Acceleration Method For Learning Fine-Layered Optical Neural Networks

***Abstract*—An optical neural network (ONN) is a promising system due to its high-speed and low-power operation. Its linear unit performs a multiplication of an input vector and a weight matrix in optical analog circuits. Among them, a circuit with a multiple-layered structure of programmable Mach-Zehnder interferometers (MZIs) can realize a specific class of unitary matrices with a limited number of MZIs as its weight matrix. The circuit is effective for balancing the number of programmable MZIs and ONN performance. However, it takes a lot of time to learn MZI parameters of the circuit with a conventional automatic differentiation (AD), which machine learning platforms are equipped with. To solve the time-consuming problem, we propose an acceleration method for learning MZI parameters. We create customized complex-valued derivatives for an MZI, exploiting Wirtinger derivatives and a chain rule. They are incorporated into our newly developed function module implemented in C++ to collectively calculate their values in a multi-layered structure. Our method is simple, fast, and versatile as well as compatible with the conventional AD. We demonstrate that our method works 20 times faster than the conventional AD when a pixel-by-pixel MNIST task is performed in a complex-valued recurrent neural network with an MZI-based hidden unit.**

***Index Terms*—Machine learning, Optical neural networks, Complex-valued neural networks, Unitary matrix, Wirtinger derivatives, Backpropagation, Mach-Zehnder interferometer**

I. Introduction

Optical neural networks (ONNs) have attracted much attention because of their high speed and extremely low power consumption [1], [2], [3], [4], [5], [6], compared with conventional digital computer systems. ONNs process information encoded by amplitude and phase of light waves in passive analog circuits exploiting optical phenomena such as transmission, resonance, interference, and diffraction. Such optical analog circuits operate in principle without energy. Linear units in ONNs perform analog multiplication of a weight matrix and an input vector that is a signal of encoded information. In the multiplication exploiting the interference [2], [7], [8], [9], signals and weights are regarded as complex numbers. Since the energy is preserved in the vector-matrix multiplication, the weight matrix is a unitary matrix.

Among such linear units, there are ones with a unitary matrix implemented in programmable Mach-Zehnder interferometers (MZIs) [7], [10], [11]. The linear unit using MZIs has two main advantages for designing ONNs: One is that the ONNs with the linear units can learn not a unitary matrix itself but parameters of the MZIs directly [12]. The other is that the ONNs can balance their performance and the limited number of their MZIs by using the characteristic that not only full-capacity unitary matrix but also its specific class is realized by MZI-based matrices [10]. For these advantages, we focus on the linear units using MZIs and discuss methods for learning their parameters.

The learning method [12] can generate specific classes of unitary matrices using fewer parameters in an MZI-based matrix than those required for realizing any unitary matrix. However, this method requires a lot of computational cost of elapsed time to learn the parameters. Two main factors make this method costly. One is to represent a weight matrix by a product of several MZI-representation matrices. This is equivalent to adopting a sequence of several linear units, resulting in a deeper neural network. The other is to use the conventional automatic differentiation (AD) [13] without any change, which machine learning frameworks such as TensorFlow [14] and PyTorch [15] are equipped with. The machine learning framework requires more computational time for the deeper neural networks. Besides, we need flexible representations for an MZI to systematically deal with various MZI-representation matrices [4], [10], [11], [12].

To solve the problems, we propose an acceleration learning method to remove the foregoing factors and adapt to the various MZI-representation matrices, retaining the compatibility with the conventional AD. Our proposed learning method is simple, fast, versatile, and easy-to-use. Our method is based on three newly developed techniques. The first is to prepare two constituent unitary matrices corresponding to two basic components of an MZI, a phase sifter (PS) and a directional coupler (DC) (or beam splitter) [7], [10], [11], and represent an MZI matrix by the combination of representation matrices of the PS and the DC. This leads to the versatility of our method and makes a matrix formulation simple. The second is to create customized complex-valued derivatives for matrices of pairs of the PS and the DC, i.e., PSDC and DCPS, by exploiting Wirtinger derivatives and a chain rule [16]. The last is to develop a function module implemented in C++ to collectively calculate values required in learning of a linear unit with the several combination matrices. By incorporating the customized derivatives to the function module, our method achieves fast learning.

Our contributions are threefold:

1) We present a fine-layered linear unit where a weight matrix is represented by a product of structured unitary matrices implemented in phase shifters (PS) and directional couplers (DC). Owing to this representation, we can simply formulate a weight matrix and easily modify it according to various MZI implementations.
2) We propose an acceleration learning method for a fine-layered linear unit. We create customized complex-

valued derivatives for products of the structured matrices, exploiting a chain rule and Wirtinger derivatives. The customized derivatives are incorporated to our newly developed function module implemented in C++ to collectively calculate their values. Since our proposed method is compatible with automatic differentiation (AD), we can easily use it in the conventional machine learning platforms.
3) We demonstrate that our proposed method works 20 times faster than the conventional AD when a pixel-by-pixel MNIST task [12] is performed in a complex-valued recurrent neural network where a hidden unit is a fine-layered linear unit based on the PSDC.

The remainder of this paper consists of the following six sections. Section II briefly reviews related work. Section III describes unitary matrices represented by MZIs. Section IV provides background knowledge on learning a complex-valued linear unit for understanding our proposed learning method. Section V explains our learning method in detail. Section VI shows our experimental settings and demonstrates the results. The final section provides our conclusion and future work.

II. Related Work

This section reviews two topics regarding how to determine parameters of programmable MZIs in a unitary matrix.

A. Constructing Linear Units By Using MZIs

A method in [7] constructs a linear unit with a weight matrix implemented in programmable MZIs. What is learned with the method is neither a unitary matrix nor parameters of programmable MZIs but a weight matrix itself. It first obtains optimized weight matrix W by learning a weight matrix with a conventional algorithm (e.g., [17]). W is decomposed with singular value decomposition (SVD) as $W=U\Sigma V^{\dagger}$, where U denotes a unitary matrix, $V^{\dagger}$ denotes the conjugate transpose of unitary matrix V, and Σ denotes a rectangular diagonal matrix. The unitary matrices U and $V^{\dagger}$ are implemented in programmable MZIs by a triangular-structure implementation method [18], [19]. The diagonal matrix Σ can be implemented in optical attenuators and phase shifters. Instead of the triangular-structure implementation method, we can employ a rectangular-structure method [11], [20]. Although this scheme can implement any unitary matrix in MZIs, it can not generate a specific class of unitary matrices by fewer MZIs. This is a problem when designing a higher-performance ONN using limited physical resources.

B. Learning Methods For Unitary Matrices

Neural networks with unitary matrices as their weight matrices have been studied to alleviate a problem of vanishing or exploding gradients in weight optimization [22], [25], [28]. Table I summarizes learning methods for unitary matrices. There are two types of constraints for generating a unitary matrix, optimization and structural constraints. A generated unitary matrix has a distinct representation capacity from a fixed specified class to a full-capacity unitary representation.

TABLE I
Learning Methods for Unitary Matrices

Constraint	Optimization	Structure		
Matrix representation capacity	Full unitary		Specified class	
			Fixed	Variable (to full)
Method	[21], [22] [26], [27]	[23]	[24] [28]	[25], [12] Ours

In the methods based on optimization constraints, a convenient one is to add the constraint to a loss function as a regularizer [21]. A more strict method is to optimize a weight matrix along Stiefel manifold whose tangent spaces are endowed with a Riemannian metric, using geodesic gradient descent [22], [26], [27]. These methods can generate a full-capacity unitary matrix. By contrast, the methods based on structural constraints generate a unitary matrix expressed by a product of structured unitary matrices such as Givens rotation [12], [24], Householder reflection [25], [28], and skew-Hermitian matrices [23]. Depending on the constituent structured unitary matrices and their parameterization, the generated matrix has unique representation capacity. Unitary matrices by the methods [25], [12] vary their capacity from a specified-class to full unitary while those [24], [28] have fixed and restricted capacity.

From the viewpoint of optical-circuit implementations, the optimization-constraint approach has a serious problem that it is difficult to obtain an exact unitary matrix. The structure-constraint approach has an advantage of being able to construct an exact unitary matrix. In particular, the method [12], which is suitable to the MZI implementation of a unitary matrix, can generate specific classes of unitary matrices using fewer parameters in structured matrices. However, this method needs a lot of computational time for learning the MZI parameters.

Thus the previous methods have some problems to realize linear units in ONNs with unitary matrices implemented in the MZIs. Our proposed learning method in Table I solves the problem that the method in [12] has and furthermore improves versatility of representation matrices of the MZIs.

III. Unitary Matrices Represented By MZIs

We show that an MZI is represented by various unitary matrices depending on its structure and describe that any $n\times n$ unitary matrix is realized by a product of MZI-representation matrices and a diagonal unitary matrix.

A. Representation Matrix of MZI

We first define a unitary matrix. Let $a_{(n)}$ denote an element of the n-dimensional unitary group $U(n)$ and $A_{(n)}$ denote the representation matrix of $a_{(n)}$. Then $n\times n$ unitary matrix $A_{(n)}$ satisfies the unitary constraints of $A_{(n)}A^{\dagger}_{(n)}=I$ and $\left|\det A_{(n)}\right|=1$, where I denote the $n\times n$ identity matrix. When

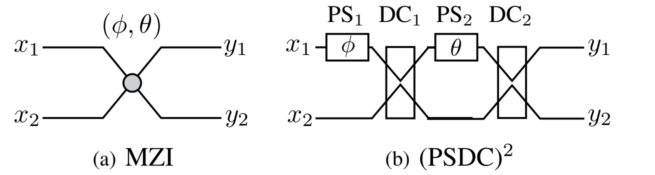


Fig. 1. Symbol of (a) MZI and structure of (b) $(\text{PSDC})^2$. $x_1,\ x_2 \in \mathbb{C}$ and $y_1,\ y_2 \in \mathbb{C}$ denote input and output complex values and ϕ and θ are real-valued parameters.

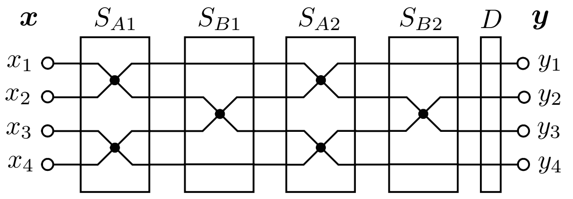


Fig. 2. Rectangular structure consisting of a product of block-diagonal matrices $(S_{A1}, S_{B1}, S_{A2}, S_{B2})$ and one diagonal unitary matrix (D) that realizes any 4×4 unitary matrix.

$n=2$, any 2×2 unitary matrix $A_{(2)}$ has four independent real-number parameters corresponding to the degree of freedom of the unitary group $U(2)$.

An MZI is a two-port optical circuit in Fig. 1(a) and linearly transforms complex-valued input vector $(x_1, x_2)^T$ to output vector $(y_1, y_2)^T$ by a variable transformation matrix with two parameters of ϕ and θ, where $(x_1, x_2)^T$ denotes the transpose of (x_1, x_2), i.e., the input vector is represented as the column vector. The MZI consists of two programmable phase shifters (PS) and two directional couplers (DC) with a fixed 0.5:0.5 power split ratio. We adopt a symbol for an MZI shown in Fig. 1(a) and illustrate a typical structure for the MZI, which is the serial connection of the PS and the DC, i.e., (PSDC)(PSDC) or $(\text{PSDC})^2$ in Fig. 1(b). Representation matrices of the programmable PS and the fixed DC, $M_{[PS(\phi)]}$ and $M_{[DC]}$, are expressed as

$$M_{[PS(\phi)]} = \begin{pmatrix} e^{i\phi} & 0 \\ 0 & 1 \end{pmatrix}, \quad M_{[DC]} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix}, \tag{1}$$

where $0 \le \phi \le 2\pi$ and $i^2 = -1$. Since $M_{[PS(\phi)]}$ and $M_{[DC]}$ satisfy the unitary constraints, a product of the PS- and the DC-representation matrices becomes a unitary matrix.

An actual transformation matrix depends on the connection of PS's and DC's. For instance, Fang's matrix (R_F) [10] corresponding to the structure in Fig. 1(b) is expressed by

$$\begin{aligned} R_F &= M_{[DC]}\, M_{[PS(\theta)]}\, M_{[DC]}\, M_{[PS(\phi)]} \\ &= ie^{i\frac{\theta}{2}} \begin{pmatrix} e^{i\phi}\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \\ e^{i\phi}\cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} e^{i\phi}\beta & i\alpha \\ ie^{i\phi}\alpha & -\beta \end{pmatrix}, \\ &\qquad \alpha = e^{i\theta}+1, \quad \beta = e^{i\theta}-1\,. \end{aligned} \tag{2}$$

Pai's matrix (R_P) [11] representing (DCPS)(DCPS) is the transpose matrix of R_F as

$$R_P = M_{[PS(\theta)]}\, M_{[DC]}\, M_{[PS(\phi)]}\, M_{[DC]} = R_F^T\,. \tag{3}$$

Besides, a matrix (R_M) for (DCPS)(PSDC) is expressed by

$$\begin{aligned} R_M &= M_{[DC]}\, M_{[PS(\theta)]}\, M_{[PS(\phi)]}\, M_{[DC]} \\ &= \frac{1}{2}\begin{pmatrix} e^{i\phi}-e^{i\theta} & i(e^{i\phi}+e^{i\theta}) \\ i(e^{i\phi}+e^{i\theta}) & -(e^{i\phi}-e^{i\theta}) \end{pmatrix} \end{aligned} \tag{4}$$

An MZI with two parameters is represented with three distinct matrices of R_F, R_P, and R_M if each of the two phases ϕ and θ is regarded as relative phase and its initial phase difference is ignored. Thus there are various MZI-representation matrices.

When selecting one of the three MZI-representation matrices, e.g., R_F, we can realize any 2×2 unitary matrix by Clements' method [20] as

$$A_{(2)} = D \cdot R_F\,, \quad D = \begin{pmatrix} e^{i\delta_0} & 0 \\ 0 & e^{i\delta_1} \end{pmatrix}, \tag{5}$$

where $0 \le \delta_0,\ \delta_1 \le 2\pi$. $A_{(2)}$ with four parameters is expressed by a product of a diagonal unitary matrix with two parameters and an MZI-representation matrix with two parameters.

B. Product of MZI-Representation Matrices

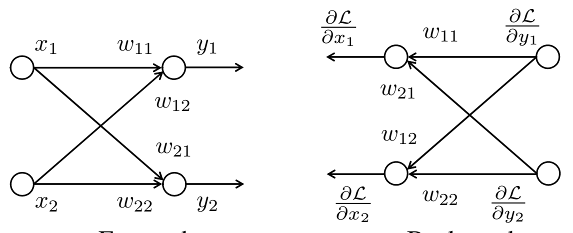
Using the 2×2 MZI-representaion matrix, we consider an $n\times n$ unitary matrix corresponding to an n-port optical circuit based on MZIs. Let us define $n\times n$ unitary matrix $T_{(p,q:n)}$ represented by a single MZI as

$$T_{(p,q:n)} := \begin{pmatrix} 1 & 0 & \cdots & & \cdots & 0 & 0 \\ 0 & 1 & & & & & 0 \\ \vdots & & w_{pp} & & w_{pq} & & \vdots \\ \vdots & & w_{qp} & & w_{qq} & & \vdots \\ \vdots & & & & & 1 & 0 \\ 0 & 0 & \cdots & & \cdots & 0 & 1 \end{pmatrix}, \tag{6}$$

where $p < q \le n \in \mathbb{Z}$ and $w_{pq} \in \mathbb{C}$ denotes the pth-row and qth-column element. The others, w_{pp}, w_{qp}, and w_{qq}, denote the elements according to the same rule. The elements of w_{pp}, w_{pq}, w_{qp}, and w_{qq} correspond to w_{11}, w_{12}, w_{21}, and w_{22}, of a 2×2 MZI-representation matrix, respectively. Then $n\times n$ matrix $T_{(p,q:n)}$ has two independent real-valued parameters such as ϕ and θ. For instance, when R_F in Eq. (2) is used, $w_{pp}=e^{i\phi}\beta/2$, $w_{pq}=i\alpha/2$, $w_{qp}=ie^{i\phi}\alpha/2$, $w_{qq}=-\beta/2$.

Any $n\times n$ unitary matrix is decomposed to $n(n-1)/2$ $T_{(p,q:n)}$'s and a single $n\times n$ diagonal unitary matrix D with n parameters by Clements' method [20]. The method sequentially determines two parameters of each $T_{(p,q:n)}$ and n parameters of the D by the procedure similar to Gaussian elimination. By changing the order of commutative matrices in the obtained $T_{(p,q:n)}$'s, the method generates a product of unitary matrices S with a regular rectangular structure. An example of the commutative matrices $T_{(p,q:n)}$'s is shown as follows. 4×4 unitary matrix $S_{((1,2),(3,4):4)} = T_{(1,2:4)}T_{(3,4:4)} = T_{(3,4:4)}T_{(1,2:4)}$.

Figure 2 shows a diagram of a linear unit with the rectangular structure generated by Clements' method, which realizes



x_1
w_{11}
y_1
w_{12}
w_{21}
x_2
w_{22}
y_2
$\frac{\partial \mathcal{L}}{\partial x_1}$
$\frac{\partial \mathcal{L}}{\partial y_1}$
$\frac{\partial \mathcal{L}}{\partial x_2}$
$\frac{\partial \mathcal{L}}{\partial y_2}$

(a) Forward (b) Backward

Fig. 3. Diagram representing the relationship between inputs and outputs for (a) forward and (b) backward process in a generic linear unit.

any 4×4 unitary matrix. The foregoing product $S_{((1,2),(3,4):4)}$ is S_{A1} and $S_{(2,3):4} = T_{(2,3:4)}$ is S_{B1}. Given R_F in Eq. (2) as the MZI-representation matrix, $S_{A1} = S_{((1,2),(3,4):4)}$ and $S_{B1} = S_{((2,3):4)}$ are expressed by

$$S_{A1} = \frac{1}{2}\begin{pmatrix} e^{i\phi_1}\beta_1 & i\alpha_1 & 0 & 0 \\ ie^{i\phi_1}\alpha_1 & -\beta_1 & 0 & 0 \\ 0 & 0 & e^{i\phi_2}\beta_2 & i\alpha_2 \\ 0 & 0 & ie^{i\phi_2}\alpha_2 & -\beta_2 \end{pmatrix}, \tag{7}$$

$$S_{B1} = \frac{1}{2}\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & e^{i\phi_3}\beta_3 & i\alpha_3 & 0 \\ 0 & ie^{i\phi_3}\alpha_3 & -\beta_3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}. \tag{8}$$

In the rectangular structure for realizing an $n\times n$ unitary matrix, $S_{A(L)}$ and $S_{B(L)}$, which we term A-type and B-type fine layers, alternately align. $S_{A(L)}$ and $S_{B(L)}$ have $\lfloor n/2 \rfloor$ and $\lfloor (n-1)/2 \rfloor$ MZIs. Regarding L, $L = \lceil n/2 \rceil$ for $S_{A(L)}$ and $L = \lfloor n/2 \rfloor$ for $S_{B(L)}$, $n \geq 3$, i.e., the total number of fine layers is n for $n \geq 3$ and 1 for $n = 2$.

In addition to the structural regularity, the rectangular structure has an advantage that we can vary matrix representation capacity with the number of fine layers from a specific class to a full-capacity unitary matrix. This characteristic allows us to control linear-unit performance by the number of optimized parameters, which corresponds to the number of MZIs in physical resources. For this reason, we select a unitary matrix with the rectangular structure for our proposed learning method.

IV. Learning Complex-Valued Linear Units

We begin by a generic real-valued linear unit to review learning of linear units based on the backpropagation algorithm with gradient descent and automatic differentiation (AD). Then we extend the real-valued AD to complex-valued one using Wirtinger derivatives.

A. Learning Linear Units and Automatic Differentiation

Automatic differentiation (AD) is equipped with most of machine learning platforms and utilized for training neural networks. It is based on the principle that all numerical computations are ultimately compositions of a finite set of elementary operations for which the derivatives are known [13]. A chain rule combines the derivatives of the constituent operations and provides the derivatives of the overall composition.

As an example for understanding the backpropagation and the AD, we consider a simple real-valued linear unit consisting of two nodes in Fig. 3. Assume that a neural network containing this linear unit performs a classification task and its result is evaluated by loss function $\mathcal{L}$. Let W be a 2×2 real-valued matrix whose element $w_{jh} \in \mathbb{R}$, $j, h = 1, 2$, and column vector $\boldsymbol{x} = (x_1, x_2)^T$, $x_1, x_2 \in \mathbb{R}$, where $(x_1, x_2)^T$ denote the transpose of (x_1, x_2). The forward process in Fig. 3(a) is expressed by

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}. \tag{9}$$

In training of the linear unit, weight w_{jh} is updated by

$$w_{jh} \leftarrow w_{jh} - \eta \nabla \mathcal{L}, \tag{10}$$

where η denotes the learning rate and $\nabla\mathcal{L} = \partial\mathcal{L}/\partial w_{jh}$ in this case. The backward process in Fig. 3(b) is expressed by

$$\begin{pmatrix} \partial\mathcal{L}/\partial x_1 \\ \partial\mathcal{L}/\partial x_2 \end{pmatrix} = W^T \begin{pmatrix} \partial\mathcal{L}/\partial y_1 \\ \partial\mathcal{L}/\partial y_2 \end{pmatrix}, \tag{11}$$

$$\frac{\partial\mathcal{L}}{\partial w_{jh}} = x_h \frac{\partial\mathcal{L}}{\partial y_j}. \tag{12}$$

Note that transformation matrices of the forward and the backward process are the transpose of each other. Eqs. (11) and (12) are derived using a chain rule as follows.

$$\frac{\partial\mathcal{L}}{\partial x_h} = \sum_{j=1}^{2} \frac{\partial\mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial x_h} = \sum_{j=1}^{2} w_{jh}\frac{\partial\mathcal{L}}{\partial y_j}, \tag{13}$$

$$\frac{\partial\mathcal{L}}{\partial w_{jh}} = \frac{\partial\mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial w_{jh}} = x_h \frac{\partial\mathcal{L}}{\partial y_j}. \tag{14}$$

B. Complex-Valued Derivative

In the case of a classification task, loss function $\mathcal{L}$ is a real-valued function even in a complex-valued neural network. We can extend the real-valued AD to complex-valued one using the following Wirtinger derivatives.

Definition 1 (Wirtinger derivatives [16]). *Let f be a real-valued non-analytic function of $z \in \mathbb{C}$, e.g., $f : \mathbb{C} \to \mathbb{R}$. Let $z = \mathrm{Re}(z) + i\,\mathrm{Im}(z)$ and $z^* = \mathrm{Re}(z) - i\,\mathrm{Im}(z)$, where $i^2 = -1$ and $\mathrm{Re}(z)$ and $\mathrm{Im}(z)$ are functions that return the real and the imaginary part of z, then Wirtinger derivatives of f with respect to z and z^* are defined as*

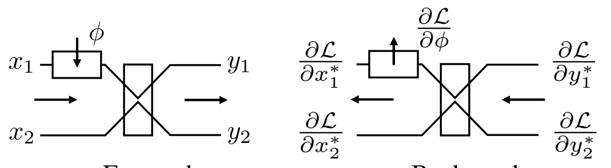
$$\frac{\partial f}{\partial z} = \frac{1}{2}\left(\frac{\partial f}{\partial\,\mathrm{Re}(z)} - i\frac{\partial f}{\partial\,\mathrm{Im}(z)}\right) \tag{15}$$

$$\frac{\partial f}{\partial z^*} = \frac{1}{2}\left(\frac{\partial f}{\partial\,\mathrm{Re}(z)} + i\frac{\partial f}{\partial\,\mathrm{Im}(z)}\right), \tag{16}$$

where z and z^* are regarded as independent variables [29].

Note that Eq. (17) holds from Eqs. (15) and (16).

$$\left(\frac{\partial f}{\partial z}\right)^* = \frac{\partial f}{\partial z^*}. \tag{17}$$



ϕ
x_1
y_1
x_2
y_2
(a) Forward
$\frac{\partial\mathcal{L}}{\partial\phi}$
$\frac{\partial\mathcal{L}}{\partial x_1^*}$
$\frac{\partial\mathcal{L}}{\partial y_1^*}$
$\frac{\partial\mathcal{L}}{\partial x_2^*}$
$\frac{\partial\mathcal{L}}{\partial y_2^*}$
(b) Backward

Fig. 4. Diagram representing the relationship between inputs and outputs for (a) forward and (b) backward process in PSDC module.

Suppose that $w_{jh}, x_h \in \mathbb{C}$ in Eq. (9). Using the gradient descent, weight w_{jh} is updated as

$$w_{jh} \leftarrow w_{jh} - \eta' \, \nabla\mathcal{L} \, , \tag{18}$$

where η' denotes a tentative learning rate. The gradient of real-valued loss function $\mathcal{L}$ defined on complex plane $z = \mathrm{Re}(z) + i\,\mathrm{Im}(z)$ is expressed by

$$\nabla\mathcal{L} = \frac{\partial\mathcal{L}}{\partial\,\mathrm{Re}(w_{jh})} + i\frac{\partial\mathcal{L}}{\partial\,\mathrm{Im}(w_{jh})} = 2\cdot\left(\frac{\partial\mathcal{L}}{\partial w_{jh}^*}\right) . \tag{19}$$

Then the weight w_{jh} is updated as

$$w_{jh} \leftarrow w_{jh} - \eta\,\frac{\partial\mathcal{L}}{\partial w_{jh}^*} \, . \tag{20}$$

The backward process is expressed by

$$\begin{pmatrix}\partial\mathcal{L}/\partial x_1^* \\ \partial\mathcal{L}/\partial x_2^*\end{pmatrix} = W^\dagger \begin{pmatrix}\partial\mathcal{L}/\partial y_1^* \\ \partial\mathcal{L}/\partial y_2^*\end{pmatrix} , \tag{21}$$

$$\frac{\partial\mathcal{L}}{\partial w_{jh}^*} = x_h^* \, \frac{\partial\mathcal{L}}{\partial y_j^*} \, , \tag{22}$$

where $W^\dagger$ denotes the conjugate transpose of W. Thus the complex-valued derivatives are derived by using the chain rule and Wirtinger derivatives defined by Definition 1.

V. Proposed Learning Method

We propose an acceleration method for learning an fine-layered linear unit where a weight matrix is represented by using unitary matrices based on two *basic units* of the PSDC and the DCPS. We first derive customized derivatives utilized in the backward process with automatic differentiation (AD) by using the chain rule and Wirtinger derivatives [29], [30], [31]. Next, we show a function module implemented in C++ which our customized derivatives are incorporated in.

A. Customized Derivatives

An MZI consists of a programmable PS and a DC with 0.5:0.5 power split ratio, two of each in our settings. Then there are three distinct structures of (PSDC)^2, (DCPS)^2, and (DCPS)(PSDC) by preventing the DC-DC sequence described in Section III-A. These matrices are represented by products of two basic matrices of the PSDC and the DCPS. To deal with linear units containing the MZIs with various representations, we prepare customized functions using the two basic matrices for the forward and the backward process in the training, instead of directly using the MZI-representation matrices.

Figures 4(a) and (b) show diagrams representing the relationship between the inputs and the outputs in the forward and the backward process in the PSDC whose structure is the same as the part of PS$_1$ and DC$_1$ in Fig. 1(b). In the forward process, phase ϕ is given as an optimized parameter of the programmable PS besides input vector $(x_1, x_2)^T$. In the backward process, $(\partial\mathcal{L}/\partial y_1^*, \partial\mathcal{L}/\partial y_2^*)^T$ is given and $(\partial\mathcal{L}/\partial x_1^*, \partial\mathcal{L}/\partial x_2^*)^T$ is passed to the next layer. For updating ϕ as $\phi \leftarrow \phi - \eta_\phi(\partial\mathcal{L}/\partial\phi)$, derivative $(\partial\mathcal{L}/\partial\phi)$ is calculated, where η_ϕ denotes the learning rate of ϕ.

Proposition 1. *In the forward process in the PSDC, the linear function of the input* $(x_1, x_2)^T \in \mathbb{C}^2$ *is expressed by*

$$\begin{pmatrix}y_1 \\ y_2\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}e^{i\phi} & i \\ ie^{i\phi} & 1\end{pmatrix}\begin{pmatrix}x_1 \\ x_2\end{pmatrix} . \tag{23}$$

When Eq. (23) holds, in the backward process, the input-output relationship and the derivative $(\partial\mathcal{L}/\partial\phi)$ *are expressed by*

$$\begin{pmatrix}\partial\mathcal{L}/\partial x_1^* \\ \partial\mathcal{L}/\partial x_2^*\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}e^{-i\phi} & -ie^{-i\phi} \\ -i & 1\end{pmatrix}\begin{pmatrix}\partial\mathcal{L}/\partial y_1^* \\ \partial\mathcal{L}/\partial y_2^*\end{pmatrix} , \tag{24}$$

$$\frac{\partial\mathcal{L}}{\partial\phi} = 2\cdot\mathrm{Im}\left(x_1^*\frac{\partial\mathcal{L}}{\partial x_1^*}\right) . \tag{25}$$

Note that the transformation matrix in the backward process is the conjugate transpose of that in the forward process and the derivative $(\partial\mathcal{L}/\partial\phi)$ is expressed by only the information passing through the interface between the phase shifter and the other layer.

Proof. Equation (24) holds based on Eq. (21). By using the chain rule and Wirtinger derivatives, $(\partial\mathcal{L}/\partial\phi)$ is expressed as

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial\phi} &= \sum_{j=1}^{2}\left(\frac{\partial\mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial\phi} + \frac{\partial\mathcal{L}}{\partial y_j^*}\frac{\partial y_j^*}{\partial\phi}\right) \\ &= \frac{i}{\sqrt{2}}\left\{x_1 e^{i\phi}\left(\frac{\partial\mathcal{L}}{\partial y_1} + i\frac{\partial\mathcal{L}}{\partial y_2}\right) - x_1^* e^{-i\phi}\left(\frac{\partial\mathcal{L}}{\partial y_1^*} - i\frac{\partial\mathcal{L}}{\partial y_2^*}\right)\right\} \\ &= i\left(x_1\frac{\partial\mathcal{L}}{\partial x_1} - x_1^*\frac{\partial\mathcal{L}}{\partial x_1^*}\right) \\ &= i\left\{\left(x_1^*\frac{\partial\mathcal{L}}{\partial x_1^*}\right)^* - x_1^*\frac{\partial\mathcal{L}}{\partial x_1^*}\right\} = 2\cdot\mathrm{Im}\left(x_1^*\frac{\partial\mathcal{L}}{\partial x_1^*}\right) .\end{aligned} \tag{26}$$

The penultimate equality is derived by using $(\partial\mathcal{L}/\partial x_1)^* = (\partial\mathcal{L}/\partial x_1^*)$ in Eq. (17) of Wirtinger derivatives. □

Proposition 2. *In the forward process in the DCPS, the linear function of the input* $(x_1, x_2)^T \in \mathbb{C}^2$ *is expressed by*

$$\begin{pmatrix}y_1 \\ y_2\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}e^{i\phi} & ie^{i\phi} \\ i & 1\end{pmatrix}\begin{pmatrix}x_1 \\ x_2\end{pmatrix} . \tag{27}$$

When Eq. (27) holds, in the backward process, the input-output relationship and the derivative $(\partial\mathcal{L}/\partial\phi)$ *are expressed by*

$$\begin{pmatrix}\partial\mathcal{L}/\partial x_1^* \\ \partial\mathcal{L}/\partial x_2^*\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}e^{-i\phi} & -i \\ -ie^{-i\phi} & 1\end{pmatrix}\begin{pmatrix}\partial\mathcal{L}/\partial y_1^* \\ \partial\mathcal{L}/\partial y_2^*\end{pmatrix} , \tag{28}$$

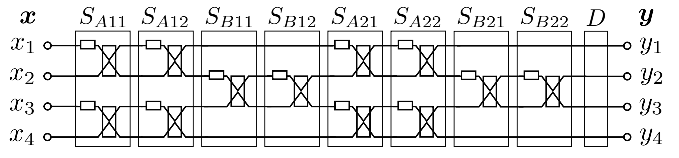


Fig. 5. PSDC-based fine-layered structure with the rectangular structure that realizes any 4×4 unitary matrix.

$$\frac{\partial \mathcal{L}}{\partial \phi} = 2 \cdot \mathrm{Im}\left(y_1^* \frac{\partial \mathcal{L}}{\partial y_1^*} \right) . \tag{29}$$

The transformation matrices in Eqs. (27) and (28) are the transpose of those in Eqs. (23) and (24). Like the PSDC, $(\partial \mathcal{L}/\partial \phi)$ is expressed by only the foregoing information.

Proof. Equation (28) holds based on Eq. (21). By using the chain rule and Wirtinger derivatives, $(\partial \mathcal{L}/\partial \phi)$ is expressed as

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \phi} &= \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial \phi} + \frac{\partial \mathcal{L}}{\partial y_1^*}\frac{\partial y_1^*}{\partial \phi} \\ &= \frac{i}{\sqrt{2}}\left\{ \left(e^{i\phi}x_1 + ie^{i\phi}x_2\right)\frac{\partial \mathcal{L}}{\partial y_1} - \left(e^{-i\phi}x_1^* - ie^{-i\phi}x_2\right)\frac{\partial \mathcal{L}}{\partial y_1^*} \right\} \\ &= i\left(y_1 \frac{\partial \mathcal{L}}{\partial y_1} - y_1^* \frac{\partial \mathcal{L}}{\partial y_1^*}\right) = 2 \cdot \mathrm{Im}\left(y_1^* \frac{\partial \mathcal{L}}{\partial y_1^*}\right) . \end{aligned} \tag{30}$$

□

By using the customized derivatives (CD) of Eqs. (25) and (29), the automatic differentiation does not need to decompose the functions to registered elementary functions such as an exponential function used in Eqs. (24) and (28). This leads to the acceleration of learning of the linear units with the PSDC and the DCPS in Section V-B.

B. Function Module

We design a function module that accelerates learning of a linear unit with *basic units* of the PSDC and the DCPS. Such a linear unit with the PSDC-based fine-layered structure that realizes any 4×4 unitary matrix is shown in Fig. 5, which corresponds to the MZI-based structure in Fig. 2. Input vector $\boldsymbol{x}$ is linearly transformed to output vector $\boldsymbol{y}$ by the sequence of fine layers with unitary matrices, $S_{A11}, S_{A12}, S_{B11}, S_{B12}, \cdots$, and D. The fine layers are regularly connected with each other, that is, the outputs in the jth fine layer are directly connected to the corresponding the inputs in the $(j+1)$th fine layer, where $1 \leq j \leq 8$. For the acceleration, we exploit the regularity in addition to the customized derivatives in Section V-A.

Algorithm 1 shows an overview of the basic-unit process in the linear unit with the fine-layered structure. The linear unit receives n-dimensional complex-valued vector $h_{in} \in \mathbb{C}^n$ and returns the collection of $h_{out} \in \mathbb{C}^{n \times L}$, where L denotes the number of fine layers consisting of the basic units. In each fine layer, h_{in} is transformed to $h_{out(j)}$ by unitary matrix $S_\star \in \mathbb{C}^{n \times n}$, where $\star = A11, A12, B11, B12, \cdots$ and $j = 1, 2, 3, 4, \cdots$ at line 2 and $h_{out(j)}$ is copied to h_{in} at line 3.

Algorithm 1: Basic-unit process in linear unit

```
Input: $h_{in} \in \mathbb{C}^n$, ($L$: Length of $S^{(f)}$–list)
       $S^{(f)}$–list of $[S^{(f)}_{A11}, S^{(f)}_{A12}, S^{(f)}_{B11}, S^{(f)}_{B12}, \cdots]$
       // $S^{(f)}_\star : \mathbb{C}^n \to \mathbb{C}^n$, $S^{(f)}_\star(h) = S_\star h$, given $h$.
       // $\star = (A11, A12, B11, B12, \cdots)$
Output: $h_{out} \in \mathbb{C}^{n \times L}$ // $h_{out}$: Collection of $h_{out(j)}$
1 for $S^{(f)}_\star$ in $S^{(f)}$–list do in order: $j = 1, \cdots, L$
2     $h_{out(j)} \leftarrow \boxed{S^{(f)}_\star(h_{in})}$
3     $\boxed{h_{in} \leftarrow h_{out(j)}}$
4 return $h_{out}$
```

In the conventional AD implemented in the Python-based machine learning frameworks, $S^{(f)}_\star$ is defined only for the forward process at line 2 in Algorithm 1. Then the AD automatically calculates values required in the backward process. Note that lines 2 and 3 are replaced with $h_{in} \leftarrow S^{(f)}_\star(h_{in})$ in Python implementation. Instead of the AD in the backward process, we prepared functions using the customized derivatives (CD), which were implemented as two distinct functions in Python and C++. We call a module with the Python-implementation functions for the forward and the backward process *CDpy* and a module with the C++-implementation functions for both the processes *CDcpp*.

In a function module which our proposed method is incorporated in, we utilize the C++-implementation functions for the forward and the backward process like *CDcpp*. Furthermore, leveraging the regular connections in the fine-layered structure, we rewire the pointer of output $h_{out(j)}$ to that of input h_{in} at line 3 in Algorithm 1 to avoid copying the output to the input. Since this pointer rewiring (PR) technique is exploited in the forward and the backward process, the function module allows us to collectively calculate the values required in both the processes through all the fine layers at high speed. The effect on speed performance is revealed in Section VI-B.

Thus our function module has the versatility on the MZI representation and accelerates the forward and the backward process for learning a linear unit with the fine-layered architecture based on the basic units of the PSDC and the DCPS.

VI. Experiments

We experimentally demonstrate that our proposed method worked much faster than the conventional AD corresponding to the previous method in [12] without sacrificing accuracy. We show our settings including a neural network (NN), an executed task for evaluating performance of a PSDC-based NN, and parameter values for learning and a computer system where learning of the NN was executed, followed by the experimental results.

A. Settings

We employed an Elman-type recurrent neural network (RNN) model shown in Fig. 6. The model was implemented in PyTorch 1.7.0 with C++ extension and Python 3.8.5. With

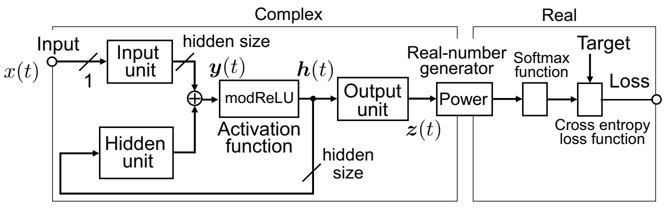


Fig. 6. Elman-type recurrent neural network for pixel-by-pixel MNIST task where the transformation matrix in the hidden unit is a product of unitary matrices and diagonal unitary matrix.

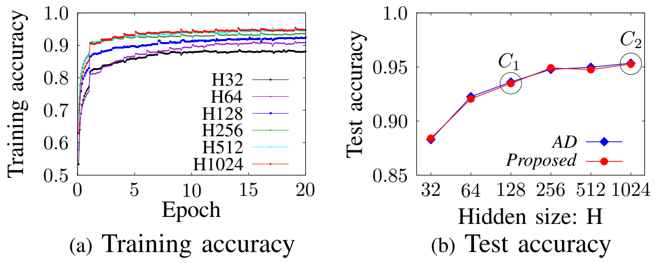


Fig. 7. (a) Training accuracy along epoch when our proposed method was applied to the RNN where the hidden unit had four fine layers and varied its size from 32 to 1024 (H32 to H1024). (b) Test accuracy of our proposed method and the conventional AD along hidden size in log-linear scale just after 20 epochs. The hidden size varied from 32 to 1024 under the condition of the number of fine layers was fixed at four. Circles C_1 and C_2 show the two test accuracies at the hidden size of 128 related to Fig. 8 and at the same setting as the result in [12], respectively.

the RNN model, we executed a classification task using the MNIST handwritten digit database [32]. The task was pixel-by-pixel MNIST task widely used for evaluating performance of RNNs that contain a hidden unit with a unitary matrix [12], [22], [23], [28], [33]. The MNIST database consists of $60,000$ training and $10,000$ test images. Each image is 256-level gray-scale 28×28 pixels representing a digit from 0 to 9. The pixels are flattened into a sequence of 784 pixels. Given the pixel sequence of a single image, the RNN sequentially receives one pixel and predicts a digit of the image just after the last pixel is processed.

The RNN in Fig. 6 was composed of two blocks: One was a complex-number processing block and the other was a real-number processing block. The complex-number processing block contained the input, the hidden, and the output unit and the activation function. The hidden unit consisted of a product of unitary matrices with the rectangular structure and a diagonal unitary matrix as shown in Fig. 5. The complex-number processing block processed an input-pixel value $x(t) \in \mathbb{C}$ at time t, which was generated by adding a zero imaginary part to a normalized real pixel value, as follows.

$$\boldsymbol{y}(t) = (W_{in} \cdot x(t) + \boldsymbol{b}_{in}) + W_h \cdot \boldsymbol{h}(t-1)\ , \tag{31}$$

$$\boldsymbol{h}(t) = \sigma(\boldsymbol{y}(t))\ , \tag{32}$$

$$\boldsymbol{z}(t) = W_{out} \cdot \boldsymbol{h}(t) + \boldsymbol{b}_{out}\ , \tag{33}$$

where $W_{in} \in \mathbb{C}^{H\times1}$, $W_h \in \mathbb{C}^{H\times H}$, and $W_{out} \in \mathbb{C}^{O\times H}$ denote the weight matrices of the input, the hidden, and the output unit. H and O denote the hidden size and the output size (the number of classes). $\boldsymbol{b}_{in} \in \mathbb{C}^H$ and $\boldsymbol{b}_{out} \in \mathbb{C}^O$ denote the biases of the input and the output unit. $\sigma(\boldsymbol{y}(t))$ means that the following activation function called a modReLU function was applied to each element of $\boldsymbol{y}(t) \in \mathbb{C}^H$, i.e., $y_j \in \mathbb{C}$, $j = 1, 2, \cdots, H$, where t is omitted for simplicity.

$$\sigma(y_j) = \begin{cases} \frac{y_j}{|y_j|}(|y_j| + b_j) & \text{if } |y_j| + b_j \geq 0 \\ 0 & \text{otherwise} \end{cases}\ , \tag{34}$$

where $b_j \in \mathbb{R}$ is an optimized bias parameter [28], [23]. We varied a hidden size and matrix representation capacity (the number of fine layers). The hidden size (H) and capacity (L) were varied from 32 to 1024 and from 4 to 20. The complex-valued signal $\boldsymbol{z}(t)$ passing the output unit was transformed to a real number in the real-number generator whose function was power function $P : \mathbb{C}^O \to \mathbb{R}^O$ expressed by $P(\boldsymbol{z}(t)) = \boldsymbol{z}(t) \odot \boldsymbol{z}(t)^*$, which is the Hadamard product of $\boldsymbol{z}(t)$ and $\boldsymbol{z}(t)^*$. The real-number processing block comprised the conventional units used for a classification task. As the loss function, a cross-entropy loss function was used.

We trained the RNN model with a mini-batch whose size was 100. Then we adopted data tensors with a feature-first structure, e.g., [hidden size, batch size] for the data tensor used in the hidden unit. The feature-first tensor structure was more efficient than a batch-first tensor structure when a small batch size was used for training the RNN model in a CPU-based computer system. For parameter optimization, we used the RMSProp optimizer with distinct learning rates (η) for the units: $\eta = 10^{-4}$ for the input unit, $\eta = 10^{-2}$ for the output unit, $\eta = 10^{-4}$ for the hidden unit, and $\eta = 10^{-5}$ for the activation function. The initial hidden state was fixed at zero and all the initial phase-shifter angles in the weight unitary matrix in the hidden unit were randomly sampled from $[-\pi, +\pi]$.

The pixel-by-pixel MNIST task on the foregoing model was executed on a computer system with Ubuntu 20.04 LTS, which was equipped with a single core-i7-10700K 3.8-GHz CPU with three-level caches and a 64-GB main memory, by multithreading with eight threads within the memory capacity.

B. *Results*

We confirmed that the RNN model was stably and successfully trained by our proposed method. Figure 7(a) shows the training accuracy along epoch when the RNN models, which had different hidden sizes from 32 to 1024 and the fixed number of fine layers of *four*, were trained for the pixel-by-pixel MNIST task by our method. Note that the *four* fine-layer structure corresponds to $(S_{A11}, S_{A12}, S_{B11}, S_{B12})$ in Fig. 5 equivalent to (S_{A1}, S_{B1}) in Fig. 2 and the four fine layers are fewer than those necessary for realizing any unitary matrix. Figure 7(b) shows the test accuracy just after 20 epochs along hidden size, which is displayed in log-linear scale. The test accuracies by our method and the conventional AD were almost the same values and increased with the hidden size in this range. Circle C_2 denotes the test accuracies at the same setting as that used in [12]. Then the test-accuracy curve by the AD (PyTorch) can be regarded as that by the AD (TensorFlow 1.x) in [12].

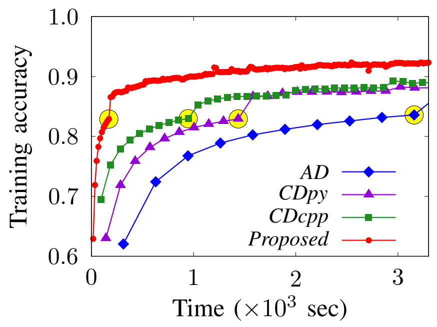


Fig. 8. Training accuracies by our proposed method (*Proposed*), *CDcpp*, *CDpy*, and the conventional AD (*AD*) along time. The hidden size and the number of fine layers were fixed at 128 (H128) and four (L4). The marks and the circle in each curve were put every 0.1 epoch and 1 epoch, respectively and work as indicators for measuring time. The curves reached at almost the identical test accuracy. In particular, *Proposed* and *AD* correspond to circle C_1 in Fig. 7(b) after 20 epochs.

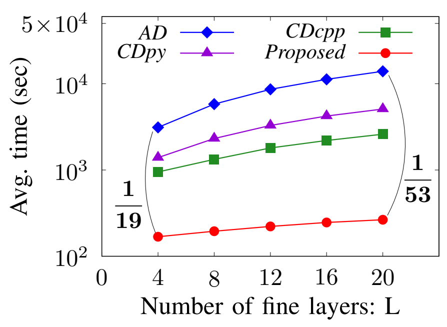


Fig. 9. Average elapsed time per epoch (sec) along the number of fine layers in linear-log scale when the pixel-by pixel MNIST task was performed in the RNN models. Our proposed method reduced the avg. elapsed times to $(1/19)$ and $(1/53)$ at the number of fine layers of four and 20, respectively.

We demonstrate that our proposed method achieved much faster learning than the conventional AD that can be regarded as the previous method in [12], keeping the same accuracy. Furthermore, we present the analysis results of our method in terms of contributions of the constituent techniques to the speed performance.

Our two main acceleration techniques are the customized derivatives (CD) in Section V-A and the collective calculation with the output-input pointer rewiring (PR) in the function module in Section V-B. To analyze the effects of the CD and the PR on the speed performance, we used four methods of the conventional AD (*AD*), *CDpy* with the PyTorch-implementation CD for the backward process, *CDcpp* with the CD implemented in C++, and our proposed method (*Proposed*) using both the CD and the PR implemented in C++ in Section V-B. Both the two method of *CDpy* and *CDcpp* did not leverage the pointer rewiring (PR) technique.

Figure 8 shows the training accuracies of the RNN model by the four methods of *AD*, *CDpy*, *CDcpp*, and *Proposed*, along time until $3,300$ sec. The RNN model had the hidden unit whose size and number of fine layers were fixed at 128 (H128) and four (L4). This setting corresponds to circle C_1 in Fig. 7(b). The four curves were critically different in time scale although they reached at almost the identical test accuracy after 20 epochs. At around $3,000$ sec, the accuracy by our method was over 0.92 while that by the AD was still 0.83. The marks in each curve are put every 0.1 epoch and the circles at one epoch work as indicators measuring the time per epoch. Thus our method significantly saved the time required for training the RNN models. This is useful to learn fine-layered neural networks using distinct models and many parameters under limited computing resources.

For the speed-performance analysis, we prepared the RNN models where each of their hidden units had the different number of fine layers from 4 to 20 and the fixed hidden size of 128. Figure 9 shows the average elapsed time per epoch (sec) of the four methods, *AD*, *CDpy*, *CDcpp*, and *Proposed*, along the number of fine layers in linear-log scale when the RNN models were trained for the pixel-by-pixel MNIST task in 20 epochs. When the number of fine layers was four, the test accuracies of *AD* and *Proposed* correspond to those at circle C_1 in Fig. 7(b) and the learning curves of the four methods are illustrated in Fig. 8. In terms of speed performance, our proposed method worked 19 and 53 times faster than *AD* at the number of fine layers of 4 and 20. *CDpy* and *CDcpp* performed about twice and 4 times the acceleration from *AD*. The remaining acceleration effect came from the pointer-rewiring (PR) technique implemented in C++, which collectively calculates the values used in the forward and the backward process. The function module is a simple yet very effective technique. Thus our proposed method accelerated the learning of the RNN model whose hidden unit was constructed with the fine-layered liner unit.

VII. Conclusion

We proposed an acceleration method for learning parameters of Mach-Zehnder interferometers (MZIs) in a fine-layered linear unit in an optical neural network (ONN). Our proposed method employs a function module in C++ to collectively calculate values of customized complex-valued functions and derivatives for a product of unitary matrices representing an MZI in the fine-layered linear unit. Consequently, our method reduces the time required for learning the parameters.

We confirmed that it worked almost 20 times faster than the conventional automatic differentiation (AD) when a pixel-by-pixel MNIST task was performed in a complex-valued recurrent neural network with an MZI-based hidden unit. Since our method is compatible with the current AD, we can easily use the method in a machine learning platform such as PyTorch.

The two directions remain as future work. One is to implement our proposed method in codes available in the state-of-the-art computer systems such as those equipped with multiple GPUs. The other is to compare ours with the other methods using linear units with unitary matrices as their weights like those in [23], [25] beyond the usage for ONNs and explore how to apply our individual techniques to them.

References

[1] N. C. Harris, J. Carolan, D. Bunandar, M. Prabhu, M. Hochberg, T. Baehr-Jones, M. L. Fanto, A. M. Smith, C. C. Tison, P. M. Alsing, and D. Englund, "Linear programmable nanophotonic processors," *Optica*, vol. 5, no. 12, pp. 1623–1631, 2018.

[2] X. Lin, Y. Rivenson, N. T. Yardimei, M. Veli, Y. Luo, M. Jarrahi, and A. Ozcan, "All-optical machine learning using diffractive deep neural networks," *Science*, vol. 361, pp. 1004–1008, 2018.

[3] T. F. de Lima, H.-T. Peng, A. N. Tait, M. A. Nahmias, H. B. Miller, B. J. Shastri, and P. R. Prucnal, "Machine learning with neuromorphic photonics," *J. Lightw. Technol.*, vol. 37, no. 5, pp. 1515–1534, 2019.

[4] W. Bogaerts, D. Pérez, J. Capmany, D. A. B. Miller, J. Poon, D. Englund, F. Morichetti, and A. Melloni, "Programmable photonic circuits," *Nature*, vol. 586, pp. 207–216, 8 October 2020.

[5] G. Wetzstein, A. Ozcan, S. Gigan, S. Fan, D. Englund, M. Soljačić, C. Denz, D. A. B. Miller, and D. Psaltis, "Inference in artificial intelligence with deep optics and photonics," *Nature*, vol. 588, pp. 39–47, 3 December 2020.

[6] X. Xu, M. Tan, B. Corcoran, J. Wu, A. Boes, T. G. Nguyen, S. T. Chu, B. E. Little, D. G. Hicks, R. Morandotti, A. Mitchell, and D. J. Moss, "11 TOPS photonic convolutional accelerator for optical neural networks," *Nature*, vol. 589, pp. 44–51, 7 January 2021.

[7] Y. Shen, N. C. Harris, S. Skirlo, M. Prabhu, T. Baehr-Jones, M. Hochberg, X. Sun, S. Zhao, H. Larochelle, D. Englund, and M. Soljačić, "Deep learning with coherent nanophotonic circuits," *Nat. Photonics*, vol. 11, pp. 441–446, 2017.

[8] R. Hamerly, L. Bernstein, A. Sludds, M. Soljačić, and D. Englund, "Large-scale optical neural networks based on photoelectric multiplication," *Phys. Rev. X*, vol. 9 021032, 2019.

[9] T. W. Hughes, I. A. D. Williamson, M. Minkov, and S. Fan, "Wave physics as an analog recurrent neural network," *Sci. Adv.*, vol. 5: eaay6946, 2019.

[10] M. Y.-S. Fang, S. Manipatruni, C. Wierzynski, A. Khosrowshahi, and M. R. DeWeese, "Design of optical neural networks with component imprecisions," *Optical Express*, vol. 27, no. 10, pp. 14 009–14 029, 2019.

[11] S. Pai, B. Bartlett, O. Solgaard, and D. A. B. Miller, "Matrix optimization on universal unitary photonic devices," *Phys. Rev. Applied*, vol. 11, iss. 6, 064044, 2019.

[12] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. Skirlo, Y. LeCun, M. Tegmark, and M. Soljačić, "Tunable efficient unitary neural networks (EUNN) and their application to RNNs," in *Proc. Int. Conf. Machine Learning (ICML)*, 2017.

[13] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018. [Online]. Available: http://jmlr.org/papers/v18/17-468.html

[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[19] D. A. B. Miller, "Perfect optics with imperfect components," *Optica*, vol. 2, no. 8, pp. 747–750, 2015.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[16] A. Hjørungnes, *Complex-valued matrix derivatives with applications in singnal processing and communications*. The Edinburgh building, Cambridge CB2 8RU, UK: Cambridge University Press, 2011.

[17] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, pp. 504–507, 2006.

[18] M. Reck and A. Zeilinger, "Experimental realization of any discrete unitary operator," *Phys. Rev. Lett.*, vol. 73, no. 1, pp. 58–61, 1994.

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

[21] N. Bansal, X. Chen, and Z. Wang, "Can we gain more from orthogonality regularlizations in training deep CNNs," in *Proc. Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. G. N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 4261–4271.

[22] S. Wisdom, T. Powers, J. R. Hershey, J. R. Roux, and L. Atlas, "Full-capacity unitary recurrent neural networks," in *Proc. Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 4880–4888.

[23] K. D. G. Maduranga, K. E. Helfrich, and Q. Ye, "Complex unitary recurrent neural networks using scaled Cayley transform," in *Proc. AAAI Conf. Artificial Intell.*, 2019, pp. 4528–4535.

[24] M. Mathieu and Y. LeCun, "Fast approximation of rotations and Hessians matrices," in *arXiv: 1404.7195v1*, 2014.

[25] Z. Mhammedi, A. Hellicar, A. Rahman, and J. Bailey, "Efficient orthogonal parametrisation of reccurent neural networks using Householder reflections," in *Proc. Int. Conf. Machine Learning (ICML)*, vol. 70, 2017, pp. 2401–2409.

[26] E. Vorontsov, C. Trabelsi, S. Kadoury, and C. Pal, "On orthogonality and learning recurrent networks with long term dependencies," in *Proc. Int. Conf. Machine Learning (ICML)*, vol. 70, 2017, pp. 3570–3578.

[27] M. Wolter and A. Yao, "Complex gated recurrent neural networks," in *Proc. Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. G. N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 10 536–10 546.

[28] M. Arjovsky, A. Shah, and Y. Bengio, "Unitary evolution recurrent neural networks," in *Proc. Int. Conf. Machine Learning (ICML)*, vol. 48, 2016, pp. 1120–1128.

[29] D. H. Brandwood, "A complex gradient operator and its application in adaptive array theory," *IEE Proc.*, vol. 130, no. 1, pp. 11–16, 1983.

[30] H. Leung and S. Haykin, "The complex backpropagation algorithm," *IEEE Trans. Signal Process.*, vol. 39, no. 9, pp. 2101–2104, 1991.

[31] S. O. Haykin, *Adaptive filter theory: 5th edition*. Pearson, 2013.

[32] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist

[33] Q. V. Le, N. Jaitly, and G. E. Hinton, "A simple way to initialize recurrent networks of rectified linear units," in *arXiv: 1504.00941v2*, 2015.