Efficient and Effective DPD Neural Network

Master's Thesis

Kun Qian

Efficient and Effective DPD Neural Network

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Kun Qian

Lab of Efficient Machine Intelligence Research Group
Department of Electrical Engineering
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Efficient and Effective DPD Neural Network

Author: Kun Qian
Student id: 5995469

Abstract

This paper presents a comprehensive investigation into novel Recurrent Neural Network (RNN) architectures for enhancing the efficiency and performance of Digital Predistortion (DPD) for Radio Frequency (RF) Power Amplifiers (PAs). The research first introduces Delta Just Another Network (DeltaJANET), a computationally efficient model that leverages a sparse delta update rule. By updating only a small fraction of the hidden state at each time step, DeltaJANET significantly reduces the complexity and computational cost associated with traditional RNNs, establishing a new baseline for efficient DPD solutions. Building upon this foundation of efficiency, the paper then proposes a second, more powerful architecture: the Temporal Convolutional Just Another Network (TC-JANET). This advanced hybrid model synergistically combines a Temporal Convolutional Network (TCN) for long-range feature extraction with a lightweight Just Another Network (JANET) unit. Key innovations, including a direct memory input module and a dynamic phase normalization scheme applied to the recurrent state, enable the TC-JANET to robustly model complex PA behaviors. A systematic multi-seed evaluation demonstrates the exceptional performance and scalability of this architecture, showing that it significantly surpasses existing benchmarks and establishing a new benchmark for high-performance DPD solutions.

Thesis Committee:

Chair:	Prof. Dr. Leo de Vreede, Faculty EEMCS, TU Delft
University supervisor:	Dr. Chang Gao, Faculty EEMCS, TU Delft
Committee Member:	Dr. Yanki Aslan, Faculty EEMCS, TU Delft

Acknowledgements

As my Master's journey at TU Delft is now coming to an end, I am sincerely grateful for the special experience of returning to academia and for the many people who supported me.

I offer my deepest thanks to my thesis supervisor, Dr. Chang Gao. His deep knowledge, keen insight into cutting-edge research, and passion for teaching with an innate ability to tailor his guidance to each student are invaluable. This thesis would not have been possible without his professional and patient guidance. His dedication in teaching me everything from reading papers to innovating went far beyond my expectations. When I faced immense challenges, he was like a lighthouse, illuminating the path forward. I am truly honored to have had such a perfect mentor.

I would also like to thank my daily advisor, Yizhuo Wu, for her timely help and patient guidance. Her extensive knowledge in wireless communications and machine learning inspired many new ways of thinking. I am also grateful to my thesis committee members, Prof. Dr. Leo de Vreede and Dr. Yanki Aslan, for their invaluable feedback which made this thesis more robust.

I am very grateful to my friend Huanqiang Duan, a former master's student of Dr. Gao, whose TCN DPD paper inspired this work and who offered great help and experience. I also want to thank my friend, Kevin Xu, for his encouragement and the much-needed relaxation during this long and difficult research journey.

Finally, I must thank my family for their unwavering financial and emotional support, which allowed me to step away from my career and embark on this new voyage. My deepest gratitude is for my wife and daughter. Though they are far away in China, every video call had the magic power to sweep away my anxiety and stress.

I am grateful for all the people I have met and the experiences I have had in Delft these past two years, including the notoriously terrible Dutch weather. It wasn't always easy, but I will cherish these memories.

Kun Qian
September, 2025
Delft, the Netherlands

Acronyms

ACLR	Adjacent Channel Leakage Ratio	ix
ACPR	Adjacent Channel Power Ratio	ix
AM	Amplitude Modulation	14
ASIC	Application-specific integrated circuit	2
AutoML	Automated Machine Learning	41
BER	Bit Error Rate	8
BO-JANET	Block-Oriented Just Another Network	26
CNN	Convolutional Neural Networks	16
D-Conv	Depthwise Separable Convolutions	17
DeltaJANET	Delta Just Another Network	i
DeltaDGRU	Delta Dense Gated Recurrent Unit	ix
DGRU	Dense Gated Recurrent Unit	2
DPD	Digital Predistortion	i
DVR-JANET	Decomposed Vector Rotation-based JANET	14
E2E	End-to-End	27
EVM	Error Vector Magnitude	ix
FIR	Finite Impulse Response	26
FPGA	Field Programmable Gate Arrays	2
GMP	Generalized Memory Polynomial	vii
GRU	Gated Recurrent Unit	vii
JANET	Just Another Network	i
LSTM	Long Short-Term Memory	vii
NMSE	Normalized Mean Square Error	ix
NN	Neural Network	viii
OFDM	Orthogonal Frequency Division Multiplexing	27

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The rapid evolution of 5G and future 6G communication technologies has created an unprecedented demand for higher data rates and greater spectral efficiency. The adoption of complex modulation schemes with high Peak-to-Average Power Ratio (PAPR) has become an essential strategy to meet these demands. However, this brings severe challenges to a critical component in the RF chain: the PA. PAs are typically operated in the nonlinear region in order to achieve maximum power efficiency, which introduces significant signal distortion, along with spectral regrowth and in-band signal degradation.

DPD has been established as the standard technique to address this challenge. Although traditional polynomial-based DPD models, such as the GMP, have proven effective in many scenarios, they often struggle to accurately model the increasingly complex and dynamic memory effects exhibited by modern wideband PAs. This limitation has strongly motivated the development of NN based DPD technologies, as NNs offer superior capabilities in fitting the highly nonlinear behavior of such systems.

However, the powerful expressive capacity of NNs often comes at the cost of high computational complexity, which can be a significant bottleneck for deployment on resource-constrained hardware. Therefore, our preliminary research was conducted under such circumstances, focusing on enhancing the computational efficiency of DPD models. By introducing a sparse update mechanism, we successfully developed the DeltaJANET model. This model significantly reduces the computational overhead of conventional RNNs while maintaining strong linearization performance, thereby establishing a solid baseline for efficient DPD design.

Despite DeltaJANET achieving high performance in terms of efficiency, we never stopped the pursuit of the theoretical limits of DPD in linearity performance. We observed that even advanced RNN DPD models are often constrained by their recurrent structure, which can struggle to capture long-range temporal dependencies and hard to align with the physical properties of RF distortion. Furthermore, standard RNNs compress the entire signal history into a single recurrent state, which can obscure important details of the most

recent inputs required for modeling rapid dynamics.

This motivated the design of a hybrid architecture that synergistically combines a powerful feature extractor for global context with an efficient recurrent core for state processing. The key inspiration was to create an architecture that explicitly decouples phase and amplitude dynamics, while ensuring the most critical short-term signal history is losslessly utilized. Therefore, this research aims at exploring the performance limit of a RNN DPD by combining efficient global feature planning and the physical insights of phase normalization into recurrent cores, architecturally tailored for the task.

1.2 Problem Statement and Research Questions

1.2.1 Problem Statement

While RNN-based DPD has unveiled significant potential, its practical application is often constrained by the trade-off between computational efficiency and linearization performance. On one hand, High-performance architectures are often too computationally intensive for resource-constrained hardware, such as Field Programmable Gate Arrayss (FPGAs) and Application-specific integrated circuits (ASICs). This creates a clear demand for more **efficient** model designs that can reduce computational cost without a huge performance cutdown, forming the first major problem in this research.

On the other hand, lightweight design was adopted in many models inevitably sacrifice the expressive capacity required to accurately model the complex, long-range memory effects of modern wideband PAs, limiting their **effectiveness**. This leads to a performance gap between theoretical potential and achievable, robust results. Addressing this second problem requires the development of novel architectures that not only possess high expressive capability, but can also **effectively translate an increased parameter count into significant, measurable improvements in linearization performance**. The goal of this research is to develop distinct solutions for each of these challenges respectively, in order to achieve either high efficiency or superior performance with stability.

1.2.2 Research Questions

The evolution of RNN-based DPD models from GRU and LSTM-based structures (such as Vector Decomposed Long Short-Term Memory (VDLSTM) [14] and Dense Gated Recurrent Unit (DGRU)) [19], to JANET-based models (such as Phase-Gated Just Another Network (PG-JANET) [13] and PNRNN [6]), and ultimately to simplified single-gate architectures reveals a consistent trend toward reducing the complexity of the recurrent gate structure while relying on increasingly complicated feature extraction techniques to minimize model size. Although this progression has generally resulted in improved linearization performance, our experimental results show that, with a sufficiently large RNN kernel size, models employing simple feature extraction and GRU-based architectures still significantly outperform their more complex

counterparts. Consequently, to further investigate the respective roles of the RNN architecture and feature extraction strategies in determining linearization performance and achieving efficient DPD, we pose the following research questions:

1. When applying the delta update rule to an RNN-based DPD model, how much temporal sparsity can be exploited to reduce computational complexity without affecting linearization?
2. Compared with JANET and GRU, what are the advantages and disadvantages of a simplified RNN structure in the design of an efficient DPD Model?
3. As a DPD model, the TCN has good linearization performance when the model size is small. If it is used as a learnable feature extraction layer and combined with an RNN to build a hybrid network, how will the linearization performance be affected?
4. What is the impact of feature augmentation techniques, such as phase normalization and the input memory context window, on the performance of the proposed TC-JANET model?

1.3 Thesis Contributions

1.3.1 Thesis Contributions

The main contributions of this thesis are summarized as follows.

1. We first introduce DeltaJANET, a computationally efficient RNN architecture for DPD that leverages a sparse delta update rule. This approach is shown to significantly reduce the computational cost of conventional recurrent models while maintaining a competitive level of linearization performance.
2. We then propose TC-JANET, a novel hybrid architecture that synergistically combines a TCN as a global feature planner with a lightweight recurrent core. The recurrent core is augmented with a Phase Normalization (PN) scheme to effectively handle the complex-valued nature of the signals and stabilize the learning process. This design is motivated by the need to more effectively model long-range memory effects and is validated by its superior linearization performance.

1.4 Thesis Outline

The rest of this thesis is organized as follows.

- **Chapter 2: Background and Related Work:** This chapter first introduces the background of PA nonlinearities and the principles of DPD. It then reviews the evolution of DPD, from the traditional polynomial-based

model to various milestone NN-based architectures, including RNN, TCN, and other prior models relevant to this work.

- **Chapter 3: Methods:** This chapter introduces the two novel architectures proposed in this research. First part details the motivations and architecture of DeltaJANET, a model designed for computational efficiency through sparse updates. Subsequently, it presents our high-performance model, TC-JANET, explaining its hybrid TCN-RNN design, the Memory Context Window module, and the core PN mechanism.
- **Chapter 4: Experimental Setup and Results:** This chapter describes the comprehensive experimental validation process. It first details the common experimental setup, including the dataset, hardware platform, and evaluation metrics. It then presents the results for the DeltaJANET model, focusing on its efficiency gains. The second half of the chapter is dedicated to the extensive results of the TC-JANET model, covering the systematic optimization of its hyperparameters and its final performance comparison against prior models on multiple metrics.
- **Chapter 5: Conclusion and Discussion:** This chapter summarizes the key findings and contributions of the thesis, discusses the implications of the results, and proposes potential avenues for future research.

1.5 Terminology

This thesis uses the following key terms:

DPD (Digital Predistortion) A key technology used to mitigate nonlinearities in power amplifiers by pre-distorting the input signal.

PA (Power Amplifier) A device used to amplify RF signals, typically exhibiting nonlinear characteristics that need to be compensated.

RNN (Recurrent Neural Network) A class of neural networks specifically designed for processing sequential data with temporal dependencies.

JANET (Just Another Network) A simplified RNN architecture that reduces computational complexity by focusing solely on the forget gate mechanism.

Delta Network A neural network architecture that exploits temporal sparsity by updating states only when input changes exceed a threshold.

ACPR (Adjacent Channel Power Ratio) A metric measuring signal linearization performance by quantifying power leakage into adjacent frequency channels.

NMSE (Normalized Mean Square Error) A metric quantifying the deviation between the transmitted signal and the ideal signal.

Γ **(Sparsity)** The proportion of inactive neurons in the network, indicating computational efficiency.

β **(Beta)** A hyperparameter in the JANET architecture that balances between memory retention and adaptation to new inputs.

OpenDPD An end-to-end learning framework for evaluating digital predistortion architectures, implemented in PyTorch.

Chapter 2

Background and Related Works

In the begining, this chapter first introduced the fundamental concepts of PA nonlinearity and the principles of DPD in RF systems in Sec. 2.1. Then, it reviews the history of DPD modeling, beginning with traditional approaches such as the GMP and the Volterra Series in Sec. 2.2. The following sections provide a detailed overview of the key NN architectures that form the foundation of modern DPD. Sec. 2.3 describes foundational gated RNNs like LSTM and GRU. Sec. 2.4 and Sec. 2.5 introduce the more advanced JANET and Delta Network concepts, which are central to our work on efficiency. Finally, Sec. 2.6 and Sec. 2.7 cover two critical technologies for high-performance modeling: the TCN architecture and the PN technique, respectively.

2.1 Digital Predistortion in RF Systems

DPD is a baseband signal processing technique designed to linearize RF PAs. They are inherently nonlinear devices, with characteristic related to the physics of the transistors from which they are built. This nonlinearity becomes particularly pronounced when they are operated near their saturation region, in order to maximize power efficiency in mordern communication systems [11]. Signal distortion was introduced by this nonlinear behavior, leading to spectral regrowth into adjacent channels and degrades the in-band signal quality. As illustrated in Figure 2.1, the fundamental principle of DPD is to apply a complementary nonlinear function to the digital input signal. With this "predistorted" signal, the amplifier's distortion is precisely canceled out when passed through the PA, resulting in a final RF output that is a linearized, amplified version of the original input. The challenge, however, is significantly intensified by the PA's memory effects, where the current output depends not only on the current input but also on its recent history. These effects are more severe for wideband signals. They originate from both short-term electrical phenomena (e.g., trapping effects) and long-term thermal phenomena. Accurately modeling and compensating for these dynamic memory effects demands more sophisticated DPD solutions than traditional static methods can provide.

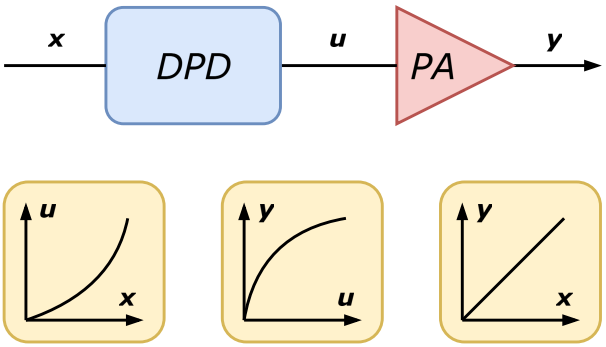

Figure 2.1: Demonstration of DPD

2.1.1 Evaluation Metrics

To quantify the effectiveness of a DPD solution, several standard metrics are employed, each evaluating a different aspect of the linearization performance.

The NMSE is a fundamental metric that reflects the accuracy of the DPD model in the time domain. It measures the normalized power of the error between the measured PA output and the ideal, desired output. For a test signal of N samples, it is calculated as:

$$\text{NMSE (dB)} = 10\log_{10}\left(\frac{\sum_{n=1}^{N}|y(n)-y_{\text{ideal}}(n)|^2}{\sum_{n=1}^{N}|y_{\text{ideal}}(n)|^2}\right) \tag{2.1}$$

where $y(n)$ is the measured output and $y_{\text{ideal}}(n)$ is the ideal reference signal.

While NMSE measures time-domain accuracy, the ACPR, or ACLR, is arguably the most critical metric for system-level performance especially in DPD application. It quantifies the spectral regrowth caused by PA nonlinearity by measuring the ratio of the leaked power in adjacent channels to the power in the main channel. This is defined as:

$$\text{ACLR (dBc)} = 10\log_{10}\left(\frac{P_{\text{adjacent}}}{P_{\text{main}}}\right) \tag{2.2}$$

It is expressed in dBc and directly related to the spectral efficiency and the ability of a transmitter to operate without interfering with neighboring channels.

Furthermore, the EVM is used to evaluate the quality of the transmitted signal in the modulation domain. As illustrated for a QAM signal in Figure 2.2, the EVM is defined as the Root Mean Square (RMS) amplitude of the error vector between the ideal constellation point and the measured point, normalized by the amplitude of the ideal signal. A low EVM is essential for the receiver to correctly demodulate the signal and maintain a low Bit Error Rate (BER).

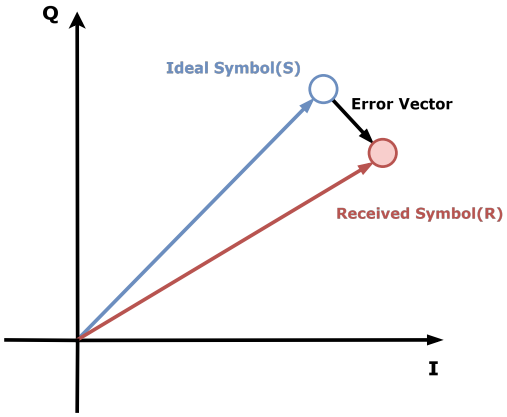

Figure 2.2: First quadrant of a QAM Constellation diagram illustrating the error vector used in EVM calculation.

Finally, the complexity of the DPD model is another important dimension to be considered for hardware implementation. One of the indcatior of model's computational cost is the total number of trainable paramters, although this should be supplemented with analysis of specific operations.

2.2 Traditional Methods and the Rise of Neural Networks in DPD

In early history, the field of DPD was dominated by polynomial-based models. These approaches aimed to mathematically model the nonlinear behavior of PAs. As the understanding of PA characteristics went deeper, these models also grew in complexity and efficacy. The progression from early memoryless approaches to more complex structures capable of modeling dynamic memory effects led to the transition towards more powerful, learning-based techniques.

2.2.1 GMP

The GMP model emerged and became an industry standard for many years [17] due to its highly effective and practical compromise. The GMP model simplifies the Volterra series by considering only a subset of its cross-terms, focusing on the interactions between the current signal and delayed versions of its envelope. The general form can be expressed as:

$$\begin{aligned} y_{\text{GMP}}(n) = &\sum_{k=0}^{K_a-1}\sum_{l=0}^{L_a-1} a_{kl}x(n-l)\,|x(n-l)|^k \\ &+ \sum_{k=1}^{K_b}\sum_{l=0}^{L_b-1}\sum_{m=1}^{M_b} b_{klm}x(n-l)\,|x(n-l-m)|^k \\ &+ \sum_{k=1}^{K_c}\sum_{l=0}^{L_c-1}\sum_{m=1}^{M_c} c_{klm}x(n-l)\,|x(n-l+m)|^k \end{aligned} \tag{2.3}$$

Here, the coefficients a_{kl}, b_{klm}, and c_{klm} represent the aligned, lagging, and leading envelope terms, respectively. However, the efficacy of the GMP model diminishes in wider signal bandwidths. This tremendously increases the number of coefficients required in order to model long and complex memory effects and consequently results in high computational complexity and parameter identification challenges.

2.2.2 Volterra Series

The Volterra series represents one of the most comprehensive theoretical frameworks for modeling nonlinear systems with memory. As a high-order extension of the Taylor series, it can theoretically approximate any arbitrary nonlinear system to a desired level of accuracy. The output $y(t)$ of a discrete-time Volterra series is given by a sum of multi-dimensional convolutions:

$$y(t) = \sum_{p=1}^{\infty} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} h_p(\tau_1, \ldots, \tau_p) \prod_{i=1}^{p} x(t-\tau_i) d\tau_i \tag{2.4}$$

where $h_p(\cdot)$ are the Volterra kernels of order p. While the Volterra series are powerful, the full version is rarely used in practice due to its explosive complexity. The number of coefficients required grows exponentially with both the nonlinearity order and the memory depth. The high computational complexity makes it hardly possible for wideband DPD applications [4]. This is why the various pruned or simplified versions, such as the memory polynomial model have become more prevalent.

2.2.3 TDNN

The performance gap of traditional models motivated the transition towards Neural Network (NN) based DPD. NNs, with their inherent ability to approximate complex, arbitrary nonlinear functions, are naturally suited for the DPD task. Early explorations in TDNN were successful as they provided a structured way for feedforward networks to process temporal information. A TDNN augments its input layer with a tapped delay line, allowing it to process not only the current input sample but also a finite window of past samples simultaneously, thereby capturing short-term memory effects.

A prominent and effective variant is the RVTDNN, which was demonstrated to be a powerful tool for PA modeling [15]. As shown in Figure 2.3, the RVTDNN processes the in-phase (I) and quadrature (Q) components of the complex baseband signal separately. For a complex input signal $x(t) = I(t) + jQ(t)$, the input to the network is a real-valued vector, $\mathbf{u}(t)$, formed by concatenating the current and a set number of M delayed I/Q samples:

$$\mathbf{u}(t) = [I(t), Q(t), I(t-1), Q(t-1), \ldots, I(t-M), Q(t-M)]^T \tag{2.5}$$

This input vector is then fed through one or more hidden layers, which use nonlinear activation functions (e.g., tanh or sigmoid) to learn the complex

mapping from the PA's input history to its distorted output. The output of a hidden neuron j is typically of the form:

$$y_j(t) = f\left(\sum_{i=0}^{2(M+1)-1} w_{ji}u_i(t) + b_j\right) \tag{2.6}$$

where w_{ji} are the weights, b_j is the bias, and $f(\cdot)$ is the activation function. The final output layer then combines the outputs of the hidden neurons to produce the predistorted signal. The RVTDNN demonstrated that NNs could effectively capture both the static nonlinearities and the dynamic memory effects of PAs, paving the way for the more advanced recurrent and convolutional architectures explored in this work.

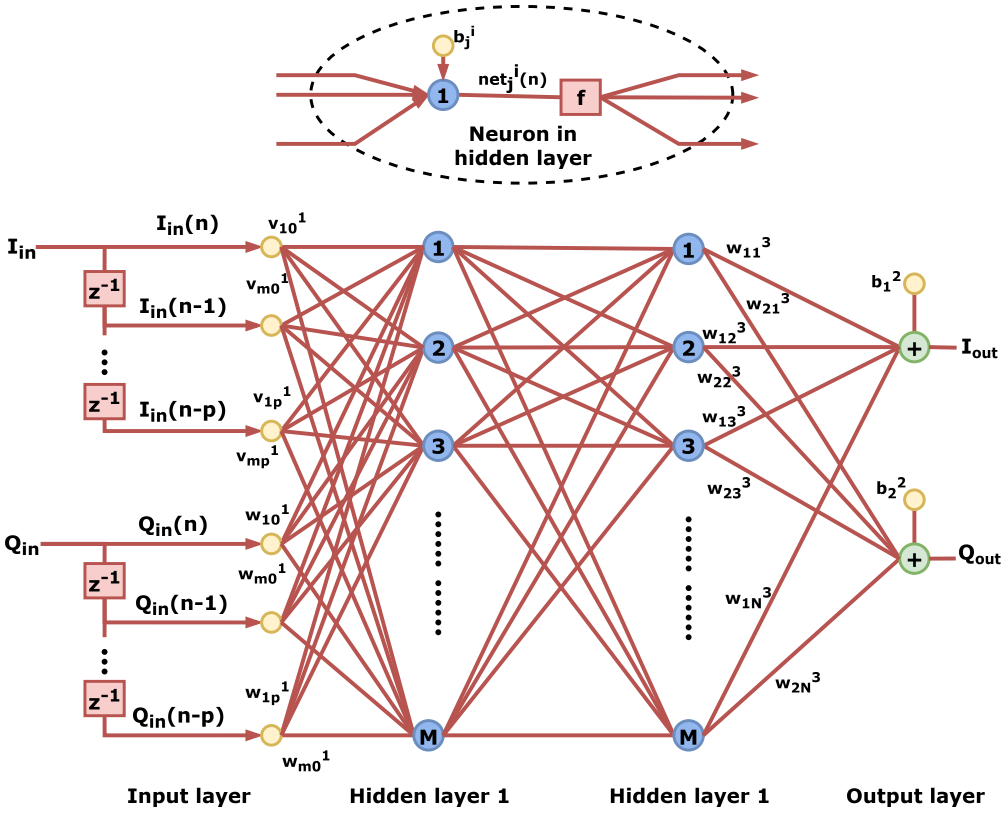

Figure 2.3: Block diagram of three-layer RVTDNN behavioral model.

2.3 RNN

RNNs are a class of NNs specifically designed for sequential data, making them exceptionally well-suited for modeling the dynamic memory effects of PAs. Unlike feedforward networks, RNNs possess an internal hidden state, h_t, which is updated at each time step, t, by combining the current input, x_t, with the previous hidden state, h_{t-1}. This recurrent structure allows them to

process sequences of arbitrary length by recursively applying the same transition function, theoretically enabling them to capture temporal dependencies. However, simple RNNs are notoriously difficult to train on long sequences due to the vanishing and exploding gradient problems. To address this, more sophisticated architectures incorporating gating mechanisms were introduced, with the most prominent being the LSTM and the GRU.

2.3.1 LSTM

The LSTM network [10] was a milestone innovation designed to overcome the limitations of simple RNNs. Its core feature is the introduction of a dedicated cell state, c_t, which acts as an information "conveyor belt," allowing long-term dependencies to flow through the network with minimal decay. The flow of information into, out of, and within this cell state is precisely regulated by three gates: an input gate (i_t), a forget gate (f_t), and an output gate (o_t). This architecture, illustrated in Figure 2.4, uses these gates to selectively add or remove information from the cell state, enabling the LSTM to effectively remember relevant information over very long time horizons. The update equations are as follows:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{2.7}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{2.8}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{2.9}$$
$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.10}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{2.11}$$
$$h_t = o_t \odot \tanh(c_t) \tag{2.12}$$

where σ is the sigmoid function, $\odot$ denotes element-wise multiplication, and $\tilde{c}_t$ is the candidate cell state.

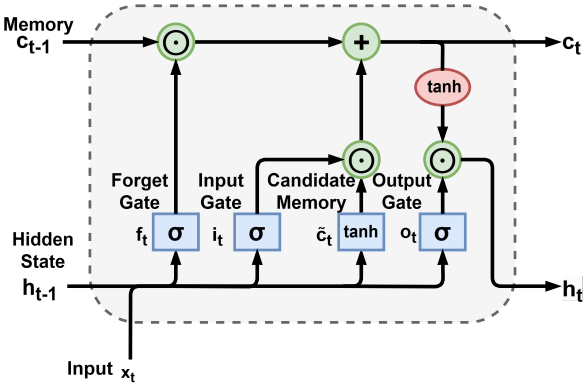

Figure 2.4: LSTM Architecture [10].

2.3.2 GRU

The GRU [2] was proposed as a more computationally efficient alternative to the LSTM. The GRU simplifies the architecture by merging the cell state and

the hidden state into a single state vector, h_t. It also replaces the three gates by a reset gate (r_t) and an update gate (z_t). The reset gate determines how to combine the new input with the previous memory, while the update gate decides how much of the previous information to keep. This streamlined structure, shown in Figure 2.5, reduces the number of parameters, which lead to faster training while often delivering comparable performance to LSTM. The GRU's update equations are:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{2.13}$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{2.14}$$
$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{2.15}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{2.16}$$

where $\tilde{h}_t$ is the candidate hidden state. These gated architectures form the foundation upon which more specialized recurrent models for DPD, such as JANET, have been developed.

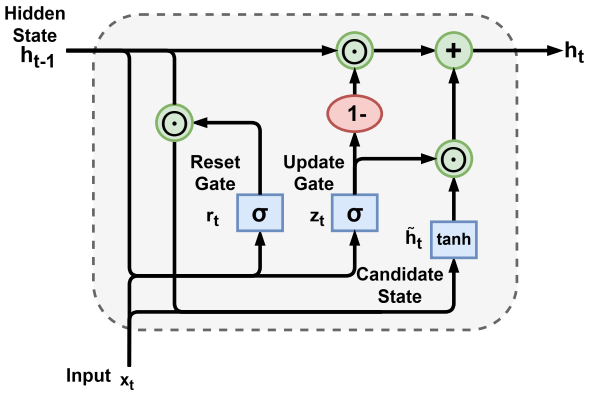

Figure 2.5: GRU Architecture [2].

2.4 JANET

JANET [18] simplifies the LSTM network by using a single forget gate, which reduces computational complexity while retaining a strong expressive capacity in modeling temporal dependencies. The state update equations are given by:

$$s_t = U_f h_{t-1} + W_f x_t + b_f \tag{2.17}$$
$$\tilde{c}_t = \tanh(U_c h_{t-1} + W_c x_t + b_c) \tag{2.18}$$
$$h_t = \sigma(s_t) \odot c_{t-1} + (1 - \sigma(s_t - \beta)) \odot \tilde{c}_t \tag{2.19}$$

Here, σ is the sigmoid function, $\tilde{c}_t$ is the candidate value, c_{t-1} is the previous cell state, and s_t is the unactivated forget gate. This architecture is shown in Fig. 2.6. A key feature is the hyperparameter β, which decouples the forget

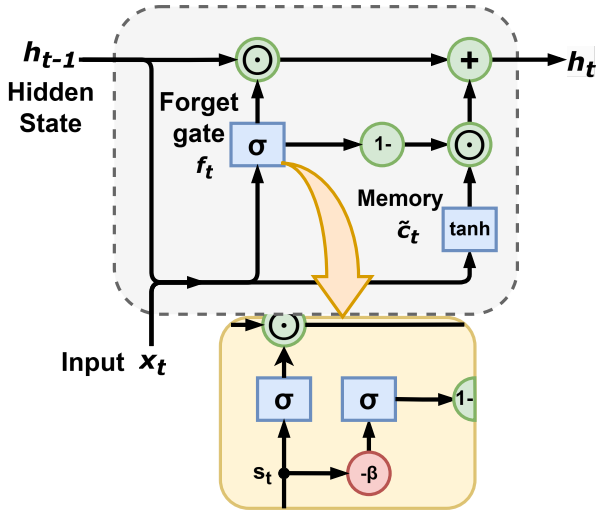

Figure 2.6: JANET Architecture [18].

and input gates, allowing the model to flexibly control the balance between retaining historical information and adapting to new inputs.

A highly specialized variant designed for the DPD task is the Decomposed Vector Rotation-based JANET (DVR-JANET)[12]. The strength of the DVR-JANET lies in its explicit decoupling of magnitude and phase processing. The model first decomposes the input signal $x(t)$ into its magnitude $|x(t)|$ and phase components, represented by $\cos(\phi(t))$ and $\sin(\phi(t))$. These are then processed by separate recurrent sub-networks (typically JANET or GRU cells). This structural separation allows the model to learn distinct and independent mappings for the Amplitude Modulation (AM)-AM and AM-Phase Modulation (PM) distortions. The core innovation is the vector rotation mechanism, where the outputs of these sub-networks are used to construct a complex rotation factor that modifies the input vector to produce the predistorted output $\hat{y}(t)$:

$$\hat{y}(t) = x(t) \cdot f(|x(t)|, h_{t-1}) \cdot e^{j \cdot g(\phi(t), h_{t-1})} \tag{2.20}$$

where $f(\cdot)$ and $g(\cdot)$ are the functions learned by the magnitude and phase sub-networks, respectively. This specialized structure, by dedicating separate pathways for amplitude and phase modeling, has demonstrated high performance, making it a particularly challenging and relevant benchmark.

2.5 Delta Networks

Delta Networks [7, 16] offer a compelling approach to reduce the computational cost of RNNs by exploiting the temporal sparsity inherent in many real-world signals. The core principle is that for continuous sequential data, the input and the resulting RNN activations often do not change significantly at every single time step. A conventional RNN, however, performs a full, computationally expensive matrix-vector multiplication for every input, leading to redundant operations when the signal is quasi-static, as illustrated in Fig. 2.7.

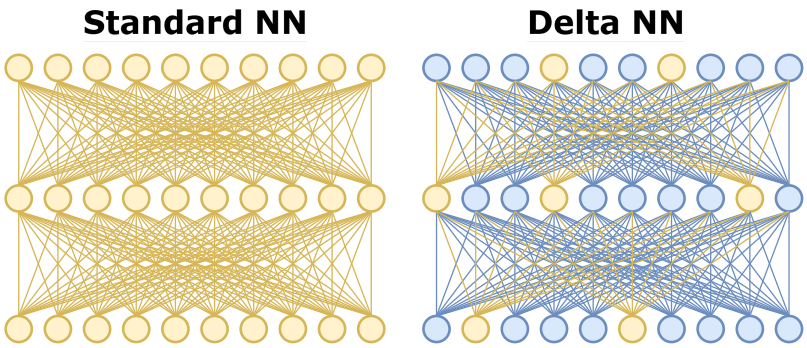

Figure 2.7: Conceptual comparison between a conventional network update (left) and a delta network update (right). (This figure is adapted and redrew from [7]

To address this inefficiency, Delta Networks transition from a clock-driven to an event-driven computation paradigm. The update for a neuron is only computed when the change in its input exceeds a predefined threshold, Θ. This is formalized by first computing a "delta" input, Δx_t. For the i-th dimension of an input vector x_t, the delta input is:

$$\Delta x_{i,t} = \begin{cases} x_{i,t} - \hat{x}_{i,t-1} & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta_x, \\ 0 & \text{otherwise.} \end{cases} \tag{2.21}$$

Here, $\hat{x}_{i,t-1}$ is a state-holding variable that stores the value of the input from its last significant change, preventing error accumulation over time. The state-holding variable is updated only when the threshold is exceeded:

$$\hat{x}_{i,t} = \begin{cases} x_{i,t} & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta_x, \\ \hat{x}_{i,t-1} & \text{otherwise.} \end{cases} \tag{2.22}$$

This same principle is applied to the recurrent hidden state, h_t, creating a delta hidden state Δh_t. The standard matrix-vector multiplication, $\mathbf{y}_t = \mathbf{W}\mathbf{x}_t$, can then be reformulated into a sequential, additive form:

$$\mathbf{y}_t = \mathbf{W}\Delta\mathbf{x}_t + \mathbf{y}_{t-1} \tag{2.23}$$

where $\mathbf{y}_{t-1}$ is the result from the previous time step. In this formulation, if the vector $\Delta\mathbf{x}_t$ is sparse (containing many zeros), the operation becomes a sparse matrix-vector multiplication. As shown in Fig. 2.8, computations corresponding to the zero elements in $\Delta\mathbf{x}_t$ can be skipped entirely, significantly reducing the number of required arithmetic operations and memory accesses.

As a concrete example, consider the application of this principle to a GRU. In a DeltaGRU, the update equations are reformulated to operate on the delta inputs, $\Delta\boldsymbol{\phi}_t$, and delta hidden states, $\Delta\mathbf{h}_{t-1}$. The pre-activation accumulations for the gates, denoted by M, are updated additively, e.g.:

$$\mathbf{M}_{z,t} = \mathbf{W}_{iz}\Delta\boldsymbol{\phi}_t + \mathbf{W}_{hz}\Delta\mathbf{h}_{t-1} + \mathbf{M}_{z,t-1} \tag{2.24}$$

$$\mathbf{h}_t = (1 - \sigma(\mathbf{M}_{z,t})) \odot \mathbf{h}_{t-1} + \sigma(\mathbf{M}_{z,t}) \odot \tilde{\mathbf{h}}_t \tag{2.25}$$

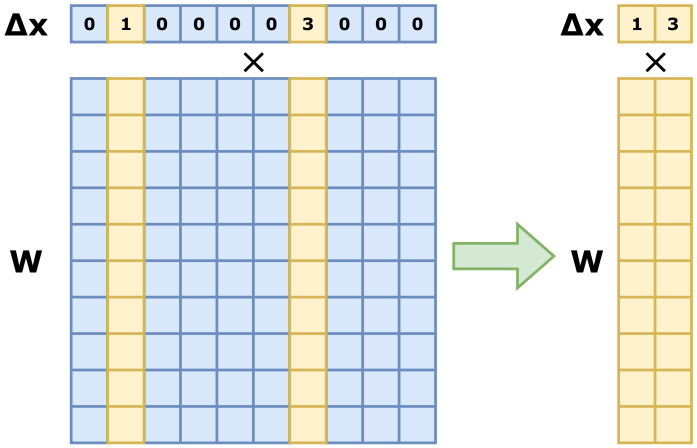

Figure 2.8: Illustration of how sparse vector $\Delta x(t)$ skips multiplications with corresponding columns in the weight matrix.(This figure is adatped and redrew from [7]

This event-driven computation avoids redundant operations, leading to substantial energy and latency savings and making the Delta Network concept an attractive foundation for efficient DPD design.

2.6 Temporal Convolutional Network (TCN) Architecture

TCN [1] represent a powerful architecture for sequence modeling that is built upon Convolutional Neural Networks (CNN), offering a compelling alternative to traditional recurrent architectures. Unlike RNNs, TCN are non-recurrent and can process entire sequences in parallel, leading to significant advantages in training and inference speed. The modern TCN architecture is typically constructed from a stack of residual blocks, each employing dilated convolutions to capture long-term history [3].

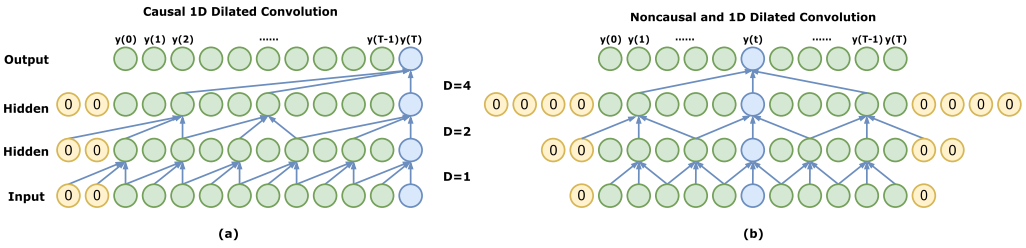

Figure 2.9: Comparison of (a) causal and (b) non-causal dilated convolutions, distinguished by the use of future inputs ($> t$) in the non-causal case.

A fundamental aspect of TCN for time-series modeling is the concept of *causality*. As shown in Figure 2.9(a), a causal convolution ensures that the prediction at time step t can only depend on inputs from time t and earlier, i.e., $(x_0, \ldots, x_t)$. This is a critical property for real-time applications as it prevents any leakage from future information. Alternatively, for offline pro-

cessing tasks like DPD where the entire signal block is available beforehand, a *non-causal* convolution can be employed, as depicted in Figure 2.9(b). This symmetric window provides a richer contextual representation for each time step, which can potentially lead to improved modeling accuracy, at the cost of introducing latency.

To achieve a large receptive field efficiently in either configuration, TCN employ *dilated convolutions*. For a 1D input sequence $\mathbf{x}$ and a filter $\mathbf{f}$ of size k, the dilated convolution operation F at element s of the sequence is defined as:

$$F(s) = (\mathbf{x} *_d \mathbf{f})(s) = \sum_{i=0}^{k-1} f(i) \cdot x(s - d \cdot i) \tag{2.26}$$

where d is the dilation factor. By stacking these convolutional layers and exponentially increasing the dilation factor at each subsequent layer (e.g., $d = b^l$ for layer l, where b is the dilation base), the TCN can expand its receptive field exponentially.

As illustrated in Figure 2.10, a standard TCN is built from a series of **residual blocks**. While conceptually similar to the residual blocks found in architectures like ResNet [9], the blocks in TCN are specialized for sequence modeling and computational efficiency. Instead of standard convolutions, a stack of Depthwise Separable Convolutions (D-Conv) was employed to decouple the temporal and cross-channel features. First, a D-Conv (implemented via $groups = in_channels$ in PyTorch) applies a single convolutional filter to each input channel independently to capture temporal patterns. This is followed by a *pointwise convolution* (a 1×1 convolution) that linearly combines the outputs of the depthwise layer to facilitate information flow between channels. This entire structure is then placed within a **residual connection**, where the input to the block is added to its output. This residual mechanism is critical for training very deep networks by preventing the vanishing gradient problem and ensuring a stable flow of information.

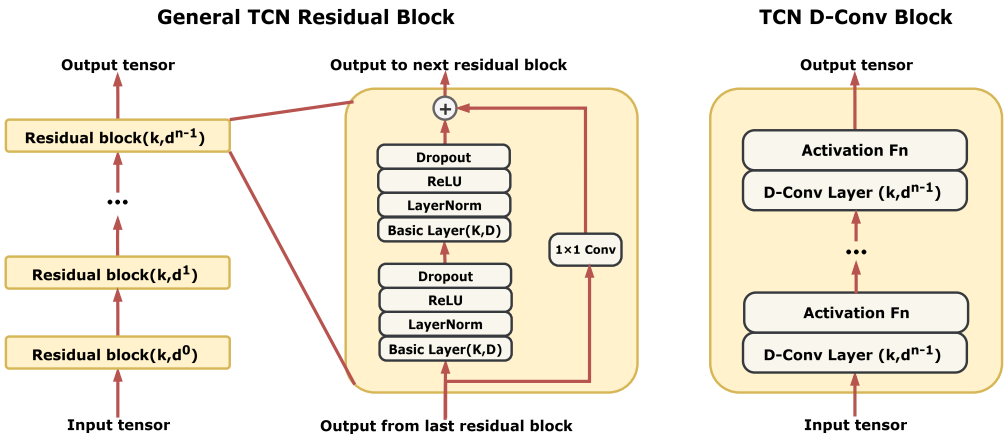

Figure 2.10: The architecture of a TCN Residual Block, as used in [3], which employs depthwise separable convolutions.

The structure and performance of a TCN are affected by several key hy-

perparameters. The `kernel_size` (k) defines the width of the convolutional filter. The `dilation_base` (b) controls the expansion rate of the receptive field. The number of layers (N) determines the depth of the network. These hyperparameters synergistically define the TCN's effective Receptive Field , which can be calculated as:

$$\text{Receptive Field} = 1 + (k-1)\sum_{l=0}^{N-1} b^n = 1 + (k-1)\frac{b^N - 1}{b-1} \tag{2.27}$$

This formula shows how a well-configured TCN can achieve a very large receptive field with a modest number of parameters, making it highly efficient for modeling the long-term memory effects required for wideband DPD.

2.7 Phase Normalization

A significant challenge in NN-based DPD is that standard networks treat the complex baseband signal as a generic two-dimensional input (I/Q), failing to leverage the physical properties of RF distortion. PA nonlinearities are primarily dependent on the signal's instantaneous envelope (amplitude) and its history, not its absolute phase [5]. Consequently, the model must learn the PA's distortion characteristics for every possible input phase angle, which cause a highly inefficient process that leads to increased model complexity.

PN is a technique designed to align the learning process with this physical reality [5, 6]. The core idea is to dynamically rotate the complex-valued input signal at each time step t such that the current sample $x(t)$ is projected onto the positive real axis, effectively normalizing its phase to a reference of zero. This is achieved by computing a sample-wise rotation factor, $r(t)$, as follows:

$$r(t) = \frac{x^*(t)}{|x(t)| + \epsilon} \tag{2.28}$$

where $x^*(t)$ is the complex conjugate of the input sample, $|x(t)|$ is its envelope, and ϵ is a small constant for numerical stability.

When applied to a model with a memory window of depth M, this rotation factor is multiplied not only with the current sample but also with the entire vector of past samples, $\mathbf{x}_M(t) = [x(t), x(t-1), \ldots, x(t-M)]$. This transforms the entire input window into a normalized domain:

$$\mathbf{x}_{\text{norm}}(t) = r(t) \odot \mathbf{x}_M(t) \tag{2.29}$$

The NN then learns a mapping from this simplified, phase-agnostic representation to the PA's distorted output. By operating in this normalized domain, the network is no longer required to learn redundant mappings for different phase angles. Instead, it can focus on the more fundamental relationship between the signal envelope, the *relative* phase differences between samples in the memory window, and the resulting distortion.

Finally, after the network produces a normalized output, $y_{\text{norm}}(t)$, the original absolute phase must be restored. This is accomplished by a denormalization step, where the output is multiplied by the complex conjugate

of the original rotation factor:

$$\hat{y}(t) = y_{\text{norm}}(t) \cdot r^*(t) \tag{2.30}$$

This approach has been shown to lead to a more efficient and accurate model of the PA's behavior, particularly for the complex cross-dependency distortions such as AM-PM and PM-PM, making it a powerful technique in the design of modern DPD systems.

Chapter 3

Methods

3.1 Overview

This chapter introduces two novel architectures mainly to address the challenges in DPD. Firstly, the DeltaJANET model presented in Sec. 3.2 is an architecture that utilizes the principle of temporal sparsity to enhance the computational efficiency of the recurrent DPD model. Secondly, the Sec. 3.3 provides a detailed introduction to TC-JANET, a high-performance hybrid architecture. In this section, The motivation and design of its core components will be explained, including the TCN-based gate mechanism, the memory context window, and a PN enhanced JANET core.

3.2 DeltaJANET: Enhancing Digital Predistortion With Sparsity

In this section, we introduce the proposed DeltaJANET architecture. First, Subsec. 3.2.1 and 3.2.2 explain how the delta mechanism is integrated with the JANET recurrent unit and how this integration benefits model efficiency. Finally, Subsec. 3.2.3 describes the optimization process for the model.

3.2.1 Architecture Design

The motivation of the design for the DeltaJANET architecture is to reduce the computational complexity of traditional RNNs in DPD applications. This architecture is built upon the JANET recurrent unit [18], which is a simplification of the LSTM that adopts only a single forget gate. The standard JANET architecture is illustrated in Chapter 2 (see Fig. 2.6), and its state update equations are given by:

$$s_t = U_f h_{t-1} + W_f x_t + b_f \tag{3.1}$$

$$\tilde{c}_t = \tanh(U_c h_{t-1} + W_c x_t + b_c) \tag{3.2}$$

$$h_t = \sigma(s_t) \odot h_{t-1} + (1 - \sigma(s_t - \beta)) \odot \tilde{c}_t \tag{3.3}$$

where h_t is the hidden state, x_t is the input, and s_t and $\tilde{c}_t$ are the intermediate activations for the forget gate and candidate state, respectively. The JANET architecture is shown in Fig. 2.6.

The core innovation of DeltaJANET is the integration of the delta network principle [16], which exploits temporal sparsity by updating only those neurons whose state change exceeds a certain threshold. A state-holding variable, $\hat{h}_{t-1}$, stores the value of the hidden state at its last update. The change in the hidden state, Δh_t, is computed, and only the dimensions where this change is significant are propagated. The full formulation of the DeltaJANET recurrent update is as follows:

$$\Delta x = x_t - \hat{x}_{t-1}, \tag{3.4}$$
$$\Delta h = h_{t-1} - \hat{h}_{t-1}, \tag{3.5}$$
$$M_{s,t} := W_{xf}\Delta x + W_{hf}\Delta h + M_{s,t-1}, \tag{3.6}$$
$$M_{\tilde{c},t} := W_{xc}\Delta x + W_{hc}\Delta h + M_{\tilde{c},t-1}, \tag{3.7}$$
$$\tilde{c}_t = \tanh(M_{\tilde{c},t}), \tag{3.8}$$
$$h_t = \sigma(M_{s,t}) \odot h_{t-1} + (1 - \sigma(M_{s,t} - \beta)) \odot \tilde{c}_t. \tag{3.9}$$

where $M_{s,t}$ and $M_{\tilde{c},t}$ are memory values that accumulate the weighted changes. By applying this delta rule, a large portion of the computations can be skipped at each time step.

3.2.2 Theoretical Operation and Memory Access Savings

In DeltaJANET for DPD tasks, the computational operations and memory accesses are mainly determined by matrix-vector multiplication (M×V) and sparse matrix-vector multiplication (M×SV). Considering the equations for Δx_t, Δh_t, and their corresponding matrix operations, the dense and sparse computational cost (C_{comp}) and memory cost (C_{mem}) can be expressed as:

$$C_{\text{comp,dense}} = n^2, \tag{3.10}$$
$$C_{\text{comp,sparse}} = (1-\Gamma)n^2 + 2n, \tag{3.11}$$
$$C_{\text{mem,dense}} = n^2 + n, \tag{3.12}$$
$$C_{\text{mem,sparse}} = (1-\Gamma)n^2 + 4n. \tag{3.13}$$

where n is the dimension of input and hidden vectors, Γ is the overall temporal sparsity, accounting for zeros in Δx_t and Δh_t. From these equations, the theoretical computation speedup and memory access reduction for DeltaJANET are given by:

$$\text{Speedup} \approx \frac{n}{(1-\Gamma)n + 2}, \tag{3.14}$$
$$\text{Memory Access Reduction} \approx \frac{n+1}{(1-\Gamma)n + 4}. \tag{3.15}$$

For easier comparison and presentation, the number of active parameters during DeltaJANET inference can be estimated as:

$$\begin{aligned}\#\text{Active Parameters} = \#\text{DeltaJANET Parameters} \times (1 - \Gamma) \\ + \#\text{FC Parameters},\end{aligned} \tag{3.16}$$

where #FC Parameters refers to the fully connected layer parameters unaffected by sparsity.

These formulations show that DeltaJANET reduces the computational cost and memory access proportionally to sparsity (Γ). As Γ increases (higher sparsity), fewer neurons are active, resulting in significant efficiency improvements while maintaining linearization performance in DPD tasks.

3.2.3 Optimizations

The primary optimization for the DeltaJANET model involves tuning its key hyperparameters to balance the trade-off between computational efficiency and linearization performance. A systematic evaluation was conducted to identify the optimal value for the hyperparameter β, which decouples the forget and input gates. Unlike in general sequence modeling tasks, the unique characteristics of DPD signals were found to favor a negative β value. Furthermore, we evaluated multi-layer DeltaJANET configurations to find the most effective network depth. We also analyzed the relationship between parameter count and performance to select a hidden size that offers a good balance between linearization accuracy and computational cost.

3.3 TC-JANET: A Hybrid NN Architecture for High-Performance DPD

In this section, I'm going to introduce another major contribution TC-JANET, which is a hybrid architecture specifically designed for high performance. Subsec. 3.3.1 provide a detailed introduction the multi-stage architectural design of this model. While Sec. 3.3.2 describes the optimization process used to find a suitable configuration and training strategy.

3.3.1 Architecture Design

The design of the hybrid model was inspired by the characteristics of each subsystem. As illustrated in Fig. 3.1, the TC-JANET model synergistically combines a TCN-based gating mechanism with a phase-normalized JANET core and is composed of three main stages: a feature engineering front-end, the TCN, and a phase-normalized JANET recurrent loop.

The pipeline begins with a carefully designed feature engineering stage. Since the nonlinear behavior of a PA is known to be highly dependent on the signal's envelope, the input features must adequately represent both phase and amplitude information. While some architectures utilize polar coordinate features such as amplitude and the sin/cos of the phase angle, we opted

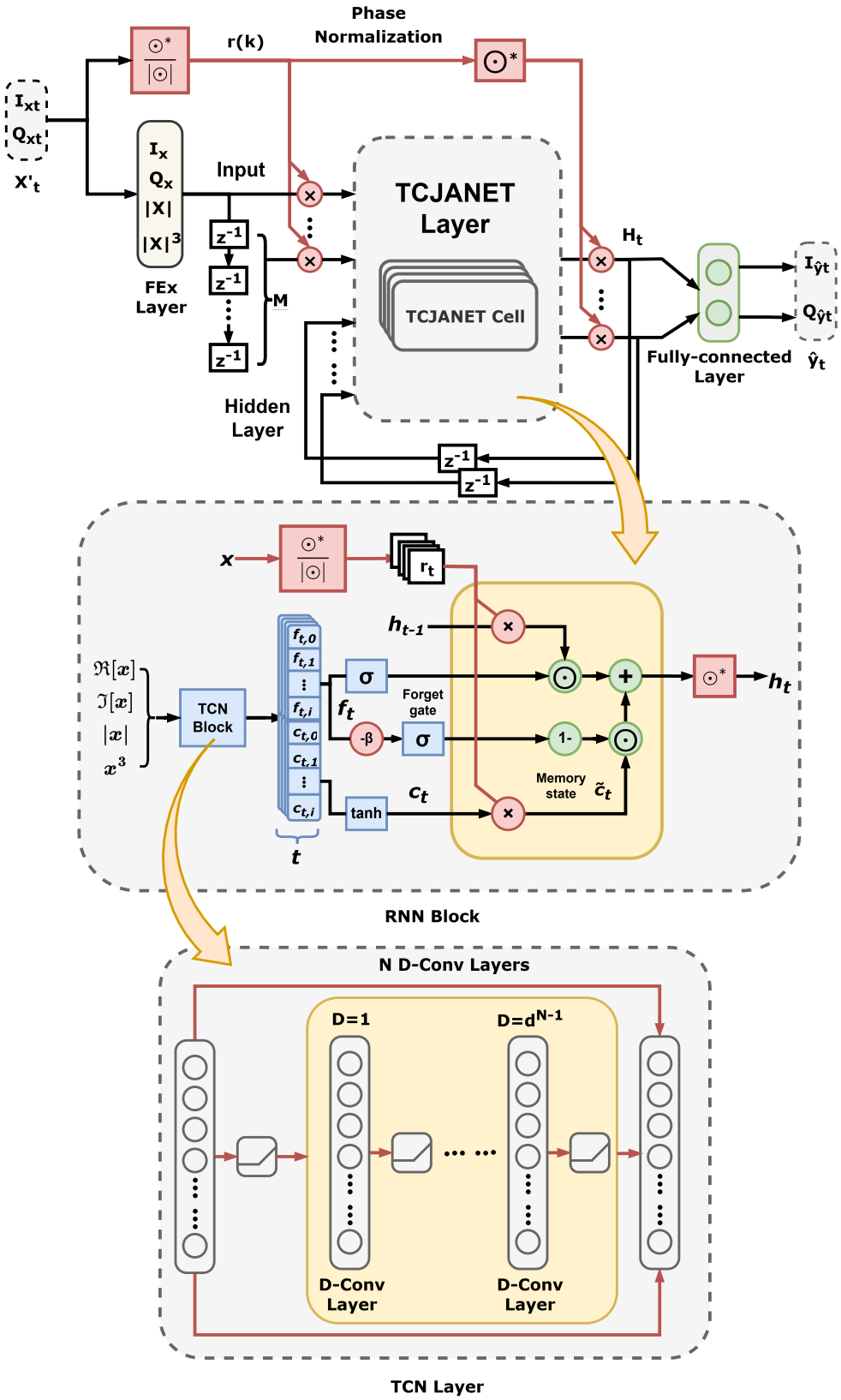

Figure 3.1: Architecture of the TC-JANET model, highlighting the TCN-based gating mechanism, memory context window, and the phase-normalized recurrent core.

for a Cartesian coordinate-based approach augmented with explicit envelope terms. For a given complex baseband input sequence $\mathbf{x} = [x(1), \ldots, x(L)]$, a feature vector $\mathbf{x}_{\text{feat}}(t)$ is constructed at each time step t. This vector comprises the baseband in-phase ($I(t)$) and quadrature ($Q(t)$) components, as well as the signal envelope (amplitude) $|x(t)|$ and its cubic term:

$$\mathbf{x}_{\text{feat}}(t) = [I(t), Q(t), |x(t)|, |x(t)|^3] \tag{3.17}$$

This choice provides a rich, non-redundant feature set that explicitly feeds the model with both the linear components of the signal and the crucial nonlinear envelope terms, facilitating the learning of AM-AM and AM-PM distortions.

To provide the model with explicit short-term memory, a "Memory Context Window" module is employed. The feature vectors from the current time step and M previous time steps are concatenated to form a wide input vector $\mathbf{z}(t)$ for the TCN:

$$\mathbf{z}(t) = [\mathbf{x}_{\text{feat}}(t), \mathbf{x}_{\text{feat}}(t-1), \ldots, \mathbf{x}_{\text{feat}}(t-M)] \tag{3.18}$$

where M is the hyperparameter `memory_depth`.

This sequence of wide vectors, $\mathbf{Z} = [\mathbf{z}(1), \ldots, \mathbf{z}(L)]$, is then processed by the TCN. In our architecture, the TCN functions as a highly efficient, non-recurrent gating mechanism, serving as a lightweight alternative to the computationally expensive matrix multiplications found in conventional RNN gating. Through its stack of dilated convolutions, the TCN effectively captures long-range temporal dependencies across the entire sequence. The output of this convolutional feature extraction process is two pre-computed sequences for the recurrent core: the forget gate signals $\mathbf{f} = [f_1, \ldots, f_L]$ and the candidate state signals $\mathbf{c} = [c_1, \ldots, c_L]$.

JANET recurrent unit still plays an indispensable role in the whole architecture, processing the gate signals produced by TCN at each time step. To avoid learning redundant mappings, PN is applied directly to the state transition. The hidden state h is represented as a complex value, $h \in \mathbb{C}^{N_h}$, and a time-varying rotation factor r_t is computed from the input signal:

$$r_t = \frac{x^*(t)}{|x(t)| + \epsilon} \tag{3.19}$$

Both the previous hidden state h_{t-1} and the current candidate state c_t are then projected into a normalized domain:

$$h_{\text{norm},t-1} = h_{t-1} \odot r_t \tag{3.20}$$

$$c_{\text{norm},t} = c_t \odot r_t \tag{3.21}$$

The JANET state update is then performed within this normalized domain. s_t is the value of the forget gate before the activation, the updated hidden state, $h'_{\text{norm},t}$, is computed as:

$$h'_{\text{norm},t} = \sigma(s_t) \odot h_{\text{norm},t-1} + (1 - \sigma(s_t - \beta)) \odot c_{\text{norm},t} \tag{3.22}$$

The resulting hidden state is then de-normalized back to the absolute phase domain before being passed to the next time step:

$$h_t = h'_{\text{norm},t} \odot r_t^* \tag{3.23}$$

Finally, the sequence of output hidden states $[h_1, \ldots, h_L]$ is passed through a fully connected layer to produce the predistorted output signal.

3.3.2 Optimizations

The performance of the TC-JANET is enhanced by two key structural optimizations that improve its feature representation for the DPD task.

A primary optimization is the use of a sliding **context window**, a principle conceptually similar to the Direct Memory Inputs explored in architectures like the VDLSTM [14]. In a standard RNN, all historical information must be compressed into the recurrent hidden state, h_{t-1}, which can dilute the influence of the most recent input samples. Our architecture avoids this by providing the TCN with an explicit, uncompressed view of the recent signal history, ensuring that the most salient short-term information is losslessly presented to the feature extractor.

Another critical optimization lies in the sophisticated gating mechanism that precedes the recurrent loop. Prior work, such as Block-Oriented Just Another Network (BO-JANET) [21], has often used a linear Finite Impulse Response (FIR) filter for this purpose, which computes a weighted average over a sliding window of inputs:

$$y(t) = \sum_{i=0}^{M} b_i \cdot z(t-i) \tag{3.24}$$

where $z(t)$ is the input feature vector and b_i are the learnable filter coefficients. However, the expressive power of such a linear filter is fundamentally limited. Therefore, in our TC-JANET architecture, this role is fulfilled by a **TCN**, which acts as a deep, non-linear generalization of a traditional FIR filter bank. The TCN's superiority stems from its inherent non-linearity, a deep hierarchical structure, and a large receptive field enabled by dilated convolutions, making it a key factor in the model's high performance.

Chapter 4

Experimental Results

4.1 DeltaJANET

This section presents the experimental evaluation of the DeltaJANET model. First, Subsec. 4.1.1 details the robust experimental methodology, including the framework, implementation details, dataset, and evaluation metrics used. Following this, Subsec. 4.1.2 presents and analyzes the extensive results, focusing on the optimization of the hyperparameters and the linearization performance and efficiency compared to prior and peer models.

4.1.1 Experimental Setup

Framework and Implementation

The experiments were conducted using the OpenDPD End-to-End (E2E) learning framework, a comprehensive platform for evaluating DPD architectures implemented in PyTorch [19]. The DeltaJANET architecture was integrated into this framework as a backbone, enabling a consistent and reproducible evaluation of its ability to linearize wideband PAs. The E2E learning structure employed backpropagation through the DPD model cascades with a pre-trained PA behavioral model to iteratively optimize performance. The training process was conducted with frame-based processing, where each frame contained 50 samples over 100 epochs, and the Adam optimizer with an initial learning rate of 5×10^{-3} and a batch size of 64, `ReduceLROnPlateau` was adopted as the learning rate scheduler with a setting of (`patience=10`, `factor=0.5`) to ensure convergence.

Datasets and Evaluation Metrics

The experiments utilized a dataset generated from a 3.5 GHz GaN Doherty PA, driven by a TM3.1a 5×40-MHz (200-MHz) 256-QAM Orthogonal Frequency Division Multiplexing (OFDM) baseband I/Q signal at 41.5 dBm average output power. The test signal's PAPR measured 10.01 dB. The ELCA40 dataset which is mainly adopted in this experiment consists of I/Q-modulated signals with a main channel bandwidth of 40 MHz. The data was divided into

training (60%), validation (20%), and test (20%) subsets. Three key metrics were employed to evaluate performance: ACPR, NMSE, and Sparsity, defined as the fraction of inactive neurons, which quantifies the computational efficiency of the DeltaJANET.

Initialization Method

The weights of the DeltaJANET backbone were initialized by a combination of Xavier initialization for the input layer and orthogonal initialization for recurrent layers. For the bias initialization, a comparison between Chrono initialization [18] and a simple zero initialization was conducted. Chrono initialization is designed for long-term dependencies, but its effectiveness was limited by the rapid dynamics of DPD signals. We compared the almost tailored Chrono method with zero initialization. Table 4.1 and Fig. 4.1 concluded that zero initialization provided similar final performance with faster converging speed. Therefore, zero initialization will be selected for the rest of the experiments.

Table 4.1: Initialization Method Comparison

Metric	Zero init	Chrono Init
ACPR (dBc)	-53.64	**-53.98**
NMSE (dB)	**-45.19**	-44.31

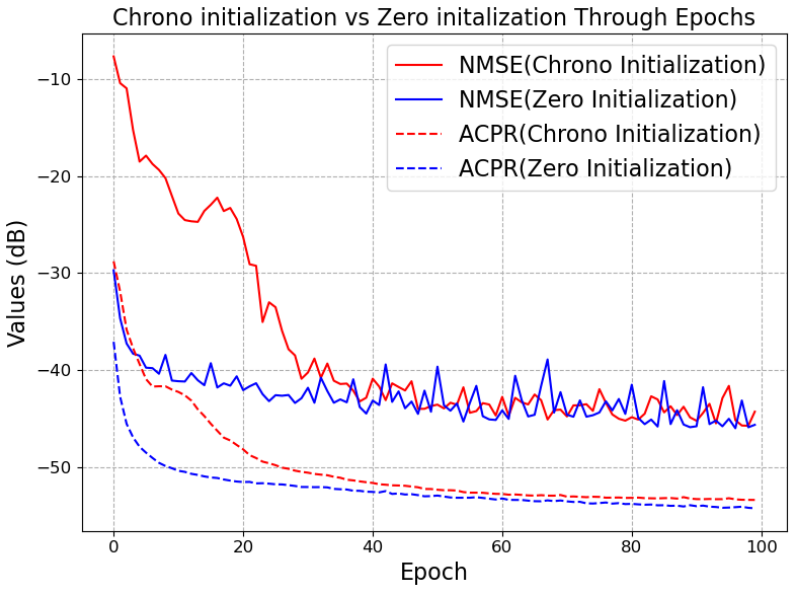

Figure 4.1: Chrono initialization vs. Zero initialization

4.1.2 Results and Discussion

Parameter Optimization

In order to balance the trade-off between computational efficiency and linearization performance, we swept a range of hidden sizes from 8 to 48. As shown in Fig. 4.2, a hidden size of 18 (corresponding to approximately 1000 parameters) seems to be an optimal point. Models with fewer parameters lacked the expressive capacity to fully capture the PA's nonlinearities, while increasing the parameter count beyond this point increased computational cost without proportional gain.

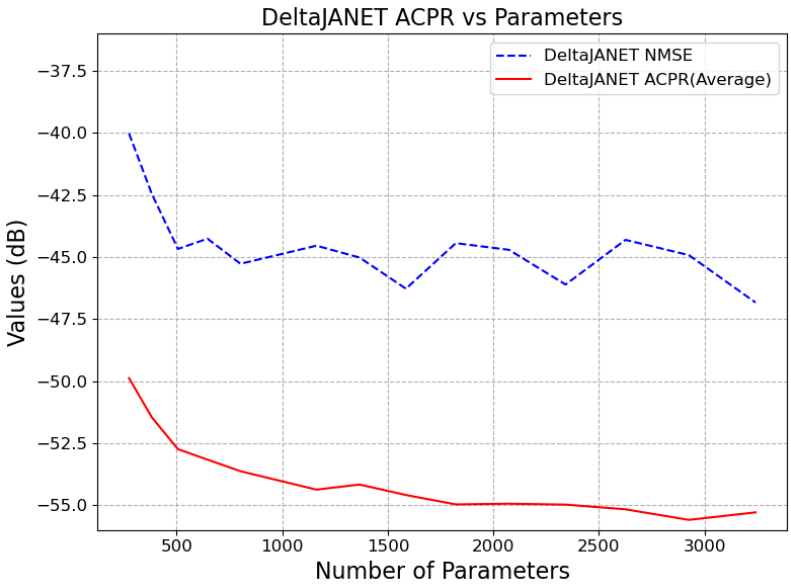

Figure 4.2: Performance vs. Number of Parameters for DeltaJANET.

Performance with Multi-Layer DeltaJANET

To evaluate the effect of network depth on performance, we compared a 2-layer DeltaJANET with the standard single-layer version. As shown in Fig. 4.3, the single-layer network consistently outperformed the 2-layer structure. We hypothesize that sparse updates in a multi-layer network may hinder gradient propagation across layers. The results suggest that a single-layer DeltaJANET with a larger hidden size is the most effective configuration. This approach fully utilizes the model's efficiency while avoiding the risks of overfitting or vanishing gradients associated with increased depth.

Effect of β

To verify the hypothesis from the original JANET paper that the β hyperparameter is dataset-dependent [18], our experiments for the DPD task found an optimal value of approximately -2.1 (Fig. 4.4)—in contrast to the paper's $\beta = 1$ for MNIST—suggesting that the dynamic nature of RF signals benefits more from rapid adaptation than from long-term memory retention.

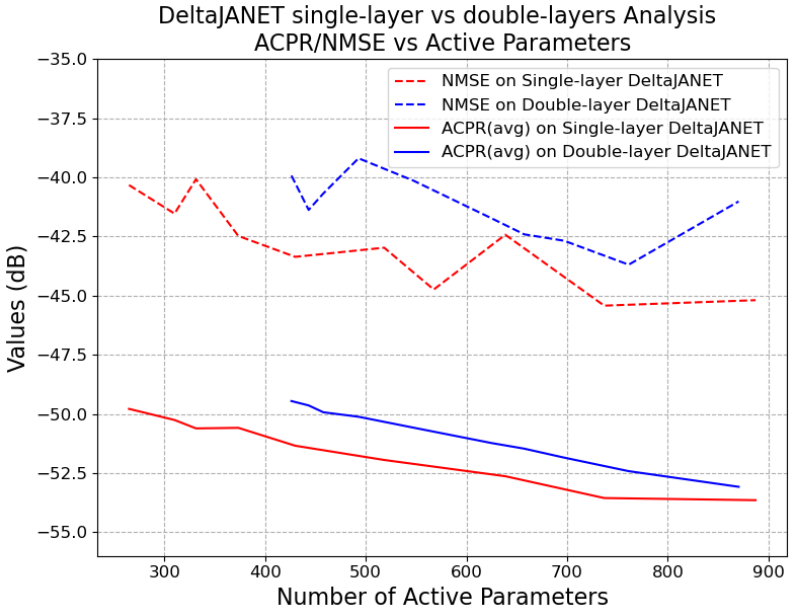

Figure 4.3: ACPR/NMSE comparison of single-layer vs. double-layer DeltaJANET.

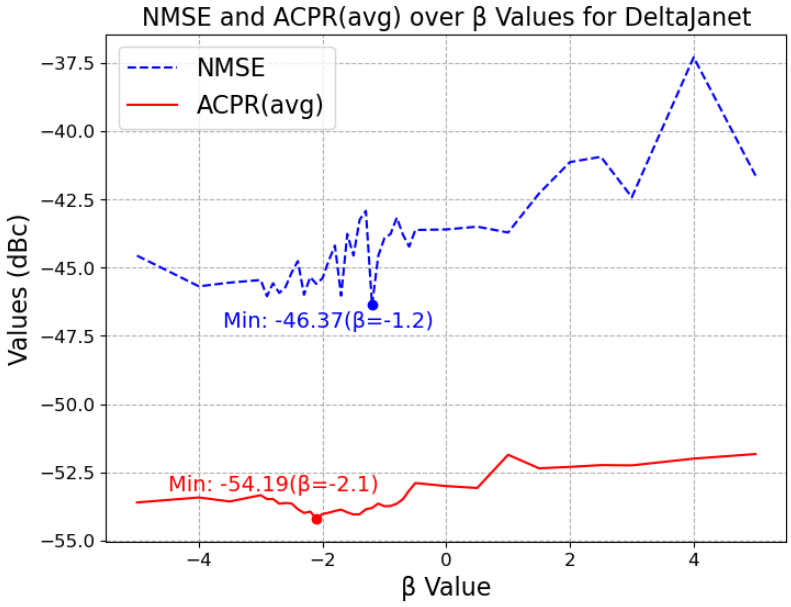

Figure 4.4: NMSE and ACLR over β values for DeltaJANET.

Comparison with DeltaDGRU

In practical DPD implementation, especially on those resource-constrained environments such as FPGAs or edge devices, minimizing the number of active parameters for lightweight models (e.g., under 1000 parameters) is a critical objective. To evaluate the efficiency of our proposed model in this context, DeltaJANET was benchmarked against a peer model DeltaDGRU architecture. The comparison was conducted under the optimal configuration of $\beta = -2.1$ and approximately 1000 total parameters for both models.

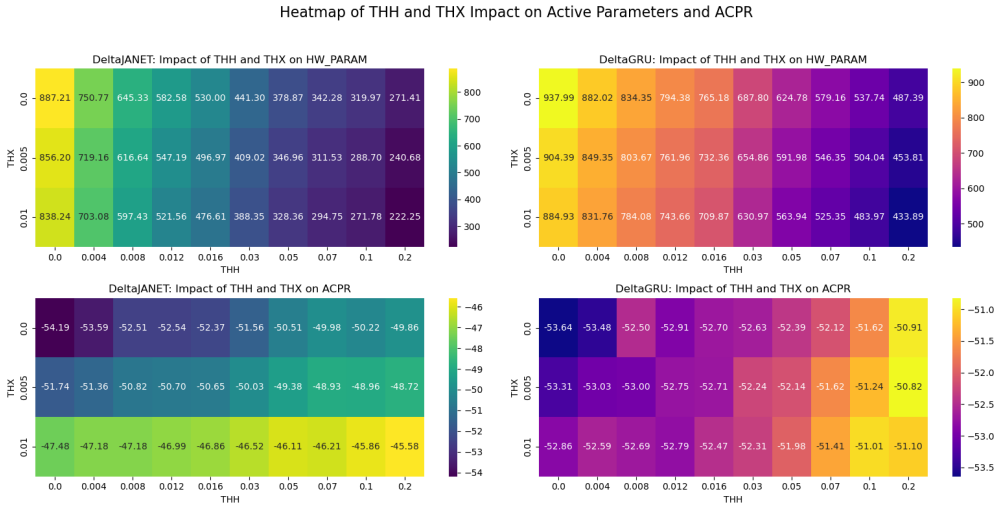

Figure 4.5: Comparison of DeltaJANET and DeltaDGRU across Active Parameters and ACPR.

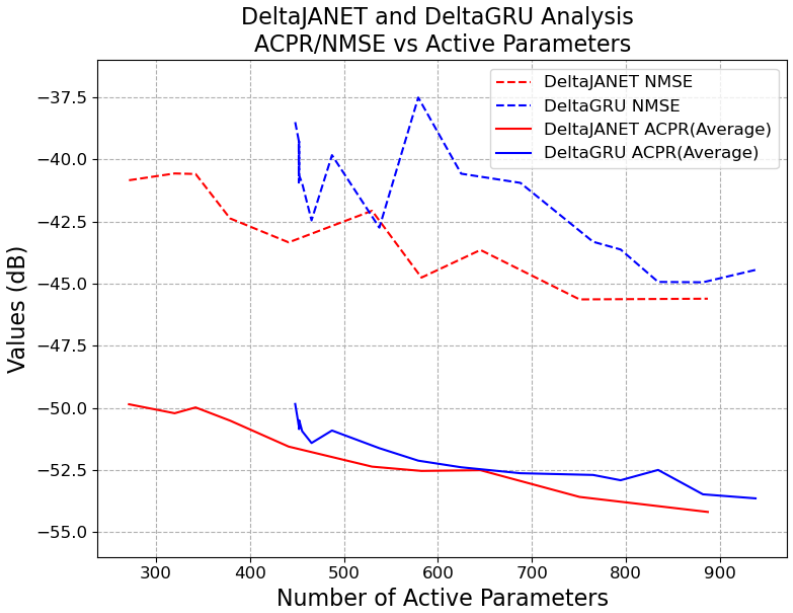

Figure 4.6: ACPR/NMSE comparison of DeltaJANET and DeltaDGRU under Equivalent Active Parameters.

The results presented in Fig. 4.5 and Fig. 4.6, highlight the great advantages of DeltaJANET, which consistently demonstrated superior temporal sparsity, achieving a higher fraction of inactive neurons than DeltaDGRU. This directly results in reduced computational cost and energy consumption during inference. To be specific, DeltaJANET achieved a better sparsity at a comparable total parameter count and reduced its active parameters to approximately 216, significantly fewer than DeltaDGRU for a similar level of linearization performance in both NMSE and ACLR. This superior parameter efficiency indicates that the architectural simplicity of JANET combined with the delta update mechanism, provides a more resource-efficient and higher liearization performance for real-time DPD applications.

4.2 TC-JANET

This section evaluates the TC-JANET model, our proposed architecture for high-performance DPD. Subsec. 4.2.1 details the experimental setup, training configuration, and the models used for comparison, and the benchmark for evaluation. Subsec. 4.2.2 then presents an ablation study of architectural components, and discusses the model's final performance across various complexity budgets against prior models.

4.2.1 Experimental Setup

Framework and Implementation

The experimental setup for the TC-JANET model followed a robust methodology to ensure a fair and comprehensive evaluation. All experiments were also conducted within the OpenDPD E2E learning framework, . The E2E DPD learning process involves backpropagation through a pre-trained 2751-parameter, -40.04 dB-NMSE DGRU PA behavioral model [19] with the measured PA dataset.

To ensure statistical significance and robustness, all key results were validated across 5 random seeds. The models underwent training for 200 epochs using the AdamW optimizer with an initial learning rate of 5×10^{-3} and a batch size of 64. An aggressively configured `ReduceLROnPlateau` learning rate scheduler (`patience=5`, `factor=0.25`) was employed. Based on a systematic study of initialization methods, the TCN's default Kaiming initialization was used for all experiments as it provided the most stable and superior average performance.

Datasets and Evaluation Metrics

The experiments utilized a dataset generated from a 3.5GHz GaN Doherty PA, driven by a TM3.1a 5×40-MHz (200-MHz) 256-QAM OFDM baseband I/Q signal at 41.5 dBm average output power. The test signal's PAPR measured 10.01 dB. The dataset, comprising 98304 samples, was divided into a 60% training set, a 20% validation set for early stopping, and a 20% test set for

performance evaluation. The core evaluation metrics of ACLR, EVM, and NMSE were used to assess performance.

4.2.2 Results and Discussion

This section evaluates the proposed TC-JANET model based on systematic, multi-seed experiments. The analysis includes two main parts: a benchmark against prior DPD models across different complexity budgets, and an ablation study of the core architectural components.

Ablation Study of Architectural Components

To validate the contribution of each key innovation within our proposed architecture, we conducted a systematic ablation study. The study starts from a standard JANET baseline and progressively adds the core components of our final model: the TCN-based gating mechanism, the PN mechanism, and a memory context window for the input. The DGRU model is also included as a baseline to represent a standard gated recurrent architecture. The performance of each variant, configured with approximately 500 parameters, is summarized in Table 4.2.

Table 4.2: Performance Comparison of Different DPD Model Configurations.Based on a DGRU PA Pre-trained Model with 2751 parameters on the dataset of ELCA200 Validation

Model[*]	**ACLR (dBc)**	**EVM (dB)**	**NMSE (dB)**
DGRU ∼487	-49.57	-41.20	-40.54
TCN ∼507	-50.16	-41.93	-41.59
JANET ∼506	-51.35	-43.37	-42.42
TC-JANET ∼506	-50.91	-45.47	-43.99
+ Context Window ∼482	-49.65	-38.54	-38.29
TC-JANET with PN ∼499	-50.76	-45.57	-43.85
+ Non-causal ∼487	-50.74	-45.63	-44.12
+ Causal Context Window ∼503	**-52.77**	**-47.12**	**-45.55**

[*] All baseline models(DGRU, TCN, and JANET) were provided with the full set of concatenated input features: I, Q, $|x(t)|$, $|x(t)|^3$, $\cos(\phi)$, and $\sin(\phi)$, while the TC-JANET only took I, Q, $|x(t)|$, $|x(t)|^3$.

The first step of our study was to replace the computationally expensive, matrix-multiplication-based gating of a standard JANET with our proposed TCN-based gating mechanism, creating the TC-JANET model. As the results show, this architectural shift alone yields a significant performance improvement in EVM and NMSE, demonstrating the superior capability of the TCN to extract rich, long-range temporal features from the input sequence.

Next, we introduced the PN mechanism into the recurrent core, resulting in the Phase-Normalized TC-JANET model. The motivation for this is that PA

distortion is primarily dependent on the signal envelope and its history, not its absolute phase. Without PN, the network is forced to learn redundant mappings for signal trajectories that are physically similar but have different phase rotations. By dynamically rotating the complex-valued hidden and candidate states to a zero-phase reference before the update, and de-normalizing them afterward, these trajectories is effectively aligned by the PN structure in the feature space. This enables the network to learn a single, normal-compliant distorted mapping, thus bringing about a more efficient and robust learning process.

The most significant performance improvement was achieved with the final addition of the input context window. This complete TC-JANET model reached an ACLR of **-52.77 dBc**, outperforming all other variants. We attribute this to a synergistic effect. The context window provides the TCN an explicit view of recent signal history, improving its feature extraction. The PN mechanism then operates on these higher-quality signals, resulting in a more accurate linearization.

Conversely, applying the context window to the TC-JANET **without** Phase-Normalization resulted in a notable performance degradation. This finding is crucial, as it suggests that simply increasing the input information to the TCN without handling the signal's constantly changing phase can be counterproductive. Without PN, the model is presented with a high-dimensional input where the I/Q components are always rotating, forcing it to learn redundant mappings. This unnecessarily complicates the learning process and prevents the model from converging to an effective solution.

Furthermore, we explored a "non-causal context window" which incorporated future time steps (e.g., $x(t+1)$) into the input. While this approach demonstrated faster initial convergence, its final performance was inferior to the causal-only model. We identified two likely reasons for this: maintaining a fixed parameter budget required reducing other model dimensions (such as `rnn_hidden_size`), which may have constrained expressive capacity. More importantly, the future sample is an overly strong predictive feature, leading to "shortcut learning." The model became too reliant on this feature and failed to learn complex long-term dependencies, hindering its convergence to a more optimal solution.

This comprehensive study confirms that the specific combination of the gating mechanism enhanced by TCN, a causal input context window, and the PN mechanism is essential to achieve the final performance.

Impact of Architectural Hyperparameters

An ablation study, illustrated in 4.2.2, confirmed the positive contribution of each key component. The introduction of PN provided a notable improvement over a baseline TC-JANET, and a further significant gain was achieved with the addition of Memory Context Window, validating our architectural design choices.

We then investigated the impact of the TCN's `kernel_size` by testing values of 3, 5, and 7 while keeping a fixed parameter budget of $\sim$500. The average

performance over 5 random seeds for each configuration is shown in Fig. 4.7. The results indicates that a `kernel_size` of 5 as the optimal choice. Although a smaller kernel of 3 was highly stable and showing faster convergence but yielded slightly worse linearization performance. In contrast, a larger kernel of 7 led to a significant performance drop, likely because a wide local context is detrimental under complexity constraints.

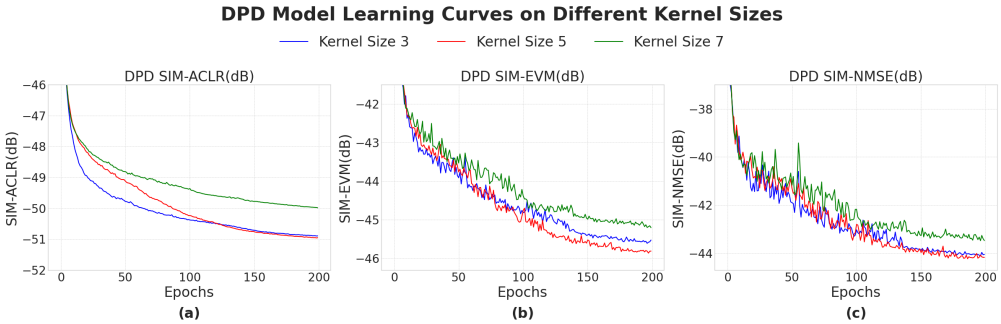

Figure 4.7: Learning curve comparison for TC-JANET with different TCN kernel sizes.

In addition, we analyzed how the `memory_depth` (MD), which controls the context window length, will impact the model performace. The results are presented in Fig. 4.8. A clear optimal point was observed at MD=2, which achieved an ACLR of **-52.77 dBc**. This was a significant improvement over both MD=0 and MD=1. However, increasing the depth further to MD=3 led to a performance degradation. This finding indicates that MD=2 provides the best balance of sufficient context without adding noise from distant samples, confirming the window's depth is a critical factor.

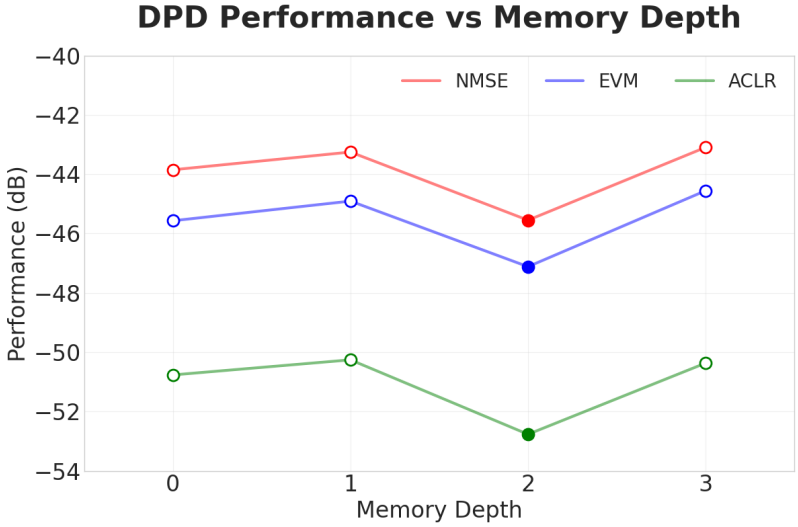

Figure 4.8: ACLR and EVM performance as a function of memory depth (MD).

Benchmark Against Prior Works

A comprehensive benchmark was performed to evaluate the proposed TC-JANET against various DPD models at three complexity levels 200, 500, and 1000

parameters. The complete performance summary is presented in Table 4.3, where all reported results are the average of the best performance from 5 random seeds to ensure statistical significance.

Table 4.3: Comprehensive Performance Comparison of Proposed TC-JANET against Prior DPD Models based on a Fixed DGRU PA Pre-trained Model on the dataset of ELCA_200 Validation, Averaged across 5 Random Seeds

Parameters	Model	ACLR Mean (dBc)	EVM Mean (dB)	NMSE Mean (dB)
∼1000	LSTM ∼1038	-51.43 ±0.35	-45.22 ±0.43	-43.83 ±0.41
	GRU ∼994	-51.89 ±0.47	-46.58 ±0.66	-44.95 ±0.58
	DGRU ∼1041	-52.76 ±0.50	-45.76 ±0.25	-44.48 ±0.30
	TCN ∼1006	-51.60 ±0.32	-42.95 ±0.20	-42.62 ±0.22
	JANET ∼1066	-53.80 ±0.37	-48.40 ±0.53	-46.68 ±0.43
	DVR-JANET ∼1097	-53.96 ±0.52	-49.30 ±0.53	-47.22 ±0.48
	TC-JANET ∼1013	**-54.66** ±0.88	**-49.92** ±0.76	**-47.93** ±0.78
∼500	LSTM ∼488	-46.87 ±0.80	-40.51 ±0.83	-39.42 ±0.76
	GRU ∼519	-47.98 ±0.70	-41.29 ±0.40	-40.31 ±0.51
	DGRU ∼486	-49.29 ±0.27	-41.41 ±0.40	-40.64 ±0.30
	TCN ∼511	-49.02 ±0.37	-41.17 ±0.22	-40.77 ±0.26
	JANET ∼506	-50.89 ±0.27	-43.50 ±0.18	-42.38 ±0.16
	DVR-JANET ∼509	-50.40 ±0.66	-44.80 ±1.17	-43.15 ±1.03
	TC-JANET ∼503	**-51.06** ±0.96	**-45.94** ±0.63	**-44.37** ±0.75
∼200	LSTM ∼192	-42.14 ±0.66	-31.21 ±0.58	-30.85 ±0.51
	GRU ∼194	-42.75 ±0.53	-33.14 ±1.15	-32.84 ±1.09
	DGRU ∼186	-44.21 ±0.75	-31.07 ±0.21	-30.27 ±0.24
	TCN ∼214	-42.64 ±1.05	-35.60 ±1.58	-34.84 ±1.31
	JANET ∼226	-45.19 ±0.52	-36.18 ±1.72	-35.63 ±1.47
	DVR-JANET ∼215	-45.96 ±1.22	-39.84 ±1.32	-38.53 ±1.21
	TC-JANET ∼212	**-46.03** ±0.87	**-40.28** ±1.23	**-39.08** ±1.06

At the low-complexity level (200 parameters), the proposed TC-JANET demonstrates its superior efficiency. As shown in Fig. 4.9, our model achieves a mean ACLR of -46.03 dBc, outperforming the DVR-JANET, which is specifically noted in prior work for its ability to achieve high linearization performance with a small number of parameters [12]. This result highlights the inherent efficiency of the proposed architecture even at a small scale.

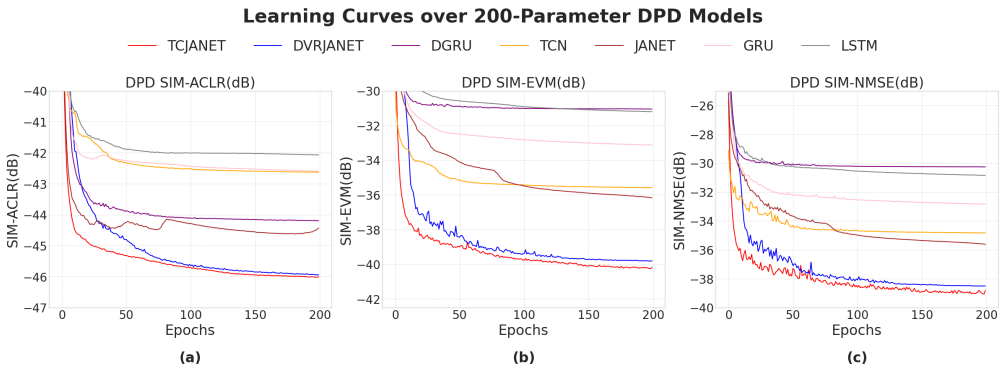

Figure 4.9: Learning curve comparison of DPD models with approximately 200 parameters.

As we increase the model capacity to the medium-scale (∼500 parameters), the advantages of the TC-JANET architecture become more apparent. Table 4.3 confirms the model achieves a mean ACLR of -51.06 dBc, outperforming other architectures in this complexity class. The model's learning curve, shown in Fig. 4.10, indicates both a rapid convergence and a high final performance level. This linearization quality is further validated in both the frequency and modulation domains. The power spectral density (PSD) plot in Fig. 4.11 shows effective suppression of out-of-band spectral regrowth, while the AM/AM and AM/PM characteristics in Fig. 4.12 confirm the correction of the PA's core nonlinearities, with the output points tightly clustered along the ideal linear gain.

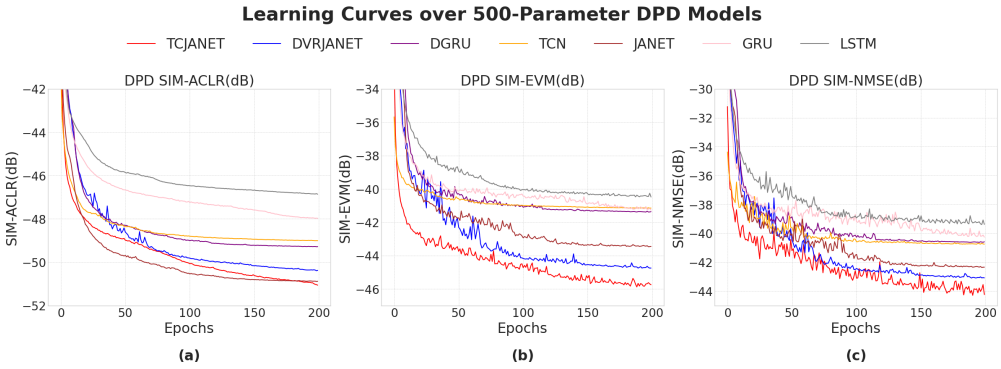

Figure 4.10: Learning curve comparison of DPD models with approximately 500 parameters.

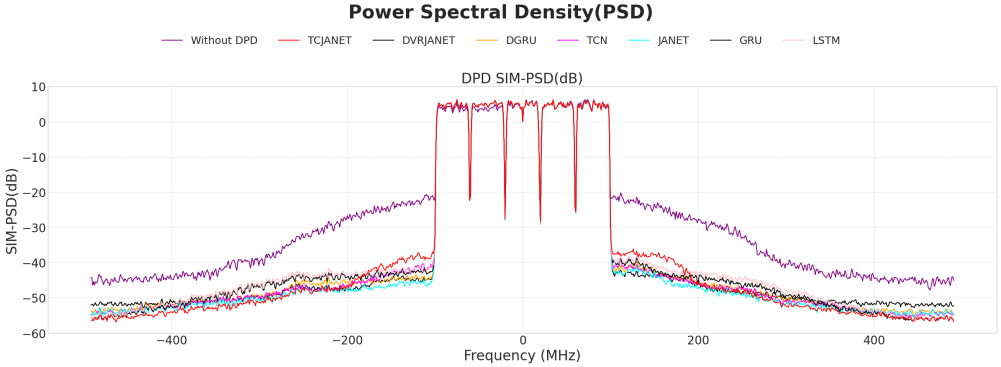

Figure 4.11: PSD comparison for the 500 parameter models.

The most significant finding of our research is revealed in the high-capacity (∼1000 parameters) benchmark. At this scale, the TC-JANET model demonstrates its exceptional scalability and establishes a new performance level, as clearly visualized in Fig. 4.13. It achieves a robust mean ACLR of **-54.66 dBc**, widening its performance gap with the next best models, DVR-JANET (-53.96 dBc) and JANET (-53.80 dBc). This result proves that the performance bottleneck in many DPD models is indeed model capacity, and that our proposed architecture is uniquely effective at leveraging increased complexity to achieve

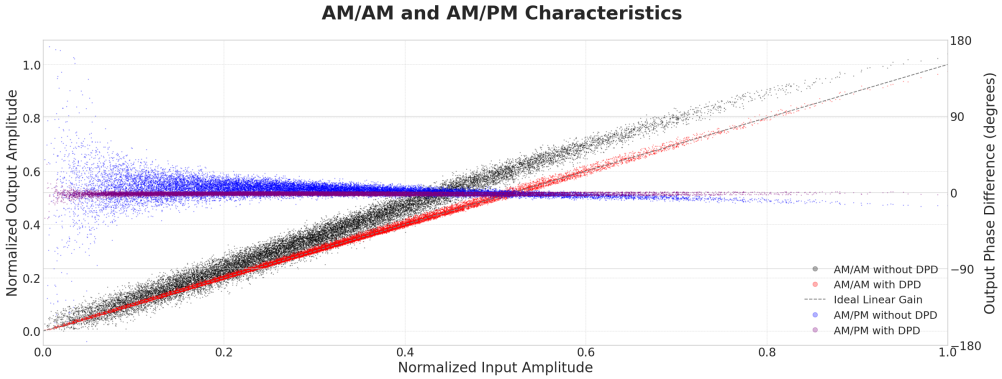

Figure 4.12: AM/AM and AM/PM characteristics for the 500 parameter models.

superior linearization. The consistent outperformance across all three metrics (ACLR, EVM, and NMSE) validates the comprehensive modeling capabilities of the TC-JANET.

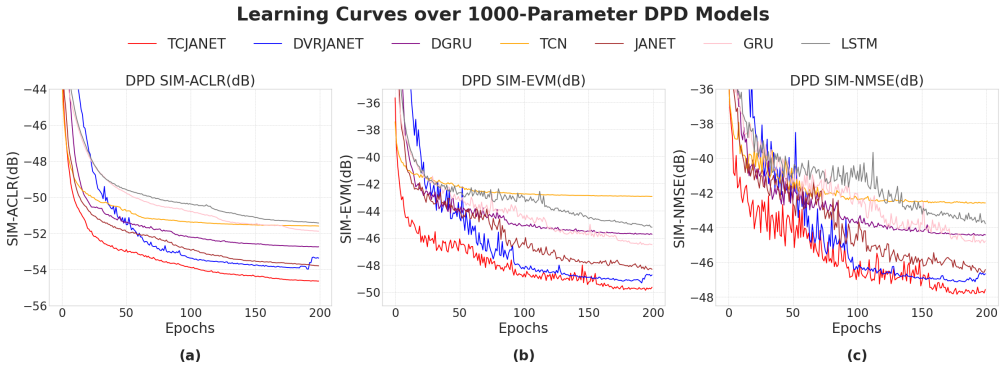

Figure 4.13: Learning curve comparison of DPD models with approximately 1000 parameters.

Chapter 5

Conclusions

This thesis proposed two novel neural network architectures for the DPD of wideband PA and presented a comprehensive investigation, progressing from an initial focus on computational efficiency to the final pursuit of superior linearization performance.

5.1 Conclusion

Based on the JANET and Delta Network mechanism, our first contribution, the DeltaJANET model, was proposed and evaluated. The experimental results confirmed our initial hypothesis: By leveraging temporal sparsity, the DeltaJANET architecture significantly reduces computational complexity compared to its non-sparse counterparts. The analysis demonstrated that DeltaJANET inherits the lightweight and expressive capacity from JANET with absorbing the key advantages from Delta mechanism, achieving strong linearization performance comparable to standard RNNs while operating with a significantly reduced number of active parameters. This successfully validated DeltaJANET as a highly parameter-efficient and computationally aware DPD solution.

Following the exploration of efficiency, a systematic and extensive series of experiments was conducted to identify the optimal hyperparameters for the DeltaJANET model. This process revealed several key findings specific to the application of sparse RNNs in the DPD context. We investigated the bias initialization of the JANET core, comparing Chrono initialization against a simple zero initialization. Although Chrono initialization is designed to enhance long-term memory, we found it less suitable for the rapidly changing signals characteristic of the DPD task. Our experiments concluded that a simple zero initialization provided faster convergence with comparable final performance, and was therefore selected. Furthermore, our results consistently showed that, a negative β value is optimal for DPD, which is totally distinct to the original paper's $\beta = 1$ for it's MNIST dataset. This confirms that β as a dataset-dependent hyperparamer and the importance of tuning it to favor adaptation over memory retention. The study also explored the impact of network depth, revealing that a single-layer architecture consistently outperformed multi-layer configurations. This suggests that for sparse

acpRNN in DPD, increasing the width of the hidden layer is a more effective strategy for enhancing model capacity than increasing its depth.

Another key finding of this research is the successful validation of a hybrid architecture, the TC-JANET, across a range of complexity budgets. Within a low complexity setting (212 parameters), the TC-JANET already establishes itself as a highly efficient architecture, achieving a robust average ACLR of -46.03 dBc and performing competitively with specialized models like DVR-JANET. As the model capacity is increased to the medium-scale (503 parameters), its advantages become more pronounced, with the average ACLR improving to -51.06 dBc, surpassing other architectures in its class. As the number of parameters goes higher (1013 parameters, the scaled-up TC-JANET achieves a robust average ACLR of **-54.66 dBc**. This result represents a decisive performance advantage over the full suite of established DPD models, including the highly competitive DVR-JANET. The performance improvement observed with each increase in model capacity confirms that the synergistic combination of an TCN-based gating, a recurrent core augmented with phase normalization, and a memory context window is a highly effective strategy for overcoming the performance limitations of existing DPD solutions.

Although this thesis presents promising results, there are still several limitations that should be noted. The evaluation was entirely conducted within the OpenDPD simulation framework, which utilized a pre-trained PA behavioral model. These studies did not take into account the impact of actual hardware defects. Situations like impedance mismatch or hardware noise can occur in physical measurement Settings. Furthermore, the hyperparameter optimization is relatively systematic but not exhaustive. The performance of the proposed model can still be improved with more extensive tunings.

5.2 Outlook

The research conducted in this thesis opens up several promising avenues for future investigation:

1. **Real-World Measurement and Validation** While the OpenDPD framework provides a robust and reproducible simulation environment, like what is mentioned in Sec. 5.1, the ultimate validation of any experiment relies on its performance from a physical hardware testbench. Therefore, future work should focus on implementing and evaluating the proposed DeltaJANET and TC-JANET models on a real-world RF measurement setup to assess their linearization capabilities under practical hardware constraints.

2. **Hardware-Aware Optimizations: Quantization and Implementation** The efficiency of the DeltaJANET makes it a strong candidate for low-power hardware implementation. Future work could explore its deployment on FPGA or ASIC platforms, inspired by existing accelerator designs for sparse RNNs [7]. Quantization are critical for making these models suitable for edge devices. Techniques like mixed-precision quantization,

for example, can provide an additional optmization on model's memory footprint and replace costly floating-point operations by fixed-point [20]. A further research direction is to develop a quantization-aware training scheme for both DeltaJANET, with the goal of co-designing the algorithm and a dedicated hardware accelerator, similar to the approach in [8].

3. **Comprehensive Hyperparameter Search for TC-JANET** Although we performed a systematic optimization of the TC-JANET, the vast hyperparameter space of this complex model warrants further exploration. A more extensive search, potentially employing Automated Machine Learning (AutoML) techniques, could uncover even better configurations of Kernel Size, Hidden Channels of TCN, and other parameters, potentially pushing the performance boundary further.

4. **Structural Optimization of TC-JANET** The TC-JANET's performance could be further enhanced through advanced structural modifications. A promising direction, inspired by the DVR-JANET [12], is the adoption of a multi-headed TCN. In this structure, amplitude-based features (e.g., $|x(t)|, |x(t)|^3$) and phase-based features (I/Q) are processed in separate, specialized pathways before being fused. This would align the architecture more closely with the physical nature of AM-AM and AM-PM distortion and could lead to improved performance and stability.

Bibliography

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[2] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734, 2014.

[3] Huanqiang Duan, Manno Versluis, Qinyu Chen, Leo C. N. de Vreede, and Chang Gao. TCN-DPD: Parameter-Efficient Temporal Convolutional Networks for Wideband Digital Predistortion. In *Proceedings of the IEEE MTT-S International Microwave Symposium (IMS)*, 2025.

[4] Changsoo Eun and E. J. Powers. A New Volterra Predistorter Based on the Indirect Learning Architecture. *IEEE Transactions on Signal Processing*, 45(1):223–227, Jan 1997. doi: 10.1109/78.552219.

[5] Arne Fischer-Bühner, Lauri Anttila, Manil Dev Gomony, and Mikko Valkama. Phase-Normalized Neural Network for Linearization of RF Power Amplifiers. *IEEE Microwave and Wireless Technology Letters*, 33(9): 1357–1360, 2023. doi: 10.1109/LMWT.2023.3290980.

[6] Arne Fischer-Bühner, Lauri Anttila, Manil Dev Gomony, and Mikko Valkama. Recursive Neural Network With Phase-Normalization for Modeling and Linearization of RF Power Amplifiers. *IEEE Microwave and Wireless Technology Letters*, 34(6):809–812, jun 2024. doi: 10.1109/LMWT.2024.3393859.

[7] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck. DeltaRNN: A Power-Efficient Recurrent Neural Network Accelerator. In *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2018.

[8] Chang Gao, Antonio Rios-Navarro, Xi Chen, Tobi Delbruck, and Shih-Chii Liu. EdgeDRNN: Enabling low-latency recurrent neural network

edge inference. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2020. doi: 10.1109/AICAS48895.2020.9074001.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*, 2015. doi: 10.48550/arXiv.1512.03385. URL https://arxiv.org/abs/1512.03385.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] A. Katz, J. Wood, and D. Chokola. The Evolution of PA Linearization: From Classic Feedforward and Feedback Through Analog and Digital Predistortion. *IEEE Microwave Magazine*, 17(2):32–40, feb 2016. doi: 10.1109/MMM.2015.2498079.

[12] T. Kobal and A. Zhu. Digital Predistortion of RF Power Amplifiers With Decomposed Vector Rotation-Based Recurrent Neural Networks. *IEEE Transactions on Microwave Theory and Techniques*, 70(11):4900–4909, nov 2022. doi: 10.1109/TMTT.2022.3209658.

[13] T. Kobal, Y. Li, X. Wang, and A. Zhu. Digital Predistortion of RF Power Amplifiers With Phase-Gated Recurrent Neural Networks. *IEEE Trans. Microw. Theory Techn.*, 70(6):3291–3302, Jun 2022.

[14] H. Li, Y. Zhang, G. Li, and F. Liu. Vector Decomposed Long Short-Term Memory Model for Behavioral Modeling and Digital Predistortion for Wideband RF Power Amplifiers. *IEEE Access*, 8:63780–63789, 2020. doi: 10.1109/ACCESS.2020.2984682.

[15] Taijun Liu, S. Boumaiza, and F. M. Ghannouchi. Application of Neural Networks to 3G Power Amplifier Modeling. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pages 2378–2382, 2005. doi: 10.1109/IJCNN.2005.1556274.

[16] D. Neil, J. H. Lee, T. Delbruck, and S.-C. Liu. Delta Networks for Optimized Recurrent Network Computation. In *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, 2017.

[17] A. A. M. Saleh and J. Salz. Adaptive Linearization of Power Amplifiers in Digital Radio Systems. *The Bell System Technical Journal*, 62(4):1019–1033, apr 1983. doi: 10.1002/j.1538-7305.1983.tb03113.x.

[18] J. van der Westhuizen and J. Lasenby. The Unreasonable Effectiveness of the Forget Gate. *arXiv preprint arXiv:1804.04849*, 2018.

[19] 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. In *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Singapore, 2024.

[20] Yizhuo Wu, Ang Li, Mohammadreza Beikmirza, Gagan Deep Singh, Qinyu Chen, Leo C. N. de Vreede, Morteza Alavi, and Chang Gao. MP-DPD: Low-complexity mixed-precision neural networks for energy-efficient digital predistortion of wideband power amplifiers. *IEEE Microwave and Wireless Technology Letters*, 2024. doi: 10.1109/LMWT.2024.3386330.

[21] Q. Zhang, C. Jiang, G. Yang, R. Han, and F. Liu. Block-Oriented Recurrent Neural Network for Digital Predistortion of RF Power Amplifiers. *IEEE Transactions on Microwave Theory and Techniques*, 72(7):3875–3885, jul 2024. doi: 10.1109/TMTT.2023.3337939.

Appendix A

Source Code

A.1 DeltaJANET Model Implementation

```
import torch
from torch import Tensor
import torch.nn as nn
import numpy as np

class DeltaJANET(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers,
                 thx=0, thh=0, bias=True, beta=-2.5):
        super(DeltaJANET, self).__init__()
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.output_size = output_size
        self.num_layers = num_layers
        self.thh = thh
        self.thx = thx
        self.bias = bias
        self.beta = beta

        # Instantiate NN Layers
        self.rnn =  DeltaJANETLayer(input_size=input_size,
                        hidden_size=hidden_size,
                        num_layers=num_layers,
                        beta=self.beta,
                        thx=self.thx,
                        thh=self.thh)

        self.fc_out = nn.Linear(in_features=hidden_size,
                                out_features=self.output_size,
                                bias=True)
        self.set_debug(1)

    def reset_parameters(self):
```

```
        self.rnn.reset_parameters()

        for name, param in self.fc_out.named_parameters():
            if 'weight' in name:
                nn.init.xavier_uniform_(param)
            if 'bias' in name:
                nn.init.constant_(param, 0)

    def forward(self, x, h_0):
        i_x = torch.unsqueeze(x[..., 0], dim=-1)
        q_x = torch.unsqueeze(x[..., 1], dim=-1)
        amp2 = torch.pow(i_x, 2) + torch.pow(q_x, 2)
        amp = torch.sqrt(amp2)
        amp3 = torch.pow(amp, 3)
        cos = i_x / amp
        sin = q_x / amp
        x = torch.cat((i_x, q_x, amp, amp3, sin, cos), dim=-1)
        h_0 = None
        out = self.rnn(x, h_0)
        out = self.fc_out(out)
        return out

    def set_debug(self, value):
        setattr(self, "debug", value)
        self.rnn.statistics = {
            "num_dx_zeros": 0,
            "num_dx_numel": 0,
            "num_dh_zeros": 0,
            "num_dh_numel": 0
        }

    def get_temporal_sparsity(self):
        temporal_sparsity = {}
        if self.rnn.debug:
            # Get RNN layer sparsity
            num_dx_zeros = self.rnn.statistics["num_dx_zeros"]
            num_dx_numel = self.rnn.statistics["num_dx_numel"]
            num_dh_zeros = self.rnn.statistics["num_dh_zeros"]
            num_dh_numel = self.rnn.statistics["num_dh_numel"]

            # Add FC layers neurons to total count
            rnn_numel = sum(p.numel() for name, p in
                self.rnn.named_parameters() if 'weight' in name)
            fc_numel = self.fc_out.weight.numel()
            total_numel = num_dx_numel + num_dh_numel + fc_numel
            total_zeros = num_dx_zeros + num_dh_zeros + fc_numel  # FC
                layers are dense, count as zeros
```

```
            temporal_sparsity["SP_T_DX"] = float(num_dx_zeros /
                num_dx_numel)
            temporal_sparsity["SP_T_DH"] = float(num_dh_zeros /
                num_dh_numel)
            temporal_sparsity["SP_T_DV"] = float(total_zeros /
                total_numel)
            temporal_sparsity["HW_PARAM"] = float(fc_numel + rnn_numel *
                float(total_zeros / total_numel))

        return temporal_sparsity


class DeltaJANETLayer(nn.Module):
    def __init__(self,
                 input_size=6,
                 hidden_size=256,
                 num_layers=1,
                 beta=-2.0,
                 thx=0.1,
                 thh=0):
        super(DeltaJANETLayer, self).__init__()

        # Hyperparameters
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.beta = beta
        self.th_x = thx
        self.th_h = thh
        self.weight_ih_height = 2 * self.hidden_size
        self.weight_ih_width = self.input_size
        self.weight_hh_width = self.hidden_size
        self.weight_hh_height = 2 * self.hidden_size
        self.x_p_length = max(self.input_size, self.hidden_size)
        self.batch_first = True
        self.debug = 1
        # Statistics
        self.abs_sum_delta_hid = torch.zeros(1)
        self.sp_dx = 0
        self.sp_dh = 0

        self.set_debug(self.debug)

        # Define the weights and biases as Parameters for each layer
        for layer in range(num_layers):
            # Input to hidden weights (input_size -> 2*hidden_size)
```

```
            weight_ih = nn.Parameter(torch.empty(2 * hidden_size,
                input_size if layer == 0 else hidden_size))
            setattr(self, f'weight_ih_l{layer}', weight_ih)

            # Hidden to hidden weights (hidden_size -> 2*hidden_size)
            weight_hh = nn.Parameter(torch.empty(2 * hidden_size,
                hidden_size))
            setattr(self, f'weight_hh_l{layer}', weight_hh)

            # Biases
            bias_ih = nn.Parameter(torch.empty(2 * hidden_size))
            setattr(self, f'bias_ih_l{layer}', bias_ih)

            bias_hh = nn.Parameter(torch.empty(2 * hidden_size))
            setattr(self, f'bias_hh_l{layer}', bias_hh)

        # Initialize the parameters
        self.reset_parameters()

    def set_debug(self, value):
        setattr(self, "debug", value)
        self.statistics = {
            "num_dx_zeros": 0,
            "num_dx_numel": 0,
            "num_dh_zeros": 0,
            "num_dh_numel": 0
        }

    def add_to_debug(self, x, i_layer, name):
        if self.debug:
            if isinstance(x, Tensor):
                variable = np.squeeze(x.cpu().numpy())
            else:
                variable = np.squeeze(np.asarray(x))
            variable_name = '_'.join(['l' + str(i_layer), name])
            if variable_name not in self.statistics.keys():
                self.statistics[variable_name] = []
            self.statistics[variable_name].append(variable)

    def reset_parameters(self):
        for name, param in self.named_parameters():
            num_gates = int(param.shape[0] / self.hidden_size)
            if 'bias' in name:
                nn.init.zeros_(param)
            if 'weight' in name:
                for i in range(0, num_gates):
```

```
                nn.init.orthogonal_(param[i * self.hidden_size:(i +
                    1) * self.hidden_size, :])
        if 'weight_ih_l0' in name:
            for i in range(0, num_gates):
              nn.init.xavier_uniform_(param[i * self.hidden_size:(i
                    + 1) * self.hidden_size, :])

def get_temporal_sparsity(self):
    temporal_sparsity = {}
    if self.debug:
        temporal_sparsity["SP_T_DX"] =
            float(self.statistics["num_dx_zeros"] /
            self.statistics["num_dx_numel"])
        temporal_sparsity["SP_T_DH"] =
            float(self.statistics["num_dh_zeros"] /
            self.statistics["num_dh_numel"])
        temporal_sparsity["SP_T_DV"] =
            float((self.statistics["num_dx_zeros"] +
            self.statistics["num_dh_zeros"]) /
                                (self.statistics["num_dx_numel"] +
                                                self.statistics["num_dh_numel"]))
    self.statistics.update(temporal_sparsity)
    return temporal_sparsity

def process_inputs_first(self, x: Tensor, x_p_0: Tensor = None,
    h_0: Tensor = None, h_p_0: Tensor = None,
                    dm_0: Tensor = None):
    if not self.batch_first:
        x = x.transpose(0, 1)
    batch_size = x.size(0)

   if x_p_0 is None or h_0 is None or h_p_0 is None or dm_0 is None:
        x_p_0 = torch.zeros(self.num_layers, batch_size,
            self.x_p_length, dtype=x.dtype, device=x.device)
        h_0 = torch.zeros(self.num_layers, batch_size,
            self.hidden_size, dtype=x.dtype, device=x.device)
        h_p_0 = torch.zeros(self.num_layers, batch_size,
            self.hidden_size, dtype=x.dtype, device=x.device)
        dm_0 = torch.zeros(self.num_layers, batch_size,
            self.weight_ih_height, dtype=x.dtype, device=x.device)
        for l in range(self.num_layers):
            bias_ih = getattr(self, 'bias_ih_l{}'.format(l))
            bias_hh = getattr(self, 'bias_hh_l{}'.format(l))
            dm_0[l, :, :self.hidden_size] +=
                bias_ih[:self.hidden_size] +
                bias_hh[:self.hidden_size]
```

```
                dm_0[l, :, self.hidden_size:2 * self.hidden_size] +=
                    bias_ih[self.hidden_size:2 * self.hidden_size] +
                    bias_hh[self.hidden_size:2 * self.hidden_size]

        return x, x_p_0, h_0, h_p_0, dm_0

    @staticmethod
    def compute_deltas(x: Tensor, x_p: Tensor, h: Tensor, h_p: Tensor,
        th_x: Tensor, th_h: Tensor):
        delta_x = x - x_p
        delta_h = h - h_p

        delta_x_abs = torch.abs(delta_x)
        delta_x = delta_x.masked_fill(delta_x_abs < th_x, 0)

        delta_h_abs = torch.abs(delta_h)
        delta_h = delta_h.masked_fill(delta_h_abs < th_h, 0)

        return delta_x, delta_h, delta_x_abs, delta_h_abs

    @staticmethod
    def update_states(delta_x_abs, delta_h_abs, x, h, x_p, h_p,
        x_prev_out, th_x, th_h):
        x_p = torch.where(delta_x_abs >= th_x, x, x_p)
        x_prev_out[:, :x.size(-1)] = x_p
        h_p = torch.where(delta_h_abs >= th_h, h, h_p)
        return x_p, h_p, x_prev_out

    @staticmethod
    def compute_gates(delta_x: Tensor, delta_h: Tensor, dm: Tensor,
        weight_ih: Tensor,
                      weight_hh: Tensor):
        mac_x = torch.mm(delta_x, weight_ih.t())
        mac_h = torch.mm(delta_h, weight_hh.t())
        new_dm = mac_x + mac_h + dm
        dm_f, dm_g = new_dm.chunk(2, dim=1)
        return dm_f, dm_g, new_dm

    def layer_forward(self, input: Tensor, l: int, x_p_0: Tensor =
        None, h_0: Tensor = None,
                      h_p_0: Tensor = None, dm_0: Tensor = None):
        weight_ih = getattr(self, 'weight_ih_l{}'.format(l))
        weight_hh = getattr(self, 'weight_hh_l{}'.format(l))
        batch_size, seq_len, input_size = input.size()

        th_x = torch.tensor(self.th_x, dtype=input.dtype)
        th_h = torch.tensor(self.th_h, dtype=input.dtype)
```

```
output = []

reg = torch.zeros(1, dtype=input.dtype,
    device=input.device).squeeze()

x_p_out = torch.zeros(batch_size, self.x_p_length,
    dtype=input.dtype, device=input.device)
x_p = x_p_0[:, :input_size]
x_prev_out = torch.zeros_like(x_p)
h = h_0
h_p = h_p_0
dm = dm_0
l1_norm_delta_h = torch.zeros(1, dtype=input.dtype,
    device=input.device)

for t in range(seq_len):
    x = input[:, t, :]

    delta_x, delta_h, delta_x_abs, delta_h_abs =
        self.compute_deltas(x, x_p, h, h_p, th_x, th_h)
    reg += torch.sum(torch.abs(delta_h))

    if self.debug:
        zero_mask_delta_x = torch.as_tensor(delta_x == 0,
            dtype=x.dtype)
        zero_mask_delta_h = torch.as_tensor(delta_h == 0,
            dtype=x.dtype)
        self.statistics["num_dx_zeros"] +=
            torch.sum(zero_mask_delta_x)
        self.statistics["num_dh_zeros"] +=
            torch.sum(zero_mask_delta_h)
        self.statistics["num_dx_numel"] += torch.numel(delta_x)
        self.statistics["num_dh_numel"] += torch.numel(delta_h)

    x_p, h_p, x_prev_out = self.update_states(delta_x_abs,
        delta_h_abs, x, h, x_p, h_p, x_prev_out, th_x, th_h)

    l1_norm_delta_h += torch.sum(torch.abs(delta_h))

    s, cat, dm = self.compute_gates(delta_x, delta_h, dm,
        weight_ih, weight_hh)

    c_hat = torch.tanh(cat)

    h = torch.sigmoid(s) * h + (1 - torch.sigmoid(s -
        self.beta)) * c_hat
```

```
            output += [h]

        output = torch.stack(output, dim=1)
        x_p_out[:, :input_size] = x_p
        return output

    def forward(self, input: Tensor, x_p_0: Tensor = None, h_0: Tensor =
        None, h_p_0: Tensor = None,
                dm_0: Tensor = None):
        x, x_p_0, h_0, h_p_0, dm_0 = self.process_inputs_first(input,
            x_p_0, h_0, h_p_0, dm_0)

        for l in range(self.num_layers):
            x = self.layer_forward(x, l, x_p_0[l], h_0[l],  h_p_0[l],
                dm_0[l])

        return x
```

A.2 TC-JANET Model Implementation

```
import torch
from torch import Tensor
import torch.nn as nn

EPSILON = 1e-8

class TCJANET(nn.Module):
    def __init__(self, hidden_size, output_size,
                 tcn_hidden_channels=9, tcn_num_layers=4,
                 batch_first=True, bias=True, beta=-2.1, memory_depth=2):
        super(TCJANET, self).__init__()
        self.input_size = 4
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.tcn_hidden_channels = tcn_hidden_channels
        self.tcn_num_layers = tcn_num_layers
        self.batch_first = batch_first
        self.bias = bias
        self.beta = beta
        self.memory_depth = memory_depth

        # JANETLayer with multiple layers integrated inside
        self.network = TCJANETLayer(input_size=self.input_size,
                            hidden_size=self.hidden_size,
                            tcn_hidden_channels=self.tcn_hidden_channels,
                            tcn_num_layers=self.tcn_num_layers,
```

```
                            batch_first=self.batch_first,
                            bias=self.bias,
                            beta=self.beta,
                            memory_depth=self.memory_depth)

        # Fully connected output layer
        self.fc_out = nn.Linear(in_features=self.hidden_size,
                                out_features=self.output_size,
                                bias=True)


    def reset_parameters(self):
        for name, param in self.fc_out.named_parameters():
            if 'weight' in name:
                nn.init.xavier_uniform_(param)
            if 'bias' in name:
                nn.init.constant_(param, 0)

    def forward(self, x, h_0=None):
        if not self.batch_first:
            x = x.transpose(0, 1)
        batch_size, seq_len, _ = x.shape

        h = h_0.squeeze(0) if h_0 is not None else torch.zeros(batch_size,
            self.hidden_size, device=x.device)

        # Network forward
        out = self.network(x, h)
        out = self.fc_out(out)

        return out

class TCJANETLayer(nn.Module):
    def __init__(self, input_size, hidden_size, tcn_hidden_channels,
        tcn_num_layers,
                 batch_first=True, bias=True, beta=-2.1, memory_depth=2):
        super(TCJANETLayer, self).__init__()
        # JANET parameters
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.janet_num_layers = 1
        self.batch_first = batch_first
        self.bias = bias
        self.beta = beta

        # TCN parameters
        self.dilation_base = 2
```

```
        self.kernel_size = 5
        self.tcn_hidden_channels = tcn_hidden_channels
        self.tcn_num_layers = tcn_num_layers
        self.tcn_activation = 'SiLU'

        # memory depth for context window
        self.memory_depth = memory_depth
        self.tcn_input_size = self.input_size * (self.memory_depth + 1)

        self.tcn = TCN(in_channels=self.tcn_input_size,
                       hidden_channels=self.tcn_hidden_channels,
                       output_size=2 * self.hidden_size,
                       num_layers=self.tcn_num_layers,
                       kernel_size=self.kernel_size,
                       stride=1,
                       dilation_base=self.dilation_base,
                       activation=self.tcn_activation)

        self.bias_ih = nn.Parameter(torch.empty(2 * hidden_size))
        self.bias_hh = nn.Parameter(torch.empty(2 * hidden_size))

        self.reset_parameters()

    def reset_parameters(self):
        self.tcn.reset_parameters()
        for name, param in self.named_parameters():
            if 'bias' in name:
                nn.init.zeros_(param)

    def forward(self, x: Tensor, h_0: Tensor) -> Tensor:
        batch_size, seq_len, _ = x.shape

        x_complex = torch.view_as_complex(x.contiguous())
        amp = x_complex.abs().unsqueeze(-1)
        amp3 = torch.pow(amp, 3)
        x_features = torch.cat([x, amp, amp3], dim=-1)

        # memory context window
        if self.memory_depth > 0:
            padding = x_features[:, 0, :].unsqueeze(1).repeat(1,
                self.memory_depth, 1)
            x_features_padded = torch.cat([padding, x_features], dim=1)

            x_features_unfolded = x_features_padded.unfold(dimension=1,
                                                           size=self.memory_depth + 1,
                                                                  step=1)
```

```
        x_features = x_features_unfolded.permute(0, 1, 3,
            2).flatten(start_dim=2)

    # Split gates for JANET update
    f, c = self.tcn(x_features).chunk(2, dim=2)

    # Add bias terms
    f = f + self.bias_ih[:self.hidden_size] +
        self.bias_hh[:self.hidden_size]
    c = c + self.bias_ih[self.hidden_size:] +
        self.bias_hh[self.hidden_size:]

    c = torch.tanh(c)
    s = torch.sigmoid(f)
    s_mod = torch.sigmoid(f - self.beta)

    # rotate factor
    r = x_complex.conj() / (x_complex.abs() + EPSILON)

    h = h_0

    outputs = []
    for t in range(seq_len):
        r_t = r[:, t].unsqueeze(-1)

        # gate
        s_t = s[:, t, :]
        s_mod_t = s_mod[:, t, :]
        c_t = c[:, t, :]

        # normalize hidden state
        h_complex = torch.view_as_complex(h.view(batch_size,
            self.hidden_size // 2, 2))
        c_complex = torch.view_as_complex(c_t.view(batch_size,
            self.hidden_size // 2, 2).contiguous())

        h_complex_norm = h_complex * r_t
        c_complex_norm = c_complex * r_t

        h_real_norm =
            torch.view_as_real(h_complex_norm).flatten(start_dim=1)
        c_real_norm =
            torch.view_as_real(c_complex_norm).flatten(start_dim=1)

        # JANET update
        h_updated = s_t * h_real_norm + (1 - s_mod_t) * c_real_norm
```

```
            # denormalize hidden state
            h_complex_norm = torch.view_as_complex(h_updated.view(batch_size,
                self.hidden_size // 2, 2))
            h_complex_denorm = h_complex_norm * r_t.conj()
            h = torch.view_as_real(h_complex_denorm).flatten(start_dim=1)

            outputs.append(h)

        out = torch.stack(outputs, dim=1)

        return out


ACTIVATION_FUNCS = ['CELU', 'ELU', 'GELU', 'Hardshrink', 'Hardtanh',
    'Hardswish', 'LeakyReLU', 'LogSigmoid',
                'Mish', 'ReLU', 'ReLU6', 'RReLU', 'SELU', 'SiLU', 'Softplus',
                    'Softshrink', 'Softsign',
                'Tanh', 'Tanhshrink', 'Hardsigmoid', 'Sigmoid', 'PReLU',
                    'Threshold', 'GLU']

class TCN(nn.Module):
    def __init__(self, in_channels, hidden_channels, output_size, num_layers,
                 kernel_size=5, stride=1, dilation_base=2, activation='SiLU'):
        super(TCN, self).__init__()
        self.bias = False
        self.in_channels = in_channels
        self.hidden_channels = hidden_channels
        self.output_size = output_size
        self.num_layers = num_layers
        self.kernel_size = kernel_size
        self.padding = (self.kernel_size-1) // 2
        self.stride = stride
        self.dilation_base = dilation_base
        self.activation_name = activation

        # Create activation function based on the activation parameter
        self.activation_fn = self._get_activation_fn(activation)

        layers = []
        # Initial pointwise projection layer
        layers.append(nn.Conv1d(in_channels=self.in_channels,
            out_channels=self.hidden_channels, kernel_size=1))
        layers.append(self.activation_fn)

        # Add dilated convolution layers based on num_layers
        for i in range(num_layers):
            dilation = self.dilation_base ** i
```

```
            layers.append(nn.Conv1d(self.hidden_channels, self.hidden_channels,
                self.kernel_size,
                                    stride=self.stride, padding=self.padding *
                                        dilation,
                                dilation=dilation, groups=self.hidden_channels,
                                        bias=self.bias))

            layers.append(self.activation_fn)


        # Output projection
        layers.append(nn.Conv1d(self.hidden_channels, self.output_size,
            kernel_size=1, bias=self.bias))

        self.network = nn.Sequential(*layers)

        self.res_proj = nn.Conv1d(self.in_channels, self.output_size, 1)


    def _get_activation_fn(self, activation):
        """Helper method to create the activation function based on name"""
        return {
            'Threshold': lambda: nn.Threshold(0.5, 0.0),
            **{name: getattr(nn, name) for name in ACTIVATION_FUNCS}
        }.get(activation, nn.Hardswish)()


    def reset_parameters(self):
        for m in self.modules():
            if isinstance(m, nn.Conv1d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
                    nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)

    def feature_extraction(self, x):
        i_x = torch.unsqueeze(x[..., 0], dim=-1)
        q_x = torch.unsqueeze(x[..., 1], dim=-1)
        amp2 = torch.pow(i_x, 2) + torch.pow(q_x, 2)
        amp = torch.sqrt(amp2)
        amp3 = torch.pow(amp, 3)
        cos = i_x / amp
        sin = q_x / amp
        return i_x, q_x, amp, amp3, sin, cos

    def forward(self, x, h_0=None):
        input = self.res_proj(x.transpose(1, 2)).transpose(1, 2)
```

```
out = self.network(x.transpose(1, 2)).transpose(1, 2)
return out + input
```