Computational Complexity Optimization of Neural Network-Based Equalizers in Digital Signal Processing: A Comprehensive Approach

Pedro Freire, Sasipim Srivallapanondh, Bernhard Spinnler, Antonio Napoli, Nelson Costa, Jaroslaw E. Prilepsky, and Sergei K. Turitsyn

(Invited Paper)

***Abstract*—Experimental results based on offline processing reported at optical conferences increasingly rely on neural network-based equalizers for accurate data recovery. However, achieving low-complexity implementations that are efficient for real-time digital signal processing remains a challenge. This paper addresses this critical need by proposing a systematic approach to designing and evaluating low-complexity neural network equalizers. Our approach focuses on three key phases: training, inference, and hardware synthesis. We provide a comprehensive review of existing methods for reducing complexity in each phase, enabling informed choices during design. For the training and inference phases, we introduce a novel methodology for quantifying complexity. This includes new metrics that bridge software-to-hardware considerations, revealing the relationship between complexity and specific neural network architectures and hyperparameters. We guide the calculation of these metrics for both feed-forward and recurrent layers, highlighting the appropriate choice depending on the application's focus (software or hardware). Finally, to demonstrate the practical benefits of our approach, we showcase how the computational complexity of neural network equalizers can be significantly reduced and measured for both teacher (biLSTM+CNN) and student (1D-CNN) architectures in different scenarios. This work aims to standardize the estimation and optimization of computational complexity for neural networks applied to real-time digital signal processing, paving the way for more efficient and deployable optical communication systems.**

***Index Terms*—Neural networks, nonlinear equalizer, computational complexity, hardware estimation, signal processing.**

Manuscript received 29 December 2023; revised 11 March 2024; accepted 5 April 2024. Date of publication 10 April 2024; date of current version 27 June 2024. The work of Bernhard Spinnler, Antonio Napoli, and Nelson Costa was supported by the European Union' Horizon Europe Research and Innovation Programme, under Grant 101092766 (ALLEGRO). The work of Jaroslaw E. Prilepsky was supported by Leverhulme Trust, under Grant RP-2018-063. The work of Sergei K. Turitsyn was supported by the EPSRC Project TRANSNET. This work was supported by the EU Horizon 2020 Program under the Marie Sklodowska-Curie Grant Agreement 813144 (REAL-NET) and Grant 956713 (MENTOR). *(Corresponding author: Pedro Friere.)*

Pedro Freire, Sasipim Srivallapanondh, Jaroslaw E. Prilepsky, and Sergei K. Turitsyn are with the Aston Institute of Photonic Technologies, Aston University, B4 7ET Birmingham, U.K. (e-mail: p.freiredecarvalhosourza@aston.ac.uk).

Bernhard Spinnler and Antonio Napoli are with Infinera R&D, 81541 Munich, Germany (e-mail: anapoli@infinera.com).

Nelson Costa is with Infinera Unipessoal, 2790-078 Carnaxide, Portugal (e-mail: ncosta@infinera.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JLT.2024.3386886.

Digital Object Identifier 10.1109/JLT.2024.3386886

I. Introduction

OVER the last few decades, neural networks (NNs) have begun to find widespread usage in a wide range of signal processing applications: filtering, parameter estimation, signal detection, system identification, pattern recognition, signal reconstruction, time series analysis, signal compression, signal transmission, etc. [1], [2], [3], [4]. Audio, video, image, communication, geophysical, and radar scanning data, are examples of important signal types that typically undergo various forms of signal processing [5], [6], [7]. The key capabilities of NNs in signal processing are: performing distributed processing, emulating nonlinear transformations and processes, self-organizing, and enabling high-speed processing communication applications [8], [9], [10]. With these properties, NNs can provide a very powerful means of solving many signal processing tasks, particularly in the areas related to nonlinear signal processing, real-time signal processing, adaptive signal processing, and blind signal processing [5], [11], [12], [13].

NN methods have also proven to be efficient in several applications in optical communications, particularly in channel equalization [14]. NN structures have demonstrated the potential to significantly enhance transmission quality in various scenarios [15], with computational complexity comparable or better than that of classical approaches [16], [17]. However, additional work is still required to reach product-level applications. One of the most pressing challenges is to demonstrate how such NN-based equalizers can be effectively implemented, taking into account both the training and inference phases. In order to evaluate the computational complexity, the metrics for an assessment should be appropriately addressed.

Real-time signal processing, as an example, is a field that enables technological breakthroughs by effectively incorporating signal processing in hardware: real-time and onboard signal processing are the keys to the evolution of phones and watches into smartphones/smartwatches. To the best of our knowledge, one of the first real-time applications of NNs was discussed in 1989 [18], and numerous works since then have deliberated the challenges of implementing such solutions in hardware exploiting the notion of computational complexity [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. Similarly, in the NN-based equalizer, computational complexity analysis is necessary.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

From a computer science perspective, computational complexity analysis is almost always attributed to the Big-O notation of the algorithm [29], [30], [31]. In general, the Big-O notation is used to express an algorithm's complexity while assessing its efficiency, which means that we are interested in how effectively the algorithm scales with the size of the dataset in terms of running time [32], [33], [34]. However, from the engineering standpoint, the Big-O is often an oversimplified measure that cannot be immediately translated into the hardware resources required to realize the algorithm (NNs) in a hardware platform [20].

Due to this problem that refers to the absence of some "universal" measure, various works started to present complexity in terms of multiply and accumulate (MAC) [19], [20], [21], [22], Kolmogorov complexity [23], the number of bit-operations (BOP) [24], [25], the number of real multiplications (RM) [26], [27], [28]. However, it is not always clear when to use each specific metric, and, more importantly, none of the metrics mentioned above show the benefits of using different strategies of quantization to reduce the complexity of implementing the multipliers.

As far as we know, no work has so far unified the computational metrics itemized above such that we have no universal metrics to compare the complexity when different types of quantization are applied to NN structures. In this paper, we solve this issue by carrying out a systematic computational complexity analysis for a zoo of NN layer types. In addition, we introduce a new useful metric: we coined 'the number of additions and bit shifts' (NABS). This metric takes into account the impact of the weights' quantization type on the reduction of the multipliers' implementation complexity used in an NN layer. Overall, we intend our work to give largely universal measures of complexity to establish a comparison baseline depending on whether the application is software- or hardware-based.

The paper is organized as follows. Firstly, Section II provides an overview of the main strategies enabling low computational complexity NN-based equalizers from training to hardware synthesis, while still maintaining attractive Q-factor gains. In Section III we describe the details of different computational metrics for the training and inference stages. Section IV outlines a method for computing the computational complexity of diverse neural network layers based on their hyperparameters, traversing from the software to the hardware level. The computational complexity growth against the design parameters of each neural network layer is discussed in Section V. Additionally, in Section VI, we explore the impact of quantization on different computational complexity metrics and present a practical study in the realm of channel equalization. This study exemplifies complexity reduction approaches and elucidates the performance and complexity trade-offs associated with compensating nonlinearities in optical coherent transmissions. Our findings are summarized in the conclusion.

II. Overview on Complexity Reduction in Neural Network Equalizers

To achieve optimal computational complexity in the pursuit of efficient NN equalizers for resource-constrained hardware, it is necessary to thoroughly investigate three crucial phases: training, inference, and hardware synthesis phases. Fig. 1 illustrates the most common techniques applied to reduce the complexity of NN-based equalizers in each of these phases.

A. Complexity Reduction in Training

First, during training, reducing complexity is crucial for the efficient and practical deployment of hardware with limited resources. Various complexity reduction techniques are employed to enhance efficiency and performance. Techniques such as transfer learning or approaches to improve generalization, such as data augmentation, domain randomization, and semi-supervised learning, can be applied. These approaches indirectly reduce the need for large amounts of original training data. Effective generalization reduces the need for complex models with a high number of parameters, resulting in faster and more efficient training.

Data pre-processing is a crucial step in preparing the input for the NN training. This can be a main factor in enhancing the model efficiency and accelerating convergence because high-quality input data contributes to stable training, improves generalizability, and leads to successful data interpretation by the model [35]. Data pre-processing involves data normalization to have the features on a consistent scale, or feature engineering, which selects and transforms the input features to emphasize the relevant information and get rid of the noisy, irrelevant data.

Transfer learning (TL) adapts the knowledge acquired in the source tasks to the related target tasks. This technique can considerably reduce the training time and resources required. TL is particularly useful when training the models on limited computational resources, as it allows the NN to inherit knowledge from a larger pre-trained model and fine-tune it for a specific task. TL was investigated in equalization tasks in both directly [36] and coherently [37] detected systems. Ref. [37] demonstrated the potential of TL to reduce the number of training epochs and the training dataset without impacting the equalizer's performance.

Domain randomization is a systematic approach for data generation aiming to improve the generalization and the robustness of machine learning models in new environments [38]. Domain randomization generates training data from a random distribution with given desired properties and stores it in a library accessible by NN. By using synthetic data, this approach reduces the dependence on real-world data, thus improving the training efficiency [39]. This technique is especially valuable when dealing with complex and dynamic environments as the model becomes more adaptable to variations encountered during deployment.

Semi-supervised learning combines labeled and unlabeled data in training, allowing the model to learn from both. Semi-supervised learning enables NN to leverage the available labeled data more effectively by incorporating information from the unlabeled samples. This method enhances the model's performance without requiring additional labeled data, which makes the model more flexible to transmission changes. This method resembles decision-directed adaptive equalization [40] for channel equalization.

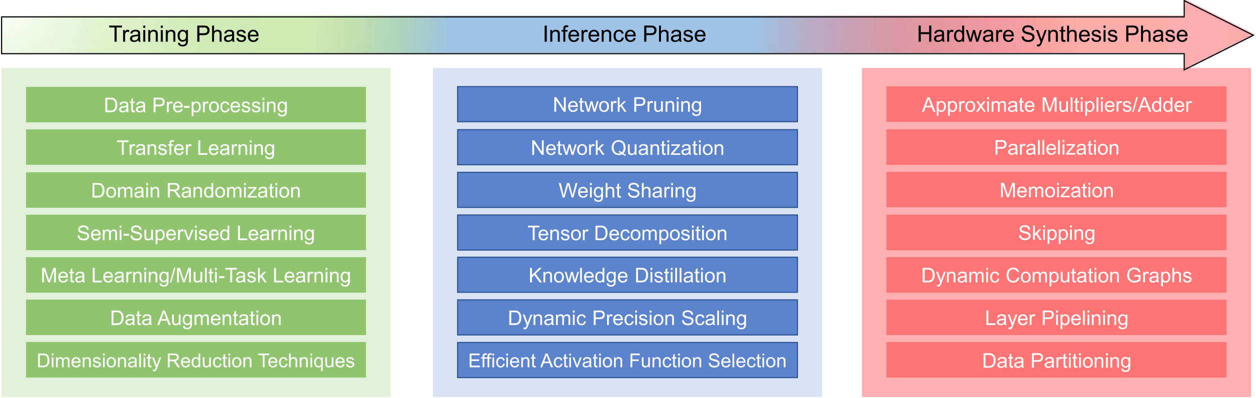


Fig. 1. Main strategies to design low complex NN-equalizers in training, inference, and hardware synthesis phases.

Meta learning involves training models that can efficiently adapt to new tasks with only a small amount of training data based on experiences gained from a variety of learning tasks [41]. This approach explicitly trains the model parameters to enable efficient generalization on new tasks with a small number of gradient steps and minimal training data, making it simple to fine-tune.

Multi-Task Learning is a paradigm where a single model is trained to perform multiple but related tasks simultaneously. In contrast, traditional single-task learning trains multiple separate models for each task independently. Multi-task learning leverages shared representations across tasks and the model's parameters are optimized jointly across all tasks. This training approach can lead to better generalization of the model and reduce the need to deploy several models for different tasks and does not require re-training when performing related tasks. This technique not only reduces the number of models that need to be trained but also reduces the complexity in the inference phase. However, this technique can exhibit a trade-off between overall performance and specific task performance. This approach has been shown to be efficient in the NN-based equalizers in both IM/DD [42] and coherent systems [43].

Data augmentation allows datasets to be more diverse and representative by artificially generating additional data points from existing data [44]. Data augmentation used in optical NN-based equalizers is a technique to improve equalization performance and decrease the training complexity of supervised learning in nonlinearity mitigation. In supervised learning tasks, normally a large training dataset is required. The model will also need to be re-trained when the channel conditions change. However, Big Data collection can be challenging. The efficient use of a limited dataset is more desirable for practical implementation. Ref. [45] showed that data augmentation reduces the size of the dataset up to 6 times while maintaining the optical performance. This technique enables a less overfitting model, fewer model parameter requirements, and a faster convergence of the training.

Dimensionality reduction techniques address the challenges associated with high-dimensional input spaces. These techniques aim to capture and retain the most informative aspects of the data while reducing the number of input features significantly [46]. Principal Component Analysis (PCA) [47], for instance, transforms the original features into a lower-dimensional space defined by principal components (a new set of uncorrelated variables), retaining the maximum variance.

B. Complexity Reduction in Inference

Next, during inference, the NN must accurately equalize the input signal using the minimum computational resources while meeting the required performance metrics. As in the real world, the environments and the resources are more constrained. This result can be achieved by using different techniques, for example, network pruning, sparse representation, knowledge distillation (KD), and tensor decomposition.

Network pruning reduces the complexity of NNs by removing redundant or less significant parameters (weights, connections, neurons, or layers) from a trained NN. Pruning can be carried out without significantly affecting equalization performance, as described in [15], [48], [49]. By eliminating unnecessary parameters, this approach reduces the model's size and computational requirements during inference. Sparse connectivity, achieved through pruning, enables faster execution of the NN on hardware. In most cases, pruning in optical channel equalization has been restricted to the feedforward NN, however, Ref. [15] extended the investigation to the case of recurrent equalizer in coherent optical transmission.

Network quantization reduces the precision of the weights and activation functions in NN, for instance, 32-bit floating-point values are converted to 8-bit integers. However, the trade-off between complexity and performance should also be carefully considered, because there might be a performance sacrifice when precision is drastically reduced. This approach has the potential to reduce complexity and memory usage, aiming to make the model more efficient for deployment on resource-constrained device [15], [50].

Weight Sharing compresses the NN weights and biases by keeping only a small number of non-zero coefficients. The redundant or similar values of weights in the NN are collapsed into a single shared weight [15]. The reduction in the number

of unique values of weights decreases the number of parameters that need to be computed and stored, leading to a more compact and memory-efficient model.

Tensor decomposition decomposes high-dimensional data into a lower-dimensional space [51]. In other words, a multidimensional tensor is broken down into a combination of simpler tensors. By decomposing tensors, especially weight tensors in NN, into smaller and more manageable components, tensor decomposition reduces the number of parameters and computations needed in the inference phase. In [52], the authors showed that the sparse decomposition of the tensor in convolutional filters can successfully reduce model complexity and memory usage during inference.

Knowledge Distillation (KD) is applied to transfer knowledge from a larger model (teacher) to a more compact one (student) using teacher predictions to assist student learning. KD can reduce the size of the model [53]. The distilled model retains the essential information from the teacher model, making it suitable for deployment in resource-constrained environments during the inference process. Ref. [54] proved to use KD to accelerate the inference of the NN equalizer by recasting the RNN-based equalizer into a feed-forward-based equalizer without significantly compromising the equalization performance.

Dynamic precision scaling (DPS) adjusts the precision of the numerical values of the weights and during computation dynamically, based on the specific requirements of each computation. With this approach, the NN can utilize lower precision when the accuracy demands allow, as DPS optimizes the utilization of available resources. This approach provides an effective reduction of complexity during inference. Ref. [55] showed that DPS could be used in both the forward pass (inference) and backward pass for training.

Efficient Activation Function Selection can play an important role in complexity reduction. The expensive activation functions, e.g. hyperbolic tangent or sigmoid can be replaced by the approximated alternatives or with the Look-up Table to reduce the computation [56]. Simpler functions such as ReLU (Rectified Linear Unit) are also commonly chosen because of their simplicity in calculation and speed.

C. Complexity Reduction in Hardware Synthesis

Complexity reduction techniques of NN in hardware synthesis play a crucial role in optimizing the NN implementation on dedicated hardware. In hardware synthesis, the NN is mapped onto the hardware architecture, and the hardware design is optimized to achieve the desired performance while minimizing resource utilization. There are some techniques to reduce the complexity of the hardware, such as multiplier/adder approximations, parallelization, memoization, and skipping. The choice of suitable techniques depends on the NN architecture, the characteristics of the target hardware, and the desired trade-off between computational efficiency and model accuracy.

Multiplier/adder approximation is to reduce the hardware resource requirements by approximating multiplier and adder which are key components of the hardware implementation for NN computation [57]. The approximation replaces full-precision multipliers and adders with less resource-intensive multipliers and adder implementations, such as approximate adders or low-precision. Binary or ternary multipliers are examples of low-precision alternatives. By using lower-precision multipliers and adder implementations, the overall hardware complexity is reduced. This can lead to more efficient use of hardware resources without significantly sacrificing model accuracy.

Parallelization involves dividing the neural network into multiple sub-networks to be processed simultaneously, aiming for faster and more efficient execution in terms of latency and throughput [58]. This methodology capitalizes on parallel hardware architectures, such as GPUs, yielding heightened computational efficiency. As detailed in Ref. [59], Parallelization can take various forms including: **Data Parallelism**, where multiple NN instances operate simultaneously on distinct data batches; **Model Parallelism**, involving the division of a single neural network across multiple processors or GPUs, with different components processed on separate devices;[1] and **Pipeline Parallelism (Inter-Layer parallelism)**, which segments the neural network computation into stages, each executed by a distinct processing unit such that the data flow resembles a sequential assembly line.

Memoization stores and reuses intermediate results of expensive computations in memory to avoid recalculation when the same input reappears [60], [61]. This can be especially beneficial in RNN or other architectures with repetitive computations, leading to improved hardware efficiency. Even the simple implementation of memoization in Ref. [61] could speed up different experiments with different workloads ranging from 7% up to 25%.

Skipping can be used to decrease the executed workload and reduce computational costs. This method selectively skips certain computations based on their relevance to the final output or the predefined conditions. Skipping approximations can be performed by a simple calculation to evaluate if a more complex computation can be eliminated [60].

Dynamic computation graph allows the structure of the NN to change dynamically at runtime [62]. While the static graph fixes the structure of the NN (like the sequence of operations and connections) before the beginning of training, the dynamic graph constructs the structure of the NN on-the-fly during execution. This approach allows flexibility as it adapts the structure efficiently depending on varying input types and conditions.

Layer pipelining splits the processing of different layers in NN into sequential stages that overlap in time to allow parallelization of the computation. This method [63] allows scalable model parallelism with high hardware utilization and training stability. Pipelining algorithm library, GPipe, from [63] is a library to train a giant NN, with efficiency (speeds up the

[1]In addition to the aforementioned parallelization techniques, another subcategory worth mentioning is intra-layer parallelism, often referred to as Tensor Parallelism. This method entails parallelizing computations within a single layer of the neural network. Specifically, it involves partitioning large tensors, such as weight matrices, of a layer across multiple devices, facilitating parallel computations on these segmented chunks.

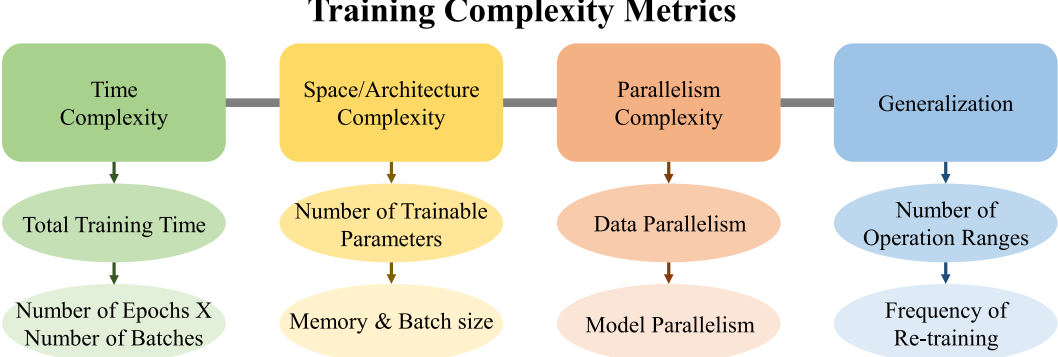


Fig. 2. Main metrics to evaluate complexity in the training phase for NN-equalizers.

process), flexibility (supports any deep network), and reliability (guarantees consistent training).

Data partitioning is an approach to divide the input data into subsets to be processed in parallel by different hardware components independently [64]. After that, the result of each partition is combined. Ref. [65] demonstrated that with their data partitioning approach, only small memory storage is required, instead of duplicating the whole data set size over all the processing units.

D. Insights for Low-Complexity NN Equalizers Implementation

For the training, we believe that one of the most important and promising techniques in reducing computational complexity and enhancing the generalization of NN-based equalizers is multi-task learning. As in real-world scenarios, the equalizers require reconfiguration and must be adjustable to compensate for the variation of impairments as the channel characteristics change [43]. By implementing multi-task learning, the model does not need to be re-trained if performing in the range of knowledge acquired in training, but the model might experience a trade-off between overall performance and specific task performance. This robust approach enables a more practical utilization of NN-based equalizers, aligning closely with real-world demands.

For the inference, we advocate for the utilization of weight clustering techniques to optimize model size and computational complexity. This approach enables us to reduce the NN model's footprint, approaching levels comparable to constrained hardware environments such as CDC complexity [15]. Weight clustering achieves this by consolidating the number of distinct multipliers in matrix multiplication operations to at least the number of clusters per input element. Additionally, it facilitates heterogeneous quantization by minimizing the number of bits required to represent the weights effectively. Such strategies are pivotal in hardware implementations, where resource constraints necessitate efficient utilization of computational resources.

In the domain of hardware synthesis, we investigate various strategies to optimize the implementation of neural network-based equalizers. This involves delving into partitioning schemes, which encompass different strategies for distributing data and computations across processing units. One such approach involves layer-wise partitioning, where each layer of the neural network is allocated to specific processing units. Alternatively, channel-wise partitioning allocates computations related to individual channels of input data to separate processing units. These partitioning schemes aim to exploit parallelism within the neural network, thereby enhancing hardware efficiency and performance.

Furthermore, we explore parallelization architectures tailored for the efficient execution of parallel processing tasks. Systolic arrays represent one such architecture that orchestrates computations through a pipeline of processing elements arranged in a grid-like fashion. Specialized neural network accelerators are also investigated, leveraging dedicated hardware components optimized for executing neural network operations in parallel. These architectures are designed to exploit inherent parallelism within neural network computations, leading to significant improvements in performance and throughput.

In addition to partitioning and parallelization strategies, we focus on memory access optimization techniques to minimize overhead during data retrieval and processing. This involves optimizing memory access patterns to minimize latency and maximize bandwidth utilization. Techniques such as data prefetching, caching, and memory banking are explored to streamline memory access operations and alleviate bottlenecks associated with partitioned data processing. By optimizing memory access, we aim to enhance overall system efficiency and throughput in hardware implementations of neural network-based equalizers.

III. Complexity Metrics (Training and Inference)

After implementing the mentioned complexity reduction strategies, it is essential to evaluate their effectiveness. This section gives a comprehensive understanding of the model's complexity during the training and real-time inference phases on the target hardware platform. Figs. 2 and 3 show the metrics used to measure the complexity of training and inference phases, respectively.

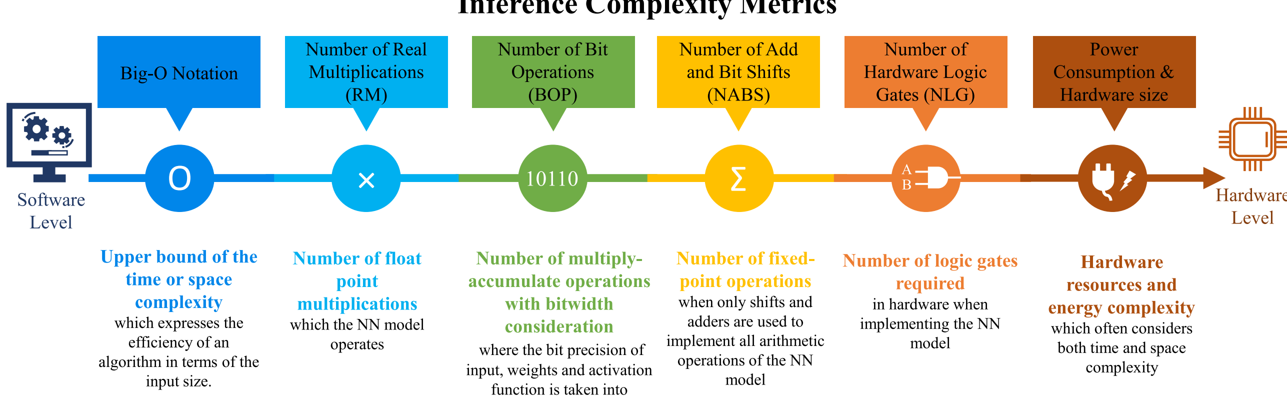


Fig. 3. Main metrics to evaluate complexity in the inference phase for NN-equalizers.

A. Complexity Metrics for Training

For training, the computational complexity of the NN should be appropriately evaluated, as it allows for efficient resource allocation and is useful for comparing different models to assess the efficiency and effectiveness of various model architectures. Several metrics can be used to assess the complexity of NNs, which can be categorized into four key areas: time, space/architecture complexity, parallelism complexity, and generalization. Adopting a multidimensional perspective is important because a single metric cannot provide a holistic understanding of the true complexity of training. Each dimension offers unique insights, guiding informed trade-offs between model performance, resource requirements, and generalization capabilities. Next, we will further detail each of these key areas of training complexity measurements:

Time complexity: The traditional measures refer to the training time and the number of epochs required to achieve the desired performance. These metrics also take into consideration the learning rate and the optimizer used. Even though the training time metric and the number of epochs metrics show, to some degree, the training complexity, these metrics are a poor benchmark since the training time depends heavily on the hardware resources used and the size of the training dataset. In addition, two NNs with the same number of epochs to achieve the same performance can have very distinct training time, also depending on the batch size. To address these issues, an additional metric is proposed. The product of the number of epochs and the number of batches (NENB), reflects the model's computational demand. More training epochs generally indicate a more complex and computationally demanding model. The number of batches refers to the number of subsets of data used during each epoch, which is affected proportionally by the dataset size and batch size. The number of epochs and the number of batches cannot be evaluated separately, as one model may require more epochs but fewer batches, while another model may require fewer epochs but more batches. Lastly, FLOPs (Floating Point Operations) can be used to measure the number of floating-point operations required to train the model. A higher number of FLOPs typically indicates higher time complexity.

Space/Architecture Complexity: The number of trainable parameters, while commonly used, may not fully capture the complexity due to different architectural designs. For example, two NNs with the same number of trainable parameters can have very distinct training complexity [66]. The model architecture indicates the complexity of the NN architecture itself, such as the depth, width (e.g. the number of layers and neurons) of the network, and specific architectural choices such as recurrent, or convolution. The next metric is the memory requirement for storing the weights, biases of the NN, and intermediate computations during training. The space complexity can be influenced by the batch size used during training. Larger batch sizes might require more memory, particularly on GPU devices.

Parallelism Complexity: This aspect consists of data parallelism and model parallelism. Data parallelism is the parallelization of training across multiple devices by splitting the dataset. Model parallelism is about distributing the model across different devices for processing. Parallelizability also refers to how scalable the training process is with added computational resources and the efficiency of distributing the training process across multiple GPUs. For example, the MLP feed-forward NN is fully parallelizable and can result in faster training. To be more specific, the training time of RNN compared to the MLP with the same number of trainable parameters can be significantly longer, as the recurrent architecture of RNN is more complex than the feed-forward structure of the MLP.

Generalization: The flexibility/generalizability is assessed by estimating the number of operational ranges in which the NN equalizer operates with an acceptable gain. If the NN can only perform a specific task, it requires frequent re-training in the future, contributing to the overall complexity. Therefore, the NN that performs well in different but related tasks without re-training is preferable [43].

B. Complexity Metrics for Inference

Accurate computational complexity evaluation is critical in the design of digital signal processing (DSP) devices to better understand the implementation feasibility and bottlenecks for each device's structure. With this in mind, we summarize the

four most commonly used criteria for assessing computational complexity, from the software level to the hardware level, in Fig. 3.

1) Real Multiplications: The first, most software-oriented, level of estimation traditionally deals only with counting the number of real multiplications of the algorithm [67], [68] (quite often defined per one processed element, say a sample or a symbol). This metric is the number of real multiplications (RM). When comparing computational complexity, the purpose of this high-level metric is to consider only the multipliers required, ignoring additions, because the implementation of the latter in hardware or software is initially considered cheap, while the multiplier is generally the slowest element in the system and consumes the largest chip area [67], [69]. This ignoring of the additions can also be easily understood by looking at the Big-O analysis of multiplier versus adder. When multiplying two integers with n digits, the computational complexity of the multiplication instance is $O(n^2)$, whereas the addition of the same two numbers has a computational complexity of $\Theta(n)$ [70].[2] As a result, if you are dealing with float values with 16 decimal digits, multiplication is by far the most time-consuming part of the implementation procedure. Therefore, when comparing solutions that use floating-point arithmetic with the same bitwidth precision, the RM metric provides an acceptable comparative estimate to qualitatively assess the complexity against some existing benchmarks (e.g. against the DSP operations for optical channel equalization tasks [68]).

2) Number of Bit-Operations: When moving to fixed-point arithmetic, the second metric known as the number of bit-operations (BOP) must be adopted to understand the impact of changing the bitwidth precision on the complexity. The BOP metric provides a good insight into mixed-precision arithmetic performance since we can forecast the BOP needed for fundamental arithmetic operations like addition and multiplication, given the bitwidth of two operands. In a nutshell, the BOP metric aims to generalize floating-point operations (FLOPs) to heterogeneously quantized NNs, as far as the FLOPs cannot be efficiently used to evaluate integer arithmetic operations [25], [71]. For the BOP metric, we have to include the complexity contribution of both multiplications and additions, since now we evaluate the complexity in terms of the most common operations in NNs: the multiply-and-accumulate operations (MACs) [25], [71], [72]. However, the BOP accounts for the scaling of the number of multipliers with the bitwidth of two operands, and the scaling of the number of adders with the accumulator bitwidth. Note that since most real DSP implementations use dedicated logic macros (e.g. DSP slice in Field Programmable Gate Arrays [FPGA] or MAC in Application Specific Integrated Circuit [ASIC]), the BOP metric fits as a good complexity estimation metric inasmuch as the BOP also accesses the MAC taking into account the particular bitwidth of two operands.

[2]The Big-O notation represents the worst case or the upper bound of the time required to perform the operation, Big Omega (Ω) shows the best case or the lower bound, whereas the Big Theta (Θ) notation defines the tight bound of the amount of time required; in other words, $f(n)$ is claimed to be $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

3) Number of Additions and bit Shifts: The progress in the development of new advanced NN quantization techniques [74], [75], [76], [77] allowed implementing the fixed point multiplications participating in NNs efficiently, namely with the use of a few bit-shifters and adders [78], [79], [80]. Since the BOP lacks the ability to properly assess the effect of different quantization strategies on the complexity, a new, more sophisticated metric is required there. We introduce the third complexity metric that counts the number of total equivalent additions to represent the multiplication operation, called the number of additions and bit shifts (NABS). The number of shift operations can be neglected when calculating the computational complexity because, in the hardware, the shift can be performed without extra costs in constant time with the $O(1)$ complexity. Even though the cost of bit shifts can be ignored due to the aforementioned reasons, and only the total number of adders has to be accounted for to measure the computational complexity, we prefer to keep the full name "number of additions and bit shifts" to highlight that the multiplication is now represented as shifts and adders.

4) Number of Logic Gates: The metric closer to the hardware level is the number of logic gates (NLG) that is used for our evaluating method's hardware (e.g. ASIC or FPGA) implementation. It is different from the NABS metric, as now the true cost of implementation is to be presented. In this case, in contrast to the other complexity metrics, the cost of activation functions is also taken into account because, to achieve better complexity, they are frequently implemented using look-up tables (LUT) rather than adders and multipliers. Additionally, other metrics like the number of flip-flops (FFs) or registers, the number of logic blocks used for general logic and memory blocks, or other special functional macros used in the design, are also relevant. As it is clear from this explanation, there is no straightforward equation to convert the NABS to the NLG as the latter depends on the circuit design adopted by the developer. Tools such as Synopsys Synthesis [81] for ASIC implementation can provide this kind of information. However, regarding the FPGA design, it is harder to get a correct estimate of the gate count from the report of FPGA tools [82].

In this paper, we advocate that the NLG metric should be applied to count the number of logic gates used to implement the hardware piece, similar to the concept of the Maximum Logic Gates metric for FPGA devices [73]. The Maximum Logic Gates metric is utilized to approximate the maximum number of gates that can be realized in the FPGA for a design consisting of only logic functions.[3] Additionally, this metric is based on an estimate of the typical number of usable gates per configurable logic block (CLB) or logic cell multiplied by the total number of such blocks or cells [73]. Concerning the correspondence between CLB and logic gates number, see Table I.

It should be noted that Table I is based on an older, now obsolete, 4-input LUT architecture [73]. Newer FPGA families now feature a 6-input LUT architecture, and to address the resource consumption for the new generation of devices, a reasonable approximation would be to increase the 'maximum gate range equivalent per LUT' figure used in [73] by 50%.

[3]On-chip memory capabilities are not factored into this metric.

TABLE I
CAPACITY RANGES FOR XC4000 SERIES CLB RESOURCES GIVEN IN REF. [73]

CLB Resource	Logic Gate Range
Gate range per 4-input LUT (2 per CLB)	1 to 9
Gate range per 3-input LUT	1 to 6
Gate range per flip-flop (2 per CLB)	6 to 12
Total gate range per CLB	15 to 48
Estimated typical number of gates per CLB	28.5

TABLE II
ESTIMATED CAPACITY RANGES FOR 6 INPUT LUT-BASED CLB RESOURCES

CLB Resource	Logic Gate Range
Gate range per 6-input LUT (8 per CLB)	6 to 15
Gate range per flip-flop (16 per CLB)	6 to 12
Total gate range per CLB	144 to 312

Note that the gate equivalence figures for FF's (registers) still hold true for the 6-input architecture. It is also worth noting that the CLB architecture has changed substantially since Ref. [73] was published, such that we include Table II linking CLB-gates with a more up-to-date 6-input architecture.

To conclude, we comment on universal metrics between the FPGA and the ASIC implementations. We emphasize that calculating an ASIC gate equivalent to an FPGA DSP slice is not a straightforward task because not all features are necessarily required when implementing the specific arithmetic function in an ASIC. However, utilizing the estimation approach laid out in Ref. [73], a figure can be obtained. Using the Xilinx Ultrascale + DSP48E2 slice basic multiplier functionality as an example (see Xilinx UG579 Fig. 1–1 in Ref. [83]) and pipelining it for maximum performance, it is possible to estimate the number of FFs and adders required for such an ASIC equivalence. Taking into account the structure of the multiplication of a m-bit number by a n-bit number, implemented using an array multiplier architecture, it is equivalent to $m \times n$ AND gates, n half adders, and $(m-2) \times n$ full adders.[4] For example, the ASIC equivalence of a 27×18 multiplier in an FPGA would have 486 AND gates, 18 half adders, 450 full adders, and 90 FFs.

Note that, estimating the NABS that can be implemented on FPGA hardware presents a considerable challenge, owing to the multitude of factors influencing FPGA implementations. These factors include resource availability such as the number of CLBs and LUTs, the efficacy of programming methodologies to effectively utilize available resources, as well as constraints related to routing, placing, clock frequency, and throughput limitations, among others. Despite the complexity of this estimation, an initial assessment can be made by considering the potential parallelism of NABs per clock cycle, primarily leveraging LUTs and DSP blocks. For instance, one can consider the Xilinx FPGA VCK190 as an example. With 899,840 LUTs and 1,968 DSP blocks available, a rough estimation can be derived in an ideal scenario **utilizing LUTs**, up to 449,920 additions and bit shifts could theoretically be implemented. This estimation assumes a simplistic model where each operation requires only 2 LUTs. However, it's important to note that this is an oversimplification, and actual resource utilization may vary due to practical considerations such as routing constraints. Additionally, **utilizing DSP blocks**, designed for efficient arithmetic operations, can potentially accommodate an additional 1,968 operations. However, this estimation presupposes optimal utilization of LUTs, which may not always be achievable in practice. Furthermore, it's important to acknowledge that the actual feasibility of implementing NABs on FPGA hardware is influenced by factors such as clock speed, memory requirements, and specific application demands. Therefore, while these estimations provide valuable insights, they serve as initial benchmarks and must be validated through comprehensive analysis and optimization efforts tailored to the specific hardware platform and application requirements.

[4]Note that a half adder is equivalent to 1 AND gate + 1 XOR gate, and a full adder is equal to 2 AND gates + 2 XOR gates + 1 OR gate

Finally, this work especially focuses on the four inference complexity metrics of the NN (RM, BOP, NABS, and NLG). Apart from the aforementioned metrics, the power consumption, memory footprint, and complexity of the activation function should be assessed. Power consumption should be considered as it can result in a bottleneck during the implementation phase. Memory footprint refers to the input size of the time series and the number of parameters that need to be saved in the memory, taking into account the quantization scheme. This complexity of the activation function, considering different types of activation functions and the approximation techniques, is another aspect to keep in mind [56].

C. Usage of the Complexity Metrics in Photonic Hardware

It can be questionable if the aforementioned metrics proposed to access the computational complexity in the inference phase of the electronic hardware can be used to access the complexity in the photonic hardware too. When discussing the complexity of computations in photonic NN hardware [84], [85] or photonics integrated circuits (e.g. PICs), it involves different considerations compared to traditional electronic computing (e.g., FPGA and ASIC). We can discuss metric by metric as follows:

1) Real multiplication is also suitable for photonic computing as in the context of NN, matrix-vector multiplications are a core operation. Photonic computing allows multiplications without the need for converting to electronic signals [85].
2) Number of bit operations (BOP) are only partially suitable for photonic computing because photonic computing can use the analog properties and continuous values of light, such as amplitude and phase, to encode information. However, digital photonic computing also exists. Digital photonic computing represents information in optical on-off keying or other discrete states. BOP can be relevant but might not be the most comprehensive metric, depending on the types of photonic computing architectures.
3) Number of adders and bit shifts (NABS) might not be a suitable metric because the photonic hardware does not directly use shifts and adders in the same way as electronic circuits. The operations in photonic systems are more about manipulating light's physical properties

TABLE III
SUMMARY OF THE THREE COMPUTATIONAL COMPLEXITY METRICS PER LAYER (THE NUMBER OF REAL MULTIPLICATIONS, THE NUMBER OF BIT OPERATIONS, THE NUMBER OF ADDITIONS AND BIT SHIFTS) FOR A ZOO OF NEURAL NETWORK LAYERS AS A FUNCTION OF THEIR DESIGNING HYPER-PARAMETERS; THE NUMBER OF NEURONS (n_n), THE NUMBER OF FEATURES IN THE INPUT VECTOR (n_i), THE NUMBER OF FILTERS (n_f), THE KERNEL SIZE (n_k), THE INPUT TIME SEQUENCE SIZE (n_s), THE NUMBER OF HIDDEN UNITS (n_h), THE NUMBER OF INTERNAL HIDDEN NEURON UNITS OF THE RESERVOIR (N_r), SPARSITY PARAMETER (s_p), THE NUMBER OF OUTPUT NEURONS (n_o), WEIGHT BITWIDTH (b_w), INPUT BITWIDTH (b_i), ACTIVATION BITWIDTH (b_a) AND THE NUMBER OF ADDERS REQUIRED AT MOST TO REPRESENT THE MULTIPLICATION (X_w)

Network type	Real multiplications (RM)	Number of bit-operations (BOP)	Number of additions and bit shifts(NABS)
MLP	$n_n n_i$	$n_n n_i \left[b_w b_i + \text{Acc}(n_i, b_w, b_i)\right]$	$n_n n_i (X_w + 1)\text{Acc}(n_i, b_w, b_i)$
1D-CNN	$n_f n_i n_k \cdot OutputSize$	$OutputSize \cdot n_f \text{Mult}(n_i n_k, b_w, b_i)$ $+n_f \text{Acc}(n_i n_k, b_w, b_i)$	$OutputSize \cdot n_f \left[n_i n_k (X_w + 1) - 1\right]$ $\cdot \text{Acc}(n_i n_k, b_w, b_i)$ $+n_f \text{Acc}(n_i n_k, b_w, b_i)$
Vanilla RNN	$n_s n_h (n_i + n_h)$	$n_s n_h \text{Mult}(n_i, b_w, b_i)$ $+n_s n_h \text{Mult}(n_h, b_w, b_a)$ $+2 n_s n_h \text{Acc}(n_h, b_w, b_a)$	$n_s n_h \left[n_i (X_w + 1) - 1\right]\text{Acc}(n_i, b_w, b_i)$ $+n_s n_h \left[n_h (X_w + 1) + 1\right]\text{Acc}(n_h, b_w, b_a)$
LSTM	$n_s n_h (4n_i + 4n_h + 3)$	$4 n_s n_h \text{Mult}(n_i, b_w, b_i)$ $+4 n_s n_h \text{Mult}(n_h, b_w, b_a)$ $+3 n_s n_h b_a^2$ $+9 n_s n_h \text{Acc}(n_h, b_w, b_a)$	$4 n_s n_h \left[n_i (X_w + 1) - 1\right]\text{Acc}(n_i, b_w, b_i)$ $+4 n_s n_h \left[n_h (X_w + 1) + 1\right]\text{Acc}(n_h, b_w, b_a)$ $+6 n_s n_h b_a$
GRU	$n_s n_h (3n_i + 3n_h + 3)$	$3 n_s n_h \text{Mult}(n_i, b_w, b_i)$ $+3 n_s n_h \text{Mult}(n_h, b_w, b_a)$ $+3 n_s n_h b_a^2$ $+8 n_s n_h \text{Acc}(n_h, b_w, b_a)$	$3 n_s n_h \left[n_i (X_w + 1) - 1\right]\text{Acc}(n_i, b_w, b_i)$ $+n_s n_h \left[3 n_h (X_w + 1) + 5\right]\text{Acc}(n_h, b_w, b_a)$ $+6 n_s n_h b_a$
ESN	$n_s N_r (n_i + N_r s_p + 2 + n_o)$	$n_s N_r \text{Mult}(n_i, b_w, b_i)$ $+n_s N_r s_p \text{Mult}(N_r, b_w, b_a)$ $+n_s N_r \text{Mult}(n_o, b_w, b_a)$ $+2 n_s N_r b_a^2$ $+4 n_s N_r \text{Acc}(N_r, b_w, b_a)$	$n_s N_r \left[n_i (X_w + 1) - 1\right]\text{Acc}(n_i, b_w, b_i)$ $+n_s N_r \left[s_p (N_r X_w + N_r - 1\right] + 4)\text{Acc}(N_r, b_w, b_a)$ $+n_s N_r \left[n_o (X_w + 1) - 1\right]\text{Acc}(n_o, b_w, b_a)$ $+4 n_s N_r b_a$
ResNet	Eq. (36)	Eq. (38)	Eq. (40)
Transformer	MultiHead: Eq. (47) Point-wise FFN: Eq. (50) Add&Norm: Eq. (53)	MultiHead: Eq. (48) Point-wise FFN: Eq. (51) Add&Norm: Eq. (54)	MultiHead: Eq. (49) Point-wise FFN: Eq. (52) Add&Norm: Eq. (55)

through optical components, for example, waveguides, modulators, and detectors.

4) Number of logic gates is not a suitable metric for the photonic hardware, however, a similar concept can be considered. Instead of the gate count as in the electronic domain, we can count the number of optical components like modulators, detectors, and waveguides in the photonic implementation.

It is important to consider other factors when assessing the trade-off between the performance and complexity of equalizers and the associated hardware. For instance, aspects such as power efficiency [84] and the comparison of physical size or chip area are crucial. Note, that a direct comparison of raw power consumption between PICs and ASICs might be misleading. To accurately assess power efficiency, we should consider throughput, which is the amount of data processed per unit of time. A more insightful metric is energy efficiency, obtained by dividing the power consumption by the achieved throughput. This allows for a fair comparison, highlighting the ability of each technology to perform a specific task with minimal energy consumption.

The chip area is another critical factor in many applications. When comparing the area occupied by a PIC and an ASIC equalizer, it is essential to consider the functionality delivered per unit area. A larger PIC might integrate functionalities beyond simple equalization, offering a higher overall value proposition despite its larger footprint [86]. Conversely, a smaller ASIC might excel in scenarios where compactness is paramount, even with a more limited functional range.

Finally, factors like scalability, reconfigurability, and sensitivity to fabrication imperfections can influence the choice between PICs and ASICs [87].

IV. MATHEMATICAL COMPLEXITY FORMULATION

In this section, we provide a brief introduction to various types of NN: dense layer, Convolutional Neural Networks (CNN), Vanilla Recurrent Neural Networks (RNN), Long Short-Term Memory Neural Networks (LSTM), Gated Recurrent Units (GRU), and Echo State Networks (ESN). We investigate the computational complexity of each network in terms of RM, BOP, and NABS. In this work, the computational complexity is formulated per layer, and the output layer is not taken into account for the complexity calculation to eliminate redundant computations if multiple layers or multiple NN types are combined. Table III in the section's end summarizes the formulas for the RM, BOP, and NABS for all NN types studied. *Furthermore, we have included all the complexity equations utilized in this study in Python code [88]. This resource is intended to aid readers in calculating the complexity of their own NN architectures.*

A. Dense Layer

A dense layer, also known as a 'fully connected layer', is a layer in which each neuron is connected with all the neurons

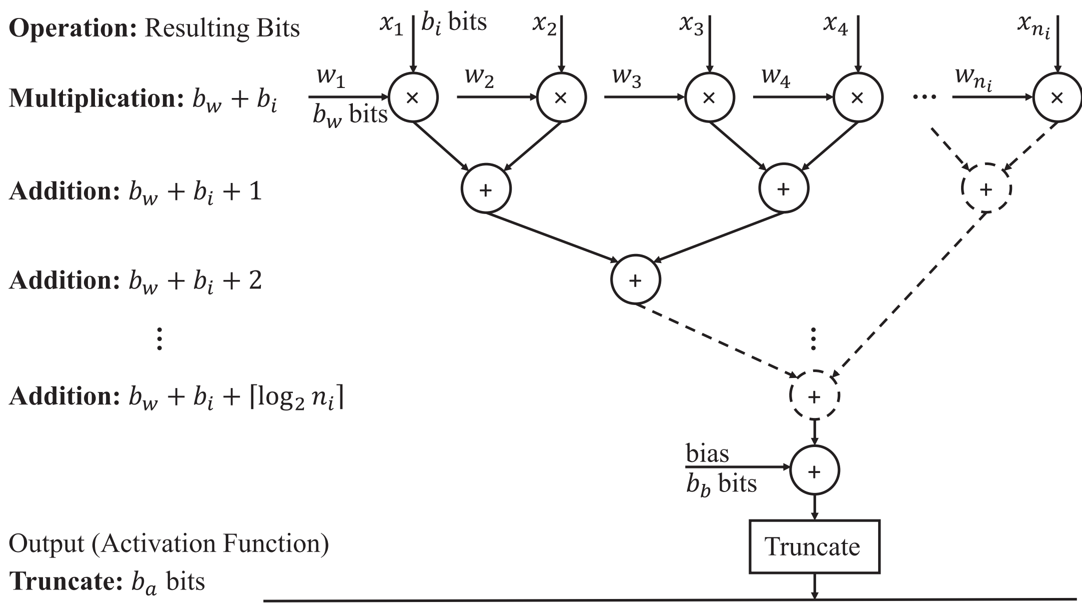


Fig. 4. Data path of a neuron in a quantized dense layer where x is the input vector with size n_i, w is the weight matrix, b_w is the weight bitwidth and b_i is the input bitwidth, b_a is the activation bitwidth and b_b is the bias bitwidth.

from the previous layer with a specific weight w_{ij}. The input vector is mapped to the output vector in a nonlinear manner by the dense layer, due to the participation of a non-linear activation function. Dense layers can be combined to form a Multi-Layer Perceptron (MLP), which is a class of a feed-forward deep NN.

The output vector y of a dense layer given x as an input vector is written as:

$$y = \phi(Wx + b), \tag{1}$$

where y is the output vector, ϕ is a nonlinear activation function, W is the weight matrix, and b is the bias vector. Writing explicitly the matrix operation inside the activation function:

$$Wx + b = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n_i} \\ w_{21} & w_{22} & \dots & w_{2n_i} \\ \vdots & \vdots & \dots & \vdots \\ w_{n_n1} & w_{n_n2} & \dots & w_{n_nn_i} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_i} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_n} \end{bmatrix}, \tag{2}$$

where n_i is the number of features in the input vector and n_n represents the number of neurons in the layer, we can readily see that the RM of a dense layer can be computed according to the simple, well-known formula:

$$\text{RM}_{\text{Dense}} = n_n n_i. \tag{3}$$

Now we calculate the BOP of a dense layer, taking into account the bitwidth of two operands, to approximate the computational complexity of NNs when the mixed-precision arithmetic is used. The bitwidth, also known as the precision, is the number of bits used to represent a certain element; for example, each weight in the weight matrix can be represented with b_w bits. Fig. 4 illustrates the data flow of the MAC operations for a neuron of a dense layer with n_i input features and b_i as input bitwidth. The multiplication of the input vector and the weights for one neuron can be mathematically represented as follows:

$$y_{\text{MUL, one neuron}} = \sum_{n=1}^{n_i} w_n x_n. \tag{4}$$

Initially, the n_i multiplications of input vector elements and weights for one neuron take place. When the multiplication of two operands is performed, the resulting bitwidth is the sum of the bitwidths of two operands ($b_w + b_i$) as shown in the first row of Fig. 4.

After that, $n_i - 1$ additions need to be made, and the resulting number of bits can be defined as follows: Considering that the result of the addition of two operands has the bitwidth of the bigger operand plus one bit, we start adding the multiplication results pairwise until only one element remains. In this case, the second row of Fig. 4 shows the first level of pairwise additions, with a resulting bitwidth of $b_w + b_i + 1$, and since this pairwise addition process is repeated for $\lceil \log_2(n_i) \rceil$ levels (until we have just a final single number), the total bitwidth of it is given by $b_w + b_i + \lceil \log_2(n_i) \rceil$, i.e. it is the bitwidth required to perform the overall MAC process. Then, the addition of the bias vector is performed. In this work, for all types of networks, we assume that the size of the accumulator defined by the multiplication of the weight matrix and the input vector, is dominant; thereby, the assumption for the bias bitwidth b_b is as follows: $b_b < b_w + b_i + \lceil \log_2(n_i) \rceil$, and the addition of bias, in the end, will not result in the overflow. Finally, the bitwidth of the resulting number is truncated to b_a, where b_a is the bitwidth of the activation function [20].

When calculating the BOP for a dense layer, the costs of both multiplications and additions need to be included. Then, the BOP formula takes the form of the sum of two constituents, BOP_{Mul} and BOP_{Bias}, corresponding to vector-matrix multiplication and

bias addition:

$$\text{BOP}_{\text{Mul}} = n_n \left[n_i b_w b_i + (n_i - 1)(b_w + b_i + \lceil \log_2(n_i) \rceil) \right], \tag{5}$$

$$\text{BOP}_{\text{Bias}} \approx n_n (b_w + b_i + \lceil \log_2(n_i) \rceil). \tag{6}$$

Equation (5) shows the cost of the number of one-bit full adders calculated from the dot product of n_i-dimensional input vector and weight matrix, as in Refs. [25], [89]. The cost takes into account the bitwidths of the weights and input, b_w and b_i. To compute the product of the two operands, we have to use $n_i n_n$ multiplications and $n_n(n_i - 1)$ additions. The multiplication cost can be calculated by the number of multiplications multiplied by $b_w b_i$, which is related to the bit operation, and the number of additions multiplied by the accumulator bitwidth required to do the operation. The final BOP is the contribution of multiplication and the addition of bias of the dense layer. For the convenience of the forthcoming presentation, let us define the short notations:

$$\text{Mult}(n_i, b_w, b_i) = n_i b_w b_i + (n_i - 1)\left(b_w + b_i + \lceil \log_2(n_i) \rceil\right),$$

and

$$\text{Acc}(n_i, b_w, b_i) = b_w + b_i + \lceil \log_2(n_i) \rceil.$$

The Acc expression represents the actual bitwidth of the accumulator required for MAC operation, as shown in Fig. 4. Then, the BOP of the dense layer expressed through the layer parameters becomes:

$$\begin{aligned} \text{BOP}_{\text{Dense}} &= \text{BOP}_{\text{Mul}} + \text{BOP}_{\text{Bias}} \\ &\approx n_n n_i \left[b_w b_i + (b_w + b_i + \lceil \log_2(n_i) \rceil) \right] \\ &\approx n_n n_i \left[b_w b_i + \text{Acc}(n_i, b_w, b_i) \right]. \end{aligned} \tag{7}$$

Now, we note that with the advancement in NN quantization techniques, there arises the opportunity to approximate multiplication by using shift and few add operations only while still maintaining a good processing accuracy, since the NNs can diminish the approximation error that the quantized approximation introduces[5] [79], [90]. As mentioned in Section III, the number of shifts can be neglected compared to the contribution of adders. The number of adders is different for different types of quantization. To be more specific, let X represent the number of adders required, at most, to perform the multiplication and let b be the bitwidth of the quantized matrix. For uniform quantization, we have $X = b - 1$. And, for example, when the weight matrix with bitwidth of b_w, is quantized, we have $X_w = b_w - 1$ as the number of adders we need at most to perform the multiplication of the weights.[6] In the case of Power-of-Two (PoT) quantization, we have $X = 0$, because each multiplication costs just a shift [78], [94]. Lastly, for the Additive Powers-of-Two (APoT) quantization, we have $X = n$, where n denotes the number of additive terms. In APoT, the sum of n PoT terms is used to represent each quantization level [74]. Eventually, the NABS of a dense layer can be derived from its BOP equation, (7):

$$\begin{aligned} \text{NABS}_{\text{Dense}} &\approx n_n n_i \left[X_w \text{Acc}(n_i, b_w, b_i) + \text{Acc}(n_i, b_w, b_i) \right] \\ &\approx n_n n_i (X_w + 1) \text{Acc}(n_i, b_w, b_i). \end{aligned} \tag{8}$$

As in (8), the multiplication term $b_w b_i$ in (7) is converted into the number of adders needed to operate the multiplication times the accumulator bitwidth required: $X_w \text{Acc}(n_i, b_w, b_i)$.

B. Convolutional Neural Networks

In CNN, we apply the convolutions with different filters to extract the features and convert them into a lower-dimensional feature set, while still preserving the original properties. CNNs can be used in 1D, 2D, or 3D networks, depending on the applications. In this paper, we focus on 1D-CNNs, which apply to processing sequential data [3]. For simplicity of understanding, the 1D-CNN processing with padding equal to 0, dilation equal to 1, and stride equal to 1, can be summarized as follows:

$$y_i^f = \phi \left(\sum_{n=1}^{n_i} \sum_{j=1}^{n_k} x_{i+j-1,n}^{in} \cdot k_{j,n}^f + b^f \right), \tag{9}$$

where y_i^f denotes the output, known as a feature map, of a convolutional layer built by the filter f in the i-th input element, n_k is the kernel size, n_i is the size of the input vector, x^{in} represents the raw input data, k_j^f denotes the j-th trainable convolution kernel of the filter f and b^f is the bias of the filter f.

In the general case, when designing the CNN, parameters like padding, dilation, and stride also affect the output size of the CNN. It can be formulated as:

$$\text{OutputSize} = \left\lfloor \frac{n_s + 2\,padding - dilation(n_k - 1) - 1}{stride} + 1 \right\rfloor, \tag{10}$$

where n_s is the input time sequence size.

The RM of a 1D-convolutional layer can be computed as follows:

$$\text{RM}_{\text{CNN}} = n_f n_i n_k \cdot \text{OutputSize}, \tag{11}$$

where n_f is the number of filters, also known as the output dimension. As in (11), there are $n_i n_k$ multiplications per sliding window, and the number of times that sliding window process needs to be repeated is equal to the output size. Then, the procedure is executed repeatedly for all n_f filters.

The BOP for a 1D-convolutional layer, after taking into consideration the multiplications and additions, can be represented as:

$$\begin{aligned} \text{BOP}_{\text{CNN}} = \text{OutputSize} \cdot n_f &\text{Mult}(n_i n_k, b_w, b_i) \\ &+ n_f \text{Acc}(n_i n_k, b_w, b_i). \end{aligned} \tag{12}$$

[5]Note that using the shifts and adders to perform multiplications can cause some quantization noise/error since we are converting from a float-point representation to a fixed-point representation with some defined quantized level of values. However, in NNs, this noise can be partially mitigated by including those quantized weights in the NN training process as in Refs. [75], [76], [77]

[6]Note that we can consider other techniques for the representation of such fixed-point multiplication to reduce its complexity e.g. the double-base number system where each multiplication with b bits, at worst, needs no more than $b/log(b)$ additions [91]. For the Canonical Signed Digit (CSD) representation, in the worst-case scenario, we have $(b+l)/2$ nonzero bits and on average it tends asymptotically to $(3b+l)/9$ [92], [93]

Equation (12) is derived from (9) and (11). The first term is associated with the convolution operation between the flattened input vector and the sliding windows, and the latter term corresponds to the addition of the bias.

The procedure to derive the NABS is similar to that described in detail in the case of a dense layer, Section IV-A. The NABS of a 1D-convolutional layer is given by:

$$\begin{aligned}\text{NABS}_{\text{CNN}} = \text{OutputSize} \cdot n_f\,[n_i n_k (X_w + 1) - 1] \\ \cdot \text{Acc}(n_i n_k, b_w, b_i) \\ + n_f \text{Acc}(n_i n_k, b_w, b_i).\end{aligned} \tag{13}$$

To obtain the 1D-convolutional layer's NABS, the multiplication in (12) is represented by the number of adders required, at most, to perform the multiplication times the accumulator bitwidth.

C. Vanilla Recurrent Neural Networks

Vanilla RNN is different from MLP and CNN in terms of its ability to handle memory, which is quite beneficial for time series data. RNNs take into account the current input and the output that the network has learned from the prior input. Even though the RNNs introduced efficient memory handling, they still suffer from the inability to capture the long-term dependencies because of the vanishing gradient issue [95]. The equation for the vanilla RNN given a time step t is as follows:

$$h_t = \phi(W x_t + U h_{t-1} + b), \tag{14}$$

where ϕ is, again, the nonlinear activation functions, $x_t \in \mathbb{R}^{n_i}$ is the n_i-dimensional input vector at time t, $h_t \in \mathbb{R}^{n_h}$ is a hidden layer vector of the current state with size n_h, $W \in \mathbb{R}^{n_h \times n_i}$ and $U \in \mathbb{R}^{n_h \times n_h}$ represent the trainable weight matrices, and b is the bias vector. For more explanations on the vanilla RNN operation, see Ref. [96]. The RM of a vanilla RNN is:

$$\text{RM}_{\text{RNN}} = n_s n_h (n_i + n_h), \tag{15}$$

where n_h notes the number of hidden units. From (15), the RM for a time step is $n_h(n_i + n_h)$. It can be separated into two terms; the $n_h n_i$ term corresponds to the multiplication of the input vector x_t and the weight matrix, and the n_h^2 term arises because of the multiplication to the prior cell output h_{t-1}. Finally, n_s, which denotes the number of time steps in the layer, should be taken into account, as the process is repeated n_s times.

The BOP for a vanilla RNN is given as:

$$\begin{aligned}\text{BOP}_{\text{RNN}} = n_s n_h \text{Mult}(n_i, b_w, b_i) \\ + n_s n_h \text{Mult}(n_h, b_w, b_a) \\ + 2 n_s n_h \text{Acc}(n_h, b_w, b_a).\end{aligned} \tag{16}$$

From (16), the first term is associated with the input vector multiplied by the weight matrix, and the second term corresponds to the multiplications of the recurrent cell outputs. The final term is the contribution of the addition between $W x_t + U h_{t-1}$ and the addition of the bias vector in (14); one can see that the size of the accumulator used in this term, is $\text{Acc}(n_h, b_w, b_a)$. It is due to the assumption that $\text{Acc}(n_h, b_w, b_a)$ is dominant because it should be greater than $\text{Acc}(n_i, b_w, b_i)$ as a result of the inequality $n_h > n_i$.

As in the case of a dense layer, the NABS of vanilla RNN can be calculated from its BOP equation by converting the multiplication to the number of adders needed at most (X) depending on the quantization scheme and the accumulator size:

$$\begin{aligned}\text{NABS}_{\text{RNN}} = n_s n_h\,[n_i (X_w + 1) - 1]\,\text{Acc}(n_i, b_w, b_i) \\ + n_s n_h\,[n_h (X_w + 1) + 1]\,\text{Acc}(n_h, b_w, b_a).\end{aligned} \tag{17}$$

D. Long Short-Term Memory Neural Networks

LSTM is an advanced type of RNNs. Although RNNs suffer from short-term memory issues, the LSTM network can learn long-term dependencies between time steps (t), insofar as it was specifically designed to address the gradient issues encountered in RNNs [97], [98]. There are three types of gates in an LSTM cell: an input gate (i_t), a forget gate (f_t), and an output gate (o_t). More importantly, the cell state vector (C_t) was proposed as a long-term memory to aggregate the relevant information throughout the time steps. The equations for the forward pass of the LSTM cell given a time step t are as follows:

$$\begin{aligned}i_t &= \sigma(W^i x_t + U^i h_{t-1} + b^i), \\ f_t &= \sigma(W^f x_t + U^f h_{t-1} + b^f), \\ o_t &= \sigma(W^o x_t + U^o h_{t-1} + b^o), \\ C_t &= f_t \odot C_{t-1} + i_t \odot \phi(W^c x_t + U^c h_{t-1} + b^c), \\ h_t &= o_t \odot \phi(C_t),\end{aligned} \tag{18}$$

where ϕ is usually the "tanh" activation function, σ is usually the sigmoid activation function, the sizes of each variable are $x_t \in \mathbb{R}^{n_i}$, $f_t, i_t, o_t \in (0,1)^{n_h}$, $C_t \in \mathbb{R}^{n_h}$ and $h_t \in (-1,1)^{n_h}$. The $\odot$ symbol represents the element-wise (Hadamard) multiplication.

The RM of an LSTM layer is:

$$\text{RM}_{\text{LSTM}} = n_s n_h (4 n_i + 4 n_h + 3), \tag{19}$$

where n_h is the number of hidden units in the LSTM cell. Similarly to RNNs, the RM can be calculated from the term associated with the input vector x_t and the term corresponding to the prior cell output h_{t-1}; however, each term occurs four times, as we can see in (18). Therefore, we have $4 n_h n_i$ and $4 n_h^2$, respectively. Moreover, we also need to include the element-wise product that is operated three times in (18), which costs $3 n_h$. Finally, the process is repeated n_s times, hence, n_s is multiplied to the overall number.

The BOP for an LSTM layer is computed based on (19), but also includes the bitwidth of the operands and the number of additions. As a result, the BOP can be represented as:

$$\begin{aligned}\text{BOP}_{\text{LSTM}} = 4 n_s n_h \text{Mult}(n_i, b_w, b_i) \\ + 4 n_s n_h \text{Mult}(n_h, b_w, b_a) \\ + 3 n_s n_h b_a^2 \\ + 9 n_s n_h \text{Acc}(n_h, b_w, b_a).\end{aligned} \tag{20}$$

To give more details on the expression, the first two terms in (20) are the contribution of the input vector multiplications and the recurrent cell output association, respectively. The term $3n_s n_h b_a^2$ refers to 3 times of the element-wise product of two operands with b_a bitwidth, see (18). For each time step, there are n_h elements in each vector that need to be multiplied. The last term is for all the additions since we assume that $\text{Acc}(n_h, b_w, b_a)$ gives the dominant contribution, as described in Section IV-C. Finally, the process is then restarted n_s times.

The NABS of an LSTM layer is derived from the (20) by replacing the multiplications with the shifts and adders including their cost, as mentioned in Section III that the shifts would not be included. The number of adders depends on the quantization technique. The NABS would be as follows:

$$\begin{aligned}\text{NABS}_{\text{LSTM}} &= 4n_s n_h \left[n_i(X_w+1)-1\right]\text{Acc}(n_i, b_w, b_i)\\ &\quad + 4n_s n_h \left[n_h(X_w+1)+1\right]\text{Acc}(n_h, b_w, b_a)\\ &\quad + 6n_s n_h b_a. \end{aligned} \tag{21}$$

The first and second terms are the results of input vector multiplications and the recurrent cell operation combined with all addition operations, respectively. In this case, the third term comes from $3n_s n_h(b_a+b_a)$. Due to the element-wise product of two operands with bitwidth b_a, the resulting bitwidth becomes b_a+b_a as mentioned in Fig. 4.

E. Gated Recurrent Units

Like LSTM, the GRU network was created to overcome the short-term memory issues of RNNs. However, GRU is less complex, as it has only two types of gates: reset (r_t) and update (z_t) gates. The reset gate is used for short-term memory, whereas the update gate is responsible for long-term memory [99]. In addition, the candidate hidden state (h'_t) is also introduced to state how relevant the previous hidden state is to the candidate state. The GRU for a time step t can be formalized as:

$$\begin{aligned} z_t &= \sigma(W^z x_t + U^z h_{t-1} + b^z),\\ r_t &= \sigma(W^r x_t + U^r h_{t-1} + b^r),\\ h'_t &= \phi(W^h x_t + r_t \odot U^h h_{t-1} + b^h),\\ h_t &= z_t \odot h_{t-1} + (1-z_t)\odot h'_t, \end{aligned} \tag{22}$$

where ϕ is typically the "tanh" activation function and the rest of the designations are the same as in (18).

The RM of the GRU is calculated in the same way as we did for the LSTM in (19), but the number of operations with the input vector x_t and with the previous cell output h_{t-1} is reduced from four (LSTM) to three times as shown in (22). Thus, the expression for the RM becomes:

$$\text{RM}_{\text{GRU}} = n_s n_h(3n_i + 3n_h + 3). \tag{23}$$

The BOP for the GRU can be calculated in the same manner as we did for the LSTM in (20). However, now, the expression is slightly different in the number of matrix multiplications as the number of gates is now lower. The BOP number can be represented as:

$$\begin{aligned}\text{BOP}_{\text{GRU}} &= 3n_s n_h \text{Mult}(n_i, b_w, b_i)\\ &\quad + 3n_s n_h \text{Mult}(n_h, b_w, b_a)\\ &\quad + 3n_s n_h b_a^2\\ &\quad + 8n_s n_h \text{Acc}(n_h, b_w, b_a). \end{aligned} \tag{24}$$

The explanation for each line here is identical to that in (20).

The NABS of the GRU is derived similarly to the LSTM case:

$$\begin{aligned}\text{NABS}_{\text{GRU}} &= 3n_s n_h \left[n_i(X_w+1)-1\right]\text{Acc}(n_i, b_w, b_i)\\ &\quad + n_s n_h \left[3n_h(X_w+1)+5\right]\text{Acc}(n_h, b_w, b_a)\\ &\quad + 6n_s n_h b_a. \end{aligned} \tag{25}$$

Again, the explanation for each term in this expression is identical to (21).

F. Echo State Networks

ESN belongs to the class of recurrent layers, but more specifically, to the reservoir computing category. ESN was proposed to relax the training process while being efficient and simple to implement. The ESN comprises three layers: an input layer, a recurrent layer, known as a reservoir, and an output layer, which is the only layer that is trainable. The reservoir with random weight assignment is used to replace back-propagation in traditional NNs to reduce the computational complexity of training [100]. We notice that the reservoir of the ESNs can be implemented in two domains: digital and optical [101]. With the optical implementation of the reservoir, the computational complexity dramatically falls, however, the degradation of the performance due to the change of domain is noticeable [102]. In this work, we only examine the digital domain implementation. Moreover, we focus on the leaky-ESN, as it is believed to often outperform standard ESNs and is more flexible due to time-scale phenomena [103], [104]. The equations of the leaky-ESN for a certain time step t are given as:

$$a_t = \phi\left(W^r s_{t-1} + W^{\text{in}} x_t\right), \tag{26}$$

$$s_t = (1-\mu)s_{t-1} + \mu a_t, \tag{27}$$

$$y_t = W^o s_t + b^o, \tag{28}$$

where s_t represents the state of the reservoir at time t, W^r denotes the weight of the reservoir with the sparsity parameter s_p, W^{in} is the weight matrix that shows the connection between the input layer and the hidden layer, μ is the leaky rate, W^o denotes the trained output weight matrix, and y_t is the output vector.

The RM of an ESN is given by

$$\text{RM}_{\text{ESN}} = n_s N_r(n_i + N_r s_p + 2 + n_o), \tag{29}$$

where N_r is the number of internal hidden neuron units of the reservoir and n_o denotes the number of output neurons. From (29), the $N_r n_i$ multiplications occur from the input vector operations, and the term $N_r^2 s_p$ is included due to the reservoir layer; to be more specific, the latter term is multiplied with the sparsity parameter s_p which indicates the ratio of zero values

in the matrix. Eq (27) results in $2N_r$ multiplications. Unlike the other network types, now we have to include the contribution of the output layer explicitly because it contains the trainable weight matrix, and this layer contributes $N_r n_o$ multiplications. Eventually, the process is repeated for n_s times.

The BOP number for an ESN can be represented as:

$$\begin{aligned}\text{BOP}_{\text{ESN}} = {} & n_s N_r \text{Mult}(n_i, b_w, b_i) \\ & + n_s N_r s_p \text{Mult}(N_r, b_w, b_a) \\ & + n_s N_r \text{Mult}(n_o, b_w, b_a) \\ & + 2 n_s N_r b_a^2 \\ & + 4 n_s N_r \text{Acc}(N_r, b_w, b_a). \end{aligned} \tag{30}$$

In (30), the first term is the input vector contribution, the second one is contributed by the reservoir layer, the third term refers to the output layer multiplications, and the fourth term stems from the multiplications in (27). Eventually, all the addition operations are accounted for by the final term.

The NABS of an ESN, which can be calculated similarly as in the LSTM case in Section IV-D, is:

$$\begin{aligned}\text{NABS}_{\text{ESN}} = {} & n_s N_r \left[n_i (X_w + 1) - 1 \right] \text{Acc}(n_i, b_w, b_i) \\ & + n_s N_r \left[s_p (N_r X_w + N_r - 1 \right] + 4) \text{Acc}(N_r, b_w, b_a) \\ & + n_s N_r \left[n_o (X_w + 1) - 1 \right] \text{Acc}(n_o, b_w, b_a) \\ & + 4 n_s N_r b_a. \end{aligned} \tag{31}$$

By changing the multiplication terms in (30) to the number of adders required at most, we obtain the ESN's NABS. The input vector multiplication contributes to the first term. The reservoir layer and all the addition operations result in the second term. The third term comes from the output layer of the ESN. The last term is the contribution of (27).

G. Residual Neural Network

Residual Neural Network (ResNet) [105] is a feed-forward NN architecture with shortcut connections to skip one or more layers and perform the identity mapping. ResNet addresses vanishing gradient and the degradation problems [105], [106]. The degradation problems relate to the saturation and the rapid degradation of the accuracy of the deeper networks when they start to converge. It has shown remarkable performance in computer vision tasks. ResNet primarily utilizes 2D-convolutional layers as its core building blocks to extract features. The output of ResNet building block, see Fig. 5, is derived as follows:

$$\begin{aligned} y_{\text{ResNet}} = {} & \phi(F(x_l) + x_l) \\ & + \phi(x_{l+1} * k_{l+1} + x_l) \\ & + \phi(\phi(y_l) * k_{l+1} + x_l) \\ & + \phi(\phi(x_l * k_l) * k_{l+1} + x_l), \end{aligned} \tag{32}$$

where ϕ is the activation function which, in the original paper, is the linear rectifier unit (ReLU)

For simplicity of the calculation for each layer, the 2D-convolution output without the activation function, assumed

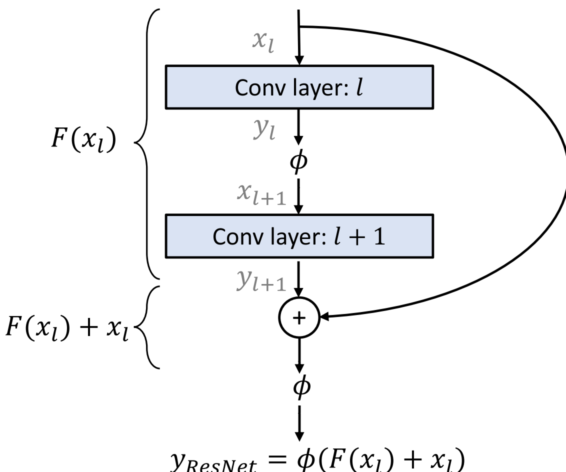


Fig. 5. Architecture of a ResNet block.

padding equal to 0, dilation equal to 1, and stride equal to 1, can be represented as follows:

$$y_{i,j}^f = (x^{in} * k^f)_{i,j} = \sum_{p=1}^{n_i} \sum_{m=1}^{n_{k_1}} \sum_{n=1}^{n_{k_2}} x_{i+m,j+n,p}^{in} \cdot k_{m,n,p}^f + b^f, \tag{33}$$

where $y_{i,j}^f$ denotes the output element of the feature map, of a convolutional layer built by the filter f, with i and j representing the row and column indices of the output element, $x^{in} \in \mathbb{R}^{n_i \times n_s}$ represents the raw input data and n_{k_1} and n_{k_2} are the dimension of the 2D kernel k^f.

The size of the feature map or convolution output, in fact, depends on other parameters like padding, dilation, and stride. The width and height of the feature map are formulated as follows:

$$\begin{aligned} \text{Width} &= \left[\frac{n_x + 2\,padding - dilation(n_{k_1} - 1) - 1}{stride} + 1 \right], \\ \text{Height} &= \left[\frac{n_s + 2\,padding - dilation(n_{k_2} - 1) - 1}{stride} + 1 \right], \\ \text{OutputSize} &= \text{Width} \times \text{Height}, \end{aligned} \tag{34}$$

where n_s is the first dimension of the input and n_x is the second dimension of the input. In the case of 2D CNN, the n_s does not refer to the time step because 2D CNN is usually applied to the image processing rather than the time series data. Note that $x^{in} \in \mathbb{R}^{n_x \times n_s \times n_i}$ Therefore, we have $y^f \in \mathbb{R}^{\text{Width} \times \text{Height}}$.

The RM of each 2D-convolutional layer can be calculated by

$$\text{RM}_{\text{2D-CNN}} = n_f n_i n_{k_1} n_{k_2} \cdot \text{OutputSize}, \tag{35}$$

where n_f is the number of filters (the output dimension) and n_i is the number of features. For each feature, (33), consists of $n_{k_1} n_{k_2}$ multiplications per sliding window, and the number of times the sliding window process needs to be repeated is equal to the output size. The calculation is repeated for each feature, in total n_i times. Then, the whole procedure is repeated for all n_f filters.

The ResNet block consists of several convolutional layers, in this case, two layers. Eq (32) shows two convolution operations. The RM of a ResNet block is given by the summation of $\text{RM}_{\text{2D-CNN}}$ of each layer as follows:

$$\text{RM}_{\text{ResNet}} = \text{RM}^{l}_{\text{2D-CNN}} + \text{RM}^{l+1}_{\text{2D-CNN}}, \tag{36}$$

where l and $l+1$ represent the first and second layers in the ResNet block, respectively.

The BOP of each 2D-convolutional layer can be formulated as:

$$\begin{aligned} \text{BOP}_{\text{2D-CNN}} = \text{OutputSize} \cdot n_f \text{Mult}(n_i n_{k_1} n_{k_2}, b_w, b_i) \\ + n_f \text{Acc}(n_i n_{k_1} n_{k_2}, b_w, b_i). \end{aligned} \tag{37}$$

The first term refers to the multiplication of the kernel and the input, and the second term refers to the bias addition. Please note that b_i is the bitwidth of the layer input. When connecting the layer in sequence, the bitwidth of the layer can be the result of the previous layer, or usually it is quantized as a result of the activation function.

The BOP of the ResNet block can be calculated from the BOP of two 2D-convolutional layers and the addition of the input (skip connection) as:

$$\begin{aligned} \text{BOP}_{\text{ResNet}} = \text{BOP}^{l}_{\text{2D-CNN}} + \text{BOP}^{l+1}_{\text{2D-CNN}} \\ + \text{Acc}(n_i n_{k_1} n_{k_2}, b_w, b_i). \end{aligned} \tag{38}$$

The addition term of the skip connection has BOP complexity of the larger operand which is the bitwidth of the convolution of the layer $l+1$, as can be observed in (32).

The NABS of the 2D-convolutional layer is formulated as follows:

$$\begin{aligned} \text{NABS}_{\text{2D-CNN}} = \text{OutputSize} \cdot n_f \left[n_i n_{k_1} n_{k_2} (X_w + 1) - 1 \right] \\ \cdot \text{Acc}(n_i n_{k_1} n_{k_2}, b_w, b_i) \\ + n_f \text{Acc}(n_i n_{k_1} n_{k_2}, b_w, b_i). \end{aligned} \tag{39}$$

The NABS of a ResNet block is a combination of two 2D-convolutional layers and the addition of the skip connection, the same way as we calculate BOP:

$$\begin{aligned} \text{NABS}_{\text{ResNet}} = \text{NABS}^{l}_{\text{2D-CNN}} + \text{NABS}^{l+1}_{\text{2D-CNN}} \\ + \text{Acc}(n_i n_{k_1} n_{k_2}, b_w, b_i). \end{aligned} \tag{40}$$

H. Transformer

The Transformer [107] is a deep learning architecture where, instead of using recurrence, it leverages attention mechanisms to capture global long-range dependencies between input and output. The Transformer model has played a dominant role in natural language processing, and become a state-of-the-art model in various language-related tasks. The Transformer architecture comprises different layers; multi-head self-attention, masked multi-head attention, feed-forward, and Add&Norm layer (residual connection and layer normalization).

The attention function is computed from a matrix Q which is a set of queries simultaneously, the matrix K and V for the keys and values. The dot products of the query with all keys are calculated and result in the output matrix:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V, \tag{41}$$

where $\frac{1}{\sqrt{d_k}}$ is a scaling factor computed from keys of dimension d_k. The dimensions of each matrix are $Q \in \mathbb{R}^{m \times d_k}$, $K \in \mathbb{R}^{n \times d_k}$ and $V \in \mathbb{R}^{n \times d_v}$

For multi-head attention, it can be defined as:

$$\begin{aligned} \text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_{\text{h}}) W^O, \\ \text{where head}_{\text{i}} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \end{aligned} \tag{42}$$

where h is the number of times we calculate self-attention with different sets of Q, K, and V. To reduce the dimension of each head, the queries, keys, and values with different learned linear projections are projected to d_k, d_k, and d_v dimensions, respectively. d_v represents the dimensional output values. $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W^O \in \mathbb{R}^{h d_v \times d_{\text{model}}}$ are learnable weight matrices.

In the masked multi-head attention, one way to mask out the inputs of the softmax function is to add the matrix M which contains 0's and $-\infty$'s. The $-\infty$'s correspond to invalid connections. The equation then is modified to

$$\text{Attention}_{\text{Masked}}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right) V \tag{43}$$

The fully-connected feed-forward network (FFN) is to perform two linear transformations with a ReLU in between, as follows:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2. \tag{44}$$

At the different positions where the linear transformations are applied, they use different parameters from layer to layer.

Regarding the Add&Norm layer, it involves a residual connection around each of the two sub-layers [105] followed by layer normalization [108]. The output of this Add&Norm layer is:

$$y_{\text{Add \& Norm}} = \text{LayerNorm}(x + \text{Sublayer}(x)), \tag{45}$$

where Sublayer(x)) refers to the function implemented by the sub-layer itself, for instance, FFN or MultiHead. The LayerNorm are then defined as:

$$\text{LayerNorm}(z) = \frac{g(z - \mu_z)}{\sigma_z} + b, \tag{46}$$

where g and b denote gain and bias, respectively, μ and σ are the mean and variance of the summed inputs within each layer, respectively.

In this paper, we aim to calculate the RM, BOP, and NABS for each layer of the Transformer separately, to facilitate the custom-built transformer architecture. The RM of the multi-head attention equals to:

$$\begin{aligned} \text{RM}_{\text{Multi-head}} = h[m d_{\text{model}} d_k + n d_{\text{model}} d_k + n d_{\text{model}} d_v \\ + 2mn d_{\text{model}}] + m d_{\text{model}} h d_v, \end{aligned} \tag{47}$$

where $md_{\text{model}}d_k$, $nd_{\text{model}}d_k$ and $nd_{\text{model}}d_v$ result from multiplications of $QW_i^Q KW_i^K$, and VW_i^V, respectively. The term $2mnd_{\text{model}}$ comes from the multiplications in Attention function. The last term results from the multiplication in the MultiHead function.

The BOP of a multi-head layer is

$$\begin{aligned}\text{BOP}_{\text{Multi-head}} = h\,[&md_{\text{model}}\text{Mult}(d_k, b_Q, b_w)\\ &+ nd_{\text{model}}\text{Mult}(d_k, b_K, b_w)\\ &+ nd_{\text{model}}\text{Mult}(d_v, b_V, b_w)\\ &+ nm\text{Mult}(d_{\text{model}}, b_q, b_q)\\ &+ md_{\text{model}}\text{Mult}(n, b_a, b_{\text{q}})]\\ &+ hd_v m\text{Mult}(d_{\text{model}}, b_q, b_w), \end{aligned} \tag{48}$$

where b_Q, b_K, b_V and b_w are the bitwidth of matrices Q, K, V, and W respectively. b_a is the bitwidth of the activation function, in this case, softmax. We assume that the resulting bitwidth from the matrix multiplications is quantized to b_{q}. The first, second, and third terms correspond to multiplications of QW_i^Q, KW_i^K, and VW_i^V, respectively. The fourth and fifth terms are the contributions of the Attention function. The last term results from the Multi-Head function. The NABS of a multi-head layer can be calculated as:

$$\begin{aligned}\text{NABS}_{\text{Multi-head}} = h\,[&md_{\text{model}}\,[d_k(X+1)-1]\,\text{Acc}(d_k, b_Q, b_w)\\ &+ nd_{\text{model}}\,[d_k(X+1)-1]\,\text{Acc}(d_k, b_K, b_w)\\ &+ nd_{\text{model}}\,[d_v(X+1)-1]\,\text{Acc}(d_v, b_V, b_w)\\ &+ nm\,[d_{\text{model}}(X+1)-1]\,\text{Acc}(d_{\text{model}}, b_q, b_q)\\ &+ md_{\text{model}}\,[n(X+1)-1]\,\text{Acc}(n, b_a, b_{\text{q}})]\\ &+ hd_v m\,[d_{\text{model}}(X+1)-1]\,\text{Acc}(d_{\text{model}}, b_q, b_w). \end{aligned} \tag{49}$$

For the point-wise FFN, The calculation is nearly the same as the dense layer. The RM can be formulated as:

$$\text{RM}_{\text{FFN}} = 2d_{\text{model}}d_{\text{ff}}. \tag{50}$$

It results from two times of the multiplication with weight matrices W_1 and W_2 in FFN. Each weight matrix for FNN has the size of $W_1 \in \mathbb{R}^{d_{\text{model}}\times d_{\text{ff}}}$ and $W_2 \in \mathbb{R}^{d_{\text{ff}}\times d_{\text{model}}}$. Note that the input vector has a dimension of d_{model}.

The BOP of the point-wise FFN is

$$\begin{aligned}\text{BOP}_{\text{FNN}} = d_{\text{model}}d_{\text{ff}}\,[&b_w b_i + \text{Acc}(d_{\text{model}}, b_w, b_i)\\ &+ b_a b_i + \text{Acc}(d_{\text{ff}}, b_a, b_i)], \end{aligned} \tag{51}$$

where we assume that after the Relu activation function, the resulting bitwidth becomes b_a.

The NABS of the point-wise FFN can be found by:

$$\begin{aligned}\text{NABS}_{\text{FNN}} = d_{\text{model}}d_{\text{ff}}\,[&(X_w+1)\text{Acc}(d_{\text{model}}, b_w, b_i)\\ &+ (X_w+1)\text{Acc}(d_{\text{ff}}, b_a, b_w)]. \end{aligned} \tag{52}$$

Regarding the Add&Norm layer, the calculation of the RM is relatively simple:

$$\text{RM}_{\text{Add\&Norm}} = n_x, \tag{53}$$

where n_x refers to the number of input of the Add&Norm layer, (45).

Whereas the BOP, the additions are taken into account as follows:

$$\text{BOP}_{\text{Add\&Norm}} = n_x b_s b_x + (b_x + b_s + 2), \tag{54}$$

where b_s is the bitwidth used to represent the scalar, b_x is the bitwidth of the input and 2 results from two explicit additions of $x + \text{Sublayer}(x)$ and the addition of the bias.

The NABS of the Add&Norm layer is calculated as:

$$\text{NABS}_{\text{Add\&Norm}} = (n_x X + 1) + (b_x + b_s + 2). \tag{55}$$

I. CDC Block - Frequency Domain Equalizer

This FDE includes the chromatic dispersion compensation and the recovery of the signal dispersion broadening [109]. The FDE compensates for the dispersion by multiplying the signal by the opposite of the transfer function for dispersion. The FDE adapts its parameters on the fly according to the estimation of the built-up dispersion after the transmission. To have a benchmark, the computational complexity of the CDC using FDE is provided as follows [68], [109]:

$$C_{\text{CDC}} = 4 \cdot \left(\frac{N(\log_2 N + 1)q}{N - N_D + 1}\right) \tag{56}$$

where N corresponds to the FFT size, q is the oversampling ratio and $N_D = q\tau_D/T$ where τ_D/T is the dispersive channel impulse response and T is the symbol interval. Note that this is the RM for two polarizations, which requires four N-point FFTs and 2 N complex multiplications. One N-point FFT requires $(N/2)\log_2 N$ complex multiplications according to the Cooley–Tukey FFT algorithm [110]. Factor 4 in the expression refers to the fact that one complex multiplication can be expressed by four real ones. The term $N - N_D + 1$ refers to the number of useful samples due to the overlap-save algorithm for blockwise FD filtering per polarization [68]. When considering two polarizations, the useful samples for two blocks reduced from $2\,N$ to $2(N - N_D + 1)$. In general, the FFT size should be optimized to minimize the complexity.

The BOP of the CDC involves complex number multiplications, therefore, first, let's define the BOP of one complex multiplication. One complex multiplication includes 4 real multiplications and 2 real additions, giving the BOP:

$$\text{BOP}_{\text{complex mult}} = 4b_1 b_2 + 2(b_1 + b_2 + 1). \tag{57}$$

Secondly, with Cooley–Tukey FFT algorithm, one N-point FFT involves $(N/2)\log_2 N$ complex multiplications and $N\log_2 N$ complex additions. When doing the multiplications between the input sample and the W in the FFT, we assume that because $W = e^{j2\pi/N}$, representing the unit circle, it only changes the phase of the complex values. Therefore, the multiplication between the complex number and W does not change the resulting bitwidth, together with the fact that in real-world systems, there

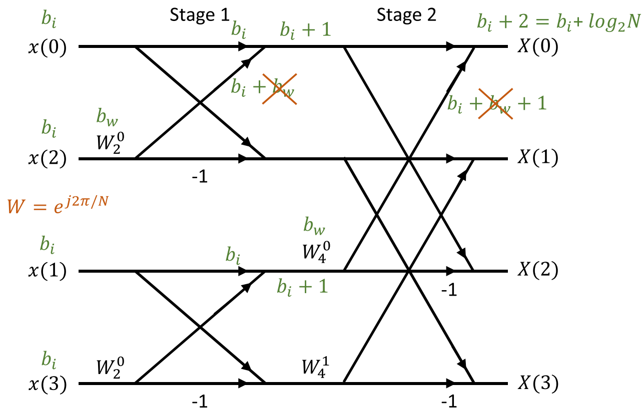


Fig. 6. Butterfly computation diagram of FFT, showing the bitwidth of each step.

is the quantization of the result back to the desired bitwidth. The resulting bitwidth of each stage of Cooley–Tukey FFT can be illustrated in the butterfly diagram in Fig 6. The BOP of one FFT can be formulated as follows:

$$\begin{aligned}\mathrm{BOP}_{\mathrm{FFT}} = &\frac{N}{2}\log_2 N[4b_i b_w + 2(b_i+1)]\\ &+ 2N\log_2 N[b_i + \log_2 N],\end{aligned}\tag{58}$$

where b_i is the bitwidth of the input sample and b_w is the bitwidth of W in the FFT.

Next, after the FFT, the resulting signal is multiplied by the transfer function of the filter, which contains N complex multiplications per polarization of the signals. The BOP of this operation is:

$$\begin{aligned}\mathrm{BOP}_{\mathrm{TF}} &= N \cdot \mathrm{BOP}_{\text{complex mult}}\\ &= N(4b_{\mathrm{FFT}}b_{\mathrm{TF}} + 2(b_{\mathrm{FFT}} + b_{\mathrm{TF}} + 1)),\end{aligned}\tag{59}$$

where b_{TF} is the bitwidth of the transfer function, and b_{FFT} is the bitwidth after the FFT. In conclusion, the BOP of the CDC contains 4 times of the BOP of the FFT and 2 times the BOP of transfer function multiplication: The NABS of the CDC can be formulated below:

$$\begin{aligned}\mathrm{BOP}_{\mathrm{CDC}} &= \frac{(4\cdot \mathrm{BOP}_{\mathrm{FFT}} + 2\cdot \mathrm{BOP}_{\mathrm{TF}})q}{2(N-N_D+1)}\\ &= \frac{2Nq}{(N-N_D+1)}\left[\log_2 N\left(2b_i b_w + (b_i+1)\right.\right.\\ &\quad \left. + 2(b_i + \log_2 N)\right)\\ &\quad \left. + 2b_{\mathrm{FFT}}b_{\mathrm{TF}} + (b_{\mathrm{FFT}} + b_{\mathrm{TF}} + 1)\right],\end{aligned}\tag{60}$$

The NABS of the CDC can be formulated below:

$$\begin{aligned}\mathrm{NABS}_{\mathrm{CDC}} = &\frac{2Nq}{(N-N_D+1)}\left[\log_2 N\left((2X_w+1)(b_i+1)\right.\right.\\ &\left. + 2(b_i + \log_2 N)\right)\\ &\left. + (2X_{\mathrm{TF}}+1)(b_{\mathrm{FFT}} + b_{\mathrm{TF}} + 1)\right],\end{aligned}\tag{61}$$

where X_{TF} represents the number of adders we need at most to perform the multiplication of the transfer function, see more explanation in Section IV-A.

J. Digital Back Propagation (DBP)

This DBP technique utilizes the symmetric split-step Fourier approach for its implementation. Here, the parameters of the linear filter and the nonlinear operators are fine-tuned to minimize the equalized BER. It was assumed that the nonlinear step would remain entirely static. When considering a single channel, the computational demand of the DBP method, measured by the necessary RM per transmitted symbol, can be approximated as [17], [109]:

$$C_{\text{DBP-1CH}} = 4qN_{\mathrm{Sp}}N_{\mathrm{StSP}}\left(\frac{N(\log_2\, N+1)}{N-N_{Dq}+1}+1\right)\tag{62}$$

where N_{Sp} refers to the total number of spans, N_{STpS} is the number of propagation steps per span, and q is the oversampling factor. The RM of DBP of each step consists of the linear part which is the RM of the CDC and the nonlinear part which is the addition of one RM refers to the multiplication of a nonlinear term.

The BOP of the DBP can be calculated, similarly to the RM of the CDC, as follows:

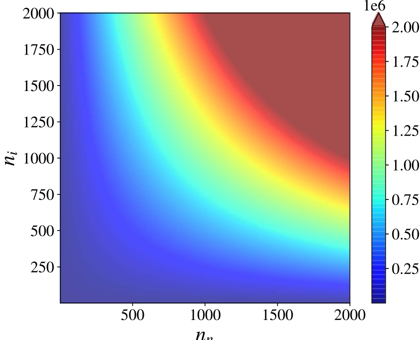
$$\begin{aligned}\mathrm{BOP}_{\mathrm{DBP}} = qN_{\mathrm{Sp}}N_{\mathrm{StSP}}&\left[\frac{2\,N}{(N-N_D+1)}\left(2b_{\mathrm{FFT}}b_{\mathrm{TF}}\right.\right.\\ &+ (b_{\mathrm{FFT}}b_{\mathrm{TF}}+1)\\ &\left. + \log_2\, N(2b_i b_w + (b_i+1) + 2(b_i + \log_2\, N)\right)\\ &\left. + 4b_{\mathrm{CDC}}b_{\mathrm{NL}} + 2(b_{\mathrm{CDC}}+1)\right],\end{aligned}\tag{63}$$

where b_i corresponds to the bitwidth of the input symbol, b_w is the bitwidth of the W value as described in the CDC. Here, we assume that multiplying the input with W does not change the resulting bitwidth as the W refers to the phase change as described in the CDC. b_{CDC} refers to the resulting bitwidth after the linear step (CDC), which usually is quantized to a certain number of bits. Lastly, b_textNL is the bitwidth of the nonlinear term that applies a nonlinear phase shift to the signal.

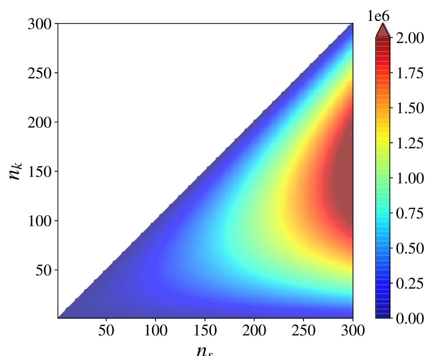
The NABS of DBP can be found as:

$$\begin{aligned}\mathrm{NABS}_{\mathrm{DBP}} = qN_{\mathrm{Sp}}N_{\mathrm{StSP}}&\left[\frac{2\,N}{N-N_D+1}\left((2X+1)(b_{\mathrm{FFT}}b_{\mathrm{TF}}+1)\right.\right.\\ &\left. + \log_2\, N((2X+1)(b_i+1) + 2(b_i + \log_2\, N)\right)\\ &\left. + (4X+2)(b_{\mathrm{CDC}}+1)\right].\end{aligned}\tag{64}$$

The multiplication is changed to the number of additions for the NABS.

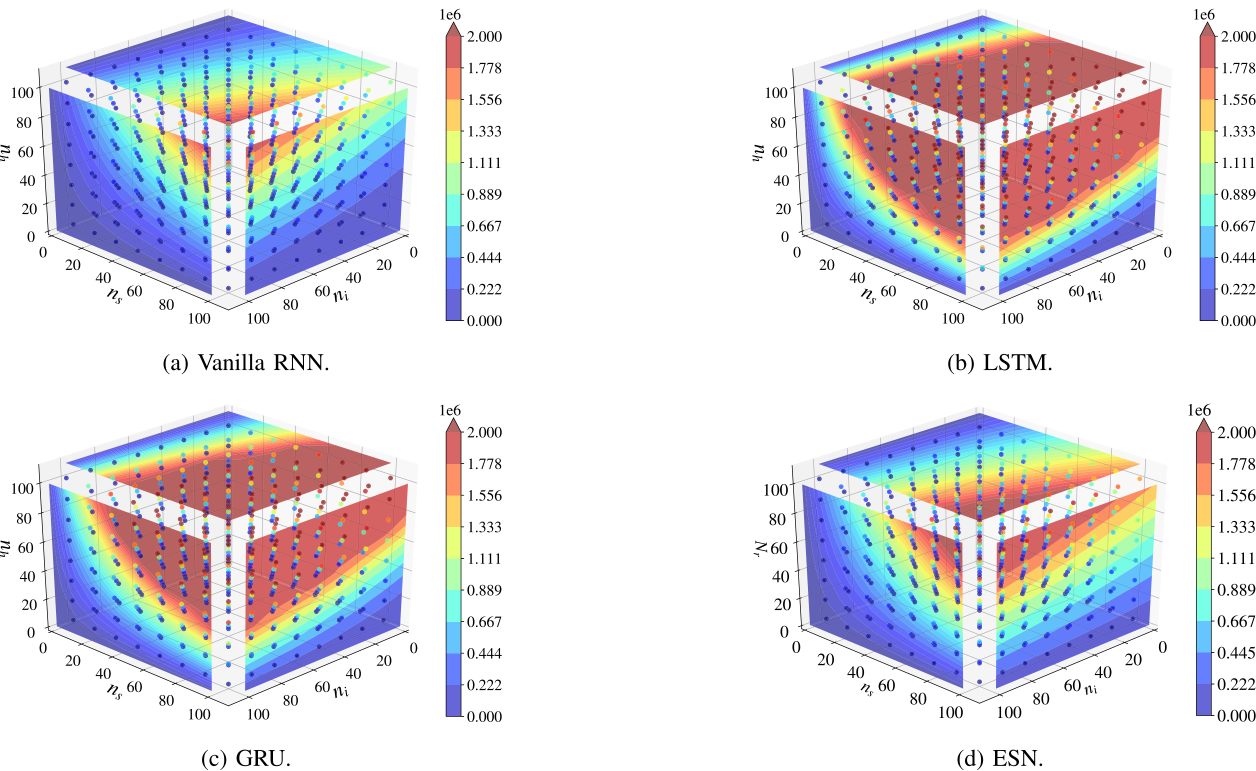


(a) Dense layer with parameters; number of features in the input vector n_i and number of neurons in the layer n_n.



(b) CNN with parameters; kernel size n_k and number of time steps n_s.

Fig. 7. Number of real multiplications (RM) of feed-forward layers.



(a) Vanilla RNN. (b) LSTM. (c) GRU. (d) ESN.

Fig. 8. Number of real multiplications (RM) of recurrent layers with respect to different values of the number of features in the input vector n_i, the number of time steps n_s and the number of hidden units n_h or N_r in ESN.

V. Comparative Analysis of the Complexities for Each NN Structure

The comparison of the complexity in terms of RM is illustrated in Fig. 7 for feed-forward NNs and in Fig. 8 for recurrent networks. In feed-forward networks, we first address the computational complexity of a dense layer. To reach over 2×10^6 real multiplications, which we use as a threshold (highlighted by a maroon color), we can have up to around 1500 input features (n_i) and 1500 neurons (n_n) which is a high value for a single dense layer. The $n_n n_i$ term in (3) forms a hyperbolic curve, as can be seen in Fig. 7(a). For the 1D-convolutional layer, as predicted by Eq. 11, we reach a high complexity (maroon) region using fewer input features than the dense layer case because now the complexity growth depends on more than 2 variables (e.g. to reach the complexity threshold, the number of time steps n_s can be set to 275 with the kernel size n_k equal to 150, and we fixed $n_i = 100$, $n_o = 1$, $padding = 0$, $dilation = 1$, and $stride = 1$). According to the exemplary chosen parameters above, the $n_k \leq n_s$ condition derived from (10) must be satisfied to obtain at least the output size equal to 1; therefore, in Fig. 7(b), the white region corresponds to unavailable output

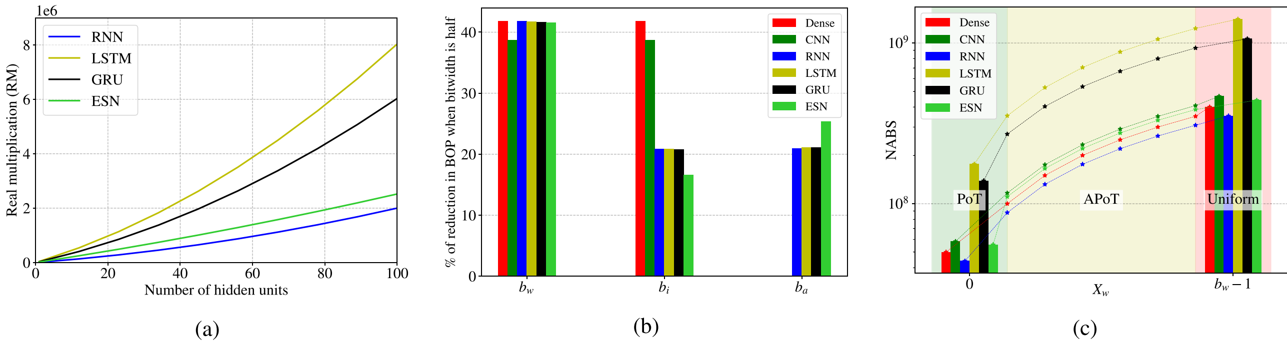


Fig. 9. (a) Complexity comparison between recurrent-based networks when $n_s = 100$, $n_i = 100$ for all networks and $n_o = 100$, $s_p = 0.5$ for ESN; (b) Reduction of BOP in percentage when reducing bitwidth of each parameter; weight bitwidth b_w, input bitwidth b_i, and activation bitwidth b_a by half or from 8 bits to 4 bits; (c) Comparison of the number of additions and bit shifts (NABS) with different quantization techniques and different network types, assume $b_w = 8$. Note that X_w is the number of adders required to represent a multiplier.

and the heat-map has a hyperbolic form because only n_s and n_k parameters vary and the other parameters are kept constant.

The RNN-based networks apparently have higher complexity than the feed-forward NNs. The vanilla RNN in Fig. 8(a) shows the least complexity among the RNN-based networks studied, while the LSTM's complexity growth is the fastest, which can be seen from the size of the maroon area in Fig. 8(b). The GRU in Fig. 8(c) shows slightly lower complexity than the LSTM because it has a lower number of gates in its architecture. If we look at the equations of GRU and LSTM in Table III, the LSTM has a multiplier of 4 for the n_i and n_h, whereas for the GRU the multiplier is 3. The ESN with fixed $n_o = 100$ and $s_p = 0.5$ in Fig. 8(d) has higher complexity than the vanilla RNN, but less complexity than the GRU, because the ESN by design has a less complex architecture due to the use of the reservoir [111]. For all RNN-based networks, we readily infer that the number of hidden units n_h, or N_r for the ESN, plays the most crucial role in defining the layer's computational complexity in terms of RM metric. In Fig. 8, we observe that the top face of all cubes which corresponds to the highest number of n_h ($n_h = 100$) has the largest maroon areas. In terms of the effect on the complexity behavior, the second most important quantity is the number of time steps n_s; we can see in Fig. 8 that the right face of the cubes, referring to the highest number of n_s ($n_s = 100$), has the second-largest maroon areas. Finally, the dimensions of the input vector n_i have the least impact on the RM, as shown by the left face ($n_i = 100$) of all cubes in Fig. 8 with the smallest maroon areas compared to other faces. See (3), (11), (15), (19), (23), and (29) for the exact dependencies.

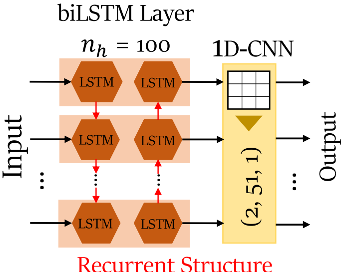
Furthermore, to highlight the computational complexity trend over those different recurrent layers, we plotted the RM versus the number of hidden units (n_h for vanilla RNN, LSTM, and GRU, or N_r in ESN) in a scenario where all other hyperparameters are constant. Fig. 9(a) depicts the result of this analysis when $n_s = 100$, $n_i = 100$ for all networks, and $n_o = 100$, $s_p = 0.5$ for the ESN. By considering those parameters as fixed, the complexity of all four recurrent layers scales quadratically with the number of hidden units (n_h^2), which is traditionally interpreted as having the same $O(n^2)$. However, Fig. 9(a) brings an important fact that there are significant differences in the computational complexity in terms of RM between all four recurrent layers. Ultimately, this means that the Big-O notation is not sensitive enough to assess the complexity of the NNs in digital signal processing. We can spot that the LSTM complexity escalates the fastest followed by the GRU, ESN, and RNN, respectively. These differences result mainly from the scaling terms on the n_h^2 of each RM complexity expression for these layers. Note that for the ESN ((29)), the complexity increases more steadily, as far as N_r is multiplied by the sparsity parameter s_p. Moreover, as noted in Section IV-F, the reservoir can be implemented in the optical domain, so the complexity can be reduced further at the expense of performance trade-off.

We notice that not only the hyper-parameters like n_h and n_s, affect the computational complexity, but also the bitwidth or precision of each parameter can impact the complexity when we quantify it in terms of the BOP. The effects produced by the bitwidth value of weight (b_w), input (b_i), and activation (b_a) are examined in this study. The full-precision or 32-bit precision can be considered over-redundant because 8-bit or less is often enough to provide comparable performance, as stated by many works in the field [112], [113], [114], [115]. Here, we did not provide the study of the BOP versus different hyper-parameters and bitwidth, because we believe that this is a straightforward analysis that approximately follows what we already studied with the RM metric. Instead, we focus on answering the following question: which variable bitwidth (b_w, b_i, or b_a) produces the highest saving in the BOP complexity when its precision is reduced? This question can guide the design of a low-complexity NN structure by identifying which parameter of the NN is the key to producing the higher saving in complexity. To address this question, we compare the reduction of the BOP when using 8-bit precision versus the 4-bit precision for each parameter (b_w, b_i, and b_a) in different network types; the results of the comparison are shown in Fig. 9(b). The bitwidth of the weight matrix b_w is the most significant parameter to consider when trying to reduce the layer's complexity, as the BOP is decreased by around 40% for all network types when we halved the precision of b_w. For the dense and 1D-convolutional layers, the precision for input b_i is as important as the b_w, while reducing the bitwidth of the bias vector b_a does not have a noticeable impact on the

BOP. In the RNN-based networks, converting from 8-bit to 4-bit precision for b_i and b_a shows a nearly equivalent reduction in the BOP, except for the ESN case, where decreasing the b_a precision results in more reduction in the BOP than when we reduce the b_i precision.

Lastly, we analyze the NABS metric considering various quantization techniques: uniform, PoT, and APoT quantization, as described in Section IV-A. Note that each technique needs a different number of shifts and adders to perform the multiplication. As mentioned before, the shifts incur no extra cost in hardware implementation; therefore, we focus on evaluating the NABS metric of each layer for certain quantization techniques versus the number of adders required at most to perform the multiplication, denoting it as X. More specifically, if the weight matrix has b_w as its bitwidth and the uniform quantization is utilized, the number of adders required at most (X_w) is equal $b_w - 1$. In the case of PoT, $X_w = 0$, and for APoT, X_w varies between 1 and $b_w - 2$, in this case. Fig. 9(c) shows the NABS versus X_w analysis when considering that: $b_w, b_i, b_a = 8$ for all networks, $n_i = 1000$ and $n_n = 2000$ for a dense layer, $n_i = 100, n_s = 300, n_o = 1$, padding $= 0$, dilation $= 1$, stride $= 1$ and $n_k = 100$ for a 1D-convolutional layer, $n_i = 100, n_s = 100$ and $n_h = 100$ for all RNN-based networks, and $n_o = 100$, $s_p = 0.5$ for the ESN.

As shown in Fig. 9(c), for all types of networks, when the PoT quantization is used, the NABS can drop around 8 times lower compared to the NABS when using the uniform quantization. Since APoT is a quantization scheme represented by a sum of PoT terms, APoT provides a smooth transition between PoT and uniform quantization. In various works, PoT was claimed to have very low complexity because the multiplications are replaced by just shifts [94], [116], [117]. However, when we consider that the multiplication in the uniform quantization can be represented by shifts and adders, and we have a fair metric like NABS to compare between different quantization techniques, the NABS when applying PoT is only around an order of magnitude lower than the NABS when using the uniform quantization. To be more specific, even though PoT converts all multipliers into bit shifters, we still have a number of adders coming from the sum operations that are not related to the multipliers, but they are key for the operational structure of the NN layers. Therefore, the NABS metric can provide a reliable assessment of the computational complexity of NNs before their implementation in hardware, where the NLG will be the ultimate metric. Note that we intentionally increased the values of the hyper-parameters for the feed-forward NNs in order to compare them in the same graph as the RNN-based networks. For the complexity with regard to NABS, the LSTM apparently needs the highest number of shifts and adders. In conclusion, the three matrices of complexity: the RM, the BOP, and the NABS, have the same trend, meaning that the LSTM requires the most computational resources, followed by the GRU. However, the complexity depends on the particular NN design and can be reduced if we can tolerate a more accurate trade-off: varying the values of hyper-parameters can affect the accuracy, but, simultaneously, work in favor of reducing the computational complexity.



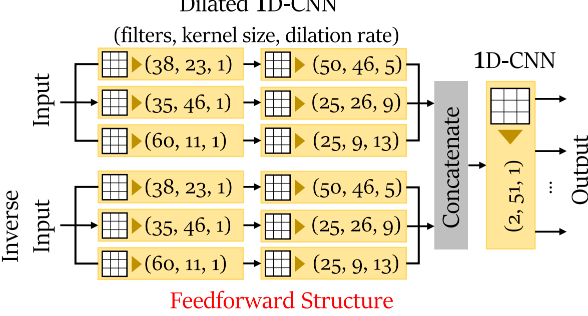


Fig. 10. NN-based equalizer structure considered: a) biLSTM+CNN and b) 1D-CNN models.

VI. A Practical Study on the Benefits of Complexity Reduction in NN-Based Equalizer

This section presents a practical study of the impact of the complexity reduction techniques on the optical performance of a numerically simulated single 64 QAM 30 GBd dual-polarization channel transmitted along 20×50 km of SSMF. The NN structures considered in this paper are a biLSTM+CNN model and a 1D-CNN model. Both models take an input window of 221 symbols to recover 171 output symbols. The biLSTM model, shown in Fig. 10, contains 100 hidden units, as explained in [15]. For the 1D-CNN model, the dilated CNN was applied, as explained in detail in Ref. [118], and the NN parameters are presented in Fig. 10(b).

A. Analysis of Performance of Quantization Techniques

We present an example of how such metrics can be powerful in reducing the computational complexity of NN-based equalizers when compared with traditional DSP equalizers such as the CDC block and the DBP method. In this analysis, the aforementioned biLSTM+CNN model was evaluated. The launch power was set to 2 dBm, which leads to the highest Q-factor after nonlinear equalization.

In this study, the performance of NN is evaluated when quantization-aware training (QAT) is implemented to determine whether training can mitigate the error introduced by the low bit precision of the NN weights. Fig. 11 summarizes the Q-factor as a function of bitwidth, considering quantization with different schemes: Uniform [119], Power-of-Two (PoT) [120] and Additive Power-of-Two (APoT) with 2 terms [121] and

TABLE IV
SUMMARY OF THE PERFORMANCE VERSUS COMPLEXITY OF TWO NN-BASED EQUALIZERS AFTER APPLYING KD (BILSTM+CNN AS A TEACHER MODEL AND 1D-CNN AS A STUDENT MODEL), WHERE THE BITWIDTH $B_i = 64$, $B_w = 32$, AND $B_a = 32$

Model Type	Q-factor	No. of Trainable Parameters	RMpS	BOPpS	NABSpS	CPU Inference Time per Window	GPU Inference Time per Window
biLSTM+CNN (Teacher)	**10.66**	**104,407**	$\mathbf{1.28}\times10^5$	$\mathbf{1.66}\times10^8$	$\mathbf{3.19}\times10^8$	5.78×10^{-3}	7.65×10^{-3}
1D-CNN (Student)	10.19	293,390	3.72×10^5	8×10^8	1.27×10^9	$\mathbf{5.12}\times10^{-3}$	$\mathbf{3.87}\times10^{-4}$

The RM, BOP and NABS are reported "per equalized symbol".

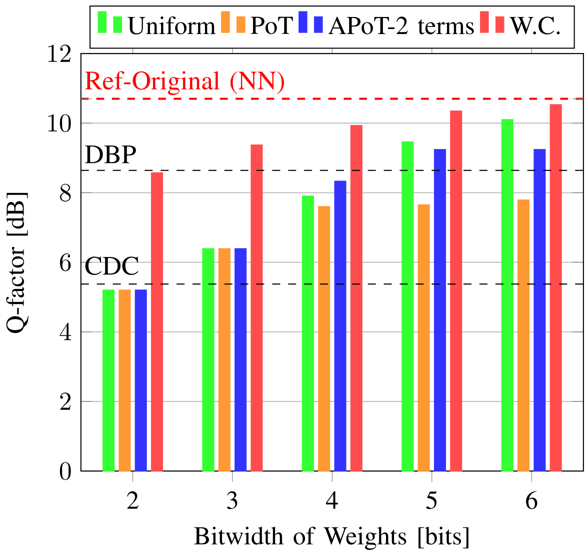


Fig. 11. Q-factor vs. bit-width when employing Uniform Quantization, PoT, APoT with two adders, and W.C. The model w/o quantization is shown as a benchmark together with the CDC and 1 StPS DBP, at 2 dBm launch power.

W.C. [122].[7] The original model without quantization, the standard 1 Step-per-span (StPS) digital backpropagation (DBP), and Chromatic Dispersion Compensation (CDC) are used as a benchmark. As expected, as the complexity is reduced (lower precision/bit-width), the NN tends to perform worse in all the complexity reduction schemes considered.

The analysis of Fig. 11 shows that the W.C. approach outperforms the other quantization techniques due to its ability to learn the optimal alphabet for each part of the NN structure instead of using a static quantized alphabet like uniform, PoT, and APoT methods. Consequently, the performance of the equalizers is less affected. Notably, with a 6-bit alphabet size of 2^6 clusters, the W.C. technique achieves similar performance to the original model, while a 2-bit alphabet size performs similarly to the 1 StPS DBP. Regarding the trade-off between optical performance and computational complexity, the APoT approach with two terms may be a feasible option, as it only requires one adder for each multiplication, as detailed in [50].

We note that the use of QAT can result in an unstable training process,[8] which requires continuous training monitoring. Moreover, when considering smaller bit levels, we recommend implementing a gradual quantization approach in which the precision is gradually decreased during the training process while optimizing the learning rate and batch size.

B. Complexity Comparison Between Two NN-Based Equalizers

In this section, we evaluate the performance versus computational complexity of the NN-based equalizers after applying KD framework to train the 1D-CNN model (Student) with the knowledge of the biLSTM+CNN (Teacher) as in Ref. [118]. The computational complexity in terms of the number of trainable parameters, RM, BOPs, NABS, and inference latency are compared. As mentioned earlier, KD in general context is used to reduce the computational complexity in terms of the NN parameters, however, in Ref. [118], the KD framework is used to recast the NN structure from biLSTM-based (recurrent-based) to a feedforward-based equalizer to allow parallelization in processing. In this case, KD was applied to focus on enabling parallelization and reducing the inference latency rather than reducing the RM, BOPs, or NABS.

Table IV shows the summary of the performance versus complexity of these two NN architectures when the precision is defaulted from Tensorflow. The bitwidth of the input (b_i) is 64 bits, and the bitwidth of the weights (b_w) and activation function (b_a) is 32 bits. It can be seen that the Q-factor of the student model is slightly lower than that of the teacher model when trained with the simulated data mentioned above.[9] Even though the number of trainable parameters, RM, BOPs, and NABS of the student model are also higher than the teacher model, the inference latency of the student is actually lower. The inference time analysis was carried out by using CPU (Intel Xeon Processor 2.20 GHz) and GPU (Tesla T4) on Google Colab [123].[10] This highlights the importance of the "parallelization" strategy, which helps reduce the complexity of the processing at the hardware synthesis level. This is crucial for real-world implementation. For the training phase in this case, the trainable parameters can indicate the complexity to some extent; however, empirically, the biLSTM+CNN has a longer training time, even though the number of trainable parameters is lower than the 1D-CNN model. This occurs because the recurrent structure of the biLSTM prevents the computation from being fully parallelizable. At each time step of the calculation, the recurrent structure takes

[7]We assume that the weights only are quantized while the inputs are still considered to be float32.

[8]The training phase of a quantized model may encounter obstacles associated with learning, such as the exploration versus exploitation trade-off, hyperparameter sensitivity, loss leading to NaN, or gradient problems.

[9]Note that in the experiment, this KD approach did not show the performance degradation in the student model [118].

[10]Note that, in this study, we did not consider the GPU-accelerated library of primitives for deep neural networks cuDNN (NVIDIA CUDA

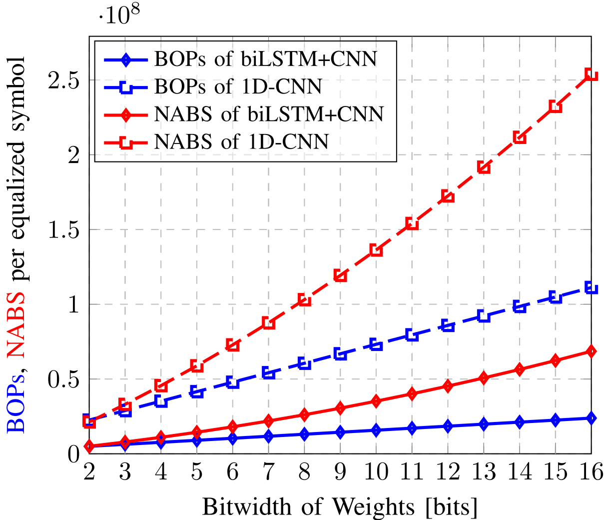


Fig. 12. BOPs and NABS per equalized symbol as a function of bitwidth of weights of biLSTM+CNN and 1D-CNN models.

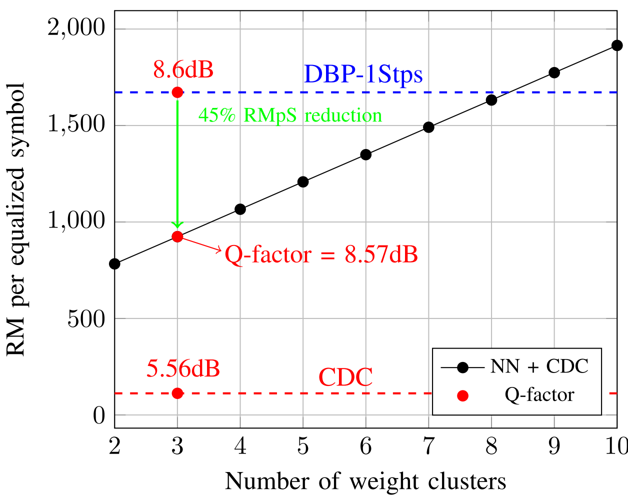


Fig. 13. Comparison of complexity (RMpS) between bLSTM+CNN and traditional channel equalizers (DBP 1STpS and CDC) as a function of the number of clusters utilized in weight clustered compression. The Q-factor in dB is emphasized for the scenario involving 3 clusters.

into account the output of the previous time step. This sequential nature makes the training and inference longer.

Fig. 12 shows BOPs and NABs as a function of the bitwidth of the weights (b_w) of the NN. It can be observed that the NABS grows with a steeper slope than the BOPs when the b_w increases. This fact highlights that towards the implementation of resource-constrained devices or hardware accelerators, both metrics should be considered carefully, because if only BOPs is assessed at higher precision of the weights, while BOPs fits the requirement, NABS which escalates faster might exceed the requirement of the implementation.

Finally, it is imperative to address whether the current set of complexity reduction techniques suffices to render NN-based equalizers an appealing choice for industrial implementation. To explore this, we conducted a comparative analysis of the RMpS between traditional DBP 1STpS and CDC implemented in the frequency domain. The results are depicted in Fig. 13.

To maintain complexity constraints, we employed the same teacher NN architectural with an adaptation involving 50 hidden units in the LSTM layer. This adjustment yielded a Q-factor performance of 8.57 dB, comparable to DBP 1STpS (8.6 dB), and 3 dB higher than CDC performance (5.56 dB) for the same experiment previously discussed.

In terms of RMpS, while achieving similar performance levels, the complexity of the NN-equalizer + CDC was reduced by 45% compared to DBP 1STpS when considering 3 weight clusters. However, upon comparing the complexity of the NN equalizer with CDC, it becomes evident that further strides are necessary to achieve lower complexity levels, particularly in terms of multiplication operations. Notably, while CDC typically requires fewer than 100 multiplications per recovered symbol, the NN, even after compression, necessitated between 500-1000 multipliers in its lower complexity configurations. In contrast, the DBP typically demands more than 1500 multipliers per recovered symbol. This underscores the ongoing challenge to optimize the NN-equalizer for reduced complexity, especially in the realm of multiplicative operations, and further investigations are still needed.

VII. Conclusion

In this study, we explore various methods for creating neural network equalizers with lower complexity in the training, inference, and hardware synthesis stages. To gauge the effectiveness of these techniques, we introduce primary metrics for evaluating NN complexity during both training and inference. During training, factors such as the number of trainable parameters, total training time, the product of epochs and batches, data parallelism, and the number of operational ranges should be considered. Moving to the inference phase, which is the heart of the real-time operation, we proposed a detailed breakdown of computational complexity into three hardware-agnostic measures: number of real multiplications (RM), number of bit operations (BOP), and number of additions and bit shifts (NABS). This granular approach provides a clear picture of complexity as we transition from software to hardware levels. Notably, the fourth metric, the number of logic gates, is hardware-dependent and requires specific setup information for calculation. The introduction of these detailed metrics allows us to establish a consistent baseline for complexity calculation, catering to varied purposes. We investigate the computation of RM, BOP, and NABS across various NN layers, including dense layers, 1D-convolutional layers, vanilla RNNs, LSTMs, GRUs, and ESN architectures in a general form.

Our evaluation of the RM metric shows how complexity evolves with changes in different hyperparameters for each NN layer. Notably, LSTM exhibits the highest complexity among recurrent layers due to its architecture featuring different gates, with complexity escalating significantly with increased hidden units. Vanilla RNN emerges as the least complex recurrent architecture. For all recurrent networks, the most impactful hyperparameter on complexity is the number of hidden units,

followed by the number of time steps, while the size of the input vector has the least influence. The dense layer stands out as the most economical in complexity due to its simple matrix multiplication.

We also emphasize the importance of bitwidth (precision) in defining BOP complexity as we move closer to the hardware level. A two-fold reduction in bitwidth, particularly in weights, drastically reduces BOP by around 40% across all types of NNs. This underscores the recommendation to prioritize low-precision bits in NN weights for a more significant reduction in complexity.

Additionally, we introduce the NABS metric to highlight the effects of different quantization techniques (uniform, PoT, and APoT). Unlike other studies claiming drastic complexity reduction with PoT, our work using NABS shows that the true complexity reduction with PoT is only around one order of magnitude compared to uniform quantization for all NN layers. We argue that NABS is a more accurate metric for identifying true complexity levels compared to previously used metrics like RM or BOP.

Finally, a practical study investigates the impact of complexity reduction methods on equalization performance. Our findings demonstrate that implementing these strategies results in NN models with reduced complexity while maintaining a Q-factor level similar to the original counterparts, showcasing the feasibility of complexity reduction for practical applications.

References

[1] A. Miller et al., "Review of neural network applications in medical imaging and signal processing," *Med. Biol. Eng. Comput.*, vol. 30, no. 5, pp. 449–464, 1992.

[2] L. A. Feldkamp and G. V. Puskorius, "A signal processing framework based on dynamic neural networks with application to problems in adaptation, filtering, and classification," *Proc. IEEE*, vol. 86, no. 11, pp. 2259–2277, Nov. 1998.

[3] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman, "1D convolutional neural networks and applications: A survey," *Mech. Syst. Signal Process.*, vol. 151, 2021, Art. no. 107398.

[4] H. Dahrouj et al., "An overview of machine learning-based techniques for solving optimization problems in communications and signal processing," *IEEE Access*, vol. 9, pp. 74908–74938, 2021.

[5] S.-i. Amari and A. Cichocki, "Adaptive blind signal processing-neural network approaches," *Proc. IEEE*, vol. 86, no. 10, pp. 2026–2048, Oct. 1998.

[6] K. Burse, R. N. Yadav, and S. C. Shrivastava, "Channel equalization using neural networks: A review," *IEEE Trans. Syst., Man, Cybern., Part C. (Appl. Rev.)*, vol. 40, no. 3, pp. 352–357, May 2010.

[7] M. Kahrs and K. Brandenburg, *Applications of Digital Signal Processing to Audio and Acoustics*. Berlin, Germany: Springer, 1998.

[8] R. Govil, "Neural networks in signal processing," in *Fuzzy Systems and Soft Computing in Nuclear Engineering* (Studies in Fuzziness and Soft Computing), vol. 38, D. Ruan, Eds. Physica, Heidelberg, 2000, doi: 10.1007/978-3-7908-1866-6_11.

[9] B. Lusch, J. N. Kutz, and S. L. Brunton, "Deep learning for universal linear embeddings of nonlinear dynamics," *Nature Commun.*, vol. 9, no. 4950, 2018, doi: 10.1038/s41467-018-07210-0.

[10] H. Huttunen, "Deep neural networks: A signal processing perspective," in *Handbook of Signal Processing Systems*. Berlin, Germany: Springer, 2019, pp. 133–163.

[11] M. A. Jarajreh et al., "Artificial neural network nonlinear equalizer for coherent optical OFDM," *IEEE Photon. Technol. Lett.*, vol. 27, no. 4, pp. 387–390, Feb. 2015.

[12] F.-L. Luo and R. Unbehauen, *Applied Neural Networks for Signal Processing*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[13] M. Ibnkahla, "Applications of neural networks to digital communications–a survey," *Signal Process.*, vol. 80, no. 7, pp. 1185–1215, 2000.

[14] D. Zibar, M. Piels, R. Jones, and C. G. Schäeffer, "Machine learning techniques in optical communication," *J. Lightw. Technol.*, vol. 34, no. 6, pp. 1442–1452, Mar. 2016.

[15] P. J. Freire et al., "Reducing computational complexity of neural networks in optical channel equalization: From concepts to implementation," *J. Lightw. Technol.*, vol. 41, no. 14, pp. 4557–4581, Jul. 2023.

[16] M. Schädler, G. Böcherer, and S. Pachnicke, "Soft-demapping for short reach optical communication: A comparison of deep neural networks and volterra series," *J. Lightw. Technol.*, vol. 39, no. 10, pp. 3095–3105, May 2021.

[17] A. Napoli et al., "Reduced complexity digital back-propagation methods for optical communication systems," *J. Lightw. Technol.*, vol. 32, no. 7, pp. 1351–1362, Apr. 2014.

[18] D. Malkoff, "A neural network for real-time signal processing," in *Proc. 2nd Int. Conf. Neural Inform. Process. Syst.*, Cambridge, MA, USA: MIT Press, 1989, pp. 248–255.

[19] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[20] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," 2016, *arXiv:1604.03168*.

[21] B. Li and T. N. Sainath, "Reducing the computational complexity of two-dimensional LSTMs," in *Proc. Interspeech*, 2017, pp. 964–968.

[22] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5687–5695.

[23] J. L. Balcázar, R. Gavalda, and H. T. Siegelmann, "Computational power of neural networks: A characterization in terms of Kolmogorov complexity," *IEEE Trans. Inf. Theory*, vol. 43, no. 4, pp. 1175–1183, Jul. 1997.

[24] M. V. Baalen et al., "Bayesian bits: Unifying quantization and pruning," *Adv. Neural Inf. Process. Syst.*, 2020, vol. 33, pp. 5741–5752.

[25] C. Baskin et al., "Uniq: Uniform noise injection for non-uniform quantization of neural networks," *ACM Trans. Comput. Syst.*, vol. 37, no. 1–4, pp. 1–15, 2021.

[26] S. Deligiannidis, C. Mesaritakis, and A. Bogris, "Performance and complexity analysis of bi-directional recurrent neural network models versus volterra nonlinear equalizers in digital coherent systems," *J. Lightw. Technol.*, vol. 39, no. 18, no. 18, pp. 5791–5798, Sep. 2021.

[27] O. Sidelnikov, A. Redyuk, and S. Sygletos, "Equalization performance and complexity analysis of dynamic deep neural networks in long haul transmission systems," *Opt. Exp.*, vol. 26, no. 25, pp. 32765–32776, 2018.

[28] P. J. Freire et al., "Performance versus complexity study of neural network equalizers in coherent optical systems," *J. Lightw. Technol.*, vol. 39, no. 19, pp. 6085–6096, Oct. 2021.

[29] W. Maass, "On the computational complexity of networks of spiking neurons," in *Proc. 7th Int. Conf. Neural Inform. Process. Syst.*, Denver, Colorado, 1994, pp. 183–190.

[30] R. Alizadeh, J. K. Allen, and F. Mistree, "Managing computational complexity using surrogate models: A critical review," *Res. Eng. Des.*, vol. 31, no. 3, pp. 275–298, 2020.

[31] S. Wiedemann, K.-R. Müller, and W. Samek, "Compact and computationally efficient representation of deep neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 3, pp. 772–785, Mar. 2020.

[32] A. M. Amin, R. R. Mahmood, and A. I. Khan, "Analysis of pattern recognition algorithms using associative memory approach: A comparative study between the hopfield network and distributed hierarchical graph neuron (dhgn)," in *Proc. IEEE 8th Int. Conf. Comput. Inf. Technol. Workshops*, 2008, pp. 153–158.

[33] S. N. Kerr, "A big-O experiment: Which function is it?," in *Proc. 43rd Annu. Southeast Regional Conf.-Volume 1*, 2005, pp. 317–318.

[34] V. D. Blondel and J. N. Tsitsiklis, "A survey of computational complexity results in systems and control," *Automatica*, vol. 36, no. 9, pp. 1249–1274, 2000.

[35] N. M. Nawi, W. H. Atomi, and M. Z. Rehman, "The effect of data preprocessing on optimized training of artificial neural networks," *Procedia Technol.*, vol. 11, pp. 32–39, 2013.

[36] Z. Xu, C. Sun, T. Ji, H. Ji, and W. Shieh, "Transfer learning aided neural networks for nonlinear equalization in short-reach direct detection systems," in *Proc. IEEE Opt. Fiber Commun. Conf. Exhib.*, 2020, pp. 1–3.

[37] P. J. Freire et al., "Transfer learning for neural networks-based equalizers in coherent optical systems," *J. Lightw. Technol.*, vol. 39, no. 21, pp. 6733–6745, Nov. 2021.

[38] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2017, pp. 23–30.

[39] P. J. Freire et al., "Domain adaptation: The key enabler of neural network equalizers in coherent optical systems," in *Proc. Opt. Fiber Commun. Conf.*, 2022, Paper Th2A–35.

[40] Q. Zhou, F. Zhang, and C. Yang, "AdaNN: Adaptive neural network-based equalizer via online semi-supervised learning," *J. Lightw. Technol.*, vol. 38, no. 16, pp. 4315–4324, Aug. 2020.

[41] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1126–1135.

[42] B. Liu, C. Bluemm, S. Calabrò, B. Li, and U. Schlichtmann, "Area-efficient neural network CD equalizer for 4× 200Gb/s PAM4 CWDM4 systems," in *Proc. IEEE Opt. Fiber Commun. Conf. Exhib.*, 2023, pp. 1–3.

[43] S. Srivallapanondh et al., "Multi-task learning to enhance generalizability of neural network equalizers in coherent optical systems," in *Proc. Eur. Conf. Opt. Commun.*, 2023, pp. 1–4.

[44] K. Alomar, H. I. Aysel, and X. Cai, "Data augmentation in classification and segmentation: A survey and new strategies," *J. Imag.*, vol. 9, no. 2, 2023, Art. no. 46.

[45] V. Neskorniuk et al., "Simplifying the supervised learning of Kerr nonlinearity compensation algorithms by data augmentation," in *Proc. Eur. Conf. Opt. Commun.*, 2020, pp. 1–4.

[46] S. Velliangiri et al., "A review of dimensionality reduction techniques for efficient computation," *Procedia Comput. Sci.*, vol. 165, pp. 104–111, 2019.

[47] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (PCA)," *Comput. Geosci.*, vol. 19, no. 3, pp. 303–342, 1993.

[48] L. Ge, W. Zhang, C. Liang, and Z. He, "Compressed neural network equalization based on iterative pruning algorithm for 112-Gbps VCSEL-enabled optical interconnects," *J. Lightw. Technol.*, vol. 38, no. 6, pp. 1323–1329, Mar. 2020.

[49] D. A. Ron, P. J. Freire, J. E. Prilepsky, M. Kamalian-Kopae, A. Napoli, and S. K. Turitsyn, "Experimental implementation of a neural network optical channel equalizer in restricted hardware using pruning and quantization," *Sci. Rep.*, vol. 12, no. 1, 2022, Art. no. 8713.

[50] T. Koike-Akino, Y. Wang, K. Kojima, K. Parsons, and T. Yoshida, "Zero-multiplier sparse DNN equalization for fiber-optic QAM systems with probabilistic amplitude shaping," in *Proc. IEEE Eur. Conf. Opt. Commun.*, 2021, pp. 1–4.

[51] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, 2009.

[52] C. Hager, H. D. Pfister, R. M. Butler, G. Liga, and A. Alvarado, "Revisiting multi-step nonlinearity compensation with machine learning," in *Proc. 45th Eur. Conf. Opt. Commun.*, 2019, pp. 1–4.

[53] N. Gautam, V. Kaushik, A. Choudhary, and B. Lall, "Optidistillnet: Learning nonlinear pulse propagation using the student-teacher model," *Opt. Exp.*, vol. 30, no. 23, pp. 42430–42439, 2022.

[54] S. Srivallapanondh et al., "Knowledge distillation applied to optical channel equalization: Solving the parallelization problem of recurrent connection," in *Proc. Opt. Fiber Commun. Conf.*, 2023, pp. 1–3.

[55] I. Taras and D. M. Stuart, "Quantization error as a metric for dynamic precision scaling in neural net training," 2018, *arXiv:1801.08621*.

[56] P. J. Freire et al., "Implementing neural network-based equalizers in a coherent optical transmission system using field-programmable gate arrays," *J. Lightw. Technol.*, vol. 41, no. 12, pp. 3797–3815, Jun. 2023.

[57] M. E. Nojehdeh, L. Aksoy, and M. Altun, "Efficient hardware implementation of artificial neural networks using approximate multiply-accumulate blocks," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2020, pp. 96–101.

[58] X. Huang, D. Zhang, X. Hu, C. Ye, and K. Zhang, "Recurrent neural network based equalizer with embedded parallelization for 100Gbps/λ PON," in *Proc. IEEE Opt. Fiber Commun. Conf. Exhib.*, 2021, pp. 1–3.

[59] D. Nichols, S. Singh, S.-H. Lin, and A. Bhatele, "A survey and empirical evaluation of parallel deep learning frameworks," 2021, *arXiv:2111.04949*.

[60] G. Armeniakos, G. Zervakis, D. Soudris, and J. Henkel, "Hardware approximate techniques for deep neural network accelerators: A survey," *ACM Comput. Surv.*, vol. 55, no. 4, pp. 1–36, 2022.

[61] L. D. Toffola, M. Pradel, and T. R. Gross, "Performance problems you can fix: A dynamic analysis of memoization opportunities," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 607–622, 2015.

[62] M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig, "Deep learning with dynamic computation graphs," 2017, *arXiv:1702.02181*.

[63] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. 33rd Int. Conf. Neural Inform. Process. Syst.*, Red Hook, NY, USA: Curran Associates Inc., 2019, Art. no. 10.

[64] B. Cannas, A. Fanni, L. See, and G. Sias, "Data preprocessing for river flow forecasting using neural networks: Wavelet transforms and data partitioning," *Phys. Chem. Earth, Parts A/B/C*, vol. 31, no. 18, pp. 1164–1171, 2006.

[65] A. Ghis, K. Smiri, and A. Jemai, "Mixed software/hardware based neural network learning acceleration," in *Proc. ICSOFT*, 2021, pp. 417–425.

[66] P. J. Freire, A. Napoli, B. Spinnler, N. Costa, S. K. Turitsyn, and J. E. Prilepsky, "Neural networks-based equalizers for coherent optical transmission: Caveats and pitfalls," *IEEE J. Sel. Topics Quantum Electron.*, vol. 28, no. 4, Jul./Aug. 2022, Art. no. 7600223.

[67] E. Jacobsen and P. Kootsookos, "Fast, accurate frequency estimators [DSP tips & tricks]," *IEEE Signal Process. Mag.*, vol. 24, no. 3, pp. 123–125, May 2007.

[68] B. Spinnler, "Equalizer design and complexity for digital coherent receivers," *IEEE J. Sel. Topics Quantum Electron.*, vol. 16, no. 5, pp. 1180–1192, Sep./Oct. 2010.

[69] S. Mirzaei, A. Hosangadi, and R. Kastner, "FPGA implementation of high speed fir filters using add and shift method," in *Proc. IEEE Int. Conf. Comput. Des.*, 2006, pp. 308–313.

[70] S. Jahani, "ZOT-MK: A new algorithm for big integer multiplication," MSc, Dept. Comput. Sci., Universiti Sains Malaysia, Malaysia, 2009.

[71] B. Hawks, J. Duarte, N. J. Fraser, A. Pappalardo, N. Tran, and Y. Umuroglu, "PS and QS: Quantization-aware pruning for efficient low latency neural network inference," 2021, *arXiv:2102.11289*.

[72] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 5363–5372.

[73] X. Staff, "Gate count capacity metrics for FPGAS," Xilinx Corp., San Jose, CA, Application Note XAPP, vol. 59, 1997.

[74] Y. Li, X. Dong, and W. Wang, "Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks," 2019, *arXiv:1909.13144*.

[75] T. Koike-Akino, Y. Wang, K. Kojima, K. Parsons, and T. Yoshida, "Zero-multiplier sparse DNN equalization for fiber-optic QAM systems with probabilistic amplitude shaping," in *Proc. IEEE Eur. Conf. Opt. Commun.* 2021, pp. 1–4.

[76] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, and J. Y. Li, "DeepShift: Towards multiplication-less neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2021, pp. 2359–2368.

[77] H. You et al., "Shiftaddnet: A hardware-inspired deep network," 2020, *arXiv:2010.12785*.

[78] P. Gentili, F. Piazza, and A. Uncini, "Efficient genetic algorithm design for power-of-two fir filters," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, 1995, vol. 2, pp. 1268–1271.

[79] J. B. Evans, "Efficient FIR filter architectures suitable for FPGA implementation," *IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process.*, vol. 41, no. 7, pp. 490–493, Jul. 1994.

[80] W. R. Lee, V. Rehbock, K. L. Teo, and L. Caccetta, "Frequency-response masking based fir filter design with power-of-two coefficients and suboptimum PWR," *J. Circuits, Syst., Comput.*, vol. 12, no. 05, pp. 591–599, 2003.

[81] P. Kurup and T. Abbasi, *Logic Synthesis Using Synopsys*. Berlin, Germany: Springer, 2012.

[82] H. Li and W. Ye, "Efficient implementation of FPGA based on vivado high level synthesis," in *Proc. 2nd IEEE Int. Conf. Comput. Commun.*, 2016, pp. 2810–2813.

[83] X. Inc., "Ultrascale architecture DSP slice," 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp

[84] M. Salmani, A. Eshaghi, E. Luan, and S. Saha, "Photonic computing to accelerate data processing in wireless communications," *Opt. Exp.*, vol. 29, no. 14, pp. 22299–22314, 2021.

[85] H. Zhou et al., "Photonic matrix multiplication lights up photonic accelerator and beyond," *Light: Sci. Appl.*, vol. 11, no. 1, 2022, Art. no. 30.

[86] W. Bogaerts et al., "Programmable photonic circuits," *Nature*, vol. 586, no. 7828, pp. 207–216, 2020.

[87] M. Tan, Y. Wang, K. X. Wang, Y. Yu, and X. Zhang, "Circuit-level convergence of electronics and photonics: Basic concepts and recent advances," *Front. Optoelectron.*, vol. 15, no. 1, 2022, Art. no. 16.

[88] F. Pedro, “Code to estimate computational complexity of neural network applications in signal processing,” Mar. 2024, doi: 10.5281/zenodo.10802496..

[89] N. Tran, B. Hawks, J. M. Duarte, N. J. Fraser, A. Pappalardo, and Y. Umuroglu, “PS and QS: Quantization-aware pruning for efficient low latency neural network inference,” *Front. Artif. Intell.*, vol. 4, 2021, Art. no. 676564.

[90] S. V. Padmajarani and M. Muralidhar, “FPGA implementation of multiplier using shift and add technique,” *Int. J. Adv. Electron. Comput. Sci.*, vol. 2, no. 9, pp. 1–5, 2015.

[91] V. S. Dimitrov, K. U. Jarvinen, and J. Adikari, “Area-efficient multipliers based on multiple-radix representations,” *IEEE Trans. Comput.*, vol. 60, no. 2, pp. 189–201, Feb. 2011.

[92] R. I. Hartley, “Subexpression sharing in filters using canonic signed digit multipliers,” *IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process.*, vol. 43, no. 10, pp. 677–688, Oct. 1996.

[93] V. Dimitrov, L. Imbert, and A. Zakaluzny, “Multiplication by a constant is sublinear,” in *Proc. IEEE 18th Symp. Comput. Arithmetic*, 2007, pp. 261–268.

[94] D. Przewlocka-Rus, S. S. Sarwar, H. E. Sumbul, Y. Li, and B. D. Salvo, “Power-of-two quantization for low bitwidth and hardware compliant neural networks,” 2022, *arXiv:2203.05025*.

[95] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[96] Z. C. Lipton, J. Berkowitz, and C. Elkan, “A critical review of recurrent neural networks for sequence learning,” 2015, *arXiv:1506.00019*.

[97] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[98] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with LSTM,” *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, 2000.

[99] R. Dey and F. M. Salem, “Gate-variants of gated recurrent unit (GRU) neural networks,” in *Proc. IEEE 60th Int. Midwest Symp. Circuits Syst.*, 2017, pp. 1597–1600.

[100] Q. Wu, E. Fokoue, and D. Kudithipudi, “On the statistical challenges of echo state networks and some potential remedies,” 2018, *arXiv:1802.07369*.

[101] M. Sorokina, S. Sergeyev, and S. Turitsyn, “Fiber echo state network analogue for high-bandwidth dual-quadrature signal processing,” *Opt. Exp.*, vol. 27, no. 3, pp. 2387–2395, 2019.

[102] S. S. Mosleh, L. Liu, C. Sahin, Y. R. Zheng, and Y. Yi, “Brain-inspired wireless communications: Where reservoir computing meets MIMO-OFDM,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 10, pp. 4694–4708, Oct. 2018.

[103] C. Sun, M. Song, S. Hong, and H. Li, “A review of designs and applications of echo state networks,” 2020, *arXiv:2012.02974*.

[104] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert, “Optimization and applications of echo state networks with leaky-integrator neurons,” *Neural Netw.*, vol. 20, no. 3, pp. 335–352, 2007.

[105] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[106] A. Veit, M. J. Wilber, and S. Belongie, “Residual networks behave like ensembles of relatively shallow networks,” in *Proc. 30th Int. Conf. Neural Inform. Process. Syst.*, Barcelona, Spain, 2016, pp. 550–558.

[107] A. Vaswani et al., “Attention is all you need,” in *Proc. 31st Int. Conf. Neural Inform. Process. Syst.*, Long Beach, California, USA, 2017, pp. 6000–6010.

[108] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016, *arXiv:1607.06450*.

[109] O. Sidelnikov, A. Redyuk, S. Sygletos, M. Fedoruk, and S. K. Turitsyn, “Advanced convolutional neural networks for nonlinearity mitigation in long-haul WDM transmission systems,” *J. Lightw. Technol.*, vol. 39, no. 8, pp. 2397–2406, Apr. 2021.

[110] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[111] E. López, C. Valle, H. Allende-Cid, and H. Allende, “Comparison of recurrent neural networks for wind power forecasting,” in *Proc. Mex. Conf. Pattern Recognit.*, 2020, pp. 25–34.

[112] H. Yu et al., “Any-precision deep neural networks,” 2019, *arXiv:1911.07346*.

[113] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, “Scalable methods for 8-bit training of neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. 2018, vol. 31. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/e82c4b19b8151ddc25d4d93baf7b908f-Paper.pdf

[114] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “HAQ: Hardware-aware automated quantization with mixed precision,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 8612–8620.

[115] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, “Accurate post training quantization with small calibration sets,” in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 4466–4475.

[116] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini, “Fast neural networks without multipliers,” *IEEE Trans. Neural Netw.*, vol. 4, no. 1, pp. 53–62, Jan. 1993.

[117] S.-E. Chang et al., “MSP: An FPGA-specific mixed-scheme, multi-precision deep neural network quantization framework,” 2020, *arXiv:2009.07460*.

[118] S. Srivallapanondh et al., “Parallelization of recurrent neural network-based equalizer for coherent optical systems via knowledge distillation,” *J. Lightw. Technol.*, vol. 42, no. 7, pp. 2275–2284, Apr. 2024.

[119] R. Goyal, J. Vanschoren, V. V. Acht, and S. Nijssen, “Fixed-point quantization of convolutional neural networks for quantized inference on embedded platforms,” 2021, *arXiv:2102.02147*.

[120] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, “Incremental network quantization: Towards lossless CNNs with low-precision weights,” 2017, *arXiv:1702.03044*.

[121] Y. Li, X. Dong, and W. Wang, “Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks,” 2019, *arXiv:1909.13144*.

[122] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” 2015, *arXiv:1510.00149*.

[123] Y. S. Voon, Y. Wu, X. Lin, and K. Siddique, “Performance analysis of CPU, GPU and TPU for deep learning applications,” *Int. J. Des., Anal. Tools Intergrated Circuits Syst.*, vol. 10, no. 1, pp. 12–18, 2021.