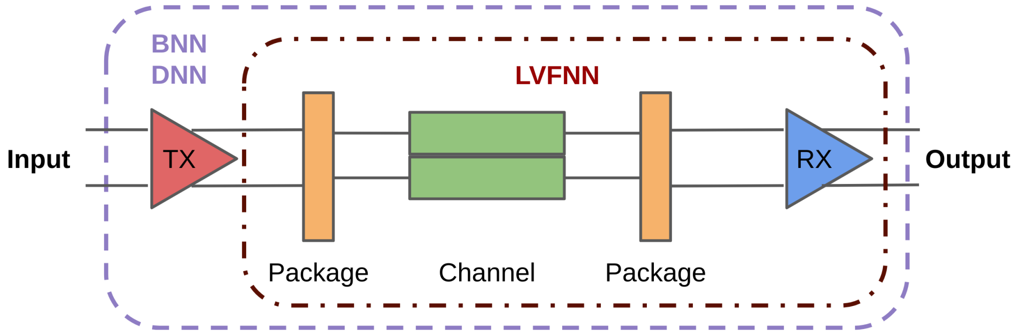


Figure 1.3: Modeling diagram of BNN, DNN and LVFNN on a HSL.

proofs that the NN could handle nonlinear behaviors (e.g. mismatch between the rising and falling time in the excitation) well with only three hidden layers. Similarly, Lu *et al.* [18] adopted a DNN model that accurately predicts the eye height at RX with variations in the channel geometry and TX/RX jitters. Another example is the Laguerre–Volterra feed-forward neural network (LVFNN) suggested by Wang *et al.* [19]. This work focus on reducing the required size of NN when modeling a nonlinear RX component with PAM-4 excitation. While all the above papers suggested a good correlation between the modeling results and SPICE references, the modeling techniques they demonstrated is performed on the entire HSL instead of a single TX or RX component as shown in Figure 1.3. Since the reported techniques use a single, large NN to model the entire link, they are oftentimes not robust enough for a real design scenario: A new transceiver model is required every time when HSL composition changes.

CHAPTER 2

MODELING TRANSCEIVER WITH FEED-FORWARD NEURAL NETWORK (FNN)

2.1 Introduction

In this Chapter, the methodology of developing cascade-able transceiver models with feed-forward neural network (FNN) is presented. As shown in Figure 2.1, the FNN transceiver models will be cascaded to perform HSL simulation with a slightly different work-flow compared to the traditional IBIS/SPICE models. This modification is mandatory because the TX/RX FNN models are expected to be resilient to certain variations in the cascading chain and hence, requiring the parameterized channel and termination information as inputs. In addition, it is worth noting that modeling with FNN creates a self-sustain system, where its HSL analysis no longer demands any commercial EDA tool: The simulation is now done through a chain of matrix multiplication based on the stored kernels in the TX/RX FNN models. Thanks to the model's cascade-bility, one could also use a single block of FNN model independently to test the buffer's performance.

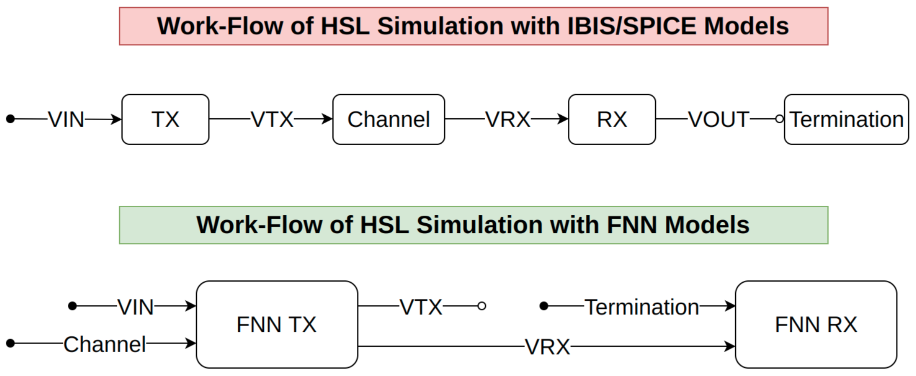

Work-Flow of HSL Simulation with IBIS/SPICE Models
VIN
TX
VTX
Channel
VRX
RX
VOUT
Termination
Work-Flow of HSL Simulation with FNN Models
VIN
Channel
FNN TX
VTX
VRX
Termination
FNN RX

Figure 2.1: Comparison of workflow between SPICE/IBIS and FNN.

2.2 Memory Effect in Nonlinear System

Given that the transceivers nowadays are generally nonlinear devices, the HSL system they form is considered to possess *memory* of which the discrete output $y(n)$ depends on both the current and the past states of the input $x(n)$. This nonlinear, causal, stable, time-invariant system is typically described by the truncated Volterra series [20]

$$y(n) = h_0 + \sum_{\tau_1=0}^{M} \sum_{\tau_2=\tau_1}^{M} \cdots \sum_{\tau_p=\tau_{p-1}}^{M} h_p\left(\tau_1, \ldots, \tau_p\right) \prod_{j=1}^{p} x\left(n-\tau_j\right), \tag{2.1}$$

where M is the memory length and p is the pth-order of the Volterra kernel (VK). Although mathematically all the VKs can be estimated by solving a least-mean-square (LMS) equation, the problem soon becomes intractable because the number of VKs to be solved is a exponential function in respect to p and M.

To lift the curse of dimensionality, many tend to apply the discrete Laguerre function as the projecting basis for VKs expansion [21]. Such function can be expressed as

$$\phi_r(\tau) = \alpha^{\frac{\tau-r}{2}}(1-\alpha)^{\frac{1}{2}} \sum_{k=0}^{r}(-1)^k \begin{pmatrix} \tau \\ k \end{pmatrix} \begin{pmatrix} r \\ k \end{pmatrix} \alpha^{r-k}(1-\alpha)^k \tag{2.2}$$

with $\phi_r(t)$ being the r-th orthonormal basis function. From there, the VKs can now be expanded as

$$\begin{aligned} h_0 &= \theta_0 \\ h_1(\tau) &= \sum_{r=1}^{r=R} \theta_r \phi_r(\tau) \\ h_2\left(\tau_1, \tau_2\right) &= \sum_{r_1=1}^{r_1=R} \sum_{r_2=1}^{r_2=R} \theta_{r_1, r_2} \phi_{r_1}\left(\tau_1\right) \phi_{r_2}\left(\tau_2\right) \\ h_p\left(\tau_1, \ldots, \tau_p\right) &= \sum_{r_1=1}^{r_1=R} \cdots \sum_{r_n=1}^{r_n=R} \theta_{r_1, \ldots, r_n} \prod_{l=1}^{p} \phi_l\left(\tau_l\right), \end{aligned} \tag{2.3}$$

where R is the max order of the Laguerre function. After denoting a function ℓ of all known parameters as

$$\ell_r = \sum_{\tau=0}^{M} \phi_r(\tau) x(n-\tau), \tag{2.4}$$

Equation (2.1) can be transformed to

$$\begin{aligned} y(n) = \theta_0 + \sum_{r=1}^{R} \theta_r \ell_r + \sum_{r_1=1}^{r_1=R} \sum_{r_2=1}^{r_2=R} \theta_{r_1,r_2} \ell_{r_1} \ell_{r_2} \\ + \cdots + \sum_{r_1=1}^{r_1=R} \cdots \sum_{r_n=1}^{r_n=R} \theta_{r_1,\ldots,r_n} \prod_{i=1}^{p} \ell_{r_i}. \end{aligned} \tag{2.5}$$

with $[\theta_0, \theta_{r1}, \cdots, \theta_{rn}]$ being the new kernels. Given that R is usually a much smaller number than T, the identification of the kernels is thereby greatly simplified [19, Tab. I, II].

However, this approach is problematic when $x(n)$ is a long sequence sampled at tiny time steps. Even with a relatively small R , at a certain point, convolution between matrices ϕ_r, θ_r, and $x(n)$ may exceed the available computation capability. In other words, to completely avoid the convolutions in Equation (2.1), the nonlinearity of the system should be addressed not as a summation but a nonlinear function f such that for a input memory matrix $\boldsymbol{X}$ of size $(n-M) \times M$, the I/O relation between $\boldsymbol{X}$ and the output $\boldsymbol{Y}$ can be written in the form:

$$\boldsymbol{X} = \begin{bmatrix} x(1) & x(2) & \cdots & x(M) \\ x(2) & x(3) & \cdots & x(M+1) \\ \vdots & \vdots & \vdots & \vdots \\ x(n-M) & x(n-M+1) & \cdots & x(n) \end{bmatrix} \tag{2.6}$$

$$\boldsymbol{Y} = \begin{bmatrix} y(M) & y(M+1) & \cdots & y(n) \end{bmatrix}^T = f(\boldsymbol{X}). \tag{2.7}$$

As demonstrated in the later sections, the exact implementation of f is determined by the structure of the hidden layers in the FNN. The bottom line is f does not necessarily take the same form for all buffers, meaning that it is up to the designers to decide if more nonlinearity should be added into f by adjusting the FNN framework.

2.3 Modeling Protocols

Besides the memory sequences, the FNN model demands information of the channel as well as the load condition in order to execute a comprehensive HSL simulation. These extra data are referred to *protocol* in this section. Protocols are appended to every row of the input memory matrix such that the FNN recognizes different configurations of HSL. Conceptually speaking, there is no fixed protocol because as long as the modeling and application use the same terms, FNN models could yield accurate prediction correspondingly. The TX/RX protocols described below are merely a proof of concept and are subject to future expansion for more advanced HSL usage.

Since the TX model is expected to make voltage predictions for both before and after the channel, two protocols are proposed on how the channel is parameterized and incorporated into the framework of the model:

1. For channels that are uniform and homogeneous (e.g., a section of microstrip line (MLIN)), use their geometric features such as width w, length l, dielectric constant ϵ_r, etc.

2. For more complexly structured channel, use the terminal frequency response such as the S-parameter.

The geometric approach is straightforward and already proofed to be NN-compatible for eye height and width prediction [22]. Meanwhile, the S-parameter approach permits a wider application given that most of the channels in HSL simulations are inhomogeneous. Both protocols are demonstrated in Chapter 3 to show that accuracy-wise, there is essentially no difference in using either of them.

Meanwhile, the load condition is specified in the RX protocol. As a proof of concept attempt, a resistor of r_t Ω is used to terminate the HSL. This assumption can be easily expanded to include more components such as capacitors and inductors by adjusting the RX protocol. Overall, the concept of protocol is very similar to the V_{DD} and V_{SS} sweep in the IBIS model: The range of the sweeping values adopted by the protocol determines the model's operation limitation.

2.4 Vector Fitting for Channel Parametrization

Vector fitting (VF) is a black-box modeling technique that extracts reduced-order passive macromodel from measured or computed frequency domain data with rational function approximations [23]. Compared to the traditional approximation function in Equation (2.8) where the unknown coefficients a_n and b_n are binded with different powers of s, VF scales the function with respect to a single column of s and reduce the approximation to a linear problem that allows higher order terms for fitting over a wide frequency range.

$$F(s) \approx \frac{a_0 + a_1 s + a_2 s^2 + \ldots + a_N s^N}{b_0 + b_1 s + b_2 s^2 + \ldots + b_N s^N} \tag{2.8}$$

Given a symmetric n-port S-parameter matrix $\boldsymbol{S}$ sampled as a function of frequency $s = j\omega$, the rational approximation function used by VF can be represented by

$$\boldsymbol{S}(s) = \sum_{m=1}^{N} \frac{\boldsymbol{r}_m}{s - a_m} + \boldsymbol{d}, \tag{2.9}$$

where $a \in \mathbb{C}^N$ denotes the complex pole, $\boldsymbol{r} \in \mathbb{C}^{n \times n \times N}$ is the complex residue, $\boldsymbol{d} \in \mathbb{R}^{n \times n}$ is the real constant and N is the number of poles required for a good fit of $\boldsymbol{S}(s)$ which depends on the shape of the responses. The approximation begins with pole identification by solving a least square (LS) linear problem in Equation (2.10), where a set of starting poles $\{q_m\}$ are used to launch the re-location process in Equations (2.11) and (2.12).

$$\sigma(s)\boldsymbol{S}(s) = p(s) \tag{2.10}$$

$$\sigma(s) = \sum_{m=1}^{N} \frac{\tilde{\boldsymbol{r}}_m}{s - q_m} + 1 \tag{2.11}$$

$$p(s) = \sum_{m=1}^{N} \frac{\boldsymbol{r}_m}{s - q_m} + \boldsymbol{d} \tag{2.12}$$

As explained in [24], the poles of $\boldsymbol{S}(s)$ are identical to the zeros of $\sigma(s)$, which can be obtained as

$$\{a_m\} = \text{eig}\left(\boldsymbol{Q} - \boldsymbol{b} \cdot \boldsymbol{c}^T\right), \tag{2.13}$$

where $\boldsymbol{Q}$ is a diagonal matrix consists of q_m, $\boldsymbol{b}$ is a column vector filled with ones, and $\boldsymbol{c}^T$ is a row vector made of residues $\tilde{r}_m$. The initial choices of $\{q_m\}$ are generally complex conjugate pairs because the purely real poles might cause Equation (2.13) to be ill-conditioned. Note that Equations (2.10)-(2.13) can be carried out iteratively with the new poles $\{a_m\}$ substituting the previous ones $\{q_m\}$ for more precise fitting. When unstable poles are encountered, VF inverts the sign of their real part to ensure overall stability.

Moreover, Gustavsen [25] improved the LS convergence by replacing Equation (2.11) with Equation (2.14) and Equation (2.13) with Equation (2.15). This allows a more relaxed fitting because the previous method is normalized by a unity term that forces $\sigma(s) = 1$ at high frequency. An additional normalization function is introduced as shown in Equation (2.16) where N_s is the number of frequency points in $\boldsymbol{S}(s)$. After the elimination of the asymptotic requirement in Equation (2.10), $\sigma(s)$ can now theoretically approach a much smaller value at infinite frequency. With the poles being identified, Equation (2.10) can be transformed to another LS equation where $\boldsymbol{r}$ and $\boldsymbol{d}$ are solved [23, eq. (A.1)-(A.8)].

$$\sigma(s) = \sum_{m=1}^{N} \frac{\tilde{\boldsymbol{r}}_m}{s - q_m} + \tilde{\boldsymbol{d}} \tag{2.14}$$

$$\{a_m\} = \text{eig}\left(\boldsymbol{Q} - \boldsymbol{b} \cdot \tilde{\boldsymbol{d}}^{-1} \cdot \boldsymbol{c}^T\right) \tag{2.15}$$

$$\text{Re}\left\{\sum_{k=1}^{Ns}\left(\sum_{m=1}^{N} \frac{\tilde{\boldsymbol{r}}_m}{s_k - a_m} + \tilde{\boldsymbol{d}}\right)\right\} = N_s \tag{2.16}$$

Although VF always yields guaranteed stable poles, the passivity of the pole-residue model is not yet examined in the previous procedures. Thus, a further tuning step with Hamiltonian matrix [26] is performed on the state-space representation of the model. Since S-parameter is only suppose to characterize a passive system, $S(s)$ is analytic on the open right half plane

[27] and should be bounded by unity:

$$\big(\boldsymbol{I} - \boldsymbol{S}(s)^H \boldsymbol{S}(s)\big) > 0, \tag{2.17}$$

where H denotes the conjugate transpose (Hermitian) operation and $\boldsymbol{I}$ is the unity matrix. Then, the pole-residue model is transformed to its state-space form in Equation (2.18) by seeking similar terms in Equation (2.10). With the extraction steps explained in [28], the state variables are listed in Equation (2.19) where $\boldsymbol{A}_M$ is constructed by concatenating the scalar a_m diagonally. The dimensions of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ and $\boldsymbol{D}$ are $nN \times nN$, $nN \times n$, $n \times nN$ and $n \times n$ respectively. Figure 2.2 illustrates an example of the transformation from pole-residue model to its state-space representation where number of ports $n = 2$ and max order of poles $N = 3$.

$$\boldsymbol{S}(\omega) = \boldsymbol{C}(j\omega \boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B} + \boldsymbol{D} \tag{2.18}$$

$$\begin{aligned} \boldsymbol{A} &= diag[\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_M] \\ \boldsymbol{B} &= [\boldsymbol{I}, \boldsymbol{I}, \ldots, \boldsymbol{I}] \\ \boldsymbol{C} &= [\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_m]^T \\ \boldsymbol{D} &= \boldsymbol{d} \end{aligned} \tag{2.19}$$

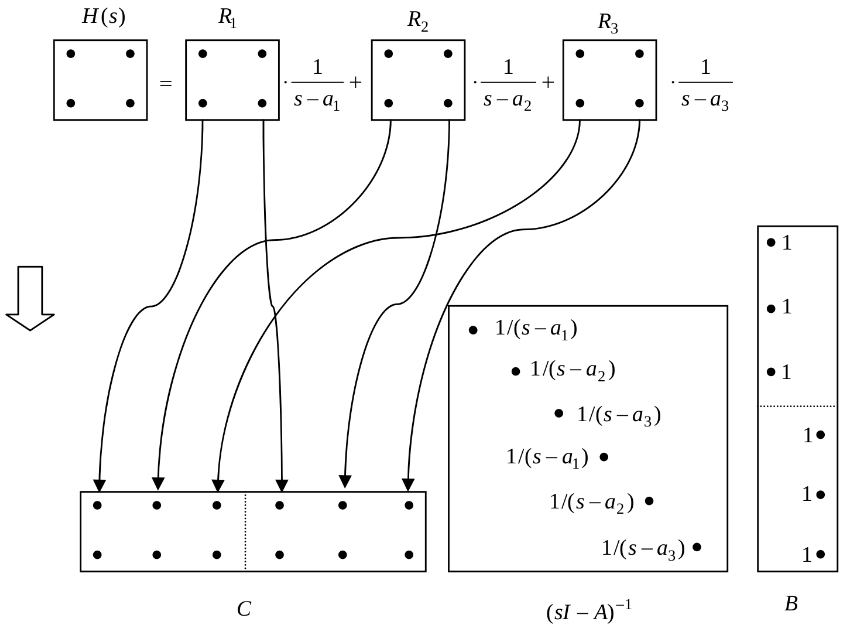


Figure 2.2: Conversion between pole-residue model and its state-space form.

From there, passivity assessment can be completed by evaluating the eigenvalues λ of the Hamiltonian matrix as

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{A} - \boldsymbol{B}\hat{\boldsymbol{R}}^{-1}\boldsymbol{D}^T\boldsymbol{C} & -\boldsymbol{B}\hat{\boldsymbol{R}}^{-1}\boldsymbol{B}^T \\ \boldsymbol{C}^T\hat{\boldsymbol{S}}^{-1}\boldsymbol{C} & -\boldsymbol{A}^T + \boldsymbol{C}^T\boldsymbol{D}\hat{\boldsymbol{R}}^{-1}\boldsymbol{B}^T \end{bmatrix}, \tag{2.20}$$

where $\hat{\boldsymbol{S}} = (\boldsymbol{D}\boldsymbol{D}^T - \boldsymbol{I})$ and $\hat{\boldsymbol{R}} = (\boldsymbol{D}^T\boldsymbol{D} - \boldsymbol{I})$. Violations are checked at any imaginary part of λ that defines a crossover frequency js where a singular value touches the unity threshold [29]. Faster singularity analysis, as indicated in [30] and [31], can be achieved by reducing the size of $\boldsymbol{H}$ based on the symmetries in the S-parameter. A symmetric state-space model is define by $\boldsymbol{S} = \boldsymbol{S}^T$ with $\boldsymbol{C} = \boldsymbol{B}^T$ and $\boldsymbol{D} = \boldsymbol{D}^T$, which simplifies Equation (2.20) to

$$\hat{\boldsymbol{H}} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ -\boldsymbol{F} & -\boldsymbol{E} \end{bmatrix} \tag{2.21}$$

where $\boldsymbol{E} = \boldsymbol{A} - \boldsymbol{B}\left(\boldsymbol{D}^2 - \boldsymbol{I}\right)^{-1}\boldsymbol{D}^T\boldsymbol{C}$ and $\boldsymbol{F} = -\boldsymbol{B}\left(\boldsymbol{D}^2 - \boldsymbol{I}\right)^{-1}\boldsymbol{C}$. After denoting each pair of eigenvalue as λ and eigenvector as $\tilde{\boldsymbol{x}}$ for $\hat{\boldsymbol{H}}$, one could obtain

$$\left[(\boldsymbol{E}+\boldsymbol{F})(\boldsymbol{E}-\boldsymbol{F}) - \lambda^2\boldsymbol{I}\right]\tilde{\boldsymbol{x}} = 0, \tag{2.22}$$

of which the λ is exactly the square-roots of the eigenvalues of the matrix

$$\boldsymbol{P} = (\boldsymbol{E}+\boldsymbol{F})(\boldsymbol{E}-\boldsymbol{F}) \tag{2.23}$$

or

$$\boldsymbol{P} = \left(\boldsymbol{A} - \boldsymbol{B}(\boldsymbol{D}-\boldsymbol{I})^{-1}\boldsymbol{C}\right)\left(\boldsymbol{A} - \boldsymbol{B}(\boldsymbol{D}+\boldsymbol{I})^{-1}\boldsymbol{C}\right). \tag{2.24}$$

Note that test matrix $\boldsymbol{P}$ is only half-sized compared to the original singularity matrix $\boldsymbol{H}$. This change accelerates the passivity assessment roughly eight times faster, which better suits the large cases like fitting for wide frequency band. Within the band where passivity violation occurs, the eigenvalues are tuned by bringing up the minimas to the zero line. After a few checking iterations, the VF model is assured to be globally passive.

2.5 Construction of FNN Framework

FNN is an artificial NN wherein the connections between the nodes mimics the design of human brain neurons by processing information unidirectionally along the layers. A typical architecture of FNN is shown in Figure 2.3, defining the mapping between the input features' vector $\boldsymbol{x}$ and the output labels' vector $\boldsymbol{y}$ through an approximation function $\boldsymbol{y} \approx f^*(\boldsymbol{x})$. In this particular case of FNN having three hidden layers, f^* can be decomposed to

$$f^*(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}))), \tag{2.25}$$

where $f^{(L)}$ denotes the Lth hidden layer of the network. While the optimized value of L remains debatable [32, 33, 34, 35], a general rule of thumb is to gradually increase it until a satisfactory accuracy is achieved.

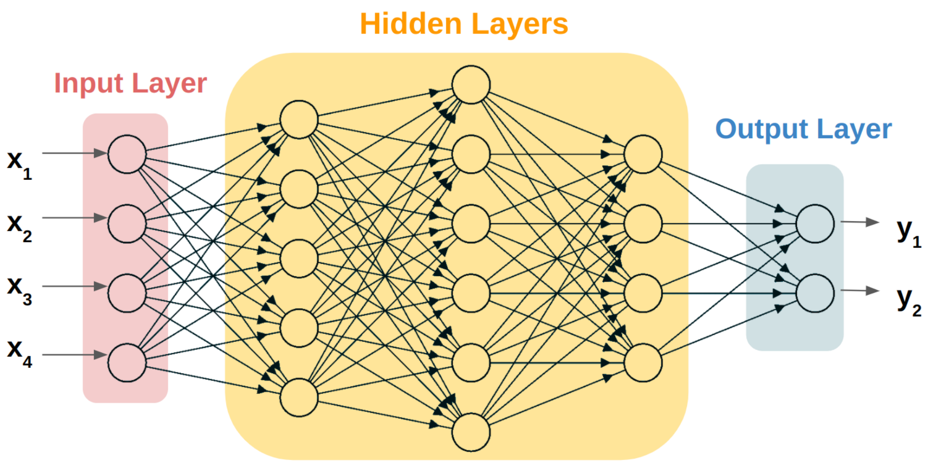


Figure 2.3: An example of FNN structure consisting of one input layer, three hidden layers and one output layer.

For a linear network, the hidden layer is oftentimes described by

$$f^{(L)}(\boldsymbol{x}; \boldsymbol{\theta}) = f^{(L)}(\boldsymbol{x}; \boldsymbol{W}, b) = \boldsymbol{x}^\top \boldsymbol{W} + b, \tag{2.26}$$

with $\boldsymbol{W}$ being the mapping weight vector and b is the scalar bias. The exact values of $\boldsymbol{\theta}$ are determined by locating global minimum of a cost function $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ through steepest gradient-based (SGD) optimization. A popular choice of $\mathcal{L}$ is the mean square error (MSE) function

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \mathcal{L}(\boldsymbol{y}, f^*(\boldsymbol{x}; \boldsymbol{\theta})) = \left\{ \frac{1}{|\mathbb{X}|} \sum_{\boldsymbol{x} \in \mathbb{X}} (\boldsymbol{y} - f^*(\boldsymbol{x}; \boldsymbol{\theta}))^2 \right\}, \tag{2.27}$$

where the Euclidean norm between the true value $\boldsymbol{y}$ and its approximation $\hat{\boldsymbol{y}}$ measures how well the model explains the observed data. The optimizer seeks to solve the problem

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \, \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{2.28}$$

such that first derivative of the cost function $\frac{d\mathcal{L}}{dx}$ approaches or lands on zero. This technique is essentially an iterative process because θ is updated each time in small steps with opposite sign of the derivative. Figure 2.4 demonstrates a simple example of SGD with $\mathcal{L} = \frac{1}{2}x^2$. In the case where multiple inputs are presented (e.g. $\boldsymbol{x}$ is a vector), the gradient is taken with respect to a directional vector that contains all the partial derivatives. The SGD algorithm moves in the direction of the negative gradient vector and halts when every element in the vector is equal or very close to zero.

For a non-linear network, a scalar-to-scalar activation function g is introduced to compensate for the nonlinearity in the hidden layer model. As illustrated in Figure 2.5, g is applied element-wise on every node in the Lth layer as

$$\begin{cases} h_j & = \boldsymbol{x}^\top \boldsymbol{W}_{:,j} + b_j, \quad j \in [1..k] \\ f^{(L)}(\boldsymbol{x}; \boldsymbol{\theta}) & = g(\boldsymbol{h}), \quad \boldsymbol{x} \in \mathbb{X} \end{cases}, \tag{2.29}$$

where k is the total number of the nodes in the Lth layer. Depending on the application, implementations for g may be vary. Some popular choices are listed below including the hyperbolic tangent in Equation (2.30), the sigmoid function in Equation (2.31) and the rectified linear unit (ReLU) [36] in Equation (2.32).

$$g(\boldsymbol{h}) = \frac{e^{\boldsymbol{h}} - e^{-\boldsymbol{h}}}{e^{\boldsymbol{h}} + e^{-\boldsymbol{h}}} \tag{2.30}$$

$$g(\boldsymbol{h}) = \frac{1}{1 + e^{-\boldsymbol{h}}} \tag{2.31}$$

$$g(\boldsymbol{h}) = max\{0, \boldsymbol{h}\} \tag{2.32}$$

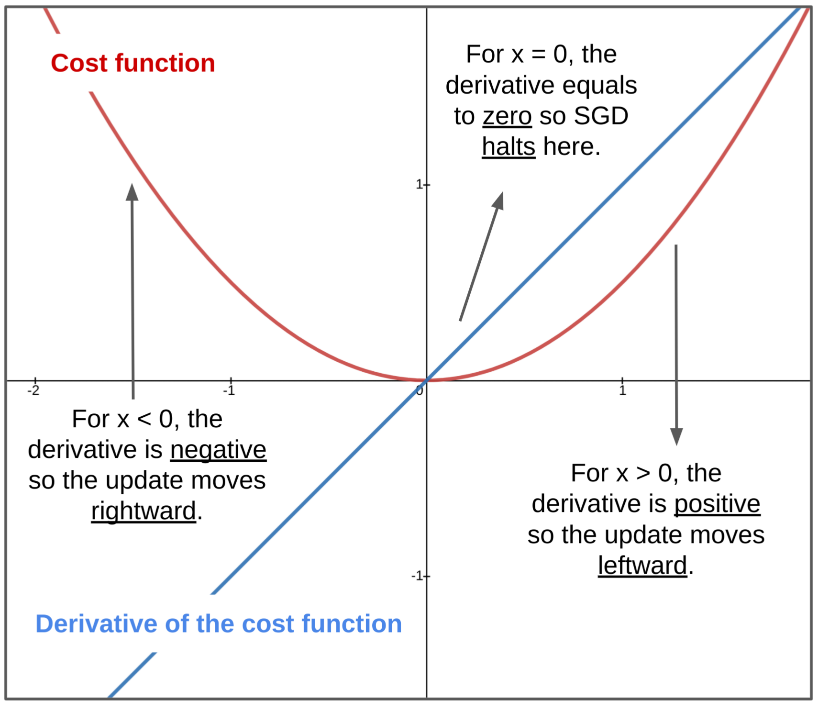


Figure 2.4: An illustration of how SGD searches the minimum in the cost function $\mathcal{L} = \frac{1}{2}x^2$ using the derivatives.

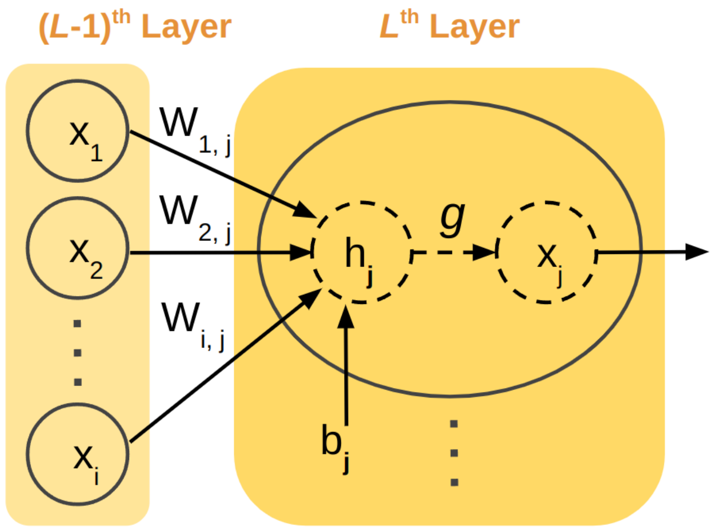


Figure 2.5: Application of activation function g in the hidden layers.

The solving process for $\boldsymbol{\theta}$ is not much different from the one for the linear system, expect the optimizer is now facing a non-convex problem. Specifically, SGD is more likely to hit a *saddle point* where the derivative of $\mathcal{L}$ equals to zero and thus loss the information about which direction to move. Such a point can either be a local maximum or minimum, which blocks the SGD to arrive at the desired global minimum (see Figure 2.6). Since global convergence is no longer guaranteed, the optimization now becomes very sensitive to the initial values of the unknown parameters. Hence, it is critical to initialize $\boldsymbol{W}$ to small random values and b to zero or small positive values before the samples are sent for training [37]. Another hyper-parameter that might help escaping the suboptimal solution is the learning rate η, which defines the size of the updating step. The general idea is to have η as an adaptive variable that is large when steeping down the gradient hill and small when it is close to a potential minimum. Modern SGD like Adam [38] from Pytorch has a built-in decaying algorithm for η, which maintains a fair balance between training time and model accuracy.

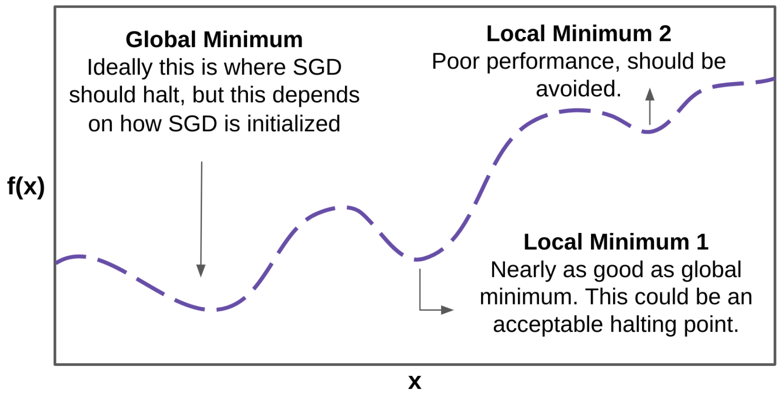

Global Minimum
Ideally this is where SGD should halt, but this depends on how SGD is initialized
Local Minimum 2
Poor performance, should be avoided.
f(x)
Local Minimum 1
Nearly as good as global minimum. This could be an acceptable halting point.
x

Figure 2.6: Non-convex condition with multiple local minima.

Even with Adam, there are times, however, that the solutions are still highly unreliable due to the complex hypothesis fitting. One example is *overfitting* as illustrated in Figure 2.7, where model B is clearly a better match for the true function but has a higher MSE compared to model A. Nevertheless, from the SGD perspective, it will always prefer model B because its fitting error is zero. To prevent overfitting in the FNN model, it is a common practice to divide the collected data into training, validation and testing sets with a ratio of 70% - 10% - 20%. The sets have to be randomly selected and shuffled to ensure unbiased evaluation. In an ideal scenario, the data within each sets should be evenly distributed with some points dedicated for corner cases. During the training phase, cost function is evaluate for both the training and the validation sets as training loss and validation loss. While the training loss is used by SGD to update $\boldsymbol{\theta}$ at each epoch, the validation loss is only calculated at the end of epoch to provide insights on the overfitting condition. Upon completion of all the epochs, one could inspect the loss plots as shown in Figure 2.8 to determine if the model overfits the function. If the model demonstrates good convergence, the reserved testing sets can be used to further verify its correctness; If overfitting occurs, one must halt the process here and deliberately calibrate how the data sets are distributed. In the end, successfully FNN training relays heavily on the given data: Sampled points in highly nonlinear and linear regions should be assigned evenly to the training, validation and testing sets.

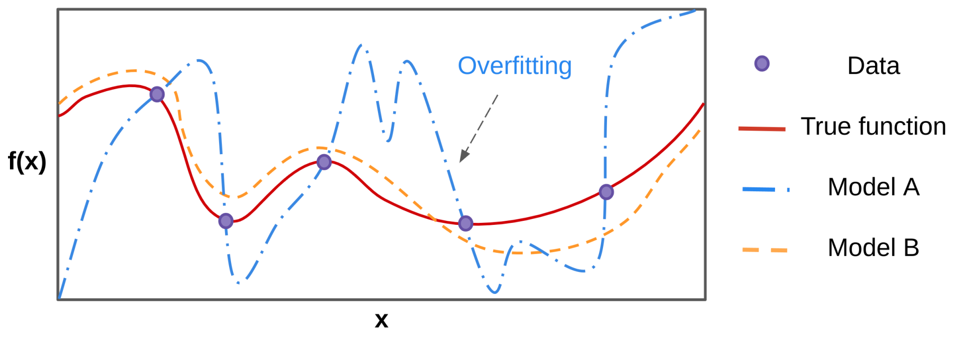


Figure 2.7: Model A overfits the true function with low MSE.

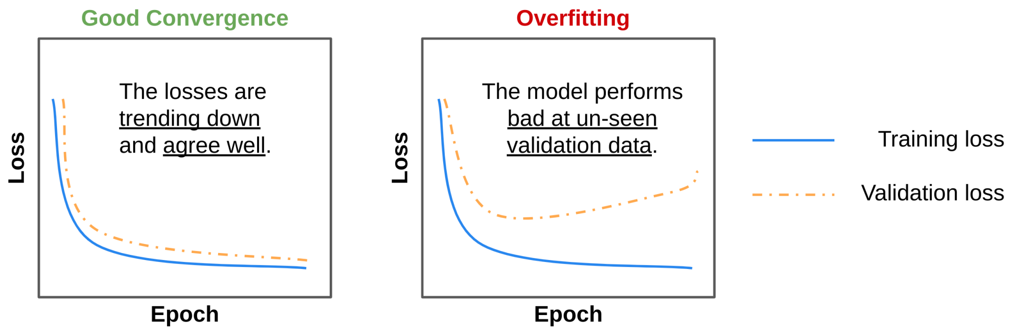


Figure 2.8: Use validation data to prevent overfitting condition.

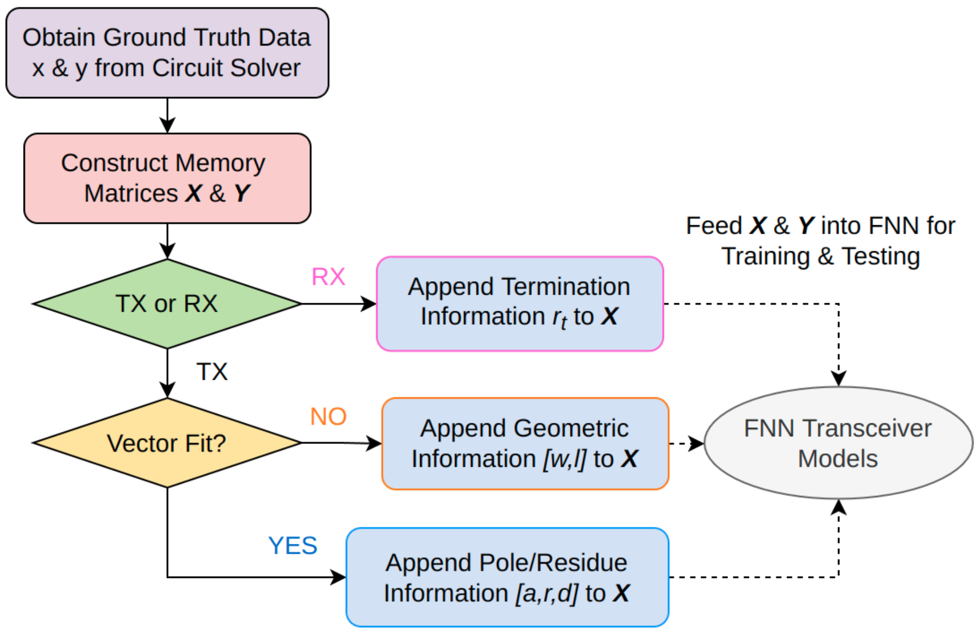
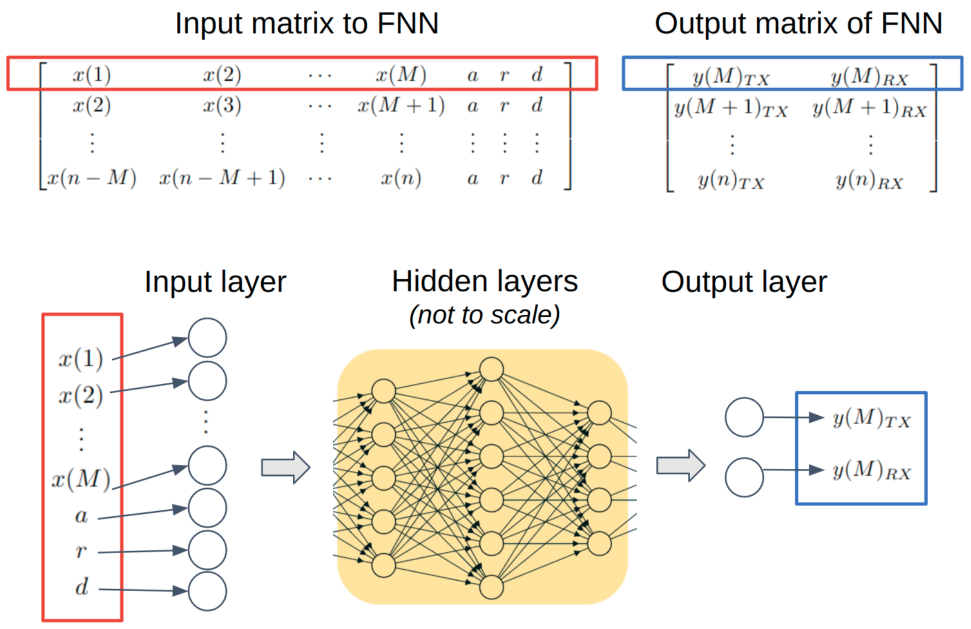

Obtain Ground Truth Data x & y from Circuit Solver
Construct Memory Matrices X & Y
TX or RX
RX
Append Termination Information r_t to X
Feed X & Y into FNN for Training & Testing
TX
Vector Fit?
NO
Append Geometric Information [w,l] to X
FNN Transceiver Models
YES
Append Pole/Residue Information [a,r,d] to X

Figure 2.9: FNN modeling workflow for the HSL transceivers.



$$\begin{bmatrix} x(1) & x(2) & \cdots & x(M) & a & r & d \\ x(2) & x(3) & \cdots & x(M+1) & a & r & d \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x(n-M) & x(n-M+1) & \cdots & x(n) & a & r & d \end{bmatrix} \quad \begin{bmatrix} y(M)_{TX} & y(M)_{RX} \\ y(M+1)_{TX} & y(M+1)_{RX} \\ \vdots & \vdots \\ y(n)_{TX} & y(n)_{RX} \end{bmatrix}$$

Figure 2.10: FNN structure for training TX with VF protocol.

2.6 Summary

In Figure 2.9, the FNN modeling workflow for the HSL transceivers is concluded using the protocol parameters indicated in the previous sections. First, the ground truth data is obtained from SPICE transient simulation in ADS and then transfered to python to construct memory matrices. An automation script is developed to facilitate this step as described in Appendix A. Then, based on the type of the buffer (TX/RX), different protocols are appended to the memory sequences. At the end, the matrices are fed to FNN for training as illustrated in Figure 2.10. The realization of VF and FNN training are done through python packages $skrf$ and $pytorch$, respectively. More details on the python programming environment can be found in Appendix B.

CHAPTER 3

APPLICATION OF FNN MODELS IN HIGH SPEED LINK (HSL) SIMULATION

3.1 Introduction

In this chapter, two transistor level gates, CMOS in Figure 3.1 and NAND in Figure 3.2, are designed in Keysight ADS and then modeled with FNN. The CMOS inverter was constructed by a PMOS and a NMOS of width/length (W/L) ratio equals to 2.5. The NAND gate was built with the same structure, except the W/L was kept at a ratio of 1 for maximum power delivery. Both gates are built upon the BSIM4 MOSFET models [39] and biased with a constant DC voltage source of 2 V. The ground truth data is obtained by running transient simulation for a 100 ns period with 10 ps time step. To ensure fairness in evaluation, only the first 50 ns of the waveform is used for training and validation while the rest is reserved for testing. The FNN employed in the following sections has three hidden layers, which all use ReLU as activation function due to its sparsity and a reduced likelihood of vanishing gradient. Adam and MSE are chosen as the optimizer and cost function, respectively. After training, the FNN models were tested with various HSL configurations, including the ones that are not available in the training setting. The correlation between the reference waveform and the FNN outputs is quantified by the coefficient of determination, R^2 score, defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{N} \left\| y^{(i)} - \hat{y}^{(i)} \right\|^2}{\sum_{i=1}^{N} \left\| \hat{y}^{(i)} - \bar{y} \right\|^2} \tag{3.1}$$

where $\bar{y} = (1/N) \sum_{i=1}^{N} \hat{y}^{(i)}$. A R^2 score closer to 1.0 implies a strong correlation, meaning the model is producing valid predictions for the test dataset. This metric is essentially a standardizes version of MSE with respect to the variance presented in the reference dataset as $R^2 = 1 - (MSE/\hat{y})$.

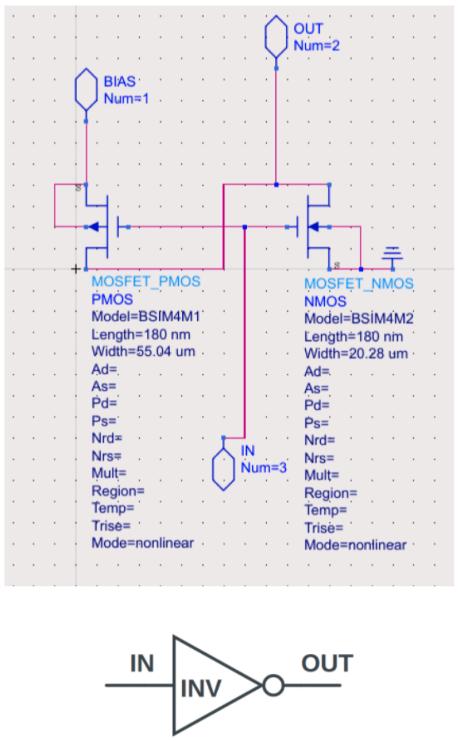


Figure 3.1: Transistor level circuit of CMOS inverter.

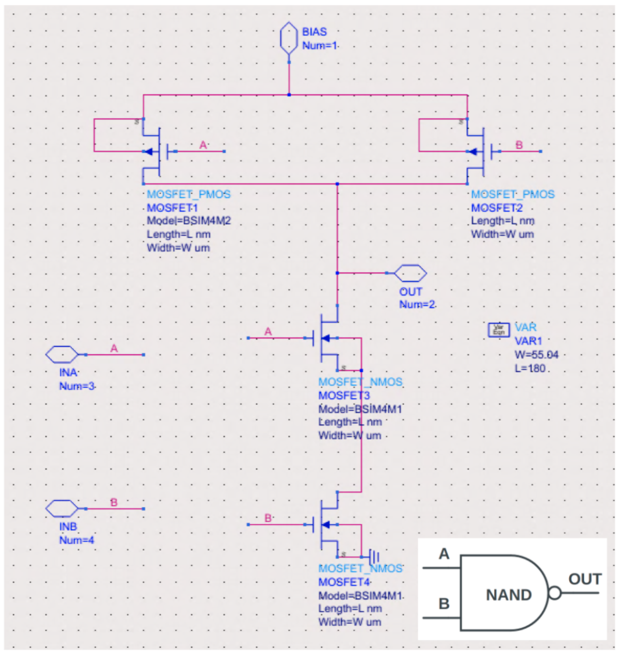


Figure 3.2: Transistor level circuit of NAND gate.

3.2 Trainig Environment

3.2.1 TX

The FNN TX models are generated using the schematic in Figure 3.3 for CMOSs and in Figure 3.4 for NAND. The labeled inputs V_{IN}, V_A and V_B are pseudo-random bit sequence (PRBS) and the labeled nets V_{TX} and V_{RX} are the expected outputs of the TX models. For demonstration purpose, each gate is trained with two protocols: Geometric and S-parameter (VF).

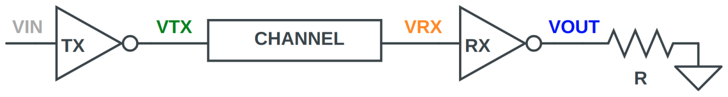


Figure 3.3: HSL configuration of cascading two CMOSs, channel and termination.

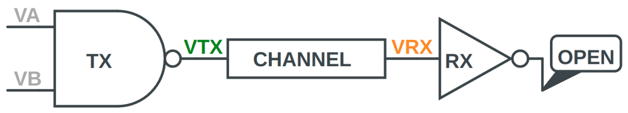


Figure 3.4: HSL configuration of cascading NAND, channel and CMOS.

In the geometric approach, the channel is a microstrip line of width w and length l. To ensure the robustness of the models, w, l and settings of PRBS are linearly swept with values shown in Table 3.1, where V_h/V_l denotes the highest/lowest voltage of PRBS dynamic range and B_r is the PRBS data rate. After collecting the voltage readings at the desired nets from ADS batch simulation, the data is sent to FNN for training. With a memory length of $M = 300$, the number of columns in the input matrix $\boldsymbol{X}$ for CMOS and NAND are 302 and 602, respectively. The extra columns in the NAND are the result of dual-input. For a multi-inputs gate, memory sequence of each excitation is concatenated horizontally to ensure all information is recorded by the FNN. The number of neurons in each layer of FNN is designed to strictly follow the width of $\boldsymbol{X}$. For example, under the geometric protocol, the neurons adopted in the CMOS model are $(M + 2)$, $\frac{M+2}{2}$, 50 , 25 and 2 from input layer to output layer accordingly.

Table 3.1: Range of sweeping values in ADS batch simulation

Parameter	w (mm)	l (mm)	V_h (V)	V_l (V)	B_r (Gbps)
Value	[0.25, 6]	[0, 45]	[1, 5]	$-V_h$ & 0	[1, 10]
Step	0.25	15	1	N.A.	1

In the VF approach, the channel is characterized by S-parameters extracted from both microstrip lines and discontinued transmission lines. The geometry of the channel is varied so that the selected S-parameter samples can cover a wide range of poles as shown in Figure 3.5. During the VF, the max number of poles are set to be 4 and the fitted results are padded with zero if fewer poles are used. Since the sequential order of poles is critical for FNN but not for VF, the poles coming from VF are re-organized to have a descending order, followed by their corresponding residues. To speed up the training, it is generally recommended to have the input features normalized to a set with a mean close to zero. Given that the flattened poles, residues and constants $[a, r, d]$ are usually multiple orders of magnitude off from this standard, normalization with log_{10} is performed before appending the set to the memory sequences. Furthermore, one might notice the channel-absent case is not available in the pre-sampled poles. This is because VF cannot fit this special case under the constrain of passivity enforcement. To add it into the training set, an extra flagging feature is applied: A padding of zeros and a flag of zero represents the without channel case; $[a, r, d]$ followed by a flag of one resembles the with channel case.

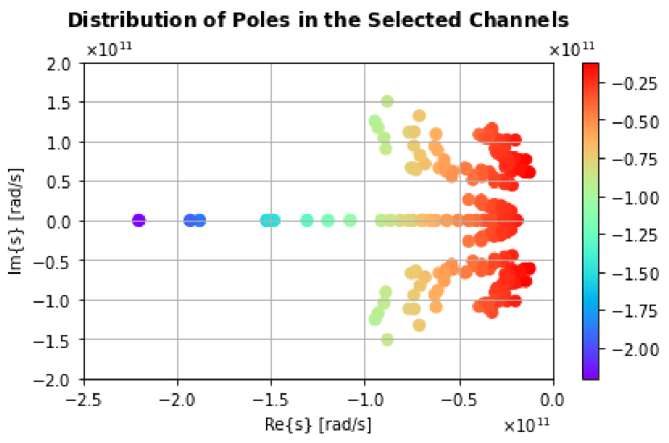


Figure 3.5: Poles used for FNN TX training with VF.

3.2.2 RX

The FNN CMOS RX model is generated using the schematic in Figure 3.3 by taking voltages of V_{RX} and V_{OUT} as references. The channels designed for VF protocols are re-used here to ensure that the RX model is capable of interpreting noisy V_{RX}. During the ADS batch simulation, the PRBS settings are varied in the same manner as above and the resistive termination $r_t = [5, 100000]$ Ω is swept logarithmically at a step of 3 Ω/decade. Since the RX model only oversees a resistor, M is chosen to be 50, which greatly reduces the size of layers in FNN. Normalization with log_{10} is applied on the termination feature before concatenating it to the memory matrix.

3.3 HSL Configuration with NRZ Excitation

Before showing the cascade-ability of the FNN models, the accuracy of a single building block is verified. Figure 3.6 illustrates the test case of a stand-alone CMOS connected to an open termination with a PRBS excitation of $V_l = 0$ V, $V_h = 4$ V and $B_r = 5$ Gbps. In this case, the CMOS RX model is employed with the termination condition set to near open ($r_t = 100000$ Ω). The V_{OUT} prediction has a R^2 score of 0.989, which confirms the validity of the FNN model.

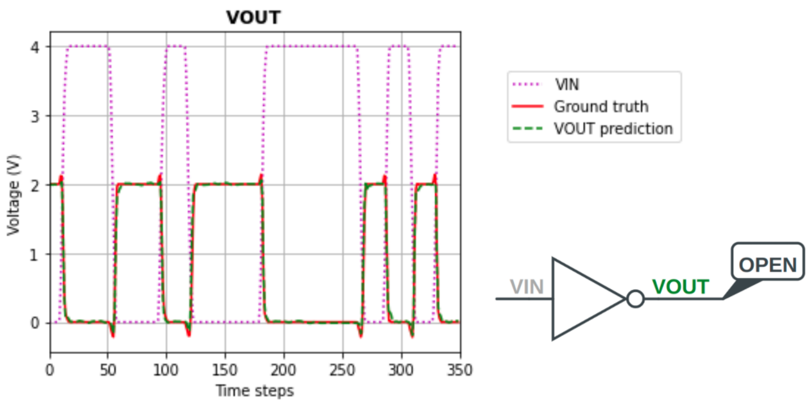


Figure 3.6: FNN prediction for a stand-alone CMOS with open termination.

Next, a comparison is drawn between the two TX protocols. For the HSL configuration shown in Figure 3.7, either of the FNN NAND TX models could be used to predict the voltage waveform at node V_C:

- Geometric: Set $l = 0$ mm and $0.25 \text{ mm} \leq w \leq 6$ mm.

- VF: Set the padding and the flag features to zeros.

Since this example is a special case of directly cascading TX with RX, the two outputs yield by the TX models essentially describe the same node: $V_{TX} = V_{RX} = V_C$. Given a PRBS excitation of $V_l = 0$ V, $V_h = 3$ V, $B_{r(A)} = 3$ Gbps and $B_{r(B)} = 4$ Gbps, the modeling results from the two protocols are plotted and compared as shown in Figure 3.8. Although tiny variations exist between the two models, both protocol yield prediction of $R^2 = 0.998$ in respect to the reference voltage. Hence we can safely assume that either of the predicted V_C is valid as input to the CMOS RX model. After feeding V_C from the VF TX model to the RX model, a cascading chain of FNN models is established where good agreement is found between the predicted and true value of V_{OUT} (see Figure 3.9). In this case, the R^2 score is 0.996.

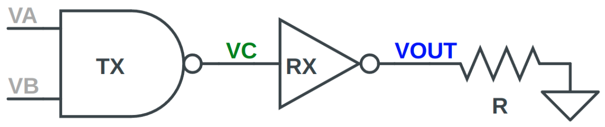


Figure 3.7: HSL configuration of cascading NAND, CMOS and termination.

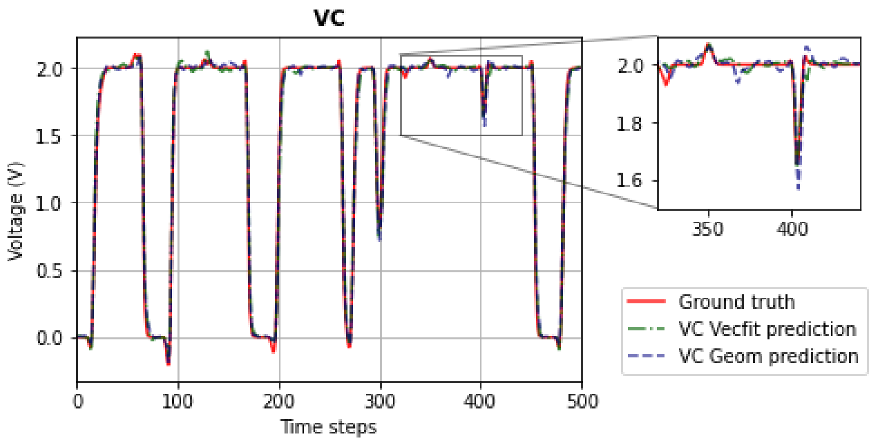


Figure 3.8: Comparison of NAND TX model predictions on V_C.

Moreover, a test configuration consisting of multiple CMOS gates is evaluated to further demonstrate the cascade-ability of the FNN models. As shown in Figure 3.10, a PRBS sequence of $V_l = 0$ V, $V_h = 5$ V and $B_r = 1$

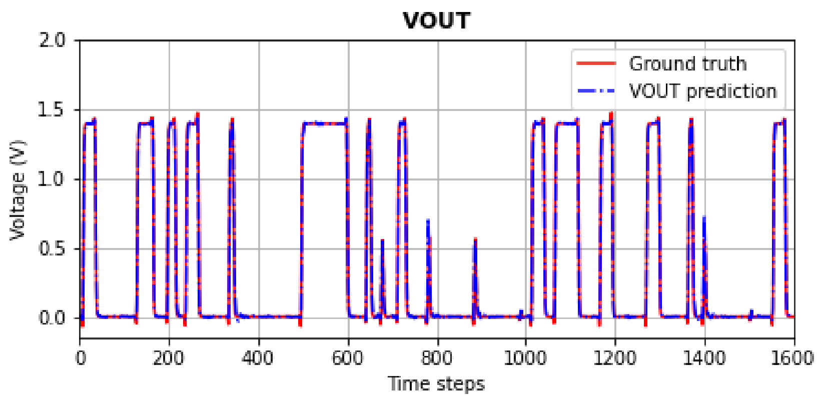


Figure 3.9: FNN prediction of V_{OUT} in Figure 3.7 with termination $r_t = 50\ \Omega$.

Gbps is fed through a HSL followed by an extra CMOS and a 2000 Ω termination. The channel within the HSL is a microstrip line of $w = 6$ mm and $l = 45$ mm. First, the CMOS TX model with geometric protocol is used to make predictions on V_{TX} and V_{RX}. Then, the predicted V_{RX} is delivered to the CMOS RX model for V_{OUT} prediction. Although there is no termination directly connected with V_{OUT}, RX sees the extra CMOS almost as an open load. Due to this, the termination feature is set to max so the prediction matches with its reference. At the end, V_{OUT2} is obtained by forwarding the predict V_{OUT} through the RX model with $r_t = 2000\ \Omega$. Predicted waveforms at each node are presented in Figure 3.11. The R^2 scores for V_{TX}, V_{RX}, V_{OUT} and V_{OUT2} are 0.999, 0.998, 0.989 and 0.986, respectively.

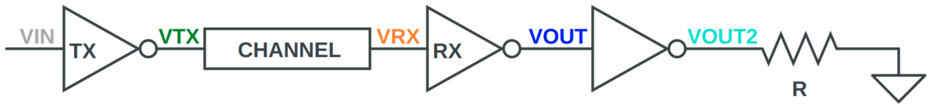


Figure 3.10: A cascading chain of multiple CMOSs, channel and termination.

Lastly, the FNN CMOS models are cascaded to reproduce the eye digram at node V_{OUT} in the HSL schematic shown in Figure 3.3. Given a PRBS excitation of $V_l = -5$ V, $V_h = 5$ V and $B_r = 10$ Gbps along with a S-parameter characterized channel, the voltage waveforms predicted by the FNN models are shown in Figure 3.12. Fairly good correlation is achieved with a R^2 score of 0.983, 0.986 and 0.982 for V_{TX}, V_{RX} and V_{OUT}.

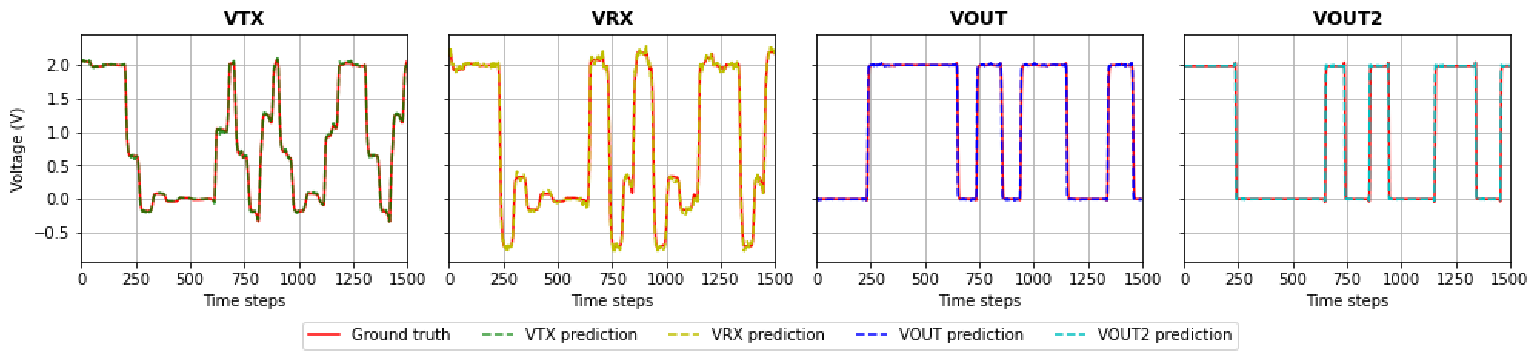


Figure 3.11: FNN predictions of V_{TX}, V_{RX}, V_{OUT}, V_{OUT2} in a three CMOSs cascading chain.

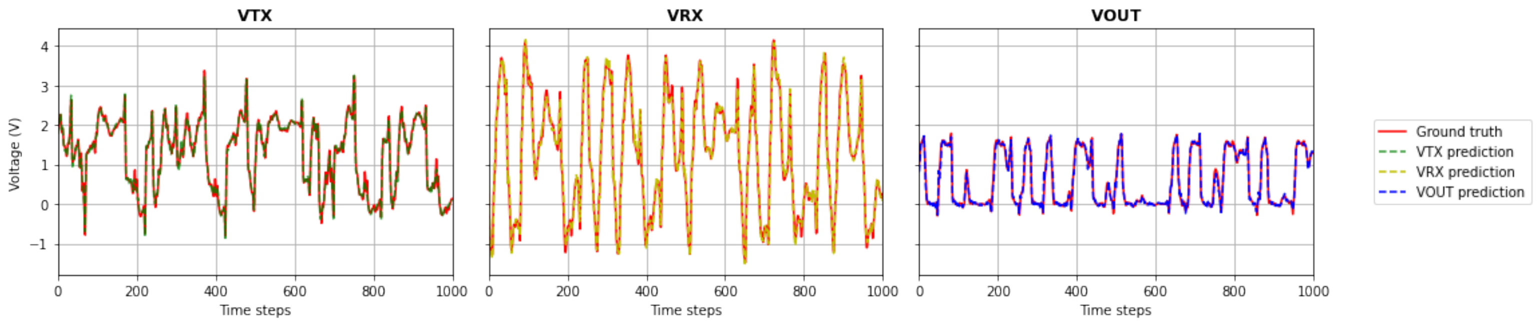


Figure 3.12: FNN predictions of V_{TX}, V_{RX} and V_{OUT} in a standard HSL configuration given a termination of $r_t = 50\ \Omega$.

A close look on the predicted and true values of V_{OUT} is given in Figure 3.13. While there are some eccentric points starring away from the true value, most of the predictions are clustered around the reference line. The eye diagrams shown in Figure 3.14 are constructed by overlaying segments of rising and falling edges within the 50 ns V_{OUT} sequence. As seen in the figure, the shape of the eye in both plots are very similar. After taking the eye height and width measurement, the reference from ADS has a reading of [0.94 V, 59.0 ps] and the FNN predicted eye yields [0.97 V, 58.2 ps]. Table 3.2 summarized the quantitative comparisons between the eyes, where the prediction perfectly matches with the reference. It is worth mentioning that among the 2000 eye diagram simulations, the average speed up factor is 15 if FNN models are used. This demonstrates that for single-end NRZ transient simulation, application of FNN models are promising in terms of both accuracy and time-efficiency.

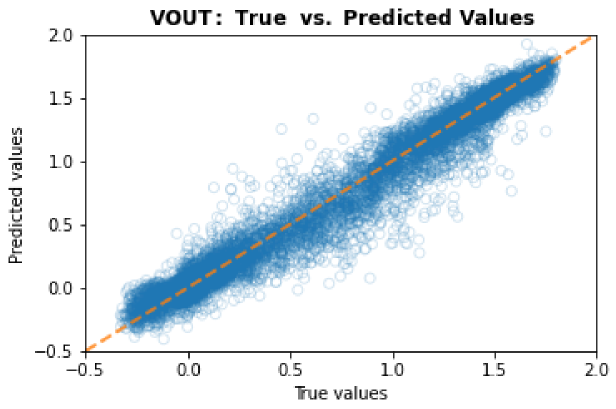


Figure 3.13: FNN Prediction versus its reference for V_{OUT} in Figure 3.3.

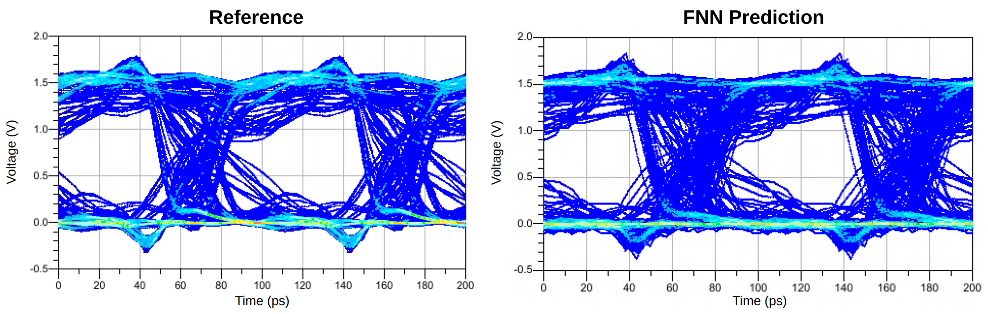


Figure 3.14: Eye diagrams constructed by the reference and the FNN prediction.

Table 3.2: Comparison of eye diagram measurements

Measurement	Eye Width	Eye Height	Zero Level	One Level
Reference	59.0 ps	0.940 V	0.020 V	1.487 V
FNN	58.2 ps	0.970 V	0.013 V	1.493 V

3.4 HSL Configuration with CTLE Equalization

3.4.1 CTLE Equalization Background

As demonstrated in the previous section, with non-ideal aspects of channel, such as impedance mismatches and dielectric losses [40], the signal intercity can be severely degraded and eventually impacts the timing budget. As the eye closes at the end of the HSL, we are observing the “smearing” of the signals, a phenomenon known as *inter-symbol interference* or ISI. The increase in the jitter causes more ISI to appear at the end of the transmission, making it impossible to correctly convert the analog signal back to the digital domain. Therefore, equalization is introduced to reduce the deterministic jitters and to recover the noise margin. This is oftentimes accomplished by compensating the fact that the higher frequency signals are naturally attenuated more than the lower ones. In other words, if the attenuation or loss is consistent through out the designed bandwidth, the frequency dependency of the ISI induced timing jitter is lifted and the eye will no longer collapse.

The equalization technique CTLE is an acronym for *continuous-time linear equalizer* which adopts a one tap circuit that effectively flattens the channel response by boosting the gain in the higher frequency region. In its simplest form, a passive CTLE can be achieved by two sets of parallel RC as shown in Figure 3.15. The resistor acts as an attenuators for the low-frequency signals and the capacitor, at the same time, allows the high-frequency signals to follow directly to V_{EQ}. This combination of RC circuit results in a gain boosting in the high-frequency signals and thereby preventing the eye from collapsing. There are three key parameters that defines the amounts of equalization provided by a passive CTLE, namely the zero frequency, the pole frequency and the DC gain. Depending on the channel’s transfer function, appropriate component values can be calculated by:

$$\omega_z = \frac{1}{R_1 C_1} \tag{3.2}$$

$$\omega_p = \frac{1}{\frac{R_1 R_2}{R_1 + R_2}(C_1 + C_2)} \tag{3.3}$$

$$A_{DC} = \frac{R_2}{R_1 + R_2} \tag{3.4}$$

where ω_z and ω_p are the zero and pole locations, and A_{DC} marks the gain-boost factor at DC. For instance, if the desired bit-rate is 5 Gbps, one should expect a properly designed CTLE to have ω_z = 5 Grad/s, ω_p = 18 Grad/s and $A_{DC} = 0.2$. As a rule of thumb, the zero should be placed at a relative low frequency, where the transfer function starts to decline; the pole should be placed at the Nyquist frequency of the transmission rate; the DC gain should be maximized without violating the passivity. In principle, after applying passive CTLE, the one and zero levels of the eye will be reduced because the gain factor could never exceed 1.0. However, this is a reasonable trade-off given that both the eye height and widths are recovered. If needed, one could also add more poles to achieve multi-band equalization or an amplifier for active gain boosting [41], which are both beyond the scope of this work.

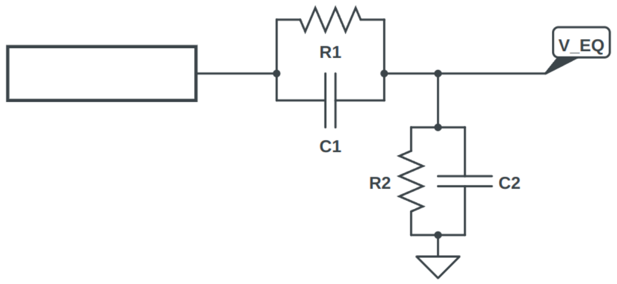
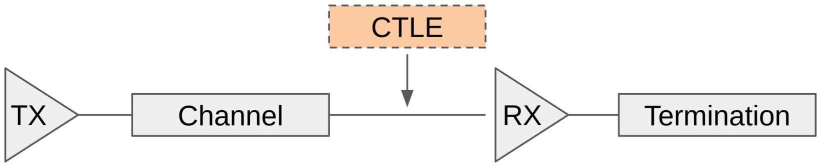


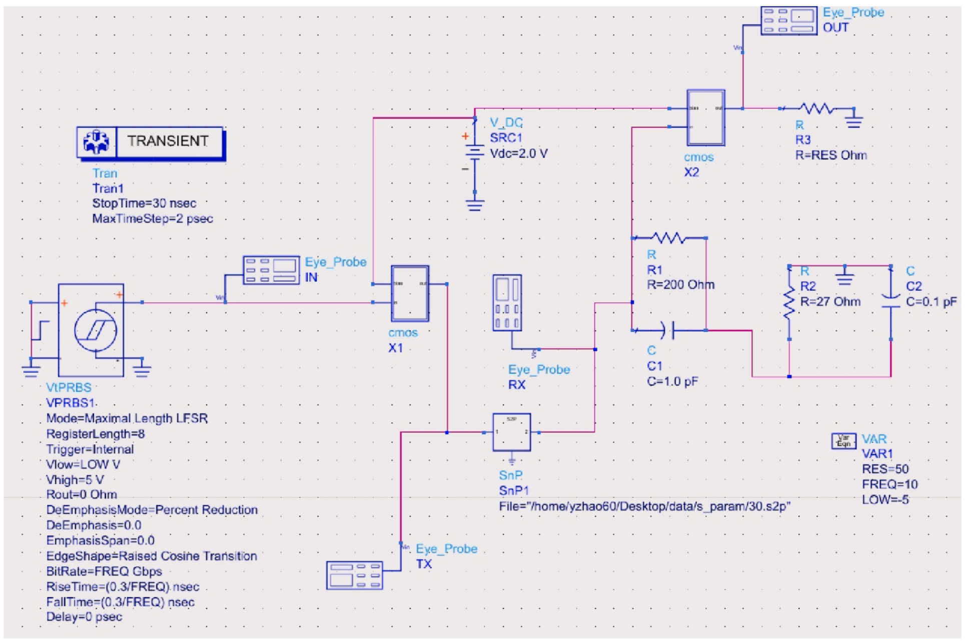
Figure 3.15: High-pass RC circuit for passive CTLE.

3.4.2 Extending FNN TX model with CTLE feature

A passive CTLE circuit of single zero and pole is placed right after channel to improve the eye diagrams at all nodes in the HSL as shown in Figure 3.16. Given a data rate of 10 Gbps, the CTLE is designed to locate the zero at 0.8 GHz, the pole at 6 GHz and a DC gain of 0.213. From Equations (3.2)-(3.4), the corresponding RC values are finalized to R_1 = 200 Ω, C_1 = 1 pF, $R_2 = 27\ \Omega$ and $C_2 = 0.1$ pF. One thing to notice is that there are four variables but only three equations are given, which means that the aforementioned values are not the only solution set for the same level of equalization. This particular topology of integrating the equalizer within the interconnect is equivalent to replacing the lossy channel with a less noisy one. In this sense, the only change with respect to the FNN models falls on the TX end because the RX model is only related to the loading conditions. More specifically, although CTLE will alter voltages at all nodes through out the HSL, it is sufficient to embed CTLE settings only in the CMOS TX model, leaving the trained CMOS RX model untouched.



(a)



(b)

Figure 3.16: Add CTLE to HSL: (a) Equalizer located in the interconnect channel; (b) ADS scheamtic of a 10 Gbps HSL with CTLE.

To account for the changes brought by CTLE, another round of training is performed on the CMOS TX model. Besides the PRBS excitations and S-parameter characterized channels, the extended TX model requires a new input set that describes the CTLE setting: pole, zero and DC gain. Backwards compatibility is ensured by leaving these three keywords blank so the FNN recognize that the CTLE is not activated in this case. Since CTLE is not a non-linear transformation, the memory length and FNN structure were kept the same as in Section 3.2. For training purposes, these settings are append at the end of each row in the input matrix $\boldsymbol{X}$ and a training/testing set of 80 ns/200 ns is generated in ADS.

3.4.3 Validation

Two sets of test are prepared for the newly trained CMOS TX model to demonstrate the effectiveness of the equalization. First, the equalization is turned off and a comparison between the ADS results and the cascaded FNN models' results is drawn; Then, the equalization is turned on and the same comparison is performed to validate the accuracy of the predicted results. In the following examples, the interconnect is a very noisy channel so the benefit of using CTLE stands out. The loading used is a 50 Ω resistor to ensure pulse symmetry. The excitation is a -5$\sim$5 V PRBS sequence of 10 Gbps.

With CTLE off, the CMOS TX model and CMOS RX model function the same way as shown in Section 3.3 except a dummy equalization setting is required at the TX. As expected, good correlation between ADS and FNN prediction is found at nodes V_{TX}, V_{RX} and V_{OUT} as shown in Figure 3.17. Their R^2 scores are 0.997, 0.997 and 0.996 respectively. Comparing to the previous NRZ results, the improvement in R^2 (>0.99) is due to the reduction in training time step (see Chapter 4 for more details). By overlaying the 200 ns test data stream, the eye diagrams at each node are generated and plotted as shown in Figure 3.18.

With CTLE on, the CMOS TX model is fed with the CTLE settings that open up the eye at 10 Gbps. After the channel, the CMOS RX model takes in the V_{RX} prediction and yields V_{OUT} correspondingly based on the loading condition. As shown in Figure 3.19, the R^2 scores at each nodes are 0.998, 0.999 and 0.999 respectively. Given that CTLE makes the channel look more "clear", it is no surprise that the scores are higher in this test case. A comparison between ADS and FNN for the CTLE eye diagrams is illustrated in Figure 3.20. The eyes are all opened up with much less jitters.

A quantitative analysis on the eye diagrams above are given in Table 3.3 and Table 3.4. From the measurement, a few conclusions can be made:

- Adding CTLE "cleans" the waveforms at all nodes (V_{TX}, V_{RX} , V_{OUT}). Although the zero and one levels are reduced due to passivity, the eye width and height are improved because of the filtering effect.
- While the CTLE provides considerable boost to the high-frequency components, it is impossible to boost the signal and noise at a different level. Therefore, even if we maximize the gain in the passive CTLE, the signal-to-noise ratio (SNR) stays the same. To further improving the jitter treatment, it is recommended to introduce digital equalization strategies like decision feedback equalization (DFE) along with CTLE.
- The CMOS RX adds nonlinearity to the HSL by forcing the analog bits to be either high (gate voltage) or low (ground voltage). This effectively opens up the eye when signal passes the RX and thereby giving more clearance in the time and amplitude margins.
- The cascade-ability of the FNN models is preserved in terms that the RX model remains unchanged (no additional training needed) when the TX/RX models are cascaded together in the CTLE ON case. Although the CMOS RX model was trained only with the V_{RX} and V_{OUT} waveforms in the CTLE OFF case, it could still handle the CTLE-modified V_{RX} because the CTLE transformation is a purely linear process.
- The FNN predictions yields eye 25 times faster than the transient solvers. The trade-off errors are all below 6% as shown in Table 3.5.
- The FNN models always underestimate the eye, which is indeed preferred by the HSL designers. An underestimation brings tighter time budget, meaning that the links are subject to over-design. On the other hand, if the predictions overestimate the eye, false-design might be made and that leads to much more severe consequence.

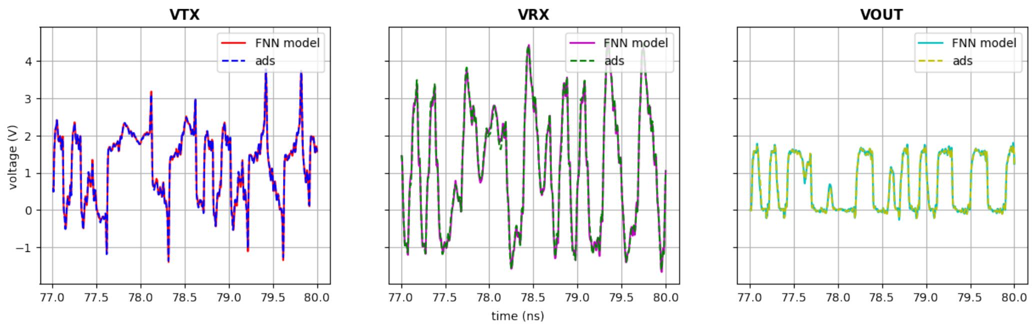


Figure 3.17: FNN predictions of V_{TX}, V_{RX} and V_{OUT} without CTLE.

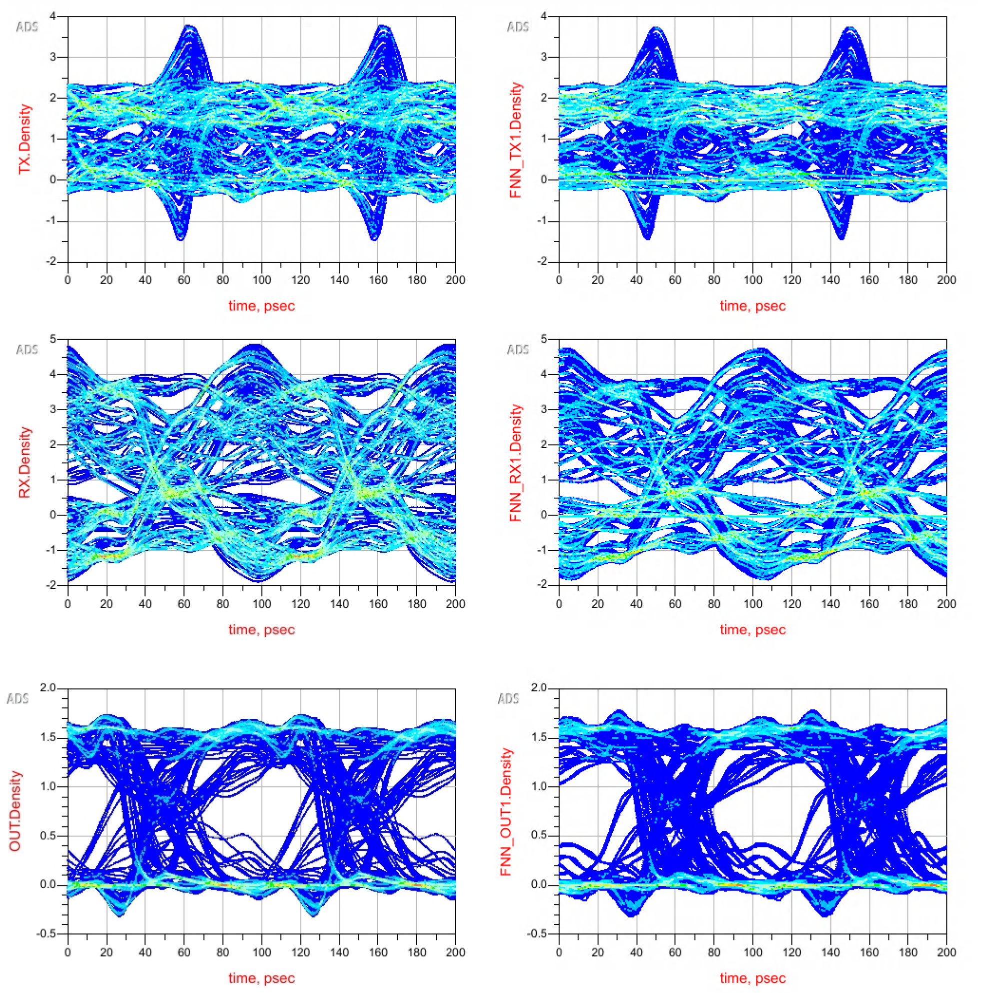


Figure 3.18: Eye diagrams without CTLE. Left: ADS references. Right: FNN predictions. Top to Bottom: V_{TX}, V_{RX} , V_{OUT}.

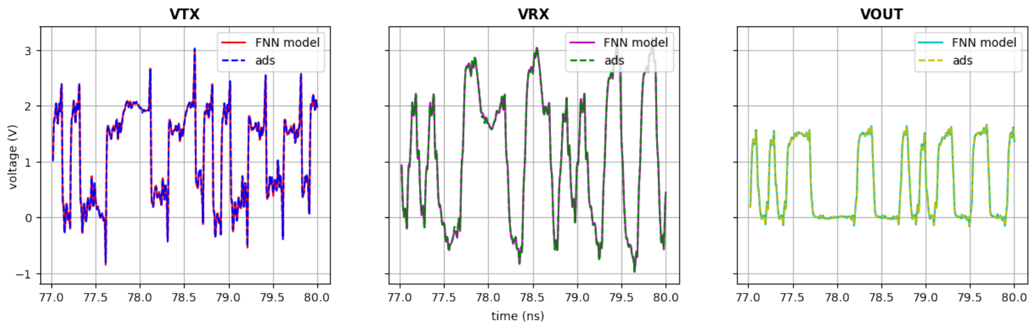


Figure 3.19: FNN predictions of V_{TX}, V_{RX} and V_{OUT} with CTLE.

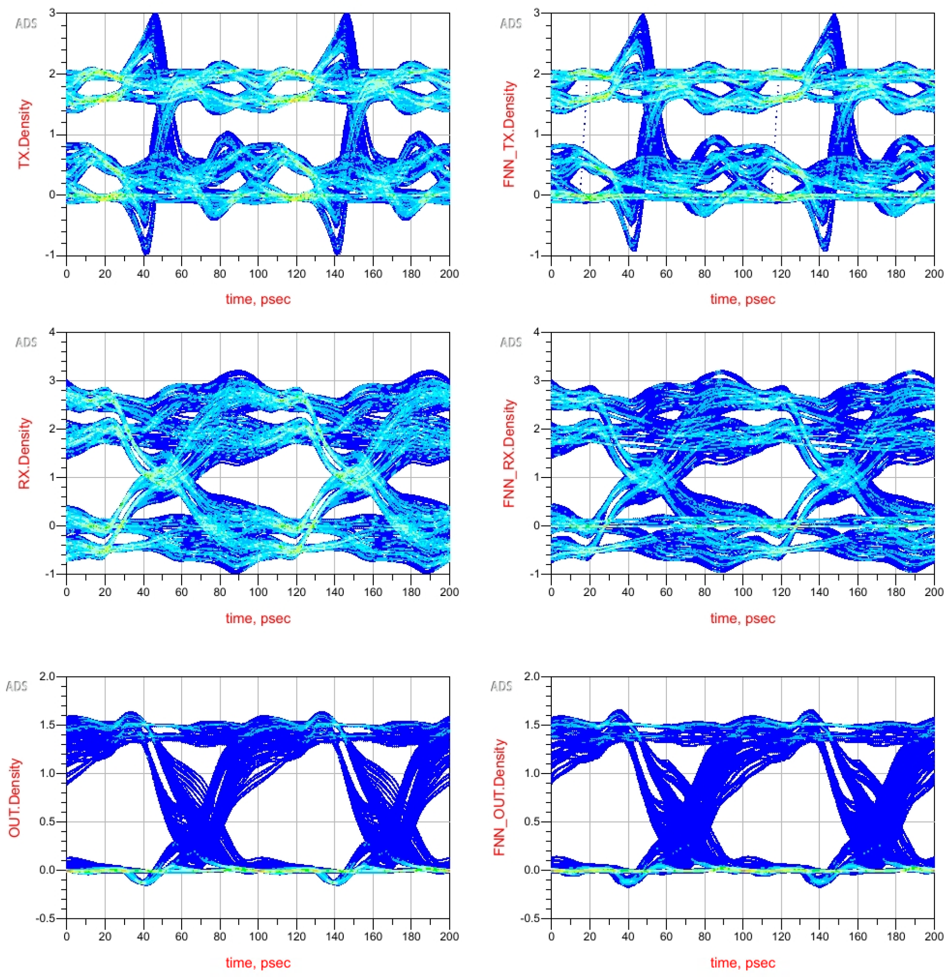


Figure 3.20: Eye diagrams with CTLE. Left: ADS references. Right: FNN predictions. Top to Bottom: V_{TX}, V_{RX} , V_{OUT}.

Table 3.3: Quantitative comparison of CTLE OFF eye diagrams in Figure 3.18

Measurement		Eye Width	Eye Height	Zero Level	One Level
Reference	VTX	N/A			
	VRX	15.0 ps	0.000 V	-0.591 V	3.153 V
	VOUT	48.4 ps	0.930 V	-0.003 V	1.515 V
FNN	VTX	N/A			
	VRX	15.8 ps	0.000 V	-0.598 V	3.110 V
	VOUT	50.2 ps	0.950 V	-0.003 V	1.527 V

Table 3.4: Quantitative comparison of CTLE ON eye diagrams in Figure 3.20

Measurement		Eye Width	Eye Height	Zero Level	One Level
Reference	VTX	87.8 ps	0.470 V	0.282 V	1.733 V
	VRX	69.4 ps	0.750 V	-0.185 V	2.244 V
	VOUT	69.0 ps	1.020 V	0.005 V	1.412 V
FNN	VTX	87.4 ps	0.460 V	0.279 V	1.730 V
	VRX	69.0 ps	0.760 V	-0.190 V	2.251 V
	VOUT	67.2 ps	1.020 V	0.005 V	1.413 V

Table 3.5: Precentage error between reference and FNN for eye diagrams in Figure 3.18 and Figure 3.20

Percentage Error (%)		Eye Width	Eye Height	Zero Level	One Level
CTLE OFF	VTX	N/A			
	VRX	5.33	0	1.18	1.36
	VOUT	3.71	2.15	0	0.79
CTLE ON	VTX	0.40	2.12	1.06	0.17
	VRX	0.57	1.33	2.70	0.31
	VOUT	2.60	0	0	0.07

3.5 HSL Configuration with Differential Signaling

3.5.1 Differential Signaling Background

In contrast to the single-ended signaling in the previous writing, differential signaling transmits information utilizing two complementary signals. Two drivers, when in odd-mode, send opposite-sign voltages down the differential pair and the eye measurement is taken as the voltage difference between the two signal lines. When strong coupling occurs (e.g. trace separation is small), the differential pair refers to each other as return path, reducing the chance of seeing discontinuities in their common reference plane. This technique is particularity useful when the transmission lines need to cross a etched gap in the ground plane: If single-ended line is used, the signal would face a huge impedance mismatch and also multiple voltage reflections. In a standard differential pair design where the transmission lines are perfectly symmetric, the voltage ripples everywhere along the lines would be canceled due to the subtraction. In this sense, the differential pair is known for its immunity to crosstalk and electromagnetic interference (EMI). Ideally, the balanced pair picks up the same amount of crosstalk from nearby lines and the crosstalk eventually vanishes when we calculate the pair's potential difference. Similarly, although each line in the pair differential still creates EMI, their fields are opposite in polarity and equal in magnitude, meaning that they got canceled out when radiating together.

The only downside relating to this method is the routing complexity. As the name suggests, more board space is required when one data stream is divided and transmitted with two traces. The layout designers have to fine-tune the differential traces to have the same length and width to ensure a balanced transmission from the TX to RX. Especially with manufacture imperfection, some of the common-mode voltages would sneak in and degrade the performance of the differential signaling. Thus, it is critical for SIPI engineers to evaluate the trade-off before converting the single-ended lines to differential signaling.

3.5.2 FNN Modeling with Differential Signals

Since differential signaling requires a pair of positive/negative (P/N) pins to represent one data stream, both FNN TX and RX models need to be modified accordingly to adapt the change. For the FNN TX model, the input PRBS excitation is now splitted into V_{IN_P} and V_{IN_N}. The outputs associated with the model, the V_{TX} and V_{RX}, are now expanded to be V_{TX_P} ,

V_{TX_N}, V_{RX_P} and V_{RX_N}. For the FNN RX model, similar concept is applied by allowing separate P/N inputs and an expended output composed of V_{OUT_P} and V_{OUT_N}. These modifications are performed on the input and output layers of the FNN structures by adding additional neurons to accommodate the extended I/O information. A differential channel is designed and placed in-between TX and RX as shown in Figure 3.21. Since the channel now contains additional information like trace asymmetry and separation, the TX protocol is extend accordingly to accept those features.

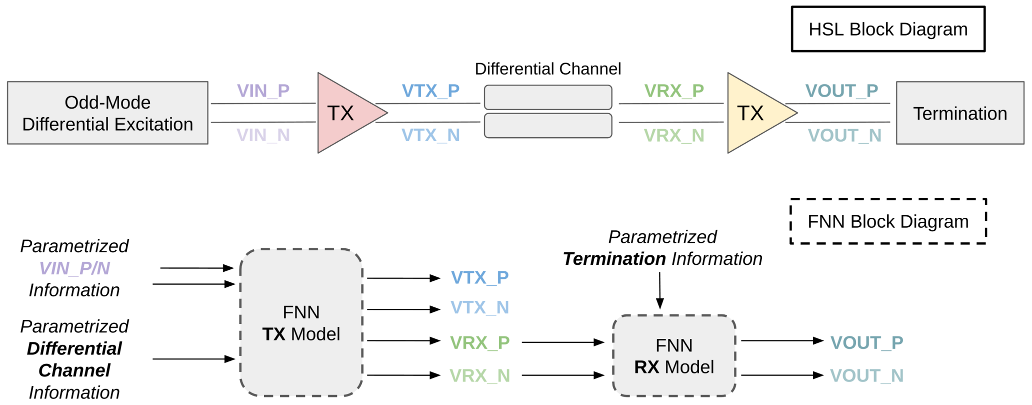


Figure 3.21: The HSL and FNN block diagrams of transforming single-ended signal to differential signals.

As a proof of concept, a simple coupled microstrip line is used with a lose coupling factor. The training is performed up to 20 Gbps with reference voltages taken from a 20 ns transient simulation. Geometric protocol is appended after the input memory sequences with variations on trace width, length and separation. After several rounds of tuning, the memory lengths of the TX and RX model are set to be 250 time steps (250 ps) and 50 time steps (50 ps), respectively. A differential eye diagram test of 200 ns is prepared in ADS to validate the accuracy of the cascaded FNN models.

3.5.3 Validation

Before stepping into the eye digram test, the predicted waveforms at each cascading node is analyzed as shown in Figure 3.22. The R^2 scores for each reference/FNN comparison is shown in Table 3.6. While all the waveforms match well with their references (R^2 >0.98), it is interesting to obverse that on average, the subtracted differential voltage out-performed the P/N pairs. The mismatch in the P/N pairs is due to the ringing ripples, where in general, the FNN requires more memory length to precisely characterize the small voltage fluctuations. However in the differential voltage calculation, these ripples canceled out and with less common-mode noise, the predictions match with the references better.

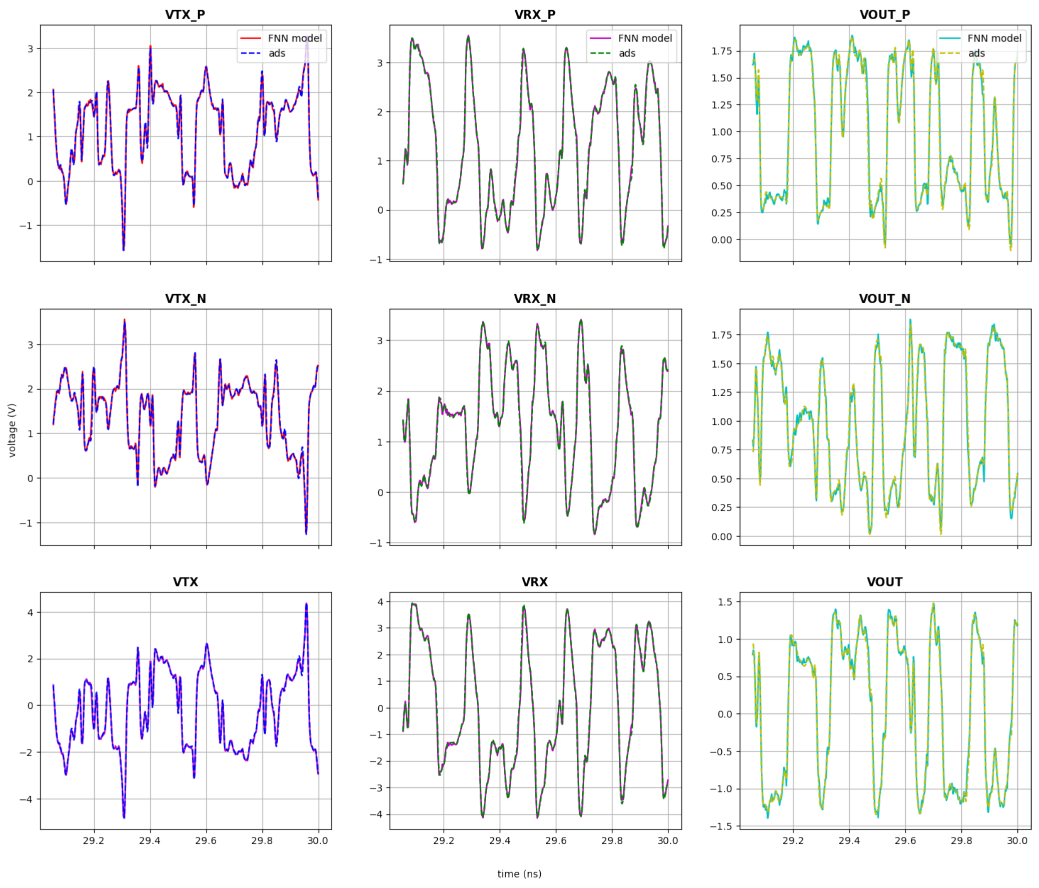


Figure 3.22: FNN predictions for differential signaling test configuration.

Table 3.6: R^2 scores for FNN predicted differential waveforms

R^2	TX	RX	OUT
Positive	0.994	0.995	0.987
Negative	0.994	0.995	0.987
Differential	0.995	0.996	0.992

Moving on to the differential eye diagrams shown in Figure 3.23. Note that the eye at TX is ignored in this test because no opening can be found in the reference due to large channel distortion. The comparison of the eye measurement is given in Table 3.7. Overall, the predictions have high correlation with the references with a speed up factor of 20. Compared to its reference, the prediction turns out to have slightly less eye opening (underestimation), especially in the time margin. One reason for that might be the information loss when the time domain waveform is transformed back to ADS as a *VTDataSource*. More specifically, When the waveforms are prepared by the FNN models, the smallest time step has to be equal to the training time step. Nevertheless, to generate eye diagrams, ADS normally processes this data source with a much smaller step and sometimes performs an internal linear or cubic interpolation depending on the channel response, thereby leading to the discrepancy we observe.

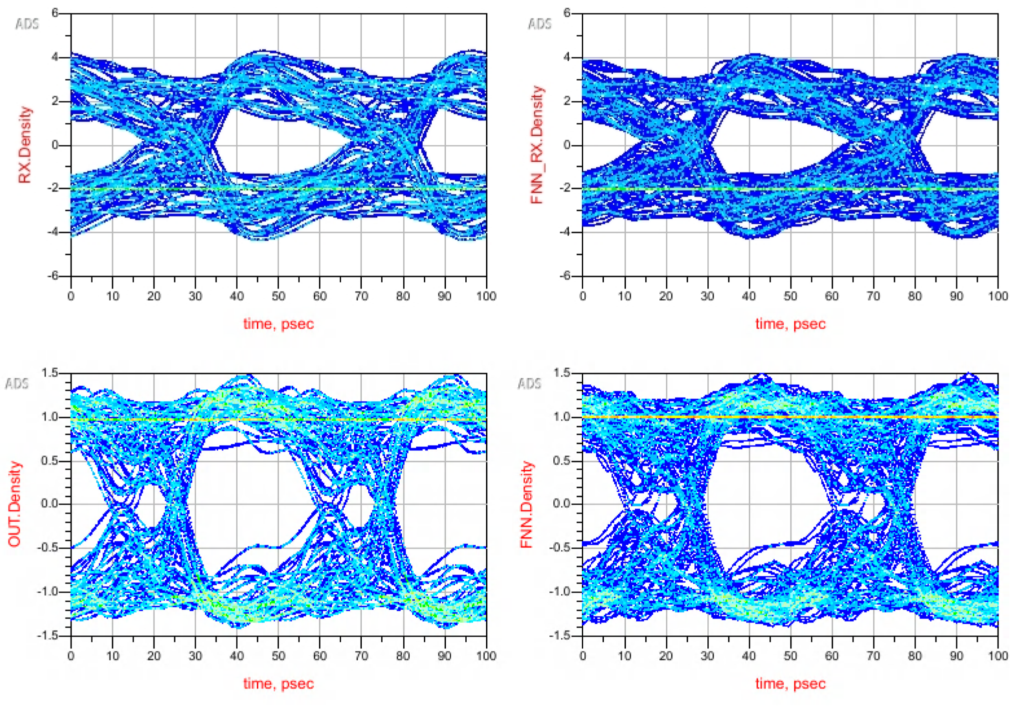


Figure 3.23: FNN predictions for differential eyes at V_{RX} and V_{OUT}. Left: ADS references. Right: FNN predictions. Top: V_{RX}. Bottom: V_{OUT}.

Table 3.7: Quantitative comparison of differential eye diagrams

Measurement		Eye Width	Eye Height	Zero Level	One Level
Reference	VRX	33.2 ps	2.000 V	-2.383 V	2.534 V
	VOUT	31.0 ps	1.010 V	-1.047 V	1.024 V
FNN	VRX	36.2 ps	1.960 V	-2.365 V	2.499 V
	VOUT	32.5 ps	1.000 V	-1.043 V	1.035 V

3.6 HSL Configuration with PAM-4 Excitation

3.6.1 PAM-4 Background

PAM (pulse amplitude modulation) -4 is a modulation scheme that doubles the system's data rate by encoding the bits with four distinct levels of amplitude. In other words, one symbol in PAM-4 contains information of two bits, which effectively reduces the transmitting bandwidth by half comparing to the NRZ scheme. An illustration of the signal levels in NRZ and PAM-4 is given in Figure 3.24, where one can clearly see how double-bits mapping in PAM-4 allows twice the information to be transmitted over the same clock rate. Another crucial factor that motivates PAM-4 development is the demand for cost reduction. Since PAM-4 requires only half as many TX/RX lanes as NRZ, budget can be saved with less connectors, cables and transceivers. The downside of this technology is the limited transmission distance because PAM-4 signaling is more susceptible to noise. For instance, if a symbol is misinterpreted due to delay and distortion from channel, it will result in a false transition with two-bit errors. False triggers in PAM-4 transmission can be quickly visualized by the 4-level eye diagram (three eyes) as shown in Figure 3.25. Similar to the eye of NRZ signals, the jitters in PAM-4 signals are also quantified by the eye levels, heights and widths, except that the three eyes may be asymmetrical and each needs to be equalized independently to mitigate any channel impairments. Overall, PAM-4 is best suited when it comes to high-speed and short-range applications, such as within data centers.

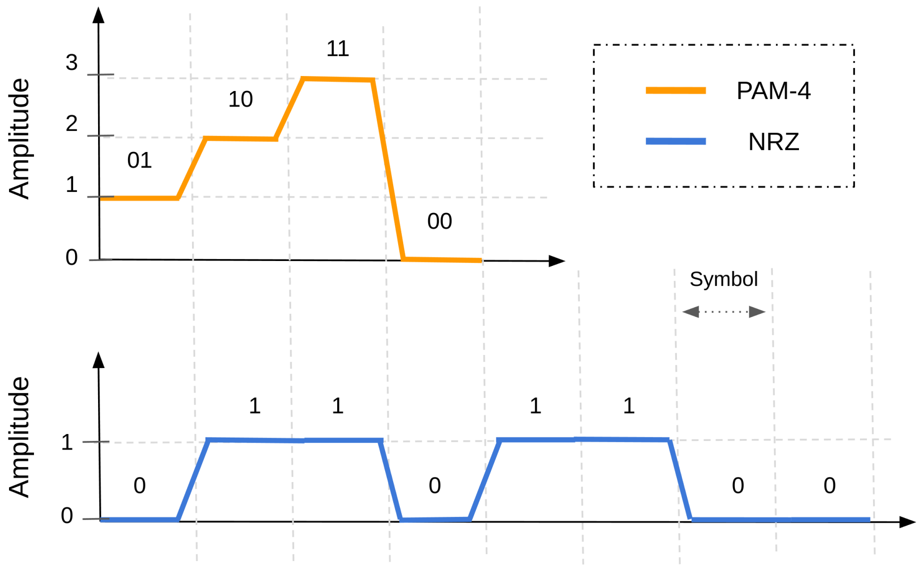


Figure 3.24: Comparison of signal levels in NRZ and PAM-4 for the same bits.

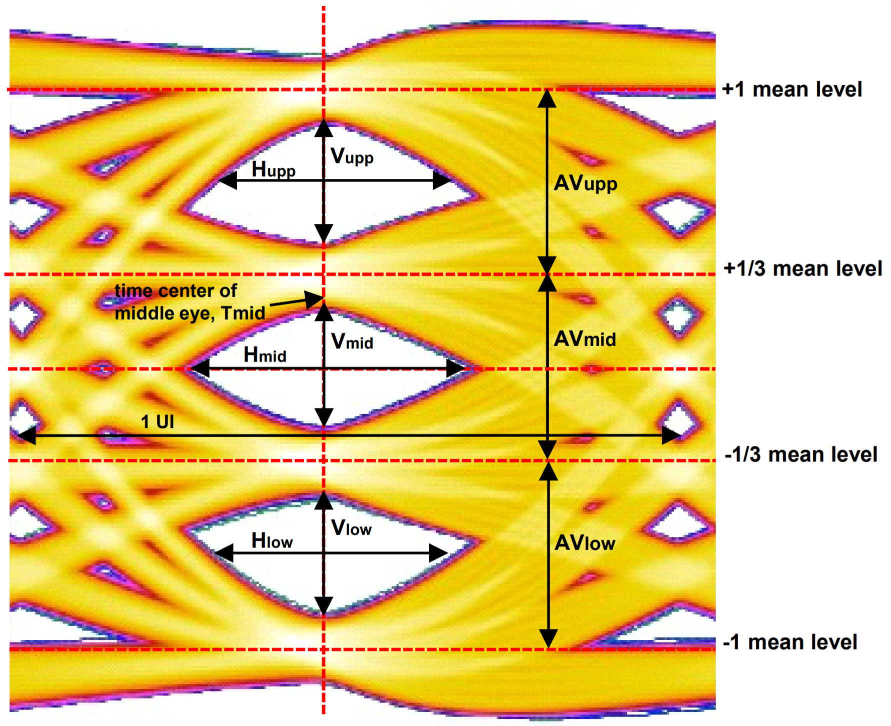
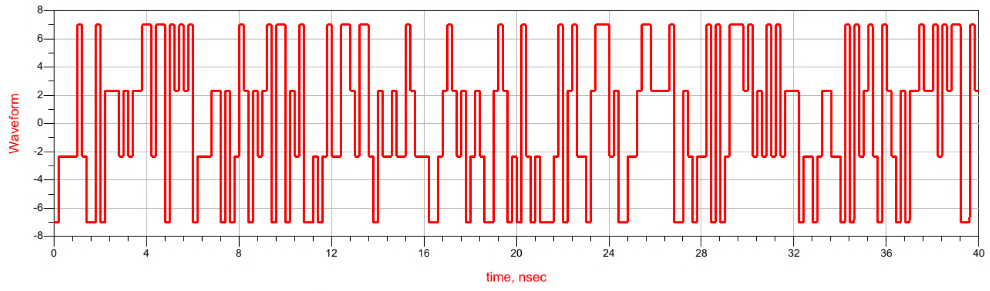


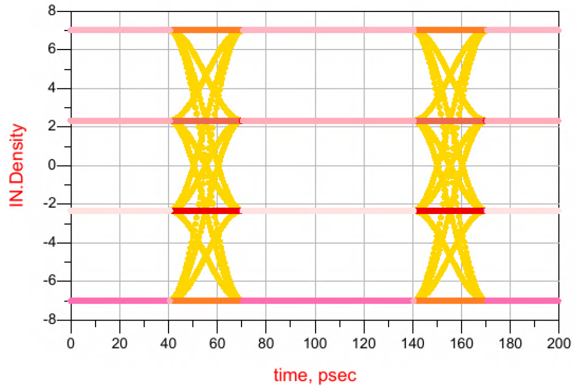
Figure 3.25: PAM-4 eye parameters definition [42].

3.6.2 FNN Modeling with PAM-4 Excitation

The PAM-4 signaling is applied to the differential TX/RX schematic in Figure 3.21 by changing the PAM-level setting in the PRBS source. The excitation now consists of four voltage levels at ± 7 V and ± 2.34 V as shown in Figure 3.26. The gate biases for all CMOS are changed to 5 V to accommodate the higher amplitude inputs. This scenario reflects one of the drawbacks of PAM-4: the high electricity/power consumption. Although in high-power link design this is not a concern, the high gate level clears indicates that PAM-4 may not be a good candidate for applications in the low-battery regimes. After the reference data at each cascading node is generated with PAM-4 excitation, the training procedure for FNN TX/RX models is exactly the same as in the differential signaling case. The models are capable of capturing waveforms up to 10 Gbps, which are equivalent to NRZ-trained models of 20 Gbps . A HSL with geometric characterized channel is prepared for the validation test.



(a)



(b)

Figure 3.26: Example of PAM-4 excitation: (a) A section of PAM-4 source PRBS waveform; (b) Input's eye diagram.

3.6.3 Validation

After training with waveforms gathered from 80ns/1ps batch transient simulations, the FNN CMOS TX/RX models are fed with a 200 ns PAM-4 PRBS sequence to perform validation test between the ground truths (ADS) and the predictions. This test configuration used a geometrically characterized differential channel of $W = 8$ mm, $L = 50$ mm and separation $S = 5$ mm. The load is a shunt 2000 Ω resistor between the positive and negative output terminals. With input and channel information, the CMOS TX model yields predictions on V_{TX_P} , V_{TX_N}, V_{RX_P} and V_{RX_N}. The last two waveforms are fed to CMOS RX model along with the loading condition to complete the waveforms shown in Figure 3.27. The R^2 scores for all the nodes are greater than 0.99. Once again, the calculated differential voltage exhibits slightly better score than the single ended case due to the rejection of common noise.

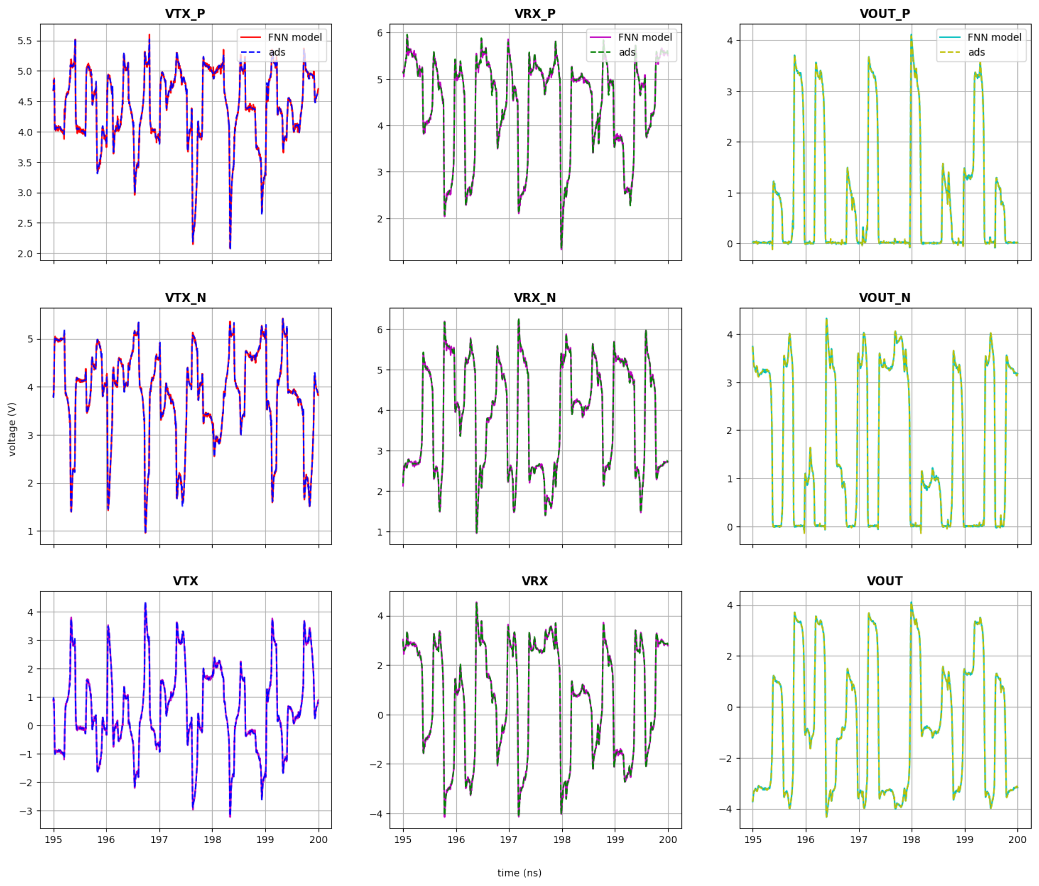


Figure 3.27: FNN predictions for differential PAM-4 signaling test configuration.

Next, the PAM-4 differential eyes are extracted from the simulation and FNN predictions. As shown in Figure 3.28, the predicted eyes match with the reference ones just as well from visual inspection. Although we wish to obtain quantitative measurement on each of the three eyes individually, ADS does not offer this option for transient simulation. The PAM-4 measurement is an exclusive function reserved for simulations with *ChannelSim* block. Unfortunately, transferring this schematic to *ChannelSim* distorts the nonlinearity in the transistor-level inverters and thereby resulting in unreasonable outputs. Therefore, a compromise is made by comparing the mean of the eye widths and heights as shown in Table 3.8. With the percentage errors lower than 6%, we conclude that the FNN PAM-4 models demonstrate good trace-ability at before and after RX nodes. It is also worth mentioning that the average acceleration factor in the PAM-4 tests is approximately 36. For each channel and loading condition, ADS took around 1 min to resolve the eye while FNN only needs 2 sec.

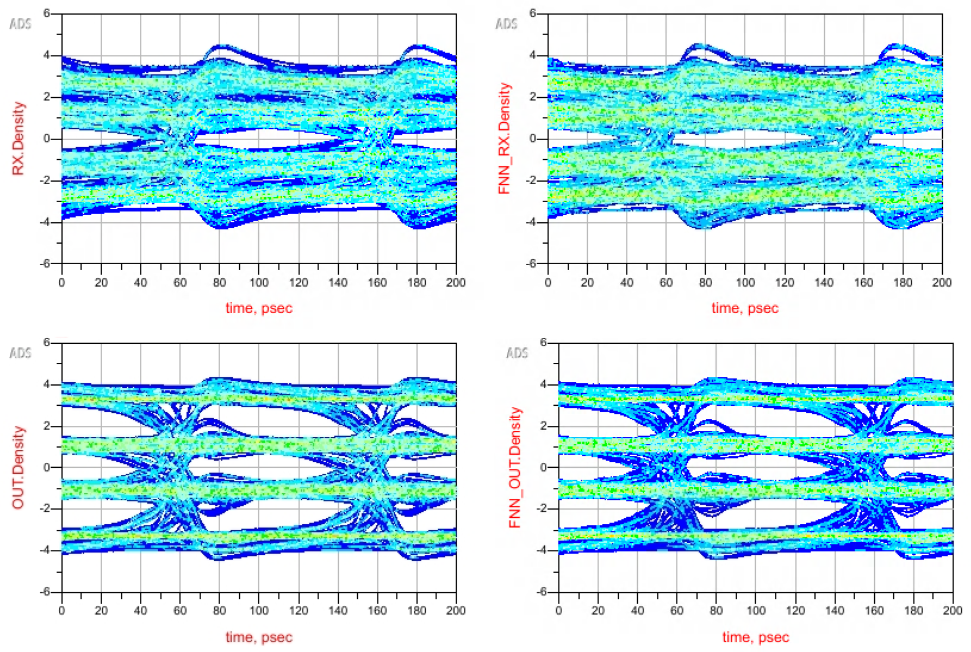


Figure 3.28: FNN predictions for PAM-4 differential eyes at V_{RX} and V_{OUT}. Left: ADS references. Right: FNN predictions. Top: V_{RX}. Bottom: V_{OUT}.

Table 3.8: Quantitative comparison of PAM-4 differential eye diagrams

Mean of Eyes		Width	Width Error	Height	Height Error
VRX	Reference	75.0 ps	4.67 %	0.85 V	5.88 %
	FNN	71.5 ps		0.80 V	
VOUT	Reference	75.5 ps	1.32 %	1.08 V	5.56 %
	FNN	74.5 ps		1.02 V	

3.7 Summary

In this Chapter, we walked through the FNN TX/RX training process for two nonlinear transistor level gates: CMOS and NAND. The trained models are demonstrated to be cascade-able in various HSL configurations, including but not limited to NRZ, CTLE, differential signaling and PAM-4. Good correlations are found between the ground truth and the FNN predicted waveforms at each cascading node. By overlaying the predicted waveforms, we observed that percentage error in eye diagram analysis is lower than 10%. With the same level of accuracy, the average simulation time taken by the FNN models is approximately 20 times faster compared to the traditional SPICE approach. Last but not least, depending on the use case, the FNN modeling algorithm can be easily expanded to include more features by appending the extra information to the model's training input sequence.

CHAPTER 4

DISCUSSIONS

4.1 Alternative Implementation

Besides the coding schema listed in Appendix B, the FNN models could also be generated with the alternative python implementations described in this section. Accuracy wise, there is no essential difference between the main implementation and these alternations since both of them adopt the same training methodology. Structure wise, this section servers as a hint for those who wish to further improve the algorithm of the FNN models from the computational efficiency perspective.

4.1.1 Cost Function

As mentioned in Chapter 2, it is up to the designers to decide a proper cost function that classifies the best weights and bias approximation from the other suboptimal solutions. Since the only hard requirement is to guide the training to the correct direction, the choice of the cost function is definitely more than just the MSE function. In fact, given that the test metric is measured with R^2 score, one should expect a slightly better model if the cost function is implemented based on the R^2 equation.

To implement this change, we need to add a custom loss function to the existing script because the R^2 loss is not one of the default functions provided by the Pytorch package. More specifically, line 6 in Listing 12 and line 7 in Listing 17 need to be replaced by a new loss function definition shown in Listing 1, of which is coded based on the R^2 formula given in Equation (3.1).

```python
def r2_loss(output, target):
    target_mean = torch.mean(target)
    ss_tot = torch.sum((target-target_mean)**2)
    ss_res = torch.sum((target-output)**2)
    r2 = 1 - (1 - ss_res / ss_tot)
    return r2
```

Listing 1: Define R^2 loss function

This R^2 loss function is then called at each training and validation loop as shown in Listing 2. For CPU only users, the lines relating to *cuda* must be omitted. Different from using the Pytorch default loss functions, the gradient of the tensors are not computed automatically. Line 7 forces the NN to record the current loss gradient and line 9 uses this data to perform backward propagation. Also note that in this case Pytorch might accidentally accumulates the gradients on all subsequent backward passes. To avoid that, line 8 is introduced so that the optimizer is initialized at each batch.

```python
1   model.train()
2   for data, label in train_loader:
3       data = data.to('cuda', non_blocking=True)
4       label = label.to('cuda', non_blocking=True)
5       target = model(data)
6       train_step_loss = r2_loss(target, label)
7       train_step_loss.required_grad = True
8       optimizer.zero_grad()
9       train_step_loss.backward()
10      optimizer.step()
11      train_loss += train_step_loss.item()
12
13  model.eval()
14  for data, label in valid_loader:
15      data = data.to('cuda', non_blocking=True)
16      label = label.to('cuda', non_blocking=True)
17      target = model(data)
18      valid_step_loss = r2_loss(target, label)
19      valid_loss += valid_step_loss.item()
```

Listing 2: Apply R^2 loss function in the training and validation loop

A test of implementing R^2 loss function is carried out using the TX/RX reference data extracted from the NRZ schematic in Figure. 3.3. First, convergence of training with R^2 is confirmed as shown in Figure. 4.1. The number of epochs needed is similar to the one in the MSE approach: Both loss functions converged within 100 epochs. Second, the trained models are put to the cascading test and the modeling accuracy is compared as shown in Figure. 4.2. From visual inspection, the R^2 trained models exhibit very high correlation with the reference waveforms. From examining the cascaded R^2 scores, the R^2 trained models have very similar yet slight better performances than the ones trained with MSE loss. Overall, we conclude that the MSE and the R^2 loss functions can be used interchangeably as long as the implementation syntax is correct.

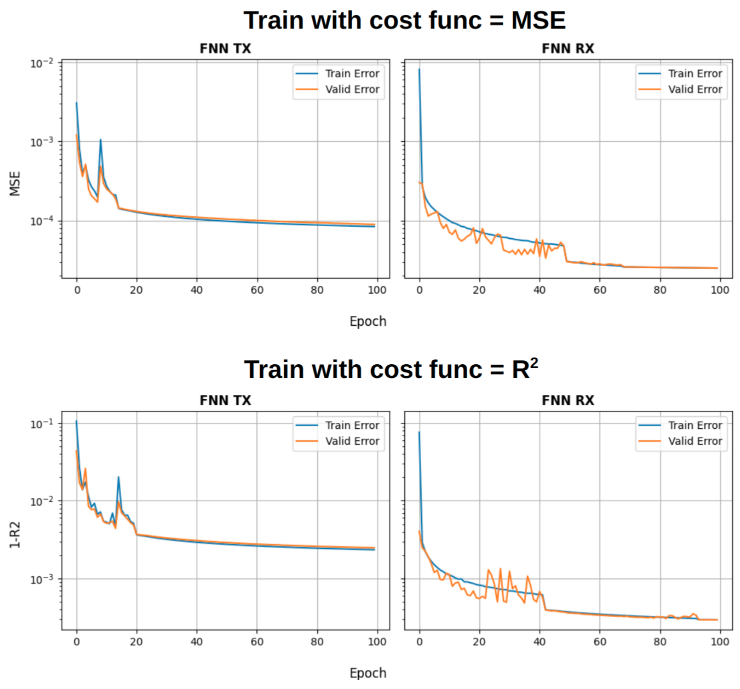


Figure 4.1: Comparison of training convergences between the MSE and R^2 loss functions.

4.1.2 Model Order Reduction

Generally, the complexity of a FNN is measured by the number of kernels in the structure; that is, the total count of neurons in each layer. Here in this work, we can simplify this concept to the column width of the input matrix $\boldsymbol{X}$, which is the sum of the memory length and the protocol parameters' length. Since an overly lengthy input could significantly affects the training efficiency as well as the model's learning ability, sometimes it is necessary

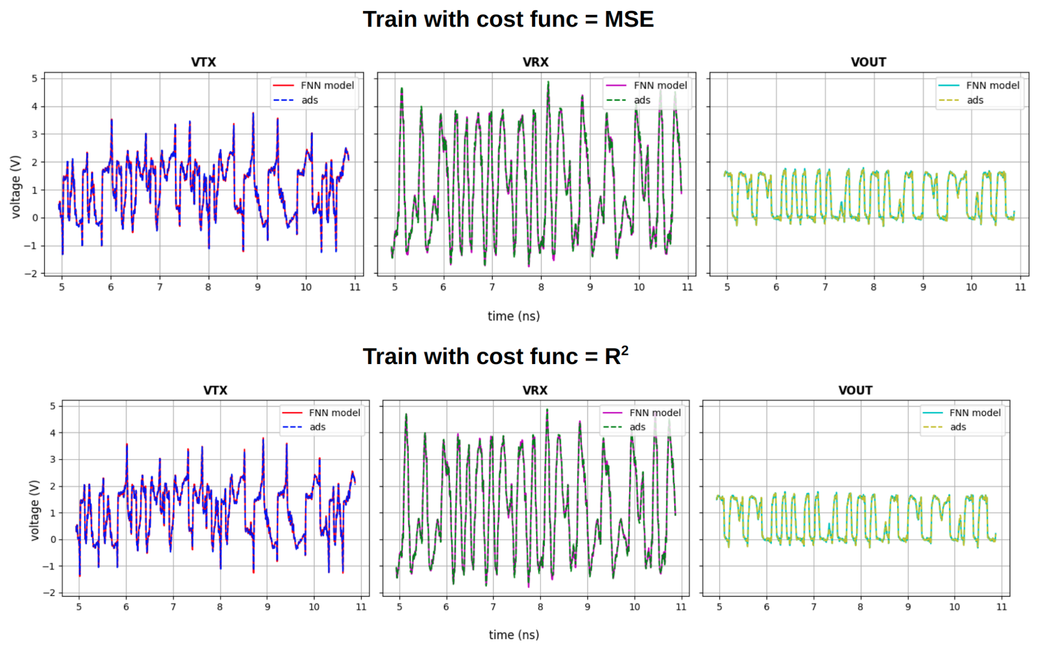


Figure 4.2: Comparison of predicted waveforms from MSE and R^2 trained models.

to reduce the dimensionally of the training data before a FNN structure is applied. A preliminary trail of model order reduction (MOR) is implemented and added into the existing script to relieve the heaving computing during kernel approximations. The MOR training example of TX CMOS is described below to illustrate this methodology.

After determining the shortest-possible memory length and protocols, the idea of Laguerre polynomials in [19] is re-applied to numerically reduce the column width of $\boldsymbol{X}$. More specifically, after MOR, each row in $\boldsymbol{X}$ does not hold any physical meanings but rather represents a Laguerre projection of the original input sequence. Depending on the order of Laguerre functions used, the number of neurons in the FNN can be greatly reduced and thereby leading to a lot more compact model while maintaining the same level of accuracy. In this example, where the FNN's input layer used to possess 250 (memory sequence) + 45 (S-parameter protocol) = 295 neurons, applying MOR of Laguerre order $R = 30$ could shrink the original FNN model by roughly 10 times. The implementation begins with the calculation of the orthonormal basis function ϕ_r as shown in Listing 3, of which a decaying factor of $\alpha = 0.83$ is chosen by brute force method. Then, the input matrix is convolved with

```python
# memory = memory sequence length
# var = number of protocol parameters
# R = order of Laguerre function
# alpha = decaying factor

def ncr(n, r):
    r = min(r, n-r)
    numer = reduce(op.mul, range(n, n-r, -1), 1)
    denom = reduce(op.mul, range(1, r+1), 1)
    return numer // denom
def calculate_phi(alpha, R, memory):
    elementM = np.arange(memory+1)
    elementR = np.arange(R+1)
    if memory < R:
        array_size = memory
        powera = np.power(alpha, R-elementM)
        power1minusa = np.power((1-alpha), elementM)
    else:
        array_size = R
        powera = np.power(alpha, elementR[::-1])
        power1minusa = np.power((1-alpha), elementR)
    combineMK = np.zeros(array_size+1)
    combineRK = np.zeros(array_size+1)
    for i in range(array_size+1):
        combineMK[i] = ncr(memory,i)
        combineRK[i] = ncr(R, i)
    ceff = np.ones(array_size+1)*(-1.0)
    ceff[::2] = 1.0
    phi = np.sqrt(alpha**(memory-R)) *np.sqrt(1-alpha) \
        *np.sum(ceff*combineMK*combineRK*powera*power1minusa)
    return phi

phi = np.zeros((R,memory+var),np.float32)
for i in range(R):
    for j in range(memory+var):
        phi[i,j] = calculate_phi(alpha, i, j)
```

Listing 3: Python implementation of ϕ_r based on Equation (2.2).

ϕ_r to perform dimensionally reduction. The rest of the algorithm shares the same training code as shown in Appendix B. With FNN model being much condensed, training speed is remarkably enhanced and a convergence is achieved with less 30 epochs. As for the cascading test, the MOR FNN TX demonstrates excellent correlation with the reference waveforms. The R^2 scores for V_TX, V_RX and V_{OUT} are 0.994, 0.996 and 0.995 respectively as shown in Figure. 4.3. In short, performing MOR is possible with FNN models as long as appropriate Laguerre order (R) and α are deployed.

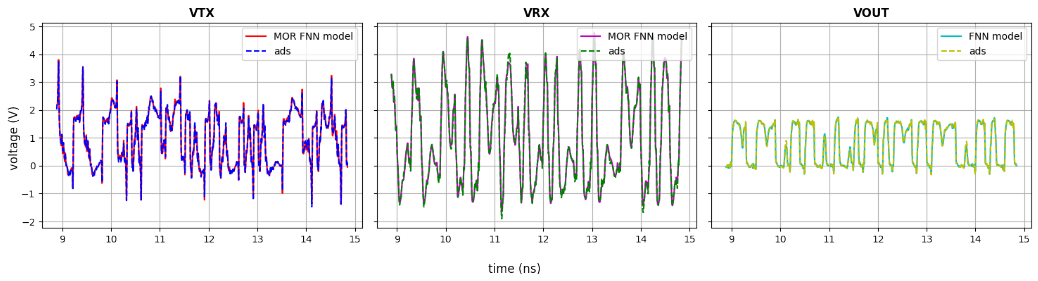


Figure 4.3: Comparison of predicted waveforms from MOR FNN and ADS.

4.2 Limitation

In the previous Chapter, we demonstrated that the FNN models could: 1) learn and mimic the nonlinearity of transistor level gates; 2) perform well in a HSL simulation. However, since modeling is a concept that can only resemble approximate forms of the natural phenomena, the FNN models are bound to always have limitations. In this section, we proceed to discuss the potential restrictions in the trained FNN models by stretching the tests to cover the corner cases. Not only can these examples guide the users to explore the full potential of the models, but they also serve as good baselines before the designers begin to transform a nonlinear device to a new FNN model.

4.2.1 Protocol Parameters

During the training phase, the protocols are swept and saved as a way for the FNN TX/RX models to recognize the HSL configuration. This places a constraint on the application of these models, more specifically on the channel and load they are allowed to cascade with while maintaining a high level of accuracy. There are two major concerns that need to be addressed:

1. Interpolation: The test protocol falls within the training range but not exactly at the sampled values.
2. Extrapolation: The test protocol is outside of the training range.

A validation schematic is setup using the HSL configuration shown in Figure. 3.7. The focus is placed on the NAND TX with geometric protocol. A new set of training is performed with an extended range of protocol as $W = [0.25, 6.25]$ mm with a step of 1 mm and $L = [0, 100]$ mm with a step of 20 mm. The expectations are that the TX model should be able to handle the interpolation test with $R^2 > 0.99$ and will eventually fail the extrapolation test when the channel's width or length is getting further away from the trained range. The only uncertainty remains is to quantify how *far* is considered to be the limit of the model.

The interpolation test is performed with a test protocol of $W = 3.75$ mm and $L = 90$ mm. As shown in Figure. 4.4, the FNN predictions match with the ADS reference well except some redundant jitters, which is due to the time sampling issue as we will discuss in the next subsection. Overall, the R^2 scores are 0.998 and 0.996 for V_{TX} and V_{RX}, meaning that as long as the test protocol is within the training range, the FNN models would be able to yield high accuracy result.

The extrapolation test is performed with a test protocol of $W = 1.25$ mm and $L = [100, 120]$ mm with a 1 mm step. The R^2 scores of V_{TX} and V_{RX} are calculated and plotted in Figure. 4.5. It can be observed that the accuracy quickly decays when extrapolating on the L protocol, especially on the V_{RX} where even a 5% extrapolation could lead to $R^2 = 0.8$. Since it is almost impossible for the FNN models to extrapolate an untrained protocol, we highly suggest that during the model training phase, widening the range of sweep as much as possible for the protocols.

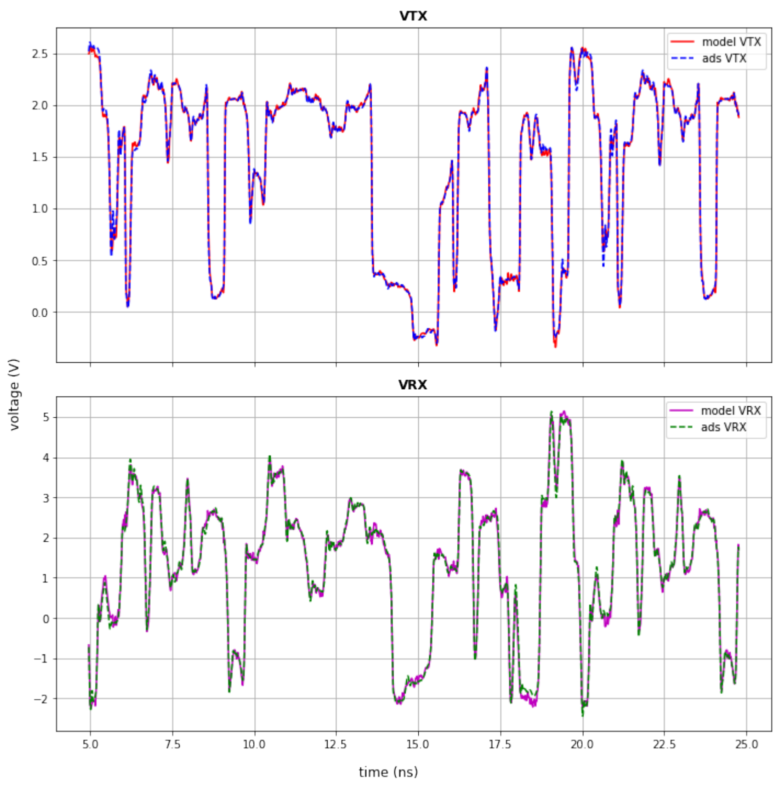


Figure 4.4: FNN prediction of V_{TX} and V_RX for the interpolation test on NAND TX.

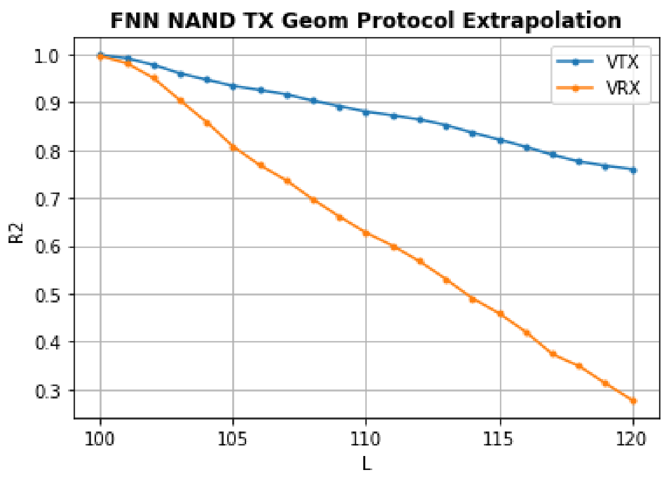


Figure 4.5: R^2 scores of FNN prediction of V_{TX} and V_RX for the extrapolation test on NAND TX.

4.2.2 Time Step Sampling

For various reasons, the users might want to simulate the FNN models at a different time step other than the one used in the training reference. To fulfill this request, a time sampling test is created with the CTLE schematic shown in Figure. 3.16 (a). In the original example, the training and testing time steps are both 2 ps. While leaving the CMOS TX model untouched, the CMOS RX model with CTLE feature is re-trained with reference voltages taken at times steps = $[1, 2, 3, 4, 5, 6]$ ps. The ADS reference V_{OUT} waveform and eye diagram of time step = 2 ps are plotted in Figure. 4.6. The newly trained RX models are cascaded with the TX model to produce the eye predictions as shown in Figure. 4.7. For each eye, the measurement details are listed in Table 4.1.

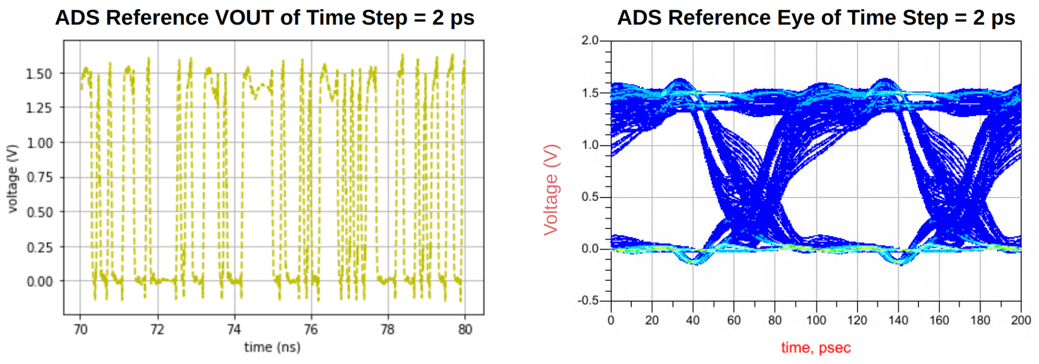


Figure 4.6: ADS reference V_{OUT} of time step sampled at 2 ps

As expected, if the training uses exactly the same time sampling as in the test, the FNN prediction yields the closest eye measurement compared to the reference. The problem occurs at coarse sampling, where the FNN is trained with a large time step and then test with a smaller one. In the case when the model is trained with 3 times the testing time step, obvious differences can be spotted at both the eye levels and the opening. A comparison of the reference and predicted V_{OUT} waveforms is given in Figure. 4.8. Starting from training time step = 4 ps (2 times the testing step), the FNN constructs the response with over- and under-shoots, which are indication of strong underestimations. This test gives a baseline for time sampling strategy when modeling with FNN: The training reference should set the time step as close as possible to the real use case, if not, finer sampling is always preferred.

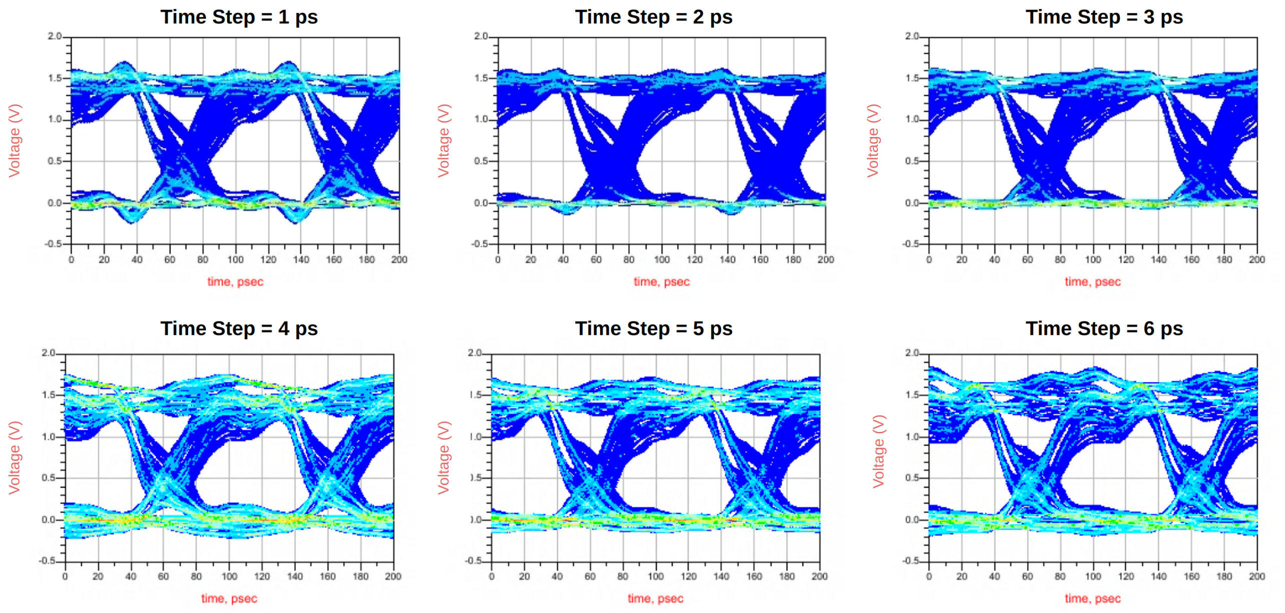


Figure 4.7: FNN predicted V_{OUT} eyes from different time step training.

Table 4.1: Eye diagram measurement of FNN CMOS RX model trained with time step (Tstep) = $[1, 2, 3, 4, 5, 6]$ ps

Measurement	ADS Reference (Tstep = 2 ps)	FNN V_{OUT} (Tstep = 1 ps)	FNN V_{OUT} (Tstep = 2 ps)	FNN V_{OUT} (Tstep = 3 ps)	FNN V_{OUT} (Tstep = 4 ps)	FNN V_{OUT} (Tstep = 5 ps)	FNN V_{OUT} (Tstep = 6 ps)
Eye Width	69.5 ps	68.5 ps	67.5 ps	68.5 ps	66.0 ps	67.5 ps	68.5 ps
Eye Height	1.02 V	0.88 V	1.01 V	1.00 V	0.76 V	0.81 V	0.76 V
Zero Level	5 mV	7 mV	4 mV	1 mV	-7 mV	-14 mV	-22 mV
One Level	1.41 V	1.41 V	1.41 V	1.42 V	1.42 V	1.42 V	1.43 V

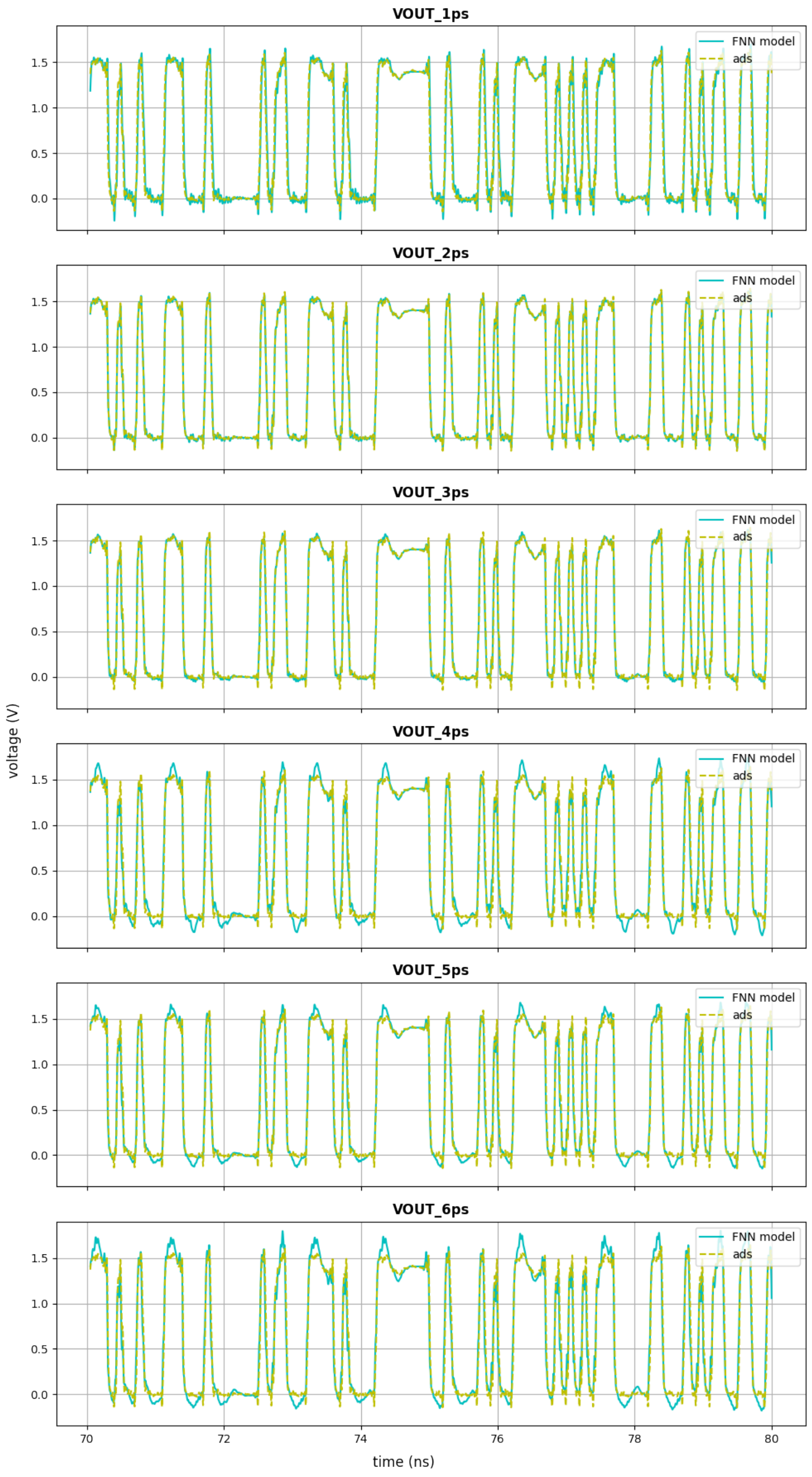


Figure 4.8: FNN predicted V_{OUT} waveforms from different time step training.

4.2.3 Memory Length

Defining proper memory lengths of the TX/RX models is critical because this controls the trade-off between the training cost and the modeling accuracy. With a longer memory length, more memory sequence columns are created in the input matrix and passed into the FNN to perform weight and bias fitting. This inevitably slows down the training and sometimes even leads to faulty predictions due to irrelevant inputs. On the other hand, if the memory length is too short, the trained model could be inadequate to explain the nonlinearities in the HSL. Since the ultimate goal is to perform accurate modeling with the least number of parameters, the methodology of choosing appropriate memory lengths for the FNN models is discussed below.

As a general rule of thumb, the memory length should be determined by the complexity of the buffers plus the protocols the buffers oversee. For instance, even for the same CMOS gate, the optimized memory lengths of the FNN TX/RX models are different due to fact that the protocols used in the TX training contain much more information than the ones in RX. The same concept applies when one wishes to add more features into the existing protocols. For example, if reactive components are accounted in the RX termination protocol, the memory length used in the previous examples ($M = 50$) might now be insufficient to represent the behavior of the RX. That being said, while it is possible to make intuitive guess on the required memory length based on the previously trained models, the better practice is to examine the training loss and R^2 test scores before a new model is put to use.

This work applies brute force method to search for the optimal memory length. To demonstrate the method's workflow, TX memory length for the differential CMOS driver in Figure. 3.21 is swept and the training/validation losses are recored as shown in Figure. 4.9. From the graphs, we can observe that the MSE loss drops below 10^{-3} when $M = 100$ and reaches its minimum value 10^{-5} when $M = 250$. Furthermore, when $M > 250$, the loss starts to increase slightly because redundant memory sequences are given.

The trained models are then cascaded with a test channel and a FNN RX model to examine how would the choice of memory length in training affect

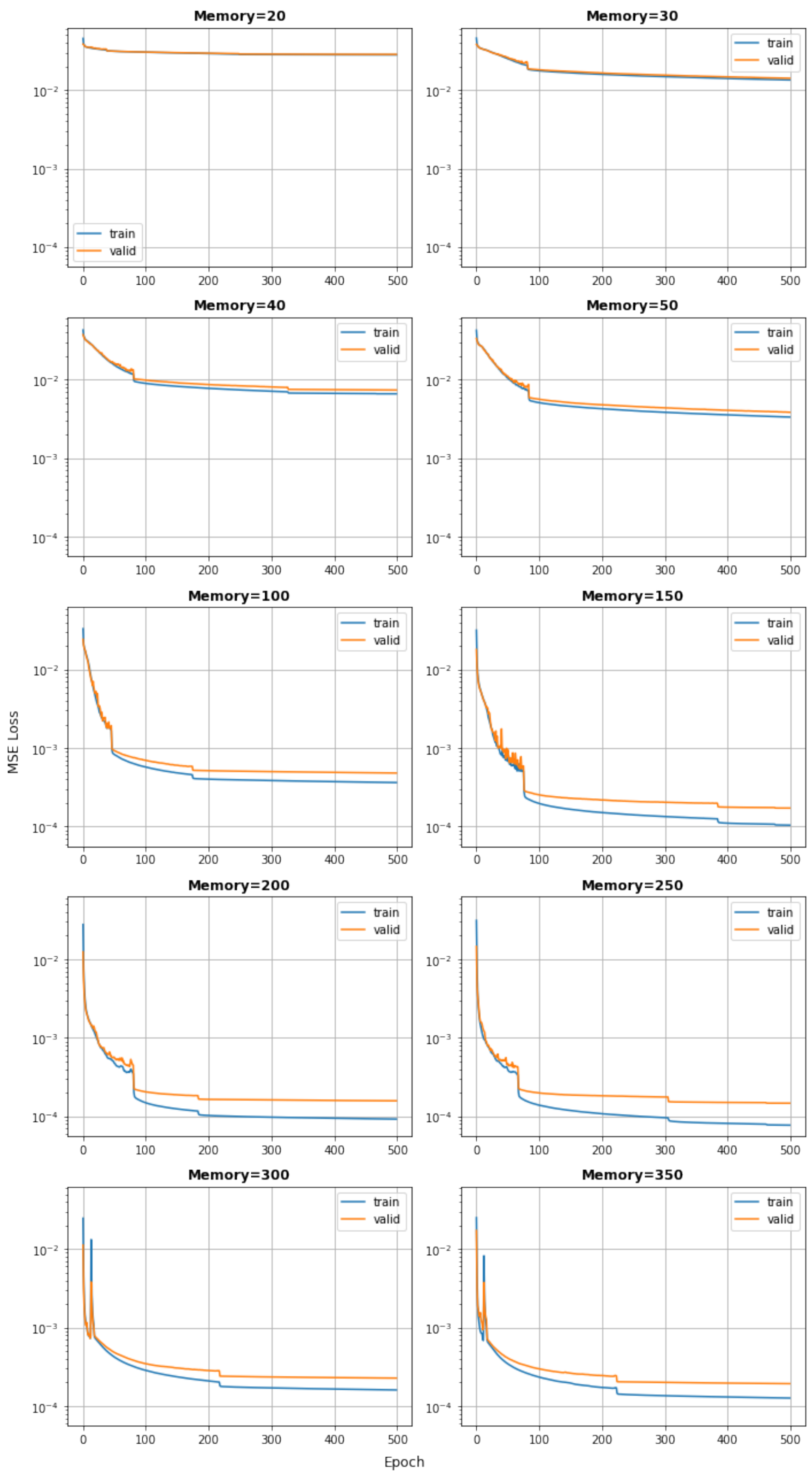


Figure 4.9: Training and validation loss of FNN TX model with different memory length.

the modeling accuracy. As shown in Figure. 4.10, the accuracy of the model improves dramatically when M goes from 10 to 100 and then maintains at $R^2 > 0.98$ in the range of M equals to 100 to 350. More specifically, the best V_{TX} and V_{RX} predictions ($R^2 > 0.995$) occur at $M = 250$, which is in accordance with the observations from the MSE loss plots. Overall, although this method seems tedious, it is very effective in ensuring that a least order FNN model is obtained.

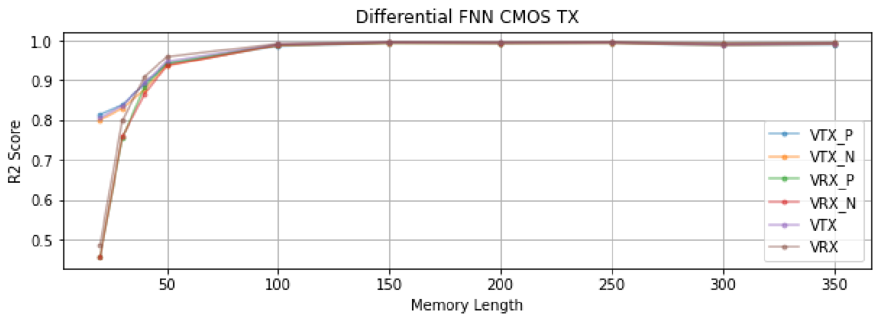


Figure 4.10: Relation between modeling accuracy and memory length.

4.2.4 Excitation

Throughout the schematics we showed in the previous Chapter, one might notices that the source excitation presented in both the training and the evaluation is always the V/T sequence of PRBS. While this particular choice of excitation makes sense in terms that almost all HSL simulations nowadays are preformed with PRBS, a general question emerges of whether the trained models are still accurate if a sinusoidal source is presented. To validate that, a stand-alone CMOS gate with an open termination is introduced as the testing schematic (see Figure. 3.6). After feeding the model with a 10 Gbps PRBS and a 5 GHz sinusoidal wave both of -5∼5 V amplitude, the voltage predictions after the CMOS gate are plotted in Figure. 4.11. Even though the result on the sinusoidal plot is not a complete failure because the FNN predicted waveform is still within the reference order of magnitude, it only yields a R^2 score of 0.613. Looking at the PRBS case where $R^2 = 0.999$ is achieved, a quick conclusion can be drawn that the existing PRBS trained model are not fully compatible with simulations using sinusoidal inputs.

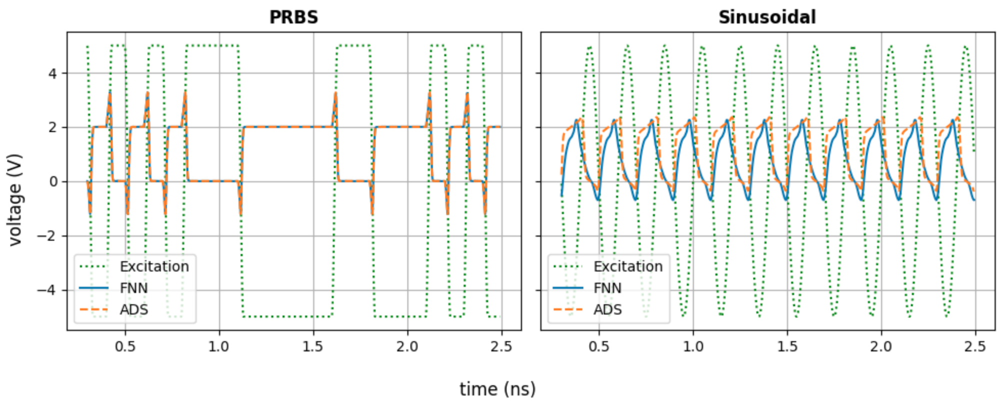


Figure 4.11: FNN prediction of the inverter output when trained with PRBS excitation.

For the sake of curiosity, a new CMOS model is trained based on the reference voltages extracted from the sinusoidal excitation. Reversely, the new model could now process the sinusoidal input in a much better manner than the PRBS one. Before jumping to the conclusion that one can only choose to persevere a single kind of excitation, we must first investigate the nature of this problem. The PRBS sequence is essentially square waves of different periods, which can be theoretically decomposed into infinite numbers of sine waves based on Fourier transformation. When the training is performed with PRBS excitation, voltages at each time stamp is assigned with a fixed multiplier (weight) as it goes through the FNN neurons. If the same scaling factor is to be placed on the sinusoidal waves as in the test case, it is guaranteed that the output will look like a distorted form of PRBS. Vice versa for the case of training the model with sinusoidal excitation. This observation provides a extremely valuable insight for where this project might go next: Is there a generic excitation, for instance a certain super-composition of sine waves, that could be trained in FNN such that the models are compatible with all kinds of voltages inputs? A preliminary plan could be first doing mini-batches of training, each performed on a sinusoidal input, and then place a Fourier wrapper at the end that draws the relation between the sinusoidal responses and the expected PRBS response. With all these possibilities, more details on how to further improve the FNN models are discuss in the next Chapter.

CHAPTER 5

SUMMARY AND FUTURE WORK

5.1 Summary

In this dissertation, machine learning methods are explored for generating the behavioral models of the nonlinear transceivers. More specifically, we purposed the methodology of developing cascade-able FNN models which are fully compatible with the modern HSL simulation. Compared to the prior works where the HSL is modeled as an entity, the FNN models are TX/RX building blocks that are both channel- and load-independent. By eliminating the dependency of the link, the FNN models are no longer bounded to a particular HSL and therefore not subject to regeneration if the link compositions changes. Through the research, there are a number of challenges associate with creating cascade-able models, namely:

1. Precisely characterize the nonlinearity of the transistor-level devices.
2. Separate the channel and load from the TX/RX modeling.
3. The TX/RX models are compatible with any configurations of HSL.
4. The TX/RX models are compatible with equalization settings.
5. The TX/RX models are compatible with PAM-4 differential signaling.
6. The TX/RX models are compact and easy to understand/obtain.
7. HSL simulation with TX/RX models yields accurate eye diagrams.

To tackle the first challenge, a nonlinear mapping between the input memory sequence and the output is established by constructing a FNN with three hidden layers. For all the neurons within the hidden layers, a nonlinear activation function is introduced and its weights and bias are adaptively adjusted during the training to minimize the difference between the predicted and the true output. It is preferred to have sufficient but not too long of a memory length, so the FNN could correctly draw the relation between the current output and the past states of the inputs in a timely manner.

For the second and third challenges, the purposed solution presented in this work is novel in terms that all the existing NN-related modeling failed to address them. To fulfill the quest of channel- and load-independent, we introduce the idea of *protocol*, which is a parameterized HSL information that is appended after the input memory sequence. By training with a certain range of protocols, the TX/RX FNN models are able to response correctly to various configurations of HSL and thereby claiming to be *cascade-able*. For the examples we demonstrated, good correlations were discovered between the predictions and the references regardless of channel or load conditions. In addition, the choice of protocols is extremely robust, meaning that the IC designers can invent their own protocols without deteriorating the performance of the FNN models. To proof that, we described the channel from both the geometric perspective and its frequency-domain representation, the VF parsed S-parameters. From the cascading-test, we conclude that either of the approaches is compatible with FNN modeling and more importantly, as long as reference waveforms are extracted, the protocols could theoretically be any useful information about a HSL.

The fourth and fifthschallenges, CTLE equalization and PAM-4 differential signaling, are resolved fairly straightforwardly given that the protocols are expandable by design. We chose to include these two features because as the data rate gets higher in the modern HSL designs, they are now essential components of a HSL simulation. The CTLE feature is implemented in the FNN RX model by appending the equalization settings as protocols: the CTLE pole, zero and gain. The PAM-4 differential signaling is implemented in both the FNN TX and RX models by expanding the memory sequences to record both the positive and negative terminals' responses. After verifying the accuracy of the cascaded outputs, we conclude that the expanded models are fully compatible with high data rate HSL simulations.

The sixth challenge is to make the FNN models more accessible for both the IC designers and the model users. From the designers perspective, generating FNN models is simple in terms that this process is fully automated as shown in Appendix A. After a raw FNN model is extracted, designer could choose to further reduce the size of the model by applying MOR for better IP protection. From the users perspective, applying FNN models in HSL simulation only requires minimum circuit design background. Since the model itself is a black-box, the users are not exposed to any complicate pin mapping that they used to see in the IBIS models. To run a HSL simulation with the FNN models, they only need to feed the excitation and channel information to the TX model and the load information to the RX model.

Last but not least, the FNN models are expected yield accurate voltage waveforms as well as the eye diagrams at each cascading nodes. From the tests we conducted in Chapter 4, we conclude that the eye diagrams given by the FNN models will be within 5% error range if the predicted waveforms maintain a R^2 score larger than 0.99. This score servers as an important metric for the designers when they are evaluating the performance of the FNN models. Any tuning of the FNN parameters should follow this guideline so that the trained models are in compliance with HSL eye diagram simulation. Overall, after the FNN models are properly trained, eye diagrams can be generated 25 time faster compared to a traditional SPICE simulation.

5.2 Future Work

Based on the limitations of the existing FNN models, the following works are proposed for the future work.

5.2.1 Generic Excitation Pattern

As indicated in the previous chapters, the FNN is nothing but a nonlinear mapping between the inputs and outputs. Therefore, the FNN models are expected to yield the best results when the training patterns are generic enough to cover a wide range of excitations. However, since it is practically infeasible to exhaust all kinds of excitations as training samples, we must shift our focus to how to properly decompose a random excitation to a finite number of signature excitations. For instance, if we decided that the signature excitations are sinusoidal waveforms, we could do a generic training of FNN using finite samples of sine waves each of different amplitude and periods. Next, after confirming the modeling accuracy, we could decompose any unknown excitations into the known sinusoidal inputs and then feed those to FNN to obtain partial response, which eventually will be summed together to restore the true outputs.

So far from all the experiments we conducted, we are almost certain that the signature excitations are indeed sinusoidal waveforms because theoretically speaking, they are capable of constructing any random excitation. Nevertheless, problems occur at the decomposition and restoring stages for the unknown excitation. Taking PBRS excitation as example, the harmonics required to construct PRBS with sine waves are almost infinite due to Gibb's effect. In order words, we could easily feed over 30 harmonics into the FNN but still not quite resembling the true PRBS inputs. Even if we disregard this difference at the input side, more severe issue emerges at the output side. Say that the partial responses we obtain are correct, re-combining them to a true PRBS response is not trivial at all. Due to the nonlinear nature of the transistors, poly-harmonics distortion (PHD) is expected at the output terminal. When PHD happens, the input and output of the harmonics are no longer guaranteed to have a one-to-one mapping, but rather a complicate mixing mapping composed of the odd- and even-modes. That being said, there are two potential remedies to untangle this mix-signal mapping:

1. Mathematically resolves the mapping by referencing X-parameter [43], which is proven to accurately characterize PHD but the concept itself is established in the frequency domain.

2. Adopts a machine-learning based structure to describe the mapping, which is straightforward to produce but may subject to other constrains such as training-sample insufficiency.

Either of the methods could potentially work if given more thoughts. Again, it is crucial for the ongoing work to tackle this problem because once the generic waveform is proven to succeed, the FNN model could completely replace IBIS or IBIS-AMI models for its superior computational efficiency.

5.2.2 Protocol and Feature Extensions

Up to this point, we have implemented many features in the protocols of FNN, but certainly this list is not even close to be comprehensive. Compared to the IBIS-AMI models, there are numbers of commonly used features that are not yet covered in the existing protocols. On the TX side, pre-emphasis and jitter settings can be added if the transistors to be modeled possess these capabilities. Furthermore, with respect to the channel protocols, it is also desirable to make the FNN models to be compatible with SPICE models for faster simulation speed. On the RX side, the assumption with load being a pure resistor must be lifted, as in most of the modern designs the load is oftentimes described by multiple SPICE or IBIS models. The post-cursor equalizations such as DFE can be added to RX protocols, too.

With all these new features, the FNN TX/RX models are then available for a much wider range of HSL applications. Luckily, as shown in Chapter 3, the expansion of the features are nothing challenging. It is just a matter of time to implement all these features and to claim FNN models as the better alternative of the IBIS-AMI models.

5.2.3 Compact FNN Models

To date, brute force method was used to determine the optimal memory length and the MOR parameters used for the Laguerre projections. Although this approach is simple and consistent, the time complexity relating to it oftentimes scales proportionally with respect to the guess range. Strictly speaking, this method requires the engineers to have a good sense of the optimized results even before the experiment is conducted.

Given the uncertainty, a more systemic approach is needed to achieve the most compact FNN models. One potential method could be Monte Carlo (MC) simulation [44], of which the optimizer takes random walk in a large solution space and evaluates the probability of a single solution set being the best result. The benefit of this method is the elimination of the guess range, which ensures that even an unexperienced person could run MC to obtain a reasonably compact FNN model. Another possible solution is to use Bayesian optimization [45], which is a global optimization strategy that is usually employed to observe I/O relation in a black-box manner. Since the objective function is unknown between the aforementioned parameters and the optimized-size FNN model, this problem is best resolved with Bayesian because if a SGD method is used, the derivatives evaluation could be extremely expensive. Overall, more research is needed to work out the best optimization method to obtain the most compact FNN models.

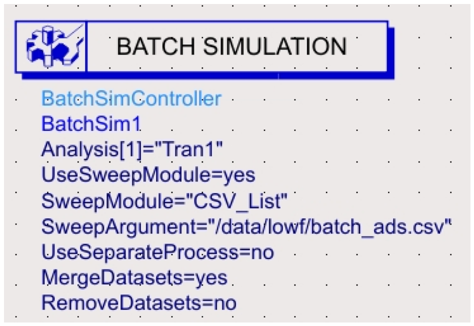
APPENDIX A

AUTOMATION SCRIPT OF DATA TRANSFERRING BETWEEN ADS AND PYTHON

Since massive data needs to be transfered between ADS and Python, automation scripts were developed in this project to ease the process. Generally speaking, ADS accepts command files that are in **.cvs* format with certain syntax. Rather than writing these files line by line, a python package *csv* is used to quickly looping through the required information and compiling it to ADS-compatible commands. On the other hand, data from ADS is extracted to FNN training script by utilizing the python package *numpy*, which grabs data from the **.cvs* file and organizes it to a matrix form.

A.1 Batch Simulation

After setting up the transient simulation schematic of the buffer in ADS, a batch list is needed to sweep through the protocol parameters and obtain ground truth data within the designed limitation of the model. The batch, or the sweeping simulation, is controlled by the component shown in Figure A.1, where the path to the batch list is detonated as *SweepArgument*. The first line in the batch list specifies the variable names and each line below that provides a set of swept data these variable should take.



```
BatchSimController
BatchSim1
Analysis[1]="Tran1"
UseSweepModule=yes
SweepModule="CSV_List"
SweepArgument="/data/lowf/batch_ads.csv"
UseSeparateProcess=no
MergeDatasets=yes
RemoveDatasets=no
```

Figure A.1: Batch simulation component in ADS.

```python
freq = np.arange(1, 5.1, 1)
amp = np.arange(3, 5.1, 1)
res = np.array([50, 2000, 100000])
w = np.array([0.25, 1.4, 2.55, 3.7, 4.85, 6])
l = np.linspace(0, 45, num=4)

total = len(freq)*len(amp)*len(res)*len(w)*len(l)
setting = np.zeros((total, 5))

with open(f'{file}.csv', mode='w') as f:
    data = csv.writer(f, delimiter=',')
    data.writerow(['FREQ','AMP', 'RES', 'MLINW', 'MLINL'])
    n = 0

    for f in freq:
        for i in amp:
            for j in res:
                for k in w:
                    for m in l:
                            data.writerow([f, i, j, k, m])
                            setting[n] = [f, i, j, k, m]
                            n+=1
```

Listing 4: Creating batch list with geometric protocol in python.

Geometric Protocol

The script that creates parameter sweeping with geometric protocol is shown in Listing 4. The variable names are separated by commas in the first line and their values are listed in the following lines. In this example, five variables (frequency, PRBS amplitude, termination resistance, MLIN width and MLIN length) are swept with a total batch number of 1080. Since the batches are fed to ADS sequentially, one needs to be extra careful with the size of the batch so the simulator won't crash due to RAM shortage.

Vector Fit Protocol

The script that creates parameter sweeping with vector fit protocol is shown in Listing 5. Besides the frequency and resistance sweeping, an additional variable that contains the path to the s-parameter files is included as SP. Different from the previous example of which the values are constants, the path is a string so it must be wrapped between the quotation marks to

```python
freqa = np.arange(1, 5.1, 1)
freqb = np.arange(1, 5.1, 1)
res = np.array([50, 2000, 100000])
sp = []

for n in range(10):
    sp.append('\"/data/sp/'+ str(n) +'.s2p\"')

n = 0
with open(f'{file}.csv', mode='w') as f:
    data = csv.writer(
        f,
        delimiter=',',
        quotechar="\'",
        quoting=csv.QUOTE_NONE
    )
    data.writerow(['FREQA', 'FREQB', 'RES', 'SP'])
    for f in freqa:
        for i in freqb:
            for r in res:
                for s in sp:
                    if i >=f:
                        data.writerow([f, i, r, s])
                        n+=1
```

Listing 5: Creating batch list with vector fit protocol in python.

match the syntax. Moreover, ADS does not allow duplicate variable names, including the built-in reserved ones such as $frequncy$. More details on the ADS input syntax requirement can be found in [46].

A.2 Reference Data Extraction

After running the batch simulation, voltages at the cascading nodes are collected by exporting the measurement as a **.cvs* file with time-dependency. The first column is the batch number, the second column is time and the following columns are voltage readings at each node. In the example shown in Listing 6, four nodes (V_{in}, V_{tx}, V_{rx} and V_{out}) were measured and the raw data has a total of six columns. The *conv* function is applied when scientific units like ps or V are presented in the **.cvs* file. The batches are divided based on the time stamp ($t = 0\ s$) rather than the batch number because there is a known glitch in ADS that generates non-integer batch value. To

```
def conv(fld):
    return float(fld[:-1]) if fld.endswith(b'V') else
    ↪  float(fld[:-3])

with open(file+'.csv') as f:
    data = np.genfromtxt(
        fname=f,
        delimiter=',',
        skip_header=31,
        converters={1:conv, 2:conv, 3:conv, 4:conv, 5:conv}
        # names = ['batch','time','vin', 'vtx', 'vrx', 'vout'],
        # max_rows=10
    )
data = data[:,1:]
parse = np.where(data[:,0] == 0)[0]
new_data = []

for i, num in enumerate(parse):
    if i != 0:
        tmp = data[parse[i-1]:num]
        new_data.append(tmp[6:])
new_data.append(data[num+6:])
```

Listing 6: Creating batch list with vector fit protocol in python.

keep uniformity in time step, the first six rows in the batch are discarded: These are V/T values ADS came up for DC simulation, where the voltages are sampled at tiny time steps. In the end, each batch is transformed to a *numpy* 2-D array of size equals to number of samples times number of measured nodes. The very last line in the example code collects all these arrays and organizes them into an ordered-list.

A.3 Eye Diagram

The eye diagram is a figure of merit for the SI engineer to quickly evaluate the HSL performance in the time domain. It is constructed by aligning and overlaying segments of bits from the long data stream acquired by a transient simulator. Since the reference data is in ADS and the FNN prediction is in python, one needs to re-locate at least one of them so the eye digram comparison can be done under the same environment. The subsections below provide three options for comparing the reference and the predicted eye.

```python
# create pads for time (x) and voltage (y) at VOUT node
pad_x = np.arange(0, data[:,0][memory_tx+memory_rx], 6.25e-12)
pad_y = np.zeros(len(pad_x))

# add padding to the predicted V/T
x = np.hstack([pad_x, data[:,0][memory_tx+memory_rx:]])
y = np.hstack([pad_y, VOUT.reshape(1,-1)[0]])

# linear interpolating V/T with reference time step = 6.25 ps
ref_x = np.arange(0, 100e-9, 6.25e-12)
ref_y = np.interp(ref_x, x, y)

# save the organized V/T in a text file for transfer
f = open(file +str(batch) +'.txt','w+')
f.write('BEGIN TIMEDATA\n# T ( SEC V R 0 )\n% time voltage\n')
for i in range(len(ref_x)):
    f.write(f'{ref_x[i]} {ref_y[i]}\n')
f.write('END')
f.close()
```

Listing 7: Creating batch list with vector fit protocol in python.

Construct Eye Diagram with ADS

This method transfers the predicted V/T waveforms back to ADS for eye diagram comparison. Given that the prediction normally does not start from $t = 0\ s$, a padding of zeros is inserted at the beginning of the predicted voltages to fill up the gap caused by the memory setting. Then the padded V/T data is linearly interpolated using python package *numpy.interp* to match the time step in the ADS reference. The organized V/T set is transfered to ADS as a text file with syntax shown in Listing 7. To load this file into ADS, select *Tools* → *Data File Tool* → *Read data file into dataset*. The file format to read is $MDIF$ and the sub type is $TIM\ MDIF$. This option allows user to perform transient simulation with customized time-domain waveform data using the component $VtDataset$. Finally, one can place eye probes at the nodes of the reference schematic and the user-defined source as shown in Figure A.2. The eye diagrams can be plotted side by side in the data display window. Although this method seems lengthy, the greatest advantage of transferring everything to ADS is that the eye height/width measurements are built-in function within the software. The quantified eye readings presented in Chapter 3 are obtained through this approach.

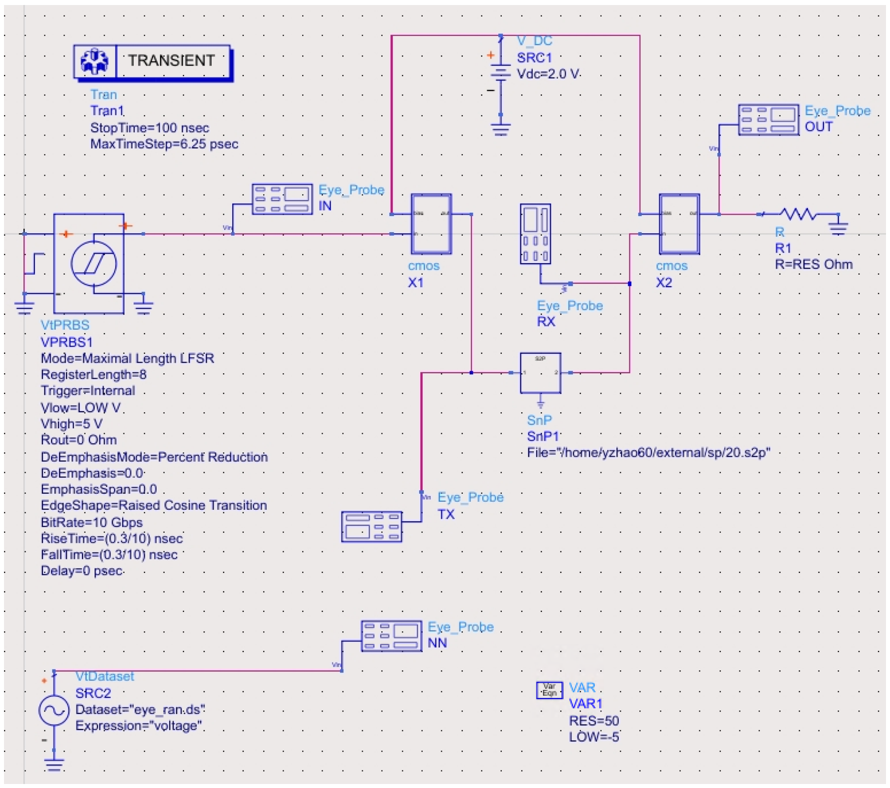


Figure A.2: Schematic for eye diagram comparison in ADS.

Construct Eye Diagram with Python

Since both the predicted and the true data are numpy arrays, it is natural to plot the eyes in python for a visual inspection. The biggest challenge for this method is to divided the long data stream into segments with proper size. In other words, the built-in eye diagram functions in ADS need to be correctly re-write as a script in python. Code in Listing 8 plots the eye diagram at the $VOUT$ node for a 100 ns transient simulation of 10 ps time step. The idea is to first linear interpolating the voltage wavform with respect to the sampling time and then separate the time sequence into bins of rising and falling edge. In this example, the source frequency is 1 GHz so the eye diagram spans for 2 ns or 200 time steps/bins. The eye diagram is constructed by repetitively overlapping the bins until all data is plotted. The opacity of the plotted lines can be tunned so one could observe the density of the eye.

```python
import matplotlib.pyplot as plt
import numpy as np

# linear interpolating the predicted voltage with time step 10 ps
time = data[:,0]*1e12     # change time unit to ps
x = np.arange((memory_tx+memory_rx)*10,100000,10)
y = np.interp(
    x,
    time[memory_tx+memory_rx:],
    VOUT.reshape(1,-1)[0]
    )

# separate rising and falling edges into mini-bins
bins = np.split(y, y.shape[0]/100)

# specify the time-constrain on the x-axis
xtick = np.linspace(0,2,num=200)

# plot configuration
ax = plt.axes()
ax.set_facecolor('k')
plt.xticks(np.arange(0, 2.1, 0.2))
plt.yticks(np.linspace(-0.5, 3.0, 7))
ax.set_ylim([-0.5, 3.0])
ax.set_xlim([0, 2])

# overlap the bins to fit the time-constrain
for i, data in enumerate(bins):
    plt.plot(xtick[:100], data, 'y-')
    plt.plot(xtick[100:], data, 'y-')
```

Listing 8: Generating eye diagram with python.

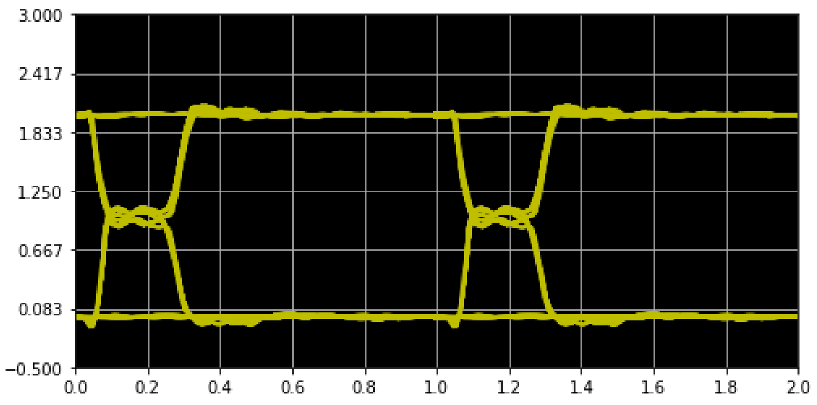


Figure A.3: Example of eye diagram plotted with python.

```
close all;
clc;

% read *.csv file as a nx1 array
file = 'eye.csv';
eye = csvread(file);

% specify number of samples per trace (half eye)
n = 100;

% plot eye diagram in new window
eyediagram(eye,2*n)

```

Listing 9: Generating eye diagram with Matlab.

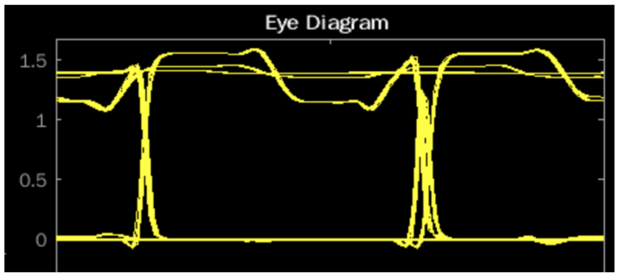


Figure A.4: Example of eye diagram plotted with Matlab.

Construct Eye Diagram with Matlab

Alternatively, one may choose to skip the writing of eye-diagram function by utilizing the pre-existing *eyediagram* package [47] in Matlab. In this case, two **.csv* files need to be prepared of which contains the reference and the predicted voltages. Both file are single-columned with their values sampling at the exact same time stamps. This means the predicted V/T has to be organized first by zero-padding the memory portion and then linearly interpolated to match with the reference. The plotting code in Matlab is included in Listing 9. Besides, if *Commnuication* or *SerDes* tool-boxes are purchased, one can simply import the file to the Eye Diagram component and configure the plot there. An example of eye diagram plotted using Matlab is shown in Figure A.4. Similar with the python method, there is no packages available for eye height/width measurement.

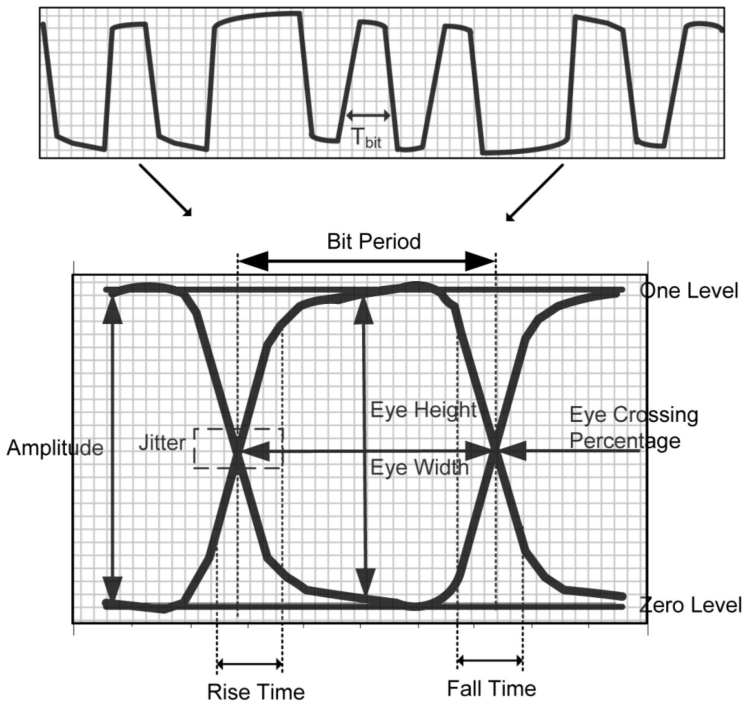


Figure A.5: Typical HSL eye diagram measurements.

Interpret Eye Diagram

As stated, the eye diagram is a common metric for quantifying the signal quality of a HSL. With a clock referecen as trigger point, the distorted PRBS bit stream is cropped and overlaid on the previous cycles. Generally, over hundreds of cycles are superimposed and the closing of the eye quantifies the bit error rate of the system. To help users to better understand this metric, some of the key terms are described below with respect to the labeling shown in Figure A.5.

One and Zero Levels. Theses are the high and low amplitudes of the captured voltages. They are calculated as the mean values taken from middle 20% of the histogram. By observing them, one can conclude if the system suffers from over- or under-shoot issue.

Eye Amplitude and Eye Height. While both of the concepts describe the voltage margin of the eye, the eye amplitude is the amplitude difference between the one and zero levels; the eye height is computed as the difference between the inner three standard deviation from the one and zero levels of the histogram. These terms are critical because they indicate the possibility of misinterpretation whether the data bit is 0 or 1 in digital domain.

Eye Crossing Percentage. This value marks the amplitude of switching bit relative to the one and zero levels. Ideally, a 50% eye crossing percentage is desired because the switching reference voltage V_{ref} is normally set as the mean between V_{high} and V_{low}. When it deviates from this value, the HSL system potentially exists high amplitude distortion and should be examined carefully for the root cause of pulse symmetry.

Bit Period and Eye Width. Both these terms are relating to the horizontal axis of the eye diagram, which is the time or unit-interval (UI) in unit of seconds. An eye diagram typically shows two UI, or $\frac{2}{BitRate}$. For example, a 10.0 Gbps data stream, the eye is plotted from 0 ps to 200 ps with $UI = 100$ ps. The eye width marks the time margin of the eye opening, which is calculated as the difference between the left and right crossing points with three inner standard deviations.

Rise and Fall Times. The rise and fall times are measures of the transition time on the upward and downward slope. By default, they are defined as the flight time between 20% and 80% values of the the maximum swing. Depending on the logic family, the fall time is sometimes slightly shorer than the rise time due to the nature of CMOS drivers, where the n transistor turns on faster than the p transistor [40].

Jitter. Jitter describes the time deviation from the ideal rise and fall timing events. A peak-to-peak (p-p) jitter is measured as the full width of the crossing edges. There are two main categories of jitter, the random jitter and deterministic jitter. The former one is Gaussian distributed and the later one is design specific. To minimize jitter, one should trace down the deterministic one and provide design guideline to eliminate ISI, crosstalk or duty-cycle distortion [48].

APPENDIX B

CUDA TOOLKIT INSTALLATION AND USAGE GUIDE FOR LINUX SYSTEM

To re-produce the NN structure presented in this thesis, follow the guidelines below for proper driver setup. Any skips in step will lead to configuration error. The sections below are organized based on build type: First, a CPU only setup is presented with an example code that parallelizes the training batches between multiple CPU cores; Second, an add-on GPU setup is introduced for users who wish to accelerate the process by utilizing GPU computational unites. Note that all software versions listed in this section are up to update as of Dec 2021.

B.1 CPU Only

Linux System Requirement

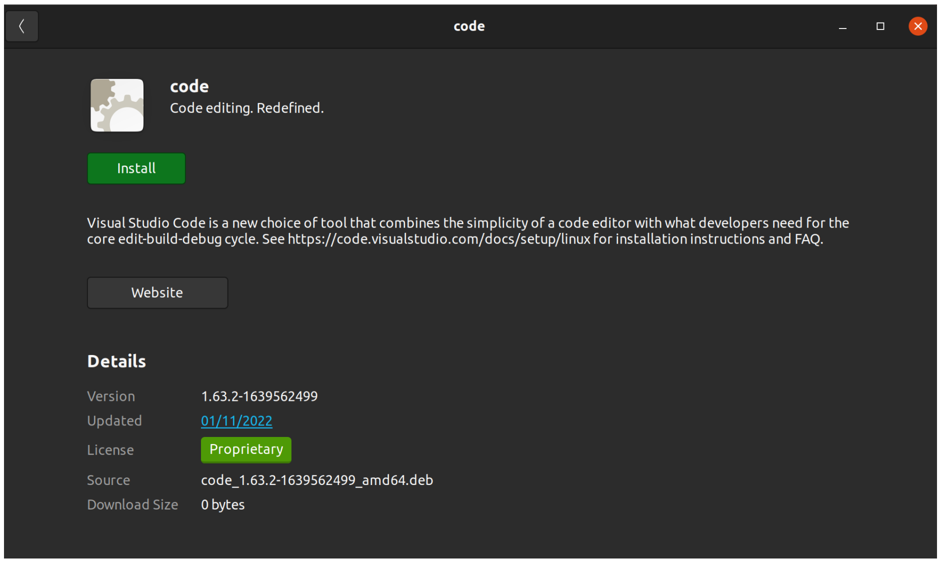
1. Download and install Ubuntu 20.04.3 LTS Focal Fossa from the official Ubuntu releases [49]. For desktop version, use the iso image named *ubuntu-20.04.3-desktop-amd64.iso*. Choose normal installation and include the third-party software.

2. Python3 should be built-in within this release. To verify which python3 version is installed, type in terminal (either bash or zsh):

   ```
   $ python3 --version
   ```

   If a recent current python3 is needed (e.g. this thesis used python 3.8.10), update the python3 package with the following commands:

   ```
   $ sudo apt-get update
   $ sudo apt-get install python3.8.10
   ```



Visual Studio Code is a new choice of tool that combines the simplicity of a code editor with what developers need for the core edit-build-debug cycle. See https://code.visualstudio.com/docs/setup/linux for installation instructions and FAQ.

Details

Version	1.63.2-1639562499
Updated	01/11/2022
License	Proprietary
Source	code_1.63.2-1639562499_amd64.deb
Download Size	0 bytes

Figure B.1: VS code installation

3. Install pip3 [50]. This is a package installer locally for python. Execute the following commands in terminal:

```
$ sudo apt-get update
$ sudo apt-get -y install python3-pip
$ pip3 --version
```

4. Download VS Code [51]. Use the *.deb* package since the linux distribution is Ubuntu. To install it, double click the file and install the package in *Software Install* window shown in Figure B.1. Same window can be found if you right click on the *.deb* file and select *Open with Software Install*.

5. VS code can be launched by clicking the windows key or click *show application* at left bottom corner of Ubuntu desktop GUI. In the search bar type *VS code*. You will be able to see the blue logo, click it and VS code will be launched. To avoid doing this every time, you can right click the logo in the task bar (on the right side of the desktop GUI after VS code is first launched) and select *add to favorite*. That way the shortcut stays in the task bar (see Figure B.2).

6. In VS code, it is recommended to install *Python* and *Pylance* extensions from marketplace shown in Figure B.3. That way the python3 code can

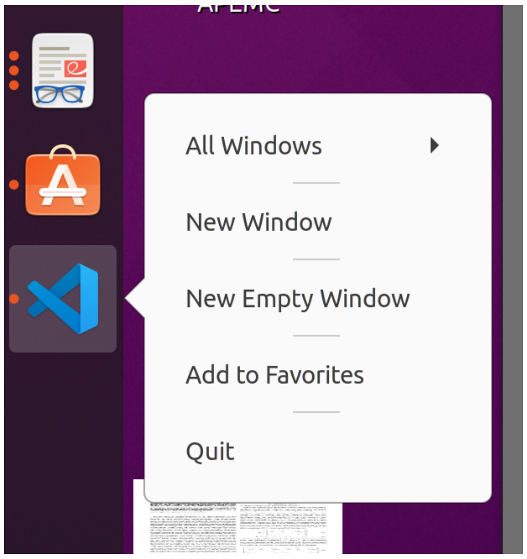


Figure B.2: Launch VS code

be automatically compiled and checked for syntax errors. You can also browse for more extensions using the search bar. You will be prompted to restart the software when some extensions are installed.

7. Select which interpreter to use in VS Code. To start, run these two commands in terminal to check the absolute path of python3 and pip3:

```
$ which python3
$ which pip3
```

   The default path is */usr/bin/**. If the packages are installed in any other folders, copy that path for the next step.

8. In VS code, click *view* → *Command Palette* → *Python: Select Interpreter*. Select *python 3.8.10 64-bit* with the path configured in the previous step. An example is shown in Figure B.4.

9. (Optional) Install python packages used in this thesis with:

```
$ pip3 install numpy
$ pip3 install pandas
$ pip3 install matplotlib
$ pip3 install sklearn
$ pip3 install skrf
```

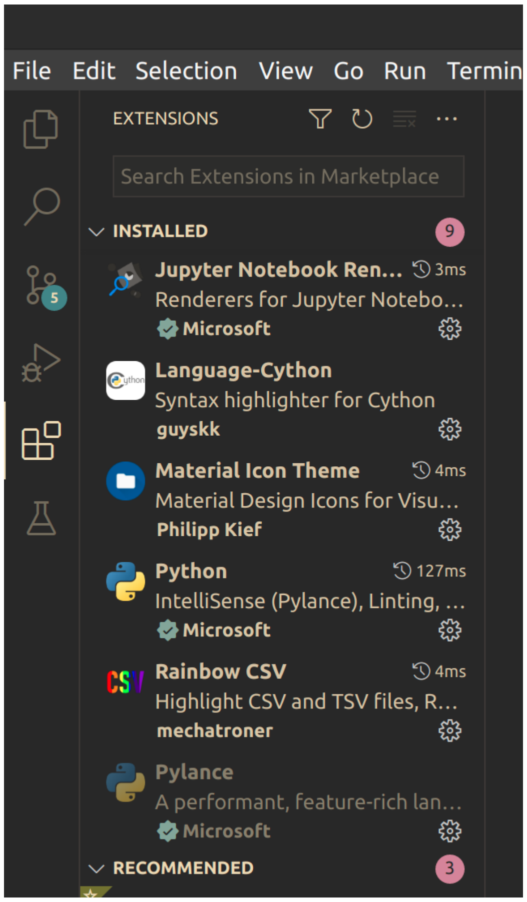

File Edit Selection View Go Run Termin
EXTENSIONS
Search Extensions in Marketplace
INSTALLED 9
Jupyter Notebook Ren... 3ms
Renderers for Jupyter Notebo...
Microsoft
Language-Cython
Syntax highlighter for Cython
guyskk
Material Icon Theme 4ms
Material Design Icons for Visu...
Philipp Kief
Python 127ms
IntelliSense (Pylance), Linting, ...
Microsoft
Rainbow CSV 4ms
Highlight CSV and TSV files, R...
mechatroner
Pylance
A performant, feature-rich lan...
Microsoft
RECOMMENDED 3

Figure B.3: VS code marketplace

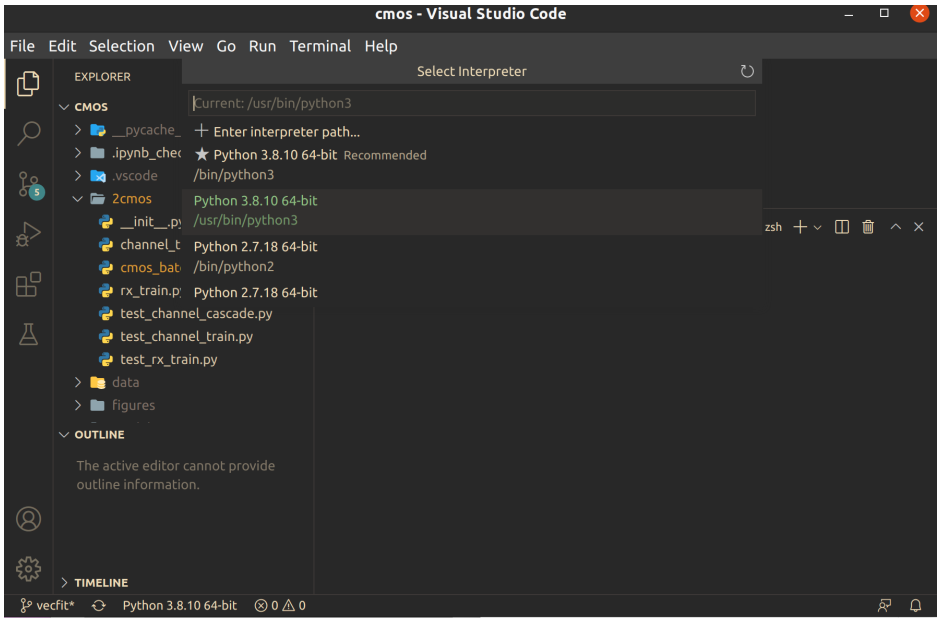


Figure B.4: Select Interpreter in VS code

Upon completion, use

```
$ pip3 list
```

to check if the items above have been successfully installed.

Pytorch

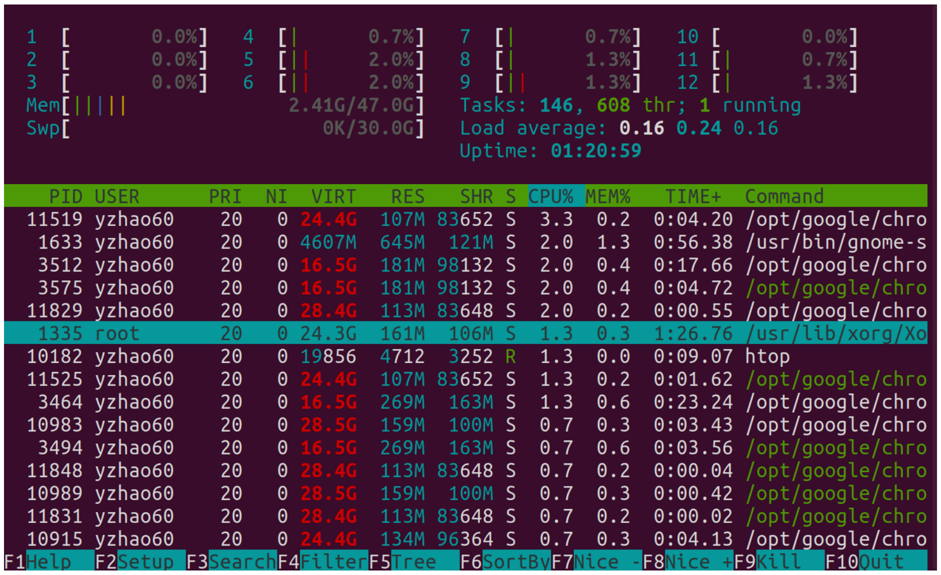
For CPU-only users, CUDA installation is not required nor recommended. Reason to ignore CUDA is straightforward: This tool is a computing platform that parallelizes GPU computation power with the CPU cores. Without the presence of GPU, it is meaningless to prepare the CPU for loading work. That being said, solo installing Pytorch does not lose means of parallelism, either. Pytorch by itself is capable of auto-distributing the forward and backward calculations evenly to every single thread CPU possesses. Install CPU version of Pytorch [52] by the following command:

```
$ pip3 install torch==1.10.1+cpu
torchvision==0.11.2+cpu torchaudio==0.10.1+cpu
-f https://download.pytorch.org/whl/cpu/torch_stable.html
```

Code Example of NN training with GPU

1. (Optional) Depends on the training data size, sometimes it might be necessary to increase swap size so the training process does not get killed due to RAM space shortage. Swap is a reserved hard disk space that can be used as RAM, in other speakings, a virtual RAM. It is relatively slower than the physical RAMs, but it stops the program from crashing when the code is dealing with more data than the system can handle. Size of swap space can be selected when creating the Ubuntu or can be adjusted later by the following terminal commands [53] (e.g 10 GB swap size):

   ```
   $ sudo swapoff -a
   $ sudo dd if=/dev/zero of=/swapfile bs=1G count=10
   $ sudo chmod 600 /swapfile
   ```



```
  1  [        0.0%]   4  [|       0.7%]   7  [|       0.7%]   10 [        0.0%]
  2  [        0.0%]   5  [||      2.0%]   8  [|       1.3%]   11 [|       0.7%]
  3  [        0.0%]   6  [||      2.0%]   9  [||      1.3%]   12 [|       1.3%]
  Mem[|||||             2.41G/47.0G]   Tasks: 146, 608 thr; 1 running
  Swp[                     0K/30.0G]   Load average: 0.16 0.24 0.16
                                       Uptime: 01:20:59

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  11519 yzhao60     20   0 24.4G  107M 83652 S  3.3  0.2  0:04.20 /opt/google/chro
   1633 yzhao60     20   0 4607M  645M  121M S  2.0  1.3  0:56.38 /usr/bin/gnome-s
   3512 yzhao60     20   0 16.5G  181M 98132 S  2.0  0.4  0:17.66 /opt/google/chro
   3575 yzhao60     20   0 16.5G  181M 98132 S  2.0  0.4  0:04.72 /opt/google/chro
  11829 yzhao60     20   0 28.4G  113M 83648 S  2.0  0.2  0:00.55 /opt/google/chro
   1335 root        20   0 24.3G  161M  106M S  1.3  0.3  1:26.76 /usr/lib/xorg/Xo
  10182 yzhao60     20   0 19856  4712  3252 R  1.3  0.0  0:09.07 htop
  11525 yzhao60     20   0 24.4G  107M 83652 S  1.3  0.2  0:01.62 /opt/google/chro
   3464 yzhao60     20   0 16.5G  269M  163M S  1.3  0.6  0:23.24 /opt/google/chro
  10983 yzhao60     20   0 28.5G  159M  100M S  0.7  0.3  0:03.43 /opt/google/chro
   3494 yzhao60     20   0 16.5G  269M  163M S  0.7  0.6  0:03.56 /opt/google/chro
  11848 yzhao60     20   0 28.4G  113M 83648 S  0.7  0.2  0:00.04 /opt/google/chro
  10989 yzhao60     20   0 28.5G  159M  100M S  0.7  0.3  0:00.42 /opt/google/chro
  11831 yzhao60     20   0 28.4G  113M 83648 S  0.7  0.2  0:00.02 /opt/google/chro
  10915 yzhao60     20   0 24.4G  134M 96364 S  0.7  0.3  0:04.13 /opt/google/chro
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure B.5: System processes interface

```
$ sudo mkswap /swapfile
$ sudo swapon /swapfile
$ /swapfile none swap sw 0 0
$ grep SwapTotal /proc/meminfo
```

2. (Optional) Use the following commands to monitor system processes (CPU) shown in Figure B.5, default refresh interval is 3 s:

```
$ sudo apt install htop
$ htop
```

3. Now we are ready to kick off the training in VS code. Create a new .py file by first importing pytorch libraries as shown Listing 10.

4. Initialize NN structure by defining layers as shown in Listing 11. This dummy NN framework has 3 layers (number of neurons): Input layer (input size), hidden layer (input size) and output layer (output size). Activation function used in between the layers is Rectified Linear Unit (ReLu) provided by Pytorch. Instructions of creating more complex NN can be found in [54], including zipped layers, self-defined activation function, etc.

```python
# for saving normalization vectors
import pickle
# for organizing datasets
import numpy as np
# for normalizing datasets
from sklearn.preprocessing import MinMaxScaler
# all functions in pytorch
import torch
# for building NN structure
import torch.nn as nn
# for building data structure
from torch.utils.data import DataLoader, TensorDataset
# for regulating training rate automatically
from torch.optim.lr_scheduler import ReduceLROnPlateau
```

Listing 10: Python libraries

```python
class NeuralNet(nn.Module):
    def __init__(self, input_size, output_size):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(input_size, input_size)
        self.fc2 = nn.Linear(input_size, input_size)
        self.fc3 = nn.Linear(input_size, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

Listing 11: Build NN structure

5. Load the framework of NN to the module with self-defined number of neurons. In this case we have a input layer & hidden layer of 100 neurons and output layer of 1 neuron. Then we define the loss function (*MSE*), optimizer algorithm (*Adam*) and training rate scheduler (*ReduceLROnPlateau*) as shown in Listing 12. More training optimization methods can be found in [55]. Choose the ones that best suit the training set.

6. Prepare the *numpy* dataset for training. Listing 13 shows a toy exam-

```python
input_size = 100
output_size = 1

model = NeuralNet(input_size, output_size)

loss_func = torch.nn.MSELoss()
optimizer =  torch.optim.Adam(model.parameters(), lr = 0.001)
scheduler = ReduceLROnPlateau(optimizer, 'min', patience = 5)
```

Listing 12: Load NN structure to module

ple of organizing time-domain waveform data to feature (input) and label (output) sets. The idea is assinging each label y_n with a feature sequence of memory $[x_{n-m}, x_{n+1-m}, \dots, x_n]$ of length equals to input size ($m = 100$). Here we used 80% of data for training / validation and the reminding 20% was left out for testing. Normalizing the datasets is not required, but could make a tremendous difference in time needed for training convergence.

7. Load the pre-processed datasets as tensors and organize the mini-batches by calling the *DataLoader* function (see Listing 14). Note that the dataset must be shuffled before passing into the loader to ensure accuracy and fairness. Otherwise, pass *shuffle=True* argument to the *DataLoader*. Not normalized sets could potentially extend the training time massively.

8. Write the training and validation loop (see Listing 15). This is a good time to open *htop* to monitor the system performance. Both *train_err* and *valid_err* should be stored during looping because they indicate: First, whether the initial learning rate is too steep; Second, whether the training is able to converge within the number of epochs. The print command allows one to constantly check if the training shows signs of non-convergence. At last, a log file of residue errors and the models are saved for testing.

9. Test the model with the unseen 20% of data. Prepare the dataset from original data to memory sequence arrays (*test_in* and *test_out*) in the same manner as we did for the training set. In the Listing 16 example below R^2 score is used to evaluate the accuracy of predicted output.

```
output_pickle = './rx_norm.p'       # norm vector file
tv_percent = 0.8        # train+validate / test ratio = 8 / 2
memory = input_size     # length of memory

# data is a numpy array from one batch in ADS simulation
# normally we include several batches to include more variation
# this example will only work for training on one particular ADS
↪  setting
# (details of parsing from ADS csv not shown here)

data_in = data[:,-2]    # un-organized input
data_out = data[:,-1]   # un-organized output
num = int((len(data_out)-memory)*tv_percent)

# create empty feature array
memory_in = np.zeros(
    (data_in.size-memory, memory+var),
    dtype=np.float32
)

# create feature and label arrays
for i in range(data_in.size-memory):
    memory_in[i,:] = data_in[i:i+memory]
    memory_out= data_out[memory:].reshape(-1,1)

# normalize datasets for faster training
input_scaler = MinMaxScaler()
input_scaler.fit(memory_in)
memory_in = input_scaler.transform(memory_in)
output_scaler = MinMaxScaler()
output_scaler.fit(memory_out)
memory_out= output_scaler.transform(memory_out)

# normalization vectors must be saved for testing
pickle.dump((input_scaler, output_scaler), open(output_pickle,
↪  'wb'))

# shuffle datasets for fairness concern
memory_mix = np.hstack((memory_in, memory_out))
np.random.shuffle(memory_mix)
```

Listing 13: Pre-process feature and label sets

B.2 GPU Add-On

The build in this section is based on a hardware set up of desktop PC with AMD 12-thread Ryzen 5, NVIDIA 3070 TI, 48 GB DDR4 RAM and 500 GB PCIe SSD (30 GB used for swap memory). Linux distribution is installed

```python
tensor_in = torch.from_numpy(memory_mix[:,:-1]).float()
tensor_out = torch.from_numpy(memory_mix[:,-1:]).float()

# separate training and validation sets
num = int(len(memory_mix))
valid_num = int(num/7)      # train / validate ratio = 7 / 1

BATCH_SIZE = 2048   # mini-batch size depends on RAM, must be
↪  power of 2

# load tensors to DataLoader of mini-batches
train_loader = DataLoader(
        dataset=TensorDataset(
            tensor_in[:-valid_num],
            tensor_out[:-valid_num]
        ),
        num_workers=12,     # must be equal to num of thread in
        ↪  CPU
        batch_size=BATCH_SIZE,
        drop_last = True    # drop the reminder set in mini batch
)
valid_loader = DataLoader(
    dataset=TensorDataset(
        tensor_in[-valid_num:],
        tensor_out[-valid_num:]
    ),
    num_workers=12,
    batch_size=BATCH_SIZE,
    drop_last = True
)
```

Listing 14: Load training dataset as mini-batches

```python
output_model = './rx.pt'    # model saving location
log = './rx_log.p'  # error saving location

EPOCH = 100     # num of loops for training
train_err = []
valid_err = []

for e in range(EPOCH):

    train_loss, valid_loss = 0.0, 0.0
    start_time = time.time()

    # set model to training mode
    model.train()
    for data, label in train_loader:
        optimizer.zero_grad()
        target = model(data)
        train_step_loss = loss_func(target, label)
        train_step_loss.backward()
        optimizer.step()
        train_loss += train_step_loss.item()

    # set model to Evaluation mode
    model.eval()
    for data, label in valid_loader:
        target = model(data)
        valid_step_loss = loss_func(target, label)
        valid_loss += valid_step_loss.item()

    curr_lr = optimizer.param_groups[0]['lr']
    scheduler.step(valid_loss/len(valid_loader))

    train_err.append(train_loss/len(train_loader))
    valid_err.append(valid_loss/len(valid_loader))

    print(f'Epoch {e}\t \
        Training Loss: {train_loss/len(train_loader)}\t \
        Validation Loss:{valid_loss/len(valid_loader)}\t \
        # LR:{curr_lr}')

pickle.dump((train_err, valid_err), open(log, 'wb'))
torch.save(model.state_dict(), output_model)


```

Listing 15: Training and validation loop

on a 500 GB PCIe MR2 internal SSD in parallel with another Windows 10 system.

```python
from sklearn import metrics

# load model and norm vectors
model.load_state_dict(torch.load(output_model))
input_scaler, output_scaler = pickle.load(open(output_pickle,
↪  'rb'))

test_in = input_scaler.transform(test_in)
pred_out =
↪  output_scaler.inverse_transform(model(test_in).data.numpy())
score = metrics.r2_score(test_out, pred_out)
```

Listing 16: Test the NN model

Graphic Driver

1. After the graphic card is inserted, switch the HDMI cable to the output of the card and turn on PC. Users may experience longer wait before the welcome screen shows up because Ubuntu needs to download a series of compatible drivers to initialize the new GPU.

2. Go to the menu by pressing the Windows key. In the search bar, type *drivers*. Click on *Software & Updates* in the results. If the icon is missing in the application list, run this command in terminal:

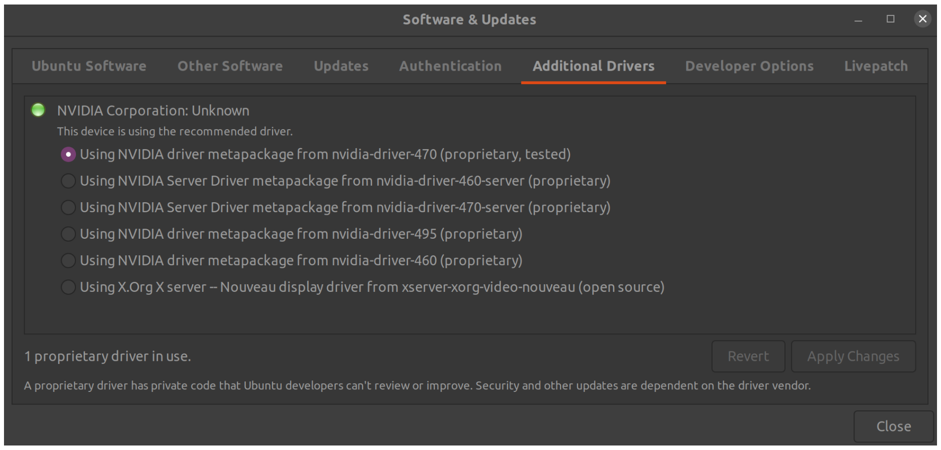
   ```
   $ sudo apt install gnome-control-center
   ```

3. Select the driver as shown in Figure B.6. This step is essential because CUDA is a higher level wrapper that utilizes the GPU driver. For more advanced GPU like NVIDIA 3090 TI, choose *nvidia-driver-495* instead.

NVIDIA CUDA and Pytorch

1. Perform clean uninstall of Pytorch (CPU version). Failed to do so would result in version conflict between Pytroch and CUDA. If pytroch was installed by pip3 as steps described above, use these commands to unload:

   ```
   $ pip3 uninstall torchaudio
   $ pip3 uninstall torchvision
   $ pip3 uninstall torch
   ```



NVIDIA Corporation: Unknown
This device is using the recommended driver.

- [✓] Using NVIDIA driver metapackage from nvidia-driver-470 (proprietary, tested)
- Using NVIDIA Server Driver metapackage from nvidia-driver-460-server (proprietary)
- Using NVIDIA Server Driver metapackage from nvidia-driver-470-server (proprietary)
- Using NVIDIA driver metapackage from nvidia-driver-495 (proprietary)
- Using NVIDIA driver metapackage from nvidia-driver-460 (proprietary)
- Using X.Org X server – Nouveau display driver from xserver-xorg-video-nouveau (open source)

1 proprietary driver in use.

A proprietary driver has private code that Ubuntu developers can't review or improve. Security and other updates are dependent on the driver vendor.

Figure B.6: Ubuntu additional driver setting (GPU)

2. Performed the CUDA pre-installation actions presented in [56]. If all steps above were executed correctly, the only error one might see is *GCC not found*. GCC is a compiler tool that comes with the meta-package named "build-essential". To fix the issue, type in terminal:

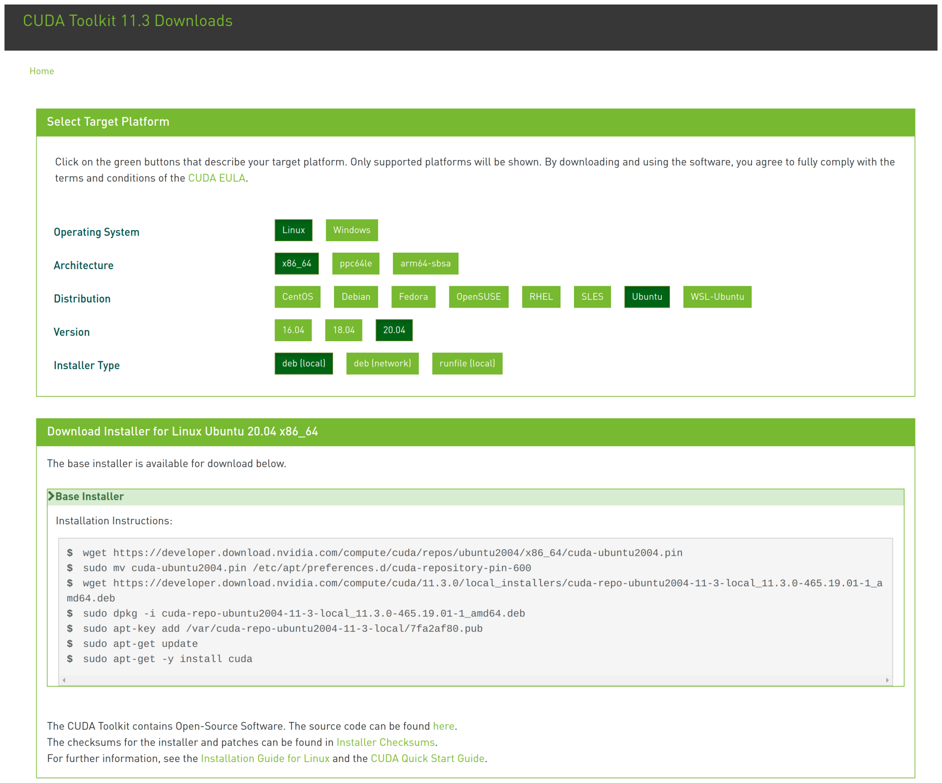
```
$ sudo apt update
$ sudo apt install build-essential
$ sudo apt-get install manpages-dev
```

3. Search online for *CUDA Toolkit V11.3.0* [57]. Any other versions of CUDA will be incompatible with Pytorch used in this thesis and thereby lead to system failure. The download page can be found at `https://developer.nvidia.com/cuda-11.3.0-download-archive`. Select option tabs as shown in Figure B.7. Installation instructions can be copied directly to terminal for execution. A dialog GUI will appear after the final line.

4. In the CUDA installation dialog, select *toolkit* ONLY. Skip the driver installation because Ubuntu 20.4 already took care of the problem.

5. At the end of CUDA installation, a prompt will appear in the command line which specifies the path of CUDA. Follow it by adding the following lines in either .bashrc or .zshrc (depends on which shell system use):

```
export PATH=/usr/local/cuda-11.3/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda-11.3/lib64
```


CUDA Toolkit 11.3 Downloads

Home

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown. By downloading and using the software, you agree to fully comply with the terms and conditions of the CUDA EULA.

Operating System	[✓ Linux] Windows
Architecture	[✓ x86_64] ppc64le arm64-sbsa
Distribution	CentOS Debian Fedora OpenSUSE RHEL SLES [✓ Ubuntu] WSL-Ubuntu
Version	16.04 18.04 [✓ 20.04]
Installer Type	[✓ deb (local)] deb (network) runfile (local)

Download Installer for Linux Ubuntu 20.04 x86_64

The base installer is available for download below.

Base Installer

Installation Instructions:

```
$  wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-ubuntu2004.pin
$  sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-repository-pin-600
$  wget https://developer.download.nvidia.com/compute/cuda/11.3.0/local_installers/cuda-repo-ubuntu2004-11-3-local_11.3.0-465.19.01-1_a
md64.deb
$  sudo dpkg -i cuda-repo-ubuntu2004-11-3-local_11.3.0-465.19.01-1_amd64.deb
$  sudo apt-key add /var/cuda-repo-ubuntu2004-11-3-local/7fa2af80.pub
$  sudo apt-get update
$  sudo apt-get -y install cuda
```

The CUDA Toolkit contains Open-Source Software. The source code can be found here.
The checksums for the installer and patches can be found in Installer Checksums.
For further information, see the Installation Guide for Linux and the CUDA Quick Start Guide.

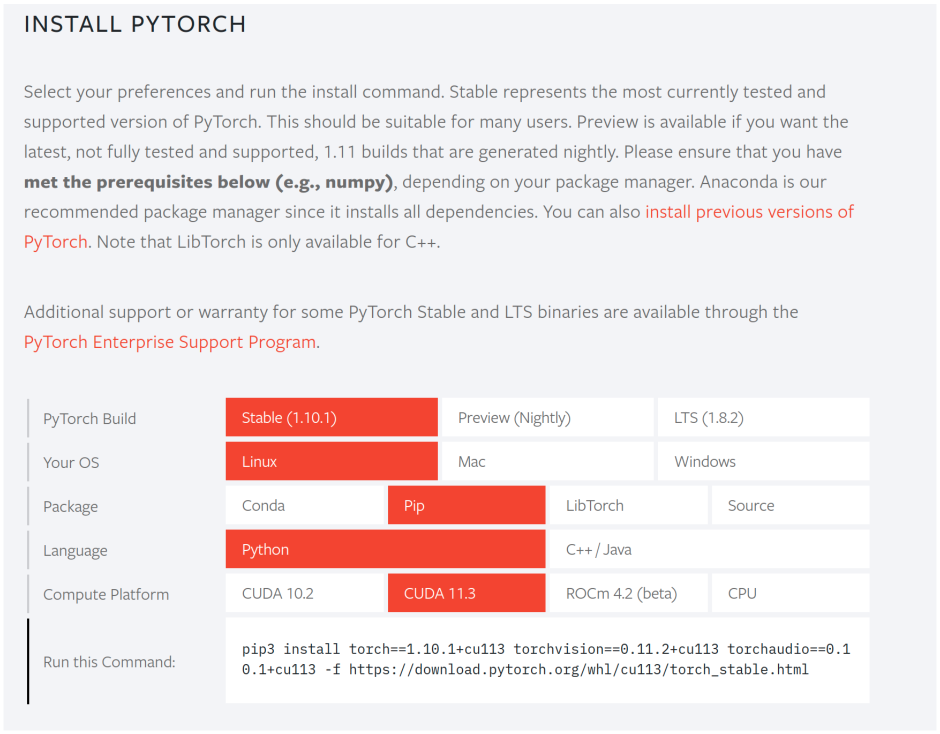
Figure B.7: CUDA toolkit v11.3.0

6. CUDA should be ready to use now. Check CUDA version by:

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Sun_Mar_21_19:15:46_PDT_2021
Cuda compilation tools, release 11.3, V11.3.58
Build cuda_11.3.r11.3/compiler.29745058_0
```

7. Finally, install Pytorch [52] with pip3. Configurations for this specific version of torch is shown in Figure B.8. In terminal, type:

```
$ pip3 install torch==1.10.1+cu113
torchvision==0.11.2+cu113 torchaudio==0.10.1+cu113
-f https://download.pytorch.org/whl/cu113/torch_stable.html
```



INSTALL PYTORCH

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, 1.11 builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

Additional support or warranty for some PyTorch Stable and LTS binaries are available through the PyTorch Enterprise Support Program.

PyTorch Build	[✓ Stable (1.10.1)]	Preview (Nightly)	LTS (1.8.2)	
Your OS	[✓ Linux]	Mac	Windows	
Package	Conda	[✓ Pip]	LibTorch	Source
Language	[✓ Python]	C++ / Java		
Compute Platform	CUDA 10.2	[✓ CUDA 11.3]	ROCm 4.2 (beta)	CPU

Run this Command:

```
pip3 install torch==1.10.1+cu113 torchvision==0.11.2+cu113 torchaudio==0.1
0.1+cu113 -f https://download.pytorch.org/whl/cu113/torch_stable.html
```

Figure B.8: Pytorch with CUDA 11.3 extension

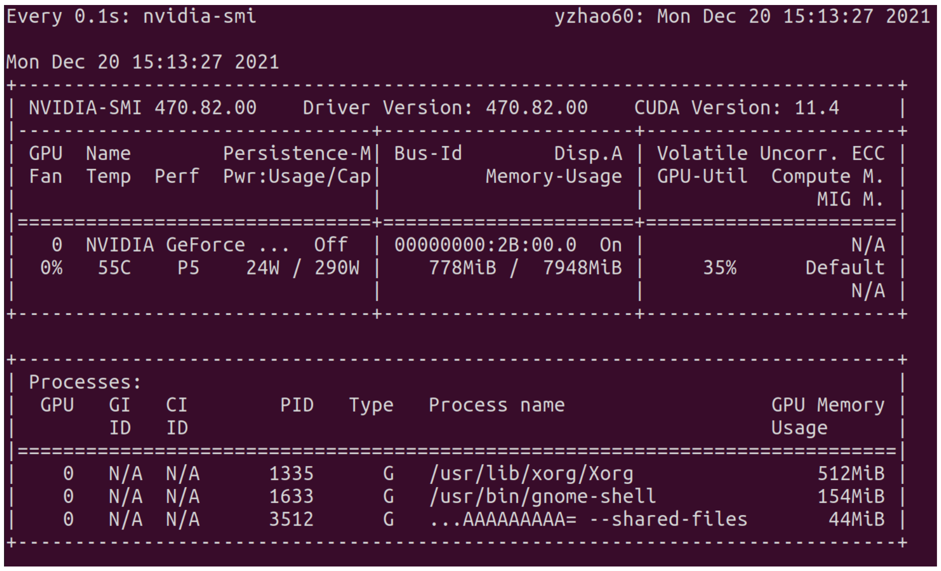
Code Example of NN training with CPU/GPU

1. (Optional) It is often-times handy to observe how GPU is utilized during the training. By reading the instantaneous system report, one could identify where the critical bottleneck happens during the training. For instance, a low GPU volatile GPU utilization combined with full CPU loading implies the NN feature resolution should be compressed before sending for training [58]. Use this command to check GPU status in time step of 0.1 s as shown in Figure B.9:

   ```
   $ watch -n 0.1 nvidia-smi
   ```

   For any amateur debuggers who wish to set up an initial step for debugging bottlenecks, *torch.utils.bottleneck* [59] is a good way to start. Although this package cannot pinpoint to the exact line that creates the problem, at least it shows some insights on the code block (e.g CPU loading) that might needs attention for further improvement.

2. Import python library and build NN structure as shown in Listing 10 and Listing 11. To load NN framework to GPU, use Listing 17 instead



```
Every 0.1s: nvidia-smi                                 yzhao60: Mon Dec 20 15:13:27 2021

Mon Dec 20 15:13:27 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.82.00    Driver Version: 470.82.00    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:2B:00.0  On |                  N/A |
|  0%   55C    P5    24W / 290W |    778MiB /  7948MiB |     35%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1335      G   /usr/lib/xorg/Xorg                512MiB |
|    0   N/A  N/A      1633      G   /usr/bin/gnome-shell              154MiB |
|    0   N/A  N/A      3512      G   ...AAAAAAAAA= --shared-files       44MiB |
+-----------------------------------------------------------------------------+
```

Figure B.9: NVIDIA system management interface

```python
use_cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if use_cuda else "cpu")
torch.backends.cudnn.benchmark = True

model = NeuralNet(input_size, output_size).to(device)

loss_func = torch.nn.MSELoss()
optimizer =  torch.optim.Adam(model.parameters(), lr = 0.001)
scheduler = ReduceLROnPlateau(optimizer, 'min', patience = 5)
```

Listing 17: Load NN module to device

of Listing 12. The extra lines call CUDA to prepare the devices. One may check the content of *device* to see if the code is running on GPU or CPU. Note the code assumes only one GPU is installed. If more GPUs are available, refer to code example in [60] to call them in parallel.

3. Pre-process the dataset as shown in Listing 13. Since now we use CPU for loading and GPU for computing, the *DataLoader* in Listing 14 needs to modified to pre-allocate memory space in GPU RAM (see Listing 18).

4. Kick off the training and validation loop (see Listing 19). Put *non_blocking* to *true* only when GPU is used. This is a good time to open *nvidia-smi* and *htop* to monitor the system performance. This saving command of

```python
# load numpy data as pytorch tensors
tensor_in = torch.from_numpy(memory_mix[:,:-1]).float()
tensor_out = torch.from_numpy(memory_mix[:,-1:]).float()

# separate training and validation sets
num = int(len(memory_mix))
valid_num = int(num/7)

BATCH_SIZE = 2048   # mini-batch size depends on GPU, must be
↪  power of 2

# load tensors to DataLoader of mini-batches
train_loader = DataLoader(
        dataset=TensorDataset(tensor_in[:-valid_num],
        ↪  tensor_out[:-valid_num]),
        num_workers=12,     # must be equal to num of thread in
        ↪  CPU
        pin_memory=True,    # pre-allocate GPU RAM space
        batch_size=BATCH_SIZE,
        drop_last = True
    )
valid_loader = DataLoader(
    dataset=TensorDataset(tensor_in[-valid_num:],
    ↪  tensor_out[-valid_num:]),
    num_workers=12,
    pin_memory=True,
    batch_size=BATCH_SIZE,
    drop_last = True
)
```

Listing 18: Pre-allocate GPU memory in *DataLoader*

errors comes at a cost of training time efficiency since GPU and CPU has to communicate at every single end of mini-batches. After a few tries of tuning, one can safely remove lines relating to external error so that maximum GPU utilization can be achieved. Testing block is the same as in Listing 16.

```python
output_model = './rx.pt'    # model saving location
log = './rx_log.p'  # error saving location

EPOCH = 100     # num of loops for training
train_err = []
valid_err = []

for e in range(EPOCH):

    train_loss, valid_loss = 0.0, 0.0
    start_time = time.time()

    # set model to training mode
    model.train()
    for data, label in train_loader:
        data = data.to(device, non_blocking=True)
        label = label.to(device, non_blocking=True)
        optimizer.zero_grad()
        target = model(data)
        train_step_loss = loss_func(target, label)
        train_step_loss.backward()
        optimizer.step()
        train_loss += train_step_loss.item()

    # set model to Evaluation mode
    model.eval()
    for data, label in valid_loader:
        data = data.to(device, non_blocking=True)
        label = label.to(device, non_blocking=True)
        target = model(data)
        valid_step_loss = loss_func(target, label)
        valid_loss += valid_step_loss.item()

    curr_lr = optimizer.param_groups[0]['lr']
    scheduler.step(valid_loss/len(valid_loader))

    train_err.append(train_loss/len(train_loader))
    valid_err.append(valid_loss/len(valid_loader))

pickle.dump((train_err, valid_err), open(log, 'wb'))
torch.save(model.state_dict(), output_model)

```

Listing 19: Training and validation with GPU

REFERENCES

[1] IBM, "How to squeeze billions of transistors onto a computer chip," 2020. [Online]. Available: https://www.ibm.com/thought-leadership/innovation-explanations/mukesh-khare-on-smaller-transistors-analytics

[2] J. Feng, B. Dhavale, J. Chandrasekhar, Y. Tretiakov, and D. Oh, "System level signal and power integrity analysis for 3200mbps ddr4 interface," in *2013 IEEE 63rd Electronic Components and Technology Conference*, 2013, pp. 1081–1086.

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

[4] Texas Instruments, "Application note 1111 an introduction to ibis (i/o buffer information specification) modeling," 2011. [Online]. Available: https://www.ti.com/lit/an/snla046/snla046.pdf

[5] B. Ross, "IBIS evolution ver.7.1," 2021. [Online]. Available: https://ibis.org/ver7.1/evol_ver7_1.pdf

[6] T. Zak, M. Ducrot, C. Xavier, and M. Drissi, "An experimental procedure to derive reliable IBIS models," in *Proceedings of 3rd Electronics Packaging Technology Conference (EPTC 2000) (Cat. No.00EX456)*, 2000, pp. 339–344.

[7] X. Hu, Z. Liu, X. Yu, Y. Zhao, W. Chen, B. Hu, X. Du, X. Li, M. Helaoui, W. Wang, and F. M. Ghannouchi, "Convolutional neural network for behavioral modeling and predistortion of wideband power amplifiers," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2021.

[8] S. Zhang, X. Hu, Z. Liu, L. Sun, K. Han, W. Wang, and F. M. Ghannouchi, "Deep neural network behavioral modeling based on transfer learning for broadband wireless power amplifier," *IEEE Microwave and Wireless Components Letters*, vol. 31, no. 7, pp. 917–920, 2021.

[9] J. Chu, W. Chen, L. Chen, and Z. Feng, "A cascaded memory polynomial-neural network behavior model for digital predistortion," in *2020 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*, 2020, pp. 1–3.

[10] 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)*, 2017, pp. 1–3.

[11] W.-T. Hsieh, C.-C. Shiue, and C.-N. 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 (ISCAS)*, 2005, pp. 3591–3594 Vol. 4.

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

[13] M. Haque, "TI IBIS file creation, validation, and distribution processes," 2022. [Online]. Available: https://www.ti.com/lit/an/szza034/szza034.pdf

[14] M. Mirmak, "IBIS modeling cookbook," 2005. [Online]. Available: https://ibis.org/cookbook/cookbook-v4.pdf

[15] Y. Wang and H. N. Tan, "The development of analog spice behavioral model based on ibis model," in *Proceedings Ninth Great Lakes Symposium on VLSI*, 1999, pp. 101–104.

[16] H. Lee and F. Rao, "Back to basics: IBIS/IBIS-AMI and the path to (LP)DDR5," 2021. [Online]. Available: https://www.signalintegrityjournal.com/articles/2020-back-to-basics-ibisibis-ami-and-the-path-to-lpddr5

[17] X. Chu, J. Li, X. Li, J. Wang, and Y. Li, "Modeling for nonlinear high-speed links based on deep learning method," in *2019 12th International Workshop on the Electromagnetic Compatibility of Integrated Circuits (EMC Compo)*, 2019, pp. 19–21.

[18] 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*, vol. 60, no. 6, pp. 1957–1964, 2018.

[19] X. Wang, T. Nguyen, and J. E. Schutt-Ainé, "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.

[20] S. Boyd and L. Chua, "Fading memory and the problem of approximating nonlinear operators with Volterra series," *IEEE Transactions on Circuits and Systems*, vol. 32, no. 11, pp. 1150–1161, 1985.

[21] R. J. G. B. Campello, W. C. Amaral, and G. Favier, "Optimal laguerre series expansion of discrete Volterra models," in *2001 European Control Conference (ECC)*, 2001, pp. 372–377.

[22] H. Ma, E.-P. 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.

[23] 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, 1999.

[24] A. Semlyen and B. Gustavsen, "Vector fitting by pole relocation for the state equation approximation of nonrational transfer matrices," *Circuits, Systems and Signal Processing*, vol. 19, pp. 549–566, 2000.

[25] B. Gustavsen, "Improving the pole relocating properties of vector fitting," *IEEE Transactions on Power Delivery*, vol. 21, no. 3, pp. 1587–1592, 2006.

[26] D. Saraswat, R. Achar, and M. Nakhla, "A fast algorithm and practical considerations for passive macromodeling of measured/simulated data," *IEEE Transactions on Advanced Packaging*, vol. 27, no. 1, pp. 57–70, 2004.

[27] Z. Zhang and N. Wong, "Passivity check of S-parameter descriptor systems via S-parameter generalized hamiltonian methods," *IEEE Transactions on Advanced Packaging*, vol. 33, no. 4, pp. 1034–1042, 2010.

[28] B. Gustavsen, "Computer code for rational approximation of frequency dependent admittance matrices," *IEEE Transactions on Power Delivery*, vol. 17, no. 4, pp. 1093–1098, 2002.

[29] S. Grivet-Talocia, "Passivity enforcement via perturbation of hamiltonian matrices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, pp. 1755–1769, 2004.

[30] B. Gustavsen and A. Semlyen, "Fast passivity assessment for S-parameter rational models via a half-size test matrix," *Microwave Theory and Techniques, IEEE Transactions on*, vol. 56, pp. 2701 – 2708, 01 2009.

[31] B. Gustavsen, "Fast passivity enforcement for S-parameter models by perturbation of residue matrix eigenvalues," *IEEE Transactions on Advanced Packaging*, vol. 33, no. 1, pp. 257–265, 2010.

[32] H.-Y. Shen, Z. Wang, and C.-Y. Gao, "Determining the number of BP neural network hidden layer units," *Journal of Tianjin University of Technology*, vol. 24, pp. 13–15, 01 2008.

[33] K. Sheela and S. N. Deepa, "Review on methods to fix number of hidden neurons in neural networks," *Mathematical Problems in Engineering*, vol. 2013, 01 2013.

[34] P. Gaurang, A. Ganatra, Y. Kosta, and D. Panchal, "Behaviour analysis of multilayer perceptronswith multiple hidden neurons and hidden layers," *International Journal of Computer Theory and Engineering*, vol. 3, pp. 332–337, 01 2011.

[35] I. Shafi, J. Ahmad, S. I. Shah, and F. M. Kashif, "Impact of varying neurons and hidden layers in neural network architecture for a time frequency application," in *2006 IEEE International Multitopic Conference*, 2006, pp. 188–193.

[36] K. Hara, D. Saito, and H. Shouno, "Analysis of function of rectified linear unit used in deep learning," in *2015 International Joint Conference on Neural Networks (IJCNN)*, 2015, pp. 1–8.

[37] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[38] Z. Zhang, "Improved Adam optimizer for deep neural networks," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 2018, pp. 1–2.

[39] G. Miller, "BSIM4 model (BSIM4 MOSFET model)," 2008. [Online]. Available: https://edadocs.software.keysight.com/pages/viewpage.action?pageId=5908883

[40] E. Bogatin, *Signal and Power Integrity - Simplified.* Pearson, 2018.

[41] P. K. Hanumolu, G.-y. Wei, and U.-k. Moon, "Equalizers for high-speed serial links," *International Journal of High Speed Electronics and Systems*, vol. 15, no. 02, pp. 429–458, 2005. [Online]. Available: https://doi.org/10.1142/S0129156405003259

[42] Optical Internetworking Forum, “Common Electrical I/O (CEI) - Electrical and Jitter Interoperability agreements for 6G+ bps, 11G+ bps, 25G+ bps I/O and 56G+ bps,” 2017. [Online]. Available: https://www.oiforum.com/wp-content/uploads/2019/01/OIF-CEI-04.0.pdf

[43] T. M. Comberiate, “Using x-parameters for signal integrity applications,” Ph.D. dissertation, Dept. of Electrical and Computer Eng., University of Illinois at Urbana-Champaign, 2014.

[44] S. Raychaudhuri, “Introduction to monte carlo simulation,” in *2008 Winter Simulation Conference*, 2008, pp. 91–100.

[45] V. Nguyen, “Bayesian optimization for accelerating hyper-parameter tuning,” in *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, 2019, pp. 302–305.

[46] G. Miller, “ADS simulator input syntax,” 2008. [Online]. Available: https://edadocs.software.keysight.com/display/ads2009/ADS+Simulator+Input+Syntax

[47] Matlab, “Eyediagram,” 2022. [Online]. Available: https://www.mathworks.com/help/comm/ref/eyediagram.html

[48] S. H. Hall and H. L. Heck, *Advanced Signal Integrity for High-Speed Digital Designs.* Wiley, 2009.

[49] “Ubuntu releases.” [Online]. Available: https://releases.ubuntu.com/20.04/

[50] “Pip documentation.” [Online]. Available: https://pip.pypa.io/en/stable/

[51] “Visual studio code.” [Online]. Available: https://code.visualstudio.com/

[52] “Install pytorch.” [Online]. Available: https://pytorch.org/

[53] “Change swap size in ubuntu 18.04 or newer.” [Online]. Available: https://bogdancornianu.com/change-swap-size-in-ubuntu/

[54] “Torch nn.” [Online]. Available: https://pytorch.org/docs/stable/nn.html

[55] “Torch optim.” [Online]. Available: https://pytorch.org/docs/stable/optim.html

[56] “NVIDIA CUDA installation guide for linux.” [Online]. Available: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

[57] “NVIDIA CUDA toolkit documentation.” [Online]. Available: https://docs.nvidia.com/cuda/archive/11.3.0/

[58] “Monitor and improve GPU usage for training and deep learning models.” [Online]. Available: https://towardsdatascience.com/measuring-actual-gpu-usage-for-deep-learning-training-e2bf3654bcfd

[59] “Torch utils bottleneck.” [Online]. Available: https://pytorch.org/docs/stable/bottleneck.html

[60] “CUDA semantics.” [Online]. Available: https://pytorch.org/docs/stable/notes/cuda.html