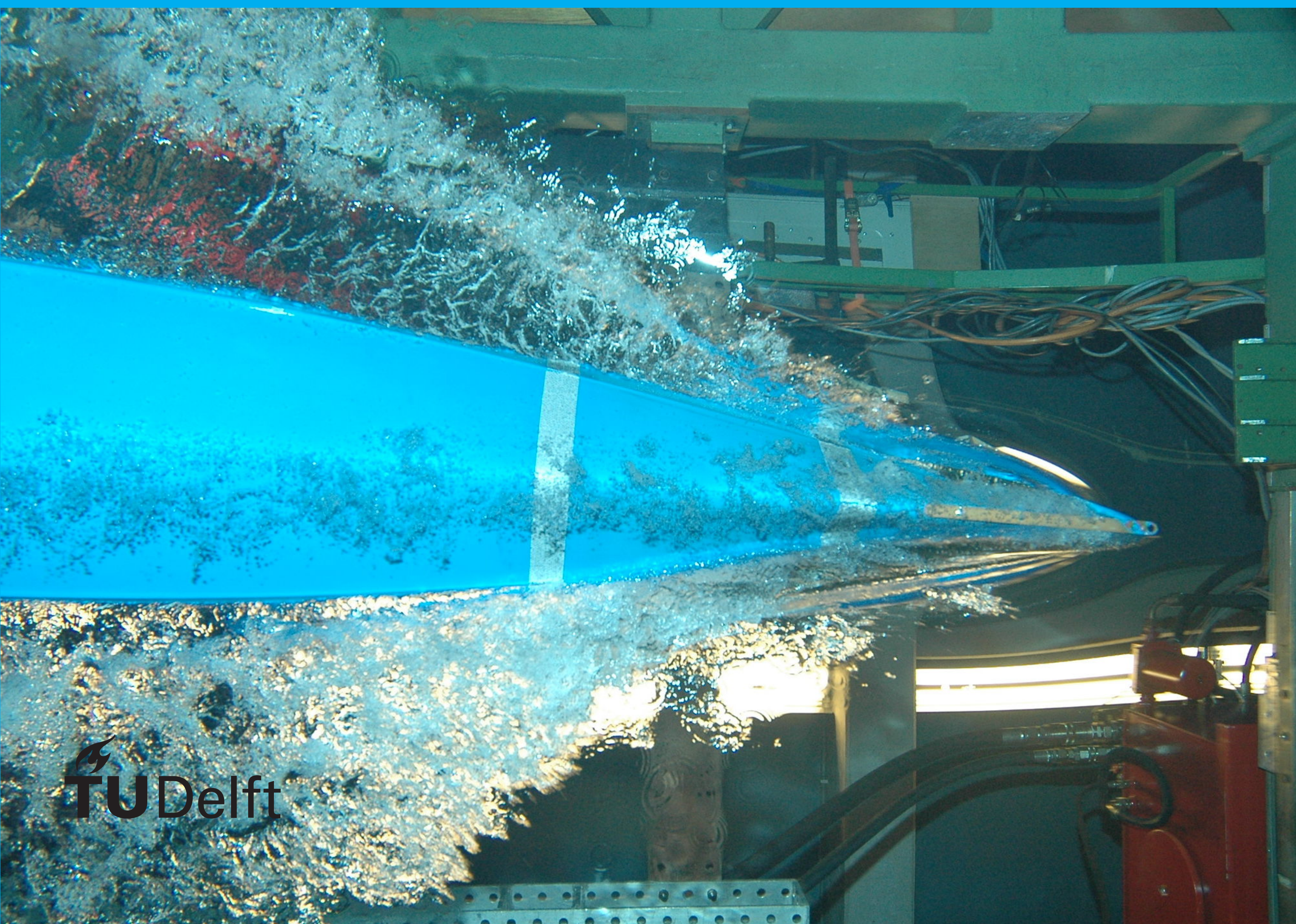
Evaluating the Performance of an RNN Hardware Accelerator on Digital Pre-Distortion

Xinyu Liao

Master Thesis



Evaluating the Performance of an RNN Hardware Accelerator on Digital Pre-Distortion

by

Xinyu Liao

to obtain the degree of Master of Science
at the Delft University of Technology,

Student number:	5656486
Committee:	Dr. Sijun Du
	Dr. Chang Gao
Project duration:	September, 2023 – August, 2024

TU Delft

Preface

After two years of learning at TU Delft, I am approaching graduation from the master's program. During these two years of study, I have been drawn in by the thoughtfully crafted curriculum and course content. The systematic approach to both education and research, with its emphasis on direct, efficient discussions and practical experiences, is a style of learning that I truly admire.

Although it does not seem like a long time, my thesis journey this past year has been incredibly enriching. I want to express my heartfelt gratitude to my supervisor, Dr. Chang Gao. Having him guide me through this process was a stroke of luck. From the very start of my thesis to the exploratory phases and the final design, he provided invaluable support. I was always amazed that he could discover potential problems quickly and guide me to solutions. His detailed feedback and thorough reviews of my thesis report were crucial to my progress. I am also deeply grateful to Dr. Sijun Du for agreeing to chair my MSc committee. His expertise and encouragement have been invaluable in helping me navigate challenges and motivating me to aim for higher standards in my academic work.

I am also profoundly thankful for the support of my best friends, who have been a source of encouragement and inspiration. Mr. Jiacong Li not only helped me greatly with code programming but also accompanied me through difficult moments. I also appreciate Mr. Maodong Yang and Mr. Chong Zhan for their assistance and patience in learning and daily life.

Finally, I want to express special thanks to my mother. Even though she was far away, her consistent, timely, and warm comfort was always there for me, reminding me that she remains my greatest supporter.

Xinyu Liao
Delft, July 2024

Abstract

As wireless communication systems grow more complex, especially with the advent of 5G networks, the demand for efficient digital predistortion (DPD) techniques to improve the linearity and performance of power amplifiers (PAs) is increasing. Advanced neural network hardware accelerators like EdgeDRNN offer promising solutions to these challenges by enhancing computational efficiency and accuracy. This thesis provides a detailed evaluation of the EdgeDRNN hardware accelerator, which is applied by the OpenDPD network to tackle the common issues of computational complexity and parameter management that often impede DPD implementations. By employing high-level synthesis (HLS) for the linear layer within the EdgeDRNN model, the performance metrics can be improved. Through our research, we estimate the performance of the combined system, which has a latency of 6.50 microseconds and a throughput of 0.26 giga-operation per second. The study highlights EdgeDRNN's performance and scalability, particularly in its ability to adapt to future wireless standards like 5G. However, the research also identifies some limitations on the performance, which sets a stage for future exploration. By designing and optimizing a new hardware accelerator for DPD, this work enables ultra-high-speed DPD solutions for next-generation wireless communication systems.

Contents

Preface ii

Abstract iii

1 Introduction 1
- 1.1 Problem Statement & Research Question . . . 1
  - 1.1.1 Problem Statement . . . 1
  - 1.1.2 Research Question . . . 2
- 1.2 Thesis Contribution . . . 2
- 1.3 Thesis Outline . . . 2

2 Background and Related Work 3
- 2.1 DPD & OpenDPD . . . 3
  - 2.1.1 Non-linearity of the Power Amplifiers . . . 3
  - 2.1.2 Digital Pre-Distortion . . . 3
  - 2.1.3 OpenDPD . . . 4
- 2.2 Neural Network Architectures . . . 6
  - 2.2.1 Linear Layer . . . 6
  - 2.2.2 Recurrent Neural Network . . . 7
- 2.3 EdgeDRNN . . . 9
  - 2.3.1 EdgeDRNN Software . . . 9
  - 2.3.2 EdgeDRNN Hardware . . . 10

3 Methods 12
- 3.1 OpenDPD Parameters and Data Export . . . 12
- 3.2 High-Level Synthesis of Linear Layer . . . 13
  - 3.2.1 High-Level Synthesis Design . . . 13
  - 3.2.2 High-level Synthesis C Simulation Testbench . . . 14
  - 3.2.3 High-Level Synthesis Behavioural Simulation testbench . . . 15
- 3.3 Complete Hardware Design . . . 15

4 Results 18
- 4.1 Experimental Setup . . . 18
- 4.2 Performance of OpenDPD Network Applied in EdgeDRNN . . . 18
- 4.3 Performance of The High-level Synthesis Linear Layer Block . . . 19
  - 4.3.1 C Validation Results . . . 19
  - 4.3.2 C Synthesis Results . . . 20
  - 4.3.3 C/RTL Cosimulation . . . 23
  - 4.3.4 RTL Exportation . . . 24
  - 4.3.5 Behavioral Simulation Results . . . 24
- 4.4 Combined Performance Estimation . . . 27

5 Conclusions 28
- 5.1 Conclusion . . . 28
- 5.2 Limitation . . . 28
- 5.3 Outlook . . . 28

A Appendix1 30

List of Figures

2.1 DPD concept 4
2.2 OpenDPD concept [1] 4
2.3 Data acquisition and Pre-processing [1] 5
2.4 PA modeling [1] 5
2.5 DPD Learning [1] 6
2.6 Linear Layer concept 7
2.7 GRU basic concept [2] 8
2.8 EdgeDRNN software basic concept 9
2.9 EdgeDRNN Step1 Model Pretrain 10
2.10 EdgeDRNN hardware accelerator concept 11

3.1 EdgeDRNN Step3 Export Paramaters 13
3.2 HLS concept 14
3.3 HLS C testbench concept 15
3.4 EdgeDRNN hardware testbench concept 16
3.5 EdgeDRNN hardware testbench 17
3.6 Hardware platform testbench with HLS 17

4.1 OpenDPD network applied in EdgeDRNN performance 19
4.2 C validation report of the HLS block 20
4.3 C/RTL Cosimulation report 24
4.4 HLS testbench simulation result 25
4.5 HLS testbench simulation result (zoomed) 25
4.6 HLS testbench continuous input simulation result 26
4.7 Linear layer CPU Result 27

List of Tables

3.1 EdgeDRNN hardware module function 16

4.1 OpenDPD performance display 19
4.2 C simulation data 20
4.3 C synthesis Solution 1 synthesis report 21
4.4 C synthesis Solution 2 synthesis report 22
4.5 C synthesis Solution 3 synthesis report 23
4.6 Solution comparison table 23
4.7 HLS performance display 26
4.8 HLS behavioral simulation data 26
4.9 Combined Performance estimation 27

1 Introduction

As wireless communication systems continue to evolve, especially with the emergence of 5G networks, the demand for efficient and robust digital pre-distortion (DPD) techniques has never been more significant. Power amplifiers (PAs) play a crucial role in these systems, but their inherent nonlinearities often lead to substantial in-band distortion and out-of-band spurious emissions. Addressing these issues is critical for enhancing the performance and linearity of PAs, which is where the linearization technique, such as feedforward, comes into play. Digital pre-distortion is a kind of linearization technique used to compensate for the nonlinear behavior of power amplifiers. The overall system can achieve a more linear output by applying a pre-distortion function that is the inverse of the PA's distortion. The DPD has been proven to be a better method to improve the linearity of the PA than feedforward methods [3]. However, traditional DPD methods face several challenges, including high computational complexity, many parameters, and the need for extensive online feature extraction. These challenges hinder the practical implementation of DPD in hardware, particularly in cost-efficient and energy-efficient designs.

Previously, the Generalized Memory Polynomial (GMP) model was the leading approach for the implementation of DPD [4]. Based on the Volterra series and including variants like the parallel Hammerstein model, GMP has shown good performance in many applications [5] [6]. Nonetheless, as the demands on wireless systems grow, GMP's ability to accurately model PA nonlinearity is being stretched [7], particularly with high peak-to-average power ratio and wideband signals [8]. To overcome these challenges, the application of neural networks has emerged as a promising alternative due to their strong capability to model complex nonlinear functions.

This thesis explores the use of the EdgeDRNN hardware accelerator [9] to improve DPD performance. In this thesis, the EdgeDRNN applies the OpenDPD network and employs high-level synthesis (HLS) for the linear layer to tackle the computational and parameter management issues commonly associated with DPD. This research on EdgeDRNN for DPD applications aims to deliver superior performance and scalability, making it well-suited for next-generation wireless communication systems.

1.1. Problem Statement & Research Question

1.1.1. Problem Statement

In wireless communication, radio frequency power amplifiers are crucial but often generate in-band distortion and out-of-band spurious emissions, mainly when operating in the saturation region. Digital pre-distortion techniques mitigate these nonlinearities and improve the linearity of the PA's output. Despite advances in DPD methods, several significant challenges remain, particularly in scalability, performance, and compatibility with emerging hardware frameworks. The use of neural networks for DPD has shown promise, but further research is required to adapt these models for efficient use within hardware accelerators like EdgeDRNN. This thesis is focusing on solving the following fundamental problems:

First, the neural network parameters from existing DPD models, such as OpenDPD, are not proven to be compatible with EdgeDRNN, which requires further research and testing to ensure that they can be adapted to EdgeDRNN.

Second, Developing efficient linear layer implementations in HLS is essential for optimizing DPD performance, but achieving a balance between computational efficiency, accuracy, and resource usage remains a challenge.

Third, different DPD components, such as the linear layer HLS block and the RNN of OpenDPD, need to be integrated effectively to explore potential performance improvements and synergies.

1.1.2. Research Question

The thesis mainly focuses on the following questions.

1. How can the parameters of the OpenDPD network be effectively exported to facilitate the implementation of the EdgeDRNN model?
2. What is the performance of an FPGA-based DPD hardware accelerator using EdgeDRNN and a Xilinx Vivado High-Level Synthesis (HLS)-generated linear layer accelerator?
3. How can the built linear layer HLS block be added to the RNN accelerator, and what is the performance of the combined system?

1.2. Thesis Contribution

This thesis provides an in-depth evaluation of EdgeDRNN for digital pre-distortion and makes several key contributions:

1. Neural network parameters generated from the OpenDPD network are extracted and adapted for use within the EdgeDRNN hardware accelerator. This process ensures that the performance characteristics of the original model are preserved while making it compatible with EdgeDRNN.
2. A linear layer HLS block is developed and integrated into the EdgeDRNN to generate the final outputs of the OpenDPD. This block is tested to evaluate its computational efficiency, accuracy, and hardware resource usage to optimize its performance for DPD applications.
3. The linear layer HLS block and the OpenDPD network's performance are thoroughly compared. The study estimates the potential improvements and synergies when these two components are combined within the EdgeDRNN hardware accelerator.

1.3. Thesis Outline

The rest of the parts of the thesis are organized as follows.

Chapter 2 provides an overview of the project's background. Section 2.1 begins by explaining the need for digital pre-distortion technology. Then, it covers the definition and fundamental concepts of DPD. Section 2.1.3 delves into the specifics of the OpenDPD software. Section 2.2 introduces the neural network employed in the project, detailing both the linear layer and the recurrent neural layer. Finally, Section 2.3 discusses and presents EdgeDRNN software and hardware.

Chapter 3 outlines the methods employed to assess EdgeDRNN's performance in digital pre-distortion. Section 3.1 details the process of exporting network parameters and handling the input and output data for testing. Section 3.2 presents and explains the design of the high-level synthesis block for the linear layer. Lastly, Section 3.3 describes the complete hardware platform used for the testing.

Chapter 4 presents the results obtained using the previously introduced method. Section 4.2 discusses the performance of the OpenDPD network when applied to EdgeDRNN. Section 4.3 provides an analysis of the C simulation results, C synthesis outcomes, and behavioral simulation results. Finally, section 4.4 estimates the performance of the combined system and explains the methodology used for this estimation.

Chapter 5 presents a comprehensive overview of the outcomes and implications of our research on the EdgeDRNN model for digital pre-distortion applications. While the model has shown promising results, some limitations and areas require improvement, which we aim to address through future research directions.

2

Background and Related Work

In this chapter, the background of the project is introduced. Section 2.1 first explains why the DPD technique is necessary. Then, the definition and basic knowledge of DPD is shown. The detailed software of OpenDPD is introduced in section 2.1.3. The Neural network used in the project is introduced in section 2.2 containing the linear layer and the Recurrent Neural Layer. In section 2.3, the EdgeDRNN software and hardware are discussed and shown.

2.1. DPD & OpenDPD

2.1.1. Non-linearity of the Power Amplifiers

Power amplifiers (PAs) are driven closer to their saturation regions to maximize efficiency. However, this will cause a growth of the spectrum out of the allocated channel. This unnecessary growth reflects the non-linear behavior introduced near the operating performance [10]. The non-linear behavior of the PA will cause intermodulation distortion when processing the input signal. In order to mitigate the intermodulation distortion, linearization techniques such as digital pre-distortion are used.

2.1.2. Digital Pre-Distortion

Digital pre-distortion (DPD) is a signal-processing technique utilized in radio frequency (RF) power amplifiers to enhance linearity. DPD works by preemptively distorting the input signal so that, when it passes through the non-linear amplifier, the distortions introduced by the amplifier are effectively canceled out [11]. This results in a more linear output, allowing more efficient use of the amplifier's power without compromising signal quality [12]. This pre-compensation of non-linear effects aims to linearize the system as a whole.

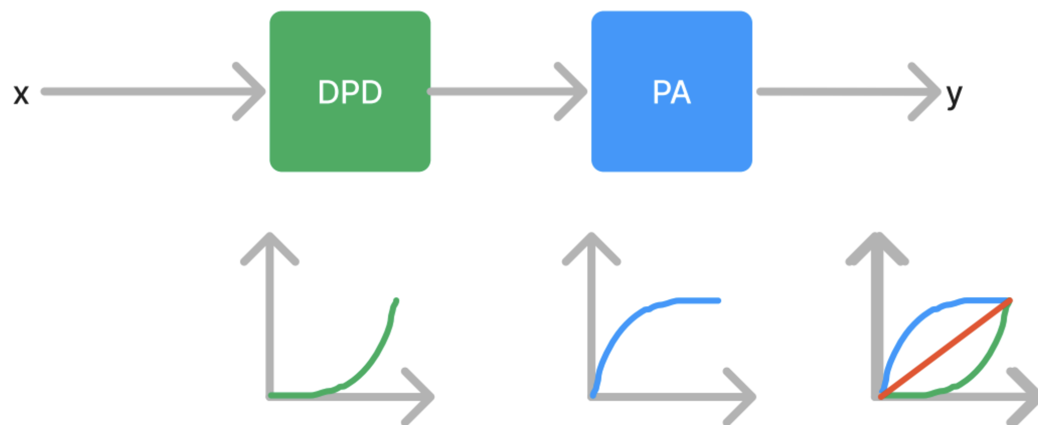


Figure 2.1: DPD concept

Figure 2.1 shows the concept of DPD. A DPD block is added before the PA, which provides an inverse of the PA curve [13]. Ideally, by adding the DPD block, the output of the PA could be linear. In a practical situation, the better the inverse curve provided by DPD, the more linear the output result. Several applications for the DPD technique already exist, such as High-symbol-rate coherent optical transceivers [14] and compensating for the Mach-Zehnder modulator in intensity modulation and direct detection optical systems [15].

2.1.3. OpenDPD

OpenDPD is an open-source end-to-end learning framework designed for wideband PA modeling and DPD [1]. Compared with traditional DPD methods, OpenDPD establishes a standardized benchmark for Deep Neural network-based DPD models, simplifying DPD design exploration and ensuring consistent testing conditions. As shown in Figure 2.2, OpenDPD contains three main parts.

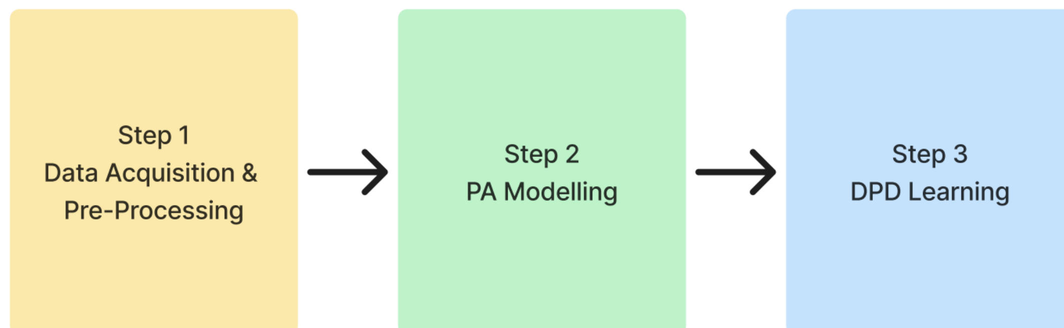


Figure 2.2: OpenDPD concept [1]

The first part is the data acquisition and pre-processing. In this part, an input signal is generated at first and passed to a Digital Analog Converter (DAC) to transfer into an analog signal for further processing. Then, the transferred signal is sent to the PA for amplification. The amplified signal may undergo attenuation to adjust its power level before transmission. After that, an Analog Digital Converter (ADC) transfers the signal back to digital. Finally, the transmitted signal is framed to organize it into manageable units for further processing.

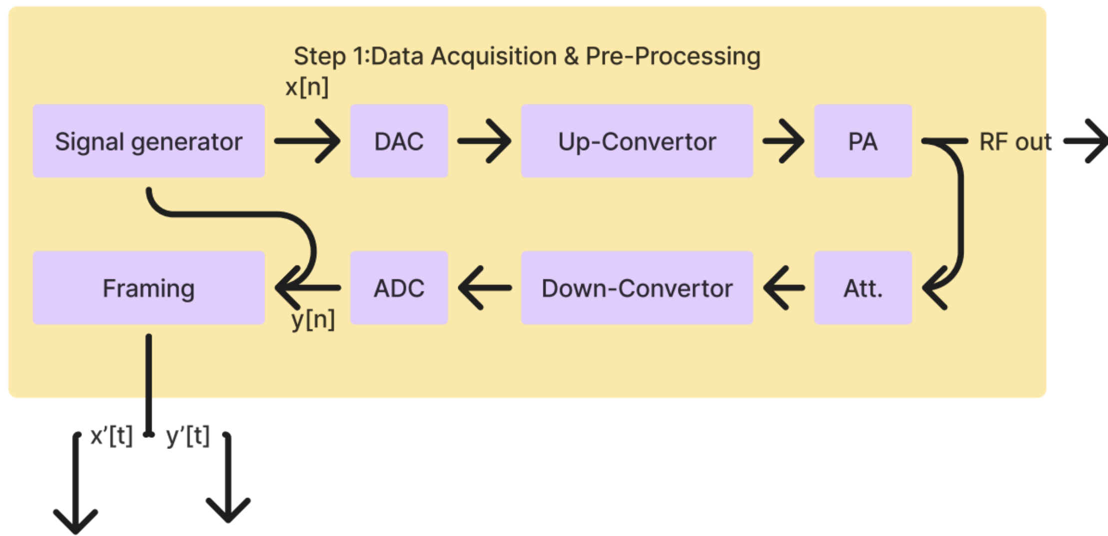


Figure 2.3: Data acquisition and Pre-processing [1]

The second part is the PA modeling. At the beginning, the framed input and the corresponding framed output are collected from the first part. The PA model is trained using a sequence-to-sequence learning approach. This method involves learning the mapping from a sequence of inputs to a sequence of outputs. The training process involves Back-Propagation Through Time, which is a technique used to train recurrent neural networks by unfolding them over time. This allows the model to learn from sequences of data. The goal during training is to reduce the Mean Squared Error (MSE) loss function. This function quantifies the squared difference between the PA model's predicted output and the target output. After calculating MSE loss and backpropagation, an optimization algorithm called ADAM [16] helps to get the optimal parameters. The training process aims to improve the parameters of the behavioral PA model to predict the output based on the input sequences accurately.

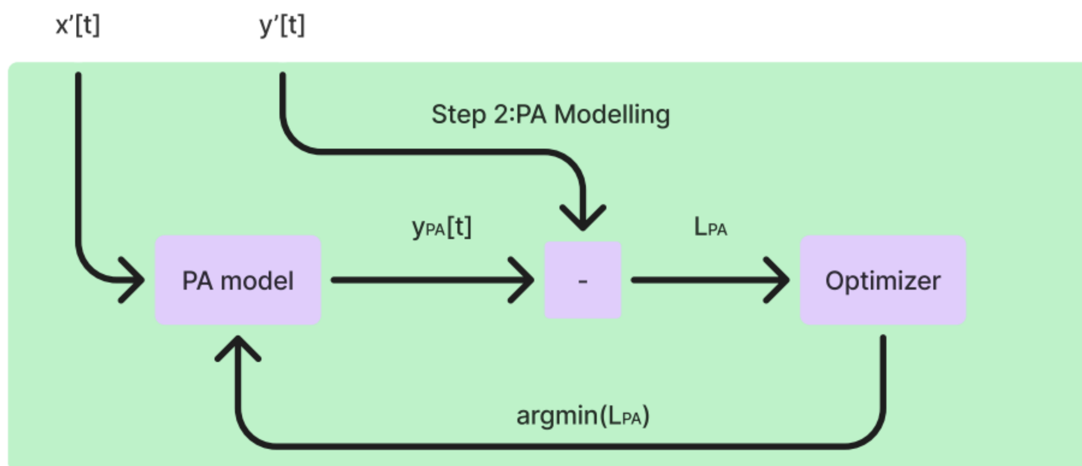


Figure 2.4: PA modeling [1]

The third step is DPD Learning. Before the pre-trained PA model, a DPD model is cascaded. The parameters of the PA model trained in the second part are first frozen. The cascaded model combines the DPD model with the PA model to account for the non-linearities introduced by the power amplifier. Through the learning process, the framed inputs collected from the first step are fed through the cascaded model and executed back-propagation across both the PA model and the DPD model to minimize the loss function. The loss function measures the squared difference between the predicted

output of the cascaded model and the target gain of the DPD-PA system. After the learning process, the output of the DPD model converges to the ideal pre-distorted PA inputs, which are used to drive the power amplifier to achieve linearization.

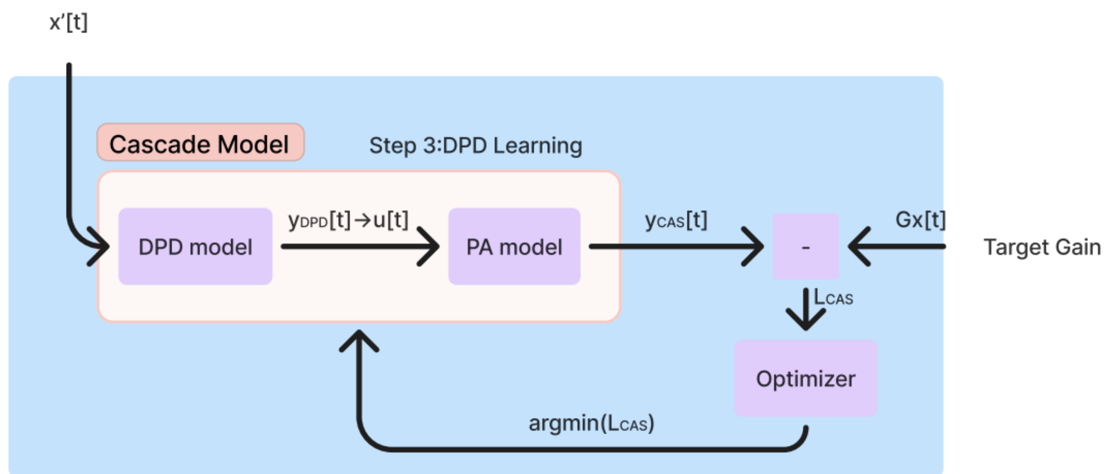


Figure 2.5: DPD Learning [1]

2.2. Neural Network Architectures

2.2.1. Linear Layer

The linear layer, also called a fully connected layer, is a fundamental part of the architecture of neural networks. As its name is called a fully connected layer, each neuron in the previous layer is connected to every neuron in the subsequent layer [17]. The connection of each neuron can be mathematically presented as $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ where $\mathbf{W}$ is a weight matrix, $\mathbf{x}$ is the input vector, $\mathbf{b}$ is a bias vector and $\mathbf{y}$ is the output vector.

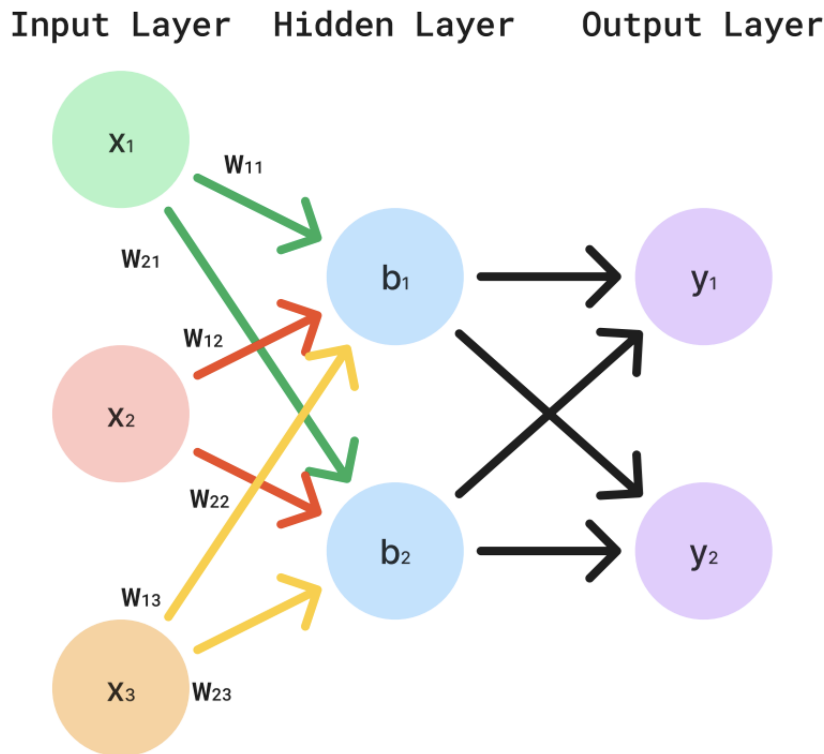


Figure 2.6: Linear Layer concept

As shown in figure 2.6, an example shows a linear layer with three inputs and two outputs.

$$Input : \mathbf{x} = [x_1, x_2, x_3]$$

$$Weights : \mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

$$Biases : \mathbf{b} = [b_1, b_2]$$

The output can be computed as

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$= \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2 \end{bmatrix}$$

The linear layer can combine the features extracted from the previous layer. Also, with the help of the linear layer, the dimension of the data can be changed to expand or reduce the feature space [18].

2.2.2. Recurrent Neural Network

The recurrent neural network(RNN) is a special kind of neural network designed to process sequential data. Compared with traditional feed-forward neural networks, RNNs can capture data from precious steps, which allows them to be effective for tasks involving sequences. The critical characteristic of RNN is its sequential processing [19]. It can process sequences one element at a time and maintain a hidden state that can capture information about the sequences. The secret state is updated at each time step and sends information forward through the sequence. The weights used across all steps are the same, which makes RNNs gain higher efficiency when learning from sequences with different lengths.

The mathematical formulation of RNN is shown as follows [20]:
At time t, the output $\mathbf{y_t}$ can be calculated as

$$\mathbf{y_t} = \mathbf{W_{yh}}\mathbf{h_t} + \mathbf{b_y}$$

where $\mathbf{h_t}$ is the hidden state at time step t.

$$\mathbf{h_t} = \phi(\mathbf{W_{hx}}\mathbf{x_t} + \mathbf{W_{hh}}\mathbf{x_{t-1}} + \mathbf{b_h})$$

with ϕ is a non-linear activation function such as ReLU or tanh, $\mathbf{x}$ is the input vector, $\mathbf{W}$ are the weight matrices and $\mathbf{b}$ are the biases vectors [21].

GRU
Gated Recurrent Unit (GRU) is a type of RNN architecture that aims to solve the gradient problem encountered in traditional RNNs [2]. Like Long Short-Term Memory (LSTM) networks [22], the Gated Recurrent Unit features gating mechanisms that manage the flow of information, helping to reduce the risk of the vanishing gradient issue. Since GRUs have fewer gates than LSTMs, they require fewer parameters, which can result in improved model accuracy [23]. By using gating mechanisms, GRU can more effectively capture dependencies in sequential data. For the gating mechanisms, the gating units control the flow of information within the network. They consist of a reset gate and an update gate. At each time step, these gates regulate the information flow by deciding what information to keep or discard. GRU network can describe a complex system such as forecasting short-term photovoltaic power [24] and speech emotion recognition [25] quite well because of its specially designed structure [26].

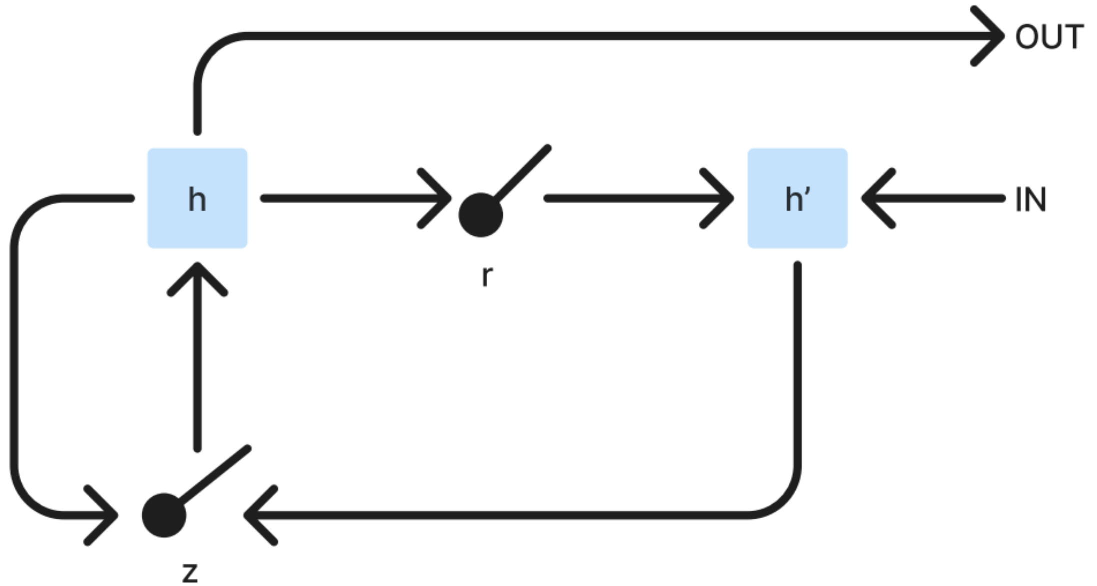


Figure 2.7: GRU basic concept [2]

As shown in figure 2.7, a basic concept of GRU is shown. The switch r is the reset gate, and the z is the update gate. h is the activation and h' is the candidate activation.

DGRU
Deep Gated Recurrent Unit (DGRU) is a variation of GRU that combines GRU concepts with deep learning techniques [27]. A DGRU is obtained by stacking multiple GRU layers on top of each other. Through this stacking, the model can capture more patterns and dependencies in the data over the various layers. With more layers, DGRU can handle the model with complex functions and dependencies. Besides, it can handle more complex tasks such as profit prediction in financial accounting information systems [28] and high-speed links [29]. In this project, the PA model is trained with a DGRU network.

DeltaGRU

DeltaGRU is a variation of GRU that applies the delta network algorithm to GRU [30]. The algorithm introduces the delta threshold for input and hidden unit activations. By modifying the threshold, the sparsity levels in activation vectors can be increased, which leads to the reduction of weight memory access and computational requirements. The modification can enhance the efficiency of the RNN model. Compared with traditional GRU models, Delta GRU has fewer parameters, making it more efficient and simpler [31].

2.3. EdgeDRNN

EdgeDRNN is a lightweight GRU-based RNN hardware accelerator focusing on the optimization of low-latency edge inference with a batch size of 1 [30] [9]. It utilizes DeltaGRU to efficiently compute large and multi-layer RNNs whose weights are stored in off-chip DRAM. This design allows EdgeDRNN to deliver high throughput performance while freeing up CPU cycles for other tasks on the system-on-chip. As a result, EdgeDRNN provides a flexible, cost-effective, and high-performance solution for various sizes of gated RNN networks, particularly emphasizing real-time applications on edge devices.

2.3.1. EdgeDRNN Software

The EdgeDRNN network training software consists of 3 main steps. Figure 2.8 shows the basic concept of the RNN training code of EdgeDRNN.

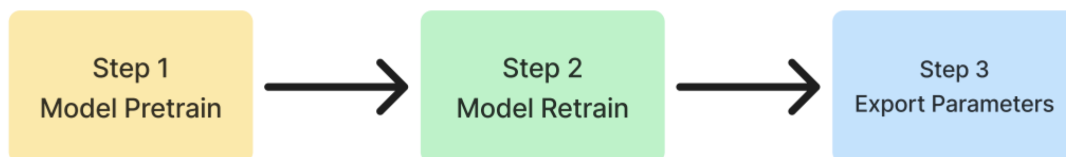


Figure 2.8: EdgeDRNN software basic concept

The first step is the Model Pretrain. As shown in figure 2.9, the step can be separated into nine main parts. The first part is project initialization. In this step, the Project class is instantiated, including precision settings, hardware info, and parsing command-line arguments. The command-line arguments are then parsed to set various hyperparameters and configurations for dataset processing, training, and model hyperparameters. The dataloader is prepared for training, validation, and testing in the next step. The dataloader is prepared by collecting data from the data set in CSV format. The loggers are then set up for logging training metrics and hyperparameters. Various callbacks are to be created for use during training, such as model checkpointing, learning rate monitoring, and custom logging. The pytorch trainer can be instantiated with configurations such as the number of epochs, GPUs, accelerator, loggers, and callbacks. The last two steps are training and testing the model.

The second step is Model Retrain. The module is almost identical to the first step except for the model preparation. In the second step, the model preparation will load the model trained in step one. Besides, the dataloader contains all data used for pretrain, retrain, and export. In the second step, the data given to the pre-trained model is different from the previously used in step 1. The retraining step is an essential part of the model development because feeding more data will improve its accuracy and generalization capability. Also, the retraining step provides an opportunity to adjust the hyperparameters to be more stable and get better optimization and performance.

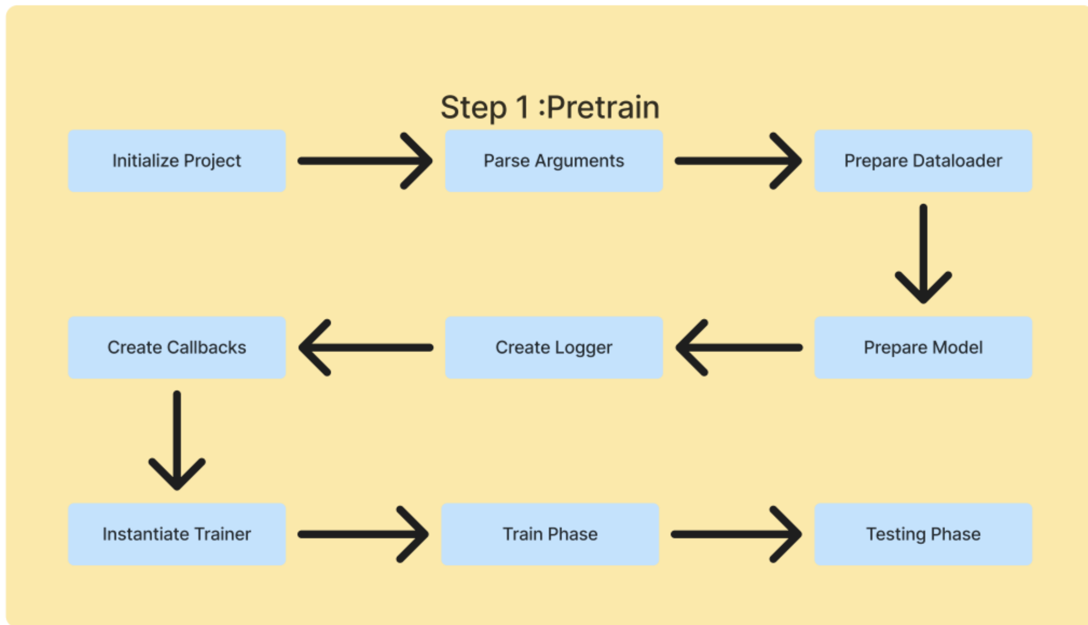


Figure 2.9: EdgeDRNN Step1 Model Pretrain

The third step is the parameter export. In this step, the parameters of the best train model are exported. A data set is also sent to the model, and the output of the RNN and fc layer is collected and exported. The export data parameters and data will be used in the hardware test. Chapter 3.1 will provide a detailed introduction to the export part.

2.3.2. EdgeDRNN Hardware

The EdgeDRNN hardware structure is designed to optimize the performance of RNNs' edge computing applications [9]. Its hardware architecture has several critical vital components. The processing elements perform computations. By using a modest number of processing elements, power consumption can be reduced while maintaining high throughput. For the memory architecture, the weights of EdgeDRNN are stored in off-chip DRAM, which is inexpensive. This kind of design enables the handling of large multi-layer RNNs, which requires a considerable amount of memory resources. After the optimization is applied to this architecture, access to the DRAM is minimized, and a reduction in weight memory access is achieved by a factor of up to 10x through sparse updates. A delta network algorithm is applied to the EdgeDRNN. This algorithm capitalizes on the temporal sparsity of activation state vectors in a network [32], allowing the accelerator to focus on significant changes in the input data rather than processing every input in total, which results in the reduction of memory access and computational overhead.

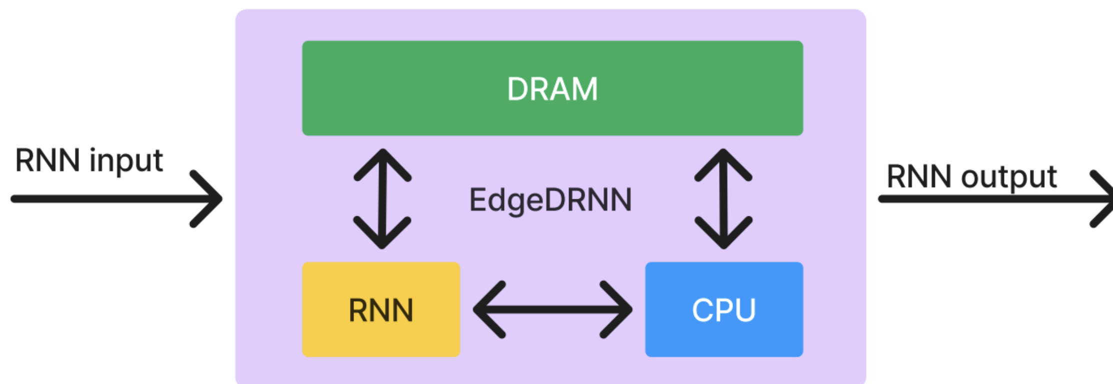


Figure 2.10: EdgeDRNN hardware accelerator concept

The EdgeDRNN can be integrated into a system-on-chip environment, which can allow the CPU to handle other tasks, such as feature extraction and I/O management. At the same time, the EdgeDRNN focuses on RNN inference. This integration helps in optimizing overall system performance and resource utilization. EdgeDRNN is designed for low-latency performance, achieving latency comparable to that of high-power GPUs while operating at a low power cost. The architecture is capable of updating a 5 million parameter 2-layer GRU-RNN in approximately 0.5 ms, making it suitable for real-time applications [33]. The EdgeDRNN architecture is designed to be flexible, which allows the delta thresholds to be adjusted dynamically. This feature enables a trade-off between latency and accuracy, accommodating various application requirements. The architecture can also be implemented on small-footprint FPGAs, making it suitable for embedded systems.

3 Methods

This chapter explains the methods used to evaluate EdgeDRNN's performance on digital pre-distortion. Section 3.1 introduces the exportation of the network parameters and the testing input and output data. In section 3.2, the design of the High-level synthesis block of the linear layer is shown and explained. Finally, section 3.3 describes the complete hardware platform for testing.

3.1. OpenDPD Parameters and Data Export

To test the network trained for OpenDPD on the EdgeDRNN hardware platform, the network parameters and a testing set must be exported in the EdgeDRNN format. So, an OpenDPD export module is built in the EdgeDRNN format. The concept of the original EdgeDRNN export module is shown in Figure 3.1. The first two parts are the initialization of the project and the EdgeDRNN, which can set up the directories for output, hardware information, and argument parsing. Also, directories for output and initializing necessary components are prepared in these steps. Then, the dataloader and the pre-trained DPD model are prepared. The data pulled from the dataloader used for this module is also different from those used for pre-train of retrain. The model selected this time is the only model that has the best performance among all the retrained models. The next step is the exportation of the parameters of the selected model. The parameters are exported after a quantization module. Because the EdgeDRNN focuses on the performance of the RNN, so only the parameters of the RNN are quantized, and the linear layer parameters are not. This time, two files will be created. File `edgedrnn_params.c` contains the RNN parameters and the weights and biases of the following Linear layer. The `edgedrnn_params.h` will contain the sizes of the three network parameters and the size of the network. It also contains the quantization parameters. The next part is feeding the prepared test data to the selected and exported network, and the output of the RNN and the linear layer are collected separately. The two outputs and the input data set are quantized and then exported in `edgedrnn_test.c`. And in the `edgedrnn_test.h` file, the sizes of the three sets are declared. The outputs of the RNN and the linear layer are called the golden results, which are used later in the hardware platform to judge whether the hardware accelerator works correctly or not.

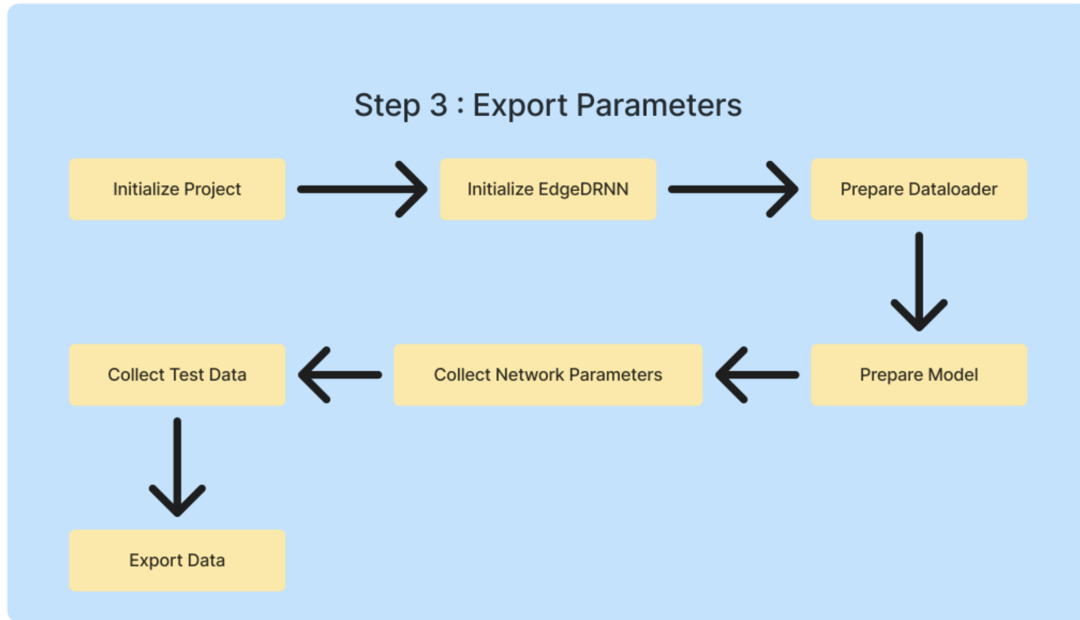


Figure 3.1: EdgeDRNN Step3 Export Paramaters

Compared with the original module, the newly built module has several differences. First, the previous dataloader is removed, and the OpenDPD data set is used in this module to collect test values. The selected method for the model was also changed in order to scan and identify the DPD model successfully.

The most significant modification of the module is the quantization part. The quantization parameters of the linear layer part have been set. The quantization method involves applying a scaling factor and then rounding down to a fixed-point number. The scaling factor is separated into two: qi specifies the number of bits used to represent the integer part of the number. This determines the range of integer values the fixed-point number can represent. qf specifies the number of bits used to define the fractional part of the number. This determines the precision of the fractional part. For example, if -0.0859 is quantized with $qi = 8$ and $qf = 8$. The quantized value should be :

$$Quantizedvalue = Round(-0.0859 * 2^{qf} = -0.0859 * 2^8 = -21.9904) = -22$$

and the clipping value is calculated by

$$2^{qi+qf-1} = 2^{15} = 32768$$

so the clipping range is -32768 to 32767.

For the RNN parameters, the $a_{qi} = 8$ and the $a_{qf} = 8$; for the linear layer, the $w_{qi} = 1$ and the $w_{qf} = 7$. Due to the mathematical formula of linear layer $\mathbf{y} = \mathbf{Wx} + \mathbf{b}$, if $\mathbf{W}$ is multiplied 2^7 and $\mathbf{x}$ multiplied 2^8, the scaling of $\mathbf{y}$ and $\mathbf{b}$ should be same and equal to 2^{15}. So, the factor $fc_{qf} = 15$ and $fc_{qi} = 9$ is set to avoid exceeding the clipping range.

3.2. High-Level Synthesis of Linear Layer

3.2.1. High-Level Synthesis Design

High-Level Synthesis is a method transforming high-level programming languages such as C or C++ into register-transfer level code, which can be used to configure FPGAs. In this project, a Linear layer HLS block using C++ is designed. The basic concept is shown in figure 3.2. The first part is structure definition. In this HLS module, two structures, `axis_t` and `axis_o`, are defined. `axis_t` is used to structure the input of the HLS module. It has two members, `data`, with a type of `ap_int<16*8>`, which means it can hold eight short values (16-bit). The other member is `last` with a type of `ap_unit<1>`, which means it has a one-bit wide for rating the TLast flag to show that the end of the streaming data is

sent. `axis_o` is used for the output and the only difference from `axis_t` is the size of `data` which is `ap_int<32*2>` which can carry 2 32-bit values. The second part is the parameters input. The weights and biases of the linear layer are then written inside the HLS block. The parameters are collected from the exportation files. Then is the input collection part. A for loop was created to separate the collected 128-bit value into eight short values and store them in a temporary array for further calculation. By getting all the required data, the output can then be calculated and stored in the temporary output, and the TLAST can be set up when the last value is transferred from the temporary output to the output port of the HLS module.

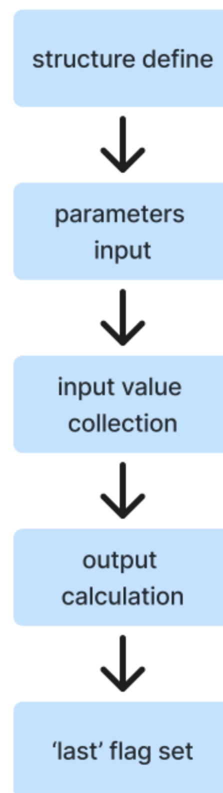


Figure 3.2: HLS concept

3.2.2. High-level Synthesis C Simulation Testbench

After the design of the linear layer HLS block is built, the following C simulation testbench is needed. This testbench is used not only for the C simulation but also for the C/RTL cosimulation. This testbench is built to test whether the hardware result of the HLS block is the same as the software results. The hardware result refers to the output of the HLS block by feeding a series of inputs. The software result refers to the production calculated in the testbench with the same input fed to the HLS. The design concept of the c simulation testbench is shown in figure 3.3.

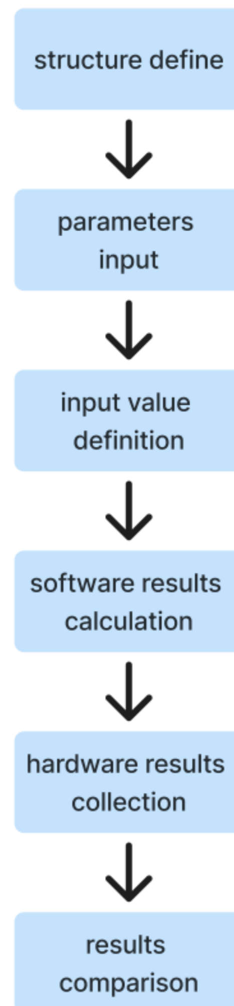


Figure 3.3: HLS C testbench concept

3.2.3. High-Level Synthesis Behavioural Simulation testbench

For the high-level synthesis (HLS) linear layer module, a testbench is created to verify the functionality of the module. This testbench, written in Verilog. It incorporates various components to test the ‘linear_layer‘ design under different conditions. Key signals, including the clock (‘ap_clk‘), reset (‘ap_rst_n‘), and data lines (‘inputs_TDATA‘, ‘outputs_TDATA‘), are declared with appropriate initial values and test sequences. The clock is generated with a 10-nanosecond period to simulate a 100 MHz frequency. The test sequence begins by initializing the inputs and applying a reset, followed by introducing test vectors to validate the module’s behavior. Output monitoring is performed using the `$monitor()` command to capture and display relevant signal changes over time. This setup ensures a comprehensive evaluation of the ‘linear_layer‘ module’s performance and functionality within the system. The testbench result will be shown in section 4.3.5.

3.3. Complete Hardware Design

The block design of the EdgeDRNN hardware testbench is shown in figure 3.5, and its brief concept diagram is shown in figure 3.6. The functions of the modules used are listed in table 3.1. In this design, data is first prepared and stored in the DDR memory, which is accessible by the Zynq PS. The Zynq PS initiates a data transfer by configuring the `AXI DMA`. The DMA controller reads data from DDR and transfers it over the AXI stream interface. The `axi_datamover` moves the data between memory-mapped and stream interfaces. The `axis_dwidth_converter` adjusts the data width to

match the requirements of the `edgedrnn_wrapper`. The data is sent to the `edgedrnn_wrapper` via the AXI stream and processed there. The `edgedrnn_wrapper` produces the output data and sends it over the AXI stream interface. The `axis_dwidth_converter` adjusts the output data width, and the `axi_datamover` then moves the data from the stream interface back to a memory-mapped interface. The AXI DMA writes the processed data back to DDR memory. The Zynq PS reads the processed data from DDR memory and compares it against the golden RNN results collected from the exported data to verify correctness.

Module name	**Module function**
ZYNQ Processing System (PS)	This includes an ARM processor and interfaces with the programmable logic through AXI interfaces.
AXI DMA (Direct Memory Access)	Facilitates data transfer between the PS and PL without CPU intervention, allowing high-speed data movement.
AXI DataMover	Transfers data between memory-mapped and streaming interfaces.
AXI Stream Data Width Converter	Converts data widths between different AXI stream components.
AXI Interconnect	Connects multiple AXI masters and slaves, managing data flow.
EdgeDRNN Wrapper	Apply the EdgeDRNN operation

Table 3.1: EdgeDRNN hardware module function

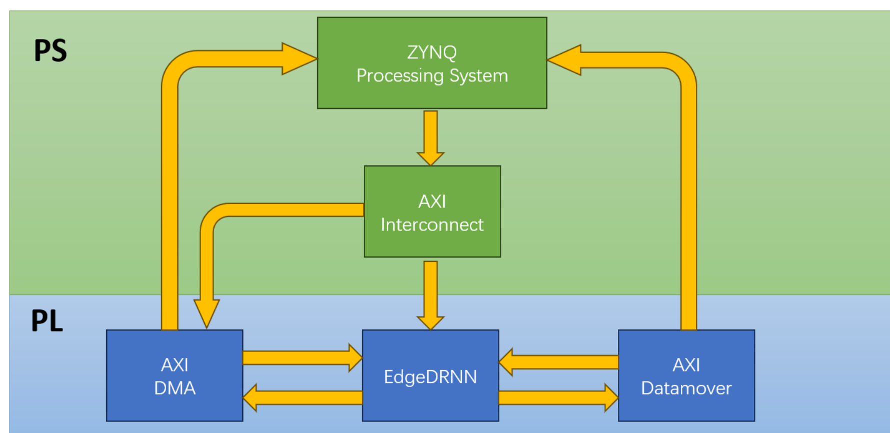


Figure 3.4: EdgeDRNN hardware testbench concept

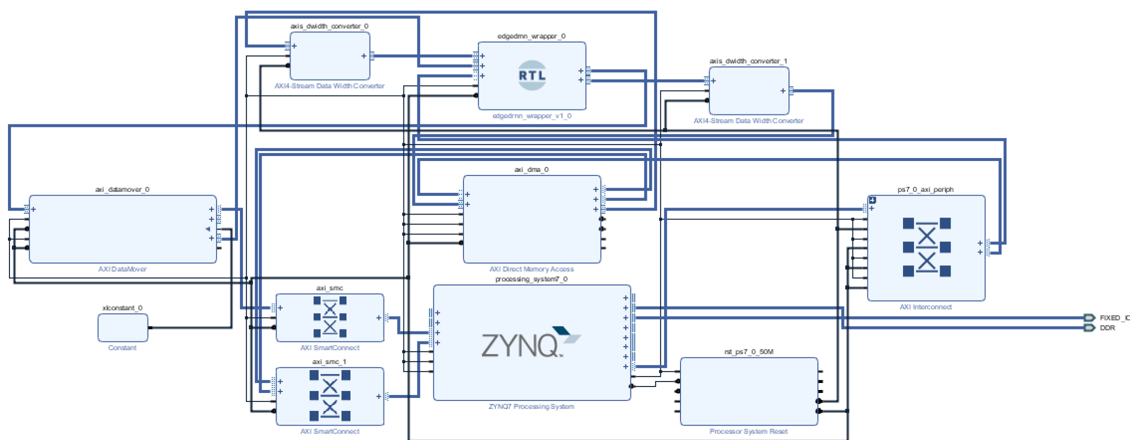


Figure 3.5: EdgeDRNN hardware testbench

The designed Linear layer HLS module is added after the `edgedrnn_wrapper`. The output width of the HLS module is 64-bit, which means the `axis_dwidth_converter` is not necessary now. The block design of the hardware platform with HLS is shown in figure 3.6. The golden results of the linear layer are used to judge whether the whole system works correctly or not.

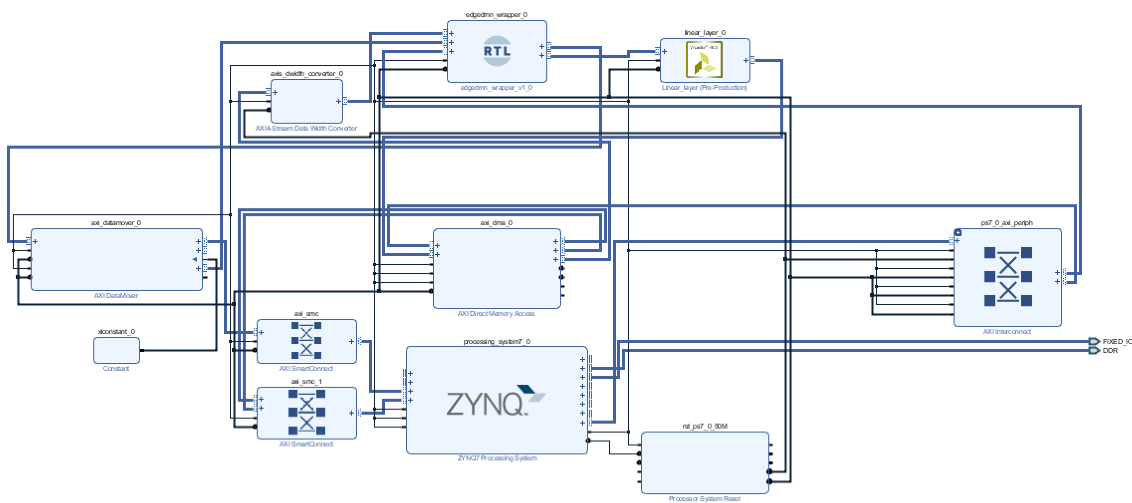


Figure 3.6: Hardware platform testbench with HLS

4 Results

This chapter shows the results obtained using the method introduced before. Section 4.2 shows the performance of the OpenDPD network applied in EdgeDRNN. In section 4.3, the C simulation results, C synthesis results, and the behavioral simulation results are all shown and analyzed. In section 4.4, the performance of the combined system is estimated, and the method used to make this estimation is explained.

4.1. Experimental Setup

The setup of the experiments in this thesis is introduced, and based on the configuration, the experiments can be reproduced. The dataset of paper [1] is used to train the PA and DPD models of the OpenDPD network. After the training and testing, the parameters can then be exported. The running environment of OpenDPD is also provided in this paper. A hardware test is necessary to test the performance of the OpenDPD network applied in EdgeDRNN. The Vivado test project used is provided in paper [9]. The Vivado version used is `2018.2`. The project is then run on a `AES-MINIZED-7Z007-G` FPGA board. For the build of the HLS module of the linear layer, the Vivado HLS version is `2018.2`. The part selection of the HLS module is `xc7z007sclg225-1`, and the set clock period is ten ns.

4.2. Performance of OpenDPD Network Applied in EdgeDRNN

From the OpenDPD export module, four files containing the model parameters and the test input and output data are generated. The files are then applied in the EdgeDRNN hardware platform introduced in section 3.3.

```
Problems  Tasks  Console  Properties  SDK Terminal
Connected to: Serial ( COM4, 115200, 0, 8 )

Connected to COM4 at 115200
Ops per Time Step          = 1632.000000

Total Time Steps           = 1000

Total Ops                  = 1632000.000000

Total Latency      (us)    = 6398.401367

Min Latency        (us)    = 5.556000

Max Latency        (us)    = 6.768000

Mean Latency       (us)    = 6.398401

Min Eff. Throughput  (GOp/s) = 0.241135

Max Eff. Throughput  (GOp/s) = 0.293737

Mean Eff. Throughput (GOp/s) = 0.255064

Benchmark Successful: RNN Outputs Correct!!!
```

Figure 4.1: OpenDPD network applied in EdgeDRNN performance

The hardware testing result of the OpenDPD is shown in figure 4.1, and the explanation of each data point is contained in table 4.1. The Latency metrics show the time efficiency of the operations. Lower latency implies faster operation execution, which is crucial for real-time applications. The throughput metrics measure the processing capability of the RNN accelerator. Higher throughput indicates the ability to process more operations per second, which benefits performance-intensive applications. Ops per Time Step and Total Ops provide a sense of the workload and computational intensity of the RNN benchmark.

	Values	Explanation
Ops per Time Step	1632.000000	The total operations are performed in each time step in the RNN.
Total Time Steps	1000	The total time steps the RNN ran during this benchmark.
Total Ops	1632000.000000	The total number of operations performed.
Total latency (us)	6398.401367	The total time taken to execute all the operations.
Min Latency (us)	5.556000	The minimum latency per operation.
Max Latency (us)	6.768000	The maximum latency per operation.
Mean Latency (us)	6.398401	The average latency per operation.
Min Eff. Throughput (GOp/s)	0.241135	The minimum effective throughput.
Max Eff. Throughput (GOp/s)	0.293737	The maximum effective throughput.
Mean Eff. Throughput (GOp/s)	0.255064	The average effective throughput.

Table 4.1: OpenDPD performance display

4.3. Performance of The High-level Synthesis Linear Layer Block

4.3.1. C Validation Results

The first step in building an HLS block is to run the C validation. C Validation ensures the C algorithm is performing the correct operation. The C testbench is constructed to compare the hardware results and software results. In this testbench, the used data is collected from the exported golden RNN value,

and the collected hardware and software results are all correct compared with the golden linear layer results, which are shown in detail in table 4.2. The C validation report is shown in figure 4.2.

Input:	-22	-50	33	20	16	-40	-7	-20
Weights:	-48	112	-32	-62	-91	-16	-73	22
Biases:	-512							
Desired output:	742							
Input:	-22	-50	33	20	16	-40	-7	-20
Weights:	104	55	-72	110	-67	47	-96	-41
Biases:	0							
Desired output:	9102							

Table 4.2: C simulation data

```
1  INFO: [SIM 2] *************** CSIM start ***************
2  INFO: [SIM 4] CSIM will launch GCC as the compiler.
3     Compiling ../../../../Linear_layer_tb.cpp in debug mode
4     Compiling ../../../../Linear_layer.cpp in debug mode
5     Generating csim.exe
6  Checking results...
7  hw_result[0]: 742
8  sw_result[0]: 742
9  hw_result[1]: 9102
10 sw_result[1]: 9102
11 Pass Test!
12 INFO: [SIM 1] CSim done with 0 errors.
13 INFO: [SIM 3] *************** CSIM finish ***************
14
```

Figure 4.2: C validation report of the HLS block

4.3.2. C Synthesis Results

The second step is to run the C synthesis, which adds RTL ports and associated I/O protocols to the C design. After running the C synthesis of the Linear layer HLS module, the synthesis report is shown in table 4.3. Through this report, it can be found that the target clock period is set to 10 ns. The Target Clock Period is the specified time interval for one clock cycle. This means that ideally, the circuit should complete one clock cycle every ten ns. The Estimated Clock Period is the actual time interval for one clock cycle as determined by the synthesis tool after analyzing the design. According to the report, the estimated clock period is 7.185 ns, which means, based on the current design, one clock cycle would take approximately 7.185 ns. The uncertainty is a measure of the possible variation or error margin in the estimated clock period. In this report, the uncertainty is 1.25 ns. This means that the actual clock period could vary by $\pm$1.25 ns around the estimated value.

Latency refers to the number of clock cycles required for a single input to pass through the entire design pipeline and produce an output. Interval is the number of clock cycles between the start of processing consecutive inputs. It defines how frequently new inputs can be fed into the pipeline. According to the report, both are 73, which is relatively high and requires optimization.

Solution 1					
Performance Estimates					
Timing (ns)					
	Clock	Target	Estimated	Uncertainty	
	ap_clk	10.00	7.185	1.25	
Latency (clock cycles)					
	Latency		Interval		Type
	min	max	min	max	
	73	73	73	73	none
Utilization Estimates					
	Name	BRAM_18K	DSP48E	FF	LUT
	Total	0	1	882	1695
	Available	100	66	28800	14400
	Utilization (%)	0	1	2	11
Interface					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	linear_layer	return value
ap_rst_n	in	1	ap_ctrl_hs	linear_layer	return value
ap_start	in	1	ap_ctrl_hs	linear_layer	return value
ap_done	out	1	ap_ctrl_hs	linear_layer	return value
ap_idle	out	1	ap_ctrl_hs	linear_layer	return value
ap_ready	out	1	ap_ctrl_hs	linear_layer	return value
inputs_TDATA	in	128	axis	inputs_V_data_V	pointer
inputs_TVALID	in	1	axis	inputs_V_data_V	pointer
inputs_TREADY	out	1	axis	inputs_V_data_V	pointer
inputs_TLAST	in	1	axis	inputs_V_data_V	pointer
outputs_TDATA	out	64	axis	outputs_V_data_V	pointer
outputs_TVALID	out	1	axis	outputs_V_data_V	pointer
outputs_TREADY	in	1	axis	outputs_V_data_V	pointer
outputs_TLAST	out	1	axis	outputs_V_data_V	pointer

Table 4.3: C synthesis Solution 1 synthesis report

The HLS pipeline is applied to reduce the system's latency and interval. The HLS pipeline directive is a feature in Vivado HLS that improves the performance of the hardware designs. The pipelining technique can overlap multiple structures. It allows different stages of various operations to be processed simultaneously, which can significantly enhance the throughput of a design, allowing the system to process new inputs at every clock cycle. By applying the HLS pipeline directive, solution two was created, and the synthesis report is shown in table 4.4. The module has a fixed latency of 5 clock cycles with a two-clock cycle interval, indicating predictable performance. At the same time, the estimated clock period is rising up to 10.283 ns. The clock cycle period is slightly above the target but within the uncertainty margin. Through the interfaces report, it can be found that 6 `ap` ports exist. The HLS block should work when receiving data from the `edgedrnn_wrapper`, which means it does not need AP control. So, an HLS interface directive is applied to set the AP control to none.

Solution 2					
Performance Estimates					
Timing (ns)					
	Clock	Target	Estimated	Uncertainty	
	ap_clk	10.00	10.283	1.25	
Latency (clock cycles)					
	Latency		Interval		Type
	min	max	min	max	
	5	5	2	2	none
Utilization Estimates					
	Name	BRAM_18K	DSP48E	FF	LUT
	Total	0	8	666	450
	Available	100	66	28800	14400
	Utilization (%)	0	12	2	3
Interface					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	linear_layer	return value
ap_rst_n	in	1	ap_ctrl_hs	linear_layer	return value
ap_start	in	1	ap_ctrl_hs	linear_layer	return value
ap_done	out	1	ap_ctrl_hs	linear_layer	return value
ap_idle	out	1	ap_ctrl_hs	linear_layer	return value
ap_ready	out	1	ap_ctrl_hs	linear_layer	return value
inputs_TDATA	in	128	axis	inputs_V_data_V	pointer
inputs_TVALID	in	1	axis	inputs_V_data_V	pointer
inputs_TREADY	out	1	axis	inputs_V_data_V	pointer
inputs_TLAST	in	1	axis	inputs_V_data_V	pointer
outputs_TDATA	out	64	axis	outputs_V_data_V	pointer
outputs_TVALID	out	1	axis	outputs_V_data_V	pointer
outputs_TREADY	in	1	axis	outputs_V_data_V	pointer
outputs_TLAST	out	1	axis	outputs_V_data_V	pointer

Table 4.4: C synthesis Solution 2 synthesis report

Solution 3 is then created, and the synthesis report is shown in table 4.5. The utilization estimation shows the resource utilization of the FPGA. It shows the percentage of each type of resource used out of the available resources on the target device.

Solution 3					
Performance Estimates					
Timing (ns)					
	Clock	Target	Estimated	Uncertainty	
	ap_clk	10.00	10.283	1.25	
Latency (clock cycles)					
	Latency		Interval		Type
	min	max	min	max	
	5	5	2	2	none
Utilization Estimates					
	Name	BRAM_18K	DSP48E	FF	LUT
	Total	0	8	665	439
	Available	100	66	28800	14400
	Utilization (%)	0	12	2	3
Interface					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	linear_layer	return value
ap_rst_n	in	1	ap_ctrl_hs	linear_layer	return value
inputs_TDATA	in	128	axis	inputs_V_data_V	pointer
inputs_TVALID	in	1	axis	inputs_V_data_V	pointer
inputs_TREADY	out	1	axis	inputs_V_data_V	pointer
inputs_TLAST	in	1	axis	inputs_V_data_V	pointer
outputs_TDATA	out	64	axis	outputs_V_data_V	pointer
outputs_TVALID	out	1	axis	outputs_V_data_V	pointer
outputs_TREADY	in	1	axis	outputs_V_data_V	pointer
outputs_TLAST	out	1	axis	outputs_V_data_V	pointer

Table 4.5: C synthesis Solution 3 synthesis report

Also, a comparison report is shown in table 4.6. Through the report, it can be found that after applying the HLS pipeline and setting the AP control to none, the estimated clock period slightly exceeded the target while the latency and interval significantly decreased, which means it is more efficient in terms of throughput and processing speed. Also, it is optimized for lower latency and higher throughput, utilizing more DSP resources but efficiently managing FFs and LUTs.

Performance Estimates					
Timing(ns)	Clock		Solution1	Solution2	Solution3
	ap_clk	Target	10.00	10.00	10.00
		Estimated	7.185	10.283	10.283
Latency (clock cycles)			Solution1	Solution2	Solution3
	Latency	min	73	5	5
		max	73	5	5
	Interval	min	73	2	2
		max	73	2	2
Utilization Estimates					
			Solution1	Solution2	Solution3
BRAM_18K			0	0	0
DSP48E			1	8	8
FF			822	666	665
LUT			1695	450	439

Table 4.6: Solution comparison table

4.3.3. C/RTL Cosimulation

C/RTL co-simulation in HLS is a crucial step in the design and verification of digital circuits. It involves simulating both the high-level C/C++ model and the low-level register transfer level model together to

ensure that they produce the same results and meet the desired specifications. Through this cosimulation, a C/RTL cosimulation report is shown in figure 4.3. In this cosimulation test report, it can be found that The built HLS block has passed the cosimulation successfully.

Cosimulation Report for 'linear_layer'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	6	6	6	NA	NA	NA

Figure 4.3: C/RTL Cosimulation report

4.3.4. RTL Exportation

C/RTL co-simulation in HLS is a crucial step in the design and verification of digital circuits. It takes the high-level C++ code that describes the algorithm and functionality and converts it into an RTL design in Verilog code. Then the generated Verilog code is packaged and exported as an ip block for the following used in Vivado lab project behavioural simulation.

4.3.5. Behavioral Simulation Results

Once the High-Level Synthesis (HLS) IP block is exported, a Verilog-based testbench is constructed to establish the input-output relationship. The results from this simulation are illustrated in figure 4.4. Upon initialization, when the signal `inputs_TVALID` is set to 1, the `inputs_TDATA` displays the input values. These 32-bit hexadecimal values correspond to the exported golden RNN values, consistent with those used in the C validation process. The resulting output is captured in `outputs_TDATA`, with a detailed view provided in figure 4.5. The hexadecimal output values of $0x02E6$ and $0x238E$ convert to decimal values of 742 and 9102, respectively, which match the results obtained from the C validation, confirming the accuracy of the HLS IP block.

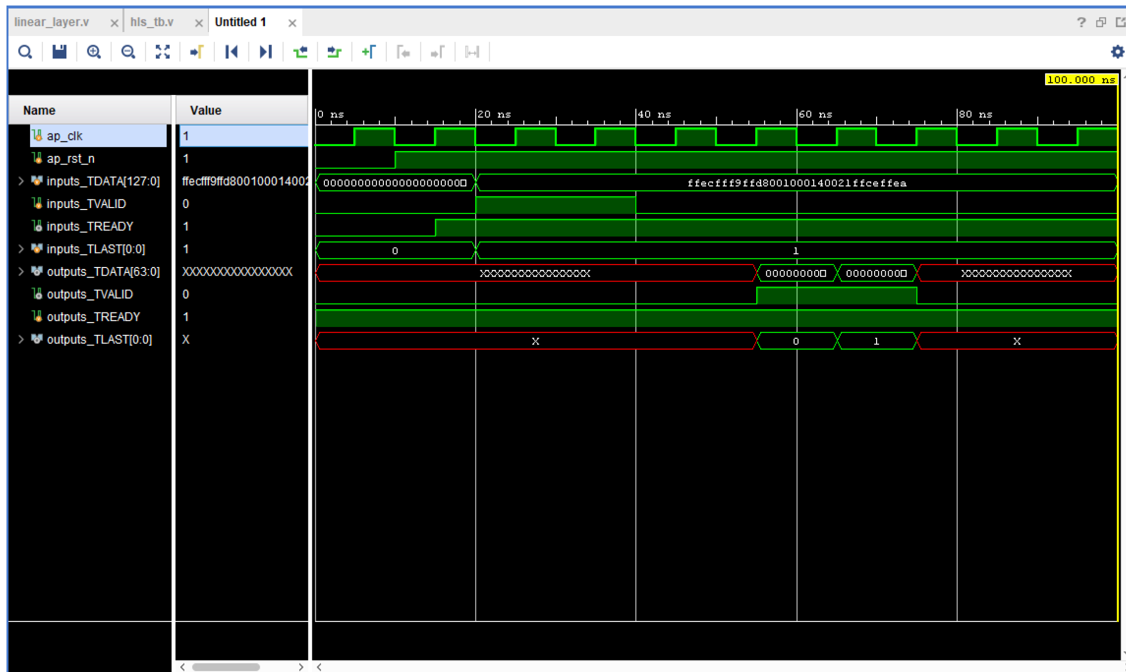


Figure 4.4: HLS testbench simulation result

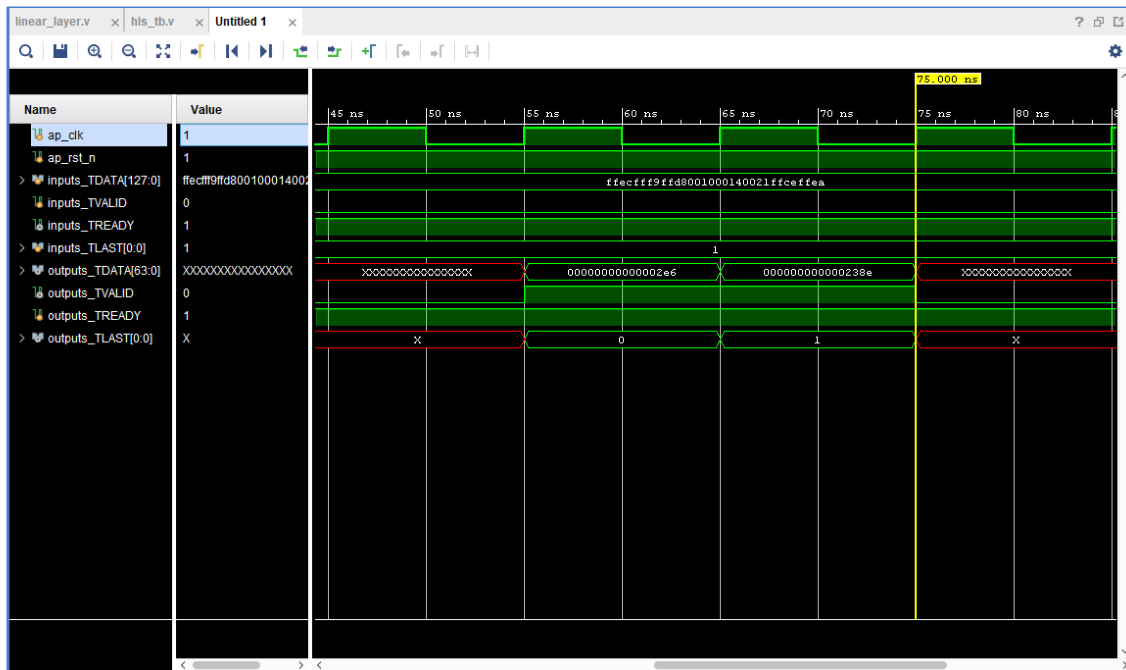


Figure 4.5: HLS testbench simulation result (zoomed)

For testing the performance of the IP block with continuous input, a testbench with several inputs is built, and the simulation result is shown in figure 4.6. In this figure, we can read the latency and calculate the throughput of the HLS IP block. The readout values are shown in table 4.7. The latency indicates the time taken from the input signal being valid to the output being ready, which suggests the time needed to process a single operation. As shown in the marks in figure 4.6, the latency can be calculated by $125ns - 20ns = 105ns$. The total operations of this module is 34. This is because it contains sixteen weights and two biases in total; each weight contains one addition and one multiplication, and each

bias contains an addition.The throughput can be calculated by the formula

$$Throughput = \frac{Operations}{Latency} = \frac{34}{105 \times 10^{-9}} = 0.3238095(Gop/s)$$

.

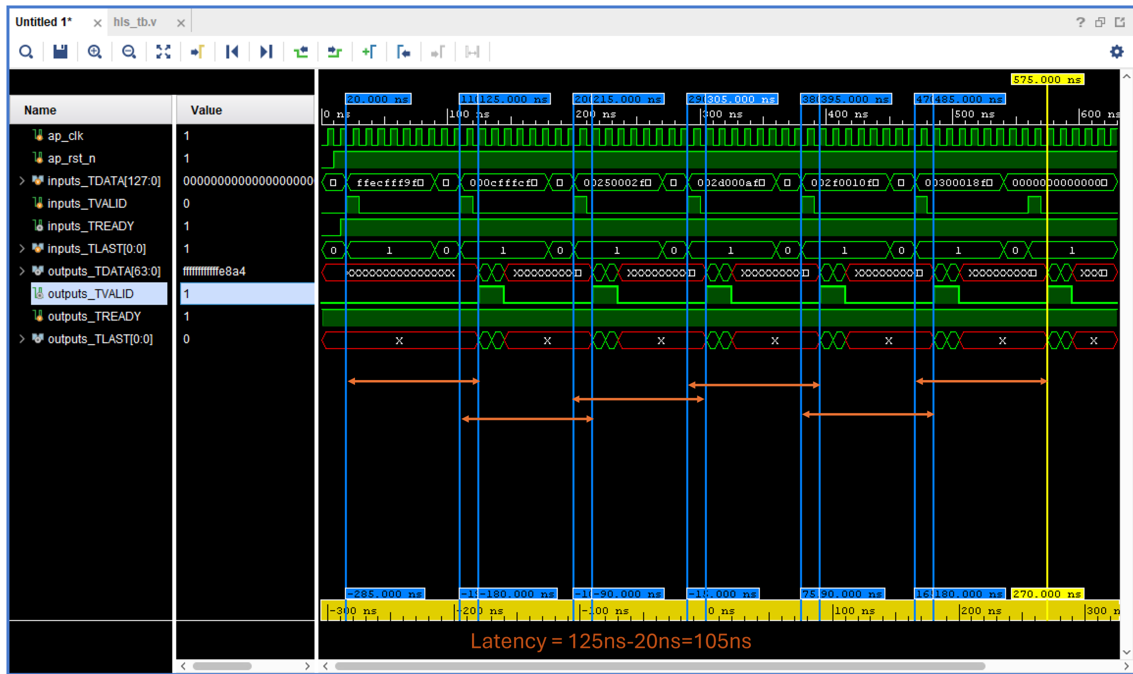


Figure 4.6: HLS testbench continuous input simulation result

	HLS linear performance
Total Latency (us)	0.630
Mean Latency (us)	0.105
Mean Eff. Throughput (GOp/s)	0.3238095

Table 4.7: HLS performance display

Due to the size of the figure, it's hard to read the inputs and outputs; the detailed readout values of Figure 4.6 are shown in Table 4.8. The data used in the continuous input simulation is also collected from the exported data set, and the results show that the HLS module works correctly.

inputs_TDATA (hex)	inputs_TDATA (dec)	outputs_TDATA(hex)	outputs_TDATA (dec)
FFEC_FFF9_FFD8_0010 _0014_0021_FFCE_FFEA	-22,-50,33,20,16,-40,-7,-20	00000000000002E6, 000000000000238E	742,9102
000C_FFFC_FFD3_0010 _FFF0_0002_FFB4_0006	6,-76,2,-16,16,-45,-4,12	FFFFFFFFFFFFEC38, 00000000000018BE	-5064,6334
0025_0002_FFBF_0006 _FFDE_0001_FFB2_0023	35,-78,1,-34,6,-65,2,37	FFFFFFFFFFFFF1D4, 0000000000001294	-3628,4756
002D_000A_FFB7_FFF7 _FFCF_0000_FFB1_0042	66,-79,0,-49,-9,-73,10,45	FFFFFFFFFFFFED93, 0000000000000BD0	-4717,3024
002F_0010_FFA9_FFF2 _FFC3_0000_FFB1_0055	85,-79,0,-61,-14,-87,16,47	FFFFFFFFFFFFEB67, 00000000000009BA	-5273,2490
0030_0018_FFA3_FFE4 _FFBC_0000_FFB1_0066	102,-79,0,-68,-28,-93,24,48	FFFFFFFFFFFFA8A4, 000000000000092B	-5980,2347,

Table 4.8: HLS behavioral simulation data

In order to find out how the HLS module improves the performance of the linear layer, a simple test is done to run the function of the linear layer in the CPU. The testing result is shown in figure 4.7. As shown in the figure, the latency is 0.81 microseconds, which indicates that the HLS module can provide a speedup of approximately 7.714 times compared to the CPU.

Figure 4.7: Linear layer CPU Result

4.4. Combined Performance Estimation

To evaluate this combined system displayed in section 3.3, performance metrics for the OpenDPD network when integrated into EdgeDRNN and the Linear layer High-Level Synthesis block are assessed separately. The results for the OpenDPD network applied in EdgeDRNN are detailed in section 4.2, while section 4.3 focuses on the performance of the Linear layer HLS block.

By analyzing these individual performance metrics, an estimation for the combined hardware design can be derived. Specifically, the latency of the combined system is anticipated to be the aggregate of the latencies from the two separate components. Conversely, the overall throughput of the system will be constrained by the element with the lower throughput. This method of estimation allows for the calculation of the mean latency and mean efficient throughput for the combined system. The results of these calculations are presented in table 4.9, providing a detailed assessment of the expected performance of the integrated hardware design.

	HLS linear performance	OpenDPD network applied in EdgeDRNN	Estimated combined system performance
Mean Latency (us)	0.105	6.398401	6.503401
Mean Eff. Throughput (GOp/s)	0.3238095	0.255064	0.255064

Table 4.9: Combined Performance estimation

5 Conclusions

5.1. Conclusion

In this thesis, we explore the implementation and evaluation of EdgeDRNN for digital pre-distortion applications, specifically focusing on its integration with the OpenDPD network. The research presented a comprehensive analysis of the latency and throughput of the EdgeDRNN hardware accelerator on digital pre-distortion.

The study focuses on the challenges associated with the high computational complexity and parameter count that often affect the deployment of DPD solutions on hardware. We successfully mitigated these issues using the EdgeDRNN hardware accelerator, providing a pathway toward more efficient DPD implementations. The successful adaptation and use of OpenDPD network parameters within the EdgeDRNN framework underscore the feasibility of our approach. Besides, a linear layer HLS block is built, and the performance containing its latency and throughput of this block is successfully collected. These results estimate the combined system consisting of the EdgeDRNN hardware accelerator and the built HLS module.

In summary, this thesis contributes to the DPD field by exploring the combination of advanced neural network architectures with an RNN hardware accelerator. The work shows how future research can explore its applicability across various communication standards and hardware platforms.

5.2. Limitation

The EdgeDRNN framework analyzed in this thesis has shown promising capability for DPD. However, the performance is still not good enough. This is because EdgeDRNN is designed for low-power applications such as speech recognition and robotic control, which means that DPD is not its initial intended target.

Besides, although the designed high-level synthesis of the linear layer functions, it could benefit from further optimization to reduce resource consumption while maintaining performance. In addition, this thesis estimates the performance of the combined system, and a detailed hardware test is necessary to get a more accurate result.

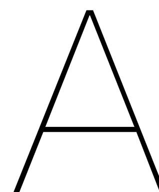
5.3. Outlook

The research presented in this thesis shows a way for future exploration and innovation within DPD. One potential direction is to develop a new hardware accelerator for DPD. As mentioned in the limitations, EdgeDRNN was not initially designed for DPD, and its performance does not meet the requirement of ultra-high-speed digital pre-distortion. So, a newly designed and optimized hardware accelerator for DPD is necessary and valuable for further research.

Moreover, further improvements to the built HLS linear layer module are necessary to improve its performance. Correct integration of the HLS module with the EdgeDRNN hardware accelerator is also essential for the combination test and future research.

In conclusion, while the EdgeDRNN framework improves the DPD technique, further research and development on hardware accelerators on DPD are essential to fully realize its potential and address

the evolving challenges of modern wireless communication systems.



Appendix1

In this appendix, the detailed C synthesis reports of the Linear layer HLS block are shown.

Solution 1:

Project: Linear_layer_CCC
Solution: solution1
Product family: zynq
Target device: xc7z007sclg225-1

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.185	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
73	73	73	73	none

Interface

- **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	linear_layer	return value
ap_rst_n	in	1	ap_ctrl_hs	linear_layer	return value
ap_start	in	1	ap_ctrl_hs	linear_layer	return value
ap_done	out	1	ap_ctrl_hs	linear_layer	return value
ap_idle	out	1	ap_ctrl_hs	linear_layer	return value
ap_ready	out	1	ap_ctrl_hs	linear_layer	return value
inputs_TDATA	in	128	axis	inputs_V_data_V	pointer
inputs_TVALID	in	1	axis	inputs_V_last_V	pointer
inputs_TREADY	out	1	axis	inputs_V_last_V	pointer
inputs_TLAST	in	1	axis	inputs_V_last_V	pointer
outputs_TDATA	out	64	axis	outputs_V_data_V	pointer
outputs_TVALID	out	1	axis	outputs_V_last_V	pointer
outputs_TREADY	in	1	axis	outputs_V_last_V	pointer
outputs_TLAST	out	1	axis	outputs_V_last_V	pointer

Solution 2:

Solution: solution2
Product family: zynq
Target device: xc7z007sclg225-1

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.283	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
5	5	2	2	function

Interface

- **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	linear_layer	return value
ap_rst_n	in	1	ap_ctrl_hs	linear_layer	return value
ap_start	in	1	ap_ctrl_hs	linear_layer	return value
ap_done	out	1	ap_ctrl_hs	linear_layer	return value
ap_idle	out	1	ap_ctrl_hs	linear_layer	return value
ap_ready	out	1	ap_ctrl_hs	linear_layer	return value
outputs_TREADY	in	1	axis	outputs_V_last_V	pointer
outputs_TVALID	out	1	axis	outputs_V_last_V	pointer
outputs_TLAST	out	1	axis	outputs_V_last_V	pointer
inputs_TDATA	in	128	axis	inputs_V_data_V	pointer
inputs_TVALID	in	1	axis	inputs_V_last_V	pointer
inputs_TREADY	out	1	axis	inputs_V_last_V	pointer
inputs_TLAST	in	1	axis	inputs_V_last_V	pointer
outputs_TDATA	out	64	axis	outputs_V_data_V	pointer

Solution 3:

Solution: solution3
Product family: zynq
Target device: xc7z007sclg225-1

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.283	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
5	5	2	2	function

Interface

- **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	linear_layer	return value
ap_rst_n	in	1	ap_ctrl_none	linear_layer	return value
inputs_TDATA	in	128	axis	inputs_V_data_V	pointer
inputs_TVALID	in	1	axis	inputs_V_last_V	pointer
inputs_TREADY	out	1	axis	inputs_V_last_V	pointer
inputs_TLAST	in	1	axis	inputs_V_last_V	pointer
outputs_TDATA	out	64	axis	outputs_V_data_V	pointer
outputs_TVALID	out	1	axis	outputs_V_last_V	pointer
outputs_TREADY	in	1	axis	outputs_V_last_V	pointer
outputs_TLAST	out	1	axis	outputs_V_last_V	pointer

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	8	-	-
Expression	-	-	0	289
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	150
Register	-	-	665	-
Total	0	8	665	439
Available	100	66	28800	14400
Utilization (%)	0	12	2	3

Comparison Report:

All Compared Solutions

solution1: xc7z007sclg225-1
solution2: xc7z007sclg225-1
solution3: xc7z007sclg225-1

Performance Estimates

- **Timing (ns)**

Clock		solution1	solution2	solution3
ap_clk	Target	10.00	10.00	10.00
	Estimated	7.185	10.283	10.283

- **Latency (clock cycles)**

		solution1	solution2	solution3
Latency	min	73	5	5
	max	73	5	5
Interval	min	73	2	2
	max	73	2	2

Utilization Estimates

	solution1	solution2	solution3
BRAM_18K	0	0	0
DSP48E	1	8	8
FF	822	666	665
LUT	1695	450	439

Bibliography

[1] Y. Wu, G. D. Singh, M. Beikmirza, L. C. N. de Vreede, M. Alavi, and C. Gao, "Opendpd: An open-source end-to-end learning benchmarking framework for wideband power amplifier modeling and digital pre-distortion," 2024.

[2] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[3] P. Kenington, "Linearized transmitters: an enabling technology for software defined radio," *IEEE Communications Magazine*, vol. 40, no. 2, pp. 156–162, 2002.

[4] M. S. . Heutmaker, E. Wu, and J. R. Welch, "Envelope distortion models with memory improve the prediction of spectral regrowth for some rf amplifiers," in *48th ARFTG Conference Digest*, vol. 30, pp. 10–15, 1996.

[5] M. Schetzen, "Nonlinear system modeling based on the wiener theory," *Proceedings of the IEEE*, vol. 69, no. 12, pp. 1557–1573, 1981.

[6] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.-Y. Glorennec, H. Hjalmarsson, and A. Juditsky, "Nonlinear black-box modeling in system identification: a unified overview," *Automatica*, vol. 31, no. 12, pp. 1691–1724, 1995. Trends in System Identification.

[7] C. Tarver, L. Jiang, A. Sefidi, and J. R. Cavallaro, "Neural network dpd via backpropagation through a neural network model of the pa," in *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, pp. 358–362, 2019.

[8] F. Ghannouchi, O. Hammi, and M. Helaoui, *Behavioral Modeling and Predistortion of Wideband Wireless Transmitters*. Wiley, 2015.

[9] C. Gao, A. Rios-Navarro, X. Chen, S.-C. Liu, and T. Delbruck, "Edgedrnn: Recurrent neural network accelerator for edge inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, p. 419–432, Dec. 2020.

[10] S. P. Yadav and S. C. Bera, "Nonlinearity effect of power amplifiers in wireless communication systems," in *2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE)*, pp. 12–17, 2014.

[11] J. Kim and K. Konstantinou, "Digital predistortion of wideband signals based on power amplifier model with memory," *Electronics Letters*, vol. 37, no. 23, p. 1, 2001.

[12] M. A. Hussein, O. Venard, B. Feuvrie, and Y. Wang, "Digital predistortion for rf power amplifiers: State of the art and advanced approaches," in *2013 IEEE 11th International New Circuits and Systems Conference (NEWCAS)*, pp. 1–4, 2013.

[13] D. Morgan, Z. Ma, J. Kim, M. Zierdt, and J. Pastalan, "A generalized memory polynomial model for digital predistortion of rf power amplifiers," *IEEE Transactions on Signal Processing*, vol. 54, no. 10, pp. 3852–3860, 2006.

[14] V. Bajaj, F. Buchali, M. Chagnon, S. Wahls, and V. Aref, "Deep neural network-based digital predistortion for high baudrate optical coherent transmission," *J. Lightwave Technol.*, vol. 40, pp. 597–606, Feb 2022.

[15] M. Yang, A. Yang, P. Guo, Z. Zhao, T. Xu, and W. Wan, "Digital pre-distortion for mach–zehnder modulators in imdd optical systems," in *2023 Opto-Electronics and Communications Conference (OECC)*, pp. 1–5, 2023.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[17] M. Minsky and S. Papert, *Perceptrons; an Introduction to Computational Geometry*. MIT Press, 1969.

[18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[19] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[20] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014.

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

[22] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.

[23] A. Faraji, S. A. Sadrossadat, W. Na, F. Feng, and Q.-J. Zhang, "A new macromodeling method based on deep gated recurrent unit regularized with gaussian dropout for nonlinear circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 7, pp. 2904–2915, 2023.

[24] Y. Wang, W. Liao, and Y. Chang, "Gated recurrent unit network-based short-term photovoltaic forecasting," *Energies*, vol. 11, no. 8, 2018.

[25] R. Rana, "Gated recurrent unit (gru) for emotion classification from noisy speech," 2016.

[26] J. Chen, H. Jing, Y. Chang, and Q. Liu, "Gated recurrent unit based recurrent neural network for remaining useful life prediction of nonlinear deterioration process," *Reliability Engineering System Safety*, vol. 185, pp. 372–382, 2019.

[27] M. S. Al-Rakhami, A. Gumaei1, M. Altaf, M. M. Hassan, B. F. Alkhamees, K. Muhammad, and G. Fortino, "Falldef5: A fall detection framework using 5g-based deep gated recurrent unit networks," 2021.

[28] Q. L. Li Xue, Khishe Mohammad, "Evolving deep gated recurrent unit using improved marine predator algorithm for profit prediction based on financial accounting information system," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 10, 2024.

[29] J. Qiu, H. Ma, and S. Tan, "Deep gated recurrent unit network for high-speed links modeling," in *2023 IEEE 7th International Symposium on Electromagnetic Compatibility (ISEMC)*, pp. 1–3, 2023.

[30] C. Gao, A. Rios-Navarro, X. Chen, T. Delbruck, and S.-C. Liu, "Edgedrnn: Enabling low-latency recurrent neural network edge inference," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, IEEE, Aug. 2020.

[31] A. G. O. I. au2, T. Mikolov, and D. Reitter, "Learning simpler language models with the differential state framework," 2017.

[32] D. Neil, J. H. Lee, T. Delbruck, and S.-C. Liu, "Delta networks for optimized recurrent network computation," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 2584–2593, PMLR, 06–11 Aug 2017.

[33] C. Gao, R. Gehlhar, A. D. Ames, S.-C. Liu, and T. Delbruck, "Recurrent neural network control of a hybrid dynamical transfemoral prosthesis with edgedrnn accelerator," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, May 2020.