Implementing Neural Network-Based Equalizers in a Coherent Optical Transmission System Using Field-Programmable Gate Arrays

Pedro J. Freire, Sasipim Srivallapanondh, Michael Anderson, Bernhard Spinnler, Thomas Bex, Tobias A. Eriksson, Antonio Napoli, Wolfgang Schairer, Nelson Costa, Michaela Blott, *Member, IEEE*, Sergei K. Turitsyn, *Senior Member, IEEE*, and Jaroslaw E. Prilepsky

(Top-Scored Paper)

***Abstract*—In this work, we demonstrate the offline FPGA realization of both recurrent and feedforward neural network (NN)-based equalizers for nonlinearity compensation in coherent optical transmission systems. First, we present a realization pipeline showing the conversion of the models from Python libraries to the FPGA chip synthesis and implementation. Then, we review the main alternatives for the hardware implementation of nonlinear activation functions. The main results are divided into three parts: a performance comparison, an analysis of how activation functions are implemented, and a report on the complexity of the hardware. The performance in Q-factor is presented for the cases of bidirectional long-short-term memory coupled with convolutional NN (biLSTM + CNN) equalizer, CNN equalizer, and standard 1-StpS digital back-propagation (DBP) for the simulation and experiment propagation of a single channel dual-polarization (SC-DP) 16QAM at 34 GBd along 17 $\times$ 70 km of LEAF. The biLSTM+CNN equalizer provides a similar result to DBP and a 1.7 dB Q-factor gain compared with the chromatic dispersion compensation baseline in the experimental dataset. After that, we assess the Q-factor and the impact of hardware utilization when approximating the activation functions of NN using Taylor series, piecewise linear, and look-up table (LUT) approximations. We also show how to mitigate the approximation errors with extra training and provide some insights into possible gradient problems in the LUT approximation. Finally, to evaluate the complexity of hardware implementation to achieve 200G and 400G throughput, fixed-point NN-based equalizers with approximated activation functions are developed and implemented in an FPGA.**

***Index Terms*—Artificial intelligence, coherent detection, computational complexity, FPGA, neural network hardware, nonlinear equalizer, recurrent neural networks.**

I. Introduction

Over the previous couple of decades, the race to find various compensation methods to mitigate the nonlinearities of the fiber and components has produced several noticeable high-performance solutions [1], [2], [3], [4], [5], [6], [7]. However, due to the high complexity of the proposed solutions, only a few published studies [8], [9], [10], [11] have been conducted to implement these solutions in hardware, e.g., in a field programmable gate array (FPGA) or application-specific integrated circuit (ASIC) Recently, machine learning (ML)-based techniques have started to penetrate more and more into different digital signal processing (DSP) applications. Therefore, it is natural now to consider how nonlinear equalizers may be designed, addressing NN-based setups while simultaneously taking into account the issues of flexibility and computational complexity.

A significant number of novel artificial neural networks (NN)-based DSP methods have been developed as a result of research on NN for optical channel equalization: these methods can often provide better performance than that rendered by "conventional" DSP approaches while maintaining competitive computational complexity in terms of the real multipliers number [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. However, such investigations typically deal with the software level. In turn, we stress that a few important extra steps are needed to perform a true evaluation of an NN-DSP device at the hardware level and that this creates some new problems and challenges.

In coherent digital optical transceivers, an FPGA is often used as a prototype to assess the performance of an ASIC [22]. As a result, it is desirable that the NN-based equalizers are implemented in the FPGA to assess the practicability of algorithms used in

Manuscript received 29 November 2022; revised 18 February 2023 and 30 March 2023; accepted 18 April 2023. Date of publication 1 May 2023; date of current version 27 June 2023. The work of Bernhard Spinnler, Antonio Napoli, and Nelson Costa was supported by the European Commission through the H2020 B5G-OPEN Project under Grant 101016663. The work of Sergei K. Turitsyn was supported by the EPSRC Project TRANSNET. The work of Jaroslaw E. Prilepsky was supported by Leverhulme Trust, under Grant RP-2018-063. This work was supported by the EU Horizon 2020 program under Marie Skłodowska-Curie Grants 813144 (REAL-NET) and 956713 (MENTOR). *(Corresponding author: Pedro J. Freire.)*

Pedro J. Freire, Sasipim Srivallapanondh, Michael Anderson, Sergei K. Turitsyn, and Jaroslaw E. Prilepsky are with the Aston Institute of Photonic Technologies, Aston University, B4 7ET Birmingham, U.K. (e-mail: p.freiredecarvalhosouza@aston.ac.uk; s.srivallapanondh@aston.ac.uk; andersm2@aston.ac.uk; s.k.turitsyn@aston.ac.uk; y.prylepskiy1@aston.ac.uk).

Bernhard Spinnler, Thomas Bex, Tobias A. Eriksson, Antonio Napoli, and Wolfgang Schairer are with the Infinera R&D, 81541 Munich, Germany (e-mail: bspinnler@infinera.com; tbex@infinera.com; teriksson@infinera.com; anapoli@infinera.com; wschairer@infinera.com).

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

Michaela Blott is with the AMD Research, Logic Drive, Citywest Business Campus, D24 T683 Dublin, Ireland (e-mail: michaela.blott@amd.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JLT.2023.3272011.

Digital Object Identifier 10.1109/JLT.2023.3272011

real-time systems. At this early stage, the FPGA implementation of the NN-based equalizers can also be based on offline processing [23]. In this article, we outline the procedures required to move both recurrent and feedforward NN-based equalizers (to be deployed in coherent long-haul optical systems) from the software level (Python) to the FPGA realization. Note that our approach can be applied to all NN architectures for channel equalization, so the aforementioned NN architectures are taken just to exemplify the case. However, it should be noted that our research is also important for the other fields in ML applications, as FPGA-based accelerators have been increasingly attracting interest due to their high performance, energy efficiency, fast development cycle, and reconfiguration capability [24]. Furthermore, the driving force behind the deployment of FPGAs is their cloud services applications [25], [26].

In this article, we make a step forward in assessing the viability of NN-based equalizers for industrial applications by benchmarking: i) their performance versus the 1-step-per-span (StpS) digital back-propagation (DBP) using 2.3 samples/symbol (Sa/symbol) in experiments, and ii) their computational complexity by comparing an FPGA implementation against the full electronic chromatic dispersion compensation (CDC) block in the time domain implementation (used, e.g., in standard DSP chain [27]) that needs much fewer resources than the 1-StpS DBP. In addition, for the first time, to the best of our knowledge, we present the FPGA implementation of an NN-based equalizer that employs the bidirectional recurrent layer with long-short-term memory(LSTM) cells (biLSTM). By transmitting a 34 GBd single-channel, dual-polarization (SC-DP) 16QAM signal over 17×70 km of large-effective area fiber (LEAF) (both simulated and experimental cases), we report ≈ 1.7 dB Q-factor improvement over a standard DSP chain while requiring only ≈ 2.5 times more FPGA resources than the implementation of the CDC block to achieve a 400 G transmission.

This article is organized as follows. Section II reviews the previous implementations of NN structures in FPGA, with a special focus on their application for the optical channel equalization task. Section III describes the steps taken to create the NN-based equalizer, from software to hardware. In this section, we introduce the 4 steps in our realization pipeline using the Xilinx tools for high-level synthesis (Vitis HLS) and hardware synthesis (Vivado). Section IV presents a complete study on the realization of nonlinear activation functions in hardware using the three most common approximators: the Taylor approximator, the piecewise linear (PWL) approximator, and the look-up table (LUT) approximator. In this section, the drawbacks of using each of these approximation techniques for performance and complexity are shown. Section V describes the experimental and simulated setups used and the performance in terms of Q-factor for both simulation and experimental datasets. This section also talks about how well different approaches to approximating the activation functions work and how much hardware they use. Finally, we report the computational complexity (utilization), latency, and throughput for all NN strategies studied in our manuscript versus respective quantities of the CDC block when using all available resources on the FPGA under investigation (VCK190 [28]) and when using only LUT and flip-flops (FF) to simulate a realization closer to the ASIC. The last section concludes our paper with a summary of our approach, the results achieved, and some open questions in this field.

II. FPGA Designs for ML-Based Equalization in Optical Transmission

The FPGA is a programmable and reprogrammable integrated circuit that is suitable for resource-constrained embedded applications, as it provides more energy-efficient computation when performing NN on the edge compared to the GPU [29], [30]. FPGA implementations have been investigated for different types of NN, for both feedforward [29], [31], [32] and recurrent NNs (RNN), including LSTM [30], [33], [34], [35]. As NNs can be used in numerous areas, the FPGA design for NN has been intensively researched in different applications: signal processing [36], industrial control applications [37], drive systems [38], and telecommunication equalization [26], [39], [40], [41], [42], [43], [44], [45].

Among the very first works discussing the NN-based equalizers in FPGA, Yen et al. [26] proposed the functional link artificial NN (FLANN) for nonlinear channel equalizers in both software simulation and hardware implementation in FPGA, considering the quadrature phase shift keying (QPSK) modulation. The FLANN was claimed to have a simpler architecture and higher computational speed in hardware compared to the multi-layer perceptron (MLP). Performance comparison was carried out to compare FLANN with the linear least-mean squares equalizer (LMSE). FLANN provided a better symbol error rate than LMSE. However, with parallel processing, FLANN required more logic cells and more area of the chip. To reduce this, the number of data bits in the decimal fraction should be reduced with a trade-off in system performance. Subsequently, the nonlinear channel equalizer based on the NN radial basis function (RBF) with a three-layer structure implemented on FPGA, was investigated in [44]. The results indicated that the bit error rate (BER) performance in the software simulation and that of the Bayesian equalizer, which is a near-optimal method of a channel equalizer, were similar. However, the hardware implementation showed worse results due to the binary value representation on the hardware. To approach the BER of the original RBF NN structure, an increase in the number of bits was recommended.

In more recent works, the convolutional NN (CNN) and binary CNN-based decision schemes for millimeter-wave (mm-wave) radio-over-fiber (RoF) optical communications were presented [45]. Such NN structures outperformed the MLP[1]-based equalizer in terms of complexity, but still achieved the BER performance within the forward error correction (FEC) limit when tested with the 60 GHz mm-wave RoF system. The FPGA-based CNN and binary CNN hardware accelerators with inner parallel optimization were implemented to verify the capability of the FPGA compared to the GPU. Their results showed that the FPGA-based hardware can significantly reduce latency, cost, and power consumption while demonstrating comparable performance.

The studies of the implementation of FPGA-based optical equalizers based on NN have recently gained additional

[1]Note that in [45], the MLP is referred to as the fully-connected NN (FCNN).

attention, but mainly for direct detection systems [39], [41], [42]. In [39], the MLP-based equalizer contained two hidden layers with 33 and 14 neurons, respectively, to equalize the 50 Gb/s passive optical networks (PONs) link. The authors implemented an 8-bit fixed point deep NNs in an FPGA and showed that deep NNs with embedded parallelization successfully reduced the required hardware resources. The authors have extended their work and reported, in detail, in [40], the impact of fixed-point resolution on receiver sensitivity and the utilization of hardware resources in the FPGA implementation of DNN equalizers for PON systems.

The parallel output RNN-based equalizer was proposed in [41]. This parallel RNN is superior to the parallel MLP demonstrated by [39] in terms of BER, as was shown for the 100 Gbps/λ PON. However, the feedback loop structure in the RNN caused hardware implementation challenges, as the output of the nth time step has to propagate back in the loop and appear at the input before the neuron begins to process the n + 1th input. As a result, the authors of that reference only evaluated the parallel MLP. In [42], the time-interleaved parallel pruned MLP-based equalizer for 100 Gbps PAM-4 links was implemented on an FPGA. The NN structure has three layers, and the hidden layer contains 51 neurons. The reported weight pruning algorithm, in this article, is a novel pruning algorithm based on weighting probability to reduce computational complexity while maintaining performance. They reported that a single pruned NN-based equalizer achieved over 55% resource reduction compared to the NN before pruning. Furthermore, up to 40% of resource utilization was reduced for 4- and 8-channel equalization.

However, work on the FPGA implementation of NN-based equalizers in the case of coherent detection optical systems is still mostly missing. We mention only [43], where the authors demonstrated the mitigation of optical fiber non-linearity in a 16-QAM self-coherent real-time system at 40 Gb/s using a FPGA implementing the sparse `K-means++` algorithm instead of an NN. In this case, the authors reported a 3 dB Q-factor improvement with respect to linear equalization only after transmission along 50 km of optical fiber using a launch power close to the optimal value of 14 dBm. In contrast to the case considered in this article, the tested scenario was a single-span short-reach system.

In our work, we describe and detail the next step in the implementation of NN equalizers: For the first time, the offline FPGA implementation[2] of an NN equalizer employing the recurrent layer (biLSTM), as well as a deep CNN structure, is presented and evaluated in the experimental data for a high-speed coherent optical transmission system.

III. Neural Network Equalizers Designing Pipeline: From Python to FPGA

In this section, we look at the design tools and process steps that were used to implement the NN in an FPGA. In subsection III-A, corresponding to Step 01 in Fig. 3, the NN architectures studied in this work are presented and the details of the training phase are depicted. In subsection III-B, specific attention is devoted to the C++ model (Step 02 in Fig. 3) and the high-level synthesis (HLS) process (Step 03 in Fig. 3) used to generate a description of the NN in VHDL (Very High-Speed Integrated Circuit Hardware Description Language). We also explain the motivation behind the use of the HLS method. In addition, we discuss some important considerations for using HLS, intending to support future research activities that involve this method. Then, in subsection III-C, we look at the physical implementation aspects of the design flow, performed there by using the Vivado design suite as shown in Step 04 of Fig. 3, which produces the final results related to the FPGA hardware.

A. Neural Network Architectures and Python Training Process

The two NN architectures for the equalizers investigated in our work are depicted in Fig. 1(a) and (b) for the biLSTM-based equalizer and the deep CNN-based equalizer, respectively. The shape of both architectures is similar, but the nature of the mathematical operations in each case is different: biLSTM contains recursive connections, as can be seen in Fig. 2 for the LSTM cell structure, whereas deep CNN is a feedforward network. In Fig. 2, the dashed line indicates that the recursive connections due to the equations of a forward pass of an LSTM cell with a time step t are given as:

$$\begin{aligned}
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i),\\
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f),\\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o),\\
\tilde{c_t} &= \phi(W_c x_t + U_c h_{t-1} + b_c)\\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c_t},\\
h_t &= o_t \odot \phi(c_t),
\end{aligned} \tag{1}$$

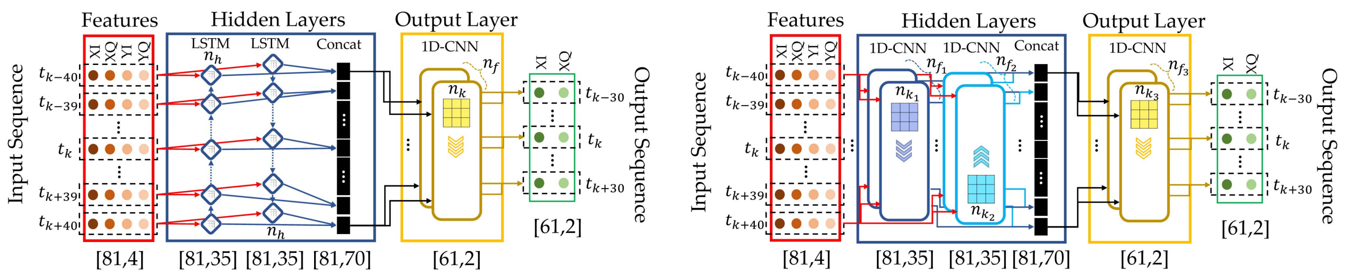
where ϕ is usually the "tanh" activation function, σ is usually the sigmoid activation function, $x_t \in \mathbb{R}^{n_i}$ is the n_i-dimensional input vector at time t, n_h is the number of hidden units, $W \in \mathbb{R}^{n_h \times n_i}$ and $U \in \mathbb{R}^{n_h \times n_h}$ representing the trainable weight matrices, and $b \in \mathbb{R}^{n_h}$ is the bias vector. $i_t \in (0,1)^{n_h}, f_t \in (0,1)^{n_h}, o_t \in (0,1)^{n_h}, \tilde{c_t} \in (-1,1)^{n_h}$, $c_t \in \mathbb{R}^{n_h}$, and $h_t \in (-1,1)^{n_h}$ denote input gate, forget gate, output gate, cell input, cell state, and hidden state vectors, respectively. The $\odot$ symbol represents the element-wise (Hadamard) multiplication.

The equation of 1D-convolutional layer can be formulized as:

$$y_i^f = \phi\left(\sum_{n=1}^{n_i}\sum_{j=1}^{n_k} x_{i+j-1,n}^{in} \cdot k_{j,n}^f + b^f\right), \tag{2}$$

where y_i^f denotes the output, or 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.

[2]Offline FPGA implementation refers to the process of designing and configuring an FPGA before it is deployed in a target system. In this article, we simulate "offline" the constraints affecting an FPGA when taking into account the NN simulation and tested on the VCK190 board with some sample data that were saved in the FPGA's memory just to verify that the model (bitstream file) was working according to its design.



(a) biLSTM+CNN. (b) Deep CNN.

Fig. 1. Structures of NN-based equalizers taking 81 symbols as input to recover 61 symbols in parallel at the output: (a) the recurrent equalizer using a bidirectional LSTM layer containing 35 hidden units, and (b) the feedforward equalizer using a 1D-convolutional layer consisting of 70 filters ($n_{f_1} = n_{f_2} = 35$).

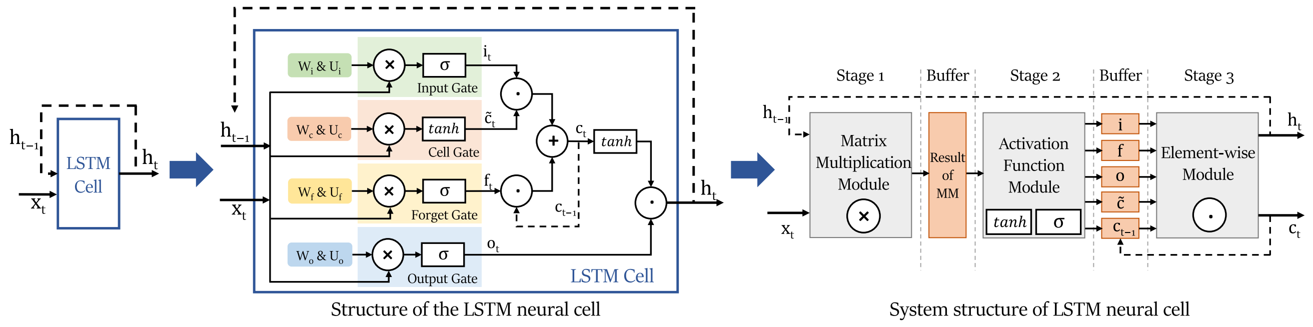


Fig. 2. LSTM cell structure showing the recursive connections (detailed in Section III-A) and the system structure of LSTM implementation in a modular way showing the buffers needed to store outputs (detailed in Section V-D).

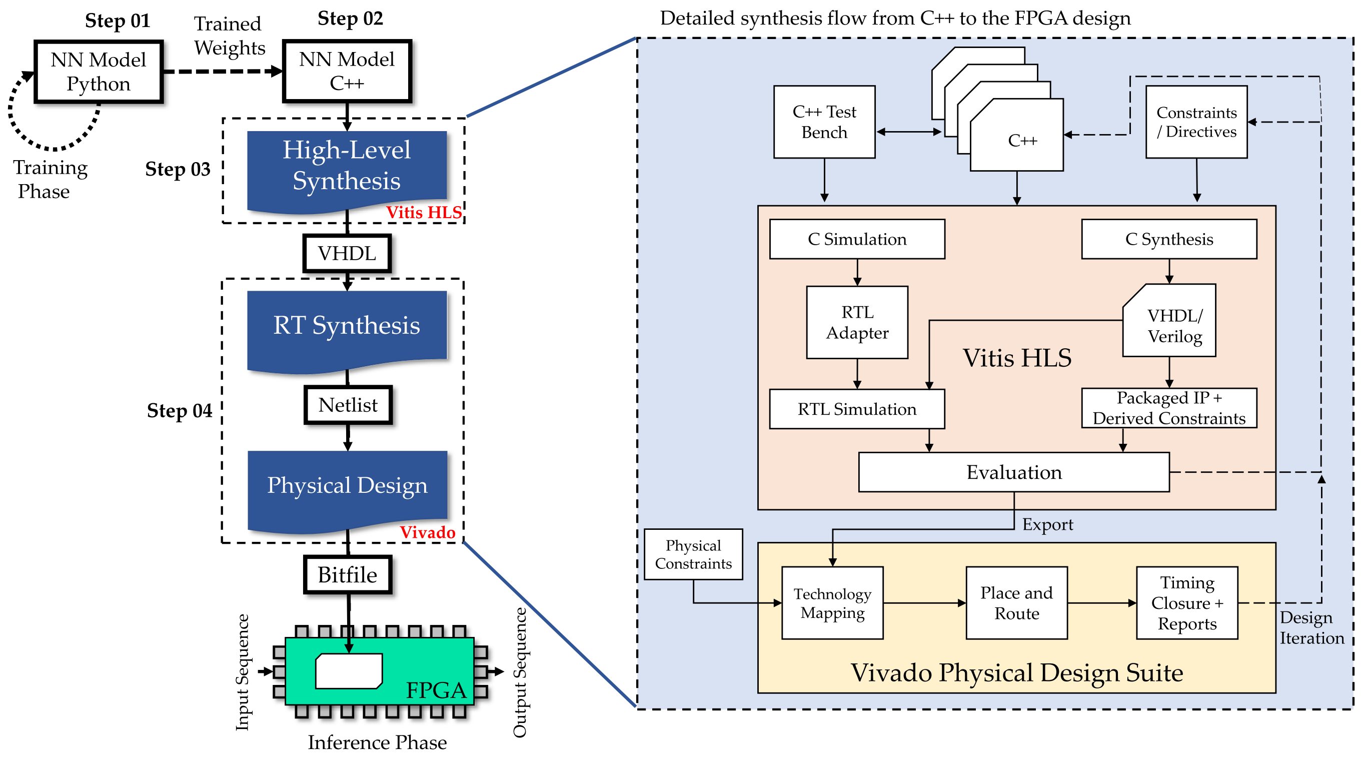


Fig. 3. High-level synthesis flow – from C++ to FPGA realization.

The NN-based equalizer is applied after the standard DSP chain.[3] In both the biLSTM and deep CNN equalizers, a total of 81 symbols are used as input. The input vector x_t consists of four features from four real values (X_I, X_Q, Y_I, and Y_Q) from x and y polarizations: $X_I + jX_Q$ and $Y_I + jY_Q$, respectively. The time domain depth is an additional dimension that characterizes the system's memory. Consequently, the input shape can be represented as (Batch size, Memory, 4). The window memory of 81 symbols as input allows us to simultaneously retrieve 61 symbols at the equalizer's output. By recovering 61 symbols in parallel at the equalizer output, we allow the FPGA realization to have a higher throughput, which is one of the main desirable design features we are looking for when building a DSP block for coherent transmission systems. It is worth noting that the NN output layer in this scenario recovers the X polarization, as the CDC block is applied independently to each polarization. However, in future research and implementation, the NN's output layer can also be modified to recover both polarizations simultaneously. In Fig. 1(a), the hidden layer consists of a biLSTM layer with $n_h = 35$ hidden units. The number of hidden units is actually the dimension of the hidden state or the output of the hidden layer. The LSTM structure in this case is bidirectional. Therefore, one LSTM layer takes the input in a forward "time" direction and another LSTM layer takes the input in a reverse "time" direction, with small arrows in Fig. 1(a) between the LSTM hidden units indicating the directions of the input fed in the LSTM layers. In Fig. 1(b), the hidden layer is made up of a CNN layer with 70 filters ($n_{f_1} = n_{f_2} = 35$), with zero padding applied to retain the shape and the kernel size $n_{k_1} = n_{k_2} = 11$. The output layer in both designs is a convolutional layer with $n_f = 2$ filters, a kernel size $n_k = 21$, and no padding. Concat block is a concatenation block. For the biLSTM, the bidirectional layer from Tensorflow already includes the concatenation. The concatenation is to combine the outputs of the forward and backward LSTMs. For the deep CNN, the concatenation is to combine the outputs of each filter of the CNN. Based on a grid search analysis, these parameters were chosen to meet the hardware limitations, throughput requirements, and optical performance required for this FPGA realization. The activation functions for both hidden layers were hyperbolic tangent ($tanh$), and the output layer is linear.

Focusing now on the biLSTM+CNN architecture implemented in this article, the mean square error (MSE) loss estimator and the classical Adam algorithm for the stochastic optimization step [46] were used when training the weights and bias of the NN. The training hyperparameters (mini-batch size equal to 2001 and learning rate equal to 0.0005) were found using the Bayesian optimization procedure described in [20]. The NN training was carried out by backpropagation for 30000 epochs with a fixed set of hyperparameters[4]. The BER is evaluated after each training epoch. For training, we used a fixed dataset with 2^{20} symbols, and, at every epoch, we picked 2^{18} random input symbols from this dataset. For the testing and validation, we used a never-before-seen dataset with 2^{18} symbols. Both NN models were trained, validated, and tested using the same datasets. The weights were saved at the epoch at which the BER measured using the validation dataset was the lowest (early stopping).

[3]Hence, the time recovery and other typical DSP blocks for coherent transmission are already handled, and the sampling rate of 1 sample per symbol is already in place.

[4]For applying the model to other launch power values, we used transfer learning [47], which allowed us to utilize less than 5 epochs to adjust the NN weights to the other launch powers.

B. *C++ Implementation of Neural Networks and the High-Level Synthesis Method to Generate VHDL*

The HLS method provides a design flow in which the desired function, i.e., an NN in the case considered, can be specified in C++ and then automatically converted to VHDL. Note that the NN implementation in C++ is coded from scratch. This strategy is preferred because it separates the FPGA technology-specific implementation features, such as clocks and logic cell topology, from the NN's functionality. Working at a higher level of abstraction, the intended functionality can be the focus of attention and be described more easily in fewer lines of code [48].

In addition, functional verification in C++ is much faster than functional simulation in VHDL. This makes it easy to test and debug the design. At this point, it is important to remember that the functions described will run on hardware. HLS supports a substantial range of C++ syntax, but not all, because some cannot be implemented in an FPGA or ASIC. In this case, memory allocation is a key part of writing the C++ code that needs to be properly assessed. In the FPGA, the memory allocations are static and are assigned during the mapping phase of the physical design flow. Dynamic memory allocations found in many C++ standard library functions cannot be supported and must be avoided or replaced with structures optimized for the implementation in an FPGA by using the libraries provided by the HLS tool supplier, in our case, Xilinx. Also, Operating System (OS) functions such as file read/write and date/time cannot be implemented in the FPGA; all data transmitted into and out of the FPGA must use input/output ports.

Consequently, two versions of the C++ codes were generated. The first is called here the test bench, while the second is the function to be implemented in FPGA. The test bench function reads the previously saved signal inputs and weights learned in Python and converts them to a fixed-point format (int32). These values are then incorporated into the function that describes the equalizers investigated in this study. The function is the C++ translation of the Python NN equalization architecture, using fixed-point arithmetic operations. After the equalization, the outputs of the function (the signal that has been equalized) are delivered to the test bench code that checks the MSE. We did not study the impact of further reducing the quantization accuracy, since this would require some further work to overcome the quantization error caused by both the input signal and the weights. We chose the int32 format because, by using it, we can take advantage of simplified integer arithmetic while observing no significant performance reduction compared to the floating-point BER evaluated in Python. Note that INT32 is the quantization format for input and weights in this article, and different types of quantization are studied in [49].

Here, it is important to highlight details on the implementation of the convolutions and the LSTM cells in our C++ code. For the convolution implementation, we used the conventional method for convolution, which consists of a series of for loops that can be partialized, as detailed in Ref. [50]. In the case of the LSTM cell, l, whose implementation is depicted in Fig. 2, our approach adhered to the methods outlined in Ref. [51]. In summary, the input data h_{t-1}, x_t, and weight matrix W are read, and the systolic array technique is used to do the matrix multiplication. Its output is temporarily stored in global memory on the chip. Then, the activation function module (e.g., Taylor approximation, PWL approximation, or LUT approximation) receives the input data from the temporary result buffer and obtains the output vectors i, f, o, and g, as shown in 1. Each gate's output is also buffered and saved in the chip's global memory. Next, we implemented the element-wise computation module, which reads the data of i, f, o and g from the buffer, completes the element-wise computation, as shown in Fig. 2 and then obtains the output h_t and cell state c_t, which will be used in the next time step. After all required time steps have been completed, the final output is written back to the host memory.

However, when using the HLS, even though the conversion to VHDL is automated, some design intervention is still necessary, and engineering decisions must be taken to achieve the desired performance. HLS supports a set of directives, or pragmas, that can be used to modify the behavior of the HLS C++-synthesis stage to facilitate these interventions [52]. By utilizing pragmas, in order to discover the best implementation, it is useful to investigate several design structures without re-coding them. Although there are a variety of different pragmas, we have primarily utilized those pertaining to pipelines, loops, and arrays.

Pipelines allow the parallel execution of operations within a function, lowering the number of clock cycles between commencing loop iterations; these clock cycles are referred to as the Iteration Interval (II). Each loop iteration does not need to end before the next one begins, i.e., the iterations can overlap. The number of pipeline stages can be controlled by setting the value of II using the HLS pipeline pragma. Setting the II to 1, as is done in this project, enables each cycle to begin with a new iteration.

Loops can be unrolled and flattened. By default, loops within a function remain rolled, which means that the loop body is executed sequentially, utilizing a single set of logic resources. The minimum loop delay is then equal to the number of loop iterations. Unrolling generates several copies of the loop body logic, enabling parallel execution and reduced latency at the expense of additional size. Loops can be entirely or partially unrolled, resulting in either one copy of the loop body every iteration for optimum throughput or fewer copies for reduced area cost. Flattening transforms a hierarchy of nested loops into a single loop, which eliminates a clock cycle delay while traveling between higher and lower nested loops and can help with better optimization of the loop logic. Given this, the loops in the feedforward NN layers can be flattened; however, the loops in the recurrent NN layers cannot be flattened due to their memory dependence and imperfect loops. Therefore, for the implementation of the recurrent NN layer, we shall only flatten the loops relating to the matrix multiplication that occurs internally within each LSTM cell.

Arrays can be partitioned and reshaped. HLS will implement arrays in the C++ code as memory blocks in the VHDL description. This can cause a restriction in the design concurrency as FPGA memory blocks only have 2 access ports, which, if the designer has also used the previously mentioned unroll and pipeline pragmas, will need to be shared between all instances of the loop body. By reshaping and partitioning the array, the size, and the number of these memory blocks can be controlled. To enable the successful flattening of the loops in each NN architecture, the non-equalized signal (input signal) and the weights of the NN architecture are partitioned here. The final stage of the HLS step is to export the generated VHDL and derived constraints for use in the physical design step.

Here, we emphasize that NNs are an excellent candidate for exploiting the benefits of HLS, as their nested architecture consisting of multiple layers and several multiply/accumulate functions can make good use of the loop and pipeline directives to investigate the trade-off between area and latency to meet the design requirements.

C. Vivado and the Synthesis Step to the FPGA Realization

The area and timing reports generated by the HLS stage are still simply estimates of the final design performance based on the technology-specific data libraries for each FPGA; the actual performance cannot be determined until the physical implementation is complete. In our work, the Vivado Design suite from Xilinx is utilized. The physical implementation is performed by Vivado in three steps: technology mapping, placement and routing, and timing analysis.

Technology Mapping: Within this step, the VHDL source code is translated into primitive logic gates and boolean equations, followed by mapping these gates onto FPGA customizable logic blocks containing D-type FFs (DFFs) and RAM-based LUTs or more specialized functional cells, such as DSPs. During the technology mapping, the design is optimized and unnecessary logic is eliminated. Note that the detailed explanation of the FPGA components can be found in Appendix A.

The next two steps constitute an iterative process executed automatically by the tool based on design constraints, such as a clock frequency. These limitations can be inherited from the HLS stage or defined in Vivado. The Vivado tool imposes sets of restrictions through established optimization strategies, which are discussed in detail in the vendor user manuals [53], and which the user can apply depending on the design goals. The optimal technique is determined by balancing computer runtime and outcomes. In [54] we can find all potential pairings of synthesis and optimization procedures in Vivado using a high-speed pulse width modulation circuit as a target design, as well as a comprehensive evaluation of the runtime versus performance of the different Vivado optimization methodologies. Since the goal of our work was to increase throughput, we did not look at solutions that would reduce chip size, power, or runtime. Therefore, the Vivado configuration called "Performance ExtraTimingOpt" is adopted in our work, since it effectively optimizes throughput by reducing timing slack [54].

Placement and Routing: This stage positions the logic blocks developed during the mapping phase, onto the specified elements of the FPGA cell array, and configures the signal routing channels between them. The placement algorithm starts from a random seed position and then moves functions to the cell array based on the degree of congestion for the parts of the die and the fanout of the driving function.

Time Analysis: This stage compares the design with the applied timing constraints to determine whether the overall performance requirement has been met. Timing analysis, in particular, requires a grasp of the FPGA structure and how the design has been mapped onto the array. It may be necessary to return to the HLS phase to apply more directives or adjust the function architecture to achieve timing closure. The time for a data (signal) to travel between two points is determined by a variety of factors, including DFF switching time, setup requirements (the time at which the signal must arrive at the destination before the capturing clock edge), logic and routing path delays, and clock edge uncertainty due to jitter and clock path skew. Here, it is pertinent to define the negative timing slack.

The negative timing slack indicates that the total delay in the data path between two DFFs is greater than the requested clock period. In this negative slack case, the NN has many nested loops, as discussed in the previous section; unrolling these loops would lead to a larger design consuming more logic area, but leaving large loops, i.e., the loops with a high number of iterations, can produce long logic multiplexer paths as the inputs to the loop logic are selected. Using Vivado timing analysis reports and annotated netlist viewer, we can identify which paths can be the cause of the failed paths. In this case, the solution was to return to the C++ source and reorder the nested loops so that the outer loop, which was not unrolled, had fewer iterations; this approach reduced the size of the logic chain in the multiplexer path. In this article, we have also decreased the clock frequency for each of the designed blocks to guarantee that a zero negative timing slack was achieved in all FPGA designs.

IV. The Nonlinear Activation Function Implementation: High Accuracy and Lower Complexity

In this section, we address the implementation of NNs' nonlinear activation function, one of the crucial components in the design of NN in hardware. In contrast to the hardware realization of the NN's weights and inputs, where we can readily proceed from the float to fixed-point representation, the activation functions' realization in hardware is not straightforward. In an LSTM cell, the sigmoid and tanh functions are deployed as activation functions, and they are computationally expensive. Both functions contain exponential functions, making it difficult to implement them on resource-constrained hardware and requiring a large chip area [55]. Therefore, function approximation techniques are required in place of the exact functions to realize them in the FPGA, and to reduce the overall computational complexity [55], [56], [57], [58]. In this article, we focus on approximating the sigmoid and tanh. We consider three different methods for the approximation: Taylor series expansion, PWL,

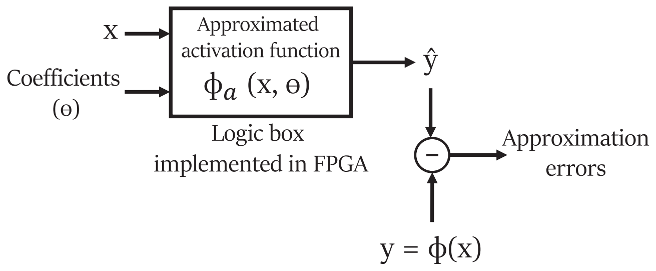

x
Coefficients (θ)
Approximated activation function
ϕ_a (x, θ)
Logic box implemented in FPGA
ŷ
−
Approximation errors
y = ϕ(x)

Fig. 4. Diagram of the input/output of approximated activation functions based on the logic box implemented in FPGA.

and LUT. As shown in Fig. 4, to implement the approximated activation functions on the FPGA, the FF,[5] LUT,[6] and DSP slices are used to build the logic box,[7] which takes the value x and coefficients to return $\hat{y}$. The coefficients are stored in the memory as input. The coefficients define the Taylor and PWL approximations, while in the LUT approximation, they represent the quantization levels list. $\hat{y}$ is the output of the approximated activation functions, while y represents the actual output of the float-precision activation function. The difference between $\hat{y}$ and y is the approximation error.

The expression for the tanh function via exponential is:

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{3}$$

while that for the sigmoid function reads as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{4}$$

A. Taylor Approximation Approach

In the Taylor series approximation, the higher the degree of an approximating polynomial n, the better the approximation. The tanh Taylor series reads as:

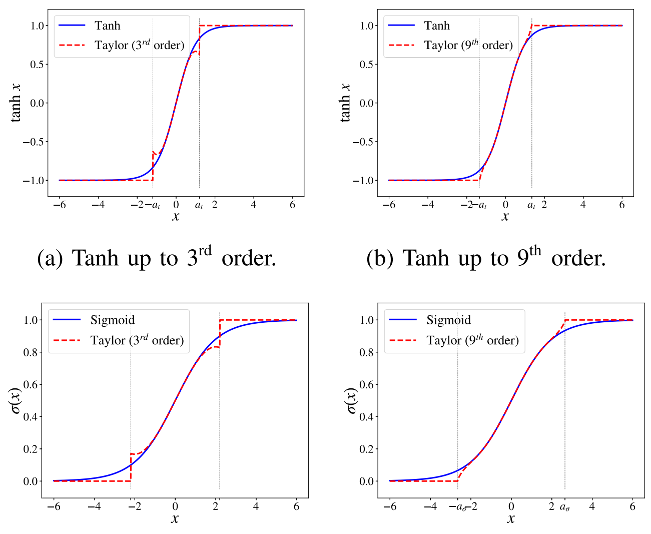
$$\begin{aligned} \tanh x &= \sum_{n=0}^{\infty} \frac{2^{2n}(2^{2n}-1)B_{2n}}{(2n)!} x^{2n-1}, \quad \text{where } |x| < \frac{\pi}{2} \\ &= x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \frac{62x^9}{2835} - \dots, \end{aligned} \tag{5}$$

where B_{2n} denotes the Bernoulli number [59], $-a_t < x < a_t$, and a_t is the boundary of the approximation region: when x is not within $[-a_t, a_t]$, the approximation error is essential. Therefore, it is important to choose the value of a_t that maximizes performance. Empirically, the slight difference in the value of a_t can noticeably affect the performance. When x is outside the Taylor

[5]FF is a basic digital storage element in an FPGA, used to store the value of a digital signal and can be used in conjunction with LUTs to implement sequential logic, such as state machines and counters.

[6]LUT is a basic building block of an FPGA used to implement equations built from Boolean logic functions, such as AND, OR, and XOR, or to store pre-calculated values for use in arithmetic or other operations.

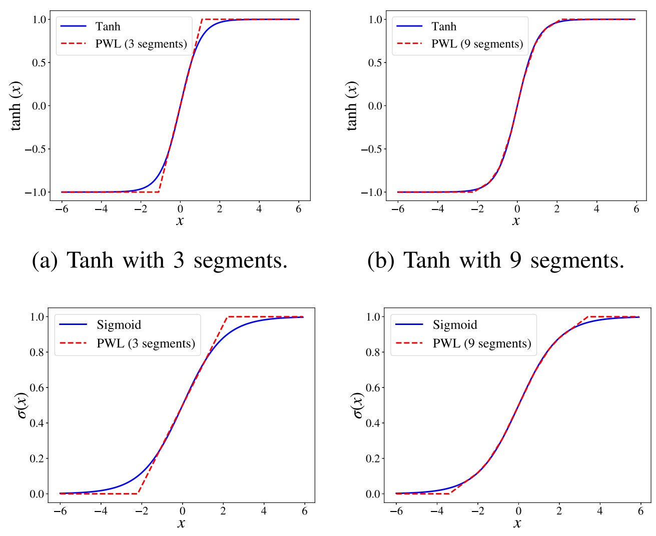
[7]FPGA uses LUTs, FF and DSP slices together to implement the digital logic, memory, and computation required by the intended applications. LUTs, FFs, and DSPs are all programmable, meaning that the user can reprogram the FPGA's logic, memory, and computation elements to suit different applications.



(a) Tanh up to 3rd order. (b) Tanh up to 9th order.

(c) Sigmoid up to 3rd order. (d) Sigmoid up to 9th order.

Fig. 5. Taylor series approximation of tanh (a)–(b) and sigmoid functions (c)–(d).



(a) Tanh with 3 segments. (b) Tanh with 9 segments.

(c) Sigmoid with 3 segments. (d) Sigmoid with 9 segments.

Fig. 6. PWL approximation of tanh (a)–(b) and sigmoid functions (c)–(d).

series approximation region, we set the value of tanh x to -1 or 1, according to the following expression:

$$\tanh x = \begin{cases} 1, & \text{if } x > a_t, \\ x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \frac{62x^9}{2835}, & \text{if } -a_t < x < a_t, \\ -1, & \text{if } x < -a_t. \end{cases} \tag{6}$$

The plots for the different order Taylor approximations are given in Fig. 5(a) and (b). The value of a_t is the result of the grid search, which maximizes the performance of our NN-based equalizer without re-training.

The Taylor series for the sigmoid function is:

$$\begin{aligned} \sigma(x) &= \frac{1}{2} + \frac{1}{2}\tanh\frac{x}{2} \\ &= \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^5}{480} - \frac{17x^7}{80640} + \frac{31x^9}{1451520} - \dots, \end{aligned} \tag{7}$$

where $-a_\sigma < x < a_\sigma$ and a_σ is the point where the Taylor series approximation of the sigmoid starts to diverge. Similarly to tanh, the values of the sigmoid approximation in regions less than $-a_\sigma$ and greater than a_σ are set to 0 and 1, respectively, as follows:

$$\sigma(x) = \begin{cases} 1, & \text{if } x > a_\sigma, \\ \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^5}{480} - \frac{17x^7}{80640} + \frac{31x^9}{1451520}, & \text{if } -a_\sigma < x < a_\sigma, \\ 0, & \text{if } x < -a_\sigma. \end{cases} \tag{8}$$

The Taylor approximation plots corresponding to (8), when the highest order of the polynomial is 3 and 9, are given in Fig. 5(c) and (d).

We evaluate the performance (in terms of Q-factor) when the approximation for both tanh and sigmoid functions is carried out simultaneously, with different orders of the approximating polynomial up to 9th order. The values of a_t and a_σ are chosen by using the grid search, aiming to maximize the Q-factor when replacing the exact activation functions with their Taylor series approximation without re-training the weights. The Taylor series approximation reduces the computational cost and time required to compute the activation function considerably, compared to the processing using the original function [57].

B. Piecewise Linear Approximation Approach

The PWL approximation, introduced in [60], is a combination of linear segments that approximates the activation or nonlinear function [56], [61]. Increasing the number of linear segments to represent the nonlinear function allows us to achieve better accuracy. The PWL approximation is a promising method to reach a higher processing speed since it consumes fewer resources on FPGA[8] compared to the Taylor approximation: to reach higher accuracy, the Taylor approach fits the nonlinear function with high-order expressions, which results in the consumption of resources, while the PWL can reach the same level of accuracy with the use of more segments, but without employing high-order operations [56].

In this article, we compare the performance of our NN-based equalizers when applying 3-, 5-, 7-, and 9-segment PWL approximations to both tanh and sigmoid.[9] The expressions for the PWL used in this article are included in Table III in Appendix B. The corresponding plots for the equations mentioned in Table III with 3 and 9 segments are depicted in Fig. 6(a) and (b) for tanh, and Fig. 6(c) and (d) for sigmoid. Note that we use grid search to find the coefficients for each expression, aiming to maximize the performance in terms of Q-factor, after the actual activation functions are replaced by the approximations over the trained weights, instead of minimizing the difference/areas between the exact function and the approximation curves. It is carried out

[8] [62] shows that the implementation of PWL can be further optimized to have zero multipliers by simplifying the shift and addition operations.

[9] Note that when the number of segments is lower than 3 segments that used to represent sigmoid or tanh in the biLSTM cell, the biLSTM model in our case is not able to learn to mitigate the approximation errors.

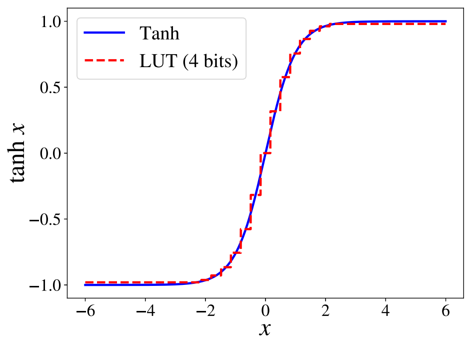


Fig. 7. LUT approximation of tanh function with the number of bits equal to 4.

because, in our case, minimizing the difference/areas between the curves noticeably degrades the Q-factor performance of the NN equalizer when the NN predicts the output with the replaced approximated activation functions.

C. Lookup Table Approximation Approach

The LUT approximation is a commonly used method for the activation functions' hardware implementation [63]. The LUT approximates the function with a limited number of uniformly distributed points. This approach offers a high-performance design, and the fastest implementation compared to other methods. At the same time, a large amount of memory is required to store the LUT on the hardware [64], [65]. The chip area requirements for the LUT approximation grow exponentially with the required approximation accuracy [65]. The number of bits used to represent values in the LUT directly affects the approximation error and the required memory size. An example of the LUT approximation of tanh with the number of bits equal to 4 is presented in Fig. 7.

The LUT approach is similar to traditional quantization, in which full precision values are assigned to uniform quantization levels, i.e. the value x is mapped to $\hat{x}$ which is the closest value of x in the quantization level list [66]. The LUT stores the values of the quantization levels ($\hat{x}$) and their corresponding $f(\hat{x})$, in our case $tanh(\hat{x})$ or $\sigma(\hat{x})$. The difference between the exact value $f(x)$ (the blue curve in Fig. 7) and the approximation $f(\hat{x})$ (the red curve in Fig. 7) introduces the approximation errors.

We investigate the Q-factor performance of our model for the LUT representation of activation functions when the number of bits used ranges from 2 to 16.

D. Reducing Approximation Error Through the Learning Via Stochastic Gradient Descent

Once the activation functions are replaced by the approximation, the NN performance can drastically drop. However, training the model with approximated activation functions can enhance the performance because the model learns to reduce the approximation error. Stochastic gradient descent (SGD) is the training approach that we apply in this article. The training can be undertaken from scratch, which means that the NN is trained when the activation functions are replaced by approximations from the beginning without any pre-assigned weights. Another approach to training is to use the weights of the model pre-trained with the true activation functions, then re-train the model after the replacement of the approximations to learn the approximation errors. The latter results in a considerably shorter training time. We report the results of the second method because, in our tests, training from scratch takes significantly longer to converge and sometimes can provide even worse results. It is worth noting that another available training approach is to only train the coefficients of the Taylor and PWL equations without re-training the NN weights; however, in our case, the performance was not acceptable when using a low number of segments in PWL and training with this approach.

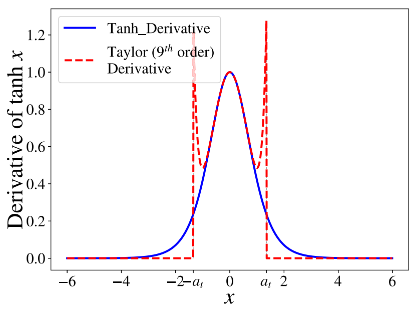
To train the NN with the approximation of the activation function via the SGD, the gradient of the approximation function must be computed. For the Taylor approximation, the Taylor series gradient is calculated with respect to the Taylor series approximation (6) for tanh and (7). Fig. 8(a) shows an example of the derivative of the tanh approximation using the Taylor series with the highest order of 9; the gradient (red curve) is not smooth due to the polynomial nature of the Taylor series. This fact can limit the training ability, especially when training from scratch, as noted in Section V-C. Concerning the PWL, the gradient is the slope of the expressions from Table III (in the Appendix section). Fig. 8(b) depicts the gradient of the PWL approximation with 9 segments. Note that due to the non-differentiability of LUT, it is challenging to learn the LUT-approximated model [66]. In this article, to train the LUT, we generate LUTs for the gradient of both sigmoid and tanh for each interval of the LUT approximations. Fig. 8(c) shows the gradient of the tanh LUT with 4 bits, corresponding to the tanh approximation in Fig. 7.

V. Results and Discussions

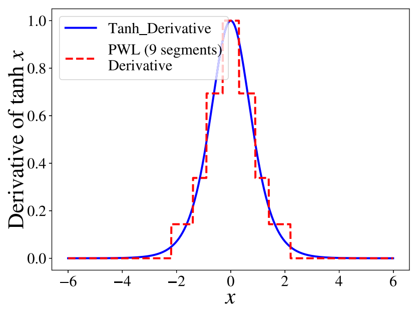
A. Experimental and Numerical Setups

We assess the performance of the NN-based equalizers with reduced complexity by using the data not only from numerical simulations but also from a real experimental setup, to make our analysis as complete as possible. The setup used in our experiment is shown in Fig. 9. At the transmitter, a DP-16QAM 34 GBd symbol sequence was mapped out of the data bits generated by a Mersenne Twister algorithm [67]. Then, a digital RRC filter with 0.1 roll-off was applied. The resulting filtered digital samples were resampled and uploaded to a digital-to-analog converter (DAC) operating at 88 GSamples/s. The output of the DAC was amplified by a four-channel electrical amplifier that drove a dual-polarization IQ Mach-Zehnder modulator, modulating the continuous waveform carrier produced by an external cavity laser at the wavelength $\lambda = 1.55\ \mu$m. The resulting optical signal was transmitted over 17 $\times$ 70 km spans of LEAF. Erbium-doped fiber amplifiers (EDFAs) are used to compensate for the loss in each fiber span at their output. The EDFA's noise figure was in a 4.5 to 5 dB range. The parameters of the LEAF are: the attenuation coefficient $\alpha = 0.225$ dB/km, the chromatic dispersion coefficient $D = 4.2$ ps/(nm $\cdot$ km), and the effective nonlinear coefficient $\gamma = 2$ (W$\cdot$ km)$^{-1}$.

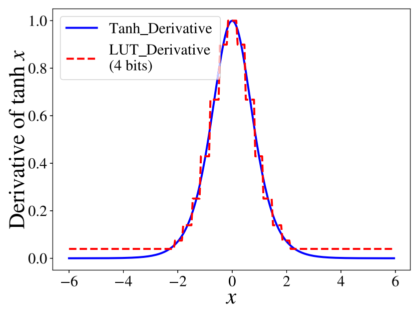
On the Rx side, the optical signal was converted to the electrical domain using an integrated coherent receiver. The resulting signal was sampled at 80 Gsamples/s with a digital sampling



(a) Gradient of Taylor approximation.



(b) Gradient of PWL approximation.



(c) Gradient of LUT approximation.

Fig. 8. The derivative of the tanh function for the approximations using (a) Taylor series with the highest order of 9, (b) PWL with 9 segments, (c) LUT approximation with the number of bits equal to 4.

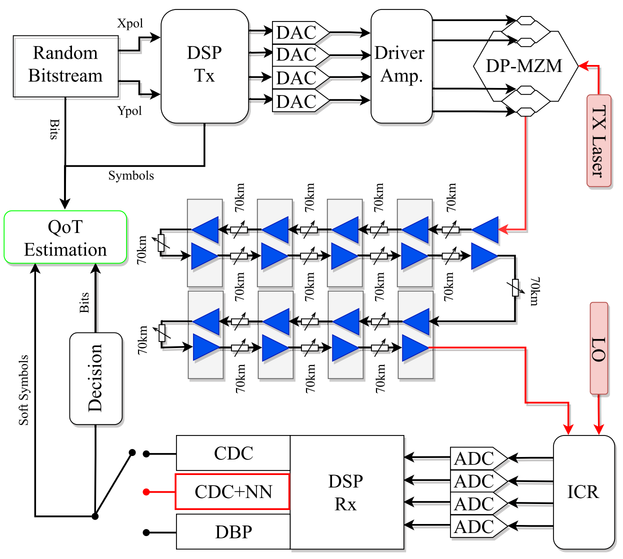


Fig. 9. Experimental setup. The input of the NN (shown as the red rectangle after DSP RX) is the soft output of the regular DSP before the decision unit.

oscilloscope and processed by an offline DSP based on the algorithms described in [68]. First, the bulk accumulated dispersion was compensated using a frequency domain equalizer, which was followed by the mitigation of the carrier frequency offset. A constant-amplitude zero autocorrelation (CAZAC)-based training sequence was then located in the received frame, and the equalizer transfer function was estimated from it. Afterward, the two polarizations were demultiplexed, and the signal was corrected for clock frequency and phase offsets. The carrier phase estimation was then carried out with the help of pilot symbols. Subsequently, the resulting soft symbols were used as the input for the NN equalizer. Finally, the pre-FEC BER was evaluated from the signal at the NN output. The performance of the system was evaluated in terms of the Q-factor, expressed through the BER as $Q = 20\log_{10}[\sqrt{2}\,\mathrm{erfc}^{-1}(2\,\mathrm{BER})]$.

Concerning the simulation, we tried to mimic the experimental transmission scenario. The propagation of the signal along the fiber was simulated by solving the Manakov equations using the split-step Fourier method with a step size of 1 km. At the receiver, after the full CDC (time domain) and downsampling to the symbol rate, the received symbols were normalized to the transmitted ones. The normalization process can be viewed as its normalization by a constant $\mathcal{K}_{\text{DSP}}$ learned using the following equation:

$$\mathcal{K}_{\text{DSP}} = \min_{\mathcal{K}} \left\| \mathcal{K} \cdot x_{h/v}(z,t) - x_{h/v}(0,t) \right\|, \tag{9}$$

where the constants $\mathcal{K}$, $\mathcal{K}_{\text{DSP}} \in \mathbb{C}$ and $x_{h/v}$ is the signal in h or v polarization. Furthermore, the Gaussian noise was added to the data signal, as to represent the additional transceiver components-induced distortions present in the experiment. As a result, the Q-factor level of the simulated data (without NN equalization) was matched to the experimental one. Note that the polarization mode dispersion (PMD) is not considered in the simulation. In the experimental data, PMD was already compensated by Infinera's DSP, so we can say that our study is not influenced by PMD.

Finally, unlike the NN equalizer, which operates with 1 Sa/symbol, the DBP used to benchmark the performance curves (the implementation described in [2]), operated with 2.3 Sa/symbol (and with 1 StpS with the scheme parameters optimized for the best performance). Regarding the CDC implementation, we designed a time-domain equalizer as in [27] with 517 taps in C++. For the realization in hardware, we followed the same design steps 3 and 4 described in Section III for the NN implementation. To be more specific, the goal of this result section is to assess the complexity of NN with respect to CDC, while guaranteeing a level of nonlinear compensation comparable to one of the widely used DBP. The CDC benchmark is the most important because our primary goal is to show the readiness of NN with respect to the already available algorithm in commercial transponders. In contrast, none of the existing DBP versions has reached the hardware level of implementation. In this context, this article shows that the NN-based equalizer achieves a performance similar to that obtained with the DBP [2] while approaching the complexity of the CDC block.

B. Quality of Transmission: Improvement Study

Fig. 10 summarizes the performance of the NN-based equalizers compared to 1 StpS DBP and CDC over different launch

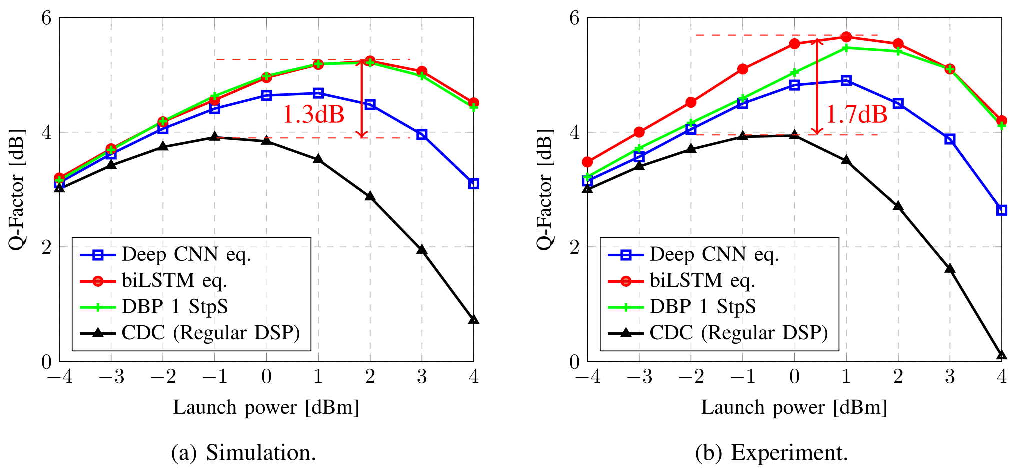


Fig. 10. Q-factor versus launch power for (a) simulation and (b) experiment corresponding to the transmission of an SC-DP 16QAM 34 GBd signal along 17 × 70 km of LEAF. The difference between the time domain CDC and the biLSTM equalizer's results is marked with red arrows. The case of the floating-point models' accuracy for the different types of NN equalizers (described in the legends), together with the 1StpS DBP and CDC performance curves.

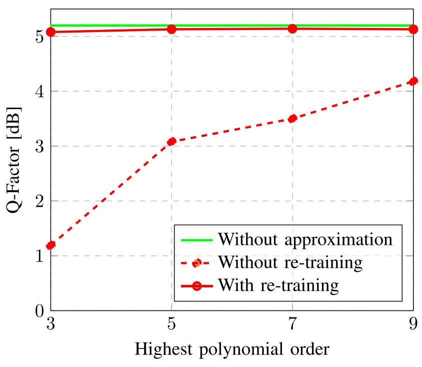
powers for simulated and experimental data. The results referring to the simulated data are given in Fig. 10(a). The biLSTM equalizer shows approximately the same performance as a 1 StpS DBP while improving the optimal power from −1 dBm to 2 dBm and the Q-factor by 1.3 dB with respect to the CDC. Regarding deep CNN, it performs worse than the biLSTM and the 1-StpS DBP; the optimal power is improved from −1 dBm to 1 dBm and the Q-factor increases by 0.8 dB compared to the CDC. On the other hand, with the experimental data,[10] we observe in Fig. 10(b) that the biLSTM outperforms the 1-StpS DBP, especially in the noise-dominated region. For the 1-StpS DBP case with the experimental data, the Q-factor increases by 1.3 dB in the simulation and by 1.5 dB in the experiment. Compared to simulation, NN-based equalizers in the experiment also lead to a higher gain for the Q-factor; when having CDC as a baseline, the gain improves from 1.3 dB (simulation) to 1.7 dB (experiment) in the case of biLSTM equalizers, and from 0.8 dB (simulation) to 1 dB (experiment) for the deep CNN. The optimal power also shifts from 0 dBm to 1 dBm in both cases. This shows that the NN has the potential to reduce the effects of both the Kerr nonlinearity and the component-induced corruptions which can be the effects of the transceivers (ADC/DAC or drive amplifier) and other effects that are not considered in the simulation such as some polarization mismatch, connector loss, different fiber parameters along the fiber, for both Tx and Rx sides.

In fact, all component impairments in the simulations were modeled with white noise (to represent the non-considered impairments from a real transmission and the non-ideal transceivers), so the equalizer could not mitigate them deterministically, whereas in the experimental case, our equalizer could enhance the Q-factor slightly further. Numerically, this can be observed by the fact that there is a 3 dB enhancement of the optimal launch power compared to the CDC in the simulation and only 1 dB in the experiment. This can be explained because, in the experiment, other effects apart from the Kerr effects were also compensated more in the linear regime, so a higher Q-factor was achieved but with a lower launch power. However, the maximum Q-factor after equalization in the simulation (5.24 dB at 2 dBm) is lower than the one achieved in the experiment for the same launch power (5.54 dB at 2 dBm). Hence, more linear impairments than nonlinear ones are recovered in the experiment, resulting in a smaller improvement in launch power. In particular, the biLSTM equalizer beats the deep CNN equalizer because the biLSTM is a recurrent-based NN that benefits from temporal sequential data learning [69], [70].

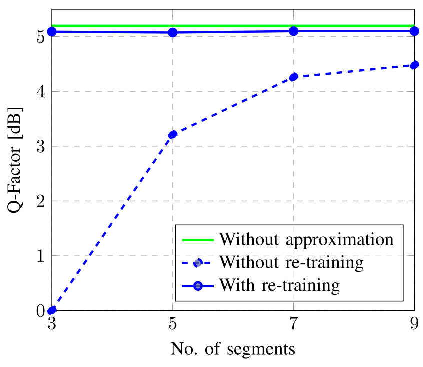
[10]The Q-factor obtained with the Python model and the FPGA implementation were virtually identical because we did not consider quantization.

C. Nonlinear Activation Function: Performance Versus Complexity Investigation

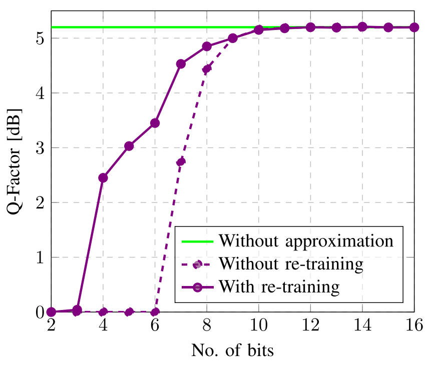
Now, having obtained the Q-factor benchmarks for the NN-based equalizers, we move on to the investigation of performance, studying different approximation techniques for nonlinear activation functions: Taylor series, PWL, and LUT. Fig. 11 depicts the Q-factor in the optimal power after equalization for three scenarios: the original NN without approximation, the NN with approximation (without re-training), and the NN with approximation (with re-training). Note that training the NN from scratch when replacing exact activation functions with approximations takes a considerably longer time to converge than retraining the original NN after replacing floating-point activation functions with their approximations. The training from scratch with the Taylor and LUT activations approximation even results in lower eventual performance. Therefore, in this figure, we only report the results of the retraining approach. Fig. 11(a) and (b), corresponding to the Taylor series and PWL, respectively, reveal the same trend. Without training, as the complexity of the approximation increases, the NN equalizer performs clearly better. However, with training, our increasing complexity barely improves the performance: the NN is able to adjust its parameters to mitigate the approximation error and provides comparable performance to the NN without approximations. The Q-factor versus complexity (order) of approximation plots, Fig. 11, highlight the remarkable performance gain



(a) Taylor series approximation.



(b) PWL approximation.



(c) LUT approximation.

Fig. 11. Q-factor versus complexity in terms of polynomial order for the Taylor approximation, pane (a), for the number of segments for PWL approximation, (b), and for the number of bits for the LUT, (c).

in all considered approaches when we re-train the model with activation functions replaced by the approximation, and we see that training can mitigate the errors from the approximation. This means that even the low-order approximations, such as the simplest PWL with three segments, can still yield results nearly identical to those rendered by the original activation functions.

Fig. 11(c) shows the performance of the LUT approximation. When replacing the activation functions with LUT without re-training, a certain number of bits is needed to provide an acceptable Q-factor level[11] For example, the minimum number of bits needed to provide a Q-factor greater than zero is 7 bits; 9 bits are needed to provide performance comparable to the model without approximation. On the other hand, when re-training the NN after the approximation, the Q-factor for the lower number of bits (from 3 to 7 bits) considerably increases. In this case, the non-differentiability makes the training challenging and limits the performance reachable in training, but the improvement is still noticeable when the number of bits is between 3 and 7. Fig. 12 shows the convergence speed of the three approximation techniques. It can be seen that the learning of Taylor and PWL is similar, whereas the re-training of LUT approximation is more difficult. Although the LUT gradient, Fig. 8(c), and the PWL gradient in Fig. 8(b) seem interchangeable, the forward propagation of the LUT approximation is still discrete, which means that with the lower number of bits we create a large gap between each quantized level. Thus, small changes that the gradient makes to update the weights might not change the quantization level to the next value. This means that the loss region is the same as it was in the last NN training interaction (trapped in a local minimum). Notably, in [71] a similar circumstance was observed; the previous reference also pointed to the instability of the training that can occur with a quantized activation function. In the case of PWL, the learning is more stable due to the continuity of the function's approximation, as

[11]Note that in this study, we followed the LUT implementation from Ref. [63], [64] which presented the LUT with equal x-error intervals. The alternative approach (activation functions with equal y-error intervals) can be used, but in our case, there is only a slight improvement in the Q-factor when the number of bits is greater than 5 and with the re-training, the performance is very close to the x-interval approach.

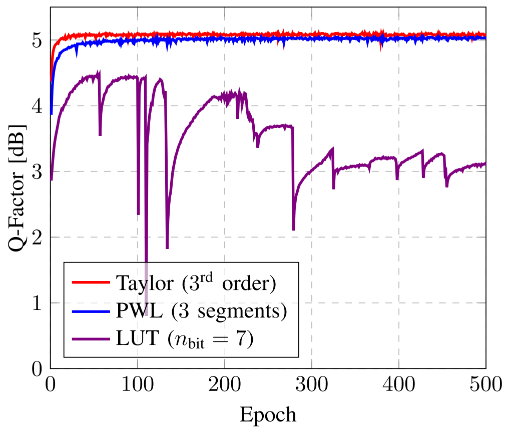


Fig. 12. Convergence study of the re-training to mitigate the approximation errors of Taylor series (3rd order), PWL (3 segments), and LUT ($n_{\text{bit}} = 7$) approximations.

each weight update generates a new loss value and a distinct point in the forward propagation.

In addition, as anticipated, we observe that when quantizing the LUT below 4 bits, no acceptable Q-factor can be reached even after the re-training. The reason for this is that when we quantize the activation function, unlike when we quantize the weights, we are limited in our ability to represent the modulation of the equalized signal. In our situation, we use 16QAM, which requires at least 4 bits to represent a constellation data point. However, as we see, even 4 bits are insufficient in this case to preserve all the essential features for the equalization process when using the quantization of the activation function.

When more bits are used, a better Q-factor can be achieved; however, more memory is then required to represent the quantization. It is worth noticing that when the number of bits is greater than 10, the Q-factor no longer improves in both scenarios (with and without re-training).

The amount of resources required (in terms of LUT, FF, and DSP slices) in the FPGA when using the approximations for tanh, is compared to that when applying the actual tanh activation function in Fig. 13. This figure depicts the resources used to build the logic behind the functionality of each approximation. Note

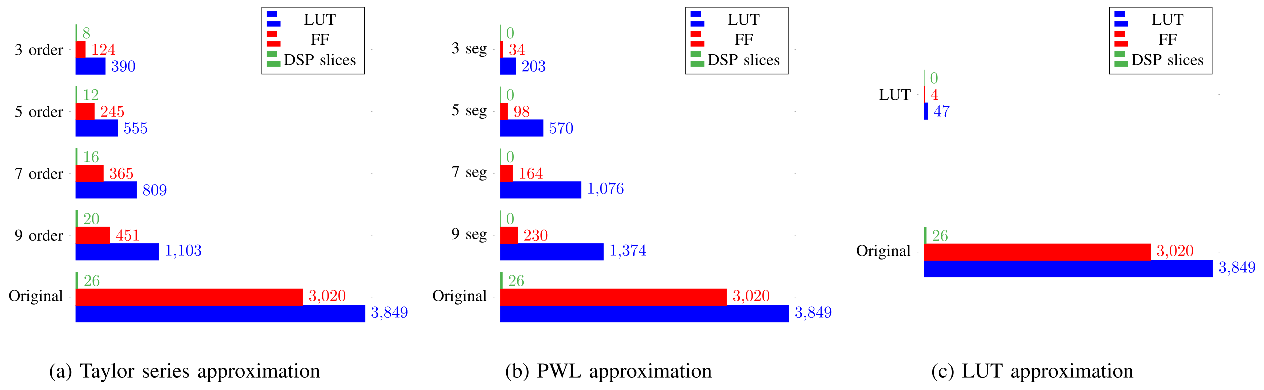


Fig. 13. Tanh implementation complexity in terms of LUT, FF, and DSP slices for the Taylor series, PWL, and LUT approximations after the Xilinx realization pipeline.

that the coefficients and values used in each panel of Fig. 4, are considered an input of the implemented box, which is accessed by the FPGA memory. The implementation complexity of the actual float activation functions is significantly higher than that of the approximations. In the Taylor series approximations, Fig. 13(a), the number of FF and LUT used to implement the approximations is drastically reduced compared to the original activation functions; to be more specific, when the highest order of the polynomial is 9, the number of FF required decreases by 6.7 times, and the number of LUT required is three times smaller. In terms of DSP slices, the 9th order approximation requires 6 DSP slices fewer than the original functions. As the approximation becomes simpler, the implementation requires fewer resources, as expected. For the PWL approximation, no usage of DSP slices is required for the implementation. Like in the Taylor series approximation, the number of FF and LUT required decreases noticeably. Compared to the original float-precision activation function, the PWL with 9 segments requires 2.8 times less LUT, and 13 times less FF. As the complexity decreases according to the number of segments, fewer resources are needed. Turning to the LUT approximation, it does not require any of the DSP slices as well, and the number of LUT and FF decreases by a factor of 80 and 775, respectively. Regardless of the number of bits in the quantized activation function, approximately the same amount of resources is required to implement the logic of the LUT approximation (see Fig. 13(c)). The LUT approximation approach is an algorithm based on evaluating the closest value in the LUT from a certain input and determining the memory address index that corresponds to that closest value to retrieve the information. As the number of bits increases, a larger memory is needed to store the LUT approximation points, which are a quantized version of the function. However, we do not account for this memory usage in our study because this is considered one of the inputs to our implemented box.

In conclusion, when performance, memory, and resources are considered, the PWL emerges as a viable candidate for hardware implementation, particularly, the 3-segment PWL variant with re-training. When the model learns to reduce approximation errors, the Q-factor of 3-segment PWL can reach a level comparable to that of the original activation functions; in addition, there is no need for DSP slices, resulting in more efficient use of resources than the Taylor approximation. With this, the RAM usage in the PWL is efficient because only a few coefficients of the approximation must be saved, whereas the LUT, which brings about difficulties during the re-training process, requires that all values of each quantization level be saved, resulting in an exponential increase in memory usage as the number of bits increases.

D. Computational Complexity Analysis for NN Equalizers Versus CDC, Implemented in Different Platforms

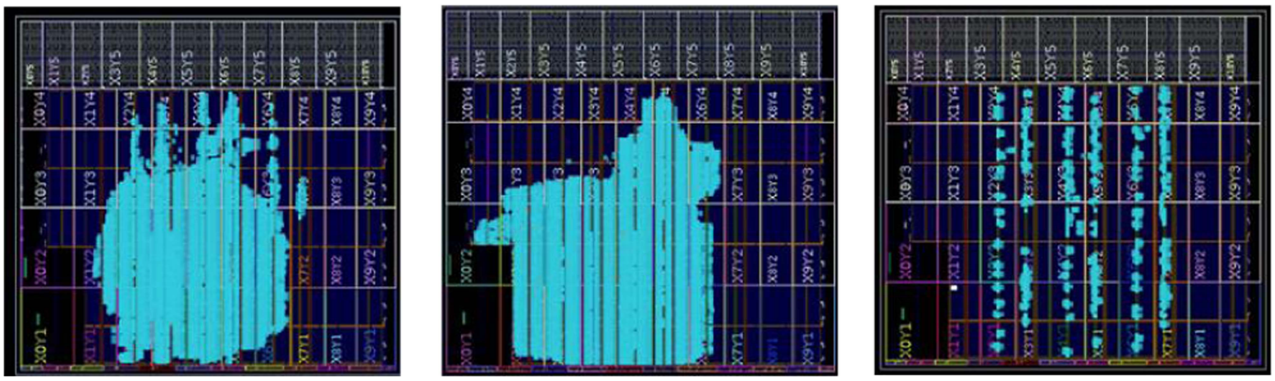
1) Standard FPGA Implementation (With DSP Slices): Fig. 14(a)–(c) show the real implementation and chip areas used for biLSTM, deep CNN equalizers, and CDC, respectively, on the state-of-the-art EK-VCK190-G-ED Xilinx FPGA chip [28] VCK190 kit features an AMD Xilinx Versal ACAP VC1902-2.[12] The device has 1968 DSP engines, 1799680 FFs, 899840 LUTs, and 400 AI engines. In this article, the AI engines are not used, as the HLS tool used in this work does not support the AI cores. The AI engines in the Versal AI Core FPGA are specialized units optimized for machine learning workloads and are not directly exposed to the programmer via the Vitis HLS tool[13]. This chip is partitioned into 40 clock regions, with the blue areas in Fig. 14 representing the used chip resources. Table I summarizes the most important information on the VCK 190 implementation of the biLSTM, deep CNN equalizers, and the CDC, in terms of latency, clock frequency [28], resources required, the utilization of the resources, and throughput. The achieved throughput $(\mathrm{T_a})$ can be calculated as follows:

$$\mathrm{T_a} = \text{clock} \times \log_2(\text{QAM}) \times n_{\text{out}}, \tag{10}$$

where clock is the clock frequency, QAM is the modulation format, $\log_2(\text{QAM})$ is the number of bits per symbol, and n_{out}, is the number of parallel symbols we recover in the output; in our case, $n_{\text{out}} = 61$.

[12]Note that "C" identifies it as a core series device and "−2" indicates the middle-speed grade.

[13]Pre-built AI libraries and IPs such as the Xilinx Deep Learning Processor (xDNN) library, which is optimized for DNNs, can be used for running on the FPGA's AI Engines. Note that the Vitis HLS tool can be used in conjunction with the xDNN library to target AI cores resources on Xilinx FPGAs [72], but this is outside our current scope.



(a) biLSTM eq. (b) Deep CNN eq. (c) CDC block

Fig. 14. Implementation in the EK-VCK190-G-ED Xilinx FPGA [28] of the (a) biLSTM eq., (b) Deep CNN eq., (c) CDC block - Time Domain.

TABLE I
VCK190 [28] IMPLEMENTATION

Type	Latency (μs)	Clock Frequency (MHz)	BRAM	SRL	DSP Slices	LUT	FF	Utilization (%) [DSP/LUT/FF]	Throughput ($Gbits/s$)	N^o FPGAs for 200G	N^o FPGAs for 400G [dual-carrier]	N^o FPGAs for 400G [56GBd]
biLSTM+CNN	33.4	270	164	109	1260	113532	224386	64.0/12.6/12.5	66	2.6	5.3	7.2
Deep CNN	19.9	244	0	125	582	118477	379829	29.6/13.2/21.1	60	1.4	2.7	3.7
CDC	1.1	524	0	1	1072	10441	5640	54.5/1.2/0.3	127	1.2	2.3	3.1

Three important conclusions can be drawn from that figure. First, although the biLSTM renders a higher Q-factor improvement, due to the equalizer's recurrent structure its feedback loop connections cause a bottleneck in the design, resulting in higher latency (33 μs) and lower clock frequency (270 MHz). On the other hand, deep CNN and CDC can be parallelized more efficiently. The parallelizability brings about a reduction in their latency to 19.9 μs for the deep CNN, and 1.1 μs for the CDC. Due to the fact that the CDC has one filter, whereas the deep CNN has 70 filters, the parallelization is easier in the CDC implementation because of hardware restrictions, resulting in an operating frequency of 524 MHz for the CDC case, and 244 MHz for the deep CNN case. Fig. 14(c) clearly shows the CDC parallelization. A long latency increases the time required to process each time step, leading to slower overall processing times and reducing the speed of the network. This can be problematic in real-time applications where a fast response is necessary. To mitigate the impact of long latency, design optimization techniques can be applied to reduce the latency and increase the performance, such as reducing the size of the memory blocks, using more efficient algorithms, and implementing pipelining. In this article, due to our offline processing consideration, all the input is already in the memory, and the request of the sequence with 81 symbols as inputs is created so that the functioning of the NN-based equalizer parallelization works.

Second, regarding the FPGA utilization, the biLSTM equalizer is the only one using Block Random Access Memory (BRAM)[14] to store future/past recurrent states, while both CNN and CDC do not need such blocks. BRAM is used to store the hidden states of an LSTM, which can then be fed back into the network at the next time step to maintain its memory, while in the case of the feedforward structures, the special LUTs are used to store the coefficients. The structure of the system of an LSTM cell is shown in Fig. 2, and the buffer used in the implementation was synthesized as BRAMs as global memory on the chip [51]. Note that the number shown in the tables is the number of BRAM blocks, which was the automatic result reported after the Synthesis step (Vivado). By using BRAM, the hidden state information can be stored in a dedicated memory block, separate from the other resources in the FPGA. This can lead to improved performance, as memory accesses are optimized and dedicated resources are used for memory storage. However, the size of the BRAM blocks and the memory requirements for the recurrent connections should be carefully considered when designing an LSTM on an FPGA. The available BRAM resources may be limited, and it may be necessary to trade off memory size for performance, depending on the requirements of the specific LSTM design. The SRL (Shift Register Look Up Table) in Tables I and II is a mode available in FPGAs whereby the LUT-RAM is configured as a shift register structure. This is more efficient, as it requires fewer cells and less routing than using individual DFFs to build shift registers. For the usage of DSP slices, LUT, and FF in each equalizer type, the biLSTM requires 64% DSP slices and 13% of LUT and FF, the deep CNN uses 30% DSP slices, 13% of LUT and 21% of FF, and the CDC needs 54% DSP slices and 1% of LUT and FF.

Third, in terms of throughput, the clock frequency is the maximum that each implementation can handle to comply with a zero-negative slack design. In this sense, the total throughput for the 16QAM modulation format is 66 G, 60 G, and 127 G, for the biLSTM, deep CNN, and CDC block, respectively.

[14]BRAM is a type of memory in FPGAs that is used to store large amounts of data, typically used in FPGA designs to implement memory-intensive functions such as image and video processing, buffers, and large arrays. Unlike other memory elements in an FPGA, BRAM is a dedicated memory that is separate from the FPGA's general-purpose FFs and LUTs.

TABLE II
IMPLEMENTATION WHERE ALL MULTIPLICATIONS ARE DONE USING LUT AND FF

Type	Latency (μs)	Clock Frequency (MHz)	BRAM	SRL	LUT	FF	Utilization (%) [LUT/FF]	Throughput ($Gbits/s$)	N^o FPGAs for 200G	N^o FPGAs for 400G [dual-carrier]	N^o FPGAs for 400G [56GBd]
biLSTM+CNN	39.2	234	164	163	566070	249763	62.9/13.9	57	3.0	6.0	8.1
Deep CNN	17	245	0	162	300426	399868	33.4/22.3	60	1.5	3.0	4.1
CDC	2.3	246	0	1	418534	24968	46.5/1.4	60	2.1	4.2	5.7

Lastly, regarding the calculation of the number of equivalent FPGAs for a certain target throughput ($\mathrm{T_{target}}$) from an experiment, we have considered the following equation:

$$\mathrm{N_{FPGA}} = \frac{\mathrm{T_{target}}}{\mathrm{T_a}} * U_t, \tag{11}$$

where $\mathrm{T_a}$ is the throughput achieved after the NN design pipeline, and U_t is the maximum utilization after the NN design pipeline (both reported in Table I). In the CNN+biLSTM case, for example, because the experiment was 16QAM single carrier transmission at both 34 GB per pol, the target throughput is 272 Gbit/200 G. So, considering the maximum utilization of 64% and the throughput achieved of 65.9Gbit by the NN equalizer, $\mathrm{N_{FPGA}}$ would be equal to 2.6 FPGAs.

In fact, we considered the FPGA estimation for three different cases:

1) 200G scenario: which is the scenario of the experiment in this paper - 16QAM single carrier configurations at both 34GBd per pol (resulting in 272Gbit/200 G).
2) 400G scenario: considering a dual carrier transmission instead of a single carrier. In this case, we just need to scale the resources of 200G by a factor of 2.
3) 400G scenario: considering a 16QAM single carrier configurations with higher symbol rate equal to 56GBd per pol (resulting in 448Gbit, with 12% FEC overhead). In this case, we simply multiply the 200G resources by $56^2/34^2 = 2.71$ because, given the increased symbol rate, the resources will grow approximately quadratically with the increase in symbol rate since our implementations were in the time domain.

For case 1, we observe that 200G transmission can be achieved using an equivalent FPGA that has the same capacity as $\approx$ 3 FPGAs (VCK190) in the case of biLSTM, $\approx$ 2 FPGAs in a deep CNN case, and $\approx$ 1 FPGAs for CDC. For case 2, 400G with dual carrier transmission can be achieved using an equivalent FPGA that has the same capacity as $\approx$ 5 FPGAs (VCK190) in the case of biLSTM, $\approx$ 3 FPGAs in a deep CNN case, and $\approx$ 2 FPGAs for CDC. Finally, in case 3, because of the increase in symbol rate, much more hardware was needed. For the 400G with 56Gbd transmission case, we would need an equivalent FPGA that has the same capacity as $\approx$ 7 FPGAs (VCK190) in the case of biLSTM, $\approx$ 4 FPGAs in a deep CNN case, and $\approx$ 3 FPGAs for CDC. In all three cases, biLSTM used approximately 2.5 times more FPGA than required by the CDC implementation.

2) ASIC Equivalent Implementation (no DSP Slices): Unlike FPGAs, ASICs are application specific, and their digital circuitry contains permanently connected gates and FF in silicon; therefore, in ASIC design, there is no configurable block (such as DSP blocks). In this subsection, we evaluate the approximations of the resource requirements and the performance in terms of throughput, clock frequency, and latency of the ASIC implementation by considering the VCK190 implementation without the usage of DSP slices. Table II contains information for the implementation of the biLSTM, the deep CNN, and the CDC equalizer on the FPGA with only LUT and FF. The biLSTM and CDC have a noticeably higher latency: 5.8 μs and 1.2 μs higher, respectively, compared to implementation with DSP slices detailed in Table I. The lower clock frequency is also observed: 234 MHz for biLSTM and 246 MHz for the CDC, resulting in a lower throughput: 57G for biLSTM and 60G for the CDC. The degradation in throughput, latency, and clock frequency highlights the fact that DSP slices speed up the execution of signal processing functions. Especially in the CDC, when we allow implementation with DSP slices, the number of LUT and FF used is 40 times and 4.4 times less, respectively. Therefore, we can clearly see the degradation of performance in terms of the throughput of the CDC when the DSP slices are not used. However, in the case of deep CNN, the latency decreases by 2 μs, the clock frequency increases by 1 MHz, and the throughput remains unchanged. We can observe that in the deep CNN implementation with the DSP slices in Table I, the number of DSP slices used is only about half that for the other two equalizers, and the number of LUT and FF is highest. Therefore, our removal of DSP slices did not affect the deep CNN because the processing time for LUT and FF was already the bottleneck in the previous implementation. Here, it is essential to recall that the clock frequency, which is essential to establishing the throughput, is chosen to guarantee zero negative timing slack. In this regard, since the routing and mapping are performed automatically by the Vivado platform, we can observe that when restricting the software from using the DSP slices, longer paths are generated during the synthesis in the biLSTM and CDC cases. The longer paths cause the clock frequency to decrease to achieve the zero-negative slack level. In contrast, the deep CNN's paths when deploying the DSP slices are already long due to the logic implementation and synthesis, and, therefore, there is no significant variation in clock frequency after removing the DSP slice in that case.

Regarding the utilization of LUT and FF: the biLSTM uses 62.9% of LUT and 13.9% of FF, the deep CNN uses 33.4% of LUT and 22.3% of FF, and the CDC uses 46.5% of LUT and 1.4% of FF. The utilization of LUT for all three equalizers is considerably increased compared to the previous case (standard FPGA implementation). As the number of LUTs and FFs increases, the equivalent number of FPGAs used to represent the biLSTM and the CDC equalization also grows.

For the same three cases we have discussed previously, we could observe the following. For case 1, 200G transmission can be achieved using an equivalent FPGA that has the same capacity as ≈ 3 FPGAs (VCK190) in the case of biLSTM, ≈ 2 FPGAs in a deep CNN case, and ≈ 2 FPGAs for CDC. For case 2, 400G with dual carrier transmission can be achieved using an equivalent FPGA that has the same capacity as ≈ 6 FPGAs (VCK190) in the case of biLSTM, ≈ 3 FPGAs in a deep CNN case, and ≈ 4 FPGAs for CDC. Eventually, in case 3 with a 16 QAM 56Gbd transmission, we would need an equivalent FPGA that has the same capacity as ≈ 8 FPGAs (VCK190) in the case of biLSTM, ≈ 4 FPGAs in a deep CNN case, and ≈ 6 FPGAs for CDC. In all three cases, biLSTM used approximately 1.5 times more FPGA than required by the CDC implementation.

Finally, we note that in this study, we established the approximate resources and performance of the equalizers implemented on ASIC by excluding the DSP slices. However, this is still not an optimized realization: in ASIC implementation, the number of resources used needs to be further optimized to reduce the utilization, increase the clock frequency, and enable high-speed processing.

VI. Conclusion and Open Challenges

In this article, we carry out a detailed study of the design of NN-based optical equalizers, addressing the steps from Python realization to FPGA implementation. To approach the real hardware implementation of NN-based equalizers, we investigated three approximation approaches (Taylor, piecewise linear, and lookup table) for nonlinear activation functions, aiming at reducing the computational complexity. The complexity, performance, and resources required for the approximations have been evaluated. We then examined the biLSTM equalizer implementation on the FPGA, assessing the complexity reduction due to the implementation using fixed-point arithmetic and nonlinear activation function approximations. Our realization showed that the biLSTM requires only ≈ 2.5 times more FPGA resources than the CDC implemented in the time domain, while still outperforming the 1-StpS DBP in Q-factor. The approximate utilization of ASIC when using only FF and LUT to implement the logic of such DSP blocks for channel equalization were also considered. The results obtained for the ASIC estimation showed a drop in throughput for the biLSTM equalizer, due to the challenges in route design to achieve zero negative slack using higher than 234 Mhz clock frequencies. The latter indicates that, for future applications, much more effort is still required in this direction.

We consider this article to be yet another piece of evidence that the deployment of NN-based equalizers in commercial applications might become a reality. Indeed, it is already quite clear that the NN-based equalizers can provide significant performance improvements when implemented on top of the existing DSP algorithms. The NN can even replace some DSP chain parts, such as the CDC block. Moreover, as we evaluated in this article, the real-time hardware implementation of NNs is already an attainable reality. Unfortunately, the complexity of the proposed implementation is still too high for NN equalization deployment in commercial optical coherent transponders. Additionally, one of the most critical aspects of the DSP block of a coherent transponder is its power efficiency. Since power efficiency is a direct consequence of the complexity, further investigations focusing on the reduction of NN's complexity (when those are specifically implemented in hardware) are important. These investigations should address several areas, including the simplification methods for NN structures, such as pruning, weight sharing, quantization, and the respective hardware implementation aspects.

To reduce complexity, approaches concentrating on different transmission scenarios and, consequently, adopting different DSP systems, may be envisioned. Core and regional optical networks are characterized by quite long optical links, where optical signals experience noticeable nonlinear distortions and the accumulated chromatic dispersion is large enough. Access and metro optical networks are characterized by short propagation distances, in which the predominant transmission effects are typically the ASE noise and limited optical power at the RX input (e.g., in the point-to-multipoint solutions). An important standpoint from the industry is how the implemented NN (particularly, the simplification and complexity reduction strategies) varies with the change of transmission scenario. If the low-complexity implementation of a given NN works well in one situation but not in another, this can pose a serious difficulty, as we often cannot afford to produce a unique chip for every circumstance. It is desirable that the NN after pruning and quantization is still capable to equalize versatile transmission setups (working, e.g., for different fiber types, span lengths, launch powers, etc.)

Finally, we list the unresolved questions that were not investigated in our work but can be crucial for further research.

- Power consumption evaluation of reduced-complexity NN equalizers.
- How to realize a NN that can work for multiple transmission scenarios with no or very limited retraining.
- Parallelization of the recurrent NN structures study.
- Implementation in the FPGA of more robust quantization levels moving from int32, as presented in this manuscript, to int8 or less, if possible, by using heterogeneous quantization together with quantization-aware training [73].
- Further flexibility analysis to avoid the need to retrain the hardware NN implementation, e.g., the hardware tests of domain adaptation/randomization and transfer learning [47], [74].

Appendix A
FPGA Notation

Offline FPGA implementation refers to the process of designing and configuring an FPGA before it is deployed in a target system. This typically involves using specialized software tools to design, simulate, and verify the digital logic that will be implemented on the FPGA. The final design is then converted into a format that can be loaded onto the FPGA, such as a bitstream file.

TABLE III
PWL Approximation Equations of Sigmoid and Tanh for 3, 5, 7 and 9 Segments

No. of segments	Functions			
	tanh		sigmoid	
	Equation	Condition	Equation	Condition
3	1	$x > 1.1$	1	$x > 2.2$
	$0.90909x$	$-1.1 < x \leq 1.1$	$0.22727x + 0.5$	$-2.2 < x \leq 2.2$
	-1	$x \leq -1.1$	0	$x \leq -2.2$
5	1	$x > 1.7$	1	$x > 2.6$
	$0.41666x + 0.29166$	$0.5 < x \leq 1.7$	$0.17223x + 0.55219$	$0.8 < x \leq 2.6$
	x	$-0.5 < x \leq 0.5$	$0.23747x + 0.5$	$-0.8 < x \leq 0.8$
	$0.41666x - 0.29166$	$-1.7 < x \leq -0.5$	$0.17223x + 0.44781$	$-2.6 < x \leq -0.8$
	-1	$x \leq -1.7$	0	$x \leq -2.6$
7	1	$x > 1.8$	1	$x > 3$
	$0.285x + 0.48699$	$1.1 < x \leq 1.8$	$0.12363x + 0.62909$	$1.4 < x \leq 3$
	$0.57214x + 0.17114$	$0.4 < x \leq 1.1$	$0.18701x + 0.54036$	$0.8 < x \leq 1.4$
	x	$-0.4 < x \leq 0.4$	$0.23747x + 0.5$	$-0.8 < x \leq 0.8$
	$0.57214x - 0.17114$	$-1.1 < x \leq -0.4$	$0.18701x + 0.45964$	$-1.4 < x \leq -0.8$
	$0.285x - 0.48699$	$-1.8 < x \leq -1.1$	$0.12363x + 0.37091$	$-3 < x \leq -1.4$
	-1	$x \leq -1.8$	0	$x \leq -3$
9	1	$x > 2.2$	1	$x > 3.4$
	$0.14331x + 0.68417$	$1.4 < x \leq 2.2$	$0.08514x + 0.71051$	$2 < x \leq 3.4$
	$0.3381x + 0.412$	$0.9 < x \leq 1.4$	$0.12644x + 0.62791$	$1.5 < x \leq 2$
	$0.269382x + 0.09185$	$0.3 < x \leq 0.9$	$0.182242x + 0.09185$	$0.8 < x \leq 1.5$
	x	$-0.3 < x \leq 0.3$	$0.23747x + 0.5$	$-0.8 < x \leq 0.8$
	$0.269382x - 0.09185$	$-0.9 < x \leq -0.3$	$0.08514x + 0.45585$	$-1.5 < x \leq -0.8$
	$0.3381x - 0.412$	$-1.4 < x \leq -0.9$	$0.12644x + 0.37209$	$-2 < x \leq -1.5$
	$0.14331x - 0.68417$	$-2.2 < x \leq -1.4$	$0.182242x + 0.28949$	$-3.4 < x \leq -2$
	-1	$x \leq -2.2$	0	$x \leq -3.4$

A D-type Flip-Flop (DFF) is a type of sequential logic element that is commonly deployed in digital systems as they are simple. DFFs can be found in digital state machines, shift registers, counters, and other digital circuits that require memory storage. They are often used to store the value of a digital signal and can be used with other logic elements, such as AND and OR gates, to build more complex digital systems.

BRAM (Block Random Access Memory) is a type of memory available in FPGAs that is used to store data. Unlike other memory elements in an FPGA, BRAM is dedicated memory that is separate from the FPGA's general-purpose flip-flops and LUTs. BRAM is used to store large amounts of data, typically used in FPGA designs to implement memory-intensive functions such as image and video processing, buffers, and large arrays.

RAM-based LUT (LUT RAM) or distributed RAM is sometimes called in the user guides as the RAM used to store the logic function equations. LUT RAM can also be configured to be used as user storage with a similar function to BRAM.

Digital Signal Processing (DSP) Blocks or Slices: Digital Signal Processing (DSP) blocks or slices are specialized components within an FPGA that are designed specifically for processing digital signals. They contain dedicated hardware resources such as multipliers, adders, accumulators, and registers that can perform complex mathematical operations at high speeds. They are optimized for efficient use of resources and can perform operations in parallel, which enables high-speed processing of large volumes of data. Additionally, DSP blocks are typically designed to support fixed-point and floating-point arithmetic, and they can be configured to support various data widths and precision. They can also be combined with other components within an FPGA to create complex signal processing pipelines.

Appendix B
PWL Equations

The equations of the PWL approximations of sigmoid and tanh can be found in Table III for 3, 5, 7, and 9 segments.

References

[1] F. P. Guiomar, J. D. Reis, A. L. Teixeira, and A. N. Pinto, "Mitigation of intra-channel nonlinearities using a frequency-domain volterra series equalizer," *Opt. Exp.*, vol. 20, no. 2, pp. 1360–1369, 2012.

[2] 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.

[3] E. Ip and J. M. Kahn, "Compensation of dispersion and nonlinear impairments using digital backpropagation," *J. Lightw. Technol.*, vol. 26, no. 20, pp. 3416–3425, Oct. 2008.

[4] L. Liu et al., "Intrachannel nonlinearity compensation by inverse volterra series transfer function," *J. Lightw. Technol.*, vol. 30, no. 3, pp. 310–316, Feb. 2012.

[5] M. Sorokina, S. Sygletos, and S. Turitsyn, "Sparse identification for nonlinear optical communication systems: Sino method," *Opt. Exp.*, vol. 24, no. 26, pp. 30433–30443, 2016.

[6] M. Sorokina, S. Sygletos, and S. Turitsyn, "Ripple distribution for nonlinear fiber-optic channels," *Opt. Exp.*, vol. 25, no. 3, pp. 2228–2238, 2017.

[7] S. K. Turitsyn et al., "Nonlinear fourier transform for optical data processing and transmission: Advances and perspectives," *Optica*, vol. 4, no. 3, pp. 307–322, 2017.

[8] S. M. Ranzini et al., "Digital back-propagation ASIC design for high-speed coherent optical system," in *Proc. SBMO/IEEE MTT-S Int. Microw. Optoelectron. Conf.*, 2015, pp. 1–5.

[9] A. Vasylchenkova, D. Salnikov, D. Karaman, O. G. Vasylchenkov, and J. E. Prilepskiy, “Fixed-point realisation of fast nonlinear fourier transform algorithm for FPGA implementation of optical data processing,” *Proc. SPIE*, vol. 11770, pp. 111–120, 2021.

[10] C. Fougstedt, L. Svensson, M. Mazur, M. Karlsson, and P. Larsson-Edefors, “ASIC implementation of time-domain digital back propagation for coherent receivers,” *IEEE Photon. Technol. Lett.*, vol. 30, no. 13, pp. 1179–1182, Jul. 2018.

[11] T. Sherborne, B. Banks, D. Semrau, R. I. Killey, P. Bayvel, and D. Lavery, “On the impact of fixed point hardware for optical fiber nonlinearity compensation algorithms,” *J. Lightw. Technol.*, vol. 36, no. 20, pp. 5016–5022, Oct. 2018.

[12] 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.

[13] D. Wang et al., “System impairment compensation in coherent optical communications by using a bio-inspired detector based on artificial neural network and genetic algorithm,” *Opt. Commun.*, vol. 399, pp. 1–12, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0030401817303346

[14] C. Häger and H. D. Pfister, “Nonlinear interference mitigation via deep neural networks,” in *Proc. Opt. Fiber Commun. Conf. Expo.*, 2018, pp. 1–3.

[15] C. Huang et al., “Demonstration of photonic neural network for fiber nonlinearity compensation in long-haul transmission systems,” in *Proc. IEEE Opt. Fiber Commun. Conf. Exhib.*2020, pp. 1–3.

[16] B. I. Bitachon, A. Ghazisaeidi, M. Eppenberger, B. Baeuerle, M. Ayata, and J. Leuthold, “Deep learning based digital backpropagation demonstrating snr gain at low complexity in a 1200 km transmission link,” *Opt. Exp.*, vol. 28, no. 20, pp. 29318–29334, Sep. 2020. [Online]. Available: http://www.opticsexpress.org/abstract.cfm?URI=oe-28-20-29318

[17] Y. Zhao et al., “Low-complexity fiber nonlinearity impairments compensation enabled by simple recurrent neural network with time memory,” *IEEE Access*, vol. 8, pp. 160995–161004, 2020.

[18] M. M. Melek and D. Yevick, “Nonlinearity mitigation with a perturbation based neural network receiver,” *Opt. Quantum Electron.*, vol. 52, no. 10, pp. 1–10, 2020.

[19] S. Zhang et al., “Field and lab experimental demonstration of nonlinear impairment compensation using neural networks,” *Nature Commun.*, vol. 10, no. 1, pp. 1–8, 2019.

[20] P. J. Freire et al., “Complex-valued neural network design for mitigation of signal distortions in optical links,” *J. Lightw. Technol.*, vol. 39, no. 6, pp. 1696–1705, Mar. 2021.

[21] M. Schaedler, F. Pittala, G. Böcherer, C. Bluemm, M. Kuschnerov, and S. Pachnicke, “Recurrent neural network soft-demapping for nonlinear isi in 800Gbit/s dwdm coherent optical transmissions,” in *Proc. 46th Eur. Conf. Opt. Commun.*, 2020, pp. 1–4.

[22] J. Song et al., “Real-time FPGA prototyping of a 15GBaud SP-16QAM coherent optical receiver with optimal interpolating for clock recovery and equalization,” *Opt. Exp.*, vol. 30, no. 15, pp. 26774–26786, 2022.

[23] M. S. Faruk and S. J. Savory, “Digital signal processing for coherent transceivers employing multilevel formats,” *J. Lightw. Technol.*, vol. 35, no. 5, pp. 1125–1141, Mar. 2017.

[24] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[25] H. Nakahara, Z. Que, and W. Luk, “High-throughput convolutional neural network on an FPGA by customized JPEG compression,” in *Proc. IEEE 28th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2020, pp. 1–9.

[26] C. T. Yen, W.-D. Weng, and Y. T. Lin, “FPGA realization of a neural-network-based nonlinear channel equalizer,” *IEEE Trans. Ind. Electron.*, vol. 51, no. 2, pp. 472–479, Apr. 2004.

[27] T. Xu et al., “Chromatic dispersion compensation in coherent transmission system using digital filters,” *Opt. Exp.*, vol. 18, no. 15, pp. 16243–16257, 2010.

[28] “VCK190 evaluation board,” 2022. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/boards_and_kits/vck190/ug1366-vck190-eval-bd.pdf

[29] L. Pettersson, “Convolutional neural networks on FPGA and GPU on the edge: A comparison,” Master Programme in Engineering Physics, Uppsala University, Disciplinary Domain of Science and Technology, Technology, Department of Electrical Engineering, Signals and Systems, 2020.

[30] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, “Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC,” in *Proc. IEEE 26th Int. Conf. Field Programmable Log. Appl.*, 2016, pp. 1–4.

[31] J. Qiuet al., “Going deeper with embedded FPGA platform for convolutional neural network,” in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[32] Z. Hajduk, “Reconfigurable FPGA implementation of neural networks,” *Eurocomputing*, vol. 308, pp. 227–234, 2018.

[33] A. X. M. Chang, B. Martini, and E. Culurciello, “Recurrent neural networks hardware implementation on FPGA,” 2015, *arXiv:1511.05552*.

[34] X. Huang et al., “Implementation and research of lstm neural network based on the FPGA,” *J. Electron. Inf. Sci.*, vol. 2, no. 2, pp. 76–79, 2017.

[35] A. X. M. Chang and E. Culurciello, “Hardware accelerators for recurrent neural networks on FPGA,” in *Proc. IEEE Int. Symp. Circuits Syst.*, 2017, pp. 1–4.

[36] A. Dhavlle and S. M. P. Dinakarrao, “A comprehensive review of ML-based time-series and signal processing techniques and their hardware implementations,” in *Proc. IEEE 11th Int. Green Sustain. Comput. Workshops*, 2020, pp. 1–8.

[37] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, “FPGAs in industrial control applications,” *IEEE Trans. Ind. Informat.*, vol. 7, no. 2, pp. 224–243, May 2011.

[38] T. Orlowska-Kowalska and M. Kaminski, “FPGA implementation of the multilayer neural network for the speed estimation of the two-mass drive system,” *IEEE Trans. Ind. Informat.*, vol. 7, no. 3, pp. 436–445, Aug. 2011.

[39] N. Kaneda et al., “FPGA implementation of deep neural network based equalizers for high-speed PON,” in *Proc. Opt. Fiber Commun. Conf., Exhib.*, 2020, pp. 1–3.

[40] N. Kaneda et al., “Fixed-point analysis and FPGA implementation of deep neural network based equalizers for high-speed PON,” *J. Lightw. Technol.*, vol. 40, no. 7, pp. 1972–1980, Apr. 2022.

[41] 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.

[42] M. Li, W. Zhang, and Z. He, “FPGA implementation of time-interleaved pruning neural network equalizer for short reach optical interconnects,” in *Proc. IEEE Asia Commun. Photon. Conf.*, 2021, pp. 1–3.

[43] E. Giacoumidis, Y. Lin, M. Blott, and L. P. Barry, “Real-time machine learning based fiber-induced nonlinearity compensation in energy-efficient coherent optical networks,” *APL Photon.*, vol. 5, no. 4, 2020, Art. no. 041301.

[44] P. Chen, H. Tsai, C. Lin, and C. Lee, “FPGA realization of a radial basis function based nonlinear channel equalizer,” in *Proc. Int. Symp. Neural Netw.*, 2005, pp. 320–325.

[45] J. Lee, J. He, and K. Wang, “Neural networks and FPGA hardware accelerators for millimeter-wave radio-over-fiber systems,” in *Proc. IEEE 22nd Int. Conf. Transparent Opt. Netw.*, 2020, pp. 1–4.

[46] A. Gulli and S Pal, *Deep Learning With Keras*. Birmingham, U.K.:Packt Pub. Ltd, 2017.

[47] 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.

[48] J. Caba, F. Rincón, J. Barba, J. A. De La Torre, J. Dondo, and J. C. López, “Towards test-driven development for FPGA-based modules across abstraction levels,” *IEEE Access*, vol. 9, pp. 31581–31594, 2021.

[49] P. J. Freire et al., “Reducing computational complexity of neural networks in optical channel equalization: From concepts to implementation,” *J. Lightw. Technol.*, 2023, early access, Jan. 05, 2023, doi: 10.1109/JLT.2023.3234327.

[50] D. Baptista, F. Morgado-Dias, and L. Sousa, “A platform based on HLS to implement a generic CNN on an FPGA,” in *Proc. IEEE Int. Conf. Eng. Appl.*, 2019, pp. 1–7.

[51] D. He, J. He, J. Liu, J. Yang, Q. Yan, and Y. Yang, “An FPGA-based LSTM acceleration engine for deep learning frameworks,” *Electronics*, vol. 10, no. 6, 2021, Art. no. 681.

[52] “Xilinx UG1399 vitis HLS,” 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[53] “Strategy implemnetation vivado,” 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug904-vivado-implementation/Strategy

[54] A. Lopez-Gasso, “What are the best vivado synthesis and implementation strategies???,” Apr. 2021. [Online]. Available: https://miscircuitos.com/vivado-synthesis-and-implementation-strategies

[55] O. Çetin, F. Temurtaş, and Ş. Gülgönül, "An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function," *Dicle Tıp Dergisi*, vol. 42, no. 2, pp. 150–157, 2015.

[56] Z. Li, Y. Zhang, B. Sui, Z. Xing, and Q. Wang, "FPGA implementation for the sigmoid with piecewise linear fitting method based on curvature analysis," *Electronics*, vol. 11, no. 9, 2022, Art. no. 1365.

[57] N. G. Timmons and A. Rice, "Approximating activation functions," 2020, *arXiv:2001.06370*.

[58] F. Temurtas, A. Gulbag, and N. Yumusak, "A study on neural networks using taylor series expansion of sigmoid activation function," in *Proc. Int. Conf. Comput. Sci. Appl.*, 2004, pp. 389–397.

[59] M. R. Spiegel, *Mathematical Handbook of Formulas and Tables*. New York, NY, USA: McGraw-Hill, 1968.

[60] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proc.-Circuits, Devices, Syst.*, vol. 144, no. 6, pp. 313–317, 1997.

[61] K. Basterretxea, J. M. Tarela, and I. del Campo, "Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons," *IEE Proc.-Circuits, Devices, Syst.*, vol. 151, no. 1, pp. 18–24, 2004.

[62] M. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proc.-Comput. Digit. Techn.*, vol. 150, no. 6, pp. 403–411. 2003.

[63] T. Yang et al., "Design space exploration of neural network activation function circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 10, pp. 1974–1978, Oct. 2019.

[64] A. H. Namin, K. Leboeuf, H. Wu, and M. Ahmadi, "Artificial neural networks activation function HDL coder," in *Proc. IEEE Int. Conf. Electro/Inf. Technol.*, 2009, pp. 389–392.

[65] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, "Efficient hardware implementation of the hyperbolic tangent sigmoid function," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2009, pp. 2117–2120.

[66] L. Wang, X. Dong, Y. Wang, L. Liu, W. An, and Y. Guo, "Learnable lookup table for neural network quantization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2022, pp. 12423–12433.

[67] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[68] M. Kuschnerov et al., "Data-aided versusblind single-carrier coherent receivers," *IEEE Photon. J.*, vol. 2, no. 3, pp. 387–403, Jun. 2010.

[69] 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.

[70] S. Deligiannidis, A. Bogris, C. Mesaritakis, and Y. Kopsinis, "Compensation of fiber nonlinearities in digital coherent systems leveraging long short-term memory neural networks," *J. Lightw. Technol.*, vol. 38, no. 21, pp. 5991–5999, Nov. 2020.

[71] P. Yin, J. Lyu, S. Zhang, S. Osher, Y. Qi, and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets," 2019, *arXiv:1903.05662*.

[72] Xilinx, "Vitis-tutorials," 2021. [Online]. Available: https://github.com/Xilinx/Vitis-Tutorials/tree/2021.2/AI_Engine_Development

[73] P. J. Freire et al., "Reducing computational complexity of neural networks in optical channel equalization: From concepts to implementation," 2022, *arXiv:2208.12866*.

[74] P. J. Freire et al., "Domain adaptation: The key enabler of neural network equalizers in coherent optical systems," in *Proc. Opt. Fiber Commun. Conf. Exhib.*, 2022, pp. 1–3.