UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Communications

Photonic neural networks for machine learning training in classification tasks

Master's Thesis

Master of Science in Telecommunication Technologies, Systems and Networks

AUTHOR: Xie , Zhenyun

Tutor: Capmany Francoy, José

Cotutor: Pérez López, Daniel

External cotutor: CREGO GIL, ELADIO

ACADEMIC YEAR: 2021/2022

Objectives

The main objective of this project is to implement and simulate a feedforward neural network based on a programmable photonic integrated circuit simulation. To do so, the block that was conventionally operated on digital signals will be replaced by a linear analog computation using a photonic matrix architecture. Additionally, the project targets the development of a framework for the training of Photonic Neural Network (PNN) based on the Particle Swarm Optimization (PSO) algorithm. To enable a comparison with existing and future works, we apply the resulting framework to the "iris" dataset classification task.

Table 1: Objectives of the thesis project

Description	**GOAL**
Implementing Photonic Neural Network for Machine Learning	100% Completion
PNN performance Optimization	Accuracy ≥ 90% & 40 times less in time consumption compared to a PNN without optimization

Methodology

This work focuses on both the study, analysis, and development of theory around the training alternatives for photonic neural networks and its practical implementation and application. For the first step, the related scientific publications have been studied in detail, especially focusing on the below aspects to understand their fundamental concepts:

- Artificial Neural Network & Machine Learning
- Programmable Photonic Integrated Circuit
- Particle Swarm Optimization

Built on top of the theory covered, the second part is to implement the photonic neural network and optimize the PNN and PSO hyper-parameters settings to obtain the best performance of the PNN model for a specific classification task.

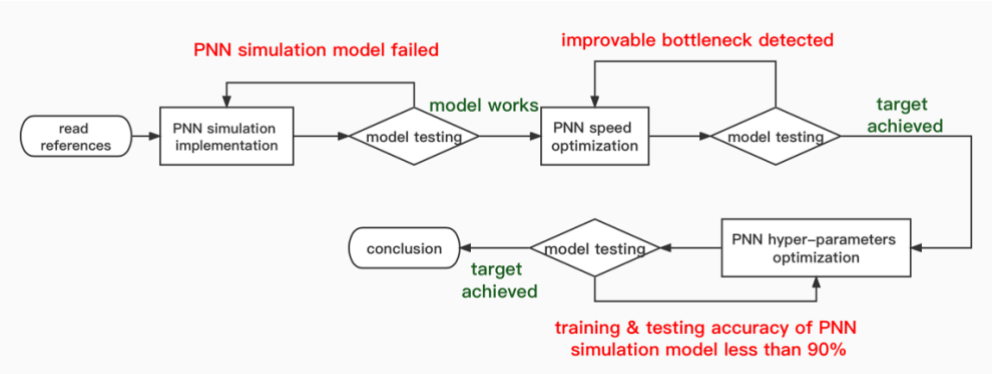


Fig.1. Diagram of process

Theoretical developments

The theoretical study has been analyzed in two aspects as mentioned before: on the one hand, reviewed the architecture and training methodologies of the artificial neural network, and machine learning. This also

includes advanced multivariable and free gradient techniques like the particle swarm optimization and its implementation.

On the other hand, the basic building blocks of the programmable photonic circuit have been studied, which are designed to implement feedforward linear optics unitary transformation between N input and N output ports to support mode transformations. This transformation is the fundamental principle of optical neural networks.

Development of prototypes and experimental work

Based on the theoretical basis, a photonic feedforward neural network simulation framework will be implemented and applied to machine learning tasks. In addition, we will optimize the performance of the PNN model both in processing speed and accuracy. To validate the performance of the PNN we will benchmark the results of a classification task experiment, the "iris" dataset, for PNN model training and testing

Results

The results include the theoretical study and the development of a framework for the control and simulation of photonic neural networks. The final results show a perfectly optimized PNN model, which is 40 times faster than a non-optimized version, and a training and testing accuracy higher than 90% for the classification task targeted for this project.

Future lines

Programmable Photonic Integrated Circuit is considered a promising emerging technology. At the same time, machine learning is also applied in various fields from healthcare to autonomous driving. This project demonstrates the application of photonic neural networks for machine learning tasks where low latency is required. Although the project is in the simulation stage now, the developed framework includes not only the simulation capability but also the control of future photonic hardware. In the near future, photonic-based neural network have the potential to bring to the market faster computing capabilities for low-latency demands. Additional future lines can be focused on solving existing technology limitations such as the optical nonlinear operations, as well as the high-performance of the Mach-Zehnder Interferometer phase adjuster which can reduce the energy consumption and improve the performance of the PNN.

Abstract

Photonic technology can provide sub-nanosecond delay, high bandwidth, and low energy consumption. Combining photonic technology with neural network models can break through the bottleneck of artificial intelligence technology development in the direction of high speed and low power consumption. This master's thesis aims to experimentally demonstrate a photonic neural network simulation performance through its study, development, training and optimization. The results showed a photonic neural network with a performance improvement of 40 times in computing time after the development and application of advanced optimization strategies. On the other hand, the corresponding relationship between the hyperparameters and the performance of the PNN model is studied and optimized, resulting in a stability improvement of the neural network. The steps followed for the modeling, implementation, and training of this PNN can be used as the workflow to follow in future PNN studies.

INDICE

I. INTRODUCTION

I.1. Background

In recent years, computer technology has been applied to many tasks that previously relied on the human senses, such as recognizing objects in images, transcribing speech, translating across languages, diagnosing diseases, driving cars, and so on. The core technology that directly contributed to these amazing developments is artificial intelligence (AI). The artificial neural network was proposed by McCulloch et al. in 1943 and was first realized in the form of mathematics and algorithms on the Von Neumann computer system based on silicon-based electronic chips, and simulated biological neural network to network information processing. Currently, artificial neural networks have made substantial progress, especially in its self-learning ability and parallel information processing abilities, which has attracted considerable attention in many application fields. However, any electronic chip with a von Neumann architecture would separate the program space from the data space, resulting in a large data loading between the memory and the computing unit. High frequent data reading and writing reduce the calculation rate and increase the latency and power consumption of a single calculation. Current researchers mainly employ two schemes, the enhanced integration level and in-memory computing, to improve computing efficiency, but the two schemes also face the following significant challenges. Firstly, it is unsustainable to continuously reduce the size of transistors to improve the computing performance due to the shrinking transistor will gradually bring significant quantum effects, which makes it difficult to further improve the computing efficiency of transistors. Next, the in-memory computing solution requires a large-scale modification of the current neural network architecture, which leads to the decreased portability and compatibility of this type of neural network algorithm.[1][2][19]

Photonics technology is a technology that uses photons as the basic carrier for information transmission and processing. Compared with traditional electronic technology, photonics has been widely used in communication, imaging, lidar, and signal processing due to its advantages of large bandwidth, low loss, and high amount of transmitted information. Programmable Photonics Integrated Circuits (PPICs) as one of the photonics technology branches has gradually developed into a solid and powerful technology with the incomparable advantages of true flexibility and reconfigurability [9][10][11]. Therefore, applying the traditional neural network model over the PPICs can take advantage of the latter two, and this superiority combination is expected to break the bottlenecks of the conventional electrical neural network, such as long delay and high-power consumption [7]. Recent research studies have demonstrated the possibility to use PPICs featuring a cascaded array of 56 programmable Mach-Zehnder interferometers in a silicon photonic integrated circuit and have shown its utility for basic vowel recognition [3][21].

I.2. Project Highlight

Considering the powerful function of PPICs, in the present project, we develop a complete framework to enable the simulation and future control of photonic neural network (PNN). The work covers the theoretical study, practical implementation and its application. In addition, we will demonstrate the training of photonic neural networks using a heuristic multivariable optimization method based on population, the particle swarm optimization (PSO) algorithm. This methodology requires the tuning of three hyperparameters and presents a high efficiency. This algorithm was applied in the past to conventional artificial neural networks.

Currently, the research on PNN is now in its initial stage, with most of the literature referring to theoretical developments and small experimental proof of concepts. One of the main reasons is the time required to create a complex programmable photonic chip (12-24 months) and the associated cost (0.5 to 1 M€). To gain a better understanding of the behavior and the properties of PNN and to know whether the theory can be applied in solving practical problems, the simulation process plays an important role in this stage. Therefore, the key significance of this thesis is to provide a simulation software platform for PNN modeling, training, and testing, through which the early exploration and research related to PNN will be conducted to ensure the feasibility of current and future development. The same framework can be employed for running the photonic hardware.

I.3. Project Outline

The thesis project work is presented within the framework of this Master's program. The project has been developed at iPronics Programmable Photonics, a spin-off company from the Photonics Research Labs (PRL) at iTEAM at the Universitat Politècnica de València. The project activities are focused on the concept proposal, theoretical analysis, and the demonstration of experimental simulation of photonic neural networks.

Chapter 2: Theory

This chapter covers the fundamental theory regarding the structure and working process of the Feedforward Neural Network (FNN), which presents one architecture framework of the artificial neural network that will be implemented in this work. We also introduce the Programmable Integrated Photonics Circuit as the core technology for this project. After the reviewing of the above essential preliminaries, we propose the core theoretical content of the Photonic Neural Network (PNN). Finally, Particle Swarm Optimization (PSO), which needs to be employed in PNN training, will be introduced.

Chapter 3: PNN simulation implementation

In this chapter, we include the implementation and development of the PNN. We start from the basic programmable unit cell (PUC) of the neural network, its transfer function to more complex

arrangements of PUC layers and complex meshes, which represents the photonic neural network. Subsequently, based on the sequential scheme, the analytical model of a photonic neural network will be created. The PNN model performance will be evaluated through an “iris” dataset classification experiment.

Chapter 4: PNN simulation model optimization

In this chapter, based on the profile result of chapter 3, the processes of the PNN model will be further optimized. The optimizations will be carried out from two main aspects, the model processing speed and the model hyper-parameters, and each optimization will be described according to the experiment result.

Chapter 5: Summary and further work

In this chapter, the conclusion and the summary will be made. The necessary further work of the PNN required to simulate and test the computation speed and power consumption will be addressed.

II. FOUNDATIONAL THEORY

The concepts of Artificial Neural Network (ANN), Particle Swarm Optimization (PSO), the main technology of Programmable Photonic Integrated Circuit (PPICs), as well as the structure of Photonic Neural Network (PNN) will be briefly introduced in this chapter.

II.1. Feedforward Neural Network

Artificial Neural Networks (ANN) is a research hot topic in the field of artificial intelligence, such networks are formed by the extensive interconnection of a large number of processing units (neurons). An ANN isn't the real construction of the human brain but an abstract and simplified simulation, the information processing in the network is realized by the interaction of neurons. An ANN training process is the dynamic evolution of the internal parameters that define the behavior of the network.

This section, firstly, will cover the basic unit (neurons) of the feedforward neural network, and its functionality. Secondly, it will show how arranging and connecting several neurons form a single-layer neural network, which is part of the foundation for more complex systems. Finally, how connecting multiple single-neural layers to form a feedforward neural network (also known as multilayer perceptron). A detailed introduction is shown below:

Neurons

In an ANN, neurons are also called processing elements or nodes. Generally speaking, a neuron structure should have the following elements:

- Input vector X (x_1, x_2, ..., x_2) and Output scalar z
- Weight vector W (w_1, w_2, ..., w_n) and bias b
- Neuron functionalities include the summation function "$\sum$" and the activation function "f_0" (note: activation in general are non-linear functions)

A basic processing unit structure model is shown in figure 2.:

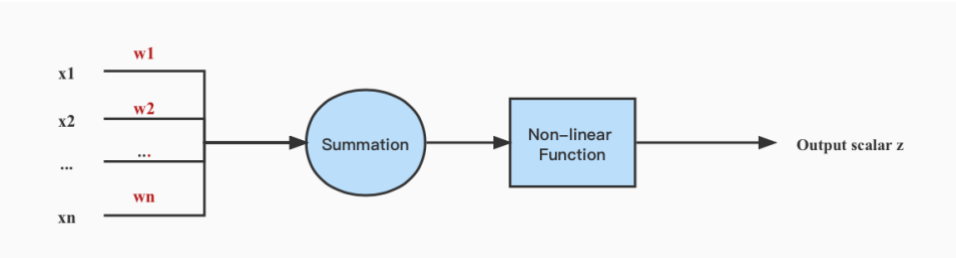


Fig.2. The basic structure of the processing elements in an artificial neural network

Hence, the output of a single neuron z is represented by the equation below:

$$z = f\left(\sum_{i,j}^{n} w_{ij} x_j + b\right) \quad (1)$$

Where f in this equation is the activation function, which is used to add nonlinear factors to solve problems that cannot be solved by linear models, such as some complex classification tasks. The chosen activation function needs to satisfy nonlinearity and differentiability. Rectified Linear Unit (ReLu) as an activation function is commonly used in neural networks. The corresponding function image is shown in figure 3 below:

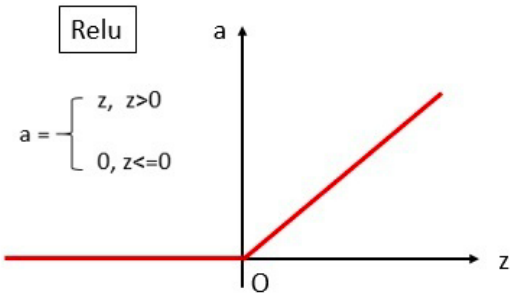


Fig.3. ReLu activation function

This function is a piecewise linear system, in which the value is always zero when z is lower than the cutoff (as the threshold to be set to control the value is not zero, usually the zero of cutoff as default). Therefore, the neurons in the neural network have sparse activation due to this one-sided inhibition.

Single-layer neural network (perceptron)

An example of a simple neural network composed of two layers of neurons, which can perform binary classification task, this model was proposed by Rosenblatt in 1958 [6]. The two layers of neurons are the input layer and output layer respectively. First, the input layer is only responsible for transmitting data into the network and not for computing. Next, the output layer needs to calculate the input of the previous layer. A simple scheme of a single-layer neural network is illustrated in the figure 4.

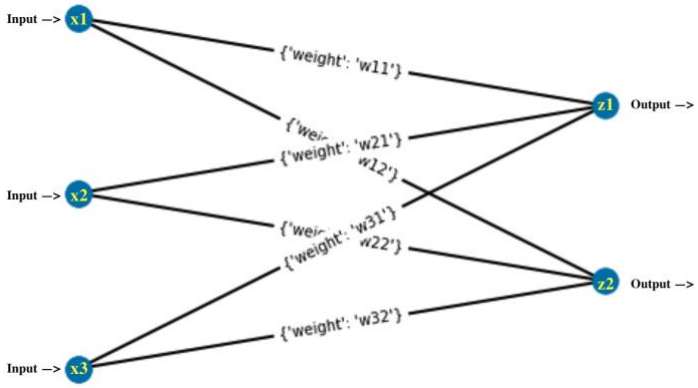


Fig.4. Single-layer neural network, where x1, x2, x3 are inputs, z1 and z2 are outputs

The nodes of x_1, x_2, and x_2 transmit data, and output z_1, and z_2 are represented as:

$$z1 = f(x_1 * w_{11} + x_2 * w_{21} + x_3 * w_{31}) \quad (2)$$

$$z2 = f(x_1 * w_{21} + x_2 * w_{22} + x_3 * w_{32}) \quad (3)$$

f in the Eq. (2) and Eq. (3) representing the activation function, the above two equations can be expressed as a propagation transfer matrix:

$$f\left[\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}\right] = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \quad (4)$$

This equation is the matrix that the neural network propagates from the previous layer to the next. The weights in the perceptron are obtained from the training, and the perceptron can be regarded as a logistic regression model, which can do simple classification tasks. [4]

Feedforward Neural Network Architecture

The Feedforward Neural Network (FNN) consists of an input layer, multiple hidden layers (also known as deep networks when many layers are present), and an output layer. Except for the output layer, both the input layer and multiple hidden layers can be connected to the next layer, and especially, this connection with two main characteristics of "summation" and "activation" is the critical piece of the FNN architecture. The framework of FNN architecture is shown in figure 5 below, in which there are three neurons in the input layer. Usually, the number of neurons in the input layer is equal to the length of the feature vector from where the system needs to learn form. One hidden layer implements the function of the summation and activation, and then transmits the instructions from the results to the next layer. At last, the output function with all information is aggregated together by the output layer.

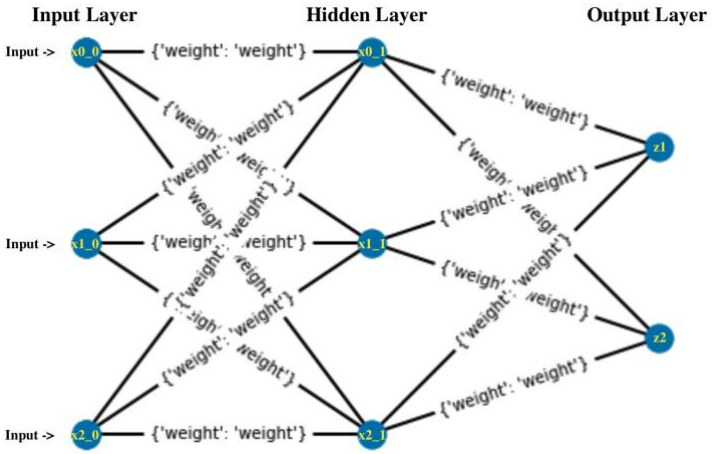


Fig.5. Feedforward neural network with only one hidden layer

based on the relationship between each layer in FNN described above, the transfer function for any layer in FNN can be represented as follows Eq. (5):

$$z^{(l)} = f_{l-1}(W^{(l)} * z^{(l-1)} + b^{(l)}) \quad (5)$$

where $z^{(l)}$ represents the output in present layer l , $W^{(l)}$ and $b^{(l)}$ are the weight and bias between layer l and $l-1$, f_{l-1} is the activation function in the previous layer $l-1$, $z^{(l-1)}$ is the output of the previous layer. Thus, the FNN can be simply regarded as a nonlinear transfer function of input x and output z, and can be controlled by adjusting the weight W.

Loss function

The loss function evaluates the error performance by FNN, which tries to estimate the data to be predicted. The better the performance of the neural network model has lower loss function. The Cross-Entropy Loss Function is a popular loss function used to quantify the difference between two probability distributions, which is given by:

$$H(p,q) = -\sum_{i=1}^{n} p(x_i) log q(x_i) \tag{6}$$

where $p(x)$ is the label sample probability distribution, and $q(x)$ is the estimated probability distribution.

II.2. Optimizer

The FNN has been briefly introduced in the subsection above, which demonstrated that the output of a neural network is determined by the parameters of weight W and bias b. Hence, the FNN training process can be considered as the process that involves the update of the parameters of the network that minimize the loss function.

Backpropagation

Backpropagation (BP) is the standard and most efficient technique to decrease the error from the loss function so as to correctly update the weights of the networks. Gradient Descent Optimization is the typical optimization algorithm applied in BP. In this case, the gradient is a vector that has magnitude and direction, which can indicate the direction that the function falls the fastest. At any given point, for an arbitrary function of $f(x)$, the vector $\frac{\partial f}{\partial x}$ is called gradient or slope of function $f(x)$, and the vector length acts as the indicator of the slope. Usually, if the value of the gradient vector is 0 or a near enough value, it can be associated to a local/global minimum/maximum. On the other hand, the vector direction represents the steepest ascent direction. Thus, the optimization process of the gradient descent algorithm aims to find the minimum value along the direction of gradient descent.

BP is used to calculate the gradient of the loss function for all weights in the network, and the value of the gradient is fed back to the optimization method, which is used to update the weights to minimize the loss function. The FNN transfer function of Eq. (5) is illustrated in Fig.6. in order to explain the backpropagation:

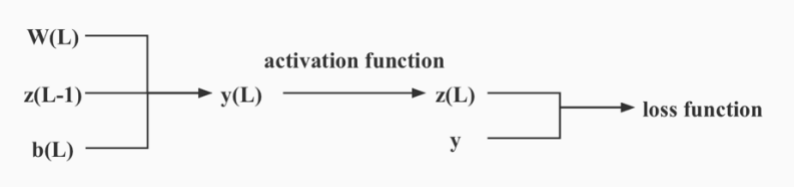


Fig.6. Scheme of transfer function

where $y(L)$ equivalents $W^{(l)} * z^{(l-1)} + b^{(l)}$, and y is the target value. For determining the tiny adjustments to the $W^{(l)}$ change in loss function (lf), which can be derived using the chain rule and gradient descent:

$$\triangle W^{(L)} = \frac{\partial loss_{function}}{\partial W^{(L)}} = \frac{\partial y^{(L)}}{\partial W^{(L)}} \frac{\partial z^{(L)}}{\partial y^{(L)}} \frac{\partial loss\_function}{\partial z^{(L)}} \quad (7)$$

Note: $\frac{\partial z^{(L)}}{\partial y^{(L)}}$ actually is doing the derivation of the activation function, which can also write σ' for sigmoid function, and $\frac{\partial y^{(L)}}{\partial W^{(L)}}$ comes out just to be $z(L-1)$.

The new $W^{(l)}$ will be updated based on the $\triangle W^{(L)}$:

$$W_{new}{}^{(L)} = W^{(L)} - \eta \cdot \triangle W^{(L)} \quad (8)$$

where η is the learning rate, and this process will iterate to update the weight and bias value until the gradient reaches 0, which means these values are minimum, and the FNN is configured with the optimum weights and biases.

There are two main drawbacks of backpropagation:

Firstly, since the loss function is highly nonlinear with weights and biases, it has many points (values) with zero gradients; therefore, the learning process often traps in the local minimum.

Secondly, backpropagation may probably face the issue of gradient vanishing and gradient exploding problems. As for the gradient vanishing, the backpropagation of the neural network is to multiply the partial derivatives of the loss function layer by layer Eq. (7); therefore, if ReLu is set as the activation function, the derivative result of this activation function is between 0 – 1. When the layer of a neural network is very deep, the backpropagation result becomes very small due to the multiplication of many numbers less than 1, which eventually become 0, as the result, the weights of the shallow layers will not be updated. On the other hand, for the gradient exploding, if $\sigma' \cdot z(L-1) > 1$, the backpropagation result will grow exponentially as the number of neural layers increases.

On top of the two previous constrains, when developing the optical version of a neural network we find an additional issue. In particular, it is difficult to calculate the gradient of an ONN as there's no clear analytical relationship between the optical phases and the weights of the matrix that allow us to compute the derivatives. Hence, population-based, gradient-free optimizers like the Particle

Swarm Optimization, can be useful to train an ONN. We extend this algorithm and its application to photonic neural networks in the next sections.

Particle Swarm Optimization

The Particle Swarm Optimization (PSO) algorithm is a heuristic algorithm, which can effectively search for problems with a huge parameter space and find candidate solutions without knowing too much information of the variable space. As the PSO algorithm has a large exploration area, it can effectively avoid falling into a local minimum value and can focus on the search near the minimum value to obtain higher accuracy. [17][24]

The basics of the algorithm are as follows: First, the objective to be optimized is a search in a D-dimensional space, in which each particle is regarded as an evaluation point. Next, the next location of each particle in the population is decided depending on its speed which is dynamically adjusted according to its own flight experience and the flight experience of other particles.

The i-th particle's position is expressed as X_{id}= $(x_{i1}, x_{i2}, \dots x_{iD})$, the best position that i-th particle has searched in the space is recorded as P_i = $(p_{i1}, p_{i2}, \dots p_{iD})$, which it's also known as "personal best" (*pbest)*. Additionally, the best position searched by all particles of the swarm is called "group best" (*gbest)* and it is expressed as P_g = $(p_{g1}, p_{g2}, \dots p_{gD})$. The velocity of i-th particle is represented as V_i = $(v_{i1}, v_{i2}, \dots v_{iD})$. For each iteration, the changing of particles' speed and its position are given by the following equations:

$$v_{id+1} = w * v_{id} + c1 * (p_{id} - x_{id}) + c2 * (p_{gd} - x_{id}) \quad (9)$$

$$x_{id+1} = x_{id} + v_{id+1} \quad (10)$$

where w represents the inertial weight, $c1$ and $c2$ represent acceleration constants.

The parameters of the above equations determine the optimized performance. Eq. (9) consists of three sections to describe the action of PSO. The first part of "$w * v_{id}$" represents the inertial of the particle's previous movement, and has the tendency to keep the current direction of exploration.

The second part, "$c1 * (p_{id} - x_{id})$" is the "cognition" section, representing the distance and direction between the particle's current position and historical optimal position, which is controlled by $c1$. For example, in any swarm's size, when $c1$ is much higher than $c2$, there is a low probability of achieving the global minimum due to no interaction between individual particles.

In the last part of "$c2 * (p_{gd} - x_{id})$" is the "social" section, which means information sharing and mutual cooperation among particles. If the coefficient of $c2$ is much higher than $c1$, it will converge faster under the interaction of particles. However, in this case, ($c2 \gg c1$) the swarm prioritize the exploitation of current optimal spaces rather than the exploration of better spaces, and easily being stuck in the local minima.

The algorithm flow of standard PSO is depicted as follows:

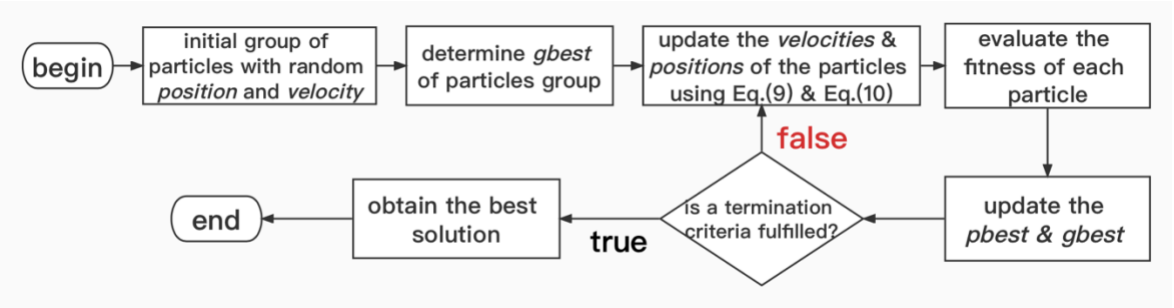


Fig.7. PSO algorithm flowchart

II.3. Programmable Photonics Integrated Circuits

Programmable Photonics Integrated Circuits (PPICs) is an emerging new paradigm that aims at designing common integrated optical hardware resource configurations, capable of implementing a wide variety of functionalities by suitable programming for the flow of optical signals. The concept of PPICs is similar to Electronic Integrated Circuits (IC), in which the IC integrates electronic devices such as transistors, capacitors, and resistors, while PPICs contain various optical or optoelectronic devices such as lasers, electro-optic modulators, photodetectors, etc. The PPICs are raising considerable interest as it is driven by the surge of a considerable number of new applications in the fields of telecommunications, quantum information processing, sensing, and neurophotonics, calling for flexible, reconfigurable, low-cost, compact, and low-power-consuming devices that can cooperate with integrated electronic devices. [9][10]

The programmable unit cell (PUC)

In PPICs, the flow of light is manipulated by waveguides connected in a mesh using 2 × 2 blocks, or “analog gates”, the on-chip equivalent of free-space optical beam splitters. The mesh connectivity determines the possible functions of the programmable circuit, and how it can be configured.

Below Fig.8. illustrates a PUC of waveguide mesh, where Mach-Zehnder interferometers (MZIs) are used to implement 2×2 coupler modules. A PUC can provide the power splitting k and phase delay $\Delta\varphi$ by placing two phase shifters $\phi 1, \phi 2$ at different arms of the MZIs (figure 8c, d). Note that the splitting ratio is maximum (=1) when k =0.5, meaning the MZI-based PUC splits power equally in both the outputs. The splitting ratio is minimum (=0) when either k = 0, or k =1. [11]

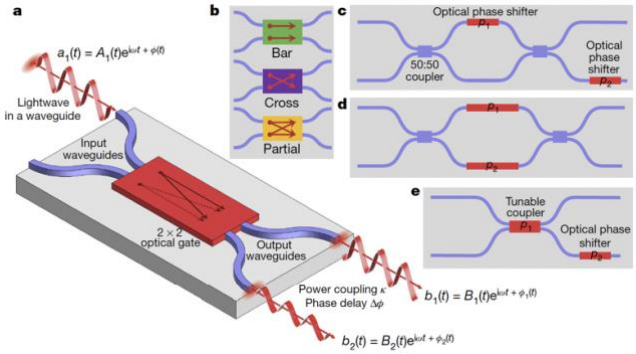


Fig.8. **Universal 2 × 2 optical gates**. **a**, A 2 × 2 programable unit cell
b, The possibility state of PUC **c–e,** The PUC implementation method [11]

A single block unit performance is defined by the following simplified matrix Eq. (11). This is the transfer function of a balanced MZI loaded with phase actuators on both arms:

$$S_{PUC} = -je^{-j\Delta}\begin{bmatrix} sin\theta & cos\theta \\ cos\theta & -sin\theta \end{bmatrix} \cdot \gamma \tag{11}$$

Where $\theta = \frac{\phi_{upper}-\phi_{lower}}{2}$, and $\Delta = \frac{\phi_{upper}+\phi_{lower}}{2}$

Here, the coupling factor $k = cos^2(\theta)$ and overall phase shift Δ can be independently tuned using the two control phases of MZI, ϕupper and ϕlower, respectively. Regarding the $-j$ in the equation, which is sourced from the array of 3-dB directional couplers. In addition, the prior matrix has a general loss term γ that includes the effect of propagation losses in the access waveguides, the tunable coupler waveguide and the insertion losses for both 3-dB couplers. [10]

Waveguide mesh

The optical waveguide networks are mainly divided into two categories: one is the forward-only network, in which the light can only be transmitted from the input to the output, and the other one is the recirculating network, where the light can be cyclically transmitted in the network. [23] In the thesis, the feedforward architecture is considered.

In feedforward mesh networks, light flows in one direction, interfering in 2 × 2 MZIs at every stage, this architecture allows a simple progressive setup, and can be reconfigured for some simple cases and problems.

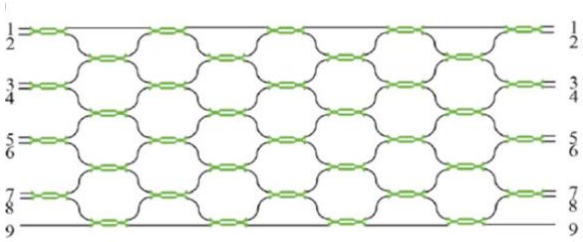


Fig.9. A rectangular arrangement of a 9 × 9 MZIs [9]

The above figure 9. shows a schematic diagram of the mesh in 9 × 9, this network architecture requires 36 MZIs to implement any linear transformation matrix between input and output.

Transfer function decomposition

William Clements and co-workers [13] proposed a method for decomposing unitary matrices into a sequence of beam splitters and phase shifters, the decomposition equation is shown as follows:

$$U = D\left(\prod_{(m,n)\in S} T_{mn}(\theta,\phi)\right) \tag{12}$$

Where T_{mn} represents the transformation matrix of the PUC the specific value (θ,ϕ) of phase shifters, D is a diagonal matrix and U is any given unitary matrix. In this way, any unitary matrix can be implemented by multiplying cascaded MZI transfer matrices. The mentioned matrix multiplications are at the heart of photonic neural network, which will be discussed in the next section.

II.4. Photonic Neural Network

A photonic neural network (PNN) is a physical implementation of an artificial neural network with optical components, and below figure 10. shows the structure of 3 layers of a photonic neural network with 3 inputs and outputs, this figure10 clearly demonstrates that each optical component represents a different part of the neural network respectively.

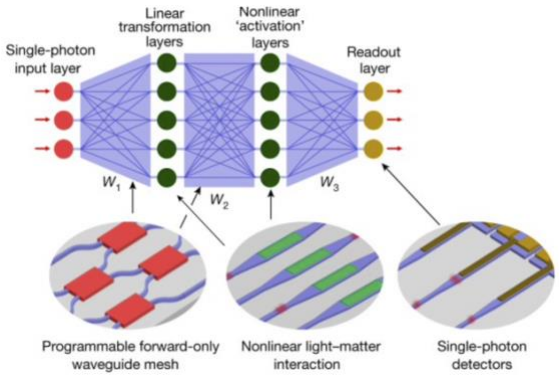


Fig.10. A 3 layers of photonic neural network model [11]

Regarding to the PNN, the full connection layer can be realized with n_m neurons. Each layer (which corresponds to a waveguide mesh) performs the linear matrix-vector Multiplication and ACcumulation (MAC) operations. The output is the same as the Eq. (5) that was mentioned in Chapter 2, where the weight matrix W_m represents the phases array of the waveguide mesh. Given weight matrix W_m that can be obtained by training the network and mapped to MZIs using singular value decomposition (SVD), each weight matrix W_m can be written as the following equation:

$$W_m = U_m^{n_m \times n_m} \Sigma_m^{n_m \times n_m} V_m^{H,n_m \times n_m} \tag{13}$$

The $U_m^{n_m \times n_m}$ and $V_m^{H,n_m \times n_m}$ are a unitary matrix with dimension $n_m \times n_m$, the $\Sigma_m^{n_m \times n_m}$ is a diagonal matrix with dimension $n_m \times n_m$ as well.

A unitary matrix $U_m^{n_m \times n_m}$ and $V_m^{H,n_m \times n_m}$ can be realized by using a cascaded array of 2×2 MZIs with a specific phase setting in phase shifters value θ and ϕ per MZI, this transform method is based on the Clements rectangular decomposition scheme, which was shown in the Eq. (12). Moreover, the number of total MZIs or PUCs that are needed to implement a unitary matrix of size N, which can be given by the next equation:

$$\frac{\mathrm{N(N-1)}}{2} \tag{14}$$

where the N is the size of the designated unitary matrix. Taking Fig.9. as an example, the mesh has an input size of 9 which contains the total $\frac{9(9-1)}{2} = 36$ PUCs.

On the other hand, the diagonal matrix $\Sigma_m^{n_m \times n_m}$ can be realized with an array of MZIs with one input and one output, and the activation function can be implemented by using common optical nonlinearities such as saturable absorption, bistability, and as well as the digital off-chip system, which transform the optical domain to digital domain and operate the nonlinearities digitally then transform back to optical domain.

So far, the fundamentals of MZIs- based PNN design have been summarized in this section. The next chapter introduces the implementation of the framework to simulate, control and train PNN.

III. PHOTONIC NEURAL NETWORK IMPLEMENTATION

This chapter is based on the theoretical content of the previous chapter for the development and implementation of the PNN simulation. We employ Python for the implementation. First, the process will start from a single basic programable unit cell (PUC), and arrange them in a layer following the waveguide mesh index pattern. PySwarms [22] as the PSO research toolkit package will be used to train the PNN simulation, although we have also developed a custom PSO for its future upgrade. To validate the PNN implementation, we also conduct a simple test to profile the performance of the PNN.

III.1 Tools for Implementation

The language for this implementation is Python, which is a high-level programming language for writing applications. Python provides a very complete basic code base that can be used directly. Therefore, Python version 3.8 is used and the below packages are also required for the implementation:

- NumPy with version 1.21.5: NumPy is a basic package for scientific computing in Python. It provides multidimensional array objects and a hybrid program for fast array operations, as the programmable feedforward waveguide mesh operates many matrices linear computation, NumPy can be used to store and process large matrices.
- Numba with version 0.55.1: Since Python is an interpreted language, the code needs to be translated into CPU machine code during execution. This is a time-consuming process, especially for math-heavy tasks like machine learning, and it will be very slow. Numba is an open-source JIT compiler for Python, which enables CPU and GPU acceleration of native Python code.
- Matplotlib with version 3.5.0: Matplotlib is one of the most popular Python packages for data visualization. It will be used to plot the experiment result.
- PySwarms with version 1.3.0: PySwarms is an extensible research toolkit for particle swarm optimization (PSO) in Python. It has a high-level declarative interface that is able to implement PSO as a PNN's optimizer.
- Python Profiler with version 3.10.4: profiler is a python program that describes program performance at runtime and provides statistics from different aspects to express. It will be used to profile the photonic neural network performance.

The implementation is done on PyCharm, which is a python Integrated Development Environment (IDE), and it is the tool that can help make it more efficient while developing.

III.2 Photonic Neural Network Implementation

The PNN implementation starts with the modeling and coding of the basic unit, the programmable unit or PUC. Then a PUC layer can be constructed by arranging the implemented PUC. Finally, connecting the implemented PUC layers can create a more complex photonic mesh. The flowchart of the implementation process will be the following:

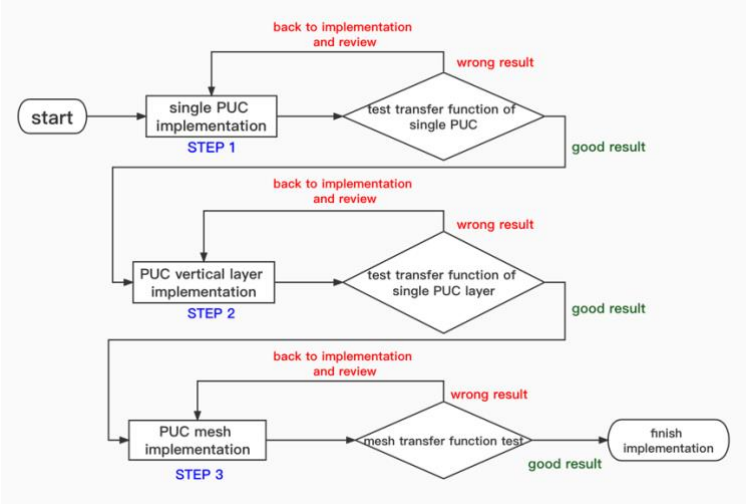


Fig.11. Photonic mesh implementation flowchart

Single PUC implementation → Step 1

The PUC, already theoretically presented in the previous chapter, is the basic programmable unit block in the optical mesh. It is the first unit in the implementation and the core element in the optical neural network framework. During the work, we selected and implemented the attributes that define the PUC. They are shown in the following table:

Table 2. PUC attributes for implementation

PROGRAMMABLE UNIT CELL (PUC)	
Attributes	**Description**
self.name	Indicates each PUC in the mesh (type integer). e.g. “0, 1, 2...”
self.pos	Indicates the location of PUC in the mesh
self.technologies	Includes the integrated technology parameters such as “*group index*”, “*passive phase offsets*” etc.,
self.ports	The naming of the PUC ports, which are identified by waveguides mesh index.
self.driven_phases	An array with the driving phases applied to each phase actuator. The two phase shifters of PUC

The attribute "*self.ports"* represents the two arms of in PUC, each arm of PUC corresponding to the waveguide mesh index. In addition, the attribute "*self.driven_phases*" is the phase state of two arms in the PUC (from 0 to 2π). These phases determine how the optical signal is transmitted through the PUC according to the Eq. (11). The driven phases can be handled by the following methods:

Table 3. The methods of PUC-driven phases of PNN simulation

METHODS	**DESCRIPTION**
set_driven_phases(driven_phases: np.ndarray)	Setting the driven phases (offset) of the PUC.
get_driven_phases()	Getting the driven phases (offset) of the PUC.

The main method of the feedforward PUC is to calculate its transfer function or spectral response given an array of wavelengths. We called the "*compute_response*", and the implementation is based on the Eq. (11). Below is the pseudocode of the compute response:

Procedure compute_response ()

Input: None

Return: The transformation matrix of PUC

1. **get** driven_phase_upper_arm % *From the PUC attributes*
2. **get** driven_phase_lower_arm % *From the PUC attributes*
3. **get** passive_phase % *From the PUC attributes*
4. **compute** theta % *based on the Eq. (11)*
5. **compute** delta % *based on the Eq. (11)*
6. **compute** transformation_matrix_position_11 % *based on Eq. (11) & theta, delta*
7. **compute** transformation_matrix_position_12 % *based on Eq. (11) & theta, delta*
8. **set** transformation_matrix_position_21 **as** transformation_matrix_position_12
9. **set** transformation_matrix_position_11 **as negative** (transformation_matrix_position_22)
   create transformation_matrix in **array** with value
   [[transformation_matrix_position_11, transformation_matrix_position_12],
   [transformation_matrix_position_21, transformation_matrix_position_22]]
11. **return** transformation_matrix

Once we programmed and validated the PUC cell, we proceeded with the implementation of an arrangement of PUCs or PUC layer.

Vertical PUC layer implementation → Step 2

Since each PUC contains its own ports number *"m"* and *"n"*, the approach of this step is to use ports number and arrange PUCs one by one in the vertical direction in order to construct a PUC layer, the below illustration Fig.12. demonstrates how a PUC layer looks like:

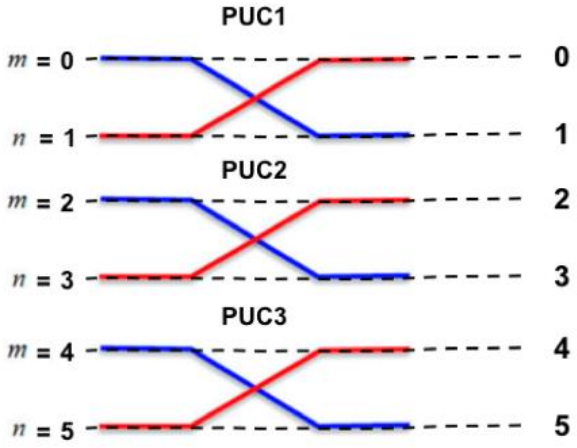


Fig.12. A PUC layer with 3 PUCs

This PUC layer contains 3 PUCs, the ports of each PUC correspond to the number of waveguides mesh index, "0 – 5" in this case. For example, the 2 arms of PUC1 are located in the waveguide index "0" and "1", thus the port *"m"* and *"n"* are 0 and 1 respectively.

Regarding the PUC layer, it should contain the following attributes:

Table 4. The PUC layer attributes for the implementation

PUC LAYER

Attributes	**Description**
self.size	Number of waveguides in the system where the PUC is imbedded
self.pucs	The PUC list of the PUC layer
self.geometric_properties	The geometric properties of PUCs in the layer (lengths, pitch, etc.). e.g. geometric properties = {"lx": 500e-6, "ly": 0}

The attribute *"self.size"* defines the PUC layer's size, for the above PUC layer instance illustrated in Fig.12, in this particular example, it represents a 6 inputs layer. On the other hand, the core contents of a PUC layer class is the list of *"self.pucs"*, which is used to save the PUCs in the corresponding PUC layer, and *"self.pucs"* can be realized by adding PUC objects sequentially. The following pseudocode exemplifies the process:

```
Procedure create PUC layer
Input: waveguide_index
Return: the list of PUCs
1.      create puc   % an empty list to save PUCs
2.      for index in waveguide_index
3.          initial a new PUC with ports name equal index
4.          puc add new PUC
5.      return puc
```

A single PUC layer can be constructed based on the above algorithm. The performance of the PUC layer is defined by its transfer function, which can be obtained by the compute response of each PUC. The following pseudocode shows the transfer matrix of PUC layer:

```
Procedure compute_transfer_function
Input: None
Return: the transfer matrix of PUC layer
1.      create T % the complex identity matrix with size of mesh
2.      for puc in puc_layer
3.          compute response of the puc
4.          accumulate response of each puc into T
5.      return T
```

Where *"T"* initially is a complex identity matrix with the size of mesh, it is created to accumulate response of each PUC in one. The *"T"* will be returned as the final transfer function result of PUC layer.

PUC waveguide mesh class implementation → Step 3

The mesh consists of several PUC layers connected through waveguides in an organized way. The size of the photonic waveguide mesh is defined by the input numbers. The smallest waveguide mesh that can be created capable of performing a unitary transformation (beyond the 2x2 basic block) is the mesh with an input equal to 3. In this case, the total number of the PUCs is 3 according to the Eq. (14). The below figure illustrates the aforementioned architecture, where each PUC layer only contains a single PUC, and they are arranged based on the ports number:

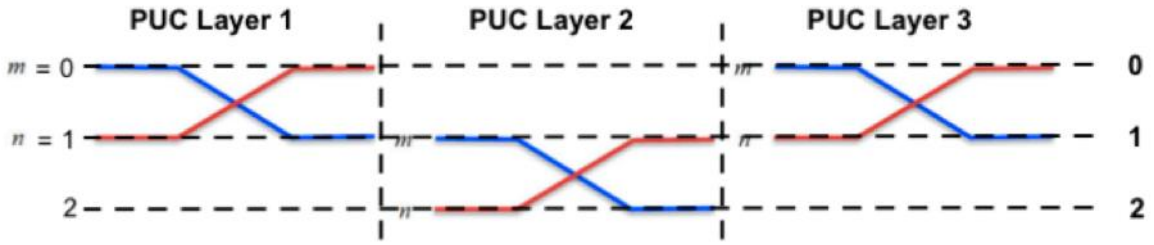


Fig.13. The smallest size of a waveguide mesh

The table below shows the basic attributes that required to create a photonic feedforward mesh.

Table 5. The waveguide mesh attributes for the implementation

FEEDFORWARD WAVEGUIDE MESH

Attributes	Description
self.size	Identification of the size of the mesh
self.layers	The list with all PUC layers

self.technologies	Includes the technologies parameters of silicon photonics such as "*group index*", "*passive phases*" etc.,
self.num_pucs	Indicates the number of PUCs in the mesh

One of the core attributes of the mesh arrangement is *"self.layers"*, which is the core of PNN and it is used to perform the multiply and accumulate (MAC) operations. It operates by adding the response of every PUC layer sequentially. The method "*create_core_puc_layers*()" creates the core that it will perform the required computations. The implementation pseudocode is following:

```
Procedure create_core_puc_layer
Input: mesh_size
Return: the list of PUC layer (core)
1.      create an empty list name layers
2.      set waveguide_index equal to mesh_size
3.      for layer_index in range of mesh_size
4.          if mesh_size is even
5.              if layer_index is even
6.                  create a new PUC layer with waveguide_index 0 to the last index
7.                  the list layers add new PUC layer
8.              else layer_index is odd
9.                  create a new PUC layer with waveguide_index 1 to second last index
10.                 the list layers add new PUC layer
11.         if mesh_size is odd
12.             if layer_index is even
13.                 create a new PUC layer with waveguide_index 0 to second last index
14.                 the list layers add new PUC layer
15.             else layer_index is odd
16.                 create a new PUC layer with waveguide_index 1 to last index
17.                 the list layers add new PUC layer
18.     return list of layers
```

Once the core of the waveguide mesh is created in the aforementioned method, the transfer function of the mesh can be easily realized. This is done efficiently by performing the dot product of the transfer function of each PUC layer in the mesh with a diagonal matrix following the Clements decomposition Eq. (12), the following pseudocode demonstrates the procedure:

```
Procedure compute_mesh_transfer_function
Input: mesh_layers, mesh_size
Return: the compute transfer function of the mesh
1.      create a complex_diagonal_matrix with mesh_size
2.      create total_transfer_function and set to 1
3.      for layer in the reverse of mesh_layers
4.          compute transfer function of the layer
5.          set total_transfer_function equal total_transfer_function dot transfer function of layer
6.      set total_transfer_function equal total_transfer_function dot complex_diagonal_matrix   % Eq. (12)
7.      return total_transfer_function
```

PUC mesh test → Step 4

To validate the performance of the framework to simulate and control photonic neural networks we run a test script (See Fig.14.). The test target is to check if the PUCs in the mesh is actually replicating the expected behavior defined by our InM input matrix. The phases values of the mesh have been calculated using the Clements before calculating the global response. The implemented compute transfer function allows computing the transfer function of the feedforward mesh for a set of wavelengths. The resulting matrix is ideally equal to the loaded one when the insertion losses of each PUC are not considered and other component ideal behavior is assumed:

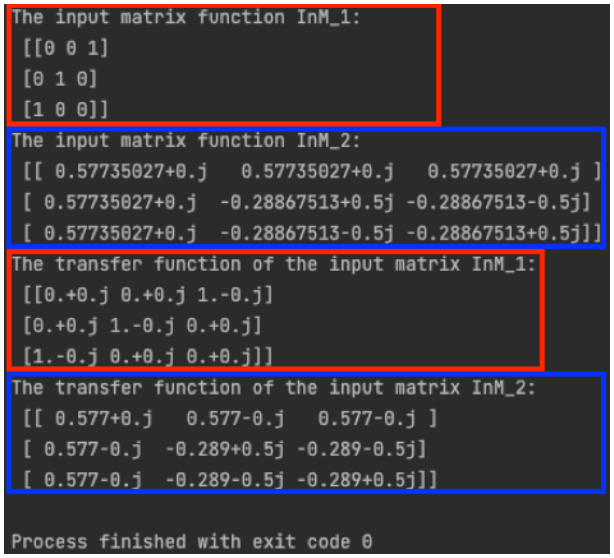


Fig.14. Implemented photonic mesh transfer function test result: where the red box shows the results for matrix function InM_1, and the blue box for the matrix function InM_2

So far, the process of implementing of the entire photonic mesh has been shown, emphasizing that the target transfer function can be achieved by gradually adjusting the phase of the MZIs

III.3 Activation Function & Optimizer

A fundamental block of neural networks is the nonlinear function that is typically located between matrix multiplication elements. Being an element-wise function of a vector, the activation functions can increase the nonlinear fitting ability of neural networks or in other works their capability to adapt and learn more complex patterns. Although optical and electro-optical non-linearities can be implemented, practical constrains and challenges are behind their typical implementation in the digital domain. Therefore, the forward pass of ReLu is implemented below:

Procedure relu_forward_pass
Input: input_fields
Return: transformed inputs

```
if abs(input_fields) >= cutoff:
    return input_fields
else:
```

4. **return** input_fields * alpha

where the “alpha” is the attenuation factor, the typical value is 0.

Once we have all the elements required for the execution of a neural network, we need to train it. As covered in the previous chapters, we have selected a heuristic multivariable optimizer for this task. In this case, we have integrated the particle swarm optimization (PSO) to our system employing the Pyswarm tool as the third part package. However, to deeply understand the algorithm we also implemented a version from scratch. To enable the integration of the algorithm to be used in the training of neural networks, we have created an interface class. The method “*fit*” will fit the model and labeled data in order to perform the minimization process. In this case, the *“objective function”* is the analytical function to be evaluated and minimized. To enable the minimization, we have implemented the *“cost function”* method with the following arguments:

Table 6. The cost function arguments in the implementation of the neural network training

COST_FUNCTION_ARGS

ARGUMENTS	**Description**
data	Training data
labels	Training features
model	PNN model to optimize
cost	The cost function to be used

With the previous cost function, we can then implement the optimizer that will run the training or learning process of the photonic neural network. In other words, the optimizer will find the configuration of the phase actuators of the mesh in order to minimize the network error and learn the specific function. To do so, the interfacing function that we have implemented integrate the network variables into Pyswarm's development kit, and follow the PSO algorithm explained in flowchart Fig. 5. The following pseudocode demonstrates the procedure:

Procedure PSO

1. **Initialize population**
2. **for** t in maximum generation:
3. **if** $f\left(x_{i,d}(t)\right) < f\left(p_i(t)\right)$ then $p_i(t) = x_{i,d}(t)$
4. $f\left(p_g(t)\right) = \min\left(f\left(p_i(t)\right)\right)$
5. **end**
6. **for** d in dimension:
7. $v_{id+1} = w * v_{id} + c1 * (p_{id} - x_{id}) + c2 * (p_{gd} - x_{id})$ *# Eq. (9)*
8. $x_{id+1} = x_{id} + v_{id+1}$ *# Eq. (10)*
9. **if** $v_{id+1} > v_{max}$ then $v_{id+1} = v_{max}$
10. **else if** $v_{id+1} < v_{min}$ then $v_{id+1} = v_{min}$

end
if $x_{id+1} > v_{max}$ then $x_{id+1} = x_{max}$
else if $x_{id+1} < x_{min}$ then $x_{id+1} = x_{min}$
end
end
end

III.4 Photonic Neural Network Test

Until this section, the photonic mesh structure with the matrix transfer method has been implemented and tested, together with the activation function and PSO optimizer. With these implementations, the simulation framework can easily create a Keras-like neural network model and train it using a proprietary photonic waveguide mesh core. Each one of these photonic Multiply and Accumulate (MAC) cores and activation layers are equivalent to a traditional neuron layer as observed in above figure 6. In the case at hand, the number N of neurons in each layer will be given by the order N of the photonic MAC core.

Therefore, in this section, we validate the PNN framework by creating and simulating the training of a photonic network applied to a typical classification problem, the iris dataset. During the process, the program records the test accuracy is recorded and the training performance will also be evaluated by using Python Profiler. After importing the related packages and iris dataset, it is necessary to define and create a PNN model (network architecture). The following snippet defines a model with 4 inputs, 4 hidden layers, and N = 4 neurons in each layer:

```
model = models.Sequential(
    [
        layers.OpticalMesh(N),
        layers.Activation(nonlinearities.ReLU(N, cutoff=0.3)),
        layers.OpticalMesh(N),
        layers.Activation(nonlinearities.ReLU(N, cutoff=0.3)),
        layers.OpticalMesh(N),
        layers.Activation(nonlinearities.ReLU(N, cutoff=0.3)),
        layers.OpticalMesh(N),
        layers.Activation(nonlinearities.Square(N)),
        layers.DropMask(N, drop_ports=[3]),
    ]
)
```

In this case, the PSO should find the optimal position in a 64 dimensions space, which corresponds to the number of tunable parameters in the model based on the formula below:

$$\left(\frac{\mathrm{N(N-1)}}{2} * 2 + num\_phase\_diag\right) * num\_layer \tag{15}$$

Hence, in this example, we employ 200 particles for the optimization, and 200 maximum interactions. To do so, we employ the following dictionary. The rest of the numbers employs typical values:

```
pso_settings = {
    "n_particles": 200,
```

```
    "max_iters": max_iters,
    "dimensions": num_phase_shifters,
    "start_opts": {"c1": 2.5, "c2": 1.5, "w": 0.4},
    "bounds": bounds,
    "bh_strategy": "intermediate",
}
```

Finally, the experiment test starts with the above settings, and is repeated 10 times to get meaningful statistical results. Note that the 200 particles are initialized independently and with random locations within the search space. Once we have the 10 experiments, we get the average results. Each experiment is run under the 2,2 GHz Dual-Core Intel Core i7 process with 8 GB 1600 MHz DDR3 RAM. The below screenshot shows the test output of the experiment.

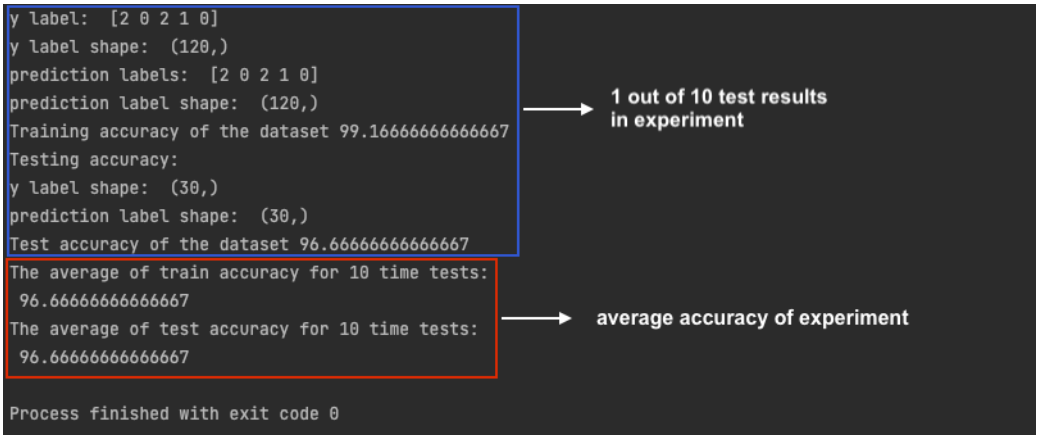


Fig.15. The model experiment result of the iris dataset for 10-time experiments:
where the blue box is test results of one experiment, and red box is the average accuracy of all the experiments

The previous test confirms the correct implementation of the photonic neural network framework and is ability to train the models that will run in the photonic hardware. The trained models can successfully perform classification tasks on the iris dataset with an accuracy comparable to electronic hardware. In addition, to evaluate and improve the performance of the photonic hardware simulator, we employ a python Profiler. The below textbox shows the profile result of PNN model training with total 1317.245 seconds.

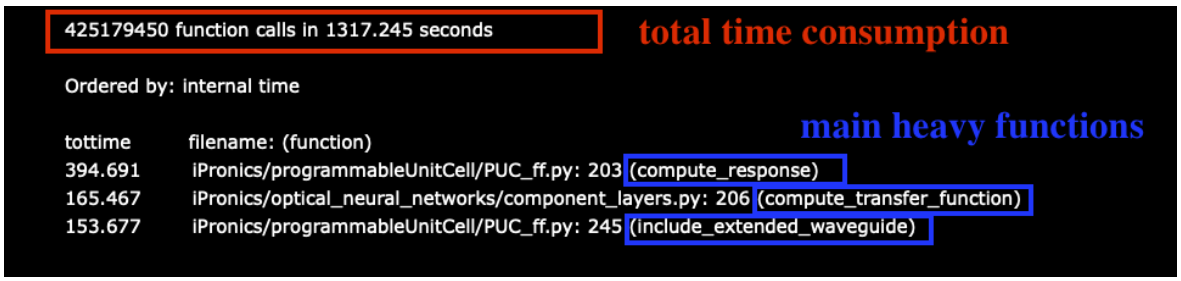


Fig.16. The profile result of the PNN model training
total time consumption is 1317.245 seconds

Results, analysis and conclusions:

The PNN model and PSO optimizer with the current arbitrary setting parameters obtain a good accuracy, and validate the ability of the framework to simulate and operate with photonic neural networks. However, it is necessary to study the dependencies of the model hyperparameters (setting variables of the optimizer), and obtain their optimal configuration. To do so, we will perform an optimization process to find out the best hyperparameters setting. In particular, we will optimize the "cutoff" in the activation function of the neural network, as well as the "n_particles", "c1", "c2", and "w" of PSO optimizer.

In addition, before starting the hyperparameter search, we will solve the following challenges to increase the efficiency of the training and simulation process:

Firstly, we will analyze the current program workflow and time consumption employing the profiler to optimize the current execution time and enable a more efficient optimization process. In particular, the longest time consumption relates to the *compute response* method in the programmable unit cell, which can be optimized by using Numba package.

Secondly, the "*n_particles*" can also affect the computation speed, the more particle number will take more computations and time, therefore the minimum number of particles without affecting the PNN training stability should be used.

Thirdly, the early stopping feature can be added if the best cost yet found does not improve by more than the tolerance, the optimizer will stop without consuming additional resources.

Finally, multiprocessing can also be implemented in the optimizer to speed up the processing, and the optimization test will run on a computing server, which has 250G RAM with 80 cores.

The next chapter of this thesis will demonstrate each optimization process step by step.

IV. PHOTONIC NEURAL NETWORK OPTIMIZATION

This chapter contains the photonic neural network optimization process, based on the analysis performed in the previous chapter. In particular, the following items that will be optimized are:

- Speed optimization:
    - compute transfer function of each PUC
    - the minimum population size (particles) of the PSO algorithm.
    - early stopping features for PSO
    - multiprocessing features for PSO
- Hyperparameters optimization:
    - cutoff setting in ReLu activation function
    - hyperparameters of PSO: *"c1", "c2"*, "*w*"

Since the simulation experiment contains a large number of computations, we employ a server-based computer with the below configuration:

RAM: 255GB

CPU: 80 cores

IV.1. Numba accelerator

The profile result shows that the compute transfer function is the most time-consuming. To solve this issue, the Numba compiler can be applied to speed up the method by translating Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library.

We just need to add a decorator "*@jit"* to the original code when we use Numba, and it will try to compile the code. If some functions or data types are not supported by Numba, then Numba will continue to use Python's original method to execute them, and more internal overhead will be generated during this process. Due to this reason, sometimes the compilation time takes longer even though we have used the Numba decorator.

To avoid this issue, we can apply the *nopython* compilation mode, where only need to change the decorator "*@jit"* to *"@jit(nopython=True)"*, Numba will force to use LLVM library and compile the indicated code. However, the exception will be thrown if some function or data type is not supported.

Numba optimization test:

To validate the implementation of the speedup optimization, we create a short script to test the performance of Numba: one compute transfer function with the Numba decorator, another same compute transfer function without Numba decorator, each one run 1000 times, and print the total time consumption using the python "*datetimes*" library, also the computed result. The expected result should be that both methods have the same output, and the one has Numba decoration takes

less time. The below screenshot Fig 17. shows the time comparison in two cases, validating the previous assumption:

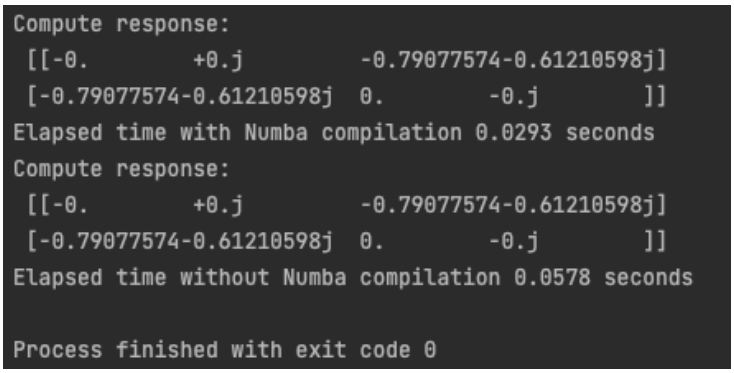


Fig.17. Numba performance testing

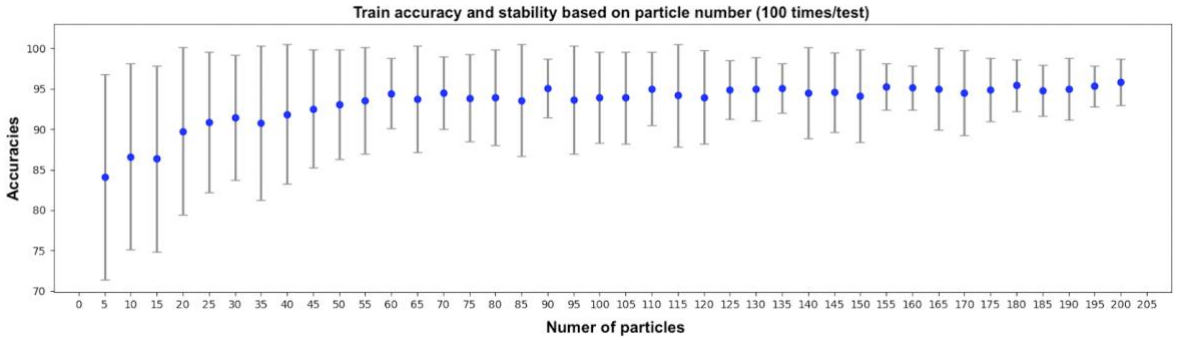
Since both compute methods have the same output result, it is verified that the Numba does not affect the calculation results. On the other hand, the method with Numba compilation takes 0.0293 seconds in 1000 times iterations, instead of 0.0578 seconds for the method without Numba. Therefore, the following conclusions can be drawn: The Numba can improve by 49.3% in the PUC compute transfer function without affecting the result.

IV.2. Number of particles optimization

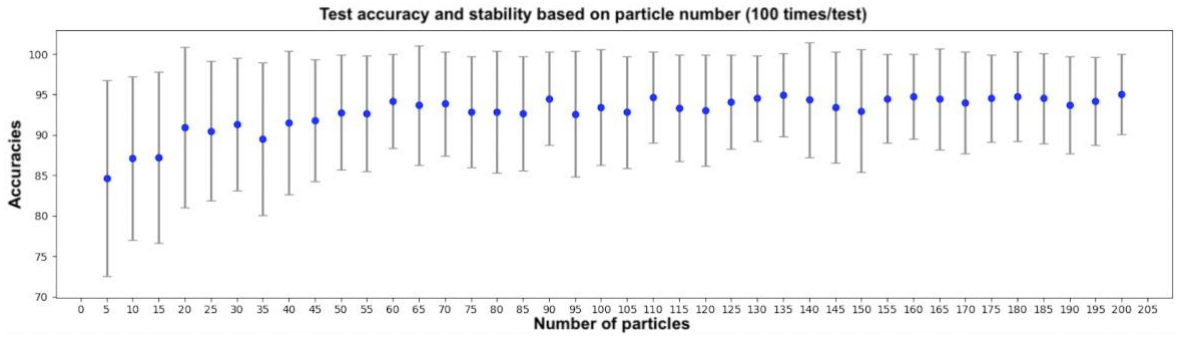
In most applications of PSO, people usually follow the recommendation of the original paper 1995 and limit the population size (particle number) to 20-50 particles. However, this value is application dependent and impacted by the search-space dimension and span. In our case, the 64-dimension space demands a greater number of particles to obtain a good performance. On the other hand, the selection of a large number of particles will consume more time and energy, thus the number of particles always incurs trade-offs. In this section, we perform an experiment to find out the optimal particle number.

Particle number optimization test:

The experiment workflow is as follows: first, the same photonic neural network that was shown in the previous chapter will be used, which has 64 phase shifters that need to be optimized by PSO. The setting of PSO is basically the same as before, except for the number of particles. In this section, the number of particles is set from 5 to 200 with a step of 5, and each test is run 100 times to reduce the random nature of the experiment. The mean and standard deviation for each experiment is evaluated. The results are summarized in the following figure:



(a)



(b)

Fig.18. Optimal particle number experiment,
the training accuracy (a) and testing accuracy(b) respectively

According to the observation in figure 18., the accuracy rate is slowly increasing when the number of particles is between 5-50, and the standard deviation is large, which indicates that the model is not well trained and the stability is low. After particles are more than 50, the accuracy tends to be stable, the standard deviations become smaller and the average accuracies are around 90%. To achieve a good trade-off between high average accuracy and low variance versus reduced population (particles) we select 60 particles as the optimal configuration. As an example, the test with a PSO with 200 particles took 1317.245 seconds in the PNN training (Fig. 18), whereas the test with 60 particles requires 322.686 seconds.

```
Test accuracy of the dataset 93.33333333333333
{'training': 97.5, 'test': 93.33333333333333}
         127216511 function calls (126891908 primitive calls) in 322.696 seconds

   Ordered by: internal time
```

Fig.19. Profile result with 60 particles

The result demonstrates that using 60 particles can significantly improve the training speed (faster converge of the model), while maintaining a good training and test performance. In conclusion: compared with 200 particles in PSO, where using 60 particles can improve 75.5% training speed without the accuracy loss.

IV.3. Early-stopping in Optimizer

A typical problem in neural network training is the execution of a large number of training epochs, where the model accuracy is not improving significantly. Early stopping is a method that allows specifying an arbitrarily large number of training epochs and stops training once the model performance has stopped improving on a holdout validation dataset. Moreover, the early stopping can also save time and resources. For instance, for 3 hyperparameters optimization of PSO, if each parameter has 20 candidates, avoiding the accumulation of epochs where the system is not being significantly improved is critical for the experiment efficiency. In other words, the algorithm can be used to terminate the tasks with poor performance in advance, or the tasks that have already reached the optimal solution, accelerating the entire process.

PySwarms provides a method to do the early stopping during the processing. To do so, we will configure the algorithm to end the process if the best cost found does not improve by more than the tolerance set by the user. In addition, since a fixed tolerance might seem too restrictive, another feature "*ftol_iter*" provides the optimizer some breathing space to improve the best cost. If there are no improvements detected over ftol_iter subsequent iterations, the optimizer stops.

Early stopping optimization test:

To validate the performance of the implementation of the early stopping in the photonic neural network training, we employ the previous PNN model, but with a maximum number of epochs of 500.

Moreover, for the PSO early stopping settings, the threshold *"ftol"* and the threshold iteration *"ftol_iter"* are needed to be considered. In this experiment, the *"ftol"* is set to 1e-4, which is the precision of the cost function. The threshold iteration "*ftol_iter*" will determine the trade-off between the stability of accuracy and computation cost: when "*ftol_iter*" is low, the amount of optimization calculation will decrease, but the model will become unstable and the accuracy will have a large variation. Therefore, the lowest "*ftol_iter*" is required while keeping the accuracy as stable as possible. The "*ftol_iter*" is set in a range from 5 to 95, the step is 5, in each experiment had been repeated 100 times to check the accuracy and the variation. The result is shown in the below figure:

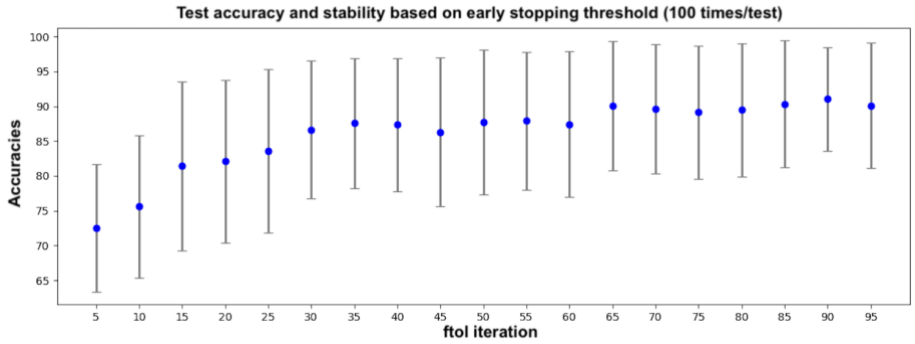


Fig.20. The accuracy in different "*ftol_iter*"

Based on the above figure, the accuracy rate exhibits a gradually increasing trend versus the "ftol_iter". Once we reach a value of 65, the benefits in terms of accuracy are not significant. This motivates the selection of this optimal value to provide both high accuracy and short variation. In order to understand the density distribution of the accuracy at this parameter setting, we generate the density distribution of the accuracy at this point:

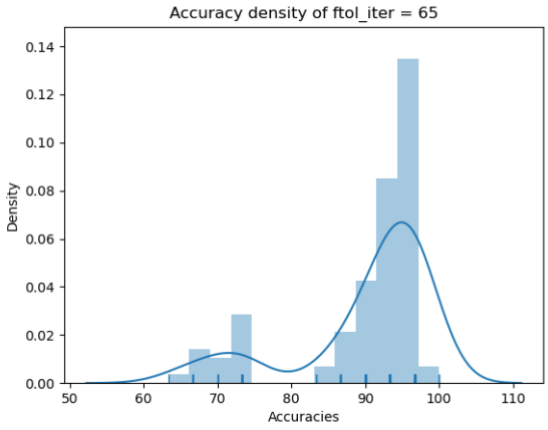


Fig.21. The density distribution of accuracy in *ftol_iter* = 65

In short, setting the early stopping threshold "ftol" to 1e-4 and the threshold iteration "ftol_iter" to 65 times, finalizes the process when the cost does not change by more than 1e-4 under 65 times iterations. The below figure demonstrates the effect of applying the early stopping on PNN with 500 times iterations:

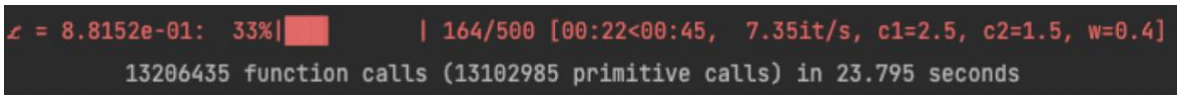


Fig.22. Performance of the early stopping on PNN with 500 iterations

It can be clearly seen that the process stops at the 164th iteration of training and the total processing time has been also reduced, which only takes 23 seconds/test, as a comparison, a full 500 iterations of the training process took 95 seconds/test, in which the result has been shown as below:

Fig.23. Time consumption of PNN training process with full 500 iterations without early stopping

In summary, the early stopping is able to save 75% of processing time, meanwhile can maintain high model accuracy.

IV.4. Multi-processing in Optimizer

As mentioned before, since Python is a dynamic and general-purpose language, it can run relatively slowly when compared with other programming languages. Apart from the above optimization methods, parallel processing or multiprocessing is also a good optimization scheme.

Parallel processing is an approach that CPU bounds tasks are run simultaneously on multiple processors in the same computer, and multiple processes can run in memory completely independently. The purpose of this working mode is to reduce the total task processing time.
PSO contains massive computations of independent nature. In particular the evaluation of each particle, increases the computational cost. To handle this benefit, we integrate the Pyswarm parallel computing method that allows parallel particle evaluation employing parallel processors, by indicating the number of processes.

PSO multiprocessing optimization test:

As the first step to enable the multiprocessing in PSO, it's important to know how many cores need to be assigned since the communication between processes will have an additional overhead. In other words, the performance could be reduced if too many processing cores are set in the task, and the total task time increase. The same PNN model and PSO optimizer without the early stopping feature has been used to select the optimum number of processes, the script is run on a computing server, which has 256G RAM and 80 cores.
For the experiment we set the number of processes from 1 to 40, and run each simulation 10 times while recording the time consumption of the processing task. The final result is plotted in the following figure:

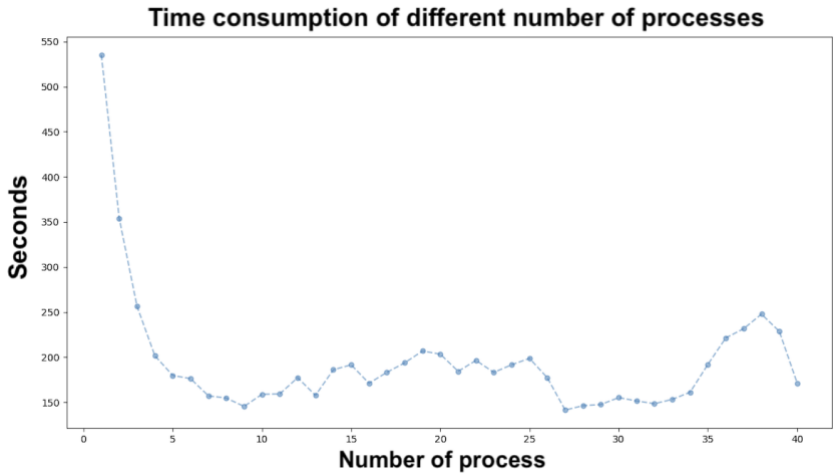


Fig.24. The time consumption with different numbers of processing

The optimum number of processes is 26 according to the result, and it takes lower than 150 seconds for 10 repetitions, as a comparison, the standard optimizer without parallel takes 540 seconds to run the 10 times simulation, which is shown in the first point of the above Fig.24. Therefore, it is obvious to conclude that using 26 multiprocessing can improve 72% speed performance compared with single-core processing. However, with 9 multiprocessing can achieve good result as well, it is not used here due to the fluctuations between 10 – 25 processors. In contrast, with 26 – 34 processors can perform a stable (flat curve) and high-speed processing.

Processing speed optimization conclusion:

Up to now, we have optimized the python processing speed. Figure 25 shows the result of time consumption after the optimization process. The result of 36.997 seconds for 10-time tests, is 35.6 times quicker than the initial simulation shown in Figure 16.

With the optimizer improved, for the next step, we will optimize the PNN's parameters to improve the performance of the photonic neural network. We will proceed as follows: first, we will find the optimal activation function, and secondly, the hyperparameter setting of the PSO optimizer.

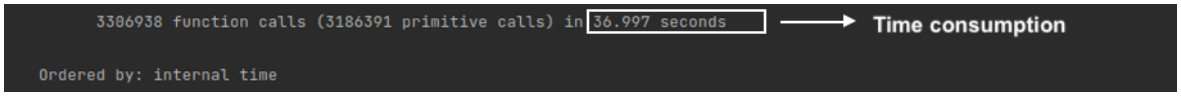


Fig.25. The time consumption after all processing speed optimization

total time consumption is 36.997 seconds

IV.5. Non-linear function parameter optimization

The ReLu activation function is a piecewise linear function where the input will be fully transmitted if the cutoff is positive, due to the slope of the function is 1, otherwise, the input will be attenuated to zero, when the cutoff is negative.

As mentioned in the theoretical section, typically the cutoff is set to zero as default. Since the cutoff acts as a threshold to control whether the function is activated, therefore, the different cutoff settings will theoretically affect the convergence of PNN and remains application dependent. In this part, the different cutoffs will be tested in order to figure out the optimum setting of ReLu which can bring the best performance of PNN.

Cutoff setting optimization test:

Firstly, we set the value range of cutoff between 0 - 1, and 0.1 as the step. Next, each step of the experiment is run 100 times while recording the PNN accuracy, not only for training but also for testing. After that, we take the average accuracy of each cutoff step and plot the below Figure 26. in order to observe how cutoff can affect the performance in PNN:

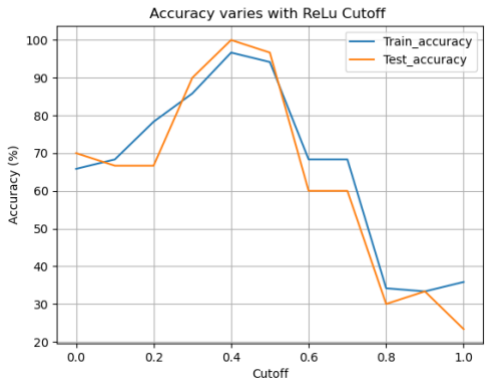


Fig.26. The effect of ReLu cutoff on PNN performance

According to the above figure, good performance of PNN can be obtained to get higher accuracy when the cutoff is between 0.3 – 0.5

IV.6. Optimizer's Hyper-parameter optimization

Hyperparameters' selection in PSO is the key influence on the performance and efficiency of the algorithm. There are no general methods to determine the optimum parameters, which are selected by user experience because of different parameter spaces and the relativity of each variable. However, the regularity of different parameters that influences the performance of the algorithm could be found for each application. In this section, the performance of PSO is analyzed based on the control hyperparameters variants, which include accelerating constant c1 and c2, as well as the inertia weight w. These parameters of PSO are applied in the photonic neural network, and PNN's accuracies are checked. [25]

The experiment is as follows: first, the accelerate constants c1 and c2 are set their boundaries from 0.5 to 3.5 with 20 steps, and the inertia weight is set from 0.1 to 0.9 with 10 steps. Since each independent simulation has a random nature, each simulation has been run 30 times per case, finally will get a result with a dimension of 20*20*10. Below Fig.27. shows the result that has been plotted in 4 dimensions:

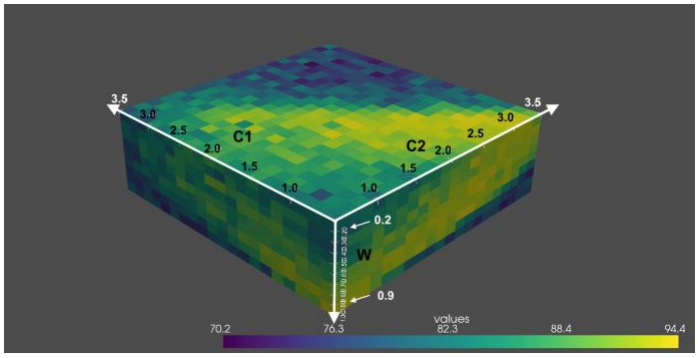


Fig.27. The test accuracy plotted in 4D

Next, to enable an efficient analysis, we can make the above cube into 9 different slices based on the different inertia weights, and observe the relation between the hyperparameters. Below figure 28 shows the related result:

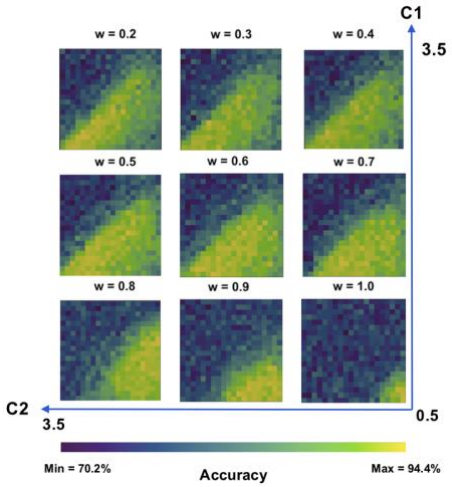


Fig.28. The accuracy heatmap in different inertia weight

The yellow area is big in the above figure which represents the PNN has better accuracy and stability, the inertia weight range w = 0.5 - 0.7 achieves less dependencies with the remaining parameters. Another observed information is when the inertia weight is greater than 0.8, the blue area in the figure gradually becomes larger, which represents the accuracy is decreasing. In addition, the optimal value of c1 and c2 are is within the ranges 0.5-1.5 and 1-2.8, respectively. The best result is obtained when w = 0.2, c1 = 1.29, and c2 = 2.71, although the most stable setting option for the inertia weight are 0.5 and 0.7. The best result is shown the below figure:

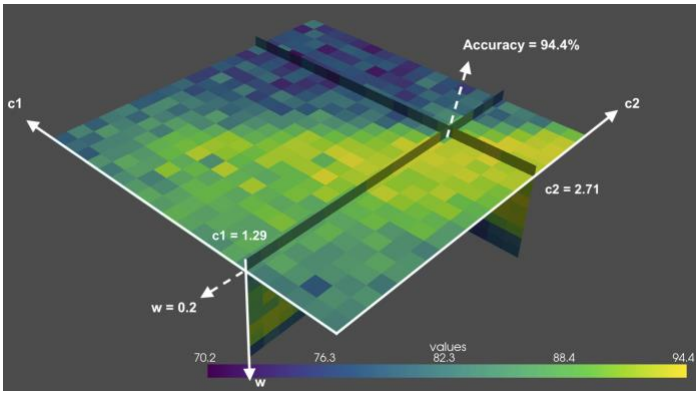


Fig.29. The best result of the PNN model, the hyper-parameters setting of PSO is w = 0.2, c1 = 1.29, c2 = 2.71 and accuracy = 94.4%

In addition, the best hyperparameters actually do sum to 4, and add a bit more on the observed fact when c1 + c2 > 4, the accuracy drops. This conclusion verified the recommendation of the original paper by Kennedy et al. in 1995[5], and it also demonstrated that the PSO hyperparameter configuration theory is applicable to PNN.

The PSO hyperparameters found in this work only targets to the specific model and dataset since the optimization process is application dependent, therefore, this process needs to be carried out for each application. Nevertheless, a more efficient way is to perform a PSO to the three hyperparameters rather than a continuous grid-sweep, this approach will be experimented in the further work.

V. CONCLUSIONS & FURTHER WORK

V.1. Conclusions

As an interdisciplinary technology of photonic and artificial intelligence, the PNN combines the network structure's advantages of high-speed and low-power consumption that can break the bottleneck of traditional electronic neural networks. This combination can pave the way for the next-generation computing platforms for artificial intelligence.

Under this work, we started by studying and analyzing the fundamentals of artificial intelligence and programmable integrated photonics:

- The theoretical learning and understanding of "the fully connected ANN and the free-gradients PSO algorithm".
- The characteristics of PUCs and the PPICs.

In the second part, we have implemented a framework to simulate in an efficient way a photonic-based neural network from the basic programmable unit cell to a complex mesh arrangement with the configuration protocol based on the Clements algorithm.

- The PNN simulator was developed using Python language, and the simulated PNN model able to perform the classification tasks on the iris dataset.
- The integration of PSO as a free gradient optimizer in the PNN framework simulator and its practical validation for training and inference.

Then, with the preliminary PNN simulator successfully realized we continued with the optimization and testing of the PNN simulator. It includes:

- Combination of the methods of Numba decorator, multiprocessor, and some other optimization. We achieved an improved performance by reducing the initial simulation time 40 times while keeping the PNN model stable and analyzed the impact of each parameter.
- The best selection of the optimal hyperparameter setting through sweeping the entire PSO hyperparameters, and the obtention of the best performance of PNN model.

Collectively, a PNN model was created by using the implemented simulator, and the created PNN can be trained under a short time and perform 94.4% accuracy in the classification task.

This implemented PNN simulator had been adopted by the spin-off company of the Universitat Politècnica de València, iPronics programmable photonics S.L., and merged into iPronics's Software Development Kit. These libraries are being employed by big technology companies and academic institutions worldwide, who are actively working on this field. Moreover, this project confirmed that the implemented PNN simulator can play an important role in short and long-term photonic neural network research.

V.2. Further work

Despite the positive result of PNN simulator in this thesis, there is still much future work to accomplish in the goal to achieve a reliable photonic computer: the PNN computation speed simulation and the neural network energy consumption are not included in the outcomes of this work. To address those items, firstly, the time consumption of PNN model some physical parameters such as photodetection rate, modulator rate, phase shifters speed as well as the matrix multiplication size should be considered and implemented in the simulator and the optoelectronic hardware.

Secondly, as for the energy efficiency, the power consumption during computation is dominated by the optical power necessary to trigger an optical nonlinearity and achieve a sufficiently high signal-to-noise ratio (SNR) at the photodetectors. The integration of this metrics would enrich the framework developed in this master thesis and help with the analysis extension.[11]

Finally, it is interesting to extend the compatibility of PNN simulator, which can be used in other machine learning and artificial intelligence frameworks such as TensorFlow, PyTorch, Scikit-Learn, etc., in this way to reach more users and ensure a soft or transparent workflow for data practitioners and machine learning developers.

ACKNOWLEDGMENT

This thesis project means lot to me, it opens a new world and helps me find the direction that I am willing to put in my enthusiasm.

First of all, I would like to thank my supervisors, Prof. José Capmany and Dr. Daniel Perez. I still remember that after I left my last job and planned to go back to university to finish my master, I was anxious about whether I would be able to join any research team to be competent in any program, I really appreciate that you gave me such an interesting project and an opportunity to join iPronics team. I am still motivated by the speech of Jose at the team meeting and it makes me more convinced of the direction that need to work on. Dani, your passion for science is also subtly inspiring me, as well as the time spent with you, your way of thinking and working will also instill my best habits.

I also want to thank you, my tutor, David Sanchez, I have learned a lot of things from your side, and I am very enjoying the brainstorming time that spent with you, your point of view always gives me new thinking. You are always so selfless in sharing knowledge with me and I learned more and more under your patient guidance. Moreover, I would like to thank the rest teammates at iPronics, you guys let me know a wider range of knowledge, and also, I am very enjoying our daily lunch and coffee time with you guys.

Yue Zhang, my girlfriend, thanks for your unconditional support in any of my decisions, and thank you for your counsel and sympathetic ear.

Finally, Dad and Mom, no amount of words can express my love for you. Although we are far apart, my love is always there for you and you are also my most peaceful harbor forever.

REFERENCE

[1] Zaharia, Matei, et al. "Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing." 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). 2012.

[2] Waldrop, M. Mitchell. "The chips are down for Moore's law." Nature News 530.7589 (2016): 144.

[3] Shen, Yichen, et al. "Deep learning with coherent nanophotonic circuits." Nature photonics 11.7 (2017): 441-446.

[4] Raschka, Sebastian. Python machine learning. Packt publishing ltd, 2015.

[5] Kennedy, James, and Russell Eberhart. "Particle swarm optimization." Proceedings of ICNN'95-international conference on neural networks. Vol. 4. IEEE, 1995.

[6] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.

[7] Sengupta, Biswa, and Martin B. Stemmler. "Power consumption during neuronal computation." Proceedings of the IEEE 102.5 (2014): 738-750.

[8] Epelbaum, Thomas. "Deep learning: Technical introduction." arXiv preprint arXiv:1709.01412 (2017).

[9] Pérez, D., Gasulla, I., Mahapatra, P. D., & Capmany, J. (2020). Principles, fundamentals, and applications of programmable integrated photonics. Advances in Optics and Photonics, 12(3), 709-786.

[10] Capmany, José, and Daniel Pérez. Programmable integrated photonics. Oxford University Press, 2020.

[11] Bogaerts, W., Pérez, D., Capmany, J., Miller, D. A., Poon, J., Englund, D., ... & Melloni, A. (2020). Programmable photonic circuits. Nature, 586(7828), 207-216.

[12] Pérez, Daniel, Ivana Gasulla, and Jose Capmany. "Field-programmable photonic arrays." Optics express 26.21 (2018): 27265-27278.

[13] Clements, W. R., Humphreys, P. C., Metcalf, B. J., Kolthammer, W. S., & Walmsley, I. A. (2016). Optimal design for universal multiport interferometers. Optica, 3(12), 1460-1465.

[14] Robert W. Schirmer and Alexander L. Gaeta, "Nonlinear mirror based on two-photon absorption," J. Opt. Soc. Am. B 14, 2865-2868 (1997)

[15] Dai, Hou-Ping & Chen, Dong-Dong & Zheng, Zhou-Shun. (2018). Effects of Random Values for Particle Swarm Optimization Algorithm. Algorithms. 11. 23. 10.3390/a11020023.

[16] W.-K. Chen, Linear Networks and Systems (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.

[17] He, Yan, Wei Jin Ma, and Ji Ping Zhang. "The parameters selection of PSO algorithm influencing on performance of fault diagnosis." MATEC Web of conferences. Vol. 63. EDP Sciences, 2016.

[18] Poor, H. Vincent. An introduction to signal detection and estimation. Springer Science & Business Media, 2013.

[19] HaiTao L, Xi C, QiMing Z, et al. "Artificial Intelligence Nanophotonics: Optical Neural Networks and Nanophotonics" (in Chinese) [Online]. Available: https://www.opticsjournal.net/Articles/OJ5d3954d7d27d0615/FullText

[20] Zhou, Hailong, et al. "Photonic matrix multiplication lights up photonic accelerator and beyond." Light: Science & Applications 11.1 (2022): 1-21.

[21] Jiang, Jiaqi, Mingkun Chen, and Jonathan A. Fan. "Deep neural networks for the evaluation and design of photonic devices." Nature Reviews Materials 6.8 (2021): 679-700.

[22] PySwarms. (2022). [Online]. Available: https://pyswarms.readthedocs.io/en/latest/

[23] Pérez, Daniel, et al. "Reconfigurable lattice mesh designs for programmable photonic processors." Optics Express 24.11 (2016): 12093-12106.

[24] Y. Xue, T. Tang and A. X. Liu, "Large-Scale Feedforward Neural Network Optimization by a Self-Adaptive Strategy and Parameter Based Particle Swarm Optimization," in IEEE Access, vol. 7, pp. 52473-52483, 2019, doi: 10.1109/ACCESS.2019.2911530.

[25] He, Yan, Wei Jin Ma, and Ji Ping Zhang. "The parameters selection of PSO algorithm influencing on performance of fault diagnosis." MATEC Web of conferences. Vol. 63. EDP Sciences, 2016.