LuxIA: A Lightweight Unitary Matrix-Based Framework Built on an Iterative Algorithm for Photonic Neural Network Training

(Article begins on next page)

LuxIA: A Lightweight Unitary matriX-based Framework Built on an Iterative Algorithm for Photonic Neural Network Training

Tzamn Melendez Carmona, Federico Marchesin, Marco P. Abrate, Peter Bienstman *Member, IEEE,*, Stefano Di Carlo *Senior, IEEE,* Alessandro Savino *Senior, IEEE*

***Abstract*—Photonic Neural Networks (PNNs) can accelerate machine learning workloads by implementing Matrix-Vector Multiplications (MVMs) in integrated photonic circuits, but existing simulation and training frameworks scale poorly to large Photonic Unitary Matrix (PUM) meshes because they explicitly construct and manipulate dense transfer matrices. This work introduces the Slicing method, which models PUM meshes as sequences of local 2×2 operations organized into computational windows and computes forward and backward propagation using only localized matrix–vector updates with linear complexity in the number of active cells. The method is implemented in LuxIA, an open-source PyTorch-based framework for end-to-end PNN simulation and training that supports multiple mesh architectures and datasets. A formal analysis shows that Slicing reduces per-pass work by one degree (from quartic to cubic) in the worst-case. Experiments on Clements, Fldzhyan, and Bell-optimized meshes trained on Iris, Digits, MNIST, and Olivetti Faces show that LuxIA matches the training dynamics and task accuracy of existing tools while substantially improving training efficiency: on large meshes and batches, LuxIA achieves up to $4.7\times$ lower training time and more than an order-of-magnitude reduction in Graphics Processing Unit (GPU) memory compared with conventional transfer-matrix frameworks, and it remains within the memory budget where competing tools fail.**

***Index Terms*—Edge-computing, photonics, simulation tools, neural networks, artificial intelligence.**

I. Introduction

THE rapid expansion of intelligent technologies across industries and daily life is driven by a growing ecosystem of dedicated Complementary Metal-Oxide-Semiconductor (CMOS)-based hardware accelerators offering increasing computing power to support advances in Artificial Intelligence (AI) [1]. However, CMOS scaling has been slowing down for several years, making further performance gains increasingly difficult [2]. Silicon photonics has emerged as a promising candidate technology to sustain the growth of computing performance, offering the ability to leverage the unique properties of light to perform computation. In particular, it has demonstrated superior energy efficiency and speed compared to traditional CMOS circuits when executing core operations required by AI workloads such as Matrix-Vector Multiplication (MVM) [3].

Although photonic AI accelerators remain in an early stage of development, their potential continues to grow. Ongoing research steadily advances toward real-world applications with the introduction of Photonic Neural Networks (PNNs) [4], a new class of Neural Networks (NNs) in which key computing primitives, such as MVM, are performed fully or partially using light (photons) instead of electricity [5], [6]. Several works have successfully demonstrated photonic implementations of various types of NNs, including Feed-Forward Neural Networks (FFNNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs), operating fully or partially in the optical domain [5], [7].

The core of many PNNs lies in the ability to perform efficient MVM operations in the optical domain. To this end, the Photonic Unitary Matrix (PUM) mesh is a key enabling component. The PUM mesh is a photonic circuit with programmable optical elements. The first PUM mesh was an array of Mach-Zehnder Interferometers (MZIs) devices [8] whose collective behavior can be modeled by a unitary transfer matrix. This mathematical property enables the use of Singular Value Decomposition (SVD) to implement general MVM optically. In a PNN, the SVD allows for the factorization of a weight matrix (e.g., from a fully connected layer) into three components: two unitary matrices and a diagonal matrix. PUM meshes directly map unitary matrices, while optical attenuators or amplifiers typically define the diagonal matrix [9].

Two primary approaches exist for designing and training a PNN. The first involves training a conventional NN in the digital domain and subsequently mapping its weights onto a photonic architecture via SVD [5]. The second approach involves an end-to-end training process, where the PNN is trained directly in the photonic domain. This method models the PNN considering the parameters and constraints the photonic circuits impose. While more complex, this latter method is crucial to accurately account for device non-idealities, optical noise, and propagation losses, making it closer to real hardware implementations [10].

In this context, designing and training PNNs requires ad-

Tzamn Melendez Carmona, Stefano Di Carlo, and Alessandro Savino are with the Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy (emails:{tzamn.melendez, alessandro.savino, stefano.dicarlo}@polito.it)

Federico Marchesin and Peter Bienstman are with Photonics Research Group, Ghent University - imec, Ghent, Belgium (emails: {federico.marchesin, peter.bienstman}@ugent.be)

Marco P. Abrate is with Department of Cell and Developmental Biology, University College London, London, UK (e-mail: marco.abrate@ucl.ac.uk)

This paper has received funding from: The NEUROPULS project in the European Union's Horizon Europe research and innovation programme under grant agreement No. 101070238.

0000–0000/00$00.00 © 2015 IEEE

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2026.3698750

vanced tools to efficiently model the photonic circuits involved in their construction. Simulators such as SIMULIA [11], RSOFT [12], and COMSOL [13] are primarily designed for simulating individual photonic components—like waveguides, MZIs, and ring resonators, while Simphony [14] shifts the simulation to the circuit-level. However, these tools are not suitable for end-to-end modeling and training of complete PNN architectures. To address this problem, specialized frameworks—such as Photontorch [15], Neurophox [16], and Neuroptica [17]—have been introduced. These tools were created for early-stage designs with relatively simple architectures, and they face scalability issues as PNNs increase in complexity [9], [18]. As PUM meshes grow, their transfer matrices become increasingly large and computationally demanding, creating a bottleneck for existing State-of-the-Art (SotA) training tools.

This paper proposes the Slicing Method, a new scheme for training large-scale PUM meshes, based on sequential windowed computations, that performs forward and backward propagation using localized 2×2 operators with linear complexity in the number of active cells. The scheme is evaluated against existing PNN tools in terms of execution time, memory usage, and numerical accuracy by training four photonic architectures—Clements [19], Fldzhyan [20], Clements Bell, and Fldzhyan Bell [21]—on four standard datasets: Iris [22], Digits [23], MNIST [24], and Olivetti Faces [25]

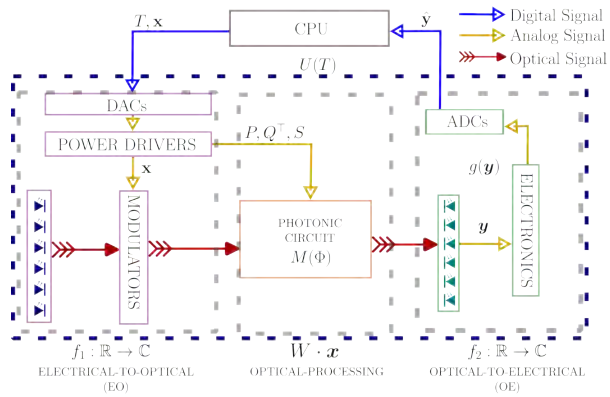
This work makes the following contributions: **(i)** introduces the Slicing Method, **(ii)** presents LuxIA, an open-source framework that instantiates the Slicing Method for several PNN architectures and integrates with PyTorch's automatic differentiation and GPU backends, **(iii)** develops a formal analysis of computational complexity and numerical stability, including an ablation study on the window size that quantifies the trade-off between runtime, memory usage, and numerical error, and **(iv)** provides an empirical comparison of LuxIA with Photontorch, Neurophox, and Neuroptica across multiple datasets and mesh sizes, showing significant reductions in GPU memory usage and training time on large meshes while preserving task accuracy.

The paper is organized as follows: Section II provides an overview of photonic accelerators, modeling approaches and surveys existing SotA tools for PNN training, emphasizing their strengths and limitations. Section III presents the proposed methodology and introduces the LuxIA framework. Section IV benchmarks the training efficiency and performance of LuxIA against other tools, and explores different use cases by training multiple PNNs models on various datasets. Finally, Section V summarizes the contributions of this work and outlines future directions.

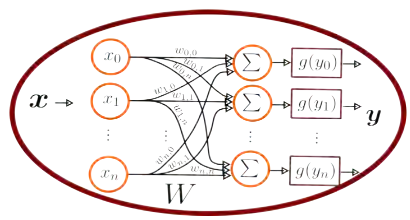
II. Background & Related Works

Photonic accelerators represent a paradigm shift in computing, leveraging the unique properties of light to execute specific operations at extremely high speed. These systems operate at frequencies reaching the terahertz range while consuming only milliwatts of power [26].

The general architecture of a photonic accelerator comprises three main processing stages forming a complete pipeline: Electrical-to-Optical (EO), optical processing, and Optical-to-Electrical (OE) [27], as illustrated in Figure 1(a). The EO provides the interface to convert digital data into optical signals using Digital-to-Analog Converters (DACs), precisely controlled lasers, and modulators. The optical processing stage serves as the computational core, where operations are executed entirely in the photonic domain, exploiting the properties of light-based computation. Finally, the OE stage converts the processed optical signals back to electrical form using high-speed photodetectors and Analog-to-Digital Converters (ADCs) [27].



(a) Hardware Architecture of a PNN.



(b) Fully Connected Layer of a NN.

Fig. 1. (a) Hardware implementation on a photonic accelerator and (b) a fully connected layer of a NN. In (a), the input vector $\mathbf{x}$ is converted to the optical domain for MVM computation, and the result is converted back to the electronic domain to apply the activation function, producing the output $\mathbf{y}$.

Figure 1(b) illustrates a fully connected layer mapped to a photonic accelerator (Figure 1(a)), where the photonic domain handles the MVM operations, and the OE stage uses electronics for the activation function $g(y)$, producing the output $\hat{\mathbf{y}}$ from input $\mathbf{x}$.

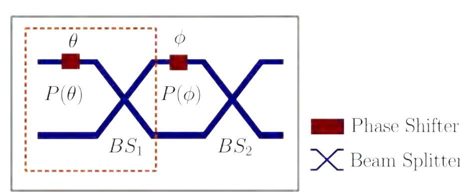


Fig. 2. A MZI is illustrated in the gray rectangle, composed of two beam splitters and two phase shifters (θ and ϕ). This full MZI serves as the single building block in various PUM mesh architectures, such as the Clements mesh [19]. In contrast, the dotted orange rectangle highlights the simpler single building block used in the Fldzhyan mesh [20], composed of a beam splitter followed by a phase shifter.

The majority of PNNs have a photonic circuit capable of performing MVM operations at their core [7]. Among various approaches—including Wavelength Division Multiplexing (WDM)-based circuits [28] and Plain Light Conversion (PLC) [29]—this work focuses on MZI-based implementations due to their proven scalability and programmability. An MZI device (used by Clements et al. [19]), shown in Figure 2, consists of two beam splitters interconnected by waveguides with programmable phase shifters [30]. The beam splitters divide incoming light into two paths, while the phase shifters (θ and ϕ in the figure) enable programmable control over optical interference patterns. Taking Figure 2 as an example, the transfer matrix of a single MZI is calculated by composing the matrices of its beam splitters (Equation 1) and phase shifters (Equation 2), resulting in its transfer matrix M as defined in Equation 3.

$$BS = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & \mathrm{j} \\ \mathrm{j} & 1 \end{bmatrix} \tag{1}$$

$$P(\theta) = \begin{bmatrix} \mathrm{e}^{\mathrm{j}\theta} & 0 \\ 0 & 1 \end{bmatrix} \tag{2}$$

$$\begin{aligned} M &= \underbrace{P(\theta)\cdot BS}_{B_\theta} \cdot \underbrace{P(\phi)\cdot BS}_{B_\phi} \\ &= \frac{1}{2}\begin{bmatrix} \mathrm{e}^{\mathrm{j}(\theta+\phi)} - \mathrm{e}^{\mathrm{j}\theta} & \mathrm{j}(\mathrm{e}^{\mathrm{j}(\theta+\phi)} + \mathrm{e}^{\mathrm{j}\theta}) \\ \mathrm{j}(\mathrm{e}^{\mathrm{j}\phi}+1) & -(\mathrm{e}^{\mathrm{j}\phi}-1) \end{bmatrix} \end{aligned} \tag{3}$$

The fact that MZIs are programmable makes them capable of performing diverse operations. MZIs can perform MVM operations, if they are arranged in a mesh structure. These mesh structures are known as PUM meshes. The term PUM refers to the fact that the transfer matrix of a PUM mesh satisfies unitary properties [8], [19], [20], [31].

This unitary property enables PUMs to perform MVM operations by leveraging the mathematical technique known as SVD. The SVD factorizes any matrix into three matrices, formally expressed in Equation 4, where W is the original matrix, P and Q are unitary matrices, and S is a diagonal matrix containing the singular values. This factorization is particularly advantageous because a weight matrix W can be decomposed into two unitary matrices, implemented by PUM meshes, and a diagonal matrix realized with attenuators or amplifiers [5], [32].

$$W = P \cdot S \cdot Q^\top \tag{4}$$

Reck et al. [8] introduced the first PUM mesh and proposed a systematic approach to optical SVD using triangular MZI arrays. Subsequent developments include the optimized rectangular design by Clements et al. [19] for improved scalability and reduced optical losses, and Fldzhyan's architecture [20] which utilizes beam splitters and phase shifters instead of full MZIs. More recently, Bell et al. [21] introduced further optimizations of these designs. The choice of the PUM architecture directly influences how a PNN is trained and its resilience to hardware imperfections.

PNN training can be accomplished by two primary methods. The first, post-training mapping, trains a standard NN and maps its weights to a PUM mesh via SVD [5], but it ignores device non-idealities and can suffer from accuracy loss in hardware [33]. The second, end-to-end photonic training, optimizes the tunable parameters in the PNN, such as programmable phase shifters in the PUM meshes. Though more complex, the latter models non-idealities like optical noise, losses, and manufacturing variations, yielding training that better matches hardware performance [10].

Effective end-to-end training requires accurate modeling of the PNN. Such training is typically achieved using the transfer matrix method, which provides the mathematical relationship between the system's inputs and outputs [34]. The transfer matrix method is preferred for PNN modeling due to its compact representation, compatibility with optimization algorithms, and computational efficiency [15], [35]. Starting from this model, the end-to-end training process follows the conventional four stages required to train a NN: forward pass (Equation 5), loss computation (Equation 6), and backward propagation (Equations 7–9).

$$\hat{\mathbf{y}} = U(T)\cdot \mathbf{x} \tag{5}$$

$$\mathscr{L} = f\big(\mathbf{y}, \hat{\mathbf{y}}\big) \tag{6}$$

$$\delta = \nabla_{\hat{\mathbf{y}}}\mathscr{L} \tag{7}$$

$$\delta^{[l]} = (T^{[l+1]})^\top \delta^{[l+1]} \tag{8}$$

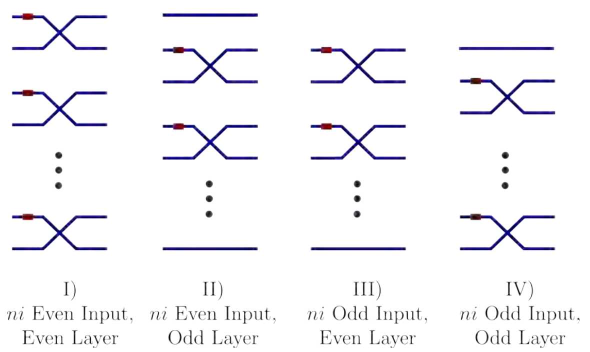
$$T^{[l]} = T^{[l]} - \eta\cdot\delta^{[l]}(\hat{\mathbf{y}}^{[l-1]})^\top \tag{9}$$

The input $\mathbf{x}$ is transformed by the system's transfer matrix $U(T)$ to produce the output $\hat{\mathbf{y}}$. The loss $\mathscr{L}$ is computed between the prediction and the true label $\mathbf{y}$. The gradient of the loss δ is then propagated backward through the system to update the trainable parameters T. As shown in Equations 8 and 9, this is a layer-wise process where the superscript index l refers to a specific "subsystem" in the end-to-end model[1]. The parameters $T^{[l]}$ of each subsystem are updated using the related gradient $\delta^{[l]}$ and the learning rate η.

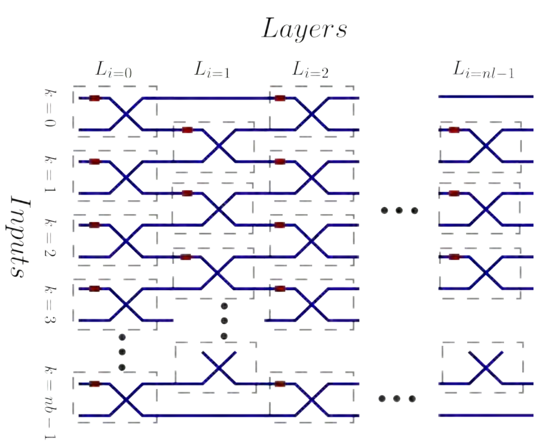
The system matrix $U(T)$ is constructed by multiplying the transfer matrices of its constituent subsystems. These can include input modulators, output detectors, PUM meshes, etc. It is important to note that each subsystem can have its trainable parameters, as illustrated in the case of PUM meshes, where the transfer matrix M comprises the trainable phase shift parameters. Harris et al. [36] were able to build a photonic accelerator with 88 MZIs, showcasing the feasibility of large-scale integration, while modern PNN architectures sytematically require the use of larger PUM meshes with new optical components [5], [7], [9], [18]. Consequently, training PNNs with large PUM meshes presents a significant scalability challenge due to the computational cost of evaluating their transfer matrices, leading to long training times [37]. To illustrate this complexity, consider a photonic mesh structure, such as the Fldzhyan mesh illustrated in Figure 3. The analysis begins by identifying the fundamental repeating unit (i.e., the single block) and computing its transfer matrix, denoted by B.

[1]The term "layer" in the context of backpropagation (indexed by l) should be understood as a computational stage or subsystem within the PNN, distinct from a conventional neural network layer, e.g., a fully connected layer.

Next, the structure is decomposed into different mesh layers, and the transfer matrix for each mesh layer is computed. The final step is to compute the transfer matrix of the entire mesh, denoted by M, by multiplying the transfer matrices of all mesh layers. Key parameters such as the number of inputs (ni), the number of layers (nl), and the number of single blocks per layer (nb) must be identified. These parameters are essential throughout all steps.



(a) Layers configuration



(b) Photonic mesh

Fig. 3. Fldzhyan mesh structure showing (a) single block, (b) mesh layers, and (c) architecture.

The transfer matrix of a single block $B_{i,k}$—uniquely identified by the two-level subindex (i,k) where i corresponds to the mesh layer number and k indicates the block number within that layer—is obtained by multiplying the transfer matrices of its internal components. As illustrated in the gray dashed rectangles of Figure 3, single blocks in the Fldzhyan mesh comprise a beam splitter and a phase shifter. Hence, the transfer matrix of the block $B_{i,k}$ can be calculated as the product of the transfer matrix of the beam splitter, BS, and the phase shifter, $P(\theta_{i,k})$, where $\theta_{i,k}$ denotes the phase shift of the block $B_{i,k}$. Then, the transfer matrix $B_{i,k}$ is computed as shown in Equation 10, with indices $i \in [0, nl-1]$ and $k \in [0, nb-1]$.

$$B_{i,k} = \underbrace{\frac{1}{\sqrt{2}} \begin{bmatrix} e^{j\theta_{i,k}} & j \\ je^{j\theta_{i,k}} & 1 \end{bmatrix}}_{BS \cdot P(\theta_{i,k})} \tag{10}$$

Once the transfer matrix of all individual blocks has been computed, the next step is to calculate the mesh layer transfer matrices, denoted by L. First, identify all single blocks that form a given layer, with their transfer matrices denoted as $B_{i,k}$ for all k. Since each layer is a system in which the number of outputs equals the number of inputs, the mesh layer transfer matrix L must be square, with dimensions $ni \times ni$.

Each mesh layer transfer matrix L_i is then constructed by placing the transfer matrices of its constituent blocks $B_{i,k}$ along the diagonal, while setting all off-diagonal elements to zero, as shown in Equation 11.

$$L_i \underset{i\in[0,nl-1]}{=} \begin{bmatrix} B_{i,0} & 0 & \cdots & 0 \\ 0 & B_{i,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_{i,nb-1} \end{bmatrix} \tag{11}$$

Regarding the Fldzhyan mesh structure, the mesh layers are arranged in four different configurations as shown in Figure 3(a), depending on two parameters: the number of inputs ni and the index of the current layer i. Equations 12 and 13 show the development of the transfer matrix when ni is even.

$$L_i \underset{i\in 2\mathbb{Z}}{=} \underbrace{\frac{1}{\sqrt{2}} \begin{bmatrix} e^{j\theta_{i,0}} & j & \dots & 0 & 0 \\ je^{j\theta_{i,0}} & 1 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & e^{j\theta_{i,nb-1}} & j \\ 0 & 0 & \dots & je^{j\theta_{i,nb-1}} & 1 \end{bmatrix}}_{\mathbb{L}} \tag{12}$$

$$L_i \underset{i\in(2\mathbb{Z}+1)}{=} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ \vdots & & \mathbb{L} & & \vdots \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} \tag{13}$$

When the number of inputs ni is odd, a similar procedure is followed, and the final Equations are shown in Equations 14 and 15.

$$L_i \underset{i\in 2\mathbb{Z}}{=} \frac{1}{\sqrt{2}} \begin{bmatrix} \mathbb{L} & & 0 \\ & & \vdots \\ 0 & \dots & 1 \end{bmatrix} \tag{14}$$

$$L_i \underset{i\in(2\mathbb{Z}+1)}{=} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & \dots & 0 & \\ \vdots & & & \\ 0 & & \mathbb{L} & \end{bmatrix} \tag{15}$$

The final step is to compute the complete photonic mesh structure's transfer matrix M by multiplying the mesh layers' transfer matrices, as given by Equation 16.

$$M = L_0 \cdot L_1 \cdot L_2 \cdot \ldots \cdot L_{nl-1} \tag{16}$$

The transfer matrix method provides a rigorous and widely used framework for modeling and training PNNs. However, as shown in Equations 10, 12 and 16, for large PUM meshes, single-block and layer matrices grow with the number of devices, and the full mesh matrix is obtained by multiplying all layers. This design leads to scalability issues and introduces the following key limitations when training large PNNs:

- *Computational Overhead:* The sparse structure of these matrices leads to useless computations, as many operations involve multiplication by off-diagonal zero elements.

- *High Memory Usage:* Storing large, dense transfer matrices imposes a substantial memory burden. This is particularly problematic when training PNNs on Graphics Processing Units (GPUs), where available memory is often a critical constraint.
- *Gradient Bottleneck:* This critical issue arises during the weights update process of the backpropagation algorithm. As described in Equation 9 and the preceding gradient computations in Equations 7 and 8, updating the parameters of the full system transfer matrix $U(T)$ requires evaluating the gradient of the loss $\mathscr{L}$ with respect to all trainable parameters T. Because the mesh matrix M couples all phase shifters, gradients for individual parameters require reconstructing the full mesh matrix in every backward pass, creating a severe bottleneck for large meshes. Consequently, each backward pass necessitates reconstructing the full mesh transfer matrix M from its constituent layer matrices L_i to correctly compute gradients for all parameters. This repeated, and costly reconstruction creates a significant computational bottleneck, especially for large-scale meshes.

Despite growing interest in PNNs, the availability of end-to-end training frameworks remains limited. Among the most notable Python-based tools are Photontorch [15], Neurophox [16], and Neuroptica [17]. These frameworks differ in backend technologies, hardware acceleration capabilities, and development activity. Table I summarizes their technical characteristics.

TABLE I
TECHNICAL SUMMARY OF PNN SIMULATION TOOLS

Tool	Backend	GPU Support	GitHub Last Commit
Photontorch	PyTorch	Yes	2022
Neurophox	TensorFlow	Yes	2021
Neuroptica	NumPy	No	2020

Photontorch is a simulation framework built on PyTorch [38], offering automatic differentiation and GPU acceleration to support gradient-based training of PNNs. It has been used to simulate architectures such as Coupled Resonator Optical Waveguides (CROWs) [39] and PNNs with PUMs MZIs for tasks like MNIST classification. Photontorch includes prebuilt PUM configurations, such as the Clements mesh [19], and provides a user-friendly interface for building custom photonic circuits. However, training large-scale circuits remains time-consuming, often requiring hours to optimize even a few hundred parameters [37].

Neurophox, developed by Pai et al., is a TensorFlow-based framework focused on rectangular and triangular PUM meshes composed of MZIs and phase shifters. It uses the transfer matrix method to model photonic components and supports GPU-accelerated backpropagation. While the framework benefits from integration with TensorFlow's optimizers and includes mesh-specific enhancements [40], its training performance degrades as the network depth and mesh size increase [37].

Neuroptica is a NumPy-based framework for simulating and training PNNs. It implements a custom training algorithm inspired by the adjoint variable method proposed in [41], which is conceptually similar to backpropagation and was originally designed for efficient in-situ training of photonic hardware. Instead of explicitly differentiating large transfer matrices, the method computes gradients through forward and adjoint propagations across the same photonic circuit. Neuroptica emulates this approach in software while relying on the transfer-matrix method to model photonic components such as beam splitters, MZIs, and phase shifters. The framework supports network construction at different levels of abstraction; however, being NumPy-based, it lacks GPU support and exhibits slow training times as model complexity increases [37].

Across all three tools, the dominant limitation is slow training, as even moderate-scale PNN architectures can require hours of optimization due to transfer-matrix differentiation [37]. While maintenance or compatibility can be mitigated through containerization [42], the underlying training inefficiency remains a fundamental obstacle to scalable PNN simulation frameworks.

III. METHODOLOGY

To address the computational challenges of large photonic mesh networks, this work introduces LuxIA, a framework based on the Slicing Method to reduce training cost. Instead of constructing a single large transfer matrix for the full mesh, the network is processed sequentially in small computational windows. The optical signal propagates stepwise through local matrix-vector operations between neighboring optical elements. This avoids storing large dense matrices and reduces both memory usage and computational cost.

The methodology is formalized through the following systematic steps:

1) *Window Partitioning:* The mesh is computationally divided into nw processing windows, where for many common architectures nw equals the number of physical layers, nl. Each window, denoted W_w, where $w \in [0, nw]$, contains a group of optical operations that can be computed independently within that slice.
2) *Cell Identification:* This step introduces the core abstraction of the Slicing Method: The cell. The purpose of the cell is to provide a uniform computational interface for all possible elements within a window. Therefore, every element—a single block or a simple bypass connection—is encapsulated within a cell object. The single block is uniquely identified by its layer $i \in [0, nl-1] \cap \mathbb{Z}$ and block number $k \in [0, nb-1] \cap \mathbb{Z}$ as $B_{i,k}$, and mapped to an active cell. Crucially, a path that does not contain a block is mapped to a bypass cell. Each cell, denoted as $\text{CELL}_{w,c}$, stores an internal flag, `is_active`, that allows the processing algorithm to treat all elements uniformly:
   - An *active cell* (`is_active = true`) encapsulates a 2×2 optical block and contains its transfer matrix (e.g., $B_{i,k}$).
   - A *bypass cell* (`is_active = false`) encapsulates a single pass-through connection and requires no further information.

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2026.3698750

This encapsulation is key to the framework's generality. As illustrated in Figure 4, window W_0 maps the single blocks $B_{0,0}$ and $B_{0,1}$ to two active cells ($\mathrm{CELL}_{0,0}$, $\mathrm{CELL}_{0,1}$), while window W_1 is represented as an ordered list of three cells: a bypass cell ($\mathrm{CELL}_{1,0}$), an active cell containing $B_{1,0}$ ($\mathrm{CELL}_{1,1}$), and another bypass cell ($\mathrm{CELL}_{1,2}$).

3) *Sequential Signal Propagation:* An input vector $\mathbf{x} \in \mathbb{C}^{ni}$ is fed into the first window, W_0. The output vector from window W_w, denoted $\mathbf{y}_w$, becomes the input vector for the subsequent window, W_{w+1}, creating a computational cascade that mirrors the physical flow of light.

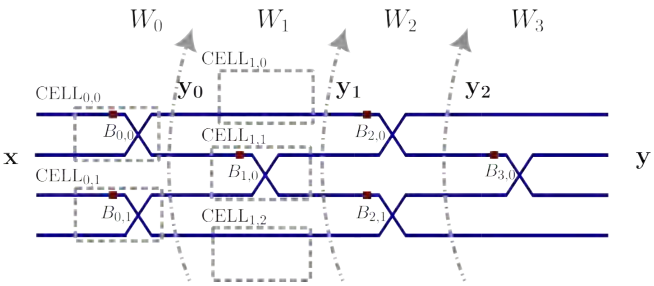


Fig. 4. Decomposition of the Fldzhyan mesh structure into sequential processing windows. Physical blocks ($B_{i,k}$) are mapped to computational cells ($\mathrm{CELL}_{w,c}$) for processing.

The formalization of the slicing concept is detailed in the Algorithm 1. The power of this framework lies in its uniform treatment of disparate physical components, made possible by the cell abstraction. Therefore, the core 'process_window' function is agnostic to the specific mesh topology (e.g., Fldzhyan vs. Clements): it only needs to process a simple, ordered list of cell objects. The high-level 'process_mesh' function orchestrates the overall signal flow. It initializes the computation with the mesh input vector and then iterates through each window, sequentially updating the signal vector by calling 'process_window' for each one. The core logic resides within the 'process_window' function. This function accepts an input vector and a list of cells for the window. A single port index, p, tracks the current position in the input and output vectors. The function iterates through the cells, and its logic branches based on the `is_active` flag of each cell:

- If the cell is active, it consumes two inputs, performs a 2×2 matrix-vector multiplication using its stored transfer matrix, and produces two outputs. The port index p is then advanced by two.
- If the cell is a bypass, it consumes one input, performs an identity copy, and produces one output ($y_p = x_p$). The port index k is advanced by one.

For an active cell representing the single block $B_{i,k}$, the mathematical operation is a transformation from $\mathbb{C}^2 \to \mathbb{C}^2$, as depicted in Equation 17.

$$\begin{bmatrix} y_p \\ y_{p+1} \end{bmatrix} = B_{i,k} \begin{bmatrix} x_p \\ x_{p+1} \end{bmatrix} \tag{17}$$

To make this generalized framework concrete, it is applied to the 4×4 Fldzhyan mesh from Figure 4, which has $ni = 4$ inputs and $nl = 4$ layers. The mapping of single blocks to

Algorithm 1 Photonic Mesh Processing (Signal Propagation)

```
function PROCESS_MESH(x, mesh)
    y ← x
    for each window in mesh.windows do
        y ← PROCESS_WINDOW(y, window)
    end for
    return y
end function

function PROCESS_WINDOW(x, window)
    y ← x                          ▷ This automatically do the bypass cells
    p ← 0
    for each cell in window.cells do
        if cell.is_active then
            y0, y1 ← cell.TransferMatrix · [x[p], x[p + 1]]^⊤
            y[p] ← y0
            y[p + 1] ← y1
            p ← p + 2
        else
            p ← p + 1              ▷ Bypass cell, identity copy
        end if
    end for
    return y
end function
```

computational cells for its four windows ($nw = 4$) is concisely summarized in Figure II.

TABLE II
CELL CONFIGURATION FOR THE FLDZHYAN 4×4 MESH WINDOWS. THE CELL INDEX c IS INCREMENTED SEQUENTIALLY WITHIN EACH WINDOW.

Window	Cell 0 ($c = 0$)	Cell 1 ($c = 1$)	Cell 2 ($c = 2$)
W_0	Active ($B_{0,0}$)	Active ($B_{0,1}$)	-
W_1	Bypass	Active ($B_{1,0}$)	Bypass
W_2	Active ($B_{2,0}$)	Active ($B_{2,1}$)	-
W_3	Bypass	Active ($B_{3,0}$)	Bypass

The fundamental active element in the Fldzhyan architecture is a phase-shifted beam splitter. For each active cell $\mathrm{CELL}_{w,c}$, which corresponds to a single processing block $B_{i,k}$, the generic matrix operation described in Equation 17 simplifies to a specific form governed by a single phase shift parameter, $\theta_{i,k}$. The inputs to block $B_{i,k}$, denoted x_p and x_{p+1}, are taken from the outputs of the previous window W_{w-1}—specifically, $y_{w-1,p}$ and $y_{w-1,p+1}$. The outputs of the current block, y_p and y_{p+1}, become the outputs of the current window W_w, labeled $y_{w,p}$ and $y_{w,p+1}$.

The transformation performed by the active cell is defined by the transfer matrix shown in Equation 18:

$$\begin{aligned} y_{w,p} &= \frac{1}{\sqrt{2}} \left(e^{\mathrm{j}\theta_{i,k}} y_{w-1,p} + \mathrm{j} y_{w-1,p+1} \right) \\ y_{w,p+1} &= \frac{1}{\sqrt{2}} \left(\mathrm{j} e^{\mathrm{j}\theta_{i,k}} y_{w-1,p} + y_{w-1,p+1} \right) \end{aligned} \tag{18}$$

This equation governs the behavior of any active cell in the architecture. If a cell is inactive, a simple bypass transformation is applied instead, where the input $x_p = y_{w-1,p}$ is passed directly to the output $y_p = y_{w,p}$.

A crucial advantage of the Slicing Method is its inherent compatibility with gradient-based optimization algorithms, the backbone of modern AI. The localized and sequential nature of the computation enables an efficient implementation of backpropagation without needing to assemble the full, system-wide transfer matrix.

The gradient of a loss function $\mathscr{L}$ with respect to a tunable parameter $\theta_{i,k}$ is calculated via the chain rule. This process propagates the error signal backward through the mesh, window by window. The update for a specific parameter depends only on local information:

$$\frac{\partial \mathscr{L}}{\partial \theta_{i,k}} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_w} \cdot \frac{\partial \mathbf{y}_w}{\partial \theta_{i,k}} \tag{19}$$

In this expression, $\frac{\partial \mathscr{L}}{\partial \mathbf{y}_w}$ is the gradient of the loss with respect to the output of the window w containing the parameter propagated backward from the final layer. The term $\frac{\partial \mathbf{y}_w}{\partial \theta_{i,k}}$ is the local gradient, which is non-zero only for the outputs of the specific cell, $\mathrm{CELL}_{w,c}$, that contains the parameter $\theta_{i,k}$.

Using the Fldzhyan block model from Equation 18, the local partial derivatives with respect to the phase parameter $\theta_{i,k}$ are:

$$\frac{\partial y_{w,p}}{\partial \theta_{i,k}} = \frac{\mathrm{j}}{\sqrt{2}} \mathrm{e}^{\mathrm{j}\theta_{i,k}} y_{w-1,p} \tag{20}$$

$$\frac{\partial y_{w,p+1}}{\partial \theta_{i,k}} = -\frac{1}{\sqrt{2}} \mathrm{e}^{\mathrm{j}\theta_{i,k}} y_{w-1,p} \tag{21}$$

This locality is the key to the method's computational efficiency. The gradient calculation only requires the input to the cell ($y_{w-1,p}$) which was stored during the forward pass, avoiding any need for large matrix manipulations. Consequently, the computational complexity for both the forward pass (inference) and the backward pass (gradient calculation) scales linearly with the number of tunable parameters, which is proportional to $ni \cdot nl$ for a typical mesh and not quadratically to the number of inputs and layers, as in the transfer matrix method.

A. Computational Complexity and Photonic Non-Idealities

In the conventional transfer-matrix formulation, the layer-wise propagation defined in Equation 16 requires the sequential construction and multiplication of dense transfer matrices $L_i \in \mathbb{C}^{ni \times ni}$. The multiplication of two dense square matrices has a computational complexity of $\mathcal{O}(ni^3)$. Therefore, constructing the full-mesh transfer matrix requires $nl - 1$ matrix multiplications, resulting in a total complexity of $\mathcal{O}(nl\, ni^3)$.

Unlike adjoint-inspired transfer-matrix approaches such as Neuroptica [17], [41], the proposed Slicing Method computes gradients directly from local optical interactions and cached signals, avoiding reconstruction and storage of the transfer matrices. Where the window size $nw = nl$, operates directly on the local 2×2 photonic interactions in Equation 17, avoiding dense matrix construction. Since each photonic cell performs only a constant-time local operation, the cost per cell is $\mathcal{O}(1)$. For a mesh layer, the number of active cells scales linearly with $ni/2$, leading to a per-layer complexity of $\mathcal{O}(ni)$, and to a full mesh propagation complexity defined by $\mathcal{O}(nl\, ni)$.

When $nw < nl$, the photonic mesh layers are grouped into windows, and local transfer matrices are then dense square matrices with a computational complexity of $\mathcal{O}(ni^3)$. In this case, each window requires $nw - 1$ matrix multiplications, yielding a complexity of $\mathcal{O}(nw\, ni^3)$. When nw is small, ni is the dominant value leading to $\mathcal{O}(ni^3)$ complexity, one order lower than the full transfer matrix formulation.

A further advantage is that photonic non-idealities can be incorporated directly at the level of local transfer matrices. Each active cell implements a 2×2 transformation as in Equation 22, where the coefficients $C_{m,n}$ encode both ideal behavior and non-ideal effects, consistent with the Fldzhyan block in Equation 10.

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}, \tag{22}$$

As an example, a 2×2 multimode-interference coupler can extend the ideal beamsplitter in Equation 1 by including insertion loss and imbalance [9]. With intensity transmission $i_{\text{loss}} \in [0, 1]$ and power imbalance $\Delta_{\text{imb}} \in [-0.5, 0.5]$, the transfer matrix becomes as the one in Equation 23, where $m_C = \sqrt{i_{\text{loss}}}$, $a_0 = \sqrt{0.5 + \Delta_{\text{imb}}}$, $a_1 = \sqrt{0.5 - \Delta_{\text{imb}}}$, capturing amplitude attenuation and power-splitting asymmetry while preserving the structure needed for efficient cell-level processing.

$$B_{\text{MMI}} = m_C \begin{bmatrix} a_0 & \mathrm{j}a_1 \\ \mathrm{j}a_1 & a_0 \end{bmatrix}, \tag{23}$$

More generally, phase offsets, fabrication variability, and thermal drift can be modeled by perturbing the coefficients $C_{m,n}$ or the phase parameters in Equations 3 and 10; because these effects remain local, they do not change the algorithmic structure or complexity of the Slicing Method. This locality enables realistic hardware effects to be modeled while maintaining linear scaling in computation and memory, which is critical for simulating and training large-scale photonic neural networks.

IV. Experimental Design

This section evaluates LuxIA by comparing it with other SotA tools in terms of training performance and hardware resource utilization. Subsection IV-A assesses functionality by comparing training and validation dynamics; Subsection IV-B evaluates memory usage and execution time; Subsection IV-C demonstrates robustness on multiple PNN architectures and datasets.

The experiments in Subsections IV-A and IV-B use the PNN shown in Figure 5, and the reference dataset used is Digits [23]. This dataset contains 1,797 grayscale images of handwritten digits (0–9), each sized 8×8. We split the dataset into training (70%), validation (10%), and testing (20%) subsets for all experiments. We selected this dataset because its similarity to the MNIST dataset [24] makes it suitable for PNN research, while also keeping the dataset size contained.

To serve as a hardware reference, all experiments run on a workstation equipped with an AMD Ryzen 9 7950X processor featuring 16 cores and 32 threads, alongside 64 GB RAM and an NVIDIA RTX A4000 GPU with 16 GB VRAM.

A. Comparison with Other Frameworks as a Training Tool

We validate LuxIA's functionality by comparing it against other SotA tools. The comparison focuses on training dynamics, rather than merely comparing outputs. For this reason, the evaluation is conducted over a complete training pipeline, including training, validation, and testing phases.

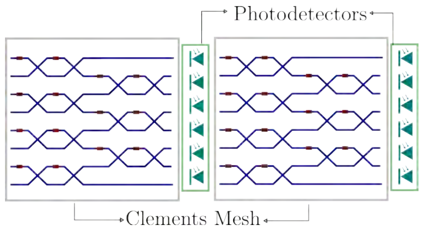


Fig. 5. PNN used for the benchmark experiments. The circuit consists of two sets of Clements meshes followed by a photodetector layer. The mesh size varies depending on the experiment: for Subsection IV-A, the size is fixed to $N = 64$; for Subsection IV-B, the size depends on the scenario. For the Batch Size-Dependent scenario, the size is fixed to $N = 800$, while for the N-Dependent experiment, N ranges from 100 to 900 in steps of 200.

Neurophox, Neuroptica, Photontorch, and our approach model the PNN in Figure 5, avoiding unfairness due to modeling errors. All tools are configured with identical parameters whenever possible to ensure a proper comparison. In particular, they all employ the Root Mean Square Propagation (RMSProp) optimizer, with the exception of Neuroptica, which uses a custom in-situ Adam optimizer based on the adjoint method [41]. The training is repeated 5 times for each tool, with different random seeds affecting parameters initialization and train/validation splits (the test set remains fixed). The learning rate is set to 0.0005, and the batch size is 512. Each training is performed for 150 epochs, and the training and validation curves are plotted over training epochs. The results of the training and validation phases are presented in Figure 6, which shows the average training loss and average validation accuracy over five runs for each tool. The shaded area represents the minimum and maximum values obtained during the multiple runs.

Figure 6(a) shows that all tools have similar training loss trends, with LuxIA converging fastest, followed by Photontorch, Neurophox, and Neuroptica. After 150 epochs, final training losses are 0.1462 (Neurophox), 0.1470 (Photontorch), 0.2258 (Neuroptica), and 0.1667 (LuxIA).

Figure 6(b) presents validation accuracy, where all tools show comparable trends. Final validation accuracies are 95.97% (Neurophox), 94.79% (Photontorch), 93.75% (Neuroptica), and 94.44% (LuxIA), confirming effective PNN training across frameworks.

Closer analysis shows that Neuroptica's training loss decreases more slowly and its validation accuracy is less stable during the first 80 epochs, likely due to the behavior of its optimizer's settings. Since its performance later aligns with the other tools, it is worth mentioning that this optimizer might benefit from further hyperparameter search, which is beyond the scope of this comparison.

For test set evaluation, LuxIA achieves an average accuracy (across all random seeds) of 94.05%, Neurophox 95.97%, and Photontorch 94.79%. Neuroptica's test accuracy is not reported, as it cannot save trained models for later evaluation. Overall, all frameworks demonstrate comparable training dynamics and accuracy, confirming that LuxIA and the Slicing method behave consistently with existing tools.

B. Evaluation of Hardware Resource Efficiency

While the previous experiments established functional training equivalence, this section evaluates the hardware resource efficiency of all tools. The goal is to comprehensively understand each tool's memory consumption and execution time. Two scenarios have been considered:

1) *Batch Size-Dependent Scenario* (subsection IV-B1): This scenario focuses on the impact of batch size on hardware resources. We fix PUM mesh size to $N = 800$ and vary the training batch size to study its impact on execution time and memory usage [43].
2) *Mesh Size-Dependent Scenario* (subsection IV-B2): This scenario focuses on the impact of PUM mesh size on memory consumption and execution time. We fix the batch size to 128 and vary the PUM mesh size, which dominates both computation and memory in transfer-matrix-based tools.

Measurements of time and memory usage were recorded using Python libraries such as `time`, `psutil`, and `pynvml`. Both Central Processing Unit (CPU)-only and combined CPU-GPU configurations were tested, ensuring that all experiments maintained identical settings across tools. Measurements were conducted over a single training and validation epoch, with the test set excluded from these measurements.

1) Batch Size-Dependent Scenario Results: In this set of experiments, the PUM mesh size was fixed at $N = 800$, and batch sizes were varied exponentially from 64 to 512. The results are reported in Figure 7.

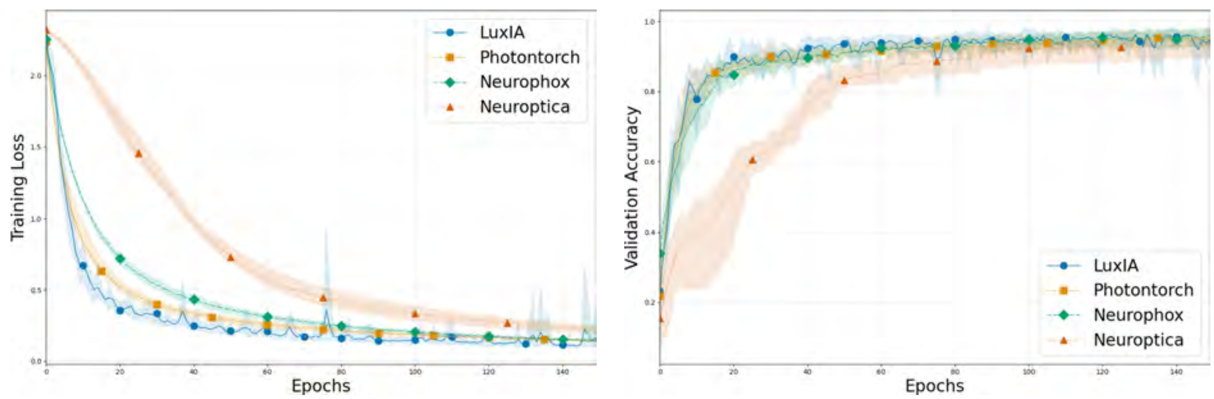
LuxIA proved to be the most efficient framework in all configurations. In CPU-only mode, it achieved competitive execution times (41.05s–30.95s) with a low memory footprint (4.47–15.99GB). With GPU acceleration, execution times dropped from 16.45s (batch size 64) to 3.48s (batch size 512), a 4.7× speedup. LuxIA also scaled GPU memory efficiently (1.51–9.99GB) and maintained minimal CPU memory usage (around 1.24GB).

Photontorch failed immediately during model construction due to excessive host memory requirements. In particular, instantiating an $N \times N$ Clements mesh with $N = 800$ on the CPU triggered a PyTorch allocation error, *"DefaultCPUAllocator: can't allocate memory: you tried to allocate 6569994240000 bytes. Error code 12 (Cannot allocate memory)"*, while initializing the internal `num_ports` × `num_ports` buffers. Reducing the batch size, forcing CPU execution, and experimenting with lower-precision data types (down to float16), still required 3.28 TB of memory for the same buffers. The failure arises before any training loop is entered, and it is caused by the size of the mesh itself rather than by the batch dimension or arithmetic precision. As a result, Photontorch could not be included in Fig. 7.

Neuroptica showed moderate memory efficiency on CPU (8.80–8.82GB across batch sizes), but at a high computational cost: execution times ranged from 4,915.36s (batch size 64) to 818.96s (batch size 512), as shown in Figures 7(a) and 7(b), making it unsuitable for time-sensitive tasks.

Neurophox ran successfully across all batch sizes and on both CPU and GPU. On CPU, memory usage increased

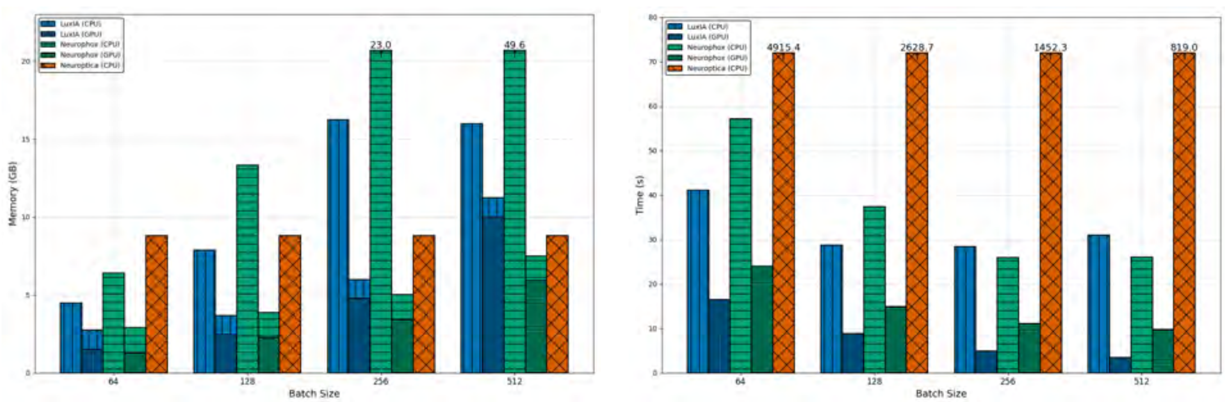
This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2026.3698750



(a) Training loss

(b) Validation Accuracy

Fig. 6. Comparison of training loss and validation accuracy with Photontorch, Neurophox, and Neuroptica



(a) Memory Usage vs Training Batch Size

(b) Execution Time vs Training Batch Size

Fig. 7. Memory usage and execution time for varying training batch sizes with a fixed PUM mesh size of $N = 800$.

from 6.41GB (batch size 64) to 49.64GB (batch size 512), with execution times between 57.18s and 26.02s. With GPU acceleration, execution times improved to 24.00s–9.69s, and GPU memory scaled efficiently (1.32–5.92GB). CPU memory remained stable at about 1.57GB across all batch sizes.

For large-scale applications (batch size 512), LuxIA computes approximately 235 times faster than Neuroptica and 2.8 times faster than Neurophox, while maintaining competitive memory usage. To better highlight differences among the top-performing frameworks, the y-axes in both plots have been truncated, particularly to prevent Neuroptica's much longer execution times from dominating the scale.

2) Mesh Size-Dependent Scenario Results: In this set of experiments, the batch size was fixed at 128, and the PUM mesh size was varied linearly from 100 to 900 in steps of 200. The results are reported in Figure 8.

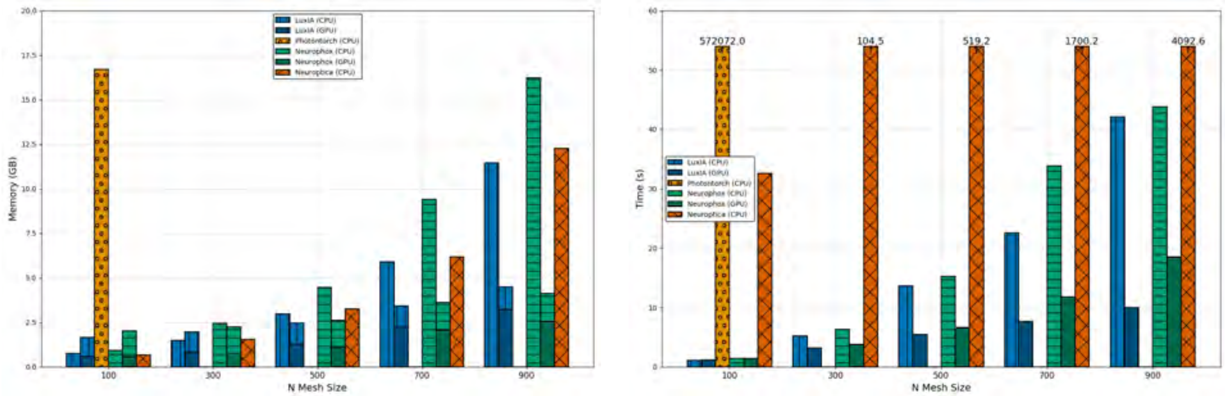
The comparative analysis of this scenario highlights substantial differences in performance and efficiency among the tested frameworks. Across all mesh sizes and hardware configurations, LuxIA consistently outperformed the alternatives. On CPU, LuxIA achieved the lowest execution times, ranging from 1.10s ($N = 100$) to 42.09s ($N = 900$), and maintained the most efficient memory usage, scaling from 0.75GB to 11.46GB. When leveraging GPU acceleration, LuxIA further reduced execution times to a range of 1.18s ($N = 100$) to 10.04s ($N = 900$), representing an approximate 4.2× speedup at the largest mesh size. Its GPU memory usage remained efficient, between 0.58GB and 3.21GB, with minimal CPU memory overhead (1.08GB to 1.27GB).

In contrast, Photontorch only completed the benchmarking process for the smallest mesh size ($N = 100$) on CPU, consuming 16.69GB of memory and taking 572071.96s (≈6.6 days) to execute. The rest of the benchmarking process was unable to run due to excessive memory requirements or computational inefficiencies. As a result, Photontorch appears only at $N = 100$ in Figure 8 and it results impractical for training and validating larger-mesh PNNs.

Neuroptica demonstrated efficient memory scaling on CPU, with usage increasing from 0.66GB to 12.25GB as mesh size grew. However, this came at the expense of execution time, which increased sharply from 32.62s to 4,092.63s (see Figures 8(a) and 8(b)). While Neuroptica's memory efficiency was comparable to LuxIA's, its longer training time became increasingly problematic as the scale of the problem grew, potentially leading to substantial delays for larger mesh sizes.

Neurophox successfully executed across all mesh sizes on both CPU and GPU. On CPU, memory usage ranged from 0.93GB to 16.21GB, with execution times between 1.41s and

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2026.3698750



(a) Memory Usage vs Mesh Size

(b) Execution Time vs Mesh Size

Fig. 8. Memory usage and execution time for varying PUM mesh sizes with a fixed batch size of 128.

43.78s. With GPU acceleration, execution times improved to 1.38s–18.53s, and GPU memory usage scaled efficiently from 0.57GB to 2.51GB. CPU memory usage remained stable, between 1.44GB and 1.61GB, across all mesh sizes.

Eventually, Appendix A contains a numerical stability analysis and a memory trade-off study to complement the comparison with the SotA tools.

C. Use cases

After validating LuxIA's functionality and efficiency, the following experiments aim to demonstrate the framework's capability to train different PNNs with different PUMs meshes and diverse datasets. Four widely-used PUM meshes—Clements [19], Fldzhyan [20], and their Bell [21] variants are evaluated against four datasets of increasing complexity: Iris [22], Digits [23], MNIST [24], and Olivetti Faces [25].

While the Digits dataset has already been described in the previous section, the other selected datasets challenge the framework in terms of memory usage and computational complexity:

- *Iris*: A dataset with 150 samples and 4 features, ideal for rapid prototyping and architecture testing.
- *MNIST*: A dataset with 70,000 samples and 784 features, creating meshes of size 784×784. While fabrication of such large meshes remains beyond current capabilities, this dataset tests the framework's ability to handle computationally intensive training with large datasets. The MNIST dataset is a standard benchmark in the field. Due to the dataset size, if the toolchain is not optimized, it may take a long time to train the model, making it a good test case to evaluate the framework's performance.
- *Olivetti*: A dataset of 400 samples with 1,024 features that has not previously been used in the context of PNNs. The face classification task presents a novel challenge for photonic computing, requiring meshes of size 1024×1024. This dataset challenges the framework due to its high feature count and complex patterns. It is an interesting test case for assessing its capabilities to train large PNNs on complex tasks.

Table III summarizes performance across datasets and PUM meshes, showing final training loss, validation accuracy, and test accuracy. All experiments used the same PNN topology (Figure 9)—two PUM meshes, photodetectors, a bias layer, and a diagonal layer, differing only by the mesh type. This topology is the most commonly used in the literature [19]–[21] and enables fair comparison, as differences in results reflect only mesh optimization. Each configuration was trained with the RMSProp optimizer, run five times, and averaged.

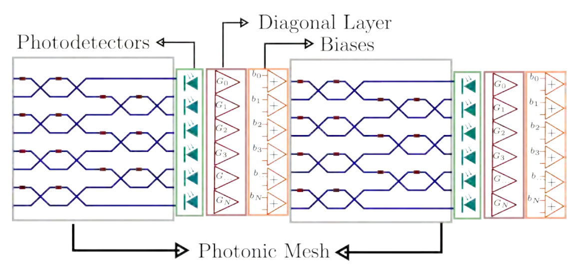


Fig. 9. PNN used in the experiments. The circuit consists of two PUM meshes (Clements, Fldzhyan, or their Bell variants) connected in series, followed by photodetectors, a bias layer, and a diagonal layer.

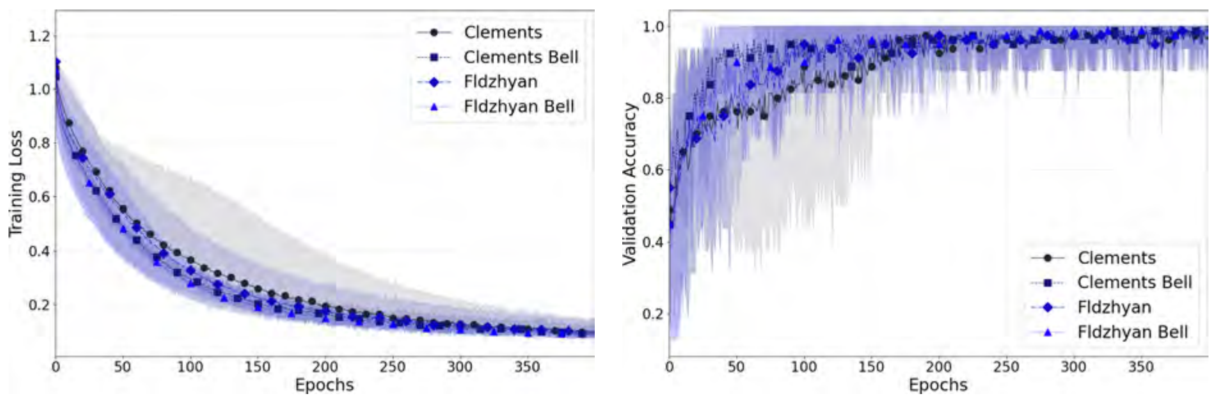
All four PNNs were evaluated on the IRIS dataset over 400 epochs. As depicted in Figure 10, the models exhibited highly effective and consistent learning behavior. The training dynamics show a rapid convergence phase within the first 100 epochs, during which training loss plummets and validation accuracy quickly surpasses 90%. The shaded regions in the plots, representing the standard deviation across 5 runs with different seeds, are initially wide but narrow significantly after approximately 150 epochs. This indicates that all PNNs stabilize to a consistent performance trajectory. Subsequently, the models settle onto a high-performance plateau, maintaining validation accuracies above 95% until the end of training. This stable, high-accuracy performance suggests that all PNNs, no matter the PUM mesh, effectively learned the data's underlying patterns without significant overfitting.

The final metrics, summarized in the second row in Table III, confirm this strong, competitive performance. Notably, the results highlight a subtle distinction between validation and test performance: while the PNN with the Clements Bell

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2026.3698750

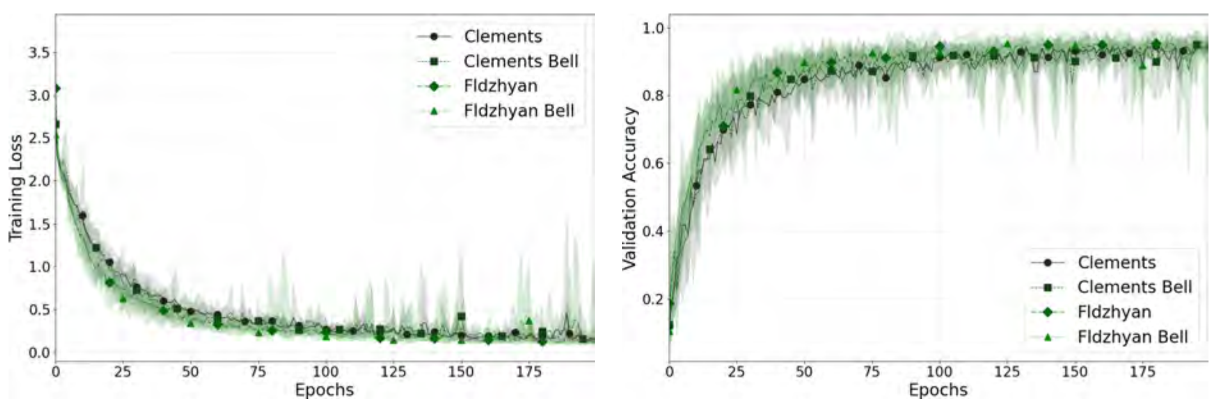
TABLE III
PERFORMANCE METRICS BY DATASET AND PNN. EACH ENTRY SHOWS THE FINAL TRAINING LOSS/VALIDATION ACCURACY/TEST ACCURACY.

Dataset (Mesh, Epochs)	Clements	Clements Bell	Fldzhyan	Fldzhyan Bell
Iris (4×4, 400)	0.0964 / 97.50% / 95.33%	0.0902 / **98.75%** / 94.00%	0.0997 / 97.50% / **96.66%**	0.0896 / 97.50% / 96.00%
Digits (64×64, 200)	0.1400 / 94.03% / **94.055%**	0.1610 / **94.72%** / 91.77%	**0.1129**/ 94.17% / 92.16%	0.3053 / 90.56% / 92.66%
MNIST (784×784, 150)	0.1026 / 97.03% / 97.17%	0.0851 / 96.06% / 97.16%	**0.0588**/ **97.16%** / 97.44%	0.0634 / 94.86% / **97.60%**
Olivetti (1024×1024, 400)	0.3038 / 62.5% / 66.00%	0.4915 / 64.37% / 64.25%	0.4533 / **73.12%** / 70.00%	**0.2648** / 70% / **70.50%**



(a) Training Loss. (b) Validation Accuracy.

Fig. 10. Training loss and validation accuracy for the Iris dataset.



(a) Training Loss. (b) Validation Accuracy.

Fig. 11. Training loss and validation accuracy for the Digits dataset.

mesh obtained the highest validation accuracy of 98.75%, the standard Fldzhyan mesh proved most effective on unseen data, achieving the top test accuracy of 96.66%. Meanwhile, the Fldzhyan Bell mesh reached the lowest training loss (0.0896), suggesting the tightest fit to the training data. The close alignment between high validation and test accuracies across all models underscores their robust generalization capabilities for this task.

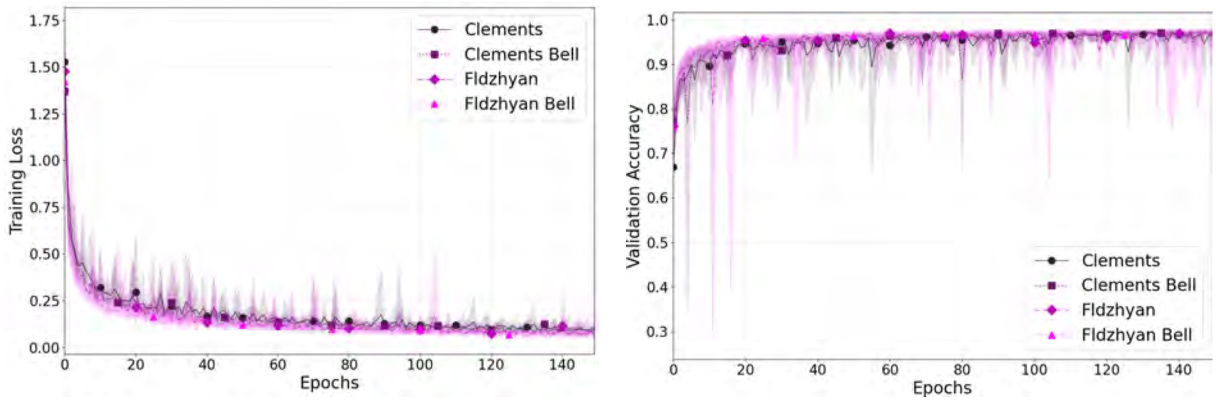
Transitioning to the more complex dataset like Digits, the models were trained for 200 epochs, revealing significant performance distinctions among the different PNNs. Figure 11 and the third row of Table III present the mean performance over five seeds.

The training curves in Figure 11 illustrate a slight divergence that emerges within the first 50 epochs. Three PNNs, Clements, Clements Bell, and Fldzhyan, demonstrate robust and stable learning trajectories. In stark contrast, the Fldzhyan Bell PNNs struggled, exhibiting slightly wider oscillations in its training loss throughout the 200 epochs and converging to a somewhat lower final validation accuracy plateau. This slight instability is quantified by its high final mean training loss of 0.3053 and low mean validation accuracy of 90.56% compared to the other PNNs, which achieved training losses below 0.16 and validation accuracies above 94%.

The third row in Table III reveals a nuanced trade-off between training fit, validation, and generalization. The Fldzhyan PNN demonstrated a superior fit to the training data, achieving the lowest final mean training loss of 0.1129. The Clements Bell PNN excelled on the validation set, reaching the highest mean validation accuracy of 94.72%. However, the ultimate measure of generalization—performance on the unseen test set—was best for the Clements PNN, which achieved the top test accuracy of 94.06%.

This outcome highlights that, although the meshes are the-

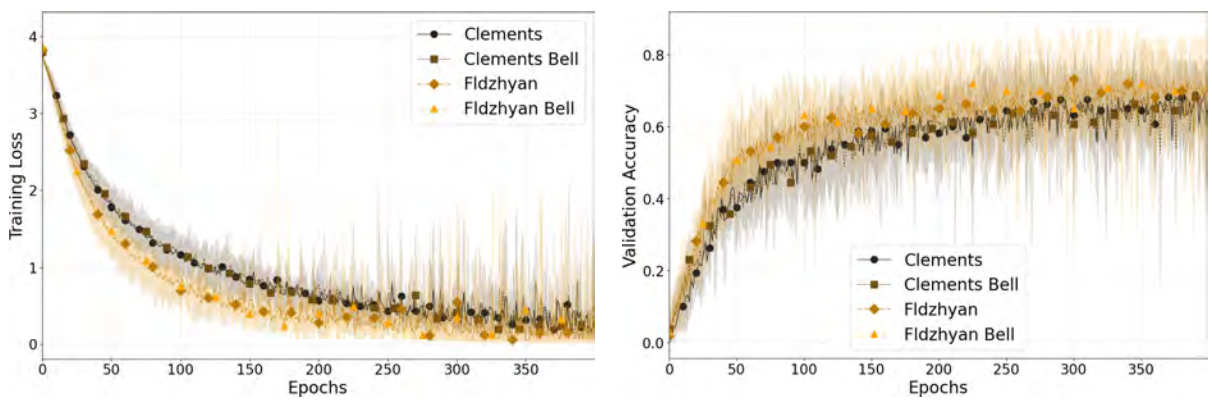
This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2026.3698750



(a) Training Loss.

(b) Validation Accuracy.

Fig. 12. Training loss and validation accuracy for the MNIST dataset.



(a) Training Loss.

(b) Validation Accuracy.

Fig. 13. Training loss and validation accuracy for the Olivetti Faces dataset.

oretically equivalent for the Digits dataset, they show distinct optimization and generalization behaviors in practice. As a result, the backpropagation algorithm adjusts the parameters in a specific way for each mesh, affecting both optimization and generalization.

For the MNIST dataset, models were trained for 150 epochs using a batch size of 512 and a learning rate of 0.000038, the mean performance over the five seeds on this task provides critical insights into the generalization capabilities of the different PNNs, as shown in Figure 12 and Table III.

On this dataset, the learning dynamics are characterized by a rapid convergence. As depicted in Figure 12(b), all four PNNs achieve over 90% validation accuracy within the first 20 epochs before settling into a stable performance plateau. The validation accuracy shows fluctuations over the five seeds, but the mean performance is over 92% for all PNNs after 100 epochs. The trajectories are more stable on the training loss side, as shown in Figure 12(a). Within the first 50 epochs, all models exhibit a fast decline in training loss, progressively decreasing until the end of training. Visually, the Fldzhyan and Fldzhyan Bell models distinguish themselves by maintaining a slightly lower training loss trajectory than the Clements-based variants, suggesting a more efficient optimization process.

At the end of training, as reported in the fourth row of Table III, confirm these observations and reveal a compelling narrative. The Fldzhyan PNN was dominant during training and validation, securing both the lowest final training loss (0.0588) and the highest validation accuracy (97.16%). Based on these standard metrics, it would appear to be the optimal model. However, the evaluation on the unseen test set presented a counterintuitive and critical result. The Fldzhyan Bell PNN, despite achieving the lowest validation accuracy of the group (94.86%), delivered the highest overall test accuracy of 97.60%. This finding strongly suggests that the Fldzhyan Bell model developed a more robustly generalizable representation, potentially avoiding subtle overfitting to the validation set that may have affected the other models. Furthermore, the peak test accuracy was 97.60%, indicating that the framework is capable of training PNNs effectively, obtaining similar results compared to those found in the literature, where for different PNNs, the accuracies range from 85% to 97% [17], [44], [45].

The Olivetti Faces dataset, with 400 samples and 1,024 features, constitutes the most demanding benchmark in terms of feature dimensionality. Figure 13 reveals significant training loss and validation accuracy instability. The learning curves exhibit frequent oscillations and a lack of convergence, particularly for Fldzhyan PNN. These fluctuations suggest challenges in optimization, likely due to the small sample size relative to input complexity.

Despite these difficulties, Fldzhyan and Fldzhyan Bell achieved the highest validation accuracy (67.50%), although both recorded the highest training loss (0.5636). In contrast, Clements Bell yielded a lower training loss (0.4915) but slightly reduced accuracy (64.37%). These results, summarized in Table III, highlight a trade-off between optimization efficiency and classification performance. The performance trends in Figure 13 suggest that further tuning or regularization strategies may be required to achieve stable convergence on tasks involving extremely high-dimensional inputs. Such further tuning is a limitation in PNNs rather than a simulation tool limitation.

The experimental results demonstrate that PNNs exhibit stable learning on simpler tasks, increased convergence variability with moderate complexity, and significant optimization challenges on high-dimensional problems, with mesh architecture and sample-to-feature ratio critically influencing performance, while the framework overall proves scalable and effective across a broad spectrum of visual recognition tasks, paving the way for future advances in photonic computing.

V. Conclusion

The growing complexity of photonic circuits exposes limitations in current PNNs simulation tools, particularly in terms of scalability and memory efficiency. The evaluation in this work shows that these tools encounter difficulties when training with large batch sizes and large photonic meshes. To address this, the Slicing Method for transfer-matrix computation was introduced and integrated into the LuxIA framework for PNN simulation and training. The Slicing Method reduces memory usage and execution time and supports multiple architectures and datasets, improving flexibility and scalability. These results indicate that LuxIA can train larger, more complex PNNs with lower computational cost than existing software frameworks. Future work includes extending the framework to even larger datasets and architectures and incorporating more detailed hardware-level non-idealities.

VI. Data Availability

The code supporting the findings of this study is available in the public GitHub repository LuxIA at https://github.com/smilies-polito/Luxia. The repository contains the project description and hosts the implementation and reproducibility materials for LuxIA.

References

[1] A. Reuther *et al.*, "AI and ML Accelerator Survey and Trends," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2022, pp. 1–10.

[2] H. H. Radamson *et al.*, "State of the Art and Future Perspectives in Advanced CMOS Technology," *Nanomaterials*, vol. 10, no. 8, p. 1555, Aug. 2020.

[3] K.-i. Kitayama *et al.*, "Novel frontier of photonics for data processing—Photonic accelerator," *APL Photonics*, vol. 4, no. 9, p. 090901, Sep. 2019.

[4] F. Pavanello *et al.*, "NEUROPULS: NEUROmorphic energy-efficient secure accelerators based on Phase change materials aUgmented siLicon photonicS," in *2023 IEEE European Test Symposium (ETS)*, May 2023, pp. 1–6.

[5] A. Tsakyridis *et al.*, "Photonic neural networks and optics-informed deep learning fundamentals," *APL Photonics*, vol. 9, no. 1, p. 011102, Jan. 2024.

[6] F. Ashtiani, A. J. Geers, and F. Aflatouni, "An on-chip photonic deep neural network for image classification," *Nature*, vol. 606, no. 7914, pp. 501–506, Jun. 2022.

[7] S. N. Khonina, N. L. Kazanskiy, R. V. Skidanov, and M. A. Butt, "Exploring Types of Photonic Neural Networks for Imaging and Computing—A Review," *Nanomaterials (Basel)*, vol. 14, no. 8, p. 697, Apr. 2024.

[8] M. Reck, A. Zeilinger, H. J. Bernstein, and P. Bertani, "Experimental realization of any discrete unitary operator," *Phys. Rev. Lett.*, vol. 73, no. 1, pp. 58–61, Jul. 1994.

[9] F. Marchesin *et al.*, "Braided interferometer mesh for robust photonic matrix-vector multiplications with non-ideal components," *Opt. Express, OE*, vol. 33, no. 2, pp. 2227–2246, Jan. 2025.

[10] T. Fu *et al.*, "Optical neural networks: Progress and challenges," *Light Sci Appl*, vol. 13, no. 1, p. 263, Sep. 2024.

[11] "Optical Device," https://www.3ds.com/products/simulia/electromagnetic-simulation/optical-device, Sep. 2023.

[12] "RSoft Photonic Device Tools | Component Design Software | Synopsys," https://www.synopsys.com/photonic-solutions/rsoft-photonic-device-tools/rsoft-products.html.

[13] "COMSOL: Multiphysics Software for Optimizing Designs," https://www.comsol.com/.

[14] S. Ploeg, H. Gunther, and R. M. Camacho, "Simphony: An open-source photonic integrated circuit simulation framework," in *Frontiers in Optics / Laser Science*. Optica Publishing Group, 2020, p. FW7F.5.

[15] F. Laporte, J. Dambre, and P. Bienstman, "Highly parallel simulation and optimization of photonic circuits in time and frequency domain based on the deep-learning framework PyTorch," *Sci Rep*, vol. 9, no. 1, p. 5918, Apr. 2019.

[16] S. Pai, B. Bartlett, O. Solgaard, and D. A. B. Miller, "Matrix optimization on universal unitary photonic devices," *Phys. Rev. Applied*, vol. 11, no. 6, p. 064044, Jun. 2019.

[17] I. A. D. Williamson *et al.*, "Reprogrammable Electro-Optic Nonlinear Activation Functions for Optical Neural Networks," *IEEE Journal of Selected Topics in Quantum Electronics*, vol. 26, no. 1, pp. 1–12, Jan. 2020.

[18] S. Shekhar *et al.*, "Roadmapping the next generation of silicon photonics," *Nat Commun*, vol. 15, no. 1, p. 751, Jan. 2024.

[19] W. R. Clements, P. C. Humphreys, B. J. Metcalf, W. S. Kolthammer, and I. A. Walmsley, "Optimal design for universal multiport interferometers," *Optica, OPTICA*, vol. 3, no. 12, pp. 1460–1465, Dec. 2016.

[20] S. A. Fldzhyan, M. Y. Saygin, and S. P. Kulik, "Optimal design of error-tolerant reprogrammable multiport interferometers," *Opt. Lett.*, vol. 45, no. 9, p. 2632, May 2020.

[21] B. A. Bell and I. A. Walmsley, "Further compactifying linear optical unitaries," *APL Photonics*, vol. 6, no. 7, p. 070804, 07 2021. [Online]. Available: https://doi.org/10.1063/5.0053421

[22] R. A. Fisher, "Iris," UCI Machine Learning Repository, 1936, DOI: https://doi.org/10.24432/C56C76.

[23] A. Seewald, "Digits - A Dataset for Handwritten Digit Recognition," Austrian Research Institute for Artificial Intelligence, Tech. Rep., Jan. 2005.

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

[25] F. Samaria and A. Harter, "Parameterisation of a stochastic model for human face identification," in *Proceedings of 1994 IEEE Workshop on Applications of Computer Vision*, Dec. 1994, pp. 138–142.

[26] M. Makarenko, Q. Wang, A. Burguete-Lopez, and A. Fratalocchi, "Photonic optical accelerators: The future engine for the era of modern AI?" *APL Photonics*, vol. 8, no. 11, p. 110902, Nov. 2023.

[27] T. Andrulis, G. I. Chaudhry, V. M. Suriyakumar, J. S. Emer, and V. Sze, "Architecture-level modeling of photonic deep neural network accelerators," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 307–309.

[28] A. Sludds *et al.*, "WDM-Enabled Photonic Edge Computing," in *2022 27th OptoElectronics and Communications Conference (OECC) and 2022 International Conference on Photonics in Switching and Computing (PSC)*. Toyama, Japan: IEEE, Jul. 2022, pp. 1–3.

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

[30] C. Wu *et al.*, "Programmable phase-change metasurfaces on waveguides for multimode photonic convolutional neural network," *Nat Commun*, vol. 12, no. 1, p. 96, Jan. 2021.

[31] S. Pai *et al.*, "Parallel fault-tolerant programming of an arbitrary feed-forward photonic network," Sep. 2019.

[32] J. Cheng, H. Zhou, and J. Dong, "Photonic Matrix Computing: From Fundamentals to Applications," *Nanomaterials (Basel)*, vol. 11, no. 7, p. 1683, Jun. 2021.

[33] S. Banerjee, M. Nikdast, and K. Chakrabarty, "Modeling Silicon-Photonic Neural Networks under Uncertainties," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Feb. 2021, pp. 98–101.

[34] T. G. Mackay and A. Lakhtakia, *The Transfer-Matrix Method in Electromagnetics and Optics*, ser. Synthesis Lectures on Electromagnetics. Cham: Springer International Publishing, 2020.

[35] F. Laporte, J. Dambre, and P. Bienstman, "Photontorch: Simulation and Optimization of Large Photonic Circuits Using the Deep Learning Framework PyTorch," in *2019 IEEE Photonics Society Summer Topical Meeting Series (SUM)*, Jul. 2019, pp. 1–2.

[36] N. C. Harris *et al.*, "Quantum transport simulations in a programmable nanophotonic processor," *Nature Photon*, vol. 11, no. 7, pp. 447–452, Jul. 2017.

[37] O. Destras, S. Le Beux, F. G. De Magalhães, and G. Nicolescu, "Survey on Activation Functions for Optical Neural Networks," *ACM Comput. Surv.*, vol. 56, no. 2, pp. 35:1–35:30, Sep. 2023.

[38] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[39] J. K. S. Poon, J. Scheuer, Y. Xu, and A. Yariv, "Designing coupled-resonator optical waveguide delay lines," *J. Opt. Soc. Am. B, JOSAB*, vol. 21, no. 9, pp. 1665–1673, Sep. 2004.

[40] I. A. D. Williamson *et al.*, "Tunable Nonlinear Activation Functions for Optical Neural Networks," in *Conference on Lasers and Electro-Optics (2020), Paper SM1E.2*. Optica Publishing Group, May 2020, p. SM1E.2.

[41] T. W. Hughes, M. Minkov, Y. Shi, and S. Fan, "Training of photonic neural networks through in situ backpropagation and gradient measurement," *Optica, OPTICA*, vol. 5, no. 7, pp. 864–871, Jul. 2018.

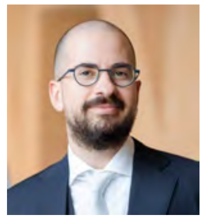
[42] J. Wenhao and L. Zheng, "Vulnerability Analysis and Security Research of Docker Container," in *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, Sep. 2020, pp. 354–357.

[43] L. Gao, Y. Zhang, J. Han, and J. Callan, "Scaling deep contrastive learning batch size under memory limited setup," in *Proceedings of the 6th Workshop on Representation Learning for NLP (RepL4NLP-2021)*, A. Rogers *et al.*, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 316–321. [Online]. Available: https://aclanthology.org/2021.repl4nlp-1.31/

[44] M. J. Filipovich *et al.*, "Silicon photonic architecture for training deep neural networks with direct feedback alignment," *Optica, OPTICA*, vol. 9, no. 12, pp. 1323–1332, Dec. 2022.

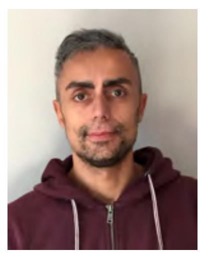
[45] I. Roumpos *et al.*, "Silicon integrated photonic-electronic neuron for noise-resilient deep learning," *Opt. Express*, vol. 32, no. 20, p. 34264, Sep. 2024.

Federico Marchesin is a PhD researcher in the Photonics Research Group at Ghent University with a background in electronics from the University of Padua. His research focuses on neuromorphic photonics, specifically integrated silicon matrix-vector multiplications for neural network accelerator applications. He specializes in the simulation and modeling of photonic circuits and has significant experience in silicon integrated design.

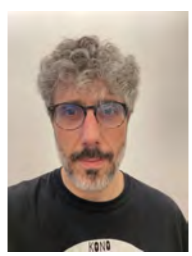


Marco P. Abrate holds a Master's degree in data science from EPFL and a Bachelor's degree in computer engineering at Politecnico di Torino, and he is a PhD candidate in neuroscience and AI at University College London. His research focuses on modelling the maturation of spatial representations in the hippocampus of developing rats using novel biologically plausible recurrent neural networks.

Peter Bienstman (Member, IEEE) He received the degree in electrical engineering from Ghent University, Ghent, Belgium, in 1997, and the Ph.D. degree from the Department of Information Technology (INTEC), Ghent University, in 2001 where he is currently a full professor. His research interests include several applications of nanophotonics. He has been awarded an ERC starting grant for the Naresco-project: Novel paradigms for massively parallel nanophotonic information processing.



Stefano Di Carlo (SM'00-M'03-SM'11) He received an M.Sc. degree in computer engineering and a Ph.D. in information technologies from Politecnico di Torino, Italy, where he is a Full Professor. His research interests include resilient and secure hardware architecture design, emerging computing paradigms, artificial intelligence, and neuromorphic computing. He is a Golden Core member of the IEEE Computer Society.



Alessandro Savino (M'14, SM'22) is an Associate Professor in the Department of Control and Computer Engineering at Politecnico di Torino (Italy). He holds a Ph.D. (2009) and an M.S. equivalent (2005) in Computer Engineering and Information Technology from the Politecnico di Torino in Italy. Dr. Savino's research contributions include Approximate Computing, Reliability Analysis, Safety-Critical Systems, Software-Based Self-Test, and Image Analysis.

Tzamn Melendez Carmona holds a M.S. equivalent in Electronics from Politecnico di Torino and he is a PhD student in Artificial Intelligence at Politecnico di Torino. His research focuses on the integration of photonic neural networks and RISC-V compliant interfaces to enhance system-level efficiency, with particular emphasis on the training dynamics and simulation of photonic computing architectures.