Deep Neural Networks for Behavioral Modeling of Analog ICs

André Carneiro Amaral

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Doctor Nuno Cavaco Gomes Horta

Doctor Nuno Calado Correia Lourenço

Examination Committee

Chairperson: Doctor Pedro Filipe Zeferino Aidos Tomás

Supervisor: Doctor Nuno Calado Correia Lourenço

Member of the Committee: Doctor Fábio Moreira de Passos

November 2022

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

This work was hosted at Instituto de Telecomunicações, funded by Fundação para a Ciência e Tecnologia– Ministério da Ciência, Tecnologia e Ensino Superior (FCT/MCTES) through national funds and, when applicable co-funded European Union (EU) funds under the project UIDB/50008/2020.

I would like to acknowledge my supervisor, Professor Nuno Lourenço, for all his support and guidance throughout the development of my Master Thesis, always bringing up innovative ideas. He was always available to analyse the results and criticize them in a constructive way which turned possible to achieve satisfying results in this work. I also would like to acknowledge Professor Ricardo Martins who was fundamental to keep me focus on the principal objectives and to give insights about what could be improved. I also want to acknowledge the Professor António Gusmão for his help in certain topics which was fundamental to have a proper insight of the problem.

I would also like to thank Professor Nuno Horta, firstly for the opportunity to integrate his group to do my Master Thesis, and secondly, for his ideas and constructive critics, which were a valuable contribution to this work.

Finally, a very special word of gratitude goes to my family who made the person I am today, empowering me with a proper education and values and always believing in me and in my capabilities. Their help was essential to be focused on my objectives and to reach my goals. My success I owe to them all.

Resumo

Este trabalho de tese de mestrado está inserido na área de Automatização de Projetos Eletrónicos e tem como principal objetivo proceder à modelação de circuitos analógicos ou dos seus respetivos componentes. Para atingir o objetivo proposto recorre-se às redes neuronais artificiais, com o intuito de criar um modelo que represente o comportamento do circuito e que seja rápido de simular. Uma vez que são gerados dados de entrada e saída através da simulação do circuito, o treino desta abordagem será supervisionado. Contribuições recentes neste campo de investigação têm demonstrado que as redes neuronais são capazes de imitar o comportamento de circuitos produzindo um erro residual, entre o comportamento simulado do circuito e a saída do modelo, acelerando as simulações. Neste trabalho é proposto o conceito de estrutura nos dados levando a uma aprendizagem da rede neuronal mais eficiente e mais generalizada para diferentes dimensões dos circuitos. Para tal são usadas redes neuronais recorrentes e Multi-Layer Perceptron (MLP) como linhas de atraso para efetuar a modelação comportamental de circuitos. Este método é aplicado a topologias de amplificadores operacionais e uma correta avaliação dos resultados obtidos é feita bem como a discussão dos mesmos. Os resultados foram obtidos usando um conjunto de dados construído através da simulação de um amplificador. Tais dados são usados para treinar um modelo “Long Shot Term Memory” (LSTM) e um modelo MLP com linhas de atraso, de modo que estes modelos simulem o comportamento do circuito para diferentes dimensioes. De modo a poder ser integrado em circuitos mais complexos, o MLP é convertido para a linguagem de hardware, o Verilog-A. Para esse propósito um ficheiro gerador de código Verilog-A é implementado em Python de modo a facilitar a conversão de redes neuronais de Python para Verilog-A. Este *script* é um avanço nesta área de investigação permitindo futuros desenvolvimentos. Relativamente aos modelos em Python a LSTM levou cerca de 60 minutos para treinar, já o MLP levou cerca de 30 minutos para treino. Em ambos os modelos um erro quadrático médio de 10^{-7} foi obtido. O MLP em Verilog-A permite uma aceleração de quase 5 vezes na simulação em transiente, permitindo também uma redução do tempo de otimização.

Palavras-chave

Automatização de Projetos Eletrónicos, Aprendizagem Supervisionada, Circuitos Analógicos, Modelação, Redes Neuronais Artificiais.

Abstract

This work is integrated into the Electronic Design Automation area, and its primary goal is to model the behavior of analog circuits. To achieve the proposed goal, Artificial Neural Networks are used to model the behavior of the analog circuit from input-output data obtained from SPICE simulation. In this work, two network structures are build, This work proposes to model the circuit behavior with a Recurrent Neural Networks model and a Multi-Layer Perceptron with a delay line model, using an operational amplifier. In addition, an automatic procedure to convert the created ANNs developed in Python using PyTorch, an industry standard and well-established state-of-the-art machine learning platform, to a Verilog-A module that can be used in SPICE level simulations was also implemented. This converter script is a step forward, enabling further developments in this field. Regarding the Python Models, the LSTM model took around 60 minutes to train, and the MLP model with two delay lines took around 30 minutes to train. For both models, a Mean Squared Error in the order of 10^{-7} was obtained. The MLP model with two delay lines was converted to Verilog-A, mimicking the Python model, proving the accuracy in the conversion between Python and Verilog-A. The first simulation in Verilog-A took 6 minutes since it had to compile libraries and modules. Once the models were compiled, subsequent simulations took only a fifth of the time of the equivalent Transistor level simulation.

Keywords

Analog Circuits, Artificial Neural Networks, Behavior Modeling, Electronic Design Automation, Supervised Learning.

Table of Contents

List of Figures

List of Tables

Acronyms

ADC	Analog to Digital Converter
AI	Artificial Intelligence
AMS	Analog-Mixed Signals
ANN	Artificial Neural Network
BGR	Band-Gap Reference
Bprop	Backpropagation
CompNN	Compositional Neural Network
CPW	Coplanar Waveguide
DAC	Digital-to-Analog Converter
DL	Deep Learning
EDA	Eletronic Design Automation
EM	Electromagnetic
FFNN	Feedforward Neural Network
GNN	Graph Neural Network
GPR	Gaussian Process Regression
HDL	Hardware Description Language
IC	Integrated Circuit
LSTM	Long Short-Term Memory
MIMO	Multiple-input Multiple-Output
ML	Machine Learning
MLP	Multilayer Perceptron
MS	Mixed Signal
MSE	Mean Squared Error
NARX	Nonlinear Auto-Regressive Neural Network with Exogenous Input
OTA	Operational Transconductance Amplifier
RBF	Radial Basis Function
ReLu	Rectified Linear Unit
RF	Radio Frequency
RMS	Root Mean Square
RNN	Recurrent Neural Network

Rprop	Resilient Backpropagation
SoC	System on Chip
TDNN	Time Delay Neural Network

Chapter 1

Introduction

This first chapter introduces analog Integrated Circuits (IC) design, emphasizing the challenges when modeling a circuit. Furthermore, the Machine Learning (ML) concept is introduced as well as the possible use of Artificial Intelligence (AI) to create accurate behavior models of analog circuits as an essential step to the production of Electronic Design Automation (EDA) tools.

1.1 Motivation: The Mixed-Signal Design Problem

Technology development sets the electronic industry under pressure due to the high demand for integrated systems. As we witnessed, even before the Covid-19 pandemic, many factories had to suspend production due to the lack of circuit components. From another perspective, IoT is joining millions of devices, sensors, and actuators together, enabling incredible breakthroughs in healthcare, education, agriculture, and many other areas. The IC continues evolving over the years and, despite not being an “eye catch” development, it is evident that the circuits are getting more complex, power-efficient, with reduced sizes and integrated, which is synonymous with improvements in this area. Most of the systems and sensors use a combination of analog/Radio Frequency (RF) and digital circuits in the same chip, commonly called “Mixed-Signal (MS) System-On-Chip (SoC).” In an MS SoC, the analog circuitry bridges the digital circuits and the physical devices.

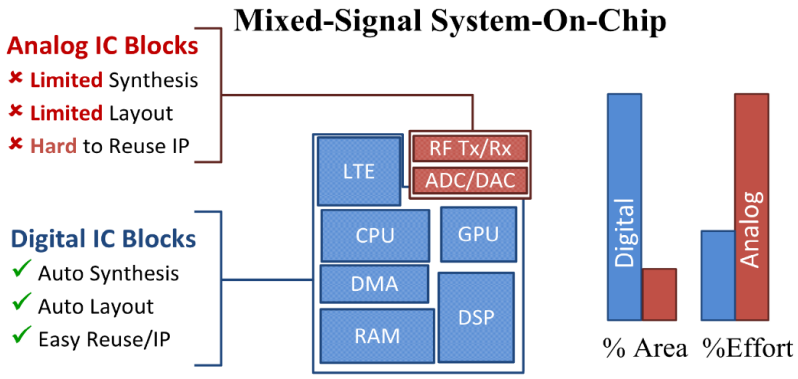

Figure 1.1 - Differences in terms of area and implementation effort between Digital and Analog from the designer perspective (from [1])

Analog circuit design is much less automated than its digital counterpart. However, the connection between the outside stimulus and the circuit is made through analog devices. Below some intrinsically analog functionalities that digital circuits cannot perform are referenced [1].

- **Sensing The System Inputs:** A signal from a transducer, microphone, or antenna needs to be sensed, amplified, and filtered to have an excellent signal-noise relationship and less distortion to digitalize the signal. Such analog front ends are pervasive in IoT, Healthcare/Wearable, and communication devices.

- **Conversion Between Analog and Digital Signals:** MS SoC circuits, such as Analog to Digital Converter (ADC), enable converting analog signals to digital signals and allow digital signal processing. In other words, they allow the signal to be processed and analyzed using digital circuity. After all the analysis and processing of the signal, a Digital-to-Analog Converter (DAC) can transform the digital signal into an analog one and have low distortion.

- **Regulate and Provide Power and Timing:** To properly use the ADC, they need a stable voltage and current reference. This can be achieved with voltage/current reference circuits and crystal oscillators.

Analyzing Figure 1.1, it is clear that, in an MS SoC, despite the higher area occupied by digital circuits, the effort in their design is reduced as the design flows are somewhat standardized and design automation is well established. On the other hand, designing analog circuits is still challenging and lacks automation support despite their lower area. In most cases, the circuit designer does analog circuits' design manually to achieve several contradictory specifications. This difference in the level of automation happens because analog circuits are generally less systematic and require exhaustive knowledge in their design.

The facts above underline the importance of developing tools to automate the design of analog circuits, reducing costs and speeding up the simulations. To sum up, it is essential to keep innovating in this area of research to reach a global automation tool that would allow the development of new technologies and deal with analog circuits more efficiently.

1.2 Overview: Analog and MS IC Design Flow

Before exploring the IC circuit design automation in more detail, it helps to have a brief overview of the IC design flow. To obtain an analog IC, a designer must follow steps in order to achieve the objective successfully. Some of these steps might differ depending on the company/designer since they can have a specific design flow. Gielen and Rutenbar's point of view, present in [2], shows a design flow strategy of a circuit with top-down design steps that are repeated from the system-level to the device level, with a bottom-up approach for verification and validation of the model. The approach is split into seven stages, as can be seen in Figure *1.2*:

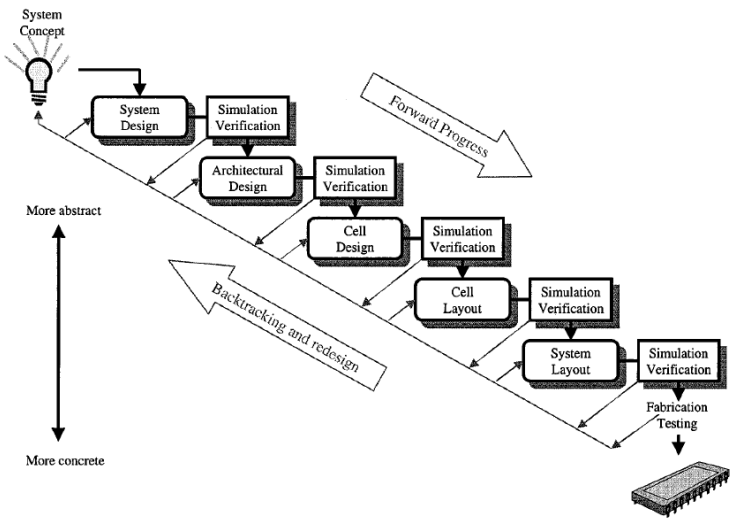

Figure 1.2: High-Level view of Analog IC Design Flow (from [2])

1) **Conceptual Design:** This is the stage in which the product is conceptualized, where specifications for a design are gathered and the overall product concept is developed. It might also be used for setting project management goals.
2) **System Design:** The system's overall architecture is designed and partitioned in this stage. It is also a stage to define which parts are hardware and software and the interfaces to be used.
3) **Architectural Design:** The decomposition of the hardware part in functional blocks is the primary concern. Those functional blocks, when connected, must fulfill the behavioral specifications. This stage includes the division between analog and digital blocks. Finally, the blocks are implemented in a hardware description language (HDL) (e.g., VHDL and VHDL-AMS). The high-level architecture is then tested through simulations to check if the specifications defined in the Conceptual Design stage are fulfilled.
4) **Cell Design:** For the analog blocks, this is the stage in which the detailed implementation of the different blocks for a given specification is made, leading to a wholly sized device/level circuit schematic. The obtained circuit design is validated against the specifications using the SPICE simulations.
5) **Cell Layout:** In this stage, a multilayer layout is obtained by translating the electrical schematic, of the different analog blocks, into a geometric representation. It involves area optimization, obtaining layouts that occupy a minimum amount of space in the chip, and parasitic reduction, which negatively impacts the circuit's performance.
6) **System Layout:** Here, the system-level layout is generated, including the system-level block place, route, and power-grid routing. It is also important to include descriptions of shielding or guarding that act as a measure to avoid crosstalk and substrate coupling. Further, the IC is verified, performing validation with the embedded software.
7) **Fabrication and Testing:** The stage where the ICs are fabricated. Simultaneously with fabrication, the devices are tested, and if any defect is detected, they are rejected.

Those previously mentioned steps are a simple representation of a complex process of designing analog IC. Backtracking and verification are required during the entire flow and are done frequently.

In [1][2], the authors provide the same stages but with a more detailed hierarchical analog design methodology. In Figure 1.3, the proposed steps are detailed.

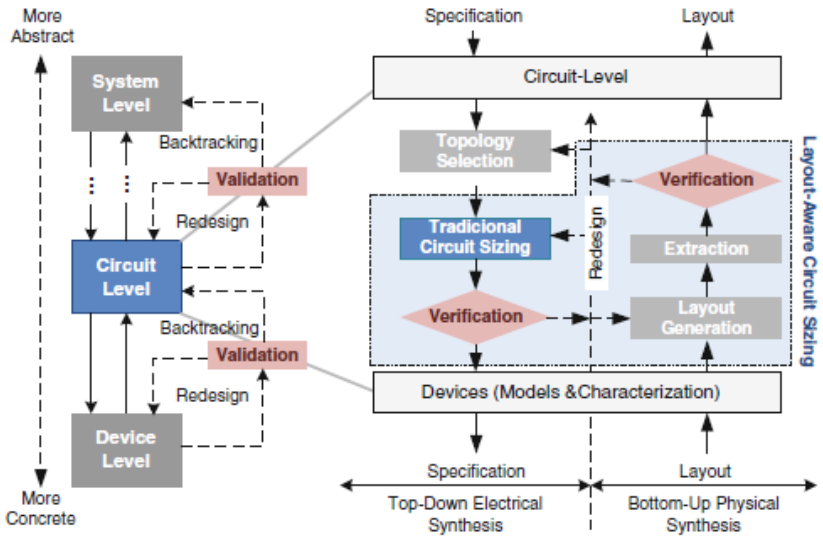

Figure 1.3 - Hierarchical level Design Detailed (*from [1][2]*)

In Figure 1.3, we can see a top-down and a bottom-up approach in the circuit level phase, which is needed to ensure circuit quality avoiding defects, before moving on to further phases. Adopting a top-down design methodology as proposed in [2] turns possible to explore complex system architectures leading to better system optimization at a higher abstraction level, which is essential since, after this step, is added more specific implementations at the lowest level. The number of hierarchical levels is dependent on how complex the system is. It has no pre-defined rule to follow and is based on the designers' experience.

The top-down approach is divided into the following processes [1]:

1) **Topology Selection:** section where the blocks and the correspondent connections are defined.
2) **Traditional Circuit Sizing:** each block gets its specifications translated from higher-level specifications. The block specifications can be, for example, the number of transistors or the gain of a circuit. This process is repeated until all the blocks have assigned their specifications

Once the top-down approach finishes, the bottom-up starts, which is composed of the following steps:

1) **Layout Generation:** the layout of each level is obtained from the structure of the lowest hierarchical levels.
2) **Layout Extraction:** the layout generated is now incorporated in a model, proper to be verified.

It is essential to remember that a verification phase where the simulations might reveal problems exists in both approaches. So, in that case, backtracking and redesigning the model happen when necessary. Analog behavioral models, the focus of this work, are indispensable parts of the analog IC design flow. They enable the simulation of large and complex electronic circuits and allow the exploration of design alternatives in a top-down approach, helping to identify the specifications for the sub-circuits in large designs. Moreover, they can be used in design automation tools to move to larger and more complex circuits. However, to reduce design iterations, analog models must account, as best as possible, for the non-ideal behaviors of the circuits.

1.3 Modeling with Machine Learning and Verilog-A

Machine Learning is a process in which a computer improves its capabilities after analyzing several past experiences. It is used for speech recognition, computer vision, natural language preprocessing, robot control, and many other areas. AI experts recognize that in some problems, it is more effective to use input-output examples to train a system rather than program it manually, anticipating the output for all given inputs [3].

Designing a circuit involves many tradeoffs, making it challenging to fulfill all the specifications required. Modeling a circuit in the architectural stage is an exhausting task that needs to capture the circuit behavior over all the constraints that need to be fulfilled. Typical analog IC design is time-consuming due to the complex relationship between the design parameters and the circuit specifications, whose number of parameters the designer must consider is generally high. In this context, novel uses of machine learning, particularly artificial neural networks (ANN) and deep learning (DL), could close the accuracy gap between Analog & Mixed-Signal (AMS) ICs' behavioral models and their corresponding spice-level (pre-and post-layout) models. Figure 1.4 represents the approach to consider when modeling the behavior of a circuit or device using ANN.

Once the ANN is accurate, it is necessary to convert it to an HDL in order to be integrated into the electrical simulation of complex circuits. To ensure compatibility with SPICE simulators, the HDL chosen is the Verilog-A, which uses mathematical descriptions to describe electrical or non-electrical behavior in terms of input and output ports for components, circuits, or even systems [5]. To make the model be more than a prove of concept framework, it needs to be converted to an HDL (e.g., Verilog-A), to be used for simulations allowing to integrate the model in complex circuits improving the simulation time. This serves as starting point of this thesis.

1.4 Work Objectives

The main goal of this work is to create, using ML methodologies, a behavior model of the device/circuit allowing to speed up simulation when used in complex circuits simulations with other sub-blocks. As it is debated in chapter 2, many studies are focused on one model to one circuit. While this work still targets a

single circuit topology for each model, it intends to create one parametric model capable of describing the behavior of multiple amplifier circuits, turning possible to generalize the model to upcoming designs.

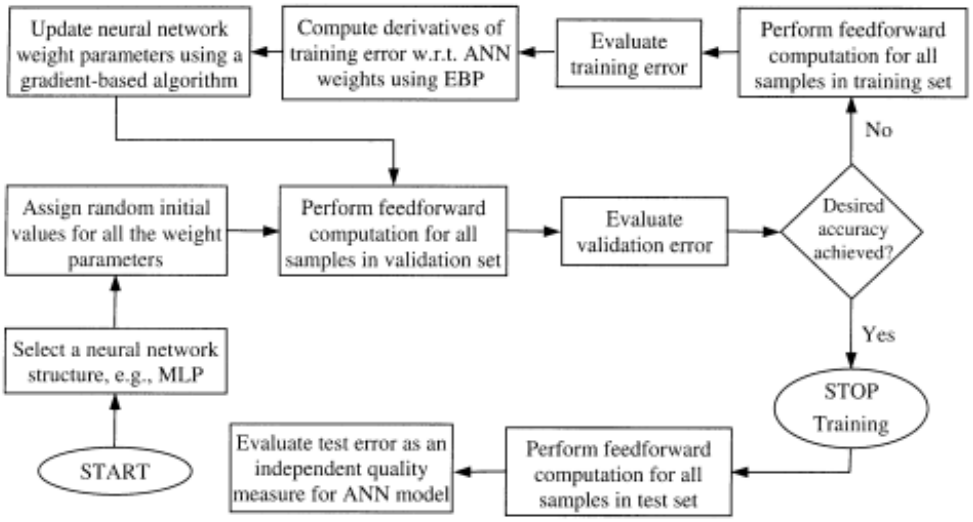

Figure 1.4 - Flowchart ANN Modeling Approach (from [4])

The steps to be taken in order to achieve the main objectives for this work are the following:

- Extract data from an amplifier circuit with different device sizes, creating a data set for each one circuit sizing, which contains the pairs input-output;
- Apply a LSTM and a MLP with delay lines. Those networks provide the circuit behavior in an output waveform in the time domain;
- Since the ANN removes the noise in data and a real circuit has noise, the error of the model is studied, and a proper noise is added to the predictions;
- The Multi-Layer Perceptron (MLP) with delay lines once it performs accurately in Python framework, it is converted to a hardware description language, Verilog-A;
- In the end, this model can be used in AIDA [7], a design automation solution developed at Instituto de Telecomunicações, to speed up the simulations of complex analog blocks composed of multiple amplifiers, such as ultra-low power analog front ends for biomedical applications [6].

1.5 Document Structure

This document structure is the following:

- Chapter 2 presents the state-of-the-art analog IC behavior modeling, presenting the numerous types and modeling techniques applied in different circuits. It also analyzes the state-of-the-art techniques to convert artificial neural networks to Verilog-A, framing the implementation which is going to be used in this work;

- Chapter 3 presents the implemented solution for capturing the behavior of an amplifier using only one DL for a circuit with different sizes. For that purpose, is used a LSTM and a MLP model. Firstly, the ANNs using PyTorch framework are presented, describing all the steps taken to obtain accurate models, for the LSTM and MLP with delay lines. Secondly a Verilog-A implementation of the MLP model is presented guided by a dummy example. This implementation is performed using an automatic conversion through a generator script build for the purpose of converting ANNs in Python to Verilog-A language;
- Chapter 4 presents the results obtained in the implementation phase, not only for the Python models (LSTM and MLP with delay line) but also for the Verilog-A MLP model and the automatic generator script;
- Chapter 5 presents the conclusions about the work, and it also explores the future work that can be done in this research field.

Chapter 2

State-Of-The-Art

In this chapter, the most relevant behavior modeling techniques for analog IC, using ML methods, are analyzed and discussed. A detailed review of the latest articles in this area will frame this work in the state-of-the-art. After that, an ML overview is presented to describe and compare the existing methods used to model the behavior of circuits/devices. Finally, is presented an overview of Verilog-A for system verifications through simulations.

2.1 Modeling of Analog/RF Circuits with Machine Learning

The design of analog IC is time-consuming due to the non-linearity relationship between the design parameters and circuit/device specifications. Generally, a handmade calculation can restrain the design space and provide a good starting point to the designer. However, the process is still cumbersome due to many iterations to achieve the expected specifications. Recently ANNs have become a solid alternative to model analog circuits behavior. In the literature, studies attempt to model circuits behavior in many different fields of analog IC, such as oscillators and operational transconductance amplifier.

In [8], the behavior of a switched-capacitor three-stage cross-coupled charge pump circuit is modeled. First, the data is obtained by circuit simulation, and the input (current between nodes, clock time, and voltage) and the correspondent outputs are preprocessed, leading to a balanced dataset. A Multilayer Perceptron (MLP) [9] is built to find the charge amount and implement a specific charge transfer function. It has 3 hidden layers (the first one with 50 neurons, the second with 20 neurons, and the last with 10 neurons), with a hyperbolic tangent as an activation function and Adam [10] as the optimizer. The resulting model deviates a maximum of 9.6% from the circuit behavior and a reduction of 7 seconds on the simulation time. After that, the model was implemented in VerilogAMS. According to the authors, training the model with the border values of the dataset leads to inconsistent results. They suggested a creation of a safety margin in the dataset, preventing the model from being trained with that border values.

Grabmann, Landrock, and Gläser in 2019 proposed to create a model to analyze the power consumption of a circuit transient [11] since manually written behavior models cannot capture the power consumption at the transistor level. As a case study, it uses a low relaxation oscillator. The inputs (frequency, time clock, and EN) are obtained from circuit simulation in SPICE with a sample period of 10ns. The data is scaled, and only the equidistant data will be kept. After that, they propose a Time-Delay Neural Network (TDNN) with 2 hidden layers, the first with 50 neurons and the second with 10 neurons. The network also contains 10 steps of delay. The hyperbolic tangent is the activation function for the hidden layers, and the output layer is the linear one. The optimization is done using the Gradient Descent algorithm. The paper results

report that the power consumption curve differed with an error of 2.7% when using the model or the circuit in the simulation. The simulation time is reduced by 95%.

In the same line, Suissa in [12] proposes a neural network method to model the power consumption of analog components. The methodology is divided into 3 steps:

- **Measurement Step:** Sweep the input parameters of the circuit over the entire operating range and obtain, as an output, the power consumption for each case;

- **Model Creation:** Using a Feed Forward Neural Network (FFNN) with one hidden layer with the sigmoid activation function and a linear activation function in the output layer. The training is performed using the Levenberg-Marquardt method;

- **Integration:** In this phase, the power consumption model is integrated with the functional behavior model of the circuit, as can be seen in Figure 2.1. This stage allows the circuit's behavior model and its power consumption behavior.

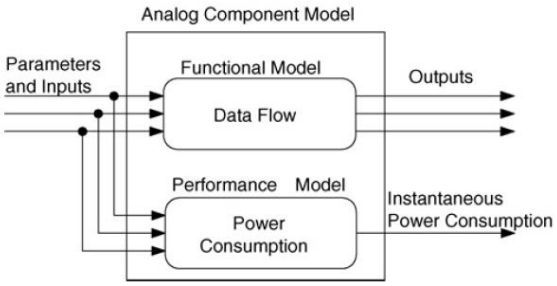

Figure 2.1 - Integration Power Consumption (from [12])

The case studies presented validate the methodology where the neural network was an amplifier, an analog to digital converter, and, finally, a wireless transceiver RF. The average error is about 1.53%, and the maximum error is about 3.06%.

In [13], a technique is proposed to model circuits or devices with collaborative stimulus generation. The main goal, reducing the difference between the model and circuit behavior, is achieved using a recurrent neural network (RNN). The algorithm architecture, shown in *Figure 2.2*, is the following:

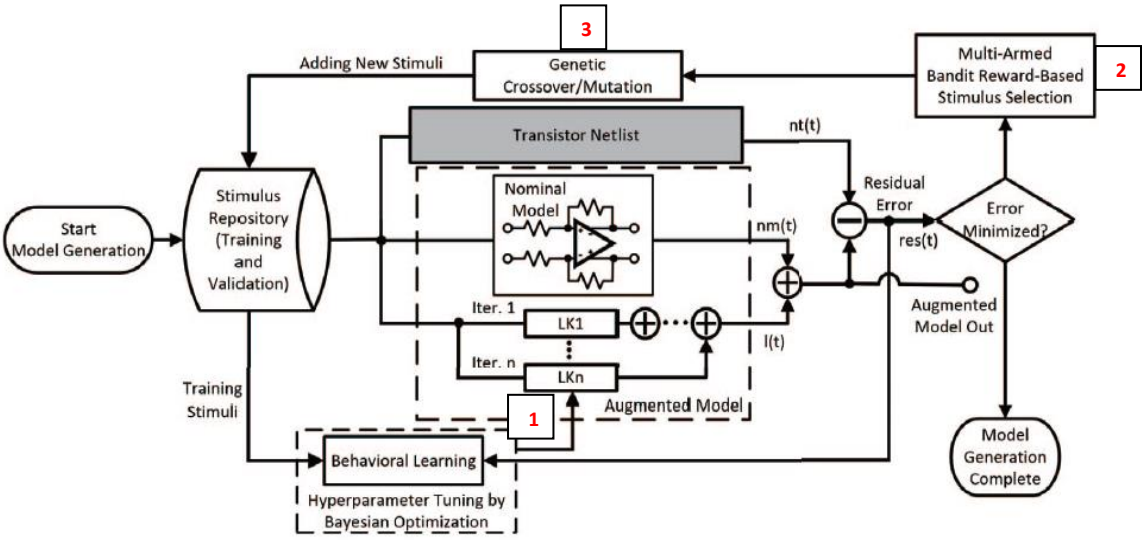

Figure 2.2 - Main Goal Model Architecture (from [13])

1) *RNN Model*

- Recurrent Neural Network is used to capture the memory and non-linearity relationship between the inputs and outputs of the circuit.
- It takes the current input, past inputs, previous outputs, and residues of previous iterations to obtain the output and the residual error, between the augmented model and the transistor netlist, in the present iteration.
- Bayesian optimization is used to find suitable hyperparameters for the RNN. It has the stimulus and the residues as input, and the output is the kernels (LKn).

2) *Test Stimulus Generation*

- Stimuli are assigned to a probability of being selected to the mutation phase. This probability is calculated as the error in that stimulus over the sum of all the stimulus errors. A higher error means that the stimulus has an underperformed learning kernel.
- In terms of the kernels, once learned, they will not be erased to ensure that previous learning behaviors are not unlearned.

3) *Genetic Crossover/ Mutation*

- The highest probable stimulus goes through an algorithm based on genetic algorithms that will generate a new stimulus that contains information and a certain level of randomness to find a new stimulus that allows training the model with new differences (between the model and the circuit) that were undiscovered.

The algorithm stops when the residuals are minimum. That is the case where the model cannot excite further differences between the circuit and the model.

Using this approach, Lei and Chatterjee [13] show the results for modeling different circuits:

1) *Low-Dropout Regulator:* the behavior model converges in 6 iterations with an RMS = 10.4 μV.
2) *RF Receiver:* the behavior model converges in 5 iterations with an RMS = 13.8 μV.
3) *Fully Differential Amplifier:* With a nominal gain of 2, a bias circuitry of 45 mm process, and a 1V supply, the model has an RMS of 69 μV.

The secondary goal is to design a diagnostic algorithm to detect transistor-level component anomalies. It is achieved using Volterra learning kernels since ML techniques are not the best for detecting anomalies due to the lack of cases when a failure exists. In Figure 2.3, there is an illustration of the proposed algorithm:

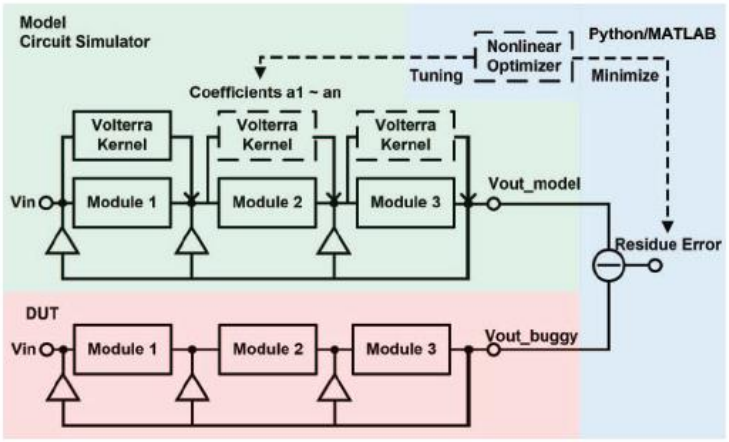

Figure 2.3 - Diagnose Algorithm Architecture (from [13])

A Volterra kernel is inserted for each module, and the residue is produced in each iteration. It is optimized using the Levenberg-Marquardt algorithm leafing to tune the kernels better. To find the bug, if the residual error is minimum and the kernel associated with a particular module gets updated, it means the correspondent modulo has a failure or a bug.

To test this approach, the authors used the following circuits, and for each one, a conclusion about the Volterra kernel placement is obtained:

1) *RF Receiver:* The error is minimum, measured by norm L2 if the Volterra kernel is placed across the buggy modules, indicating a correct diagnosis of the bug in the module.
2) *Sigma Delta:* A smaller error occurs when the kernel is across the first integrator rather than in the second integrator, indicating that the bug is in that first integrator.

To model the behavior of a Multiple-Input Multiple-Output circuit (MIMO), Hasani *et al.* in [14] propose the use of a Compositional Neural Network (CompNN) built with a nonlinear autoregressive Neural Network

with Exogenous Input (NARX) [15] and a TDNN. Each feature has a small-sized NARX allowing for an observable model where the behavior of each feature can be analyzed separately. The output of each NARX is the input of the TDNN, which computes the final output of the MIMO circuit. A CMOS Bandgap Voltage Reference circuit (BGR) is presented as a case study. *Figure 2.4* represents the features and architecture of the CompNN for this particular case.

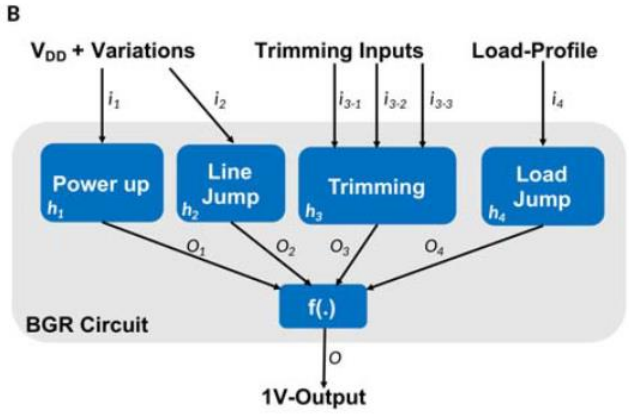

Figure 2.4 - CompNN architecture for MIMO BGR (from [14])

The h_1, h_2, h_3, h_4 are the small-sized NARX neural network for each of the input's features, power up, line jump, trimming and load jump, respectively. For trimming, the NARX has 3 delay components and 10 neurons in the hidden layer. For load jump and line, jump features, it has 3 delay components and 7 neurons in the hidden layer. Those parameters are found using the Grid Search method. The O_1, O_2, O_3, O_4 are the output of each small-sized NARX. The f(.) represents the TDNN that merges the outputs of each small-size NARX and generates the model output. It contains 200 neurons in the hidden layer. The data is acquired by input-output circuit SPICE simulation, and it is divided into the train (70%), test (15%), and validation (15%) sets. For training is given stop criteria, such as no improvement on the test set of the cost function being minimized, learning rate gets higher than 10^{10}, maximum training epochs are reached, and error is less than 10^{-7}. Finally, the results show an error-rate at the 1-V Output of 5% and a decrease in the simulation time by a factor of 17.

In [16], a technique is proposed to capture the behavior of an oscillator in its transient mode. The main goal is to capture the oscillator output waveform. Oscillators cannot be defined in a state-space realization because, in trapezoidal waveforms, the plateau turns impossible to trigger the transition of the state-space model, so the authors proposed a feedforward neural network (FFNN) with a Periodic Unit, which simulates the oscillator transient.

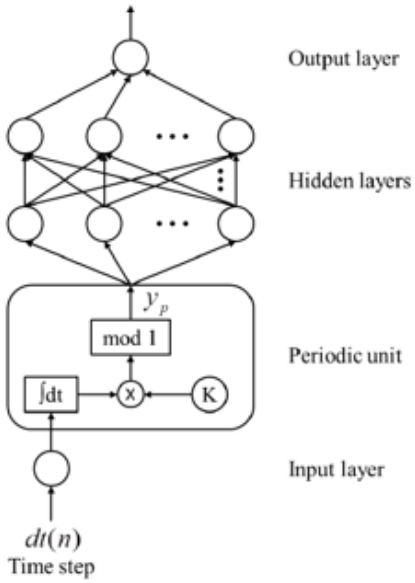

Figure 2.5 - Oscillator Model (from [16])

Figure 2.5 represents the model proposed by the authors. Analyzing it in more detail:

1) *Periodic Unit:* the inputs are a chronological sequence of time steps. The output of this unit is given by equation (1), where the phase is normalized between [0,1], solving the arbitrary phase behavior.

$$y_p(n) = (K_{osc} \sum_{1}^{n} dt(i))\ mod 1 \tag{1}$$

2) *Hidden Layers*: the nodes' values are obtained by equation (2), meaning that the value is obtained with the sum of the multiplication of the previous neuron value, of the previous layer, with the weight difference between the actual neuron layer and the previous one, plus a bias of the actual neuron layer.

$$y_j(n) = \tanh\left(\sum_{l=1}^{N} y_{prec}^{l}(n) w_h^{l,j} + b_h^j\right) \tag{2}$$

3) Output: the output value is obtained by equation (3), meaning that the output value is the sum of the multiplication of the previous layer neurons' output with the weight of the output neuron, adding the bias of the output one.

$$y_j(n) = \sum_{l=1}^{N_{prec}} y_{prec}^{l}(n) w_{out}^{l} + b_{out} \tag{3}$$

To train the FFNN, since the *mod1* function is discontinuous, it is assumed that *mod1* value is 1 in the discontinuities. Equation (4) is the objective problem that is achieved using the Levenberg–Marquardt Algorithm. Note that the minimizing parameters are the weight, the bias, and the oscillatory frequency so $phi = [w, b, K_{osc}]$.

$$minimize_{phi}\ \frac{1}{2}\sum^{N}(y_{out} - y_{predicted})\text{^}2 \quad (4)$$

As a case study is presented, a behavioral model of an invert ring oscillator with an oscillatory frequency of 0.1689GHz in the first case, and the correspondent FFNN has 10 neurons in a single hidden layer. In the second case, the frequency changed to 0.5734GHz, also with an FFNN with 10 neurons in the hidden layer. In both cases, the simulation time reduces by a factor of 10, comparing it to the circuit simulation.

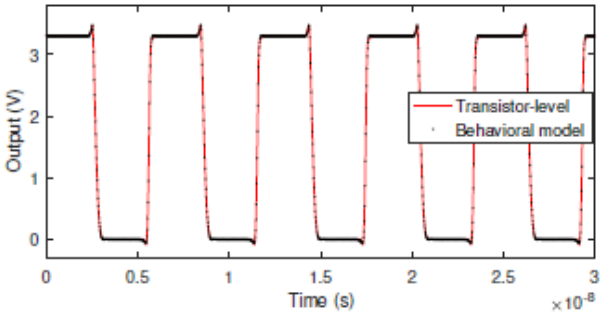

Figure 2.6 - Output Waveform for Oscillator Model, Case 1 (from [16])

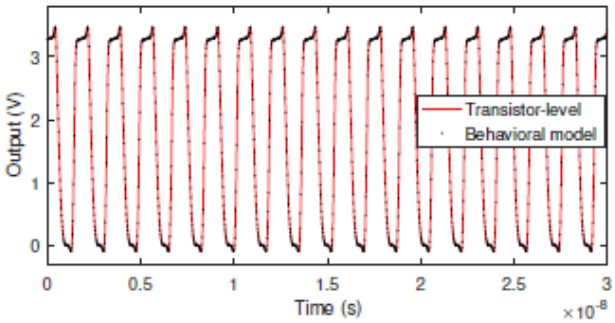

Figure 2.7 - Output Waveform for Oscillator Model, Case 2 (from [16])

Figure 2.6 and Figure 2.7 show the model's results comparing the model's output waveform with the circuit simulation.

Deepening the oscillators' behavior modeling, in [17], the same authors of [16] proposed a complementary approach to the previous one. In this case, a buffer is added to the oscillator's output. The presence of a buffer allows the oscillator to generate high-quality signals. Figure 2.8 shows the adjustment made to the previous model.

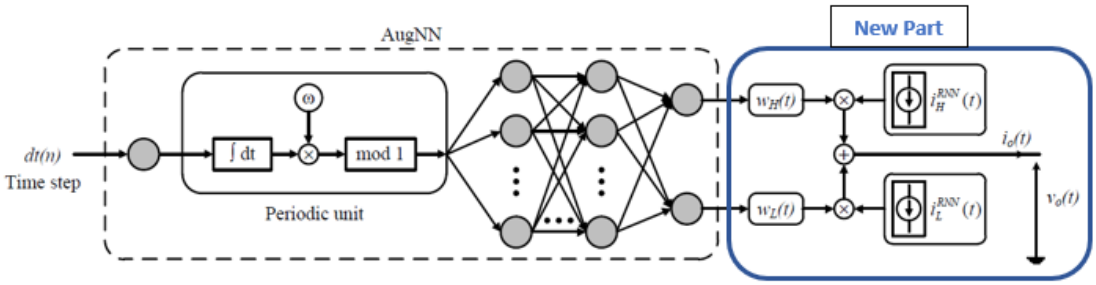

Figure 2.8 - Oscillator with Buffer Model (from [17])

The model was adjusted by adding a recurrent neural network (RNN) to capture the behavior of the buffer, which means obtaining the high and low currents, which are fundamental to obtaining the output current. Equation (5) combines the weights provided from the AugNN with the currents obtained with the RNN producing the output current.

$$i_0 = w_h i_h + w_l i_l \tag{5}$$

To train the RNN is connected a multilevel piecewise-linear voltage source at the buffer output to generate the excitation identification signals fixed at high and low. The FFNN training is performed as it was described in [16], as well as its equations remain the same. This approach is applied to a Ring Oscillator with a Buffer and 1.9481GHz of oscillatory frequency. The RNN has 10 neurons in its single hidden layer, and the FFNN in the AugNN has 15 neurons in the hidden layer. The results show a root mean square error (RMSE) of 0.0035 and a reduction in the simulation time of up to 96%.

Adding to what is present in [16] in [18] is added a voltage controller to, through voltage variation, control the instantaneous frequency. To proceed to that mapping is used a second FFNN is inside the Periodic Unit, as it can be seen in Figure 2.9.

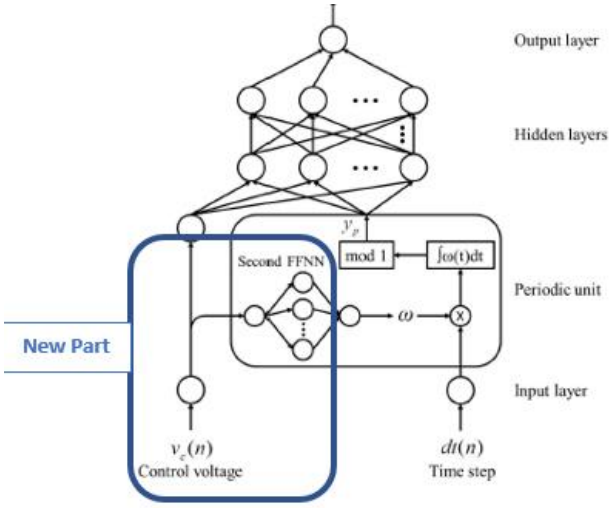

Figure 2.9 - Oscillator with a Voltage Controller Mode (from [18])

The second FFNN output is given by equation (6).

$$w(n) = \sum_{j=1}^{N} y_{h,pu}^{j} . {w_{out,pu}}^{j} + {b_{out,pu}}^{j} \tag{6}$$

Where w(n) is the oscillatory frequency, $y_{h,pu}^{j}$ is the output of the hidden layers of the second FFNN (which is inside the periodic unit), ${w_{out,pu}}^{j}$ is the weight of the output neuron of the second FFNN and ${b_{out,pu}}^{j}$ is the bias of the output neuron of the second FFNN. The $y_{h,pu}^{j}$ is given by equation (7).

$$y_{h,pu}^{j} = tanh\,(v_c(n) . {w_{h,pu}}^{j} + {b_{h,pu}}^{j}) \tag{7}$$

Where *tanh* is the hyperbolic tangent, $v_c(n)$ is the voltage controller input and ${w_{h,pu}}^{j}, {b_{h,pu}}^{j}$ are the weight and bias, of the respective hidden layer neurons. So, the output of the periodic unit is given by equation (8).

$$y_p(n) = (\sum_{1}^{n} w(i).dt(i))\ mod 1 \quad (8)$$

As can be seen from *Figure 2.9*, apart from the second FFNN, which allows controlling the oscillatory frequency, the rest of the model remains the same so the original FFNN equations are equal to what is proposed in [16] and the training methodology is the same, although the minimizing parameters are $phi = [w_{original}, b_{original}, w_{second}, b_{second}]$.

To prove the usefulness of that approach is proposed a Voltage Controller Oscillator (VCO) circuit model with a voltage controller sweeping in a range of [1V, 3V]. The training data is obtained through SPICE simulations. The model has the original FFNN with 2 hidden layers with 20 neurons in the first hidden layer and 10 in the second one, and the second FFNN has a single hidden layer with 5 neurons. The simulation time reduces 93%, and the model contains a root mean squared error (RMSE) of 0.61%.

In [4], ANNs are used to model the behavior of RF/microwave circuits. The authors state that using neural networks is a better alternative to conventional numerical methods, such as Support Vector Machine (SVM). They presented examples of how ANNs model signal propagation delays in Very Large-Scale Integration (VLSI) interconnect networks, coplanar waveguide discontinuities, and MESFETs to prove their point. In [19], the same authors present a study on ANN nonlinear techniques for modeling large/small signals of transistors and dynamic recurrent neural networks, which are then used to model circuits. To review the proposed techniques is used practical microwave examples. In [20] is proposed a method for modeling coplanar waveguide circuits (CPW) using ANN that is based on Electromagnetic (EM) simulations. In this approach CPW transmission lines, 90º bends, short circuit stubs, open-circuit stubs, step-in-width discontinuities, and symmetric T-junctions are modeled individually using ANN based on EM. This multilayer feedforward ANN is composed of one input layer, one hidden layer, and one output layer, and the training is performed using an error-backpropagation learning algorithm. As a case study, the authors proposed to model a CPW folded double-stub filter and a 50-3-dB power-divider circuit. The proposed architecture (EM-based ANN) can be applied in other classes of microwave and millimeter-wave circuits. One of the major drawbacks of this approach is that it takes a long time in the training phase to obtain an accurate model, and the efficiency can be low, so in [21] is proposed a solution to model nonlinear RF microwave devices using a Radia Basis Function (RBF) MLP with Resilient Backpropagation (Rprop) algorithm for training. To address this problem, the authors proposed a “divide to conquer” technique where the complex problem is divided into sub-problems, which are modelled by individual neural networks. This approach is claimed by the authors to improve the efficiency of the EM-based ANN. The RBF network is used to approach the input variables locally, and its output is the input of the MLP network that will perform a global approach since it improves the generalization capacity of the model, acting as an output network. It is graphical explained in Figure 2.10.

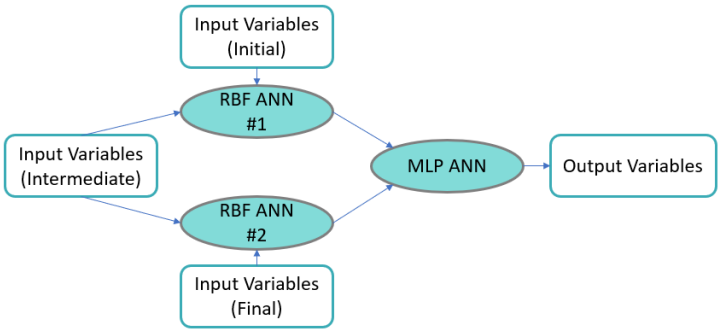

Figure 2.10 - RBF/MLP Proposed Model (adapted from [21])

To demonstrate the improvement made in EM-based ANN, the authors first proposed a uniplanar compact-photonic bandgap (UC-PGB) rectangular waveguide. In the hidden layers of the three ANN are used 10 neurons, there are 10000 training epochs, and the obtained MSE is around $2 * 10^{-5}$. Second, is proposed a patch antenna with PGB substrate. In the hidden layers of the three ANN are used 15 neurons, there are 10000 training epochs, and the obtained MSE is around $2 * 10^{-4}$. The authors also conclude that using a single MLP ANN or a single RBF ANN has less generalization capacity, independently of the number of hidden neurons, rather than combining them. The results can be seen in Figure 2.11 and Figure 2.12.

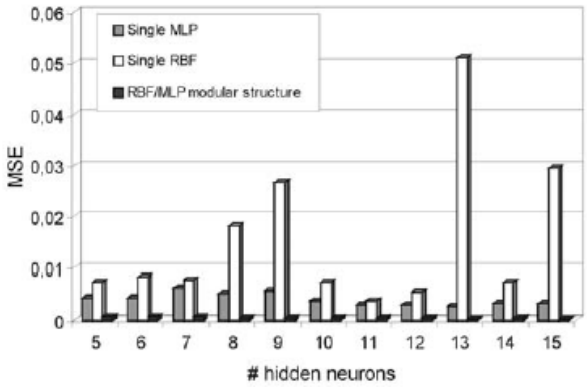

Figure 2.11 - RBF vs MLP vs RBF + MLP for UC-PBG waveguide (from [21])

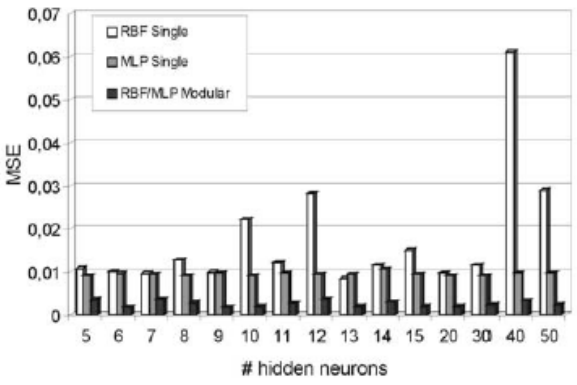

Figure 2.12 - RBF vs MLP vs RBF + MLP for patch antenna PBG substrate (from [21])

Another way to address the behavior modeling problem is using the Support Vector Machine (SVM). In [22] is used an SVM for regression problems to model analog circuits, the kernel used is the radial basis function (RBF), the error function is ε since it allows a better sparsity of the solutions, and the ε-function error contains a fixed and symmetrical margin, however, if the data comes from real-world problems a fixed and asymmetric margin or even a non-fixed and asymmetric margin is preferable. To preprocess the data is necessary to use cross-validation to prevent overfitting and an exponential Grid Search from finding the parameters C (penalty degree for regression errors), γ and ε (RBF parameters). The use of exponential Grid Search will allow more straightforward parallelization. The dataset should represent the data well, including the circuit dynamic, static properties, excitation waveform, and frequency spectrum.

As a case study is presented a SCFL Buffer Cell with the objective to approximate the voltage and current output curves, the results are a mean squared error (MSE) of $2.62 * 10^{-14}\ A^2$ for the current and $3.20 * 10^{-6}\ V^2$ for the voltage. For the Resistive Mixer circuit, an MSE of $2.89 * 10^{-7}\ V^2$ is obtained for the output voltage. Finally, is analyzed a Voltage Controller Oscillator and Figure 2.13 shows the proposed model, where the three SVM are used to estimate the parameters that allow the signal generator to output a voltage that is similar to the Voltage Controller Oscillator voltage output.

Figure 2.13- VCO SVM Model (from ([22]))

Training the model only needs 10 to 20 input-output pairs of measurements due to the reduction of the complexity provided by the SVM method. The model output is compared to the adjacent period jitter, where it shows a competitive accuracy. However, no MSE is presented. It is also worth mentioning that is not referred to in this article the time reduction, which is a crucial aspect when modeling circuits, and it is an important metric to compare with the ANN model for the same circuits. In [23], an SVM is used to classify the region as infeasible or not and prune the ones that are infeasible, excluding a large portion of the design space. The generated feasibility classifier can work in conjunction with the performance macro models in analog synthesis, providing the designer with a good starting point.

2.2 ANNs vs. SVM: Advantages and Drawbacks

Since the state-of-the-art review presented methods for modeling circuits with ANN and SVM and this work's focus is on using ML for modeling, it is essential to discuss the advantages and disadvantages of both. Therefore, the following advantages were identified for the ANNs:

- High-speed "simulations" and low computation complexity;
- No complete knowledge about the physics of the devices is needed;
- Overcome the accuracy limitation of the numerical models;
- Can handle optimization constraints;
- Due to a high sample efficiency, they show a higher accuracy in fewer training epochs.

These advantages make the ANNs the most suitable technique for modeling the circuit's behavior since they can also deal with nonlinear problems mapping the relationships between the input and output data. As for drawbacks, the ANNs have the following ones:

- A lot of data is required to achieve high accuracy;
- Numerous hyperparameters have to be tuned to improve accuracy and avoid overfitting;
- It can diverge with a high learning rate.

The other model presented in the state-of-the-art that is capable of mimicking the behavior of analog circuits was the SVM, which has the following advantages:

- Requires less data than the ANNs and does not converge to a local minimum of the cost function;
- Higher accuracy in classification problems;
- Has well-documented approaches to avoid overfitting, and, using the Kernel Trick, it can map nonlinearly separable data to a feature space where the data is linearly separable.

Several disadvantages of the SVMs were identified:

- Not suitable for large datasets;
- Have difficulties dealing with noisy data;
- Hyperparameters need to be tuned specifically to each circuit dealt with, leading to a lack of generalization;

As mentioned in the previous sections, the use of ANN for modeling circuit behavior has been widely researched and discussed. These neural networks with the number of hidden layers, number of neurons, and weights correctly set can provide accurate results allied with high-speed “simulations”, avoiding the need for a detailed and time-consuming physical base formulation. Considering all the advantages and drawbacks of the ANNs and SVMs, the ANN methodology is therefore chosen for this work.

2.3 A (Very) Brief Machine Learning Overview

Since the work has a huge component of ML, it is useful to give an overview of the used methods in order to get a better comprehension of the models proposed in the state-of-the-art. It is also important to review some dimensionality reduction methods as well as model selection methods, which are important to tune the model with the best hyperparameters. These sections are intended to ease the reading of the remainder of this thesis only. A deep description of ML is not the scope and can be found in dedicated literature.

2.3.1 Supervised and Unsupervised Learning

ML systems can be divided into four classes: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.

Supervised Learning

This class means uses a full set of labeled data for training a model. Each example in the training dataset is tagged with the answer the algorithm should come up with. This training approach makes the model capable of when receiving new data, comparing it with the training data, and making a prediction. Figure 2.14 is presented a flow diagram of the supervised learning approach.

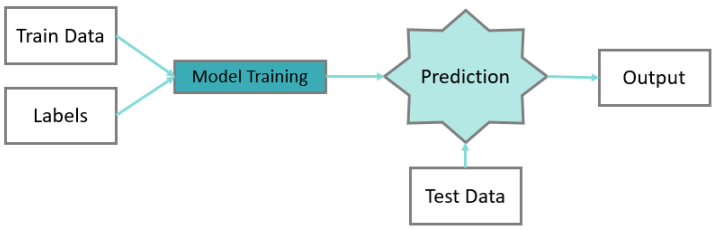

Figure 2.14 – Supervised Learning Flow

In this type of learning, all the classical machine learning methods can be applied, such as Linear Regression, Support Vector Machine, Logistic Regression, Naïve Bayes, among others, since those methods require labeled training data in order to be able to compare a new data with it to provide an accurate prediction.

Unsupervised Learning

This group is considered the opposite of supervised learning since it uses unlabeled data. It has the goal of describing the association, relation, and patterns among the set of input measures. This is a growing area since obtaining labeled datasets is no easy task. This type of learning can be applied using neural networks, which attempt to automatically find relationships in the data by extracting useful features and analyzing its structure. Accordingly, to the problem, unsupervised learning can organize the dataset in the following ways:

1) *Clustering:* The deep learning model looks for training data that are similar to each other and groups them together. After that, several clustering methods can be applied, such as, for example, C-Means Clustering or Hierarchical Clustering
2) *Anomaly detection:* In this approach, the deep learning model can detect the outliers in a dataset.
3) *Association:* The deep learning model can predict the remain attributes by looking at a key attribute on a certain data point.
4) *Auto Encoders:* This approach takes the input data, compress it into a code, and tries to recreate the input data from that compressed code. In images, this approach can remove visual noise, improving image quality. Another application of this method is the fact that when looking at the given input data, it can randomly generate new data that is similar to the input one. This neural network is also commonly used for dimensionality reduction.

Semi-Supervised Learning

It is the intermediate learning process, where in the training set, some data has labels, and other does not. This method is useful when labeling data is time-consuming and very difficult, for example, medical images. For this purpose, generative adversarial networks (GANs) are commonly used. This network is divided into two networks (as can be seen in Figure 2.15). The first one is called the generator and is responsible for

creating new data that is similar to the input one, the second one, which is called Discriminator, is responsible do evaluate the new inputs and deciding if they make part of the training set or not [24].

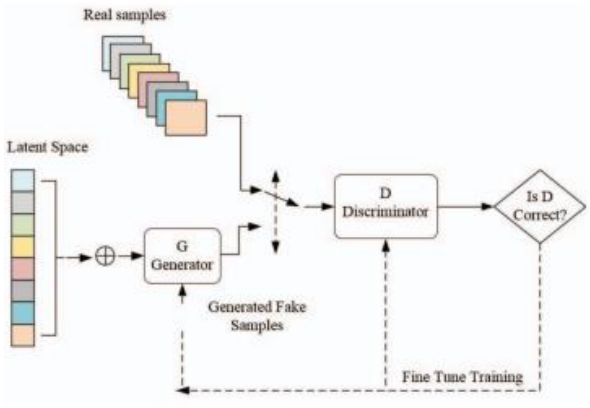

Figure 2.15 - GANs Design Flow (from [24])

Reinforcement Learning

Reinforcement Learning (RL) also stands between supervised and unsupervised learning because no set of actions is given to the agent, and it acts in an environment where sequential decisions need to be made with limited feedback. The main goal is to develop an agent capable of interacting with the environment and learning how to behave in it, and depending on the actions taken, received rewards or punishments, which are the only information the agent has, in a try-error-try way [24]. The agent's objective is to develop a policy that indicates which action should be taken in order to maximize the reward. To attain that goal, the agent needs to explore and interpret the given reward and come up with optimal decisions for any state. One relevant characteristic of RL is that in any situation, the agent needs to choose between exploiting its knowledge or exploring actions that have never been tried before.

2.3.2 Tribes of Machine Learning

Machine Learning is a vast area with a wide number of methods to be applied, and in [25], the authors define five tribes in which ML area can be divided, illustrated in Table 2.1. All five tribes have unique methodologies, so when using ML, it is important to know the specifications of the method, analyze the different approaches, and then select the one that better adapts to the problem.

Table 2.1 - Tribes of Machine Learning (from [25])

Tribe	Origins	Master Algorithm	Examples of Methods
Symbolist	Logic	Inverse Deduction	Decision Trees
Connectionist	Neuroscience	Backpropagation	Deep Neural Networks, Perceptron
Evolutionaries	Evolutionary Biology	Genetic Programming	Genetic Algorithm
Bayesians	Statistics	Probabilistic Inference	Naïve-Bayes, LDA, PCA
Analogizers	Psychology	Kernel Machines	SVM, KNN

Briefly, the Symbolist tribe uses symbols, rules, and logic to represent knowledge. The Connectionist tribe aims to recognize relationships and patterns between inputs and outputs in a dynamic way using weights, which are adjustable. The Evolutionaries tribe generates a population and obtains the fitness of that population and decides who will reproduce or not in order to generate offspring which are better than the previous population in order to achieve the goal through generations. The Bayesians tribe is based on the Bayes theorem to obtain the likelihood value. Finally, the Analogizers tribe aims to optimize a function by taking into consideration the constraints.

In light of this work, modeling the behavior of a circuit or device, obtaining, for example, the output waveforms, is a regression problem. The Bayesians tribe and Analogizers tribe are commonly used (but not only) for classification problems, so they are discarded from the possible methods to use in this work. The Symbolist tribe is also discarded because in cases with a lot of constraints developing an efficient solution presents a tremendous challenge and is time-consuming. As mentioned in the motivation chapter of this work, one of the main goals when applying Machine Learning to analog circuits is to improve the simulation time, so Evolutionaries tribe approaches might be discarded due to the simulation time that is associated with natural selection of the fittest individuals in each different strategy.

As mentioned in previous sections, the use of ANN for modeling circuit behavior has been widely researched and discussed. These neural networks with the number of hidden layers, the number of neurons, and weights properly set are capable of providing accurate results with very fast simulations. Considering all the tribes and the reasons provided, the ANN methodology is the chosen one for this work, so a brief overview of Artificial Neural Networks is done in the next section.

2.4 Artificial Neural Network Overview

Widely researched, ANNs prove to be flexible, adaptative to any type of problem, from the simplest to the more complex ones, and perform predictions in a short time. Those advantages allying to the fact that ANNs are very efficient for predictions, are the main reason why it is used in this project because it is intended to have an efficient modulation of the circuit behavior in a short period of time. In this section, the ANNs general architecture and behavior are explained as well as its hyperparameters, their influence on the model behavior, and how to automate its choice.

2.4.1 General Architecture

First, it is important to define the basic unit of the ANNs, the Neuron. This unit is the elementary artificial neural network capable of being trained. The artificial neural network is a composition of numerous neurons. Figure 2.16 is represented the basic unit, a Neuron.

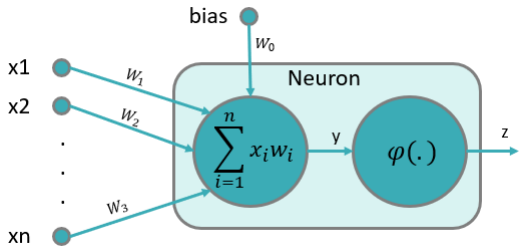

Figure 2.16 - Elementary ANN - Neuron

In equation (9) is given the output value of a neuron:

$$z = \phi\left(\sum_{i=1}^{n}(w_i x_i) + b w_0\right) \tag{9}$$

The x_i are the input values, ϕ is the activation function that is explained in section 2.3.3, w_i are the weights and b are the bias, which are hyperparameters and its tuning is also explained. If bias is considered to be 1, the previous equation can be rewritten is a simplistic way presented in the equation (10).

$$z = \phi\left(\sum_{i=0}^{n}(w_i x_i)\right) \tag{10}$$

To deal with more complex problems, the neuron itself is not capable of capturing patterns and behaviors between inputs and outputs of non-linear problems, so neurons linked to others originate a multilayer perceptron, which is capable of dealing with more complex problems. The ANN is illustrated in Figure 2.17.

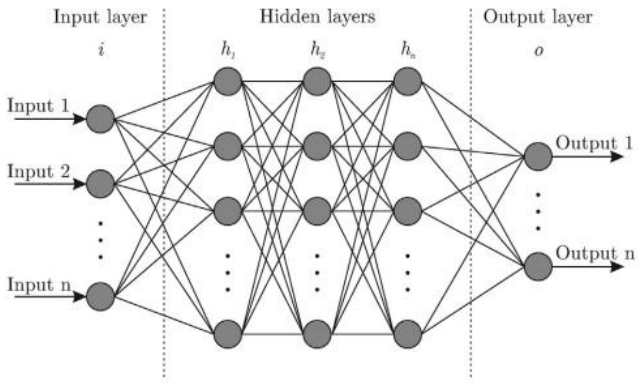

Figure 2.17 - ANNs General Model (from [25])

ANNs have the input layer with the number of neurons equal to the number of features present in the dataset. The output layer has the same number of neurons as the number of outputs desired. The number of hidden layers and its number of neurons is a hyperparameter to be tuned, and there is no rule to follow. Its tuning is based on the experience of the operator.

For the case of fully connected ANNs, all the neurons are linked to each other's, meaning that the input of the next layer neurons are the outputs of all the previous layer neurons, as Figure 2.17 shows. The input layer receives the sum of the input values multiplied by their respective weights and applies an activation function generating the output of the neuron in the hidden layers or in the output one. The output of the previous layer neurons is multiplied by their respective weight and then sums them all, and the result passes through an activation function (a non-linear mathematical function) which gives the output of the neuron. This learning process consists of adjusting the neural network weights so that the network's transformation of input into outputs optimizes the loss function [25].

The main objective of the model's designer is to create and train a model with limited examples as references in order to obtain an efficient model capable of mapping a given input to an output that is close to the expected one. There are numerous types and variants of the ANNs. Despite being easy to interpret, the output value is hard to materialize and describe the behavior of the network during the training and test phase.

As the complexity of the problem increases, new types of ANNs are developed, such as recurrent neural networks, Elman recurrent neural networks, and long-short term memory recurrent neural networks, but all of these variants keep the same principle of the multilayer perceptron explained previously.

Backpropagation

The learning phase of the ANN model consists of updating the weights of each neuron using the backpropagation algorithm. Given the inputs of the network and the initial weights and bias, the output ($\hat{y}$) is obtained by forward propagation. The output ($\hat{y}$) is compared to the desired output (y), and the loss L is computed between them. Since this work deals with regression problems, the appropriate loss function is the Mean Squared Error (MSE) given by equation (11).

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{11}$$

The derivative of the loss with respect to the weight is computed for each neuron's weight and consequently, the weights are updated according to equation (12).

$$w_{i+1} = w_i - \Delta w \tag{12}$$

The w_{i+1} is the new value of the weight, w_i the previous weight value, and Δw the wight update proportion, which is given by the equation (13).

$$\Delta w = \left(\eta \frac{\partial L}{\partial w} w_i\right) \tag{13}$$

The η is the learning step that controls how quickly or slowly the neural network learns a problem. A large learning step makes the model learn faster at the cost of arriving at a sub-optimal solution for the weights,

and a small learning step makes the model learn slow but reach an optimal solution or even the globally optimal solution for the weight's values, so the value of this parameter should be chosen with criteria, and its tuning is explained in the following sections.

2.4.2 Optimizers

To optimize the value of the weights, numerous types of algorithms to perform that task exists. The Gradient Descent is the most used algorithm, and backpropagation is used to find the best gradient leading to the train of the neural network. It calculates in which direction the weights should be altered so that the loss function can be minimized. If we compile the equation (12) with equation (13) the Gradient Descent equation is obtained, which is displayed in equation (14).

$$w_{i+1} = w_i - \left(\eta \sum \frac{\partial L}{\partial w} w_i\right) \tag{14}$$

The Gradient Descent has an iterative variant, the Stochastic Gradient Descent, where the algorithm sweeps through a subset of the training set and update the weights. The subset can have 1 sample of data. Mathematically this algorithm (present in equation (15)) is identical to the Gradient Descent apart from the gradients where in this method it uses the one sample gradient instead of the sum of the gradients .

$$w_{i+1} = w_i - \left(\eta \frac{\partial L}{\partial w} w_i\right) \tag{15}$$

For relatively simple problems, this optimizer is a well-chosen approach, although in more complex problems, it might take a very long time to converge and might even be stuck at a local minimum. To fight those problems is useful to use variants of this optimizer. The most common one is the addition of a Momentum Term to the Gradient Descent optimizer. It accelerates the convergence making the weights flow in the relevant direction reducing the fluctuations in the irrelevant direction. Although, as it is represented in equation (16), it requires the tunning of one more parameter, which needs to be carefully chosen.

$$w_{i+1} = \boldsymbol{\gamma}.\boldsymbol{w_i} - \left(\eta \frac{\partial L}{\partial w} w_i\right) \tag{16}$$

If the momentum term coefficient ($\boldsymbol{\gamma}$) is too high, it may skip a local minimum and keeps increasing indefinitely, so making to the Gradient Descent optimizer, already provided with the Momentum Term, a slight alteration to modify the current weight value (called Nesterov Accelerated Gradient in equation (17)) is possible to slow the algorithm when approaching a local minimum and then accelerate it in the other occasions solving the momentum term problem.

$$w_{i+1} = \boldsymbol{\gamma}.\boldsymbol{w_i} - \left(\eta \frac{\partial L}{\partial w} (w_i - \boldsymbol{\gamma}.\boldsymbol{w_i})\right) \tag{17}$$

Recently, new optimizers were discovered giving more powerful tools to optimize the values of the weights. Adam optimizer uses and adaptative moment estimation as an algorithm technique for the gradient descent. This optimizer is useful for large datasets, which has a numerous amount of data or parameters, since it requires less memory, and it is efficient. Mathematically the Adam optimizer is present in equation (18), where m_t and v_t are two moment vectors which are represented by equations in (19).

$$w_{i+1} = \boldsymbol{w_i} - \left(\frac{\eta}{\sqrt{v_t + e}}\right) * m_t \tag{18}$$

$$m_t = (1 - \beta_1)\frac{\partial L}{\partial w} + \beta_1 m_{t-1}$$

$$v_t = (1 - \beta_2)(\frac{\partial L}{\partial w})^2 + \beta_2 v_{t-1} \tag{19}$$

AdamW is an improvement of the Adam optimizer. AdamW has an improved implementation of weight decay, which is a form of regularization lowering the chance of overfitting.

2.4.3 Regularization

Machine Learning models, including ANNs, are subject to overfitting. A model is said to have overfitting when the error between the proposed curve referred and the data is zero or almost zero, so the proposed curve fits in perfection the data curve, leading to a model that is too well adjusted to the given training data. This effect leads to a model that is not generalized [26], leading to a negative impact on the performance of the model in unseen data. In Figure 2.18, it is possible to see the overfitting effect, where the testing error (when the model performs in unseen data) starts to increase significantly.

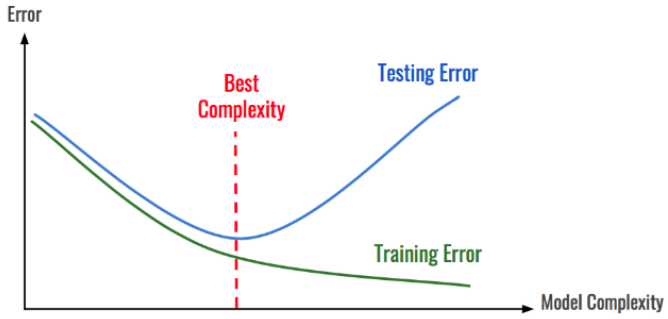

Figure 2.18 – Overfitting (from [27])

To prevent this problem is advisable to simplify the model, not using a high number of neurons in the hidden layers as well as not using a large number of hidden layers. This parameter tuning is explained in the next sections. Other criteria and methods can be applied to deal with overfitting, such as introducing a stop criterion called Early Stopping, which means if in a predefined number of iterations the testing error keeps increasing, training will stop to prevent overfitting. If we look at Figure 2.18, the “Best Complexity” point is exactly where the Early Stopping method should stop the training process. It is the point where after that if the model keeps training, it will lead to the overfitting effect. Another approach is to add a penalty (L1 or

L2) to the loss function so that the model is obligated to make compromises on the weight's values since they can no longer be arbitrarily large, or it will penalize the loss function. This generalizes the model and helps to prevent overfitting. Another effective approach is to add dropout layers and dropout connections. This deep learning technique randomly deactivates (masks) neurons and connections during the training phase making the network redundant since it can no longer rely on the information in specific neurons and connections because they might be deactivated, therefore, making the model more general. Posterior of the training phase, all the neurons and all the connections are reestablished [27].

2.4.4 Hyperparameters Tuning

The main disadvantage of the ANN is the number of hyperparameters that should be adjusted in order for the model to perform as well as expected. In this sub-section, an overview of each hyperparameter, explaining not only their importance in the network but also strategies to find the most appropriate values, is provided.

Hidden Layers

The more hidden layers, the more complex the ANN. A complex neural network is not always synonymous with good results because the model tends to overfit the training set. Apart from this disadvantage, a complex network also increases the time and computational power required in the training phase. In contrast, an ANN with a low number of hidden layers might not be able to capture the input-output relationship accurately. So, it is important to pay attention when choosing this parameter's values. Unfortunately, there is no formula or metric to choose the correct number of hidden layers. The current approaches are based on experience, intuition, and slight adjustments, which consist in gradually increasing the number of hidden layers until the results (i.e., accuracy) show no further improvement, meaning that adding more layers can increase the overfitting effect.

Hidden Neurons

Since the number of neurons in the input layer is determined by the number of features, and the number of neurons in the output layer is determined by the number of desired outputs, only the number of neurons in the hidden layers needs to be carefully chosen since a large number of it may lead to overfitting effect and a small one may lead to underfitting effect. Once again, there is no formula or metric capable of defining the right number of neurons in that layer. So, the choice of the number of neurons relies on the model's designer experience and on some heuristic methods present in [28] and in [29]. The number of neurons in the hidden layers should be between the number of input layer neurons and the number of output layer neurons. The hidden layer size should not have more neurons than twice the number of neurons in the input layer. However, as mentioned by the author of [29], despite this heuristic approach, the number of hidden neurons always depends on the problem. Apart from this heuristic, try-error is the de facto approach, such as starting with the same number of hidden neurons for every hidden layer and gradually increasing

the number of neurons, taking into consideration the performance, until the point where the model starts overfitting.

Learning Rate

This parameter is important when the ANN is updating the weights. This hyperparameter indicates how fast the learning is performed, in other words, how fast the model responds to the estimated error in each update and the capability to converge to a minimum. As mentioned in section 2.3.2, choosing this parameter is a hard task since a large learning rate makes the model learn faster at the cost of arriving at a sub-optimal solution for the weights, and a small learning rate makes the model learn slow but reach an optimal solution or even the globally optimal solution for the weight's values. As with any other hyperparameters, there is no function or metric to pick up the right value, so try-error is the only way to do that. So, since it is desired a model that performs faster, start choosing a high value for the learning rate and gradually decreasing it until the model performance does not improve can be an approach to find the correct learning rate for each problem.

Activation Function

According to the neuron's value, the activation function decides if the neuron should be activated or not and its behavior and capabilities. There are an enormous variety of activation functions that have already been studied, which have their purpose, advantages, and disadvantages. In the backpropagation algorithm is important to have differentiable activation functions. Otherwise, this algorithm does not work since it is not able to compute the loss derivative with respect to the weights. In regression problems, where the output is a real number, the activation function should be linear (Figure 2.19), or a Rectified Linear unity (ReLu) computed through $f(z) = \max\{0, z\}$ (Figure 2.20). It is important to mention that RELU is not differentiable in 0. The main advantage of the ReLu is that being a non-linear function, it is capable of encoding non-linear relationships. The main disadvantages are since it does not have an upper bound, it is susceptible to output values very high compared to the desirable ones.

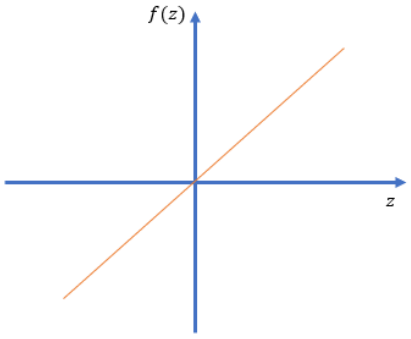

Figure 2.19 - Linear Activation Function

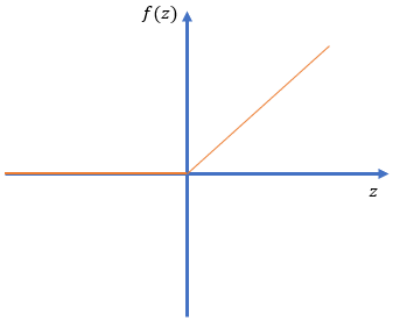

Figure 2.20 - ReLu Activation Function

To correct the ReLu drawbacks some variations of that method are performed such as Leaky ReLu and exponential linear unity (ELU) function. The ELU in z = 0 is smoothed so it makes the discontinuity problem vanish.

In classification problems, if it is being performed a binary classification, the sigmoid or hyperbolic tangent functions are commonly chosen since in the cases where values are too high, it establishes a limit. If the problem has multiple classes, it is important to know if it has non-exclusive classes or mutually exclusive classes. In the first case, one data point can have multiple labels assigned, so the sigmoid function must be used as an activation function. For the second case, if a data point has a certain label, it cannot have another one, and in that case, softmax is used as the activation function.

2.5 ANNs and Verilog-A: An Overview

Once the ANNs for the circuit models are created and trained, their descriptions must be implemented in HDL, such as Verilog-A, to use the model during circuit simulation. This section presents a summary of state-of-art models involving Verilog-A and neural networks. In [31] is presented an approach to convert a neural network with backpropagation to Verilog-A using a top-down methodology. In Figure 2.21, we can see the neural network with its weights, and on the right is the correspondent Verilog-A model that takes as inputs the weights and returns the bias for each neuron in each layer. Before starting coding in Verilog-A is extracted from the neural network its respective equations. In the first part, the needed variables are declared. More specifically, in the *directional specifications,* the input and output are defined, in *discipline definition* are specified, which inputs and outputs are electrical, and in *parameter declaration* and *variable declaration,* auxiliary variables are created. The second part is where the behavioral statements are declared, and the initial conditions, such as bias and weights, are defined. In the third part, the teaching signal is set, and the model is trained using the equations that represent the neural network. In the last part, the output is obtained, and the respective error and the weights are updated.

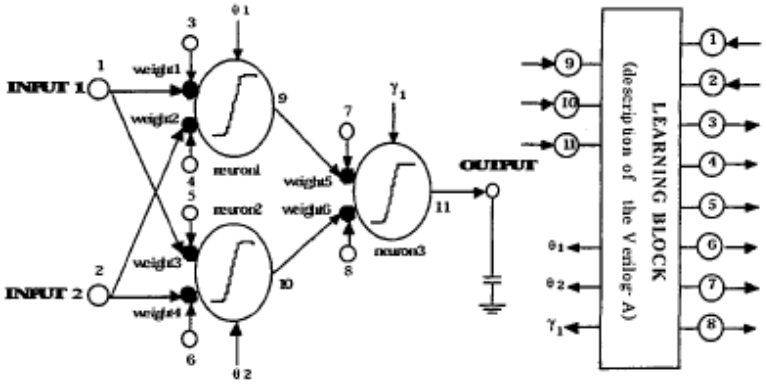

Figure 2.21- In Left the Neural Network, in the Right the correspondent Verilog A model (from [31])

In [32] is proposed the translation of an RNN behavior model to Verilog-A in order to make a modified nodal analysis of the model. For that purpose, the differential equations of the RNN need to be converted to differential equations. It also allows using a different time step in evaluating the model.

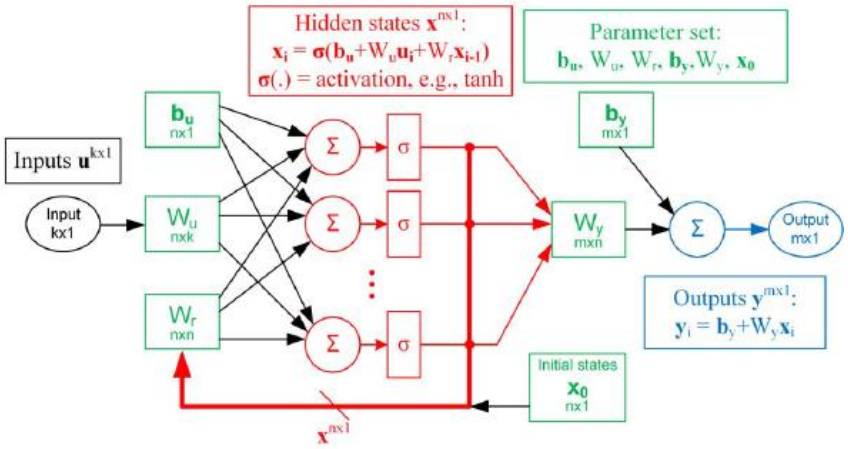

Figure 2.22 - RNN Model focuses on the hidden layer and output layer (from [32]).

Figure 2.22 presents the structure and the model equations for one hidden layer and the output one. It is important to mention that this RNN will account for a key property of circuits, the zero-in zero-out (ZIZO), meaning that the DC current, when all terminals are biased at the same potential, is zero. From Figure 2.22 it is also possible to extract equations (20) and (21). The first one relates to the output of the hidden layer (x_i) with the activation function (σ), bias (b_u), the respective model parameters (W) and the input (u_i). The second one represents the model output as function of the output of hidden layer and the model parameters.

$$x_i = \sigma(b_u + W_u u_i + W_r x_{i-1}) \tag{20}$$

$$y_i = W_y x_i \tag{21}$$

To convert the difference equations to differential equations in order to implement the model in Verilog-A is important to take into consideration the approximation of the RNN inputs and hidden states and the range of the model parameters (expressions in (22)).

$$\begin{gathered} u_i = u + (1-a)h\dot{u} \\ x_{i-1} = x - ah\dot{x} \\ x_i = x + (1-a)h\dot{x} \\ 0 < a < 1 \\ h > 0, \text{ where h is the RNN time-step} \\ \Omega_0 \end{gathered} \tag{22}$$

Finally, by combining equations (20) and (21) with the approximations of the RNN is possible to reach the differential equations (23), (24), and (25).

$$x + (1-a)h\dot{x} = \sigma[W_r(x - ah\dot{x}) + W_u(u + (1-a)h\dot{u}) + \sigma^{-1}(\Omega_0)] - \Omega_0 \tag{23}$$

$$y = W_y x \tag{24}$$

$$i[1:n] = x - (1-a)h\dot{x} - \sigma[W_r(x - ah\dot{x}) + W_u(u - (1-a)h\dot{u}) + \sigma^{-1}(\Omega_0)] + \Omega_0 \tag{25}$$

Using these equations is possible to build the Verilog-A system present in Figure 2.23.

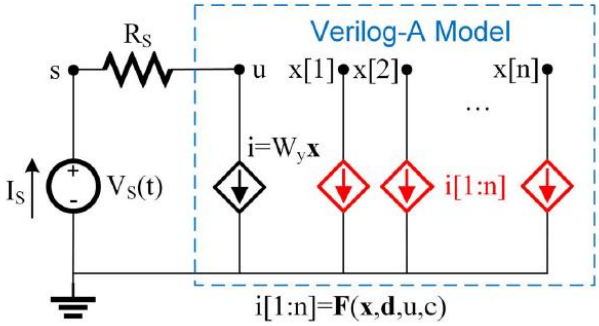

Figure 2.23 - Verilog-A Model (from [32])

It is important to mention that when the "a" parameter is close to 1, it affects the numerical stability of the model (it can affect the convergence of the model). To solve this problem is proposed the use of a time-step for the transient simulation that is 1/10 of the time-step of the RNN. As a case study, to prove the usefulness of this approach, is used an Active Rail Clamp Circuit with 6 transistors and 2 passive elements in a 130 nm CMOS. The data is acquired through physical circuit simulation, and the model is trained using the gradient descent algorithm. To evaluate the model, it is converted to Verilog-A and is used with an unseen and more complex stimulus, a piecewise-linear voltage source with 1Ω of output resistance and a waveform generated by an IEC61000-4-2 ESD tester. In Figure 2.24 and Figure 2.25 is presented the respective results.

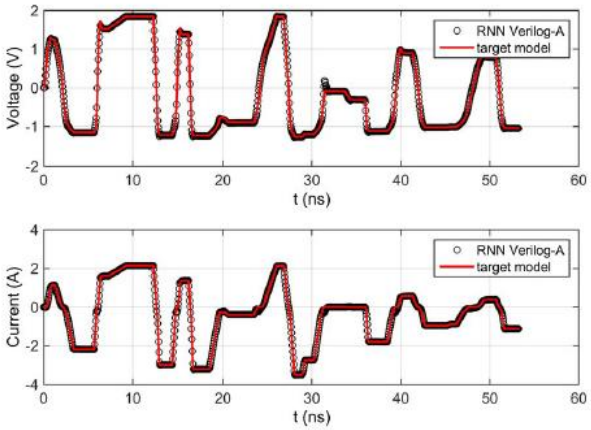

Figure 2.24 - PWL Results (from [32])

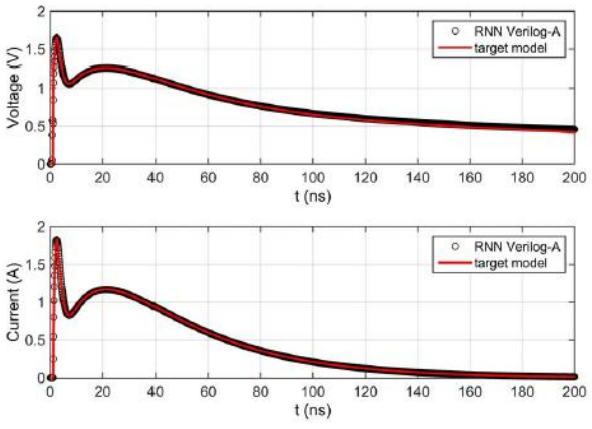

Figure 2.25 - IEC Results (from [32])

The voltage error is 1.39% for PWL and 1.91% for IEC. The current error is 0.77% for PWL and 0.20% for IEC.

The authors in [33], due to the strong dependence, in this research area, on EDA-vendor solutions, create a Verilog-A API capable of converting an ANN from ML tools to Analog Circuit Design tools. To ensure easy use of this API, the ML model should be serializable to a computer-readable format. It also should be loadable and executable inside an analog circuit simulation context and should be compatible with one or more widely accepted ML libraries. In this work, two different APIs are built. The first one is to make the model inference which includes two simple VerilogA system tasks for load and evaluating the model. The second API is used to construct and train the model.

As a study case, the authors of [33] use a Gaussian Process Regression (GPR) to build a behavioral model. The data is obtained through SPICE simulations, and, in python, it is scaled. To optimize the train process, it is performed using the GPU and CPU in parallel. A VerilogA script is obtained through the API. The authors also report that the TensorFlow model takes 1.5 seconds to run, while at transistor level, it takes 0.1 seconds to run. The RMSE reported in the resistive load is 1.04% of the maximal current. As a conclusion and analysis of the results, the authors claim that the error is due to the model deficiencies and not the API, and the API allows fast prototyping and easy debugging and allows different area's expertise to work together in interdisciplinary collaboration.

2.6 Conclusions

Modeling circuits behavior is a task that ANNs can perform successfully, and it brings to the analog IC design automation many advantages: simulations are faster, less computational complexity is required, and the models can be generalized. Apart from ANNs, SVMs have also been used for behavioral modeling. Still, due to the reasons detailed in section 2.2, ANN is the method chosen to use in this work.

The major drawback of the previously mentioned works is the lack of generalization when applying the created model to another circuit sizing. That is what the work can give capabilities to the models to distinguish different sizes of the devices and associate them to the respective input and output. The work also implements a generator script to convert ANN from Python to Verilog-A code which is also an innovation. To perform extensive circuit analysis and integrate the model into complex circuits is important to convert the model to a hardware language such as Verilog-A. This work also innovates since it brings new generator scripts capable of generating Verilog-A code from Python ANN code.

To summarize, Table 2.2 presents a summary of techniques to implement ANN in Verilog-A as well as scripts to facilitate the conversion of ANN from Python to Verilog-A. Table 2.3 presents a comparative summary of the state-of-the-art, explained in detail in chapter 2.

Table 2.2 Neural Networks in Verilog-A Summary

Reference	Neural Network Type	VHD Language
[31]	Backpropagation	Verilog-A
[32]	RNN	Verilog-A
[33]	API (TensorFlow to Verilog-A)	Verilog-A

Table 2.3 - Modeling of Analog/RF Devices Summary

Ref.	Device/Circuit	Method(s)	Objective	HDL	Error (circuit vs model)	Time (time reduction)	Data (simulator / dataset [in], [out])
[4]	RF- microwave components and MESFET	ANN (several)	Review of ANN based CAD for microwave designs.	-	-	-	Spectre / $[v_{gs}, V_{ds}, f]$, [Drain Current]
[8]	Three-stage cross-coupled charge pump	ANN (MLP)	Model a charge Pump circuits	Verilog-AMS	4.85%	7s	Spice / $[v_{dd}, R_{load}, f_{clk}]$, [Output Voltage]
[11]	Low Relaxation Oscillator	ANN (TDNN)	Model to analyze the Power consumption	SystemVerilog	2.70%	2min	Spectre / $[f, clk, EN]$, [Supply Current]
[12]	Analog-n/d	ANN (Bprop)	Modeling the power consumption.	Verilog-XL	1.53%	-	- / $[f, V_{out}]$, [Power]
[13]	RF Receiver, Sigma Delta, Low-Dropout Regulator	ANN (RNN) + Genetic Algorithm	Surrogate Model with Collaborative Stimulus	Verilog-A	RMSE: $31\mu V$	30* Faster	- / [Stimulus], [Adapted Stimulus]
[14]	CMOS band-gap voltage reference circuit (BGR)	CompNN (NARX + TDNN)	Capture the dynamic of analog MIMO circuits	Verilog-A	5.00%	17* Faster	- / [Trimming, Load Jump, Line Jump], [1V-Out]
[16]	Inverter Ring Oscillator	ANN (FFNN)	Capture the periodicity of oscillator output waveform	Verilog-A	-	10* Faster	Spectre / [Time Step, f_{osc}], [Output Voltage]
[17]	Ring Oscillator with a Buffer	ANN (FFNN + RNN)	Capture the periodicity of the oscillator + buffer output waveform	Verilog-A	RMSE: 0.0034	10* Faster	Spectre / [Time Step, f_{osc}], [Output Voltage]
[18]	Transistor-Level VCO Circuit	ANN (2 FFNN + RNN)	Model the oscillatory frequency and see the effects on the output waveform.	Verilog-A	RMSE: 0.0061	10* Faster	Spectre / [Time Step, $f_{osc}, V_{controller}$], [Output Voltage]
[19]	RF-microwave components,HMT and MESFETs	ANN (several)	Review of model development and nonlinear modeling of microwave devices	-	-	-	-
[20]	Coplanar Waveguide Circuits	ANN (EM based)	Efficient modeling of CPW components for accurate performance estimations	-	0.16%	-	- / $[f, W, G]$, [Line Impedance]
[21]	UC-PGB rectangular waveguide, patch antenna with PGB substrate	ANN (MLP and RBF)	Efficient modeling of RF devices for nonlinear microwave applications.	-	MSE: $2*10^{-4}$	-	- / [FET Length, FET Width], [Drain Current]
[22]	SCFL Buffer Cell, Resistive Mixer, Voltage Controller Oscillator	SVM ($\varepsilon - SV\ regression$)	Robust modeling of GaAs transistors and circuits.	-	MSE: $3*10^{-7}$	-	- / [V(n), I(n-k)], [Output Current]
[23]	Analog Circuits- CMOS	SVM	Efficient active learning scheme for feasible design space selection	-	-	-	Spice / [Heigth, Width]

Chapter 3

Implementation

The work implements two models to capture the behavior of an amplifier. As an innovation, this work takes into consideration, for the first time in the literature, a single LSTM model that can be reused among several circuit sizings. The second model implemented is an MLP with two delay lines, which not only is another model capable of modeling the behavior of a circuit. An amplifier, the Operational Transconductance Amplifier (OTA) is used as the working example in this thesis, and the finalized model is converted to Verilog-A using an automatic generator script making possible to accelerate the design optimization of an analog front-end circuit for biomedical applications [34].

3.1 Data Acquisition

As with any deep learning model, data is a relevant part of it, allowing us to make accurate models. It is also extremely important to extract a good set of data because if a model receives a poor dataset, it gives a poor performance, no matter how complex it is. For this work, the samples, which were obtained from the Spectre Simulator, are from the operational transconductance amplifier (OTA) present in Figure 3.1. The differential input has the positive voltage and negative voltage, and as outputs, it has the positive voltage output and negative voltage output.

For each circuit size is used nine datasets, with two input waves and two output waves, the first four datasets, the amplitude varies along the time and in the last 5 datasets the frequency changes keeping the amplitude constant. Table 3.1 presents a sample of the dataset of the waves extracted from the Spectre simulation of the circuit. For these models, it was considered 93 different sizing examples for the OTA. These sizing examples were obtained from the multiobjective optimization of the complete analog front end using AIDA [43].

In Figure 3.2 is shown the schematic of the circuit whose behavior will be mimicked using, firstly, an LSTM model and, secondly, an MLP with two delay lines as it will be further explained.

Table 3.1 - Wave Dataset

V_{ip}	V_{op}	Out +	Out –
2.50000e-01	2.50000e-01	8.70962e-02	8.70962e-02
2.50007e-01	2.49993e-01	8.65948e-02	8.72738e-02
2.50010e-01	2.49990e-01	8.68060e-02	8.77894e-02
2.50429e-01	2.49571e-01	7.45603e-02	9.90171e-02
2.49505e-01	2.50495e-01	1.00598e-01	7.28204e-02

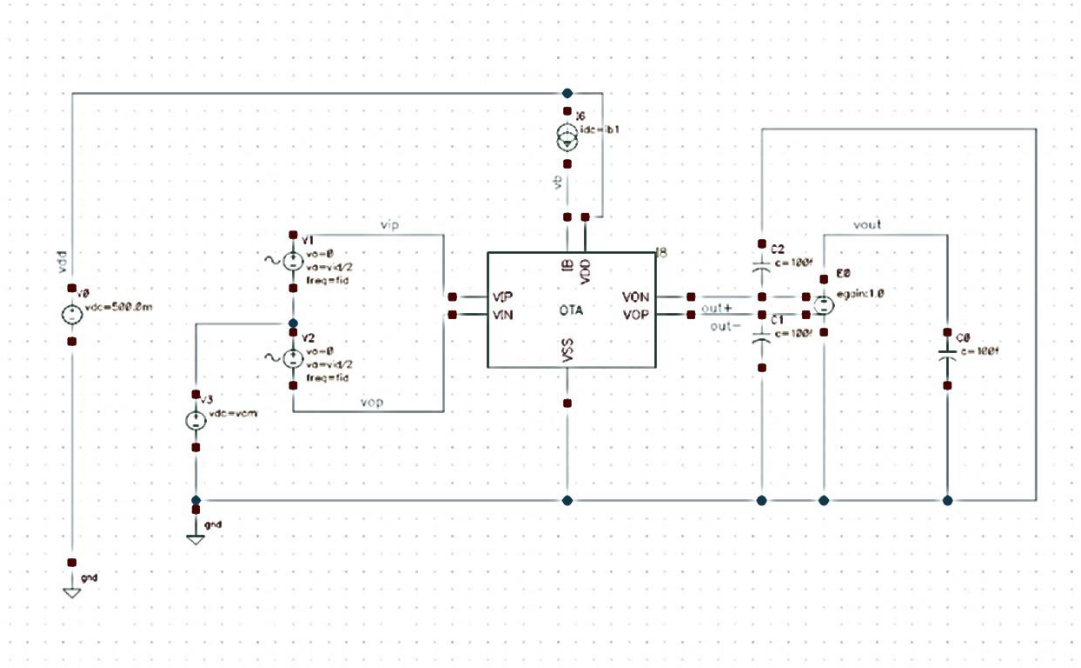

Figure 3.1 – Test Bench

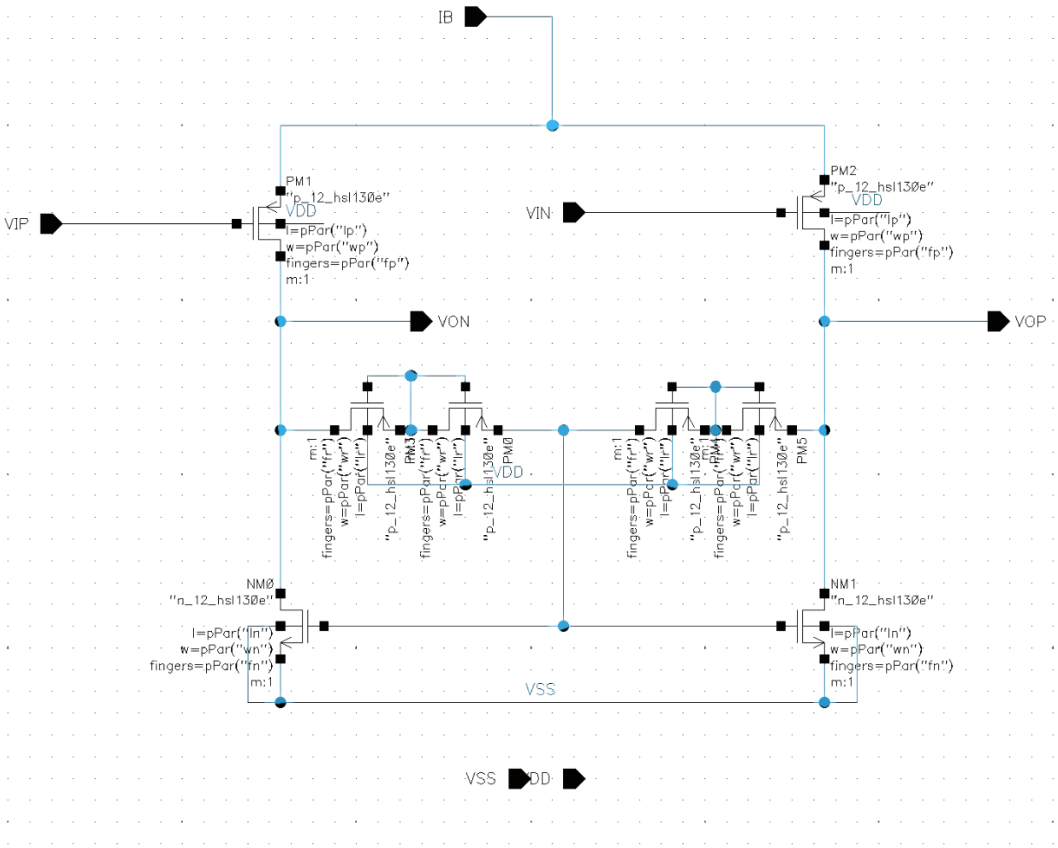

Figure 3.2 - OTA

3.2 Behavioral Model Using LSTM

In the first approach, DL is used to create a model capable of mimicking the behavior of the OTA for different circuit sizes. For that purpose, is used the programing language Python and the open-source machine learning framework PyTorch [35]. In the following sub-section are explained all the steps taken to achieve the desired behavior model using an LSTM [36], as well as a summary table that wraps up all the layers and all the decisions needed to create an ANN model.

3.2.1 Data Pre-Processing

To build a proper dataset for this work, for each circuit sizing, the nine datasets, containing the waves, are opened, split into features and labels. In order to keep the temporality of the data, since this problem deals with simulations over time, for each dataset, a sliding window (code snippet in Figure 3.3) is applied to obtain tuples of features and the corresponding label. At this point, the nine pre-processed datasets features are concatenated row-wise as well as the labels, obtaining the X dataset and the Y dataset, respectively. The size of the devices is taken into consideration, so to all the sizes of the devices a polynomial feature is applied, of order 2, to extract more relevant relations of the circuit sizes and to have more features turning the learning phase more robust. Those new features are concatenated column-wise with the X dataset.

```
# Sliding Window Function
# input: features, labels, sequence length
# outputs: sets of features and labels sequentialy

def sliding_windows(data1, data2, seq_length):
    x = []
    y = []

    for i in range(len(data1)-seq_length-1):
        _x = data1[i:(i+seq_length)] # input: the seq_length of data
        _y = data2[i+seq_length] #Output: the next value after the input
        x.append(_x)
        y.append(_y)

    return np.array(x),np.array(y)

# this function take as arguments the data and a certain sequence length and append to an x an y variable the respectiv
# previous data and the value that needs to be predicted
```

Figure 3.3 – Sliding Window Function

Figure 3.4 presents the steps taken to pre-process the data for a single sizing solution. After doing the pre-processing for all nine datasets of the 93 circuit sizes, all the datasets are concatenated, turning into a large feature dataset containing the waves and the polynomial features of all the 93 sizing solutions. The labeled dataset also contains the labels for the 93 circuit sizes. At this moment, the features and labels datasets are scaled, using the MinMaxScaler, to make each feature and label vary in a given range.

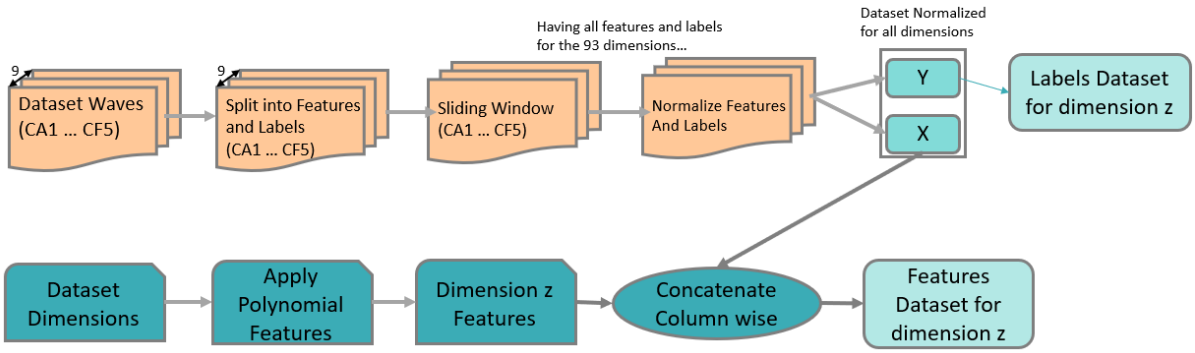

Figure 3.4 - Data Pre-Processing for one dimension

3.2.2 LSTM Model Structure

Having the data well curated and pre-processed, it is time to develop a DL model capable of learning the data. As it was first proposed, a LSTM neural network, an upgrade of the RNN, is applied to mimic the amplifier's behavior. For a better understanding of the model, it is relevant not only to explain the LSTM function but also to enlighten the typologies and states of the LSTMs. First, the LSTM is chosen, to the detriment of the RNN, since the main disadvantage of the RNN model is that it only “remembers” the previous output and for this work is essential, since the previous samples of the waveforms influence the next ones, to keep track of longer outputs history. Figure 3.5 is enlightened the main blocks of an LSTM network.

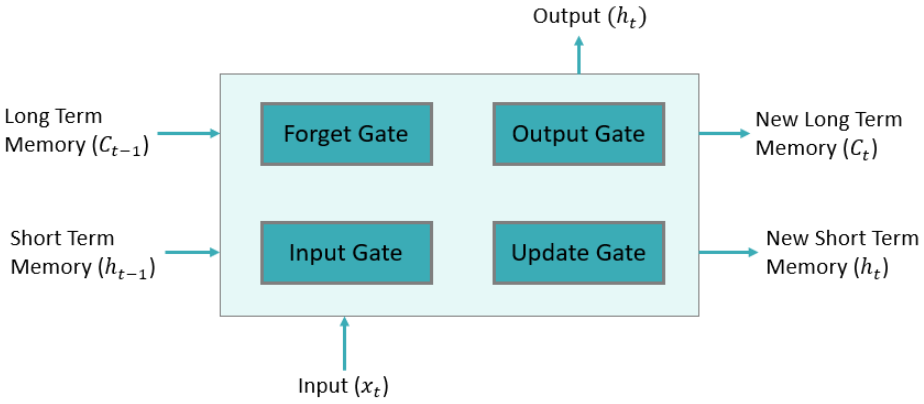

Figure 3.5 - LSTM Neural Network Model

Figure 3.6 illustrates the interior of an LSTM model, which is mathematically formulated in (26). This mathematics are already implemented in the PyTorch LSTM Layer [37], which is used in this work.

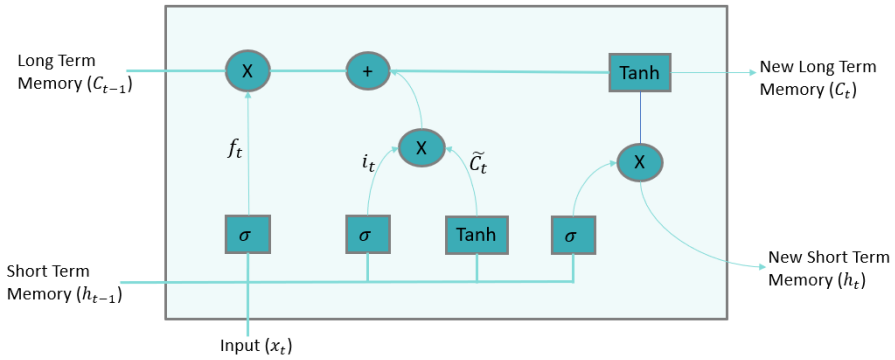

Figure 3.6 - LSTM Interior

$$
\begin{aligned}
h_t &= \sigma(w_o[h_{t-1}, x_t] + b_o) * \tanh(C_t) \\
C_t &= f_t * C_{t-1} + \tilde{C}_t * i_t \\
f_t &= \sigma\left(w_f[h_{t-1}, x_t] + b_f\right) \\
i_t &= \sigma(w_i[h_{t-1}, x_t] + b_i) \\
\tilde{C}_t &= tanh(w_c[h_{t-1}, x_t] + b_c)
\end{aligned}
\tag{26}
$$

It is also important to notice that LSTM can avoid vanishing gradients, which is an important upgrade when compared to the RNN. However, they can suffer from exploding gradients, so it is important to take this fact into consideration when training the model. To solve this issue, gradient clipping is an important approach, and its usage is explained further in the training phase of the model.

Relatively to the topologies, an LSTM can be Sequence to Sequence, where given n samples it predicts the next n samples, or Vector to Sequence, where given 1 sample it predicts the next n samples or Sequence to Vector, where given n samples it predicts a unique output. This last topology is chosen for this work.

Relatively to the states, an LSTM can be Stateless, where after each sequence prediction, the hidden and cell states are initialized to zero, or Stateful, where after each sequence prediction, the hidden and cell states keep their values, which serve as the hidden and cell states for the following sequence. This last approach is chosen since, in this work, the following sequence is related to the previous one.

The LSTM model contains 1 LSTM layer and 2 fully connected layers. In the initialization method of the LSTM model class, the layers are initialized, and the hidden and cell state is initialized to zero. In the forward path, the feature dataset is split into waves and circuit sizes, and the waves are the input of the LSTM layer. The LSTM output is reshaped and concatenated with the circuit size column-wise. This concatenation is the input of a fully connected layer with ELU as its activation function, which is needed since it is important to add non-linearities for better learning of the sizes. After this layer, a dropout layer with a probability of drop of 0.1 is added to prevent overfitting. Then the output of the first activated fully connected layer is the

input of the second fully connected layer, which has a linear activation function since the problem is a regression one. Figure 3.7 summarizes the model structure presenting the total number of trainable parameters extracted from the PyTorch framework.

```
==========================================================
Layer (type:depth-idx)                   Param #
==========================================================
DataParallel                             --
├─LSTM: 1-1                              --
│    └─LSTM: 2-1                         817,200
│    └─Linear: 2-2                       119,750
│    └─ELU: 2-3                          --
│    └─Dropout: 2-4                      --
│    └─Linear: 2-5                       502
==========================================================
Total params: 937,452
Trainable params: 937,452
Non-trainable params: 0
==========================================================
```

Figure 3.7 - LSTM Model Structure

3.2.3 LSTM Model Training Phase

To train the model, making it capable of mimicking the behavior of the amplifier, a set of considerations regarding the network hyperparameters was done:

Loss Function: to find the best loss function, both MSE and mean absolute error, which are the most appropriate losses for regression problems, are considered, and the model is trained and tested with both. However, the MSE performs best, so it was the chosen loss function.

Optimizer: to choose the best optimizer, Stochastic Gradient Descent, Adam and AdamW were tested, and Adam gave the model a better performance. To prevent overfitting AdamW was used, since it applies weight decay. AdamW is a variant of the optimizer Adam that has an improved implementation of weight decay [38]. For the AdamW optimizer, the default values of the parameters were kept, $\beta_1 = 0.9,\ \beta_2 = 0.999$ and weight decay = 0.01.

Learning Rate: to train the model was used an adaptative learning rate [39]. The starting learning rate is $5 * 10^{-5}$ and along the training phase, if the loss increases in two consecutive epochs, the actual learning rate is multiplied by a factor of 0.7, reducing the learning rate in order to make the loss keep decreasing over the epochs.

Batches: since the dataset is big, it is necessary to use a data loader (which has a PyTorch framework [40]) to create batches and feed the network with batches instead of the full data. The batch size is around 10000, which is 1% of all data size, lower than the 10% threshold to be considered a big batch size. The drawback of using such a low value for the batch size is that the training times get higher. The data and the batches are not shuffled to keep the temporality and the order of the sequences due to what was explained in the pre-processing data section.

Gradient Clipping: in order to avoid the exploding gradients, gradient clipping with a maximum norm of 0.25, is introduced in the training phase. If the gradient gets too large, it is rescaled to keep it small. More precisely if ‖g‖ ≥ 0.25 then the gradient is $0.25 * \frac{g}{\|g\|}$. The maximum norm value is found through trial and error, searching for the best value possible.

Having those important parameters chosen, the train is made by giving to the model the features, and it predicts a label, compares it with the true label, and computes the loss, then the loss is backpropagated through the network, and the weights are updated. This step is performed along all the epochs. It is also worth to mention that the data is feed to the model through batches due to the large amount of data and in this way is possible to make the train phase faster and with accuracy, avoiding memory issues.

3.2.4 Circuit Noise

As expected, the model trained is immune to the circuit noise since the ANN for each input pair generates a unique prediction, so it is up to the designer to find ways to learn the noise and introduce it in the predicted wave. For that purpose, in the training phase, the error between the predictions and the label is computed, then the mean and standard deviation of that error are computed. Having those two parameters, it is possible to create a gaussian noise that is added to the prediction. In this case, since the error is computed as (27) and gaussian noise is given by (28), the prediction of the model with noise is given by (29).

$$Error = Predictions - True\ Labels \tag{27}$$

$$Noise = Gaussian(Error.mean, Error.standard\_deviation) \tag{28}$$

$$Noisy\ Predictions = Prediction - Noise \tag{29}$$

In this way, it is possible to mimic the amplifier behavior for the different device sizes even with more realistic detail. Another suitable approach could be using Variational Auto-Encoders to learn the noise from the input data. It was not considered in this work, as detailed noise characterization and modeling were out of the scope of this work, but it is worth exploring in future work.

3.2.5 Implemented LSTM Model Summary

In this subchapter, a detailed description of the LSTM model implementation was given. In the data acquisition section was explained how the data is obtained through the Spectre Simulations. In order to have a proper data for the model, the pre-processing phase was described. After that, since the model uses LSTM, a brief explanation of the LSTM inside work was explained and so it's advantages regarding to RNN and its drawbacks and how to avoid them. Having the background about LSTM the model implemented was declared and all the choices were explained. The training phase was introduced, giving in detail some hyperparameters tuning and its choices for this model. Finally, since the ANN is not capable to reproduce the noise in the data, it is explained how it was done using the gaussian noise.

In Table 3.2 is presented the model summary for a clear understanding of what hyperparameters are implemented in the model, as well as their values, the number of layers used, and the number of neurons used in each layer. It is important to mention that this combination was obtained after several trials and might exist other combinations that also provide good results.

Table 3.2 - Model Summary

Hyperparameter	Value Used
LSTM Layer	1 Layer (6 nodes)
Hidden Layers	2 Layers (450, 250 nodes)
Output Layer	1 Layer (2 nodes)
Activation Function	ELU – Hidden Layers / Linear – Output Layer
Optimizer	AdamW (learning rate = $5 * 10^{-5}$, $\beta_1 = 0.9\ and\ \beta_2 = 0.999$.)
Learning Rate Scheduler	Factor = 0.7, patience = 2
Regularizer	Dropout (probability of 0.1), Weight Decay (0.01)
Loss function	MSE
Batch Size	10000 (1% of total data size)
Gradient Clipping	0.25

In the Figure 3.8 is presented the model scheme, where in the blue box is represented the LSTM flow, in the orange box the Linear Layers flow and the relation with the device's sizes, and in the green box is represented the noise added to the predictions in order to create an accurate mimic of the circuit behavior.

Note that *FC* stands for Fully Connected Layer, *h_i* stands for hidden state and the LSTM box illustrates the Sequence to Vector topology applied in this work where the input is a 3D array, in the form of [batch size, sequence length, input features], and the input feeds the network with a sequence and the output is only considered the last vector.

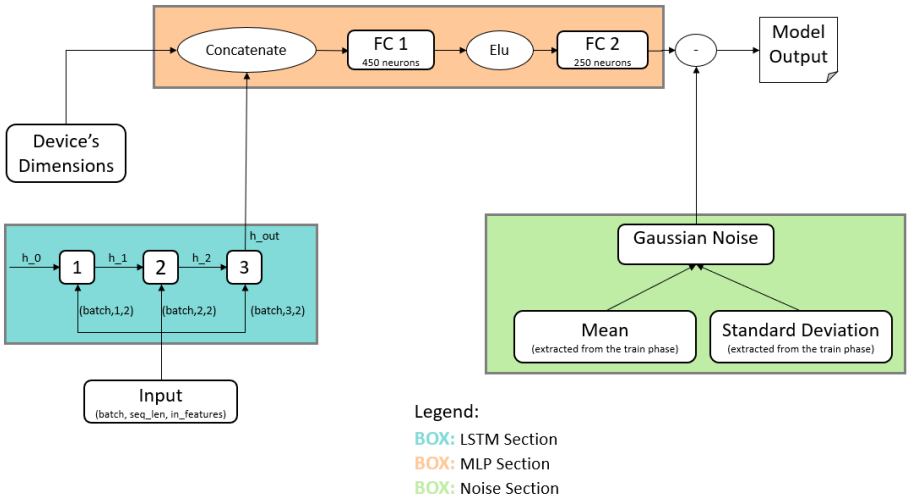

Figure 3.8 - Model Scheme

The LSTM model is a strong basis to be applied to other amplifiers, inside the same topology, providing accurate results.

3.3 Behavioral Model Using MLP with Delay Line

In order to have another model capable to mimic the behavior of the circuit, a MLP with Delay Line is created. In the following subsections an overview about the MLP implementation is detailed. The data used to train this model is the one used to train the LSTM model.

3.3.1 Data Pre-Processing

To build a proper dataset for this work, for each dimension, the nine datasets, containing the waves, are opened, splitted into features and labels. In order to keep the temporality of the data, since this problem deals with simulations over the time, instead of using a sliding window (as it was used in the LSTM model) is applied, to the original waves dataset, two delay lines, one with 1 delay and other with 2 delays (Figure 3.9) which are concatenated with the original waves dataset, column-wise. At this point, the nine pre-processed datasets features are concatenated row-wise as well as the labels for all 93 circuit sizes, obtaining the X dataset, which contains 6 columns with the original waves and the shifted ones, and the Y dataset, respectively. The size of the devices is taken in consideration in the same way they were considered in the LSTM model, so it is applied, to all the sizes of the devices, a polynomial feature, of order 2, to extract more relevant relations of the circuit sizes and to have more features to make easier the learning phase. Those new features are concatenated column-wise with the X dataset.

```
# One delay line with shift of 1
df_x_s1 = pd.DataFrame(a[:,0:2])
df_x_s1 = df_x_s1.shift(1)

# Second delay line with shift of 2
df_x_s2 = pd.DataFrame(a[:,0:2])
df_x_s2 = df_x_s2.shift(2)
```

Figure 3.9 – Delay Lines in Data

Figure 3.10 presents the steps taken to pre-process the data for a single dimension. After doing the pre-processing for all nine datasets of the 93 circuit sizes, all the datasets are concatenated turning into a big feature dataset containing the waves and the polynomial features of the device sizes for the 93 sizes. The label dataset also contains the labels for the 93 circuit sizes. At this moment the features and labels datasets are scaled, using the MinMaxScaler, to make each feature and label vary in a given range.

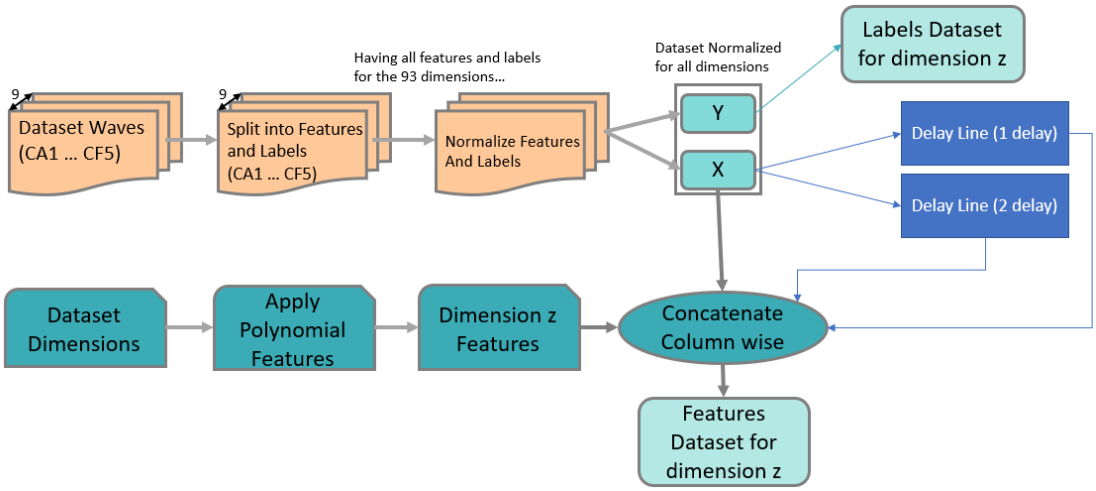

Figure 3.10 - Data Pre-Processing for one dimension with delay lines

3.3.2 Deep Learning MLP Model Structure

Having the data well curated and pre-processed, it is time to develop a deep learning model, capable to learn the data. For this purpose, the MLP model contains 5 fully connected layers. In the forward path the 6 input features pass through the first fully connected layer, which has ELU as its activation function, since it is important to add non-linearities for the better learning of the circuit sizes. After this layer, a dropout layer with probability of drop of 0.1 is added to prevent overfitting. Then the device's sizes are concatenated to the output of the first fully connected layer and flows through a second fully connected layer with activation function ELU. The output of this layer passes through another equal layer, the third fully connected layer, however the number of neurons is different, as it is possible to see in the following sections, and the activation function is the Sigmoid. The output of the third fully connected layer is the input of the fourth fully connected layer which has a ELU activation function, finally it passes through the fifth fully connected layer with linear activation function. In Figure 3.11 is possible to see the model structure in the PyTorch framework.

```
==========================================================
Layer (type:depth-idx)                   Param #
==========================================================
DataParallel                             --
├─Model: 1-1                             --
│    └─Linear: 2-1                       1,400
│    └─Dropout: 2-2                      --
│    └─Linear: 2-3                       22,900
│    └─Linear: 2-4                       10,100
│    └─Linear: 2-5                       2,525
│    └─Linear: 2-6                       52
==========================================================
Total params: 36,977
Trainable params: 36,977
Non-trainable params: 0
==========================================================
```

Figure 3.11 - MLP Model Structure

3.3.3 Deep Learning MLP Model Training Phase

To train the model, making it capable to mimic the behavior of the amplifier, a set of considerations regarding to the network hyperparameters was done:

Loss Function: is used the same as in the LSTM model, the MSE.

Optimizer: is used the same as in the LSTM model, the AdamW with the same parameter's configuration.

Learning Rate: the used learning rate is $1 * 10^{-4}$. For this model the adaptative learning rate was not used.

Batches: as in the LSTM model, the batch size used is around 10000 which is 1% of all data size which is lower than the 10% threshold to be considered a big batch size.

Gradient Clipping: in order to avoid the exploding gradients is introduced in the training phase, a gradient clipping with a maximum norm of 1. If the gradient gets too large, it is rescaled to keep it small. More precisely if $\|g\| \geq 1$ then the gradient is $1 * \frac{g}{\|g\|}$.

Having those important parameters chosen, the train is made by giving to the model the features, and it predicts a label, compares it with the true label, and computes the loss, then the loss is backpropagated through the network, and the weights are updated. This step is performed along all the epochs. To add noise in the prediction the strategy used is the same that was used and explained in the LSTM model.

3.3.4 Implemented MLP Model Summary

In this sub chapter, a detailed description of the implementation work was given. In data acquisition section was explained how the data is obtained through the Spectre Simulations. In order to have proper data for the model the pre-processing phase was described, including the delay line construction. After that, the MLP model was implemented according to all the explained parameters. The training phase was explained,

giving in detail some hyperparameters tuning and its choices for this model. Finally, since the ANN is not capable to reproduce the noise in the data, it is explained how it was done using the gaussian noise.

In Table 3.3 is presented the model summary for a clear understanding of what hyperparameters are implemented in the model, as well as their values, the number of layers used and the number of neurons used in each layer. This combination is obtained applying the same criteria used in the LSTM model.

Table 3.3 – MLP Model Summary

Hyperparameter	Value Used
Input Layer	1 Layer (6 nodes)
Hidden Layers	3 Layers (200, 100, 25 nodes)
Output Layer	1 Layer (2 nodes)
Activation Function	ELU & Sigmoid – Hidden Layers Linear – Output Layer
Optimizer	AdamW (learning rate = $5 * 10^{-5}$, $\beta_1 = 0.9\ and\ \beta_2 = 0.999$.)
Regularizer	Dropout (probability of 0.1), Weight Decay (0.01)
Loss function	MSE
Batch Size	10000 (1% of total data size)
Gradient Clipping	1

In Figure 3.12 is presented the model scheme, where in the blue box is represented the two Delay Line flow, in the orange box the MLP Linear Layers flow and the relation with the device's sizes and in the green box is represented the noise added to the predictions in order to create an accurate mimic of the circuit behavior. Note that *FC* stands for Fully Connected Layer.

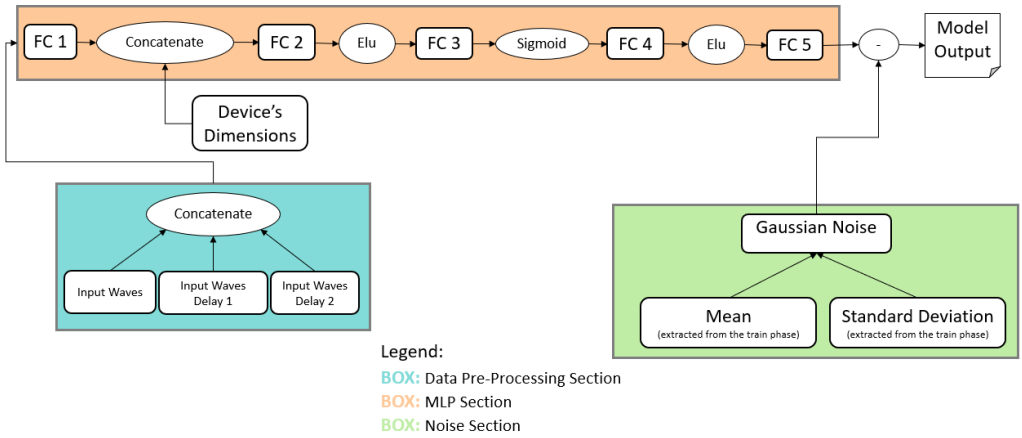

Figure 3.12 – MLP Model Scheme

The Deep Learning model is a strong basis to be applied to other amplifiers, inside the same topology, and it can be converted to Verilog-A to accelerate the design optimization of an analog front-end circuits.

At the end of the results chapter is presented a comparison of the results between the LSTM model and the MLP model as well as important facts for the difference between the models.

In the next section, a detailed overview of the Verilog-A programming environment is presented as well as the MLP model implementation, already implemented in Python/PyTorch, in Verilog-A language is explained with detail.

3.4 Conversion of the model to Verilog-A

In order for the model accelerate the design optimizations of analog front-end circuits is essential to convert the model to Verilog-A. Despite being a particular language directed to circuits, it is important to implement the MLP model in this language to prove that the simulation time is accelerated, and the behavior produced by the model is accurate. To better understand the Verilog-A implementation and how the generator script was created, a dummy example is presented alongside the necessary explanations.

3.4.1 Verilog in General

To convert the model to Verilog-A is relevant to explain the structure of the Verilog-A models. In this work, the Verilog-A is provided by Cadence Design Systems. The interface part, the connectivity of the components, and the behavioral description are defined as *modules.* Figure 3.13 represents the scheme of the Verilog-A general structure, which serves as the basis for implementing the neural network layers in HDL.

```
// VerilogA General Module

`include "constants.vams"
`include "disciplines.vams"

module Verilog_Module (in, out);

    // Directional Specifications
    input xxx;
    output yyy;

    // Descipline Definition
    electrical xxx, yyy;

    // Parameter Declaration
    parameter real p_xxx = 1;
    parameter real p_yyy = 1;

    // Variable Declaration
    real var

    analog begin

        // Behavioral Statements - to reproduce the behavior of the model

    end

endmodule
```

Figure 3.13 - Verilog-A Code Block

A module definition is delimited by the words *module* and the *endmodule.* The discipline definition, in this case *electrical,* has two natures, Voltage and Current, so the analog block defines the input and output

variables as Voltage or Current. After that, the parameters and variables are declared. A parameter differs from a variable because the first one has a fixed value, and the second one can have its value changed. Note that the variables can only be real or integer. The *analog block* implements the mathematical relationships between the input and output signals [41]. For each ANN layer is necessary to create a code block to define the parameters and the forward path of the layer.

In Verilog-A, it is only necessary to define the forward path of each layer since the model have already been trained in Python, so in Verilog-A, it is only needed the trained weights, which in this case, a generator script is made, in Python language, to extract those weights in the vector Verilog-A nomenclature ({w1, w2, w3,…, wn}). The bias is extracted in the same way as the weights. Once the *module* is created a block with the input's wires, outputs wires and behavior statements of the *module* is generated. Having all the modules needed for the ANN, a structural top level (schematic or Verilog) is created, the blocks are connected using wires and the simulation is performed.

3.4.2 Fully Connected Layers in Verilog-A

First, is necessary to declare two integer parameters to store the input and output sizes. Second, real-valued parameters of vector type are declared to store the weights and biases. As required by the Verilog-A the *module* input parameters are defined as input and output and the discipline is also declared, which in this case is *voltage*. In the *analog begin* block is defined the forward path of a fully connected layer.

Since Verilog-A does not accept matrices, the weights matrices are divided into vectors, and the forward path is performed using ANN logic and an accumulator.

$$a[i] = \sum_{j=1}^{n} V(x[j]) * W_j + a[i] \tag{30}$$

The equation (30) represents the forward path, where *i* is the neuron number which varies between 1 and the total number of neurons in the output layer of that fully connected layer. The *x* represents the input values and *w* the weights vectors of the input layer of that fully connected layer. Note that since the accumulator is used, it is necessary to initialize the *a* variable to zero before re-entre the loop again. After applying the equation (30) it is time to address to the output variable the correspondent *a* value. The equation (31) presents the loop cycle to address this matter.

$$\sum_{i=1}^{n} V(y[i]) = \sum_{i=1}^{n} a[i] + b[i] \tag{31}$$

In Figure 3.14 is present an example of the Verilog-A code for the MLP model, obtained through the automatic generator script. It is possible to see the weights vectors, bias declaration and the forward path, which follows the equation (30), the weights update which follows the equation (31).

```
module ANN_Middle_Layer_FC1(x,y);

  parameter integer x_dim = 1;
  parameter integer y_dim = 1;

  // Weights vectors declaration
  parameter real W20[0:2] = {-0.160902,0.267084,0.498560};
  parameter real W21[0:2] = {-0.326666,0.002737,-0.572759};

  // Bias vectors declaration
  parameter real W22[0:2] = {0.360297,0.096587,-0.260858};
  parameter real b[0:2] = {-0.552001,-1.896233,1.313081};
  real a2[0:2];

  // Input and Output Declaration
  input [0:2]x;
  voltage [0:2]x;
  output [0:2]y;
  voltage [0:2]y;
  genvar i;

  analog begin
      a2[0] = 0;
      a2[1] = 0;
      a2[2] = 0;

      //Forward Path
      a2[0] = V(x[0]) * W20[0] + a2[0];
      a2[0] = V(x[1]) * W21[0] + a2[0];
      a2[0] = V(x[2]) * W22[0] + a2[0];
      a2[1] = V(x[0]) * W20[1] + a2[1];
      a2[1] = V(x[1]) * W21[1] + a2[1];
      a2[1] = V(x[2]) * W22[1] + a2[1];
      a2[2] = V(x[0]) * W20[2] + a2[2];
      a2[2] = V(x[1]) * W21[2] + a2[2];
      a2[2] = V(x[2]) * W22[2] + a2[2];

      // Weights Updates
      for(i = 0; i<= 2; i = i + 1) begin
          V(y[i]) <+ a2[i] + b[i];
      end
  end
endmodule
```

Figure 3.14 - MLP Layer in VerilogA

3.4.3 Activation Functions ELU and Sigmoid in Verilog-A

To add non-linearities to the model, the ELU activation function is created. The *module* input parameters are declared as input and outputs, and the discipline as electrical. In the *analog begin* block the mechanism of the ELU is created according to the equation (32).

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha.(e^{-x} - 1), & x \leq 0 \end{cases} \tag{32}$$

In Figure 3.15 is possible to see the Verilog-A code implemented to compute the ELU activation function.

```
module ELU(n_deact, n_act);

  input n_deact;
  output n_act;
  electrical n_deact, n_act;
  real result;
  real p_num;
  parameter real alpha = 1;

  analog begin
      p_num = V(n_deact);
      if(p_num > 0)
          result = p_num;
      else
          result = alpha * (exp(p_num) - 1);
      V(n_act) <+ result;
  end

endmodule
```

Figure 3.15 - ELU in Verilog-A Language

The same strategy is used to create the Sigmoid activation function, however its construction follows the expression (33).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{33}$$

In Figure 3.16 is possible to see the Verilog-A code implemented to compute the Sigmoid activation function.

```
module Sigmoid(n_deact, n_act);

  input n_deact;
  output n_act;
  electrical n_deact, n_act;
  real result;
  real p_num;
  parameter real alpha = 1;

  analog begin
    p_num = V(n_deact);
    result = 1/(1+exp(-p_num));
    V(n_act) <+ result;
  end
endmodule
```

Figure 3.16 - Sigmoid in Verilog-A Language

3.4.4 Circuit Sizes, Concatenation, Delay Lines, and Scaling in Verilog-A

For the model to recognize different circuit sizes is necessary to concatenate them and pass them through the fully connected layers with non-linear activation functions, so in Verilog-A a *module* is created, and it has 1 input which is the output of a fully connected layer or activation function and 1 output which is the input concatenated with a vector of circuit sizes.

In the Python data pre-processing was implemented two delay lines in the dataset, so it is also necessary to do the same in VerilogA, so a *module* with 1 input (the original wave) and 2 outputs (the original wave and the delayed one) is created to implement the delay lines. First, a time delay is chosen and then the

built in function *absdelay* is used having as input parameters the original wave and the time delay. In this way is possible to obtain waves that are delayed from the original one.

Since in ML is good practice to scale all the feature dataset and all label dataset, it is possible to obtain the minimum and maximum value of the dataset, which is used for scale and invert the scale of the data. So, in Verilog-A it is also implemented, for scaling, the MinMaxScale expression where the minimum and maximum value are the same used to scale the dataset of features in Python. To invert the scale, it is implemented the inverted expression of the MinMaxScale in Verilog-A using as minimum and maximum value the ones obtained when scaling the labels. The equation (34) presents the MinMaxScale formula to scale de data where d_{max} and d_{min} are the desirable range, which in this case is 1 and 0, respectively. (35) is the equation to revert the MinMax scaling.

$$X_{scaled} = \frac{(X - X_{\min})}{(X_{max} - X_{min})} * (\mathrm{d}_{max} - d_{min}) + d_{min} \tag{34}$$

$$X = \frac{X_{scaled} * (X_{max} - X_{min})}{(\mathrm{d}_{max} - d_{min}) + d_{min}} + X_{min} \tag{35}$$

Note that to facilitate further work in this research area, this three modules code are also obtained through the generator Python script since it need networks parameters which can be obtained directly through Python.

In Figure 3.17 is presented a sample of code to implement the scaling. It has the wave as input and the min and max value of the scaler are constant and obtain when scaling the data in data pre-processing. Figure 3.18 represents the Verilog-A code for the delay lines. Once again, those codes were obtained through the automatic generator script build to convert Python code to Verilog-A code.

```
module ScalerMMS(in_wave, out_wave);

  input in_wave;
  voltage in_wave;
  output out_wave;
  voltage out_wave;

  real parameter aux_for_min_set = 0.225;
  real parameter aux_for_max_set = 0.2749;
  real x_std;
  real x_scaled;

  analog begin

      x_std = (V(in_wave)-aux_for_min_set) / (aux_for_max_set - aux_for_min_set);

      x_scaled = x_std * (1 - 0);

      V(out_wave) <+ x_scaled;

  end
endmodule
```

Figure 3.17 - Scaler in Verilog-A Language

```
module DelayLines(in_wave, out_wave);

  input in_wave;
  voltage in_wave;
  output [0:2]out_wave;
  voltage [0:2]out_wave;
  real delay_wave;
  real a;
  real time_delay1;
  real time_delay2;
  real b[0:1];
  parameter real fixed_delay1= 5.00000e-05;
  parameter real fixed_delay2= 6.35141e-05;

  analog begin
      time_delay1 = fixed_delay1;
      time_delay2 = fixed_delay2;
      a = V(in_wave);

      b[0] = absdelay(a, time_delay1);
      b[1] = absdelay(a, time_delay2);

      V(out_wave[0]) <+ a;
      V(out_wave[1]) <+ b[0];
      V(out_wave[2]) <+ b[1];

  end
endmodule
```

Figure 3.18 - Delay Line Verilog-A Code

3.4.5 Generator Script From Python to Verilog A

Create a neural network in Verilog-A can be challenging. It requires to deal with Verilog A limitations such as the absence of matrices and lack of support libraries. So, using python to create a Verilog-A script is a step forward to automate the conversion of the Python code to Verilog-A code. Taking in consideration all the constraints and the fact that in the state-of the-art the presence of ANN in Verilog-A is rare, a Python Script was created in order to create various types of layers and activation functions in Verilog-A language. In that script all the layers are defined by a function (Figure 3.19) with input parameters necessary to build the layer (Weights matrix and Bias matrix), once the function is called with the right parameters it pre-processes automatically the weights to construct the weight vectors according to the Verilog A language and automatically generates the Verilog-A file for that layer with the right weights, variables and forward path.

```
def FC_HiddenLayer_to_VerilogA(W, B, input_dim, output_dim):
```

Figure 3.19 - Function Fully Connected Layer

In these scripts the weights matrix and bias matrix, that entre in the script function, need to be extracted as present in Figure 3.20:

```
l = model.fc1.state_dict()

W = l['weight'].detach().cpu().numpy()

B = l['bias'].detach().cpu().numpy()
```

Figure 3.20 - Weight and Bias Input Form PyTorch

Inside the function, the weights and bias matrix are decomposed into vectors to be further used in the forward path. For now, the generator Python Script can implement the fully connected layer and three activation functions, the ReLU, ELU and Sigmoid, in Verilog-A. This script is extensible to new layers, new activation functions and any parts needed for the ANN models (such as concatenations, delay lines) that need to be converted to Verilog A, it is just needed to follow the idea on this script, having in mind some knowledge about the Verilog A language and its structure.

3.4.6 Dummy Example of Simple MLP in Verilog-A

During the implementation of the thesis MLP model in Verilog-A, a dummy example was used with the purpose to see if the programming logic was right before using it in a large ANN. This dummy example also has the purpose to explain, with some detail, the implementation of a MLP in Verilog-A for a better understanding since this is a grey area with lack of support and state-of-art. So, in this dummy example, the feature is a simple vector ranging from 0 to 10 with a step of 0.1 and the label is the *sin* function of the feature values. Note that since it is a dummy example it does not have the purpose to build an accurate ANN but to explain and confirm if the conversion from Python to Verilog-A is well performed. The ANN used has 1 input and 1 output and 2 hidden layers with 3 neurons each and each hidden layer output passes through a ReLU activation function, Figure 3.21 shows the simple ANN build for this dummy example.

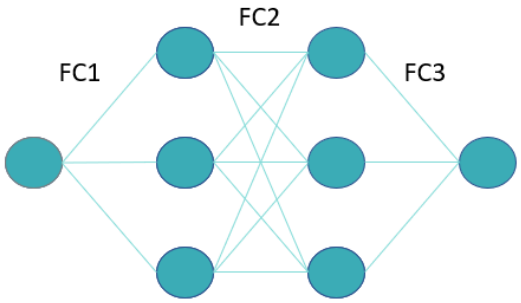

Figure 3.21 - Dummy ANN

So for this ANN is necessary to create 3 fully connected layers *modules* and one *module* for the ReLU activation function, one for the first fully connected layer (FC1) which has 1 input neuron and 3 neurons of output, one for the second fully connected layer (FC2) which has 3 input neurons and 3 output neurons and one for the third fully connected layer (FC3) which has 3 input neurons and 1 output neuron. Inside each module the forward path is applied according to equations (30) and (31). In Figure 3.22 is presented the code flow for the fully connected layers, which, in the script created to generate the VerilogA code from Python, is automatically prepared to, based on the input and output nodes, generate the Verilog-A script.

Figure 3.23 presents the code flow for a ReLU activation function in VerilogA which is also obtained through the generator script explained in the previous section.

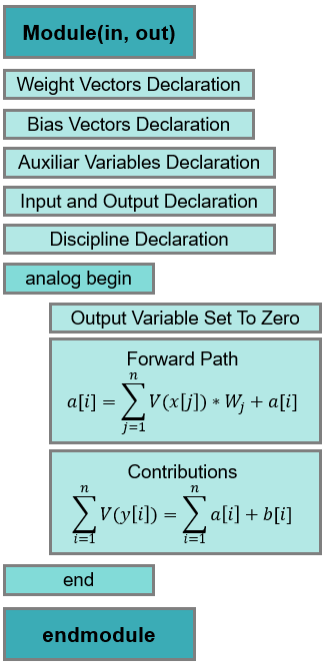

Figure 3.22 - Verilog-A FC Scheme

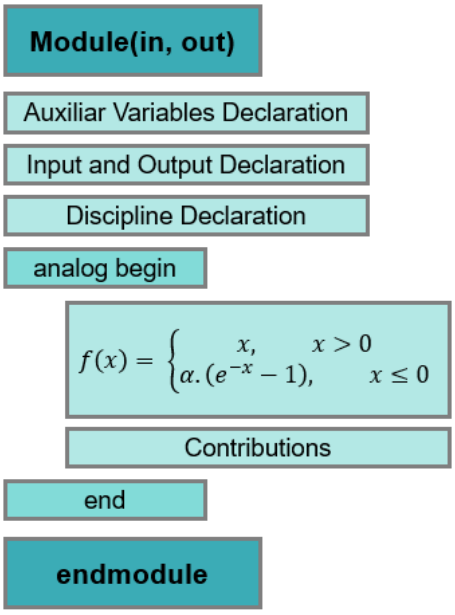

Figure 3.23 - Verilog-A ReLU Scheme

Once all the needed modules are created (for this example 3 fully connected and 1 ReLU) it is time to build the schematic in order to be able to perform the simulation of the circuit. So, in Cadance framework, using ADE Explorer the circuit is build linking the blocks (which inside have the *module* code) and specifying the number of wires, Figure 3.24 presents the circuit schematic for this dummy example.

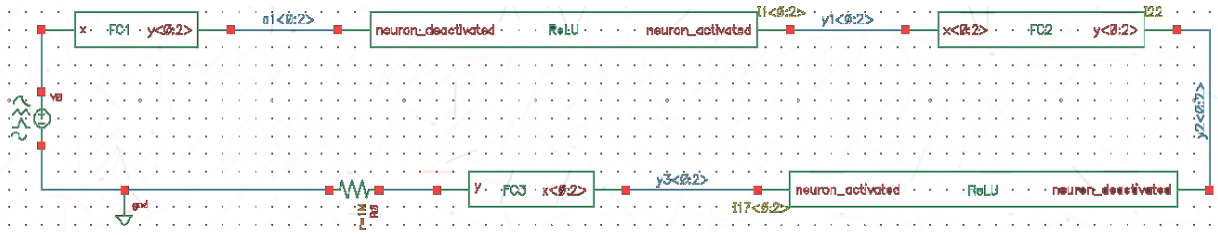

Figure 3.24 - Dummy Example Schematics

Running the simulator, the output of each fully connected layer is extracted to a *csv* file and compared with the Python output for the same layer output. Once again, this is a dummy problem so the accuracy of the network is not important and the similarity between the Python output and Verilog-A output is what most matters. It is also worth to mention that for the FC1, FC2 the output obtained are 3 curves, one for each neuron output in each layer. The output of the first fully connected layer (FC1) in Python and VerilogA is present in Figure 3.25.

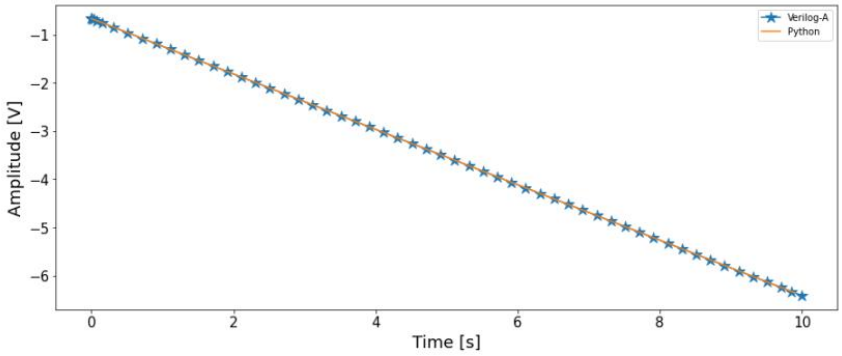

Figure 3.25 - Output FC1 Dummy Example

Figure 3.26 presents the output for the second fully connected layer (FC2) for Python and Verilog-A.

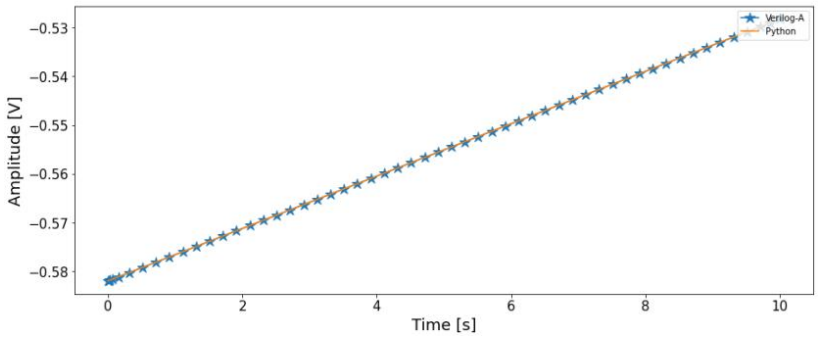

Figure 3.26 -Output FC2 Dummy Example

Finally, Figure 3.27 presents the results obtained for the third fully connected layer (FC3) which is also the output of this dummy neural network.

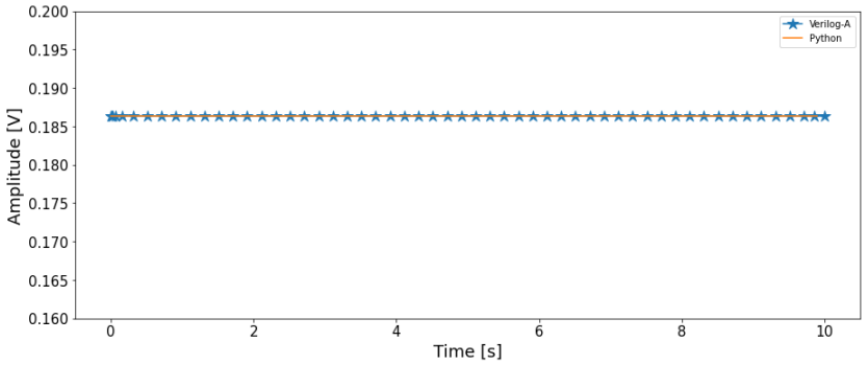

Figure 3.27 - Output FC3 Dummy Example

Looking to the comparative plots is possible to conclude that the code implemented in Verilog-A allowed the building of the same neural network previously implemented in Python. In order to test the generalization of the Python to VerilogA generator script other tests were performed and the results were also accurate, so the equation (30) and (31), which are the core of the Verilog-A code, are well constructed and general.

For the MLP with two delay lines, one of the thesis model, the same approach was performed using the same script to obtain the Verilog-A code, however, since the problem in hands required it, two delay lines were added to the data and the output of the first fully connected layer was concatenated with the circuit sizes. This two additions to the MLP model (2 delay lines and sizes concatenation) were also evaluated in a dummy example before being implemented in the thesis model (MLP with delay line). In the results section is presented, with detail, the outcomes of the implementation of the MLP with two delay lines in Verilog-A.

3.4.7 Model in the Designers Point of View

Since this model is implemented in Verilog-A with the intention to be used in the simulation of complex circuits, it is crucial to distance from the ML and model methodologies and provide the analog circuit's designer with a top view of the work, allowing them to perceive without any doubts which are the inputs and the expected output. One of the intention of this work, as mentioned previously, is to modelate the behavior of the OTA, so it is necessary to provide a top architecture, abstracted from the model of ML used to modelate the behavior.

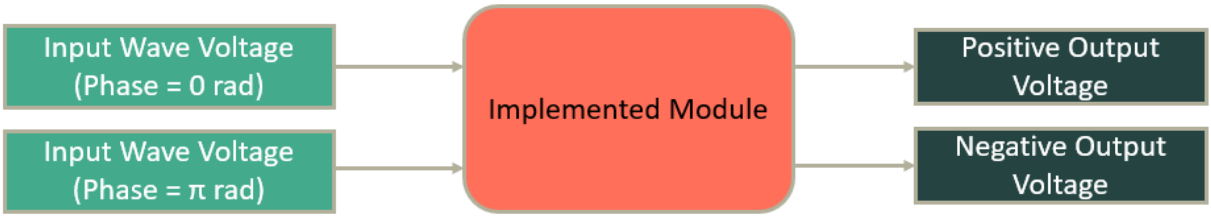

Figure 3.28 - Top-View Designers

In Figure 3.28 is possible to see the top view of the module, allowing the designer to perceive that the module has two analog inputs, the Voltage of the waves and the Voltage of the same wave although with a phase of π radians and two outputs the positive and the negative one. So when the designer wants to make use of this module it is only necessary to provide the correct inputs and the black-box module will perform all computation needed to obtain in the output the same output that an OTA would provide for the same input.

Chapter 4

Results

As mentioned in the implementation section, these results are obtained using Python programming language with PyTorch as an auxiliary framework to implement the neural networks. The machine used to train the model is a computer with 2 GPUs Nvidia RTX 2070, which allows reducing the training time. Using those 2 GPUs, 1 epoch takes approximately 3 minutes to run, using a regular personal computer it takes about 14 minutes.

In this section are presented the results of the models for the training sets alongside with the respective loss, which, as mentioned in the previous chapter, is the MSE. Knowing how the model performs in the training data, it is time to evaluate it in unseen data, so a test set is used to obtain the performance of the model.

To validate the innovative part of this work, despite the fact that using an LSTM for this problem brings an innovation itself, the model is also evaluated for different circuit sizes to confirm that for the circuit with size x the MSE is lower when it has the right size and higher when it has the wrong size.

A MLP with two delay lines is also implemented and evaluated for the same data used in the LSTM model evaluation. This model also takes in consideration the different circuit sizes.

It is also presented the results of the MLP with two delay lines in the HDL, Verilog-A, to show that is possible to implement ANN in that language allowing breakthroughs in this area when integrating those models in the AIDA software.

4.1 LSTM Model Results

First of all, in this section the results obtained for the LSTM model are presented. The train sets are used to prove that the network is learning properly, the test sets are used to consolidate the generalization of the model, and then the results for the different circuit sizes are presented, which is an innovation part of this work.

4.1.1 Train Performance on PyTorch LSTM Model

After feeding the model with training data in batches for all the epochs, obtain metrics and plots using the train dataset, nothing is more useful than to check whether or not the model is learning properly. Only with a successful training performance is possible to achieve a satisfactory performance for unseen data which is the primary goal when applying ANN. So randomly, 2 datasets were picked and used in the training phase, and its MSE and plots are obtained. It is important to clarify that to analyze the plots, the blue line represents the true label and the orange one represents the predicted curve. The plots represent the amplitude curve along the simulation time for the prediction of the model and for the true label generated

in the Spectre simulator. Note that the MSE is the mean between the MSE of the positive output and the negative output.

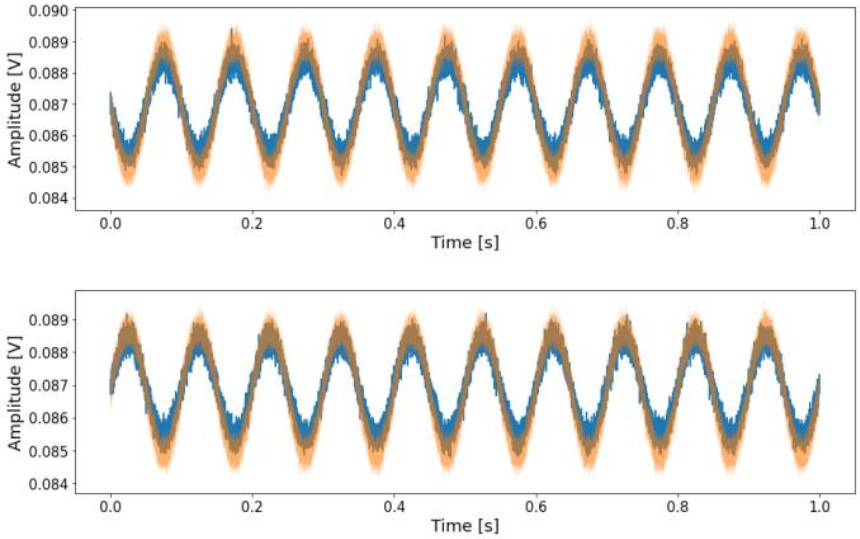

Figure 4.1 - Output Train Set 1 Waveform (amplitude varying)
(Blue line – True Label, Orange line – Predictions)

Figure 4.1 is the plot obtained for the train dataset where the amplitude varies along the time. Analyzing the plot, it is clear that the predicted curve follows the label curve, for the positive and negative output, with an acceptable accuracy, moreover the MSE is $2.3348 * 10^{-7}\ V^2$ which is considerably low.

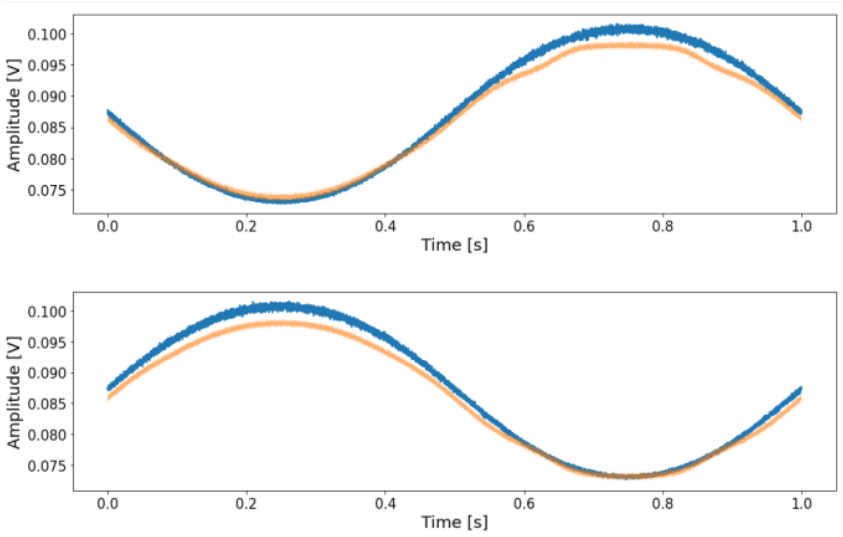

Figure 4.2 - Output Train Set 2 Waveform (frequency varying)
(Blue line – True Label, Orange line – Predictions)

Figure 4.2 is the plot obtained for the train dataset where the frequency varies along the time. Analyzing the plot, it is clear that the predicted curve follows the label curve with an acceptable accuracy, moreover the MSE is $2.7162 * 10^{-6}\ V^2$ which is also considerably low.

As mentioned in the implementation chapter, the label curves are obtained through a circuit simulation, so, as in any circuit, the noise is present. The prediction of the model mimics the noise too, being possible to see that the predicted curves have also noise and they are not only a simple thin line. In this way it turns the results more realistic and mimic the circuit with more precision. To make the noise, as explained in the implementation section, the mean value is $-4.3837 * 10^{-5}$ and the standard deviation value $2.5118 * 10^{-4}$.

4.1.2 Test Performance on PyTorch LSTM Model

From the results presented in the previous section, is possible to conclude that the model learned, with a high accuracy, the input data, being capable to mimic the waves it has seen before. However, as in any DL and ML project it is extremely necessary to evaluate the model on a different dataset, never seen by the model. For that purpose, 3 datasets, with different input waves, are generated using the SPICE Simulator and the same amplifier circuit. The same metric, the MSE, is used to evaluate the results as well as the plot of the predicted waves and the true output label waves. In Figure 4.3 is present the plot for the first test set used, which is build varying the amplitude. As it can be seen the model can generalize well to unseen data and it has a MSE of $4.9505 * 10^{-7}\ V^2$.

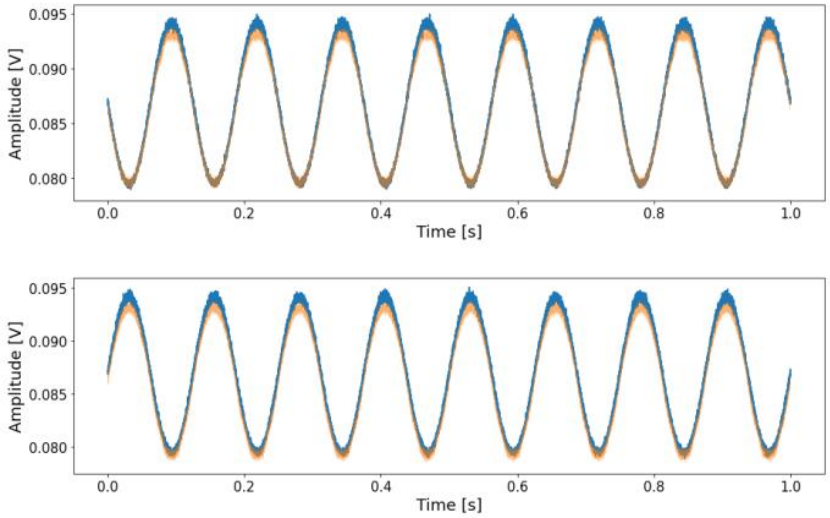

Figure 4.3 - Output Test Set 1
(Blue line – True Label, Orange line – Predictions)

In Figure 4.4 is present the plot for the second test set used. As it can also be seen the model can, once again, generalize well to unseen data and it has a MSE of $3.9313 * 10^{-6}\ V^2$. However, it can be seen that the amplitude is not so accurate as the frequency is, although still accurate.

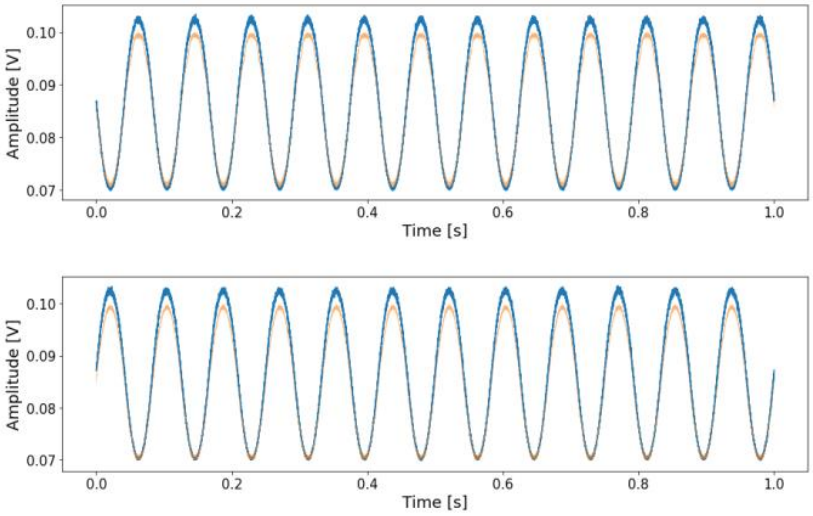

Figure 4.4 - Output Test Set 2
(Blue line – True Label, Orange line – Predictions)

In Figure 4.5 is present the plot for the third test set used, where the frequency varies. As it can also be seen the model generalizes well to unseen data and it has a MSE of $1.7136 * 10^{-7}\ V^2$.

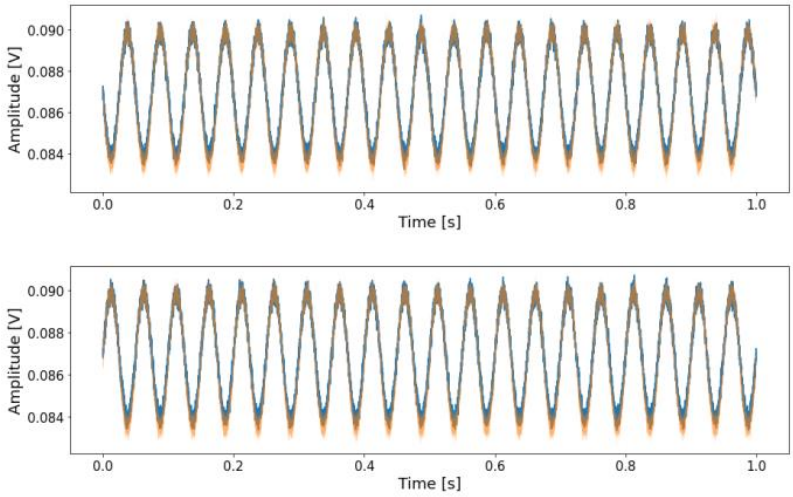

Figure 4.5 - Output Test Set 3
(Blue line – True Label, Orange line – Predictions)

As it can be seen, the model mimics with accuracy the unseen data producing a realistic copy of the behavior of the circuit. It is important to mention that these results are obtained with the right circuit dimension for the used point, in other words, if the test set is extracted with a certain circuit size the same circuit size is kept for the predictions. This relationship is important since the model learned the sizes of the devices and kept them associated to the respective input waves. Therefore, is relevant to show the model results for the different sizes. In the next section those results are outlined. Once more, also in the test results is possible see the presence of the built-in noise in the prediction, making the prediction near to the

real behavior of the amplifier circuit To create the noise, as explained in the implementation section, the mean value is $-4.3837 * 10^{-5}$ and the standard deviation value $2.5118 * 10^{-4}$.

4.1.3 PyTorch LSTM Model and Circuits Sizes

A proposed innovation to the state-of-the-art is to make the model general for different circuit device sizing. Figure 4.6 is presented a short table with some points and the corresponding sizing. The complete data contains, in total, 92 different circuit sizes.

	wp	wr	wn	lp	lr	ln
0	9.91E-05	4.44E-05	8.76E-05	9.48E-06	7.32E-06	4.89E-05
1	9.92E-05	4.44E-05	7.69E-05	1.41E-05	1.35E-05	4.81E-05
2	9.91E-05	5.68E-05	7.25E-05	8.48E-06	5.02E-06	4.89E-05
3	9.91E-05	4.44E-05	8.74E-05	1.39E-05	5.02E-06	4.68E-05
4	9.91E-05	4.44E-05	8.74E-05	9.48E-06	5.02E-06	4.76E-05
5	9.91E-05	4.44E-05	8.74E-05	9.48E-06	5.02E-06	4.91E-05
6	9.99E-05	4.44E-05	8.74E-05	9.48E-06	6.80E-06	4.91E-05
7	9.91E-05	4.44E-05	8.74E-05	9.48E-06	6.82E-06	4.91E-05
8	9.84E-05	4.63E-05	7.63E-05	8.82E-06	1.31E-05	4.96E-05
9	9.82E-05	4.63E-05	7.63E-05	1.40E-05	1.44E-05	4.96E-05
10	9.82E-05	4.63E-05	7.63E-05	1.40E-05	1.44E-05	4.98E-05

Figure 4.6 – Circuit Sizing Dataset

As a note for a better understanding of the table the *w* stands for width, the *l* stands for length, the *n* for nmos devices, *p* for pmos devices and *r* for resistors.

So, in Table 4.1 is possible to see the MSE of the model for different circuit sizes, obtained from random points and random sizes in order to have a sample of the model behavior when varying the sizes.

Table 4.1 – Model Error for across multiple circuit sizings

Sizing	MSE Point 0	MSE Point 53	MSE Point 90
0	$2.2676 * 10^{-6}$	$1.5458 * 10^{-6}$	$2.2509 * 10^{-6}$
25	$2.2685 * 10^{-6}$	$1.5446 * 10^{-6}$	$2.2535 * 10^{-6}$
33	$2.2694 * 10^{-6}$	$1.5442 * 10^{-6}$	$2.2521 * 10^{-6}$
53	$2.2677 * 10^{-6}$	$1.5409 * 10^{-6}$	$2.2555 * 10^{-6}$
77	$2.2730 * 10^{-6}$	$1.5436 * 10^{-6}$	$2.2508 * 10^{-6}$
90	$2.2678 * 10^{-6}$	$1.5437 * 10^{-6}$	$2.2506 * 10^{-6}$
92	$2.7066 * 10^{-6}$	$1.5412 * 10^{-6}$	$2.2516 * 10^{-6}$

Analyzing Table 4.1 is possible to see that when the point matched the correct circuit sizing the MSE is lower, meaning that the model can distinguish the different device's sizes. Therefore, it innovates relatively to the previous state-of-the-art, where a single model was only capable of modeling a circuit for one dimension. It is important to mention that those results are obtained using the predicted wave with noise. In this way the construction of a model capable to mimic the behavior of the circuit for multiple sizes was achieved.

4.2 MLP with Delay Line Results

Now it is time to analyse the MLP with delay line results. The same evaluation strategy used in the LSTM model is applied for this model. Once again, it is important to note that the MSE is a mean between the MSE of the positive and negative output.

4.2.1 Train Performance on PyTorch MLP Model

With the same principle of the LSTM, it is important to acknowledge that the model is capable to learn and mimic the circuit behavior, so in that sense it is relevant to present the train results. Once again, the blue line stands for the true label and the orange one stands for the MLP predictions.

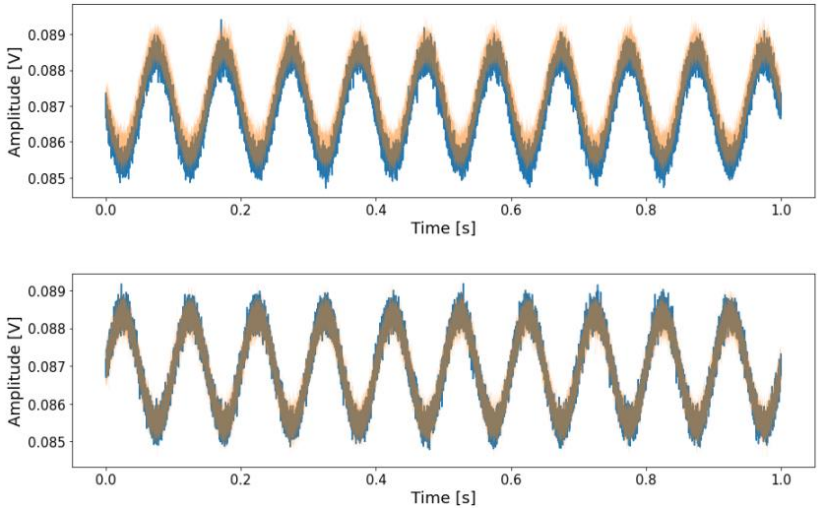

Figure 4.7 – Train set 1 (Varying Amplitude)
(Blue line – True Label, Orange line – Predictions)

In Figure 4.7 is possible to see that the model output clearly follows the true label wave, giving a value for the MSE of $1.6254 * 10^{-7}\ V^2$.

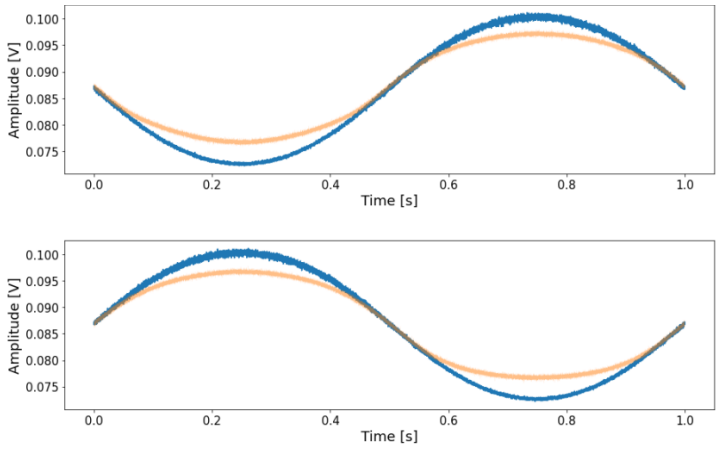

Figure 4.8 – Train set 2 (Varaying Frequency)
(Blue line – True Label, Orange line – Predictions)

In Figure 4.8 it is also possible to conclude that the model learns the input data, being capable to mimic the behavior of the amplifier. For this set of data, the MSE value is $5.7439 * 10^{-6}\ V^2$. Now it is important to see if the model is capable to generalize for unseen data, so, as it was done to the LSTM model, the same datasets for test are used and the results are the following. Note that to reproduce the noise, as explained in the implementation section, the mean value is $-4.3837 * 10^{-5}$ and the standard deviation value is $2.5118 * 10^{-4}$.

4.2.2 Test Performance on PyTorch MLP Model

In order to see if the model is capable to generalize to unseen data, 3 new datasets (the same used to evaluate the LSTM model) are generated and given to the model to make its predictions.

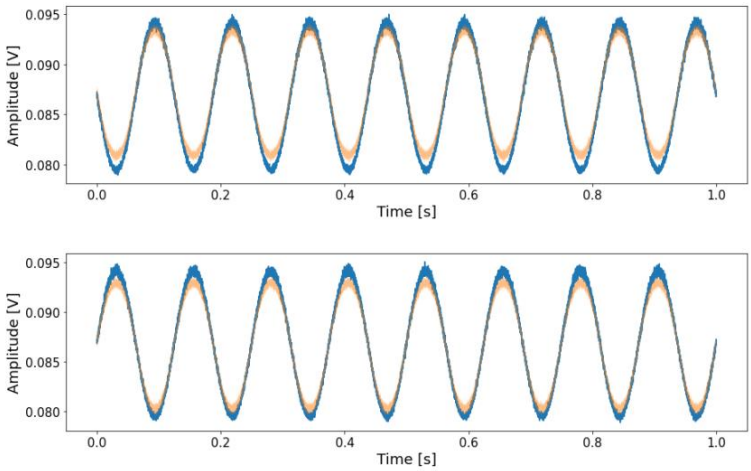

Figure 4.9 - Test Set 1 MLP Results
(Blue line – True Label, Orange line – Predictions)

In Figure 4.9 is possible to see that the model is capable to generalize for unseen data, since the predicted wave from the model follows with accuracy the true label wave, producing a MSE value of $5.9297 * 10^{-7}\ V^2$.

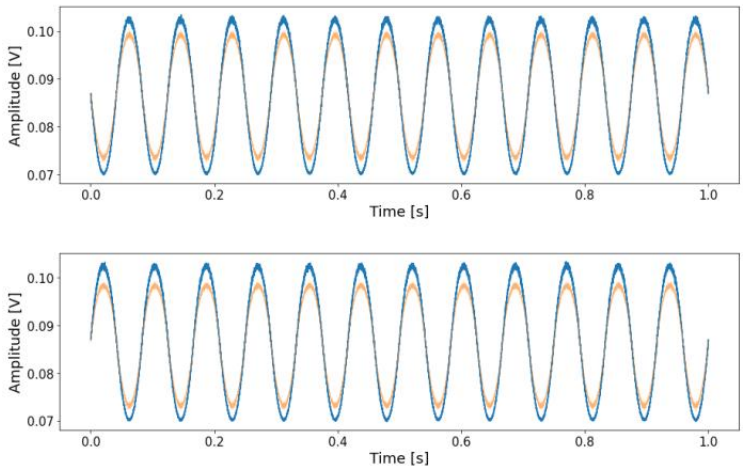

Figure 4.10 - Test Set 2 MLP Results
(Blue line – True Label, Orange line – Predictions)

The Figure 4.10 it is also explicit in terms of the results obtained and, in this case, the MSE produced is $9.0773 * 10^{-6}\ V^2$.

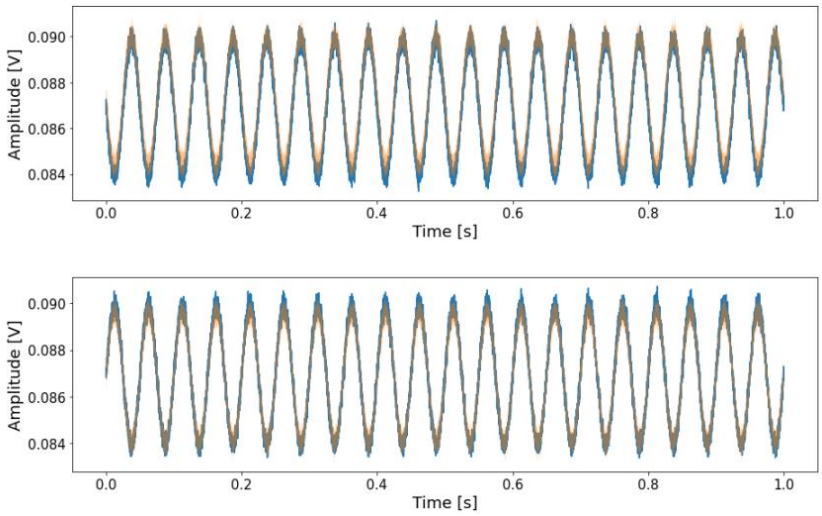

Figure 4.11 – Test Set 3 MLP Results
(Blue line – True Label, Orange line – Predictions)

Finally, for Figure 4.11 the MSE obtained is $1.7319 * 10^{-7}\ V^2$. In this way is possible to ensure the quality of the model and its capability to generalize to new data.

4.2.3 PyTorch MLP Model and Circuit Sizes

Once again to verify if the innovation proposed is achieved, taking in consideration the table in Figure 4.6 where is presented a short table with some points and the correspondent size. This table is the same used in the LSTM model.

So, in Table 4.2 is possible to see the MSE of the model for different circuit sizes obtained for random points and random sizes in order to have a representative sample of the model behavior when varying the sizes.

Table 4.2 - Sizes Results MLP Model

Sizes	MSE Point 0	MSE Point 53	MSE Point 90
0	$5.7075 * 10^{-6}$	$1.9010 * 10^{-5}$	$5.7517 * 10^{-6}$
25	$7.6790 * 10^{-6}$	$3.3400 * 10^{-5}$	$6.5505 * 10^{-6}$
33	$7.6798 * 10^{-6}$	$3.3453 * 10^{-5}$	$6.5527 * 10^{-6}$
53	$6.1241 * 10^{-6}$	$1.4128 * 10^{-5}$	$6.8016 * 10^{-6}$
77	$6.4763 * 10^{-6}$	$2.7688 * 10^{-5}$	$5.7773 * 10^{-6}$
90	$5.8932 * 10^{-6}$	$1.9226 * 10^{-5}$	$5.7226 * 10^{-6}$
92	$5.8953 * 10^{-6}$	$1.9245 * 10^{-5}$	$5.7377 * 10^{-6}$

Analyzing Table 4.2 is possible to see that when the point is equal to the dimension the MSE is lower, meaning that the model is capable to distinguish the different device's sizes, therefore, once again, it innovates relatively to the previous state-of-the-art, where a single model was only capable to model a circuit for one dimension. It is important to mention that those results are obtained using the predicted wave with noise. Once again, in this way the construction of a model capable to mimic the behavior of the circuit for multiple circuit sizes was also achieved using a different model, in this case a MLP model with two delay lines.

4.3 MLP vs. LSTM Results Comparison

Having the results of each model, it is essential to perform a comparison between them. First of all it is necessary to compare the MSE produced by each model for each dataset, either for train or evaluation, so in Table 4.3 is present a summary of the MSE values for the train datasets and test datasets for each model, allowing an easy comparison of the results.

Table 4.3 - MSE LSTM vs MLP

MSE (V^2)	*Train Set 1*	*Train Set 2*	*Test Set 1*	*Test Set 2*	*Test Set 3*
LSTM Model	$2.3348 * 10^{-7}$	$2.7162 * 10^{-6}$	$4.9505 * 10^{-7}$	$3.9313 * 10^{-6}$	$1.7136 * 10^{-7}$
MLP Model	$1.6254 * 10^{-7}$	$5.7439 * 10^{-6}$	$5.9297 * 10^{-7}$	$9.0773 * 10^{-6}$	$1.7319 * 10^{-7}$

In Table 4.3 is possible to conclude that the MSE for the LSTM model is slightly lower than the MLP with 2 delay lines model, however the losses have the same order so it is possible to conclude that the models' performances are similar. There are reasons behind this results and conclusion. The LSTM model can learn long term dependencies and propagate them through the next sequences in the opposite side in the MLP with delay line the time aspect is only inserted in the input features and the number of delays used are find by trial and error, making the MLP approach more complex in terms of data pre-processing. The LSTM training phase is performed using backpropagation through time which allows the output weights and hidden weights to be trained by unfolding the network in time, so the weights are an average through the propagation in time, leading to a loss reduction. Another important heuristic conclusion, present in Table 4.4 is that, for the same number of samples in the training dataset the LSTM model have more parameters to train than the MLP with delay line, which leads to an increase of time and memory when training the LSTM model and, in some cases, lead to out-of-memory errors during the training phase. So, the choice between an LSTM and MLP with delay lines has a trade-off between accuracy or training time and memory. It is also important to mention that the MSE of the MLP could be lower than the LSTM one if a higher number of delay lines was implemented, however the precise number of delay lines is only obtained by trial and error. So, in order to conclude, using an MLP with the right number of delay lines is better than using an LSTM model since it allows to obtain accurate results in less training time and with less parameters to train.

Table 4.4 - Heuristic Comparison

Heuristics	*Train Time*	*Trainable Parameters*
LSTM Model	$57\ minutes$	$937.452\ parameters$
MLP Model	$31\ minutes$	$186.152\ parameters$

Since the model capability to be general for multiple circuit sizes is captured in the Fully Connected Layers, its implementation has the same difficulty for both models. In the previous sections is possible to conclude that the LSTM model and the MLP model are both capable to distinguish the different sizes and give a better performance when using the right circuit size.

For both models, one possible way to improve the results is when the data is acquired extract more representative points of the picks and valleys of each waves in order for the model to be capable to have a better learning of those parts of the waves. This topic is addressed with some detail in the conclusions section, more properly in the future work subsection.

4.4 MLP in Verilog-A Language

Having the MLP model with two delay lines created and trained in Python (using the hyperparameters specified in previous sections), all the weights and bias matrices are available to be feed to the generator script in order to obtain the Verilog-A code for the five fully connected layers and activation functions, which

are the ELU and Sigmoid. Through the generator script, is possible to obtain the concatenation Verilog-A code, needed to concatenate not only the circuit sizes with the output of the first fully connected layer but also to concatenate the 6 input waves. Using the generator script, the delay line is also implemented in Verilog-A code. To scale the data, a scaler is also implemented using the MinMaxScaler principle with maximum value 0.2749 and minimum value 0.225. To obtain the output wave a de-normalizer is also implemented using 0.3193 as maximum value and 0.006527 as minimum value. Having all the Verilog-A codes is possible to create symbols for each module and connect them in the schematic as the figure Figure 4.12 presents.

It is important to mention that to change the circuit size is necessary to override the weights vector, which in Verilog-A is left as a changeable parameter giving the user the possibility to change it according to the desired circuit.

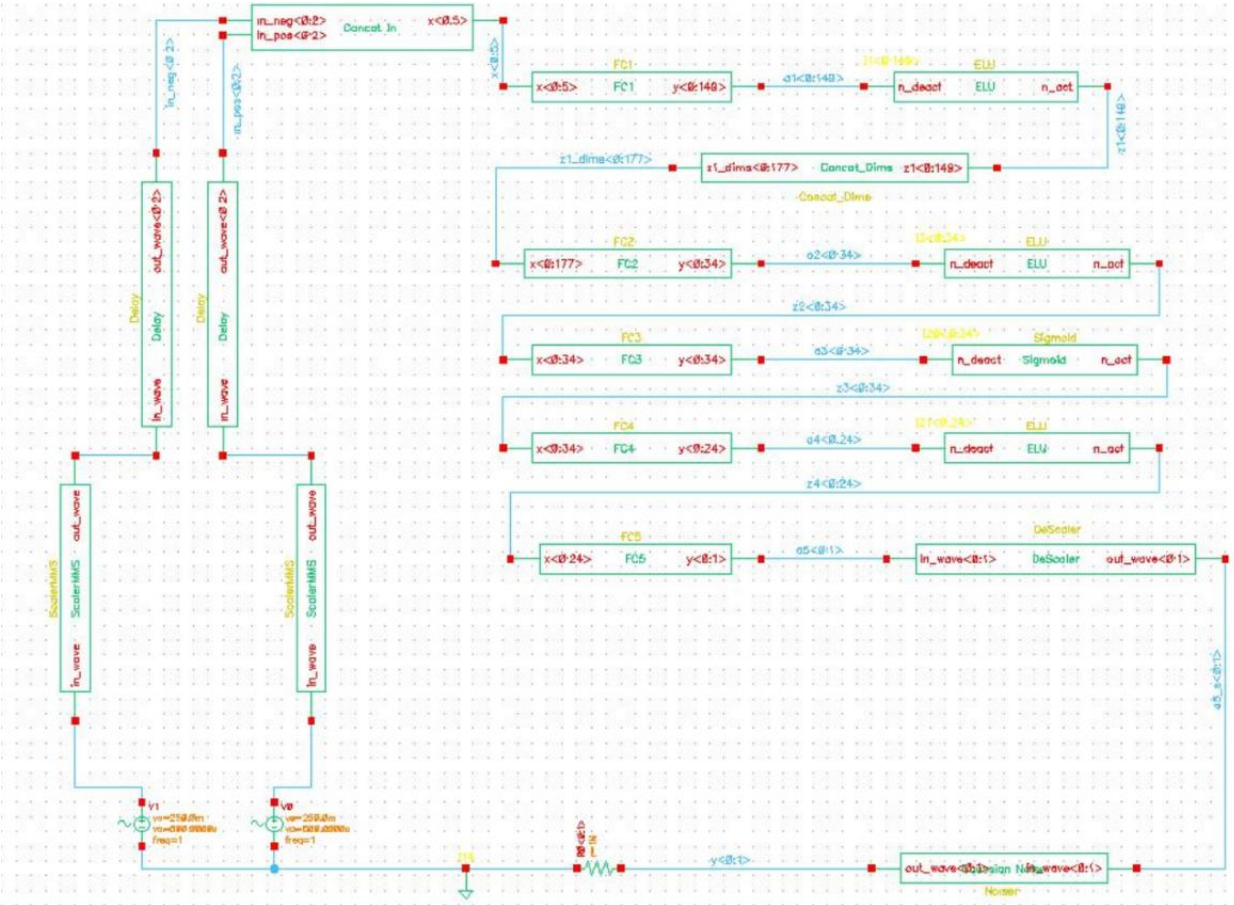

Figure 4.12 - MLP VerilogA Schematic

Figure 4.12 presents the schematic of the MLP with two delay lines in Verilog-A. The *ScalerMMS* and *DeScaler* blocks are responsible for scaling and de-scale the data using the MinMaxScaler approach. The *Delay* blocks are responsible for introducing two delay lines in the data. The Concat_In and Concat_Dims are responsible for performing the concatenation between the waves and between waves and the circuit

sizing vector, respectively. After those blocks, we have the MLP with the Fully Connected Layers (FC1, FC2, FC3, FC4, FC5) and the activation functions (ELU and Sigmoid). The wires perform the connection between the blocks, and it is important to have the right number of wires, for example the FC1 has 149 neurons of output so the wires that connects the FC1 to the ELU must have a total number of 149.

Having the model schematic prepared it is time to define the two input waves (one of them is similar to the other but with phase π) and start the simulation. The first simulation of the MLP model in Verilog-A took around 6 minutes since the cadence software needed to compile the libraries and the modules. The subsequent simulations only took 1 or 2 seconds to be finished.

For this case, the same waves presented in the MLP results section are used to keep the same analysis principle. In the plots are presented the true label, the Python model prediction and the VerilogA model prediction, in order to demonstrate that the MLP with two delay lines was successfully implemented in Verilog-A language. To analyse the results, the blue line is the true label, the purple line is the Python MLP model prediction and the dot line in yellow is the Verilog-A MLP model prediction.

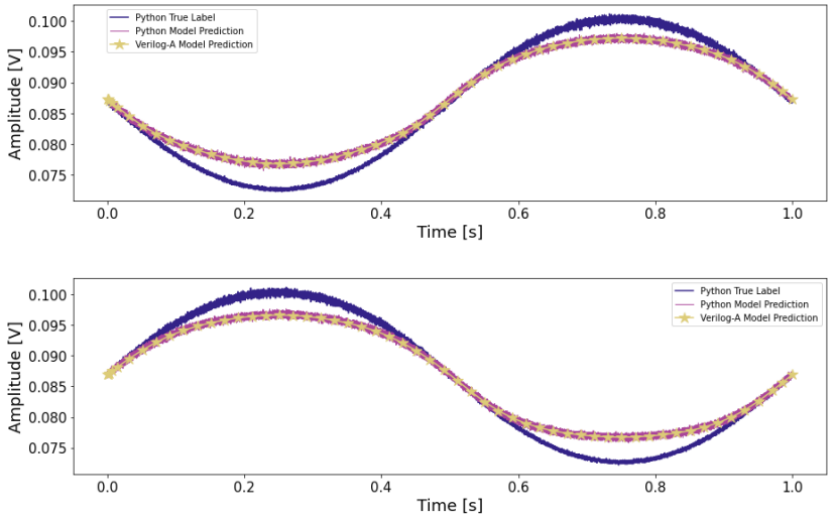

Figure 4.13 - Verilog-A Train Set 2 Result
(Blue line – True Label, Pink line – Python Predictions, Yellow dots – Verilog-A Predictions)

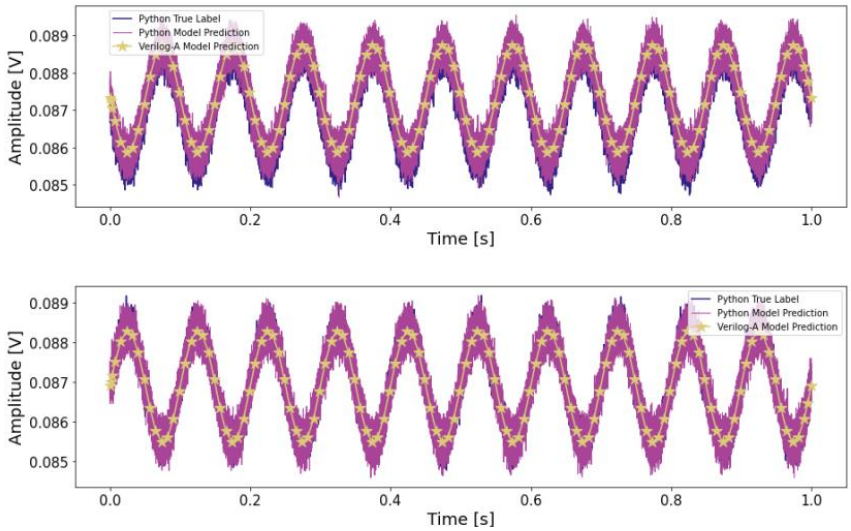

Figure 4.14 - Verilog-A Train Set 1 Result
(Blue line – True Label, Pink line – Python Predictions, Yellow dots – Verilog-A Predictions)

First of all, the results presented in Figure 4.13 and Figure 4.14, the train datasets, shows that the Verilog-A MLP model is well constructed, following the performance of the Python model. It is important to notice that the Verilog-A model have a gaussian noise added, which can be seen in the schematic, in the right down corner of the Figure 4.12, the *Noiser module*, however in the plot it is not shown since is used dots and not using dots would create an incomprehensive plot and lack of conclusions could be taken.

To keep testing the implementation of this model in the HDL, the 3 test sets are used to prove that for all waves used in this work, for train and test, the model in HDL is capable to make right predictions, mimicking the circuit behavior as accurate as the Python model.

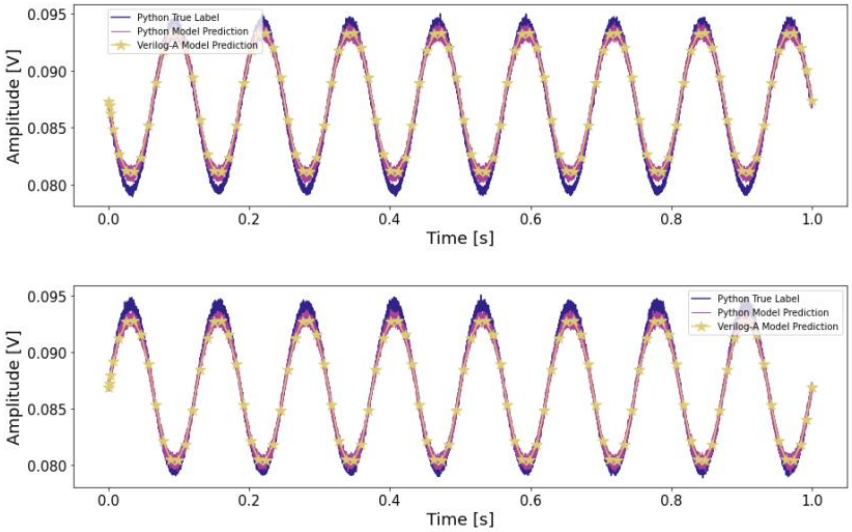

Figure 4.15 - Verilog-A Test Set 1 Result
(Blue line – True Label, Pink line – Python Predictions, Yellow dots – Verilog-A Predictions)

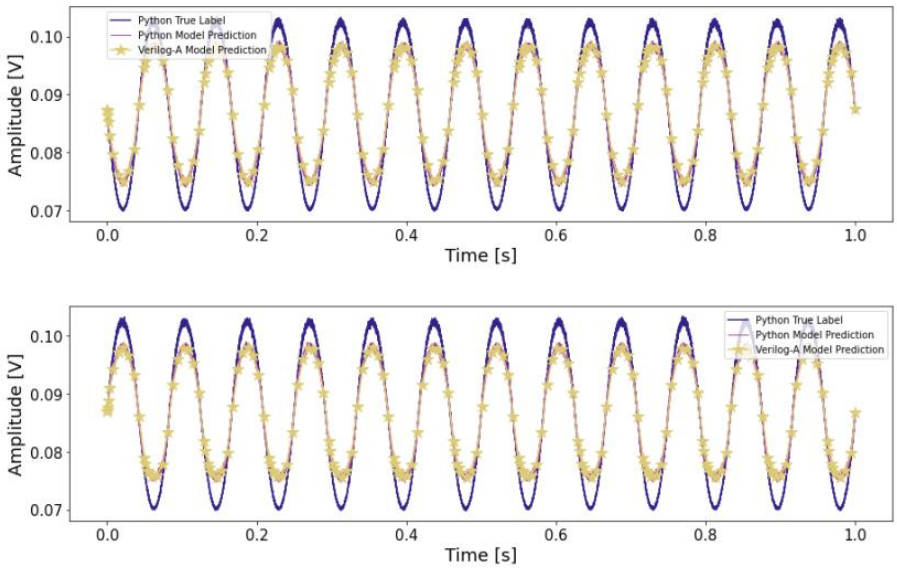

Figure 4.16 - Verilog-A Test Set 2 Result
(Blue line – True Label, Pink line – Python Predictions, Yellow dots – Verilog-A Predictions)

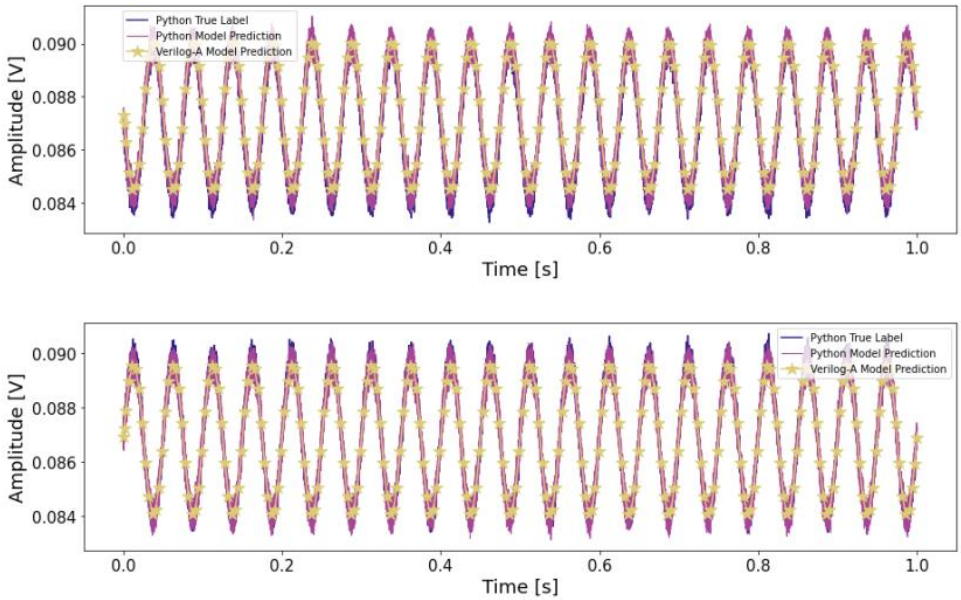

Figure 4.17 - Verilog-A Test Set 3 Result
(Blue line – True Label, Pink line – Python Predictions, Yellow dots – Verilog-A Predictions)

In Figure 4.15, Figure 4.16 and Figure 4.17 the results for the 3 test sets are presented, proving, once again, that the Verilog-A model is capable to mimic the behavior of the circuit, being a strong basis for future integration in the AIDA software to accelerate complex front-end circuits. In this way the objective of converting the Python neural network into Verilog-A language is achieved.

It is relevant to distinguish the model architecture from the simulation model. In Figure 4.12 the Delay, Connection, Fully Connected, Sigmoid, Elu, Scaler, De-Scaler and Noiser blocks make part of the model

architecture, while in the simulation model, to the model architecture blocks, are added the sources, the resistor and the ground components.

Relatively to simulation time, at transistor level it takes 2.68 seconds to generate the output and in the Spectre simulation of the Verilog-A model the time spend to obtain the output is 0.593 seconds. This proves that the model achieved other goal, being faster than the transistor level simulation, in this case 5 times faster. To give a more insight information about the time results, at transistor level the optimization for 128 points and 500 generations takes 46 hours to obtain the output, using the Verilog-A model it takes 10 hours.

Taking in consideration all the results obtained, the implemented models are a significant step forward when comparing with previous approaches, present in the state-of-the art since it allows a single model to mimic the behavior of an amplifier (OTA) for multiple device sizes. The generator script is a tool that will allow to speed up the conversion of ANN to Verilog-A making this research area grow since it has more support to address the conversion issue.

Chapter 5

Conclusions

As in other engineering work, taking conclusions from the implementation and the results is relevant because it allows the engineer to perceive what can be further explored and what can be improved in his work. Only with this critical analysis is possible to make relevant breakthroughs in any science field. In the Electronic Design Automation field, the mantainence of behavior models for analog circuits lacks of exploration so only with cooperation between Circuit Designers and Machine Learning experts is it possible to achieve results in this research area which always needs to be carefully analyzed. In this section a conclusion about the implementation and the results obtained is performed and, since it is an open problem, where there is always room for improvements, some future work to improve and extend this work is suggested.

5.1 Work Conclusions

This work tested a new approach to modeling the behavior of analog circuits. Firstly an LSTM model is used, which, relatively to state-of-the-art, presents an innovative idea to model the behavior of an amplifier and the sizes of the circuit devices, making the model sensitive to those parameters. So, in other words, not only a novel model was used in this research field but also it can generalize for different circuit sizes. In order to compare two behavioral models, an MLP model with two delay lines was created, allowing to conclude that both models have similar performance; however, the number of delay lines plays an important role, with the correct number, which can only be achieved with trial-error, the MLP model can achieve better performance. The models were built in the Python/PyTorch framework. The results are accurate and prove that those models can model the amplifier's behavior. However, it has always room for improvement, so these results seem promising and justify further improvements, optimization and research.

It was necessary to convert to an HDL the ML models to allow to bring them out from the Python and use them to solve a real problem, which in this case consisted in modeling analog circuits. So, the proposed HDL was the Verilog-A, which presents much less available information (compared to pyhton for example), making the work more challenging. In order not only to facilitate the conversion of the ANN from Python to Verilog-A but also to facilitate further models, a generator script was created. This generator script currently supports Fully Connected Layers, activation functions, and a few extra modules necessary to integrate an ANN in Verilog-A. Still, it is open to be augmented with other types of layers to automatize this conversion. Another essential drawback is that the Verilog-A compilation in SPECTRE has a limited memory, which in this work, layers with more than 150 neurons made the program fail in the compilation part.

Overall, the work achieved accurate results, allowing us to explore two different behavioral models. It also allowed us to take a step forward in the Verilog-A language and make a generator script capable of automatically converting the ANN to this HDL.

5.2 Future Work

This work went from curating the data, implementing ML models using state-of-the-art ML frameworks, converting them to Verilog-A, and simulating the circuit in industry-standard circuit simulators. It proves that it is possible to integrate/interface these technologies to implement neural networks in this HDL and use them to model circuit behavior. But, as with any other science and engineering work, there is always a margin for improvements and extensions. This work is not an exception. Throughout the document, some possible extensions were already mentioned.

The model's quality derives not only by the data used but also by the hyperparameters used. This work used the adaptative learning rate as one of many techniques to find the best value for this parameter, however, there are other algorithms to find the best values for the other parameters, so a Grid Search algorithm can be used to find in detail those values. For example, one hyperparameter in the MLP model that could improve its performance is the number of delay lines.

The noise was captured by adding a gaussian curve to the prediction, which has the mean and standard deviation of the training error. This approach, although practical, is simple. As a future work, Variational Auto Encoders could be used to learn more complex noise distributions in the training phase. In [42], the authors suggested one approach to follow to attain this objective.

As mentioned in the document, the Verilog-A code has limited memory, leading to models with a low number of neurons. Although, in this work, the number of neurons chosen was enough to make the model learn, it might be necessary to increase the number of neurons in other cases. So, to overcome this issue, the Verilog-A code can be optimized.

Having the ANN implemented in Verilog-A allows for evaluating the performance of the complex circuit with the model incorporated into it, which can accelerate simulation-based circuit optimization tools, such as the in-house AIDA software [43].

These models can generalize for different circuit sizes; however, one possible improvement for future work is to give the model the capability to generalize for multiple topologies of amplifiers. The model must be given an encoding to distinguish the different topologies to attain that goal. One hot encoding is trivial but likely sufficient for such a task as the encoding needs more information, so using a Graph Neural Network could extract that information about each circuit and feed the model is a strategy worth considering. The authors in [44] propose an approach to give a sense of structure to the input data, reducing the number of trainable parameters and making a model capable of generalizing for different topologies.

References

[1] Martins, R., Horta, N. and Lourenço, N., “Automatic Analog IC Sizing and Optimization Constrained” (1st ed.), Springer, 2017.

[2] Gielen, G. G. E. and Rutenbar, R. A. (2000) “Computer-Aided Design of analog and Mixed-Signal Integrated Circuits,” Proceedings of the IEEE, pp. 1825-1854.

[3] Jordan, M. I. and Mitchell, T. M. (2015) , “Machine learning: Trends, perspectives, and prospects.” Science, 349(6245), 255–260.

[4] Zhang Q. and Vijay K. D. (2003). “Artificial Neural Networks for RF and Microwave Design—From Theory to Practice”. IEEE Transactions on Microwave Theory and Techniques, Vol. 51.

[5] Miller, I., FitzPatrick, D., & Aisola, R. (1997), “Analog design with Verilog-A”. Proceedings of Meeting on Verilog HDL.

[6] Bose S., Shen B., Johnstion M. (2020). “A Batteryless Motion-Adaptive Heartbeat Detection System-on-Chip Powered by Human Boady Heat”. IEEE Journal of Solid-State Circuits, Vol. 55, No.11.

[7] https://www.aidasoft.com/Home

[8] Grabmann, M., Landrock, C. and Gläser, G. (2021), “Machine Learning in Charge: Automated Behavioral Modeling of Charge Pump Circuits”. SMACD / PRIME 2021.

[9] Goodfellow I., Bengio Y., Courville A. (2016) Deep Learning, MIT Press.

[10] Kingma D. P., Ba J. (2014). “Adam: A Method for Stochastic Optimization”. In: CoRR, abs/1412.6980.

[11] Grabmann, M., Landrock, C. and Gläser, G. (2019), “Power to the Model: Generating Energy-Aware Mixed-Signal Models using Machine Learning”. SMACD 2019, Lausanne, Switzerl.

[12] A. Suissa, et al. (2010). “Empirical method based on neural networks for analog power modeling”. IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 29 (5) 839–844.

[13] Lei, J.Y and Chatterjee, A. (2021), “Automatic Surrogate Model Generation and Debugging of Analog/Mixed-Signal Designs Via Collaborative Stimulus Generation and Machine Learning”. 26th Asia and South Pacific Design Automation Conference.

[14] Hasani, R. M., Haerle, D., Baumgartner, C. F., Lomuscio, A. R. and Grosu, R. (2017). “Compositional neural-network modeling of complex analog circuits”. International Joint Conference on Neural Networks.

[15] Xie H., Tang H., Liao Y-H. (2009). “Time Series Prediction based on NARX neural networks: An advanced approach”. 2009 International Conference on Machine Learning and Cybernetics.

[16] Yu, H., Swaminathan, M., Ji, C. and White, D. (2017). “A method for creating behavioral models of oscillators using augmented neural networks”. Conference on Electrical Performance of Electronic Packaging and Systems.

[17] Yu, H., Swaminathan, M., Ji, C. and White, D. (2018). “Behavioral Modeling of Steady-State Oscillators with Buffers Using Neural Networks”. Conference on Electrical Performance of Electronic Packaging and Systems.

[18] Yu, H., Swaminathan, M., Ji, C. and White, D. (2018). “A Nonlinear Behavioral Modeling Approach for Voltage-controlled Oscillators Using Augmented Neural Networks”. IEEE/MTT-S International Microwave Symposium - IMS.

[19] Vijay K. D., Mustapha C. E., Yonghua F., Jianjun X., Zhang Q. (2001). “Neural Networks for Microwave Modeling: Model Development Issues and Nonlinear Modeling Techniques”. Int. J. RF Microw. Computer-Aided Eng.

[20] Watson P.M., Gupta K.C. (1997). “Design and optimization of CPW circuits using EM-ANN models for CPW components”. IEEE Trans. Microw. Theor. Tech. 45 (12) 2515–2523.

[21] Passos M.G., Silva P.d.F., . Fernandes H.C. (2006). “A RBF/MLP modular neural network for microwave device modeling”. Int. J. Comput. Sci. Netw. Secur. 6 (5A) (2006) 81–86.

[22] Ceperic V., Baric A. (2004). “Modeling of analog circuits by using support vector regression machines”. 11th IEEE International Conference on Electronics, Circuits and Systems.

[23] Ding M., Vemuri R. (2005). “An active learning scheme using support vector machines for analog circuit feasibility classification”. 18th International Conference on VLSI Design held Jointly with 4th International Conference on Embedded Systems Design, IEEE, 2005, pp. 528–534.

[24] Naeem M., Rizvi S. T., Coronato A. (2020). “A Gentle Introduction to Reinforcement Learning and its Application in Different Fields”. IEEE Access (Volume: 8).

[25] Akram A., Lowe-Power J. (2020). “The Tribes of Machine Learning and the Realm of Computer Architecture”

[26] Bre, F., Gimenez, J. M., and Fachinotti, V. D. (2018). “Prediction of wind pressure coefficients on building surfaces using artificial neural networks.“ Energy and Buildings, 158, 1429–1441.

[27] Bilbao I., Bilbao J. (2017). “Overfitting problem and the over-training in the area of data”. 8th IEEE International Conference on Intelligent Computing and Information Systems (ICICIS 2017).

[28] Depois, J. (2019) “Memorizing is not learning¡‘ (available at https://hackernoon.com/memorizing-isnot-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42)

[29] Huang, G.-B. (2003). “Learning capability and storage capacity of two-hidden-layer feedforward networks.“ IEEE Transactions on Neural Networks

[30] Ke, J., and Liu, X. (2008). “Empirical Analysis of Optimal Hidden Neurons in Neural Network Modeling for Stock Prediction.“ 2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application.

[31] Suzuki K., Nishio A., Kamo A., Watanabe T. and Asai H., (2000). “An application of Verilog-A to modelling of back propagation algorithm in neural networks”. 43rd IEEE Midwest Symposium on Circuits and Systems.

[32] Chen Z., Raginsky M., Rosenbaum E. (2017). “Verilog-A Compatible Recurrent Neural Network Model for Transient Circuit Simulation”. IEEE 26th Conference on Electrical Performance of Electronic Packaging and Systems.

[33] Tzenov P., Sokar A. (2021). “Machine Learning in the Analog Circuit Simulation Loop”. SMACD / PRIME 2021.

[34] Kosari A., Breiholz J., Liu N., Calhoun B., Wentzloff D. (2018). “A 0.5V 68nW ECG Monitoring Analog Front-End for Arrhythmia Diagnosis”. Journal of Low Power Electronics and Applications.

[35] https://pytorch.org/

[36] Hochreiter S., Schmidhuber J. (1997). “Long Short-term Memory”. Neural Computation.

[37] https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

[38] Loshchilov I., Hutter F. (2019). “Decoupled Weight Decay Regularization”. The International Conference on Learning Representation (ICLR).

[39] Ezzeldin A., Saeed M., Elfeky S., Khater A. (2020). “Neural Network with Adaptative Learning Rate”. 2nd Novel Intelligent and Leading Emerging Sciences (NILES2020).

[40] https://pytorch.org/docs/stable/data.html

[41] Suzuki K., Nishio A., Kamo A., Watanabe T., Asai H. (2000). “An Application of Verilog-A to Modeling of Back Propagation Algorithm in Neural Networks.”. 43rd IEEE Midwest Symp. On Circuits and Systems.

[42] Chung J., Kastner K., Dinh L., Goel K., Courville A., Bengio Y. (2016). “A Recurrent Latent Variable Model for Sequential Data”.

[43] https://www.aidasoft.com/Solutions

[44] Gusmão A., Horta N., Lourenço N., Martins R. (2021). “Bringing Structure into Analog IC Placement with Relational Graph Convolutional Networks”. SMACD 2021.