



UNIVERSITÀ
DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Ingegneria Informatica, delle Comunicazioni ed Elettronica

ELABORATO FINALE

RETI NEURALI OPTOELETTRONICHE

*Simulazione di una rete neurale optoelettronica integrata e
confronto con un'implementazione su FPGA.*

Supervisore
Philippe Velha

Laureando
Giovanni Solfa

Anno accademico 2022/20023

Ringraziamenti

Desidero esprimere la mia sincera gratitudine a tutti coloro che, nel corso di questi anni, hanno alimentato la mia passione per queste materie. In particolare, desidero ringraziare il professor Philippe Velha per il suo costante supporto nello sviluppo di questa tesi. Vorrei anche esprimere la mia riconoscenza a Katia e Luca, i miei genitori, che hanno creduto in me e reso possibile tutto questo, così come a Giada, mia morosa, che mi ha sostenuto durante tutti questi tre anni.

Inoltre, desidero estendere il mio ringraziamento a tutti coloro che, prima di me, hanno dedicato tempo e impegno alla ricerca e allo sviluppo delle tecnologie informatiche, plasmando la realtà che conosciamo oggi. Mi auguro profondamente che questa potenza e conoscenza non vengano mai utilizzate a fini distruttivi, arrecando danni a persone o cose.

Indice

Sommario	3
1 Rete neurale	5
1.1 Introduzione	5
1.2 Teoria delle reti neurali	5
1.2.1 Percettrone (Perceptron)	5
1.2.2 Tipologia di rete neurale	6
1.2.3 Strato	7
1.2.4 Addestramento (Training)	8
2 Caratteristiche del circuito ottico	11
3 Revisione della letteratura	13
3.1 Introduzione	13
3.2 Processore ottico basato su MZI	14
3.2.1 Introduzione	14
3.2.2 Teoria	15
3.3 Unità di elaborazione basata sull'interferenza multimodale	16
3.3.1 Introduzione	16
3.3.2 Teoria	16
3.4 Un neurone completamente ottico con funzione di attivazione sigmoidea	17
3.4.1 Introduzione	17
3.4.2 Teoria	17
4 Simulazione e valutazione di una rete neurale optoelettronica	19
4.1 Introduzione	19
4.2 Caratteristiche vantaggiose	19
4.3 Caratteristiche svantaggiose	19
4.4 Simulazione e implementazione	19
4.4.1 MZI	20
4.4.2 Moltiplicazione di matrici MZI	20
4.4.3 Doppio SOA	21
4.4.4 Singolo strato	23
4.4.5 Set di dati	23
4.4.6 Risultati e considerazioni	24
5 Implementazione FPGA	27
5.1 Introduzione	27
5.2 Schema	27
5.3 Stato dell'arte	27
5.4 Risultati e considerazioni	29

6 Confronto ONN-FPGA	33
6.1 Gestione degli ingressi e delle uscite	33
6.2 Facilità di sviluppo	33
6.3 Integrabilità	33
6.4 Throughput	33
6.5 Accuratezza	34
6.6 Latenza	34
6.7 Consumo energetico	34
6.8 Dimensioni	34
6.9 Conclusioni parziali	35
7 Conclusioni	37
Bibliografia	41
A Codice simulazione ONN	43

Sommario

L'intelligenza artificiale (AI - Artificial intelligence) è un campo in continua crescita, con applicazioni sempre più diffuse e complesse. Le reti neurali (NN - Neural netowrk) sono un componente fondamentale dell'AI, ma la loro implementazione su circuiti digitali presenta dei limiti in termini di velocità, efficienza energetica e scalabilità. In risposta a queste sfide, è emersa l'ottica integrata come una possibile alternativa promettente per la realizzazione di NN. La presente tesi propone un confronto approfondito tra le reti neurali ottiche (ONN - Optic NN) e le reti neurali digitali (DNN - Digital NN), attraverso la simulazione di una ONN e l'implementazione di una DNN su un Field-Programmable Gate Array (FPGA). La simulazione della ONN è stata condotta utilizzando la libreria Photontorch di Python, che ha permesso di sviluppare e simulare componenti ottici personalizzati. L'obiettivo è quindi riuscire a valutare e confrontare le prestazioni delle ONNs, considerando in particolare la loro capacità di elaborare segnali ad alta velocità. Tuttavia, la simulazione della ONN ha presentato sfide significative a causa delle estese tempistiche computazionali e delle limitazioni nell'attuale sviluppo della libreria. Successivamente, la stessa topologia usata per la ONN è stata implementata su un FPGA utilizzando l'ambiente di sviluppo Vivado. Sono stati sviluppati blocchi personalizzati e sfruttati componenti preesistenti per replicare il design della NN sulla piattaforma hardware. Queste implementazioni hanno quindi permesso di valutare e confrontare le prestazioni della ONN con una DNN in termini di velocità di elaborazione, potenza, accuratezza e latenza. Attraverso il confronto dettagliato tra le caratteristiche delle ONNs e delle DNNs, sono emerse considerazioni importanti riguardanti l'efficienza energetica, la velocità di elaborazione e la complessità di implementazione. Sebbene le ONNs promettano prestazioni superiori in termini di velocità, la loro implementazione pratica richiede ancora approfondimenti e soluzioni per superare le sfide attuali, come la necessità di un modo efficiente per simulare reti complesse e software che integrino la creazione di differenti topologie in modo grafico con la possibilità di validarle. In conclusione, questa tesi offre una panoramica dettagliata delle reti neurali ottiche, evidenziando i loro punti di forza e le sfide da affrontare in rapporto alla tecnologia digitale esistente.

1 Rete neurale

1.1 Introduzione

L'apprendimento automatico (ML - Machine learning) è una tecnica di ottimizzazione e analisi statistica che agevola l'impiego di vasti volumi di dati per formulare previsioni, svolgere operazioni di classificazione e perseguire altri obiettivi analoghi. Questa metodologia di analisi dei dati e della loro utilizzazione si attesta come particolarmente diffusa in virtù della sua capacità di apprendimento mediante algoritmi di ottimizzazione, eliminando così la necessità di specificare formule precise e universali.

Il ML si può dividere in tre macro categorie:

- ML supervisionato
- ML non supervisionato
- ML con rafforzamento

In termini più tecnici, il ML supervisionato si basa su un paradigma di apprendimento in cui il modello viene addestrato utilizzando un insieme di dati etichettati, fornendo quindi una corrispondenza tra gli ingressi e i risultati desiderati. Ciò consente agli algoritmi di ottimizzazione di adattare i parametri interni del modello per minimizzare l'errore tra l'uscita predetta e l'uscita desiderata. D'altra parte, il ML non supervisionato si concentra sull'apprendimento dalla struttura intrinseca dei dati, senza la presenza di etichette esplicite. Gli algoritmi di questo tipo di approccio cercano di identificare modelli, strutture o raggruppamenti all'interno dei dati senza conoscere in anticipo l'uscita desiderata. Infine, il ML con rafforzamento costituisce un paradigma in cui un sistema apprende attraverso il feedback ottenuto dall'ambiente dopo aver fatto una scelta. In questo contesto, il processo decisionale e la valutazione del feedback si basano sull'impiego di una funzione di costo che il sistema mira a minimizzare nel lungo termine. Tale funzione di costo rappresenta un criterio di valutazione del comportamento del sistema, guidandone l'adattamento e l'ottimizzazione delle decisioni nel corso del tempo.

1.2 Teoria delle reti neurali

1.2.1 Percettrone (Perceptron)

La prima categoria, così come le altre sottoclassi, include diverse categorie di intelligenza artificiale tra cui le reti neurali artificiali (ANNs - Artificial NNs). Questo tipo di approccio si basa sull'unità fondamentale chiamata percettrone o neurone artificiale (artificial neuron) che, quando connessa ad altre unità simili, consente la creazione di NNs. L'adozione di questa terminologia deriva dai concetti utilizzati in biologia per descrivere un neurone e il cervello, dai quali sono stati tratti idee per sviluppare tali modelli. In particolare, similmente al cervello animale, ogni neurone esegue una somma pesata dei segnali in ingresso cambiando i pesi in base a stimoli precedenti o lo stato attuale del sistema, per poi applicare una funzione di soglia che determina l'uscita, così da propagare l'informazione ricevuta. Analogamente i modelli matematici dei neuroni prevedono di eseguire una combinazione lineare del vettore di ingresso, seguita da una funzione di attivazione che simula il processo di soglia nella comunicazione neurale.

Nell'illustrazione 1.1, è possibile osservare il vettore di ingresso $X = [x_1, x_2, \dots, x_n]$, accompagnato dai corrispondenti pesi $W = [w_1, w_2, \dots, w_n]$. Attraverso una combinazione lineare dei valori di ingresso $X^1 = (x_1w_1 + x_2w_2 + \dots + x_nw_n)$, essi vengono utilizzati come ingresso per una funzione di attivazione $H(z)$, la quale, produce quindi l'uscita effettiva $Y = H(X^1)$ del neurone. È importante notare che ogni neurone è composto da due componenti distinte: una parte iniziale lineare e una parte

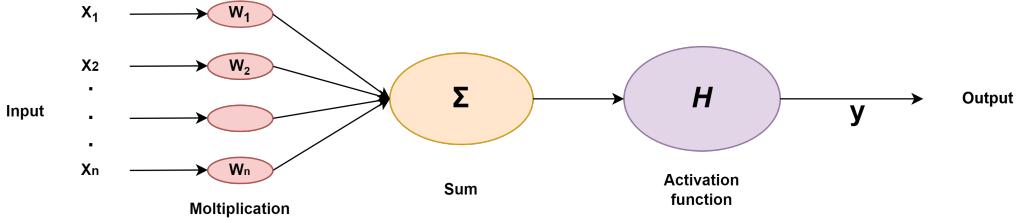


Figura 1.1: Uno schema di base di un percepitrone.

successiva non lineare. Inoltre, per completare la definizione, viene introdotto un ulteriore ingresso noto come bias (solitamente con valore unitario) e un peso specifico associato determinando così un ulteriore parametro dal valore di $x_{n+1}w_{n+1}$.

1.2.2 Tipologia di rete neurale

I neuroni che costituiscono una ANN sono organizzati in strati, ciascuno dei quali rappresenta un gruppo di neuroni che riceve segnali in ingresso e trasmette la propria uscita allo strato successivo. Il primo strato, in cui arrivano gli ingressi, è denominato strato di ingresso, mentre l'ultimo strato, da cui preleviamo le uscite della rete, è chiamato strato di uscita. Gli strati intermedi tra lo strato di ingresso e lo strato di uscita sono noti come strati nascosti. La connessione tra i vari strati avviene collegando le uscite dello strato m con gli ingressi dello strato $m + 1$ ($m \in N$), in un approccio denominato "Denso" (Dense) o "Completamente connesso" (Fully connected). È possibile modificare questa struttura, ottenendo diverse configurazioni della rete, mediante operazioni di potatura (pruning), ovvero il taglio selettivo delle connessioni ottimizzando così la rete o dando a questa diverse proprietà. Un'altra alternativa per modificare le connessioni della rete è utilizzare connessioni ricorrenti che collegano un neurone a se stesso o un neurone ad un altro neurone in uno strato precedente, creando così un effetto di "memoria" in quanto le successive uscite dipendono anche dalle uscite correnti.

Attraverso diverse configurazioni degli strati e delle connessioni, nonché l'integrazione di altri processi informativi (e strutture matematiche), è possibile creare diverse tipologie di ANNs. Le tipologie principali di ANNs sono le seguenti:

- Rete neurale multistrato (Multi-layer NN)
- Rete neurale convoluzionale (Convolutional NN)
- Rete neurale ricorrente (Recurrent NN)
- Rete neurale profonda (Deep neural network)

La prima tipologia è la rete neurale multistrato, che rappresenta la forma più comune di ANN. Questa è costituita da diversi strati composti da un numero variabile di neuroni, con una densità di connessioni tra gli strati che può differire a seconda dell'architettura specifica. La rete neurale convoluzionale è invece la tipologia predominante nelle NNs che analizzano immagini. Essa è composta da una rete neurale multistrato, ma con l'aggiunta di uno strato di convoluzione iniziale che genera gli ingressi dell'effettiva NN. Questo approccio permette di cogliere le relazioni spaziali e locali all'interno delle immagini. La rete neurale ricorrente è una tipologia di ANN in cui sono presenti neuroni che, attraverso le loro connessioni, generano un ciclo all'interno della rete. Ciò consente alla rete di avere un comportamento dipendente non solo dagli ingressi attuali, ma anche dalla storia degli ingressi precedenti. Questa caratteristica di memoria si rivela particolarmente utile in problemi che richiedono considerazione del contesto temporale. Infine, le reti neurali profonde rappresentano una classe delle NNs caratterizzate da un numero elevato di strati e interconnessioni tra di essi. Queste reti consentono di modellare complessità crescenti e di apprendere rappresentazioni gerarchiche dei dati, aprendo la strada a una maggiore capacità di rappresentazione e comprensione delle informazioni. Attraverso l'adozione di queste ed altre diverse tipologie di ANNs riportate in figura 1.3, è possibile affrontare una vasta gamma di problemi e sfruttare le specifiche peculiarità di ciascuna architettura per ottenere risultati ottimali nei vari contesti applicativi.

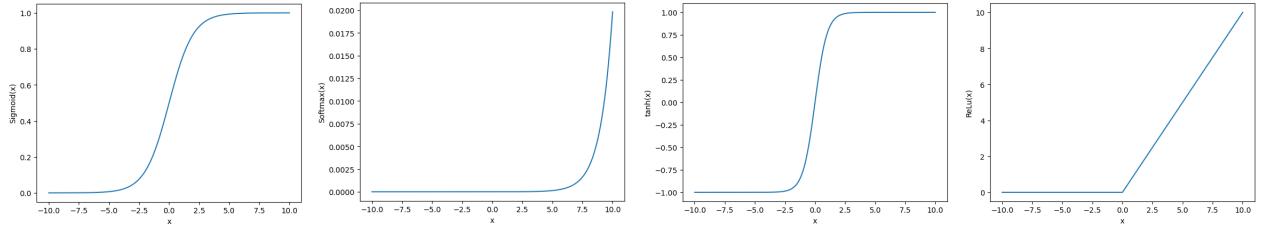


Figura 1.2: L'immagine mostra quattro funzioni di attivazione, rispettivamente: funzione Sigmoid, funzione Softmax, funzione Tanh, funzione ReLu.

1.2.3 Strato

Uno strato, simile a un neurone, può essere visto come composto da due parti sequenziali: una parte lineare e una parte non lineare. La parte lineare dello strato coinvolge la moltiplicazione dei valori di ingressi e dei relativi pesi, inclusi i bias, per generare gli ingressi della parte non lineare. Questa parte può essere sintetizzata tramite l'utilizzo di una matrice di pesi che, quando moltiplicata per il vettore in ingresso, compreso il bias, produce i risultati parziali. In generale, è possibile creare una matrice di pesi di dimensioni $N \times M$, dove, $M - 1$ rappresenta il numero di ingresso (con il bias escluso), M rappresenta dunque il numero totale di ingressi compreso il bias ed infine N rappresenta il numero di uscite generate dalla parte lineare dello strato. Ogni elemento w_{ij} nella matrice indica il peso al quale va moltiplicato l'ingresso j , il quale risultato parziale forma uno degli M valori dell'addizione che generano l'uscita i .

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{m-1} \\ bias \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

La parte non lineare di uno strato implementa una funzione di attivazione per ciascun uscita derivante dalla parte lineare. Tra le funzioni di attivazione più comuni utilizzate nelle NNs vi sono:

- Unità lineare rettificata (ReLU - Rectified linear unit): questa funzione di attivazione restituisce il valore di ingresso se è positivo, altrimenti restituisce zero.

Matematicamente:

$$f(x) = \max(0, x) \quad (1.1)$$

- Sigmide (Sigmoid): la funzione sigmoid mappa l'ingresso in un intervallo compreso tra 0 e 1, rendendola adatta per problemi di classificazione binaria.

È definita come:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (1.2)$$

- Tangente iperbolica (Tanh): la funzione tangente iperbolica è simile alla sigmoid, ma mappa l'ingresso in un intervallo compreso tra -1 e 1.

È definita come:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (1.3)$$

- Softmax: questa funzione di attivazione viene comunemente utilizzata nell'ultimo strato di una rete neurale per problemi di classificazione multiclasse. La funzione softmax mappa i valori di ingresso in una distribuzione di probabilità, in modo che la somma di tutte le uscite sia uguale a 1. Ciò consente di interpretare le uscite come probabilità relative alle diverse classi.

Queste funzioni di attivazione, visualizzabili graficamente alla figura 1.2 introducono le caratteristiche non linearità all'interno delle NNs, consentendo loro di apprendere relazioni complesse e di modellare

dati non lineari. La scelta della funzione di attivazione dipende dal tipo di problema che si intende risolvere e dalle caratteristiche dei dati in esame. La presenza di una parte non lineare è essenziale altrimenti, senza la componente non lineare, tutti gli strati della NN potrebbero essere ridotti a un singolo strato lineare, limitando quindi la capacità della rete di modellare relazioni complesse tra gli ingressi e le uscite desiderate.

1.2.4 Addestramento (Training)

L'utilizzo di una rete neurale supervisionata richiede una fase di training, durante la quale gli algoritmi di ottimizzazione sono impiegati per determinare i pesi appropriati per ciascun strato (cioè i valori ottimali per ogni matrice). In questa fase, per ciascun ingresso viene calcolato l'uscita prevista e, attraverso una funzione di errore, vengono apportati miglioramenti ai pesi al fine di ridurre l'errore appena calcolato. Il processo mediante il quale l'errore viene propagato all'indietro nella rete e vengono scelti i cambiamenti da apportare ai pesi è noto come retropropagazione (backpropagation). Tale processo sfrutta la regola di derivazione di G. W. Leibniz per calcolare il gradiente e la derivata in ogni punto della rete, al fine di minimizzare la funzione di errore. Da questa idea fondamentale e in base alla tipologia di errore, sono stati sviluppati numerosi algoritmi per la fase di addestramento delle NNs. L'addestramento della rete neurale implica così un processo iterativo che richiede l'ottimizzazione progressiva dei pesi degli strati attraverso l'analisi dell'errore e l'aggiornamento dei parametri corrispondenti. L'obiettivo finale è raggiungere una configurazione dei pesi che minimizzi l'errore globale della rete e consenta una predizione accurata delle uscite per nuovi ingressi non ancora osservati.

A mostly complete chart of
Neural Networks

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

©2016 Fjodor van Veen - asimovinstitute.org

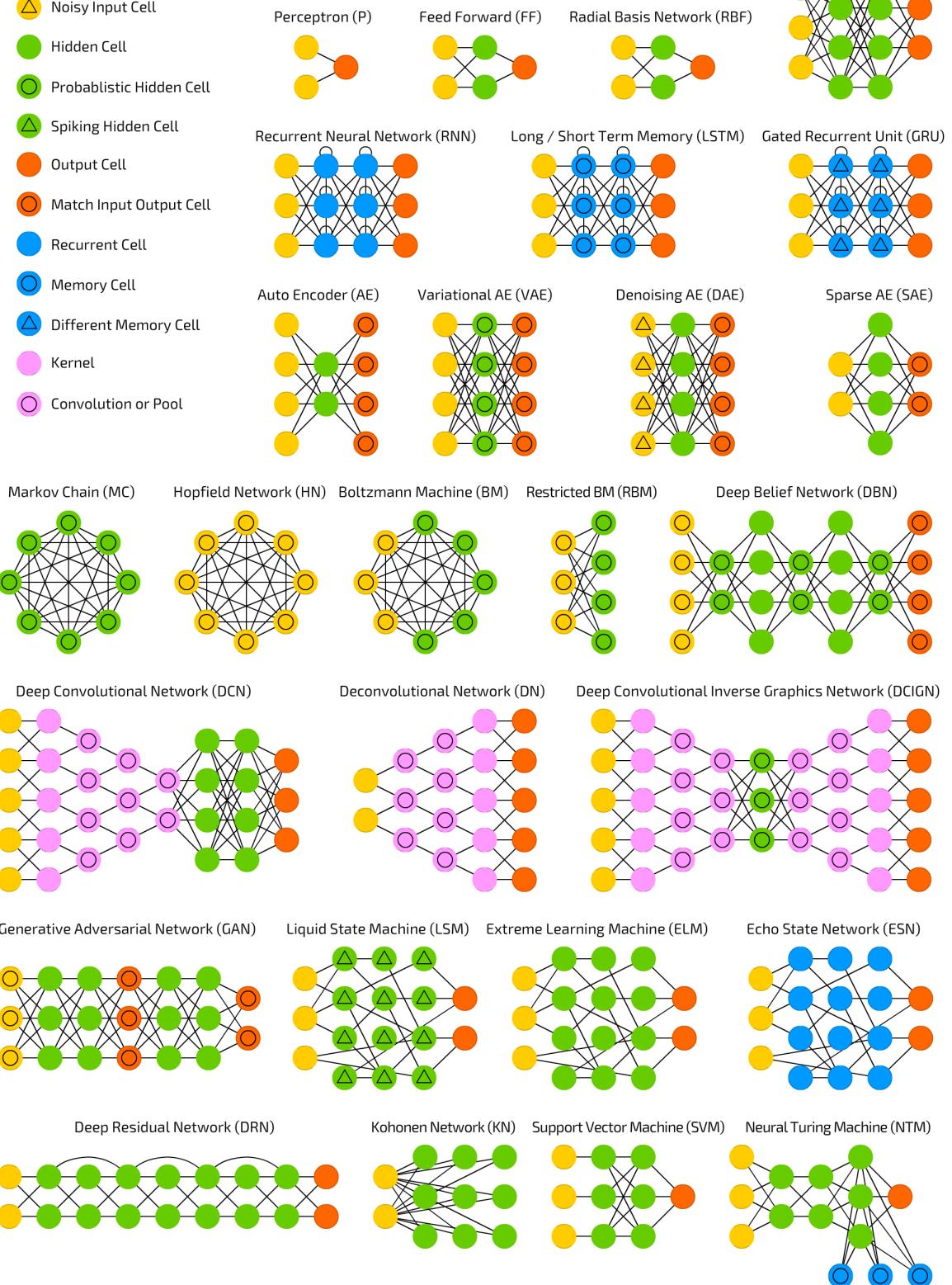


Figura 1.3: Principali tipologie di NNs. Source [5]

2 Caratteristiche del circuito ottico

Le attuali implementazioni di NNs si basano principalmente sull'elaborazione digitale dei dati, che presenta limitazioni significative in termini di velocità di calcolo e capacità di gestione di grandi quantità di dati. Negli ultimi anni, è emerso un crescente interesse per esplorare nuove tecnologie al fine di migliorare le prestazioni delle NNs. Una di queste direzioni di ricerca promettenti riguarda l'utilizzo delle ONNs, che sfruttano la luce come mezzo per l'elaborazione dei dati. Questo nuovo paradigma di elaborazione si basa sull'integrazione di circuiti ottici all'interno di schede elettroniche, creando così circuiti integrati ottici (PIC - Photonic integrated circuit). L'adozione di ONNs presenta numerosi vantaggi potenziali a livello teorico, che includono:

- Velocità di elaborazione dei dati: l'elaborazione più veloce e l'efficiente gestione di grandi quantità di dati a lunghe distanze rappresentano una sfida per le attuali implementazioni basate sull'elettronica, che incontrano dei limiti in termini di velocità di calcolo e capacità di trasmissione dei segnali elettronici. Grazie all'utilizzo dell'ottica, è possibile ottenere velocità di elaborazione dei dati superiori rispetto all'elettronica digitale. La luce infatti si propaga a velocità notevolmente più elevate rispetto all'elettronica permettendo inoltre di effettuare elaborazioni parallele ad alta velocità. Questa caratteristica consente di gestire grandi volumi di dati in tempi ridotti, consentendo prestazioni più rapide e tempi di risposta più brevi.
- Integrazione distribuita: l'ottica può essere utilizzata per creare sistemi distribuiti in cui i segnali vengono trasferiti su fibre ottiche ad alta velocità e basse perdite rispetto l'elettronica digitale. Questo approccio consente di distribuire la concentrazione di elementi e la densità di energia su una vasta area, riducendo la necessità di dissipare grandi quantità di calore da un singolo punto. Tuttavia, è importante considerare che i principali sistemi di elaborazione ottici sono basati su circuiti analogici e questo porta a dover considerare la natura non lineare del sistema che può causare funzionamenti indesiderati, richiedono così, l'imposizione di limitazioni delle prestazioni per mantenere il sistema stabile all'interno di limiti prestabiliti. È quindi necessario prestare attenzione alla gestione di tali effetti non lineari al fine di garantire un funzionamento ottimale del sistema nel contesto delle prestazioni desiderate.
- Riduzione dei consumi e risparmio energetico: l'ottica, in particolare l'ottica passiva, permette una riduzione dei consumi per elaborare i dati. Infatti, i componenti passivi sono elementi che non richiedono l'uso di energia. Oltre alla diminuzione dell'energia elettrica, non necessaria per tali componenti, con tale tecnologia si diminuisce la necessità di distribuire in modo capillare i segnali di potenza all'interno del circuito e, così facendo, si diminuiscono anche le dispersioni energetiche e la complessità.
- Aumento dei canali tramite multiplexing a divisione di lunghezza d'onda (WDM - Wavelength Division Multiplexing): grazie alla tecnologia WDM, è possibile integrare più segnali a diverse lunghezze d'onda nello stesso sistema, consentendo una maggiore parallelizzazione e un aumento del rendimento complessivo. Questo approccio risulta topologicamente inattuabile in una rete neurale che sfrutta l'elettronica digitale.
- Riduzione delle interferenze dovute al rumore esterno: i segnali ottici sono generalmente meno sensibili al rumore e alle interferenze esterne rispetto ai segnali elettronici. La luce può viaggiare su distanze più lunghe senza subire perdite significative o disturbi, rendendo le ONNs più robuste alle interferenze esterne e ai disturbi ambientali. Ciò contribuisce a migliorare l'affidabilità e la stabilità delle prestazioni del sistema.

- Integrazione ottica - elettronica: è possibile integrare facilmente l'ottica e l'elettronica nello stesso sistema, sfruttando le potenzialità di entrambe le tecnologie. Questo approccio permette di combinare la velocità e le basse perdite dell'ottica con la capacità di memorizzazione e la semplicità dell'elettronica digitale, consentendo di realizzare sistemi più efficienti e versatili.

Oltre ai vantaggi precedentemente discussi, la produzione e la ricerca attuale sono due aspetti di fondamentale importanza nello sviluppo delle tecnologie ottiche. Dal punto di vista produttivo, lo stato attuale dell'elettronica integrata offre un ambiente ottimale in cui sfruttare l'esperienza delle industrie specializzate nella creazione di circuiti integrati basati su CMOS. Questa sinergia permette di capitalizzare le competenze esistenti e di massimizzare le potenzialità dei sistemi ottici. Parallelamente, la fase attuale di esplorazione nella ricerca apre nuove opportunità per le aziende e i ricercatori di entrare nel settore in un momento strategico. Questo periodo di scoperta permette di tracciare nuove direzioni e di essere protagonisti nel campo delle tecnologie ottiche. Data la prospettiva promettente di sostituire una considerevole porzione del mercato dell'elettronica digitale impiegata per il calcolo, coloro che si distinguono in questa fase iniziale possono aspirare a diventare futuri leader di settore.

3 Revisione della letteratura

3.1 Introduzione

Le NN, un campo basato sulla teoria del ML, ha trovato una vasta gamma di applicazioni nelle quali rientrano la classificazione di immagini, il riconoscimento vocale, la traduzione del linguaggio, la presa di decisioni, la ricerca sul web, il filtraggio dei contenuti sulla rete sociale e le raccomandazioni su siti di e-commerce. Al fine di eseguire efficientemente i compiti di calcolo parallelo tipici delle NNs, sono stati sviluppati diversi approcci ed esperimenti. Le unità di elaborazione grafica (GPU) sono state identificate come particolarmente adatte e hanno svolto un ruolo significativo nel successo dell'apprendimento automatico nelle applicazioni reali. Inoltre, sono state sviluppate soluzioni specializzate come i circuiti su FPGA e i circuiti integrati specifici per applicazioni (ASIC), tra cui le unità di elaborazione tensoriale (TPU) di Google, il TrueNorth di IBM e l'Intel Nervana. Queste soluzioni elettroniche sono focalizzate sulla rappresentazione avanzata dei dati, con architetture di memoria ottimizzate per eseguire moltiplicazioni matriciali ad alta velocità e una larga banda di comunicazione esterna per consentire il parallelismo dei modelli e dei dati. La ricerca si è successivamente ed ulteriormente estesa anche ad architetture analogiche, di quale la principale soluzione sfrutta i memristors.

Al fine di superare le limitazioni dell'attuale architettura informatica per le NNs e migliorare la velocità di calcolo e l'efficienza energetica, sia l'ambito accademico che l'industria si stanno focalizzando sullo sviluppo di ulteriori architetture alternative. I primi studi sulle reti neurali ottiche, note come reti neurali fotoniche (PNN - Photonic Neural Networks) o ONNs, risalgono a oltre trent'anni fa, ma le prime implementazioni presentavano limitazioni legate all'ingombro, alla mancanza di scalabilità e alla mancanza di tecnologia adeguata. Tuttavia, dopo lo sviluppo di nuovi materiali e dell'industria ad essi legata, l'integrazione fotonica si è rivelata una tecnologia abilitante che ha permesso di superare alcune limitazioni delle prime soluzioni. L'implementazione ottica delle NNs sfrutta il notevole parallelismo offerto da gradi di libertà come la lunghezza d'onda della luce, la polarizzazione e la modalità di propagazione. Trasformazioni lineari possono essere eseguite utilizzando componenti ottici passivi, senza consumo di energia e con tempi di latenza minimi, mentre le non linearità ottiche possono essere sfruttate per implementare la funzione non lineare in ogni neurone. Questo suggerisce che l'implementazione ottica delle NNs potrebbe superare le soluzioni elettroniche in termini di velocità di calcolo ed efficienza energetica.

Nel riferimento [8] vengono presentate e classificate le principali architetture di PNNs studiate nell'ambito della ricerca. Gli autori del riferimento forniscono un riassunto delle ricerche condotte sulle ONNs, classificandole come segue:

- PNN senza memoria: questa categoria include le architetture di ONNs come le reti neurali multistrato e le reti neurali convoluzionali. Queste ONNs non presentano un componente di memoria a lungo termine, ma sono progettate per eseguire operazioni di elaborazione parallela su dati senza la necessità di una memoria interna.
- PNN con memoria: questa categoria comprende le reti neurali spiking e il reservoir computing. Le reti neurali spiking sono ONN che emulano l'attività dei neuroni biologici, utilizzando impulsi di segnale discreti, noti come "spike", per trasmettere informazioni. Il Reservoir Computing è un'architettura di rete neurale ottica che sfrutta un insieme di neuroni ricorrenti casualmente connessi, chiamato "reservoir", per eseguire operazioni di calcolo complesse.

L'immagine riassuntiva del riferimento, figura 3.1, mostra in modo sintetico le diverse architetture di ONNs descritte sopra, fornendo una panoramica visiva delle varie categorie e delle loro caratteristiche.

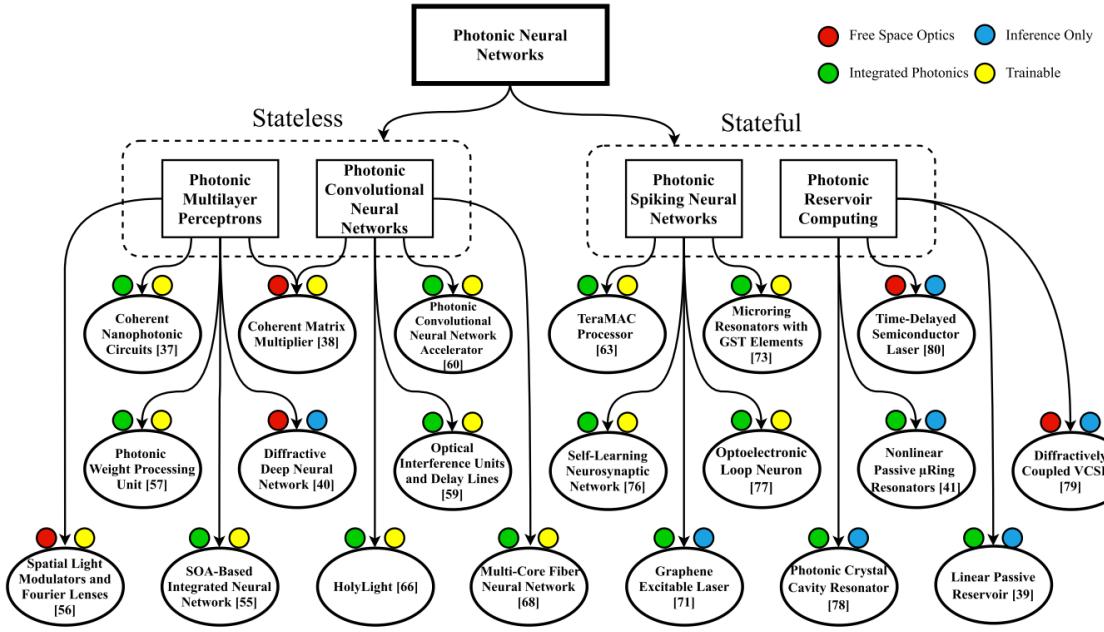


Figura 3.1: Classificazione delle diverse ONNs. Fonte [8]

La presente tesi si concentra sullo studio e l'analisi di tecnologie appartenenti alla categoria delle NNs multistrato con circuiti ottici integrati. In particolare, sono state prese in considerazione le seguenti soluzioni:

- Processore ottico basato su MZI: questa soluzione si basa sull'utilizzo di processori ottici a matrice che utilizzano interferometri a zona di Mach-Zehnder (MZI - Mach-Zehnder Interferometer) come elementi fondamentali per l'elaborazione ottica.
- Unità di elaborazione basata sull'interferenza multimodale (MMI - Multi mode interferometer): questa soluzione si basa sull'utilizzo di unità di elaborazione ottica che sfruttano il componente MMI per eseguire operazioni di calcolo.
- Un neurone completamente ottico con funzione di attivazione sigmoidea: questa soluzione riguarda lo sviluppo di un neurone ottico completo che implementa una funzione di attivazione sigmoidale. Questo neurone ottico sfrutta le proprietà non lineari dei componenti ottici per eseguire la trasformazione non lineare richiesta nell'elaborazione delle NNs.

Queste soluzioni tecnologiche sono state scelte per la loro rilevanza nel contesto dell'implementazione di ONNs multistrato e offrono benefici come l'elaborazione parallela, l'efficienza energetica e la capacità di eseguire operazioni non lineari. In seguito, nella tesi, saranno approfonditi gli aspetti teorici di ciascuna di queste soluzioni al fine di comprenderne il funzionamento. Lo studio e la ricerca condotti hanno permesso di selezionare e testare un esempio significativo tratto dalla letteratura, consentendo così un confronto tra un'implementazione ottica e una elettronica, e di trarre alcune conclusioni.

3.2 Processore ottico basato su MZI

3.2.1 Introduzione

L'articolo [16] presenta l'implementazione di una matrice di trasformazione lineare attraverso un insieme di MZI configurabili su chip. Lo studio in particolare si concentra sulla programmazione di un processore ottico 4x4 utilizzando una matrice di trasformazione unitaria implementata da una rete neurale a singolo strato. La matrice di trasformazione lineare viene ottenuta moltiplicando le matrici di trasformazione degli interferometri opportunamente calibrati al fine di ottimizzare la corretta classificazione di quattro classi. Infine, il processore ottico viene programmato sperimentalmente utilizzando i regolatori di fase corrispondenti. I risultati sperimentali mostrano un'accuratezza di

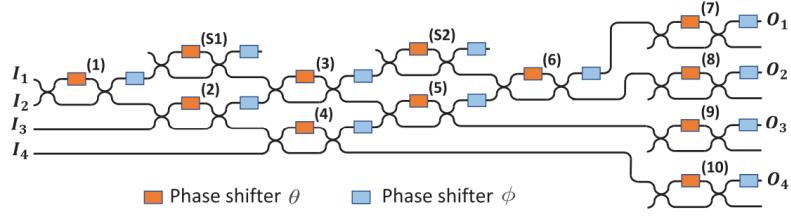


Figura 3.2: Struttura del processore ottico 4x4 basato su MZI. Fonte [16]

classificazione del 72%, inferiore alla simulazione su un computer digitale che ha raggiunto il 98,9%. La scelta del componente base viene motivata dagli autori sostenendo che le strutture lineari basate sull’interferometria ottica nelle ONNs sono di grande interesse per la loro velocità di calcolo e l’efficienza energetica.

3.2.2 Teoria

Nell’implementazione dell’articolo scientifico in analisi [16] gli autori utilizzano come parte elementare un MZI, esso è costituito da due accoppiatori direzionali e due sfasatori (PS - phase shifter) termici secondo l’immagine 3.2. Possiamo osservare che il PS interno permette di controllare l’intensità del segnale di uscita mentre il PS esterno controlla la differenza di fase tra le due uscite. Matematicamente questo si può descrivere come una trasformazione unitaria 2x2 come indicato nell’articolo:

$$\left[D_{M Z I}^{(n)} \right] = \begin{bmatrix} u_{11}^{(n)} & u_{12}^{(n)} \\ u_{21}^{(n)} & u_{22}^{(n)} \end{bmatrix} = j e^{j \frac{\theta^{(n)}}{2}} \begin{bmatrix} e^{j \phi^{(n)}} \sin \left(\frac{\theta^{(n)}}{2} \right) & e^{j \phi^{(n)}} \cos \left(\frac{\theta^{(n)}}{2} \right) \\ \cos \left(\frac{\theta^{(n)}}{2} \right) & -\sin \left(\frac{\theta^{(n)}}{2} \right) \end{bmatrix}$$

ove l’elemento $u_{pq}^{(n)}$ indica la trasmissione del MZI numero n tra l’entrata q e l’uscita p con $p, q \in \{1, 2\}$, mentre $\phi^{(n)}$ e $\theta^{(n)}$ sono le fasi introdotte rispettivamente dal PS esterno ed interno. Grazie a questa descrizione è possibile ottenere la matrice di trasformazione unitaria $[T_{SU(N)}]$ dei primi sei MZI in accordo alla numerazione data all’immagine 3.2. Infatti inserendo in modo opportuno tutti gli elementi $u_{pq}^{(n)}$ all’interno di sei differenti matrici 4x4 in base all’appartenenza e la posizione del MZI in considerazione e moltiplicando le sei matrici così ottenute possiamo scrivere la matrice complessiva come:

$$[T_{SU(N)}] = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{21} & u_{22} & u_{23} & u_{24} \\ u_{31} & u_{32} & u_{33} & u_{34} \\ u_{41} & u_{42} & u_{43} & u_{44} \end{bmatrix} = \left[D_{M Z I}^{(6)} \right]_{4 \times 4} \times \left[D_{M Z I}^{(5)} \right]_{4 \times 4} \times \cdots \times \left[D_{M Z I}^{(2)} \right]_{4 \times 4} \times \left[D_{M Z I}^{(1)} \right]_{4 \times 4}$$

Successivamente i ricercatori inseriscono nel design altri quattro MZI che matematicamente si possono descrivere come una matrice diagonale:

$$\begin{bmatrix} u_{11}^{(7)} & 0 & 0 & 0 \\ 0 & u_{11}^{(8)} & 0 & 0 \\ 0 & 0 & u_{11}^{(9)} & 0 \\ 0 & 0 & 0 & u_{11}^{(10)} \end{bmatrix}$$

Così facendo è possibile calcolare il risultato complessivo come moltiplicazione matriciale delle due matrici:

$$[W_{4 \times 4}] = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} = [\sum] \times [T_{SU(N)}] = \begin{bmatrix} u_{11}^{(7)} & 0 & 0 & 0 \\ 0 & u_{11}^{(8)} & 0 & 0 \\ 0 & 0 & u_{11}^{(9)} & 0 \\ 0 & 0 & 0 & u_{11}^{(10)} \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{21} & u_{22} & u_{23} & u_{24} \\ u_{31} & u_{32} & u_{33} & u_{34} \\ u_{41} & u_{42} & u_{43} & u_{44} \end{bmatrix}$$

Si ottiene così una matrice nella quale sono contenuti i pesi implementati dalla rete esprimibili

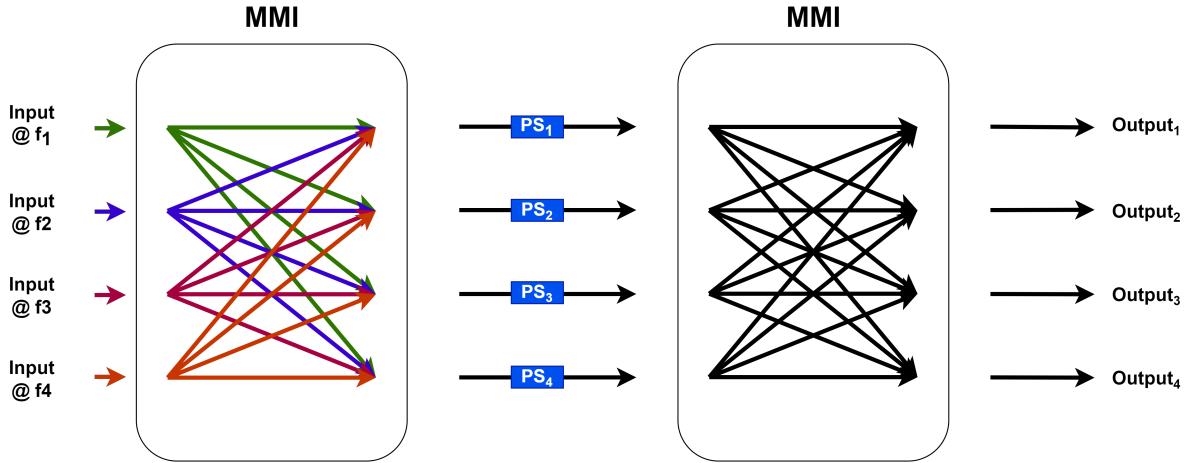


Figura 3.3: Topologia dell’ unità di elaborazione basata su MMI.

come prodotto e somme di ognuna delle fasi introdotte da ogni PSs di ogni MZI. Si può dunque riassumere il risultato di questo processore ottico come la moltiplicazione matriciale di ingresso per la matrice dei pesi:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

3.3 Unità di elaborazione basata sull’interferenza multimodale

3.3.1 Introduzione

Nel contesto dell’elaborazione ottica, l’implementazione di matrici di trasformazione lineare e l’integrazione su larga scala sono sfide cruciali da affrontare. Attualmente molti schemi di elaborazione ottica, come il precedente, non sono facilmente scalabili a causa dell’aumento quadratico del numero di elementi ottici con la dimensione della matrice computazionale. L’articolo [11] introduce un’interessante soluzione: un’unità di elaborazione convoluzionale ottica su chip, realizzata su una piattaforma in nitruro di silicio a bassa perdita. La scalabilità lineare del design proposto offre un solido potenziale per l’integrazione su larga scala, aprendo nuove prospettive nell’elaborazione ottica. Gli autori dimostrano la capacità e le caratteristiche dell’architettura tramite l’implementazione di operazioni convoluzionali parallele tramite kernel correlati. Nonostante questa correlazione dei kernel, viene dimostrata sperimentalmente la classificazione in dieci classi di cifre scritte a mano dal database MNIST.

3.3.2 Teoria

La topologia proposta si compone di due interferometri MMI 4x4 (quattro ingressi - quattro uscite) e quattro PSs che connettono gli ingressi del secondo MMI con l’uscita del primo come mostrato in figura 3.3. Il design descritto prevede di utilizzare quattro lunghezze d’onda, una per ogni ingresso così da poter avere un numero di segnali indipendenti pari al numero di ingressi. Il numero di uscite indica invece il numero di kernel implementabili. Il design studiato dagli autori si completa poi con la gestione degli ingressi e dei ritardi di questi: ogni dato in ingresso è modulato in egual modo su quattro frequenze diverse che poi, opportunamente ritardato, risulta essere l’ingresso del primo MMI a una frequenza diversa per ogni ingresso. Matematicamente il sistema può essere descritto come una moltiplicazione matriciale tra una sequenza di matrici che definiscono il singolo componente.

MMI:

$$[M_{4 \times 4}] = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

ove l'elemento m_{uv} con $u, v \in \{1, 2, 3, 4\}$ indica la risposta dell'ingresso v all'uscita u .

PS:

$$[\phi_{4 \times 4}] = \begin{bmatrix} e^{j\phi_{11}} & e^{j\phi_{12}} & e^{j\phi_{13}} & e^{j\phi_{14}} \\ e^{j\phi_{21}} & e^{j\phi_{22}} & e^{j\phi_{23}} & e^{j\phi_{24}} \\ e^{j\phi_{31}} & e^{j\phi_{32}} & e^{j\phi_{33}} & e^{j\phi_{34}} \\ e^{j\phi_{41}} & e^{j\phi_{42}} & e^{j\phi_{43}} & e^{j\phi_{44}} \end{bmatrix}$$

ove, in maniera analoga, $e^{j\phi_{kz}}$ esprime la fase introdotta dal PS del canale k alle diverse frequenze z . Nel caso specifico dell'implementazione trattata dagli autori, essendo presente un solo PS per ogni canale, si può semplificare la matrice a un gruppo minore di fasi:

$$[\phi_{4 \times 4}] = \begin{bmatrix} e^{j\phi_1} & e^{j\phi_1} & e^{j\phi_1} & e^{j\phi_1} \\ e^{j\phi_2} & e^{j\phi_2} & e^{j\phi_2} & e^{j\phi_2} \\ e^{j\phi_3} & e^{j\phi_3} & e^{j\phi_3} & e^{j\phi_3} \\ e^{j\phi_4} & e^{j\phi_4} & e^{j\phi_4} & e^{j\phi_4} \end{bmatrix}$$

Il sistema si può quindi vedere come:

$$[W_{4 \times 4}] = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} = [M_{4 \times 4}] \times ([M_{4 \times 4}] \odot [\phi_{4 \times 4}])$$

ove il simbolo \odot è il prodotto di Hadamard, ossia il prodotto di due matrici con uguali dimensioni di cui il risultato è la moltiplicazione tra gli elementi delle due matrice in posizione uguale.

3.4 Un neurone completamente ottico con funzione di attivazione sigmoidea

3.4.1 Introduzione

Nell'articolo [12] viene presentato un tipo di neurone ottico che implementa in ottica la somma pesata di un segnale ottico WDM e la funzione di attivazione sigmoide. Gli autori sfruttano un modulo di soglia ottica innovativo, recentemente sviluppato, e un sistema di ponderazione codificato in lunghezza d'onda. La struttura, ad alto livello, si può descrivere come un classico neurone digitale: l'ingresso del neurone ottico composto da quattro ingressi ottici a frequenza diversa sfruttando la tecnica WDM viene opportunamente ponderato, sommato e successivamente elaborato dalla struttura incaricata di realizzare la funzione di attivazione. La struttura non lineare è realizzata attraverso l'utilizzo di un dispositivo basato su interferometri a zona di Mach-Zehnder uniti ad amplificatori ottici a semiconduttori (SOA-MZI, semiconductor optical amplifier Mach-Zehnder interferometer), seguito da un altro dispositivo SOA che funge da convertitore di modulazione a guadagno incrociato e convertitore in lunghezza d'onda (XGM-WC, Cross-gain Modulation-Wavelength Converter).

3.4.2 Teoria

Il sistema proposto utilizza la tecnica WDM: ogni segnale di ingresso $[X_1, \dots, X_n]$ viene modulato su una lunghezza d'onda dedicata $[\lambda_1, \dots, \lambda_n]$ utilizzando la modulazione in ampiezza dei segnali. L'ingresso del neurone viene poi immesso nel corrispettivo peso implementato da uno degli attenuatori ottici variabili (VOA - variable optical attenuators), i quali, uno per ogni frequenza, creano il vettore dei pesi $W = [W_1, \dots, W_n]$. Gli ingressi pesati vengono poi convogliati in un singolo ramo ottico, dove il livello complessivo di potenza del segnale risultante rappresenta la somma ponderata dei livelli di potenza dei segnali ottici di ingresso. L'informazione ottica viene poi elaborata dalla funzione di attivazione ottica, completando così il neurone ottico.

Il dispositivo di attivazione sigmoidale fotonica è composto da un interferometro SOA-MZI e un SOA. L'interferometro SOA-MZI è un componente costituito da un MZI in cui vengono inseriti due SOAs nei due bracci interni. Inoltre, sono presenti quattro accoppiatori direzionali, due per ogni braccio interno, che consentono l'inserimento di segnali aggiuntivi in entrambe le direzioni all'interno dei due SOAs. Il

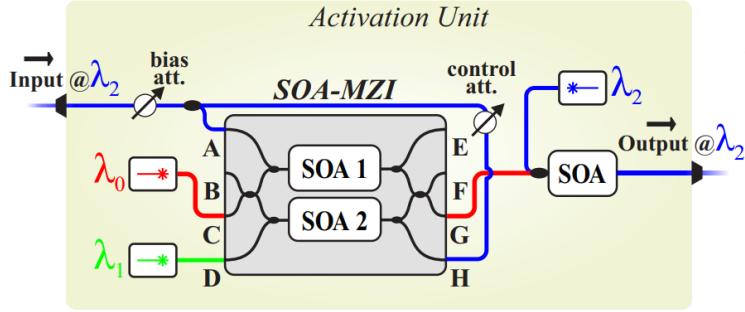


Figura 3.4: La figura mostra lo schema della funzione sigmoidea ottica. Fonte [12]

SOA esterno, invece, funge da convertitore di lunghezza d'onda, consentendo così di ottenere un uscita a una frequenza specifica e costante. Lo schema 3.4 mostra in modo chiaro la topologia implementata. Il sistema, che implementa la funzione sigmoide, è poi completato da una sorgente ad onda continua (CW - continuous wave) di lunghezza λ_{00} che viene indirizzata all'ingresso "C" del SOA-MZI, mentre un'altra sorgente ad CW a λ_{01} viene inserita nel SOA₂ attraverso il braccio di controllo "D". Con questa configurazione e con entrambi i segnali CW ad alti livelli di potenza ottica i due SOA, SOA₁ e SOA₂, sono costretti ad operare nel loro regime di profonda saturazione realizzando uno schema a guadagno differenziale. Viene poi previsto la divisione del segnale in ingresso in due parti così da creare un segnale di controllo, in contro-propagazione rispetto al vettore in ingresso, attenuato da uno specifico VOA ottenendo così il corretto bias della funzione di attivazione. L'ingresso viene, come mostrato in figura 3.4, diviso in due flussi identici prima di essere inviato alle porte "A" e "H" dei rami del SOA-MZI come fasci di controllo co- e contro-propaganti. Grazie a questa implementazione, all'uscita "G" del SOA-MZI si ottiene una copia invertita del segnale di controllo a lunghezza d'onda λ_0 . Questa copia invertita viene successivamente iniettata come segnale di controllo nel successivo SOA, che funziona come un XGM-WC. Questo SOA ripristina sia la lunghezza d'onda che la logica del segnale iniziale utilizzando un ulteriore fascio ottico CW a λ_2 come segnale di ingresso.

4 Simulazione e valutazione di una rete neurale optoelettronica

4.1 Introduzione

In questo capitolo è presente la maggior parte del lavoro svolto, esso consiste nella simulazione di diverse implementazioni di componenti al fine di realizzare una rete neurale ottica, supportate da codice complementare che agevola il processo. Questo processo avviene utilizzando la libreria "Photontorch" [9]. La simulazione è stata eseguita in ambiente Python, sfruttando la libreria già citata basata su "Torch" [13], che consente la simulazione di circuiti ottici-elettronici attraverso una descrizione del circuito. La libreria si basa sulla programmazione ad oggetti, in cui i componenti fondamentali del circuito sono rappresentati da elementi come guide d'onda, specchi, accoppiatori, e altri ancora. Unendo e inizializzando tali componenti all'interno di una classe figlia della classe "photontorch.Network", è possibile definire il circuito desiderato e gestire le connessioni tra i vari componenti. La programmazione poi si completa instanziando un elemento "circuito" e l'ambiente di simulazione con i parametri voluti.

4.2 Caratteristiche vantaggiose

Un aspetto importante di questa libreria è la possibilità di implementare i circuiti in modo gerarchico, consentendo la creazione di un circuito di base che può essere inizializzato più volte all'interno di un circuito più complesso, semplificando così la creazione complessiva. Inoltre, è possibile creare componenti personalizzati definendone il comportamento e le funzionalità, superando così i limiti imposti dai componenti predefiniti nella libreria. Ulteriori aspetti positivi sono: la possibilità di simulare il circuito nel tempo ed in frequenza con tempistiche e precisione scelta dall'utente e la possibilità di utilizzare o creare dei componenti con dei parametri, i quali si possono essere successivamente ottimizzati in base al risultato voluto della rete implementata.

4.3 Caratteristiche svantaggiose

Un limite significativo della libreria allo stato attuale è l'incapacità di simulare fenomeni multifrequenziali come la dispersione cromatica nelle fibre ottiche: sebbene sia possibile implementare diverse sorgenti a diverse lunghezze d'onda nel circuito, durante la simulazione i fenomeni a ciascuna frequenza vengono calcolati in modo indipendente dagli altri. Un altro limite, facilmente superabile, è la mancata implementazione di elementi elettronici e elettro-ottici come sorgenti di corrente a frequenze diverse, amplificatori elettronici, fotodiodi o laser. Tuttavia, è possibile creare tali elementi generando nuovi componenti e trattando la corrente come se fosse un segnale ottico, facendo attenzione a distinguere la luce dalla corrente.

4.4 Simulazione e implementazione

Grazie a questa libreria, è stato possibile simulare e ottimizzare diversi elementi a livello di simulazione, tra cui la moltiplicazione tra un ingresso e una matrice utilizzando MZI, uno strato di una NN con 4 ingressi e 4 uscite con una matrice MZI per la parte lineare e un doppio SOA per la parte non lineare. Infatti il lavoro di questa tesi è stato in parte riuscire a simulare, ad usare ed integrare principalmente due componenti ottici, MZI e SOA, al fine di poter ottenere le parti necessarie per implementare ed addestrare uno strato di una ONN simulata.

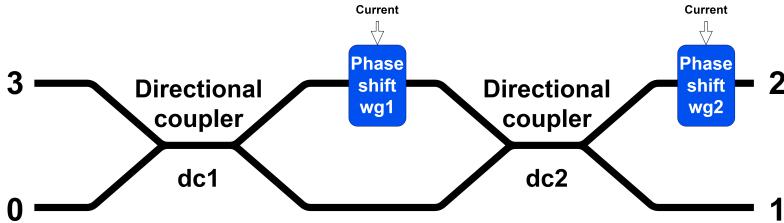


Figura 4.1: E' rappresentata la struttura di un MZI.

4.4.1 MZI

Il componente MZI è stato completamente riprogettato a causa dei risultati non soddisfacenti ottenuti durante alcuni test. La nuova implementazione del MZI prevede l'utilizzo di due accoppiatori direzionali, due dispositivi per la modulazione di fase e le linee guida necessarie per il collegamento dei componenti.

```

1 class MZI(pt.Network):
2     def __init__(self, kc1=.5,kc2=.5,
3                  wga=pt.Waveguide(trainable=True, phase=p.random.random()*2*np.pi),
4                  wgb=pt.Waveguide(trainable=True, phase=p.random.random()*2*np.pi)):
5         super(MZI, self).__init__()
6         self.wg1 = wga
7         self.wg2 = wgb
8         self.wg3 = pt.Waveguide(trainable=False)
9         self.wg4 = pt.Waveguide(trainable=False)
10        self.dc1 = pt.DirectionalCoupler(coupling=kc1, trainable=False)
11        self.dc2 = pt.DirectionalCoupler(coupling=kc2, trainable=False)
12        self.link(3,"3:dc1:2", "0:wg1:1", "3:dc2:2", "0:wg2:1", 2)
13        self.link(0,"0:dc1:1", "0:wg3:1", "0:dc2:1", "0:wg4:1", 1)

```

Come si può dedurre dalla sezione di codice, è stata definita una nuova classe, derivata da photon-torch.Network, e la sua funzione di inizializzazione. In questa funzione, vengono inizialmente aggiunti e inizializzati i vari componenti da creare al fine di creare il componente voluto. È importante notare che i PSs vengono simulati utilizzando una semplice sezione di guida d'onda a cui è possibile aggiungere o togliere una fase che può variare durante il training. Gli altri elementi sono inizializzati con parametri di default e statici, quindi non vengono considerati come "pesi" durante il training dell'intera rete. Le ultime due righe invece descrivono le connessioni tra i vari elementi interni alla classe e le porte di entrata e uscita.

4.4.2 Moltiplicazione di matrici MZI

Nella parte lineare della rete, vengono inizializzati dieci MZI collegati come mostrato nello schema di figura 3.2, con la possibilità di aggiungerne altri due per future espansioni. Come già detto nella sezione 3.2, gli MZI implementano una moltiplicazione matriciale che permette di moltiplicare gli ingressi per valori dipendenti dalla fase di ciascun MZI. Il circuito può essere espanso per creare una matrice di dimensioni NxN, con la possibilità di automatizzare la generazione del codice per la creazione della classe. Il codice qui riportato consente di generare un file contenente la definizione di un circuito con un numero N impostabile di ingressi e uscite che implementa la moltiplicazione per una matrice.

```

1 ##### parte uno #####
2 N_neurons = 5
3 with open("Matrix{}x{}.txt".format(N_neurons,N_neurons),"w") as file :
4     file.write("class Circuit(pt.Network): \n\tdef __init__(self):\n\t\tsuper().__init__()")
5
6
7 ##### parte due #####
8     for i in range(N_neurons):

```

```

9         file.write("\t\tsel.src{} = pt.Source()\n".format(i+1))
10
11    for i in range(N_neurons) :
12        file.write("\t\tsel.det{} = pt.Detector()\n".format(i+1))
13
14    for i in range(N_neurons) :
15        file.write("\t\tsel.out{} = MZI(wga = pt.Waveguide(trainable=True,
16            phase= np.random.random()*2*np.pi),
17            wgb = pt.Waveguide(trainable=True,
18            phase= np.random.random()*2*np.pi))\n".format(i+1))
19
20    j = 0
21    for i in range(N_neurons):
22        j = j + i
23
24    for i in range(j):
25        file.write("\t\tsel.mzi{} = MZI(wga = pt.Waveguide(trainable=True,
26            phase= np.random.random()*2*np.pi), wgb = pt.Waveguide(trainable=True,
27            phase= np.random.random()*2*np.pi))\n".format(i+1))
28
29 ##### parte tre #####
30 mziN = 1
31 last_prev_layer = 0
32 last_previous_layer = 0
33 for i in range(N_neurons-1):
34     file.write("\t\tsel.link(\"src{}:0\", ".format(i+1))
35     for k in range(i+1+i):
36         if(k % 2 == 0):
37             file.write("\t\"0:mzi{}:1\", ".format(mziN))
38             mziN = mziN + 1
39         else:
40             file.write("\t\"3:mzi{}:2\", ".format(
41                 mziN - (last_previous_layer - last_prev_layer )-1))
42         last_prev_layer = last_previous_layer
43         last_previous_layer = mziN - 1
44     file.write("\\"3:out{}:2\", \\"0:det{}\"\n".format(i+1,i+1))
45 file.write("\t\tsel.link(\"src{}:0\", ".format(N_neurons))
46 for i in range(last_prev_layer,last_previous_layer):
47     file.write("\\"3:mzi{}:2\", ".format(i+1))
48 file.write("\\"0:out{}:1\", \\"0:det{}\"\n".format(N_neurons-1,N_neurons))

```

Il codice è molto semplice e si può dividere in tre fasi: la prima in cui si scrive sul file di destinazione tutta la parte iniziale e costante della definizione della classe, una seconda fase in cui si scrive l'inizializzazione dei componenti e l'ultima in cui si scrive la definizione delle connessioni tra i componenti. È importante notare i seguenti aspetti per usufruire al meglio di tale codice:

- È necessario definire in anticipo una classe chiamata "MZI" come menzionato in precedenza
- gli ultimi due output, corrispondenti ai rivelatori "N_neurons" e "N_neurons-1", sono invertiti rispetto all'ordine cardinale.
- i risultati vengono salvati su un file "MatrixNxN.txt", dove "N" rappresenta il numero di neuroni configurati per la rete.

4.4.3 Doppio SOA

La parte non lineare del circuito è stata implementata utilizzando un doppio "SOA". Questo componente è incluso nella libreria standard di photontorch e si basa sul modello descritto in [7]. Nel contesto di questo modello ad alta non linearità, il doppio SOA è stato sfruttato per due caratteristiche fondamentali, utilizzate per creare una funzione che ricorda la funzione sigmoide, ma adattata alle esigenze della rete considerata. Il primo SOA è stato utilizzato per sfruttare la non linearità legata alle perdite di attivazione, introducendo una grande attenuazione nei segnali al di sotto di una soglia specifica. Questo ha permesso di creare una transizione netta tra la regione di "off" (spento)

e la regione di "on" (acceso) per i segnali al di sopra della soglia. Il secondo SOA è stato utilizzato in saturazione, in modo da ottenere una risposta, dopo una soglia, lineare o quasi lineare rispetto al valore di ingresso a differenza del normale comportamento esponenziale in regimi standard. Questo ha permesso di ottenere un'uscita che riflette in modo coerente l'ingresso, andando così a limitare i valori massimi ottenibili in uscita. Sono riportati di seguito in figura 4.2 i grafici che illustrano la relazione tra l'ingresso e l'uscita dei due SOA. Il primo grafico mostra il comportamento del SOA nel caso di "off" (spento) e "on" (acceso), evidenziando l'effetto di attenuazione al di sotto della soglia. Il secondo grafico rappresenta la risposta in saturazione del secondo SOA, evidenziando una relazione quasi lineare tra l'ingresso e l'uscita dopo il guadagno iniziale.

L'unione in cascata dei due componenti, con l'output del primo collegato all'ingresso del secondo,

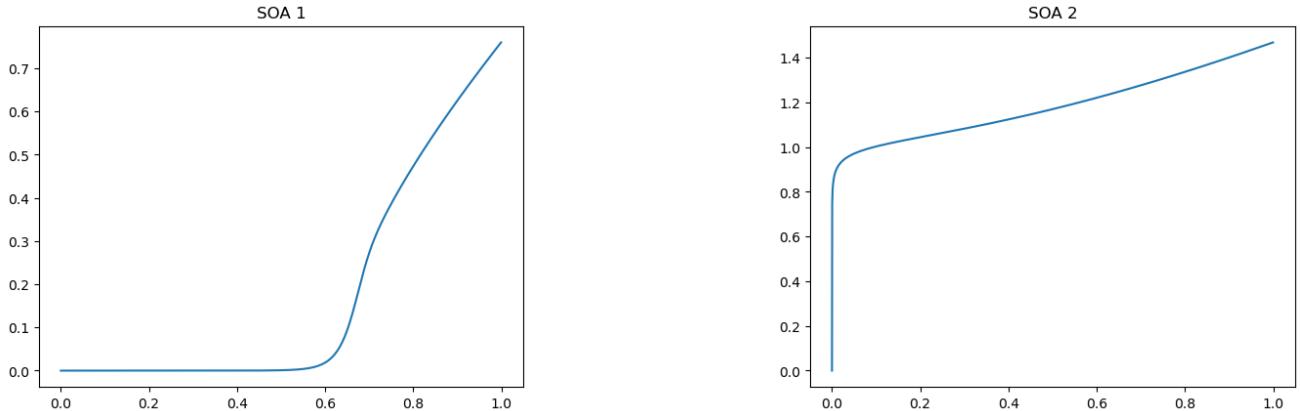


Figura 4.2: Le figure mostrano l'andamento della relazione tra ingressi e uscite dei due differenti SOA singolarmente simulati.

produce un comportamento altamente non lineare desiderato. Questa configurazione permette di ridurre l'ampiezza dei segnali al di sotto di una soglia specifica, mantenendo un rapporto costante per i segnali al di sopra di tale soglia. Nell'intervallo che collega i due regimi di funzionamento, si osserva un aumento del guadagno, contribuendo così ad amplificare i segnali di piccola ampiezza intorno alla soglia.

È importante sottolineare che, l'esistenza di una componente non lineare attiva, introduce potenza all'interno della rete e ciò è di vitale importanza. Senza questa componente, se la rete fosse implementata solo con elementi passivi, si avrebbe una degradazione e una perdita del segnale, che potrebbe essere confuso o indistinguibile dal rumore presente in un'implementazione fisica della rete neurale.

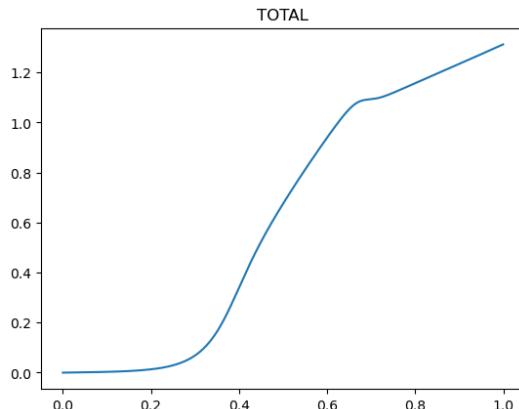


Figura 4.3: La figura mostra il grafico della relazione tra ingressi e uscite del sistema complessivo dei due SOA.

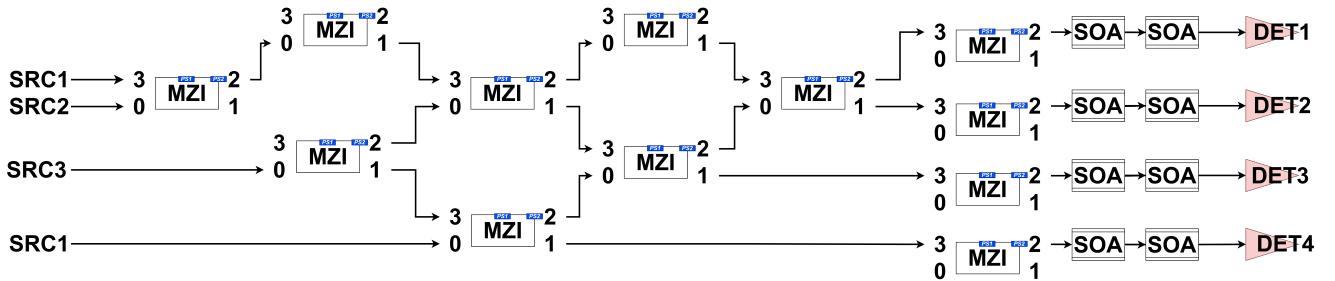


Figura 4.4: La figura mostra lo schema della rete neurale ottica simulata.

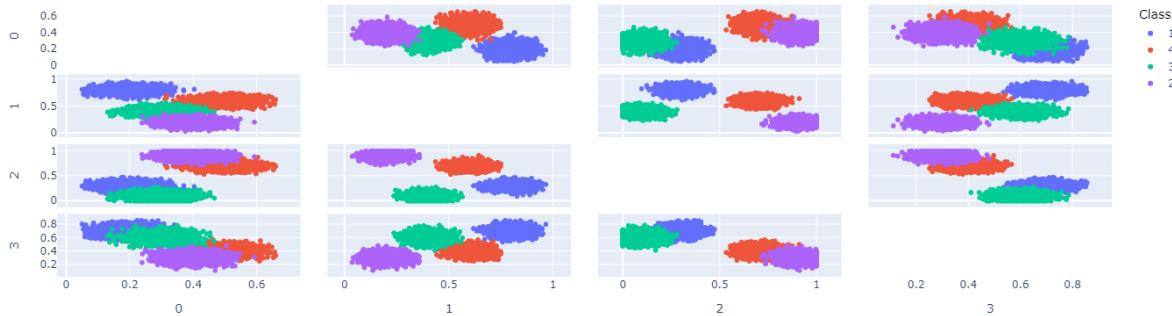


Figura 4.5: Grafico a dispersione dei dati generati artificialmente.

4.4.4 Singolo strato

Dopo aver caratterizzato e fatto funzionare il doppio SOA e il complesso di MZI, è possibile utilizzarli per creare una varietà di NNs. In questa tesi è presentata un'implementazione di una rete neurale con un strato di 4 ingressi e 4 uscite, utilizzando il design descritto in precedenza per la parte lineare e aggiungendo la parte non lineare fornita dai SOAs in cascata. Per strutturare il circuito, ho creato diverse classi sfruttando la gerarchia tra di loro: la classe figlia è chiamata "circuito" e al suo interno sono inizializzate la parte lineare e non lineare, insieme alle sorgenti e ai rivelatori. La parte lineare è denominata "Layer4x4" e rappresenta il design ampiamente descritto in precedenza, mentre la classe "ActivationFunction" funge da funzione di attivazione, sfruttando il sistema dei due SOA descritto in precedenza. Il design risultante è illustrato nella figura 4.4, mentre nel codice completo fornito in appendice A è possibile osservare l'implementazione dettagliata.

4.4.5 Set di dati

Per l'addestramento e la sperimentazione, è stato utilizzato un set di dati generato artificialmente per adattarsi alle esigenze e alla semplicità del design della ANN. I dati non sono reali e sono stati generati utilizzando una distribuzione gaussiana in uno spazio 4D, con diverse medie in base alla classe di appartenenza, in cui ogni dimensione varia nell'intervallo compreso tra 0 e 1. In figura 4.5 è possibile vedere i dati organizzati in un grafico a dispersione (scatter matrix). Complessivamente, sono state considerate quattro classi, consentendo l'adozione di una codifica one-hot per l'uscita. Questa codifica assegna il valore 1 alla posizione ordinale corrispondente alla classe di appartenenza, mentre tutti gli altri valori sono impostati a 0. Se analizziamo il vettore $[0, 0, 1, 0]$ osserviamo quindi che la classe di appartenenza è la terza. Al fine di semplificare l'interpretazione dei dati, è stata ulteriormente adottata un'etichettatura binaria, in cui l'output assume il valore 1 nella posizione con il valore più

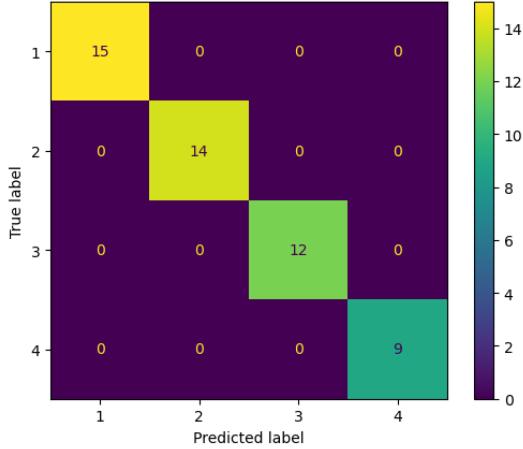


Figura 4.6: Matrice di confusione della rete neurale a singolo strato implementata.

elevato tra i valori del vettore in uscita e 0 negli altri casi.

$$class = \begin{cases} 1, & \text{if } argmax(Output) = 1 \\ 2, & \text{if } argmax(Output) = 2 \\ 3, & \text{if } argmax(Output) = 3 \\ 4, & \text{if } argmax(Output) = 4 \end{cases} \quad (4.1)$$

Questa scelta è stata motivata principalmente dal fatto che l'obiettivo della ricerca non era quello di implementare una rete neurale concreta, ma di verificare l'effettiva funzionalità e le prestazioni del design scelto.

4.4.6 Risultati e considerazioni

La rete neurale funziona in modo impeccabile, raggiungendo un'accuratezza del 100% come evidenziato dalla matrice di confusione associata visibile in figura 4.6. La latenza e il throughput della rete possono invece essere dedotti dai grafici temporali delle simulazioni. È fondamentale sottolineare che il tempo di esecuzione non dipende né dal numero di input né dalla potenza del segnale, bensì dalla dinamica fisica della luce che si propaga all'interno del sistema. Nei quattro casi riportati di seguito in figura 4.7, uno per ogni classe, si può chiaramente osservare che il risultato rimane stabile dopo un intervallo di tempo di 300 ps . Ciò significa che è possibile inserire un nuovo ingresso ogni 300 ps , corrispondenti a una frequenza di $\frac{1}{300\text{ ps}} = 3,33\text{ GHz}$. Analogamente, la latenza complessiva del design può essere ottenuta moltiplicando il tempo per il numero di strati, che in questo caso ammonta a $1\text{ strato} \times 300\text{ ps} = 300\text{ ps}$. Inoltre, considerando il riferimento [16], è possibile effettuare una stima dei possibili consumi energetici. Gli autori riportano che per il corretto funzionamento della loro rete neurale è necessario un consumo di circa 609 mW. Tuttavia, è importante considerare che a questo consumo vanno aggiunti il consumo della parte non lineare, il consumo delle sorgenti di segnale e eventualmente la potenza necessaria a i componenti elettrici per amplificare i segnali in uscita. Il consumo dei SOA, in accordo ai datasheet riportati dall'azienda del riferimento [2], si può stimare a circa 5W considerato che ogni SOA accetta un valore massimo di 1.330W, mentre il consumo delle sorgenti possiamo stimarle singolarmente come 0.8W e complessivamente come 3W in accordo con i datasheet forniti dall'azienda Innolume [3]. Nel complesso, il consumo si potrebbe riassumere in circa 8.610 W solamente per la parte ottica escludendo i controlli necessari per generare gli ingressi in modulazione di ampiezza e l'effettiva acquisizione degli output. Alcune considerazioni necessarie da fare sono a riguardo all'effettiva possibilità di raggiungere i valori precedentemente considerati:

- l'accuratezza è sicuramente molto ottimistica in quanto nella simulazione mancano elementi di rumore dovuto alla natura della luce o a elementi di errore dovuti a difetti nella rete introdotti nella fabbricazione. Questi fattori possono influire sull'accuratezza del modello e richiedere ulteriori tecniche di pre-elaborazione dei dati o regolarizzazione per gestire il rumore.

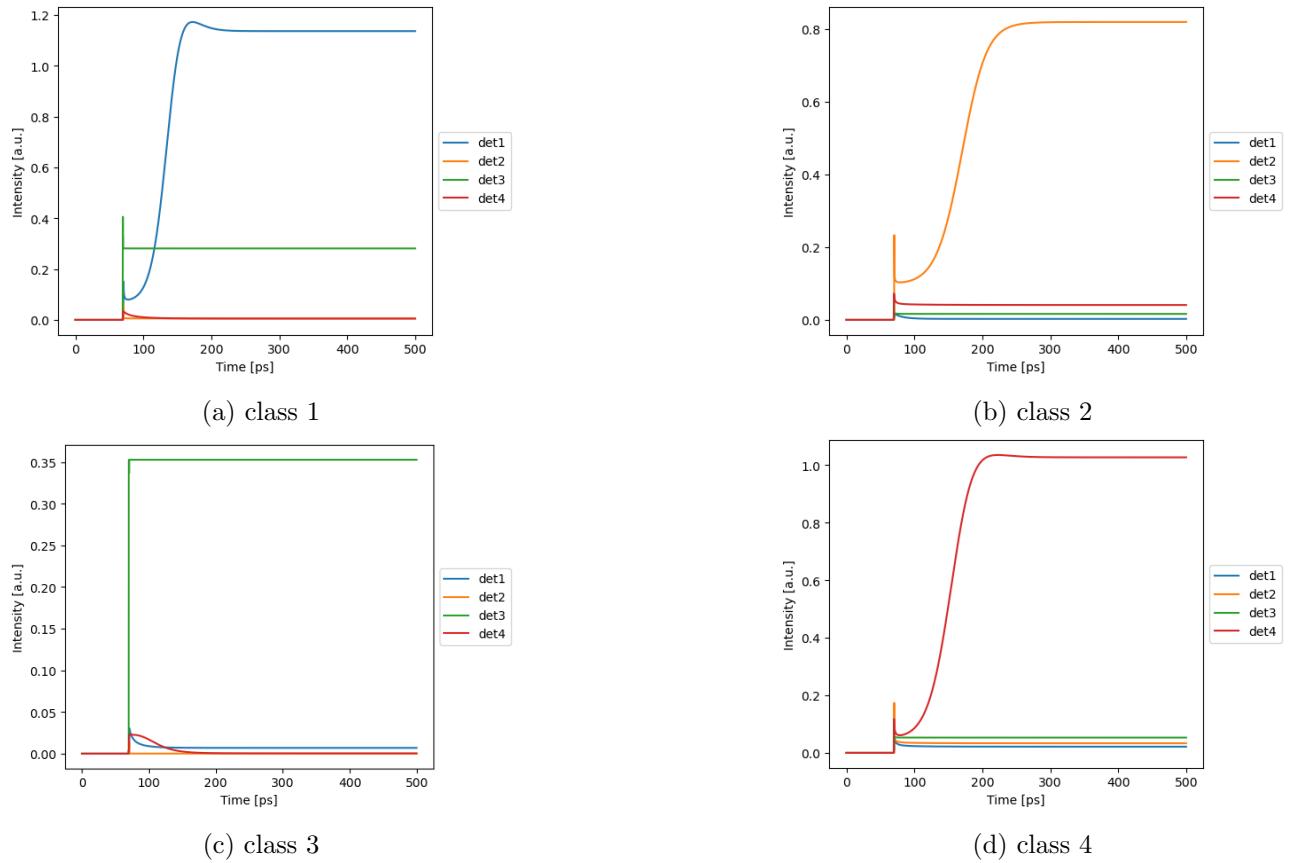


Figura 4.7: Simulazione temporale per quattro diversi ingressi.

- Il throughput e la latenza non considerano in nessun modo i difetti reali dell'architettura come un variazione di indice di rifrazione, effetti di spread del segnale nel tempo e in frequenza, la capacità di generare e percepire segnali ad alta velocità. Un altro fattore determinante è la topologia dell'architettura stessa, poiché la quantità di ingressi e uscite ha un impatto significativo sulle tempistiche. In particolare, all'aumentare del numero di ingressi e uscite di ciascun strato utilizzato, la lunghezza del percorso massimo aumenterà linearmente o più rapidamente, ritardando l'emissione di un segnale stabile e influenzando così la latenza e il throughput del sistema.

Riassumendo i valori importanti sono riportati in tabella:

Accuratezza	Latenza	Throughput	Potenza
100%	300 ps	3,33 Ghz	8.610 W

5 Implementazione FPGA

5.1 Introduzione

L'implementazione riportata di seguito rappresenta una rete neurale con due strati completamente connessi e la funzione di attivazione ReLU, ciascuno composto da quattro ingressi e quattro uscite. Tale implementazione è stata presentata al fine di consentire un confronto diretto tra l'approccio basato sull'elettronica digitale e l'implementazione ottica. È importante sottolineare che entrambe le topologie non sono state ottimizzate per una reale implementazione pratica e mancano di scalabilità. Pertanto, il confronto rappresenta un primo confronto teorico tra le diverse tecnologie. Questo approccio è stato adottato a causa delle limitazioni di risorse, sia in termini di tempo che di disponibilità economica, che hanno impedito un'implementazione effettiva delle tecnologie all'avanguardia.

5.2 Schema

Il sistema è stato implementato utilizzando una scheda "Digilent Nexys 4 DDR". Come si può osservare dalla immagine 5.1, il sistema è composto da un processore e una parte logica. Il processore consente di caricare gli ingressi e leggere le uscite attraverso un'interfaccia AXI, un particolare protocollo che permette al processore di accedere ai valori della rete neurale nella parte logica (PL) come fossero registri. Grazie a specifici indirizzi di memoria indicati nel software di sviluppo, è possibile leggere e scrivere questi valori normalmente. La PL implementa una rete neurale a due strati completamente connessi, utilizzando la funzione di attivazione ReLU come si può osservare in foto 5.7. Il sistema è completato dall'aggiunta di un display a 7 segmenti, tramite il quale è possibile visualizzare uno degli ultimi quattro valori in uscita generati dalla rete in base alla selezione fatta dai pulsanti "Up" (BTNU) e "Down" (BTND). La rete neurale è stata implementata utilizzando una combinazione di blocchi predefiniti forniti da Vivado (software usato per la programmazione di fpga) e blocchi personalizzati. Per l'implementazione, sono stati utilizzati blocchi AXI GPIO di Vivado come ingresso e uscita dal processore, moltiplicatori e sommatori float32, FIFO AXI e blocchi generatori di costanti. Oltre questi blocchi già disponibili in Vivado, è stato sviluppato un blocco per la duplicazione degli stream AXI utilizzati nella creazione dei segnali interni alla rete e un blocco che implementa la funzione di attivazione ReLU. Il design della rete neurale e il suo funzionamento sono molto semplici: i valori degli ingressi vengono impostati dal processore, successivamente viene attivato un segnale di controllo che avvia la propagazione dei segnali all'interno della rete durante la quale vengono svolti i calcoli e permettendo così di ottenere i corretti valori in uscita. Ogni segnale, come si osserva in figura 5.2, attraversa complessivamente due neuroni per poi essere mostrato a schermo e inserito in una coda FIFO alla quale si accedere dal processore tramite il protocollo AXI. Ogni neurone è implementato secondo il seguente schema: i segnali di ingresso vengono inviati tramite AXI stream al moltiplicatore corrispondente, che moltiplica l'ingresso per un valore costante definito durante la fase di progettazione. I valori in uscita dal moltiplicatore vengono gestiti sempre come AXI stream e sommati in modo appropriato tramite tre blocchi dedicati, per essere poi elaborati dal blocco "ReLU". Questo blocco calcola, senza memorizzare informazioni, il risultato della funzione matematica ReLU per ciascun ingresso e successivamente trasmette il valore alla FIFO. La figura 5.3 mostra un riscontro grafico di quello descritto fino ad ora illustando il diagram block di un singolo neurone artificiale.

5.3 Stato dell'arte

L'analisi del funzionamento e del design evidenzia chiaramente che le prestazioni del sistema dipendono principalmente da due fattori limitanti: il moltiplicatore float32 e il processore utilizzato. Questi costituiscono i due punti critici poiché il processore utilizzato nella scheda non è in grado di scrivere quattro valori ad ogni ciclo di clock, in quanto opera alla stessa velocità della parte logica. Inoltre, il

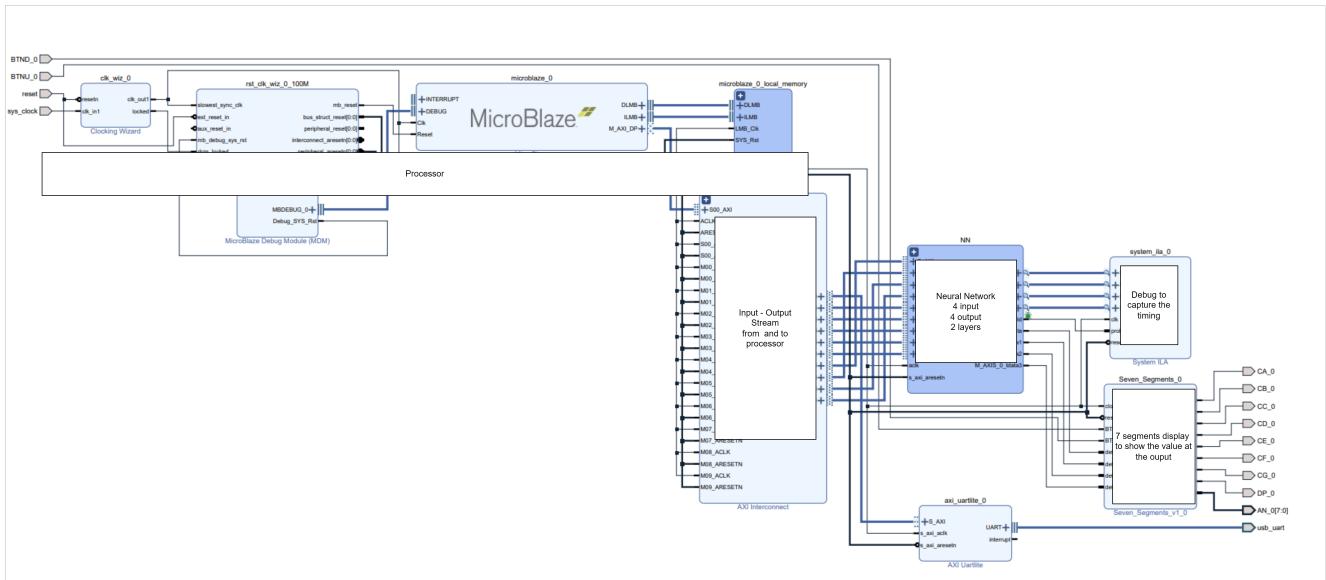


Figura 5.1: Schema a blocchi dell'implementazione della NN su FPGA.

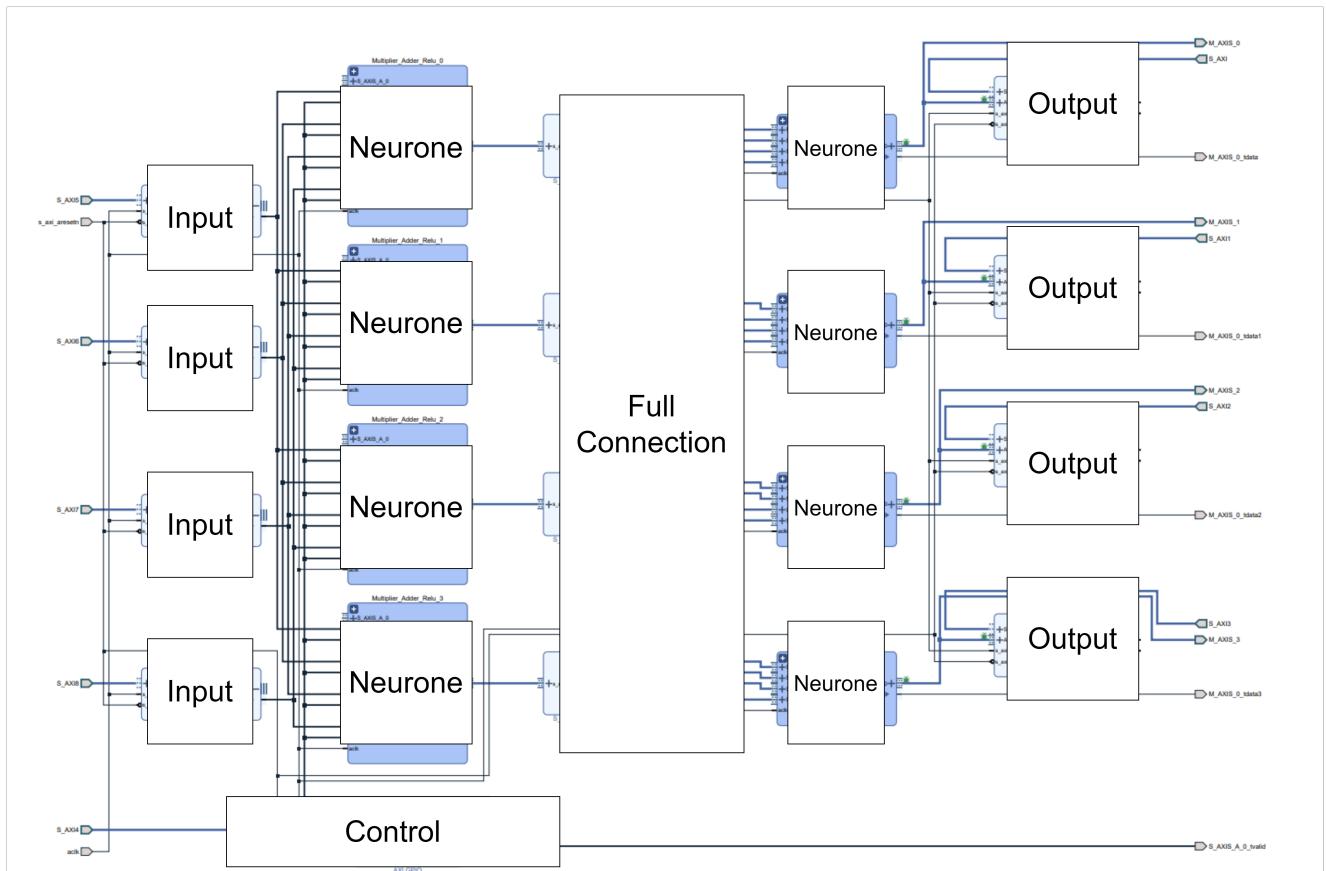


Figura 5.2: Schema a blocchi della NN, in particolare la topologia dei singoli strati e come i percettroni sono organizzati.

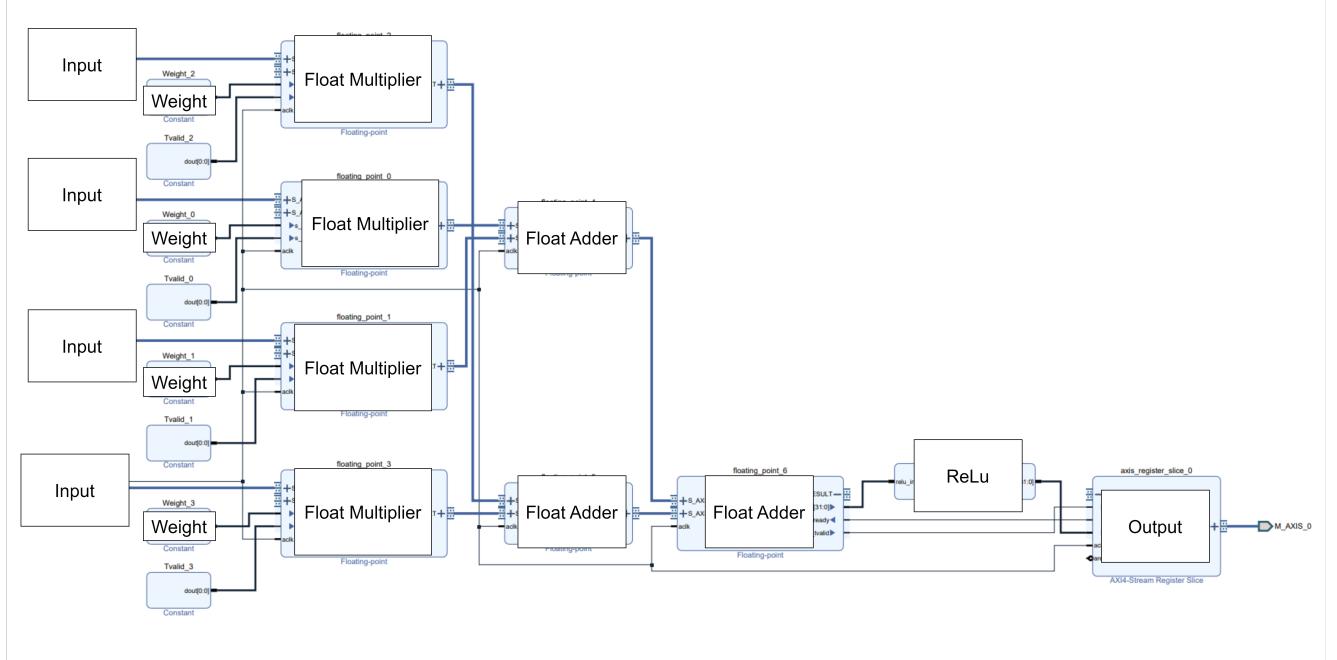


Figura 5.3: Schema a blocchi dell’implementazione di un percettrone.

moltiplicatore float32 è il blocco più complesso dell’architettura, limitando così la velocità complessiva del sistema. Nella letteratura sono state riportate due implementazioni in cui la velocità massima raggiunta è di 505 MHz [10] e 489 MHz utilizzando un moltiplicatore double [15]. Questo suggerisce che, in teoria, sarebbe possibile gestire $500 * 10^6$ vettori in ingresso e generare quindi $500 * 10^6$ vettori in uscita ogni secondo, assumendo l’assenza di limiti dimensionali nell’architettura e problemi di instradamento o trasmissione dei dati.

5.4 Risultati e considerazioni

La rete neurale implementata funziona come previsto: la figura 5.7 mostra una foto in cui viene visualizzato il valore in esadecimale in float32 del risultato, sullo schermo della scheda, che trasformato in float vale 1.08423793316. Inoltre essa funziona in modo ottimale come si evince dalla figura 5.4. Il sistema è stato volutamente progettato utilizzando un’implementazione semplice e con l’uso di dati appositamente semplificati, seguendo le indicazioni descritte nella sezione 4.4.5 in modo da poter avere un confronto diretto tra le due tecnologie. La funzione principale della rete è la classificazione degli ingressi in quattro categorie diverse, utilizzando la codifica one-hot che rappresenta l’indice dell’uscita con il valore più elevato. La classe 0 viene considerata come valore predefinito nel caso in cui non ci sia una corrispondenza chiara con le altre categorie:

$$class = \begin{cases} 4, & \text{if } argmax(Y) = 4 \\ 3, & \text{if } argmax(Y) = 3 \\ 2, & \text{if } argmax(Y) = 2 \\ 1, & \text{otherwise} \end{cases} \quad (5.1)$$

ove Y è il vettore di uscita $Y = [y_1, y_2, y_3, y_4]$ e $argmax()$ restituisce l’indice del valore massimo. La rete è stata allenata e testata in precedenza in un ambiente Python, al fine di determinare i pesi ottimali da utilizzare come costanti all’interno della rete. Questo processo di allenamento ha consentito di adattare e testare la rete neurale per eseguire correttamente la classificazione dei dati secondo le specifiche desiderate in ambiente più semplice ed ad alto livello. Tramite l’analisi temporale dei segnali mostrati in figura 5.6 è possibile estrapolare informazioni cruciali riguardanti le performance della rete, come il throughput e la latenza. Il throughput è fortemente condizionato dalla velocità di esecuzione dell’intero sistema, mentre la latenza dipende anche dal design dell’architettura stessa. È

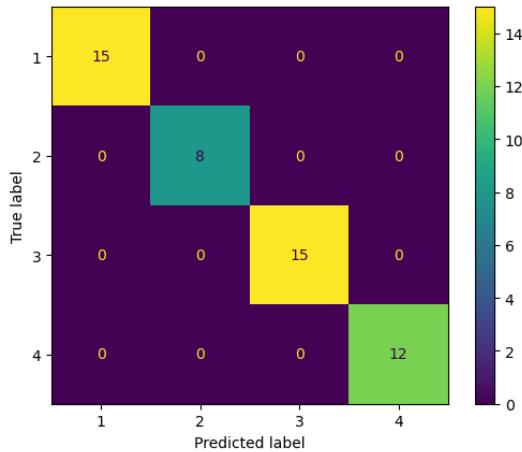


Figura 5.4: Matrice di confusione della rete neurale sviluppata su FPGA.

```
*control_ptr = 0x1;
*control_ptr = 0x0;
```

Figura 5.5: Sono mostrate le due istruzioni consecutive del processore per accendere e spegnere la NN su FPGA

interessante osservare che il segnale di controllo rimane attivo per diversi cicli di clock, nonostante nel codice del processore sia prevista l'istruzione di spegnimento subito dopo quella di accensione come mostra il codice in figura 5.5.

L'osservazione che il segnale di controllo rimane attivo per diversi cicli di clock, nonostante l'istruzione di spegnimento sia presente nell'istruzione successiva del codice del processore, suggerisce la necessità di un processore dedicato al controllo dei flussi di dati in ingresso e in uscita con una frequenza superiore rispetto alla parte implementata nella logica della FPGA. Questo consentirebbe di migliorare le prestazioni complessive del sistema. Analizzando le immagini fornite, è possibile misurare il tempo (misurato in cicli di clock) che intercorre tra l'attivazione della rete neurale, tramite il segnale di controllo, e l'arrivo dei valori validi in uscita indirizzati alla FIFO. Calcolando la differenza tra queste due tempistiche, si ottiene una latenza dell'architettura pari a $68 = 580 - 512$ cicli di clock a una frequenza di 100 MHz, corrispondenti a periodo di 680 ns.

Un ultimo valore importante è la potenza usata, la quale stimata da Vivado, ammonta a 0.565 W. Riassumendo i valori importanti sono riportati in tabella:

Accuratezza	Latenza	Throughput	Potenza
100%	680 ns	100 Mhz 500Mhz massimo	0.56 W

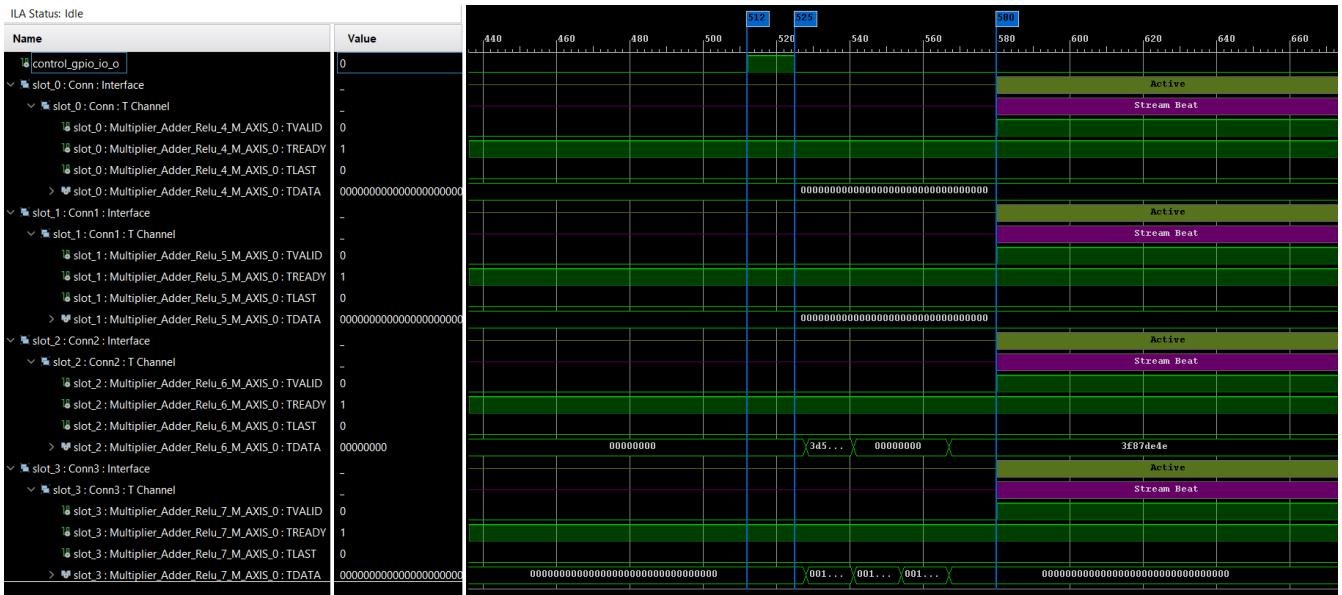


Figura 5.6: Grafico delle tempistiche dei segnali dell’implementazione FPGA per l’elaborazione di un singolo valore.

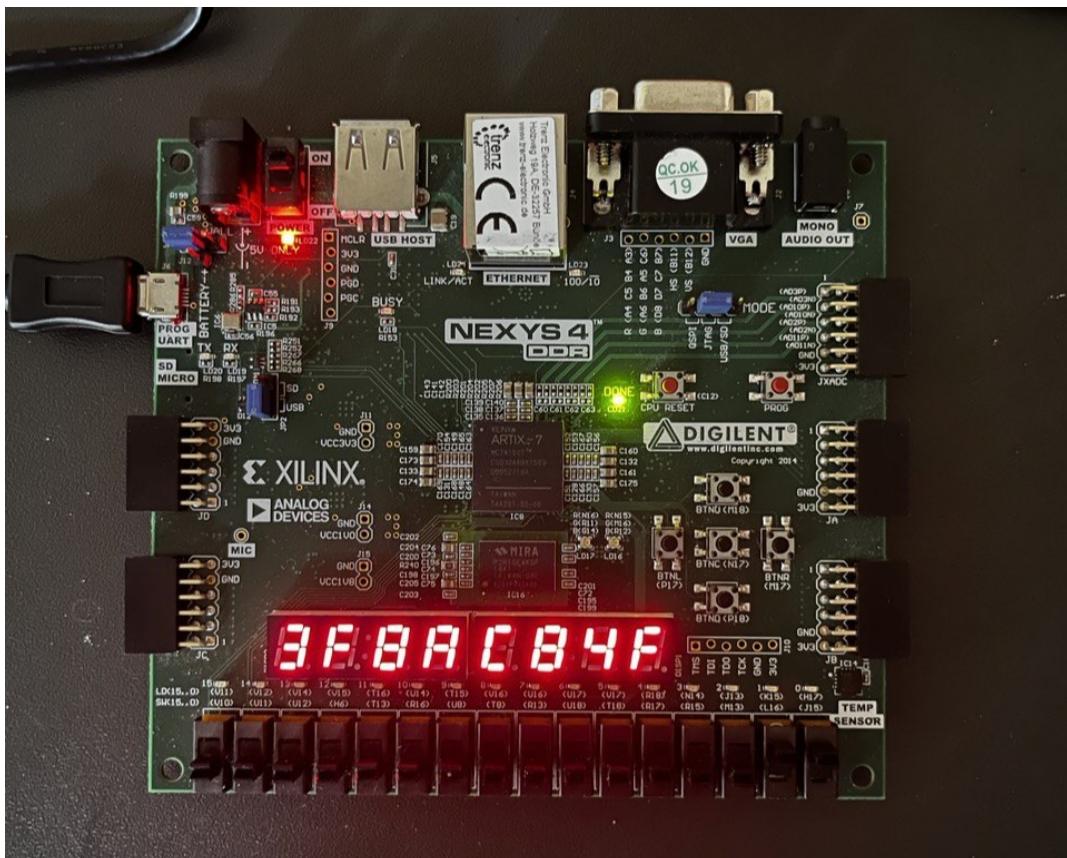


Figura 5.7: La foto mostra un’immagine della scheda durante il funzionamento del sistema.

6 Confronto ONN-FPGA

Nel presente capitolo, verrà condotto un dettagliato confronto tra le ONNs e NNs implementate su FPGA alla luce delle analisi fatte precedentemente. L'obiettivo di questo confronto è valutare le prestazioni, l'efficienza e le caratteristiche di entrambe le architetture, al fine di determinare quale sistema sia più adatto per specifiche applicazioni. Verranno prese in considerazione diverse metriche, come l'accuratezza, la velocità di elaborazione, la latenza e la complessità di implementazione.

6.1 Gestione degli ingressi e delle uscite

Un aspetto da considerare è la gestione degli ingressi e delle uscite delle NNs. Sebbene entrambe le tecnologie possano operare a velocità di elaborazione nell'ordine dei gigahertz, può risultare complesso gestire alti livelli di frequenza. Questo può comportare una maggiore complessità di progettazione e un maggiore rischio di commettere errori di sincronizzazione soprattutto nel caso in cui si riesca ad aumentare ulteriormente le frequenze. Un altro aspetto da tenere in considerazione, per una rete che funziona a sempre più alte frequenze, è la capacità di gestire la generazione dei dati in ingresso e la ricezione degli output in uscita. Inoltre è da considerare che in alcune particolari applicazioni non c'è la necessità di elaborare i dati a così alta frequenza ma è di particolare importanza minimizzare i consumi o le dimensioni.

6.2 Facilità di sviluppo

La facilità di sviluppo è un altro aspetto da considerare. Le NNs implementate su FPGA, allo stato attuale di sviluppo, offrono una maggiore flessibilità e facilità di programmazione rispetto alle ONNs. Questo è dato grazie alla disponibilità di strumenti di sviluppo per linguaggi di programmazione comuni per i dispositivi FPGA e soprattutto grazie alla possibilità di sviluppare e testare algoritmi ad alto livello in ambienti di sviluppo integrato (IDE-Integrated Development Environment) che permettono di avere un'visione dell'insieme dettagliata e semplificata. D'altra parte, le ONNs richiedono una conoscenza più specifica dei componenti ottici e delle loro interazioni, rendendo lo sviluppo potenzialmente più complesso. Allo stato odierno mancano anche software specifici per la creazione, la simulazione, l'addestramento e la verifica (testing) delle ONNs.

6.3 Integrabilità

La potenza di integrazione è un fattore cruciale nel confronto tra le NNs implementate su FPGA e ONNs. L'impiego di FPGA, una tecnologia ben consolidata, consente un'alta integrazione dei componenti, inclusi moduli di memoria e unità di elaborazione di utilizzo generico come CPU, ADC, DAC e una vasta gamma di sensori e periferiche. Questo consente di creare NNs personalizzate che sfruttano appieno le risorse disponibili sull'FPGA. D'altra parte, le ONNs richiedono una maggiore attenzione all'integrazione e alla compatibilità dei componenti ottici nel sistema: le ONNs si basano principalmente su tecnologie analogiche in continua evoluzione, richiedendo quindi componenti ottici specifici progettati da zero e su misura.

6.4 Throughput

Il throughput è un parametro fondamentale per valutare l'efficacia di una tecnologia nell'ambito delle NNs, infatti un alto valore indica che la rete neurale è in grado di gestire un elevato volume di dati in modo efficace. Le NNs implementate su FPGA possono offrire un alto throughput grazie alla loro capacità di parallelizzare i calcoli. Tuttavia, le ONNs possono sfruttare la larghezza di banda elevata delle comunicazioni ottiche per ottenere throughput ancora più elevati, soprattutto in caso di elaborazione su larga scala. Secondo i dati riportati nelle tabelle delle sezioni 4.4.6 e 5.4, a livello

teorico, si può osservare che queste due tecnologie differiscono di un ordine di grandezza in termini di throughput. Tuttavia, tale differenza può aumentare significativamente se si utilizza una ONN che sfrutta diverse lunghezze d'onda tramite l'uso di tecnologie come WDM o DWDM (dense WDM). Ad esempio, considerando una banda di frequenza compresa tra 1546 e 1555 nm e una spaziatura di 100 GHz tra le frequenze, si potrebbe ottenere la capacità di avere almeno 10 canali in accordo con quanto riportato da Flexoptix [1], aumentando ulteriormente il throughput di un altro ordine di grandezza portando così il throughput a circa 33 GHz.

6.5 Accuratezza

L'accuratezza dei risultati ottenuti è un altro aspetto chiave. Sia le ONNs che quelle implementate su FPGA possono ottenere risultati accurati, ma l'accuratezza dipende da diversi fattori, come la complessità del modello, la quantità di dati di addestramento, le tecniche di ottimizzazione utilizzate. Un ulteriore aspetto rilevante da considerare è la natura del sistema di calcolo utilizzato. La FPGA si basa su un sistema di calcolo digitale, in cui l'accuratezza dei dati dipende dalla quantità di bit utilizzati per rappresentarli. Aumentando il numero di bit, è possibile aumentare la precisione dei calcoli. D'altra parte, le ONNs operano generalmente su un sistema analogico, in cui l'accuratezza dipende anche dal rumore presente nella rete e dalla sensibilità dei componenti ottici utilizzati.

6.6 Latenza

La latenza è un fattore critico che influisce sull'utilizzabilità delle uscite delle NNs, specialmente in applicazioni in tempo reale come la guida autonoma o i traduttori istantanei. Un confronto diretto può essere effettuato prendendo in considerazione i valori riportati nelle tabelle di riferimento delle sezioni 4.4.6 e 5.4 . Tuttavia, è necessario considerare che il valore di latenza della FPGA deve essere dimezzato poiché questa tecnologia ha due strati anziché uno come la ONN ottenendo così un valore di 340 ns per la FPGA e 300ps per la ONN da moltiplicare in seguito per il numero di strati utilizzati. In sintesi si può così osservare che le due reti differiscono di tre grandezze d'ordine evidenziando quindi un vantaggio nelle implementazioni ottiche.

6.7 Consumo energetico

Il consumo energetico è un aspetto cruciale in molte applicazioni reali, soprattutto considerato l'evolversi dei sistemi IoT a basso consumo. Le NNs implementate su FPGA possono essere ottimizzate per ridurre il consumo energetico, consentendo un'elaborazione efficiente dei dati. Le ONNs, d'altra parte, possono sfruttare le proprietà di trasmissione ottica per ridurre il consumo energetico complesso rispetto alle implementazioni puramente digitali. Tuttavia, dagli studi e dai valori stimati nei capitoli precedenti, emerge chiaramente che, allo stato attuale delle tecnologie e delle implementazioni proposte, l'implementazione di NNs su FPGA offre un netto vantaggio in termini di consumo energetico rispetto alle ONNs. Pertanto, per quanto riguarda esclusivamente il consumo energetico, le NNs implementate su FPGA rappresentano attualmente la scelta preferibile per molte applicazioni.

6.8 Dimensioni

Le dimensioni delle NNs, in termini di area occupata e risorse richieste, possono condizionare la loro applicabilità in diversi contesti. Le NNs implementate su FPGA offrono la possibilità di ottimizzare l'area occupata, utilizzando al meglio le risorse disponibili, compresa la possibilità di trasformarle in circuiti elettronici integrati per ridurre ulteriormente lo spazio occupato. Questa flessibilità consente di adattare l'architettura della rete neurale alle specifiche esigenze dell'applicazione, minimizzando l'occupazione di area e massimizzando l'efficienza delle risorse utilizzate. D'altra parte, le ONNs integrate possono richiedere ancora il piazzamento di componenti ottici non integrati, e di conseguenza l'aumento delle dimensioni complessive del sistema. Ogni componente ottico non integrato ha dimensioni nell'ordine dei millimetri, il che può comportare una maggiore occupazione di spazio. L'articolo [17] riporta l'implementazione di 56 MZI in uno spazio di 7x3.6 mm, mentre un SOA non integrato ha dimensioni di circa 36x45mm come si può osservare in [2]. Le dimensioni della FPGA sono visibili nel

datasheet [6] il quale riporta dimensioni di 45x45 mm per l'intero circuito integrato. È evidente quindi che le dimensioni devono essere valutate attentamente per ogni specifica applicazione e topologia, soprattutto se si utilizzano componenti ottici non integrati.

6.9 Conclusioni parziali

In questo capitolo sono state esaminate le caratteristiche e le prestazioni delle ONNs e delle NNs implementate su FPGA. Sono stati identificati gli aspetti positivi e negativi di entrambe le tecnologie, evidenziando le loro differenze. I risultati ottenuti sono di fondamentale importanza per supportare la scelta della tecnologia più adatta in base alle specifiche esigenze dell'applicazione. Ad esempio, se la velocità di elaborazione è una priorità, le ONNs potrebbero essere più adatte grazie alla loro elevato throughput. D'altra parte, se la flessibilità, la facilità di sviluppo e il consumo sono importanti, le NNs implementate su FPGA potrebbero essere la scelta migliore.

7 Conclusioni

All'interno di questa tesi è stata effettuata un'analisi dello stato attuale delle ONNs utilizzando una simulazione e confrontando questa con l'implementazione di una topologia teoricamente equivalente su FPGA. Nonostante le limitazioni di tempo e risorse finanziarie che hanno impedito un'implementazione in hardware di soluzioni all'avanguardia, è possibile trarre alcune conclusioni significative.

Come mostrato all'interno capitolo 4, le ONNs presentano delle prestazioni elevate in termini di velocità di elaborazione e di latenza, migliorabili se ulteriormente sviluppate. Rispetto alle tecnologie attualmente disponibili sul mercato, l'opportunità di ampio miglioramento rappresenta un aspetto positivo se paragonato all'elettronica digitale basata sui transistor. Infatti, allo sviluppo odierno, un transistor può arrivare a misurare 2nm [4], una dimensione paragonabile con la stessa dimensione di un atomo, complicando così gli ulteriori sviluppi. L'articolo [14] mostra in dettaglio i limiti di questa tecnologia e della tanto considerata legge di "Moore", secondo la cui ogni 18 mesi il numero di transistor in un chip raddoppia. Tuttavia le NNs basate sull'ottica presentano anche degli aspetti da affrontare, come esploso in questa tesi. Un primo problema da risolvere riguarda la capacità di utilizzare segnali iniziali a bassa potenza e di trovare alternative che richiedano meno energia per la parte non lineare del processo. La possibilità di avere una ONN a basse potenze amplierebbe le possibilità di utilizzo, competendo così anche con il mondo dell'elettronica a basso consumo. Un altro aspetto determinante per l'uso commerciale di questa tecnologia risiede nello sviluppo di strumenti abilitanti ossia l'insieme dei software che facilitano la costruzione, la simulazione, la validazione e infine programmi con cui si può fare l'addestramento del design scelto. Infatti uno dei fattori limitanti di questa tesi è stato riuscire a simulare e fare l'addestramento della ONN, complessità che, nello sviluppo dell'implementazione su FPGA, non è pervenuta. Inoltre nuove ricerche potrebbero aprire la possibilità di scegliere le implementazioni in base all'applicazione specifica. Di seguito sono riportati alcuni esempi:

- l'utilizzo di un singolo strato in loop, per emulare più strati così da poter soddisfare l'esigenza di aver una rete di piccole dimensioni;
- la creazione di strutture generiche che consentano un addestramento diretto in hardware, così da avere un struttura performante e allo stesso tempo generica.
- la creazione di strutture specifiche per gestire grandi flussi di dati con relativamente poca latenza.

In conclusione, dopo le numerose considerazioni e analisi svolte, e alla luce dello stato attuale delle ricerche, ritengo che questa tecnologia sia promettente in tutti quei contesti in cui la potenza non è un problema e vi è una grande quantità di dati da elaborare, come nei server, nelle dorsali di rete, nonché in alcune applicazioni spaziali.

Tabella delle Abbreviazioni

Acronimo	Inglese	Italiano
AI	Artificial intelligence	Intelligenza artificiale
NN	Neural network	Reti neurali
ONN	Optic neural network	Reti neurali ottiche
DNN	Digital neural network	Reti neurali digitali
FPGA	Field-Programmable Gate Array	-
ML	Machine learning	Apprendimento automatico
ANN	Artificial neural network	Reti neurali artificiali
ReLU	Rectified linear unit	Unità lineare rettificata
WDM	Wavelength Division Multiplexing	Multiplexing a divisione di lunghezza d'onda
PNN	Photonic Neural Networks	Reti neurali fotoniche
MZI	Mach-Zehnder Interferometer	Interferometro a zona di Mach-Zehnder
MMI	Multi mode interferometer	Interferometro multimodale
PS	Phase shifter	Sfasatori
SOA	Semiconductor optical amplifier	Amplificatore ottico a semiconduttore
SOA-MZI	Semiconductor optical amplifier Mach-Zehnder interferometer	interferometro a zona di Mach-Zehnder unito ad amplificatori ottici a semiconduttori
XGM-WC	Cross-gain Modulation-Wavelength Converter	convertitore di modulazione a guadagno incrociato e convertitore in lunghezza d'onda
VOA	Variable optical attenuators	Attenuatori ottici variabili
CW	Continuous wave	Onda continua
PL	-	parte logica
DWDM	Dense WDM	Multiplexing denso a divisione di lunghezza d'onda

Bibliografia

- [1] Flexoptix. <https://www.flexoptix.net/en/dwdm-channels>. ultimo accesso 30/06/2023.
- [2] Innolume. <https://www.innolume.com/innoproducts/semiconductor-optical-amplifiers-soa/>. ultimo accesso 29/06/2023.
- [3] Innolume. <https://www.innolume.com/innoproducts/>. ultimo accesso 29/06/2023.
- [4] Introducing the world's first 2 nm node chip. <https://research.ibm.com/blog/2-nm-chip>. ultimo accesso 1/07/2023.
- [5] The mostly complete chart of neural networks, explained. <https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464>. ultimo accesso 30/06/2023.
- [6] Xilinx. <https://www.xilinx-adm.com/files/ed/XC7K355T-2FFG901C.pdf>. ultimo accesso 30/06/2023.
- [7] G.P. Agrawal and N.A. Olsson. Self-phase modulation and spectral broadening of optical pulses in semiconductor laser amplifiers. *IEEE Journal of Quantum Electronics*, 25(11):2297–2306, 1989.
- [8] Lorenzo De Marinis, Marco Cococcioni, Piero Castoldi, and Nicola Andriolli. Photonic neural networks: A survey. *IEEE Access*, 7:175827–175841, 2019.
- [9] Joni Dambre Floris Laporte and Peter Bienstman. Highly parallel simulation and optimization of photonic circuits in time and frequency domain based on the deep-learning framework pytorch. *Scientific reports* 9.1 (2019): 5918.
- [10] K. Manolopoulos, D. Reisis, and V. A. Chouliaras. An efficient multiple precision floating-point multiplier. In *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*, pages 153–156, 2011.
- [11] Xiangyan Meng, Guojie Zhang, Nuannuan Shi, Guangyi Li, José Azaña, José Capmany, Jianping Yao, Yichen Shen, Wei Li, Ninghua Zhu, et al. Compact optical convolution processing unit based on multimode interference. *Nature Communications*, 14(1):3000, 2023.
- [12] George Mourigas-Alexandris, A Tsakyridis, N Passalis, Anastasios Tefas, K Vrysokinos, and Nikolaos Pleros. An all-optical neuron with sigmoid activation function. *Optics express*, 27(7):9620–9630, 2019.
- [13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- [14] James R. Powell. The quantum limit to moore's law. *Proceedings of the IEEE*, 96(8):1247–1248, 2008.
- [15] SUNKARA YAMUNA RANI and BELLAM VARALAKSHMI. Implementation of double precision floating point multiplier in vhdl. *Sign (S)*, 2(10):881–888, 2015.

- [16] Farhad Shokraneh, Simon Geoffroy-Gagnon, Mohammadreza Sanadgol Nezami, and Odile Liboiron-Ladouceur. A single layer neural network implemented by a 4×4 mzi-based optical processor. *IEEE Photonics Journal*, 11(6):1–12, 2019.
- [17] Shuoyi Zhao, Liangjun Lu, Linjie Zhou, Dong Li, Zhanzhi Guo, and Jianping Chen. 16 × 16 silicon mach–zehnder interferometer switch actuated with waveguide microheaters. *Photon. Res.*, 4(5):202–207, Oct 2016.

Allegato A Codice simulazione ONN

```
1 ## Imports
2 # Python
3 %matplotlib inline
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 # Torch
8 import torch
9
10 # PhotonTorch
11 import photontorch as pt
12
13 # Progress Bars
14 from tqdm.notebook import tqdm
15 from tqdm.notebook import trange
16
17 # save the record
18 import inspect
19
20 # avoid truncation in print
21 import sys
22 np.set_printoptions(threshold=1000)
23 np.set_printoptions(edgeitems=3)
24
25
26 # Generation of data for training and testing
27
28 # Data
29 ## Data parameters & settings
30 data_size = 10000
31 test_data_size = 50
32 limit = {"high":1,"low":0}
33 generator = np.random.default_rng()
34 classes = {"class1": [1,0,0,0], "class2": [0,1,0,0], "class3": [0,0,1,0], "class4": [0,0,0,1] }
35 mean = {"mean1": [0.2, 0.8, 0.3, 0.7],
36           "mean2": [0.4, 0.2, 0.9, 0.3],
37           "mean3": [0.3, 0.4, 0.1, 0.6],
38           "mean4": [0.5, 0.6, 0.7, 0.4]}
39 std_deviation = {"std_deviation1": [0.05, 0.05, 0.05, 0.05],
40                   "std_deviation2": [0.05, 0.05, 0.05, 0.05],
41                   "std_deviation3": [0.05, 0.05, 0.05, 0.05],
42                   "std_deviation4": [0.05, 0.05, 0.05, 0.05] }
43
44 ## Generation the data set
45 # the final structure of the data should be something like
46 #      [[[input1, input2, input3, input4], [output1, output2, output3, output4]],
47 #      N      [[input1, input2, input3, input4], [output1, output2, output3, output4]],
48 #      /      [[input1, input2, input3, input4], [output1, output2, output3, output4]],
49 #      T      .
50 #      I      .
51 #      M      .
52 #      E      .
53 #      S      [[input1, input2, input3, input4], [output1, output2, output3, output4]],
```

```

54 #      [[input1, input2, input3, input4], [output1, output2, output3, output4]],  

55 #      [[input1, input2, input3, input4], [output1, output2, output3, output4]]]  

56  

57 ##### training data #####  

58 data = np.empty((data_size,2,4)) #2 and 4 to create a couple of input-output array.  

59 # the array are composed of 4 values  

60  

61 for i in range(data_size):  

62     class_to_generate = np.random.randint(1,5)  

63     data[i][0] = [generator.normal(loc=mean.get("mean{}".format(class_to_generate))[0],  

64         scale=std_deviation.get("std_deviation{}".format(class_to_generate))[0]),  

65         generator.normal(loc=mean.get("mean{}".format(class_to_generate))[1],  

66             scale=std_deviation.get("std_deviation{}".format(class_to_generate))[1]),  

67             generator.normal(loc=mean.get("mean{}".format(class_to_generate))[2],  

68                 scale=std_deviation.get("std_deviation{}".format(class_to_generate))[2]),  

69                 generator.normal(loc=mean.get("mean{}".format(class_to_generate))[3],  

70                     scale=std_deviation.get("std_deviation{}".format(class_to_generate))[3]),]  

71     data[i][1] = classes.get("class{}".format(class_to_generate))  

72 #make sure that all the data are between [0,1]  

73 data = data.clip(0,1)  

74  

75  

76 ##### test data #####  

77 test = np.empty((test_data_size,2,4))  

78 for i in range(test_data_size):  

79     class_to_generate = np.random.randint(1,5)  

80     test[i][0] = [generator.normal(loc=mean.get("mean{}".format(class_to_generate))[0],  

81         scale=std_deviation.get("std_deviation{}".format(class_to_generate))[0]),  

82         generator.normal(loc=mean.get("mean{}".format(class_to_generate))[1],  

83             scale=std_deviation.get("std_deviation{}".format(class_to_generate))[1]),  

84             generator.normal(loc=mean.get("mean{}".format(class_to_generate))[2],  

85                 scale=std_deviation.get("std_deviation{}".format(class_to_generate))[2]),  

86                 generator.normal(loc=mean.get("mean{}".format(class_to_generate))[3],  

87                     scale=std_deviation.get("std_deviation{}".format(class_to_generate))[3]),]  

88     test[i][1] = classes.get("class{}".format(class_to_generate))  

89 #make sure that all the data are between [0,1]  

90 test = test.clip(0,1)  

91  

92 with np.printoptions(threshold=np.inf):  

93  

94     with open("test_file_1layer.txt","w") as file:  

95         file.write("{}\n".format(test))  

96     with open("data_file_1layer.txt","w") as file:  

97         file.write("{}\n".format(data))  

98  

99  

100 ## Data visualization  

101 ### Plot data for traning  

102 import plotly as px  

103 #preapare data  

104 x=data[:, 0, 0]  

105 y=data[:, 0, 1]  

106 z=data[:, 0, 2]  

107 color=data[:, 0, 3]  

108 symbol = []  

109 for i in data[:,1]:  

110     symbol.append(str(torch.argmax(torch.tensor(i, dtype=torch.get_default_dtype()).item()+1))  

111 symbol = ["square" if item == "1" else item for item in symbol]  

112 symbol = ["circle" if item == "2" else item for item in symbol]  

113 symbol = ["diamond" if item == "3" else item for item in symbol]  

114 symbol = ["cross" if item == "4" else item for item in symbol]  

115  

116 #prepare plot  

117 fig1 = px.graph_objs.Scatter3d(x=x, y=y, z=z, marker=dict(color=color, symbol=symbol, opacity=0.9),  

118                                         line=dict(width=0.02), mode='markers')  

119

```

```

120 mylayout = px.graph_objs.Layout(scene=dict(xaxis=dict( title="value 1"), yaxis=dict( title="value 2"),
121                                         zaxis=dict(title="value 3")))
122
123
124 #Plot and save html
125 px.offline.plot({"data": [fig1], "layout": mylayout}, auto_open=True, filename=("data_training.html"))
126 ### Plot training scatter matrix
127 import plotly.express as px
128 #prepare data
129 scattering_matrix_data = []
130 for i in data[:]:
131     scattering_matrix_data.append([i[0][0],i[0][1],i[0][2],i[0][3],str(torch.argmax(torch.tensor(i[1]),
132                                         dtype=torch.get_default_dtype()).item()+1)])
133
134 #plot
135 fig = px.scatter_matrix(scattering_matrix_data, dimensions=[0, 1, 2, 3], color=4)
136 fig.update_traces(diagonal_visible=False)
137 fig.show()
138 ### Plot data for testing
139 import plotly as px
140 #prepare data
141 x=test[:, 0, 0]
142 y=test[:, 0, 1]
143 z=test[:, 0, 2]
144 color=test[:, 0, 3]
145 symbol = []
146 for i in test[:,1]:
147     symbol.append(str(torch.argmax(torch.tensor(i, dtype=torch.get_default_dtype()).item()+1)))
148 symbol = ["square" if item == "1" else item for item in symbol]
149 symbol = ["circle" if item == "2" else item for item in symbol]
150 symbol = ["diamond" if item == "3" else item for item in symbol]
151 symbol = ["cross" if item == "4" else item for item in symbol]
152
153 #prepare plot
154 fig1 = px.graph_objs.Scatter3d(x=x, y=y, z=z, marker=dict(color=color, symbol=symbol, opacity=0.9),
155                                 line=dict(width=0.02), mode='markers')
156
157 mylayout = px.graph_objs.Layout(scene=dict(xaxis=dict( title="value 1"), yaxis=dict( title="value 2"),
158                                         zaxis=dict(title="value 3")))
159
160
161 #Plot and save html
162 px.offline.plot({"data": [fig1], "layout": mylayout}, auto_open=True, filename=("data_test.html"))
163 ### Plot test scattering matrix
164 import plotly.express as px
165 #prepare data
166 scattering_matrix_data = []
167 for i in test[:]:
168     scattering_matrix_data.append([i[0][0],i[0][1],i[0][2],i[0][3],str(torch.argmax(torch.tensor(i[1],
169                                         dtype=torch.get_default_dtype()).item()+1))])
170
171 #plot
172 fig = px.scatter_matrix(scattering_matrix_data, dimensions=[0, 1, 2, 3], color=4)
173 fig.update_traces(diagonal_visible=False)
174 fig.show()
175 # Network - Circuit
176 ### MZI
177
178 class MZI(pt.Network):
179     def __init__(self, kc1=.5,kc2=.5,wga=pt.Waveguide(trainable=True, phase= np.random.random()*2*np.pi),
180                  wgb=pt.Waveguide(trainable=True, phase= np.random.random()*2*np.pi)):
181         super(MZI, self).__init__() # always initialize first
182         self.wg1 = wga #pt.Waveguide(trainable=True, phase= 0)
183         self.wg2 = wgb #pt.Waveguide(trainable=True, phase= 0)
184         self.wg3 = pt.Waveguide(trainable=False)
185         self.wg4 = pt.Waveguide(trainable=False)

```

```

186     self.dc1 = pt.DirectionalCoupler(coupling=kc1, trainable=False)
187     self.dc2 = pt.DirectionalCoupler(coupling=kc2, trainable=False)
188     self.link(3,"3:dc1:2", "0:wg1:1", "3:dc2:2", "0:wg2:1", 2)
189     self.link(0,"0:dc1:1", "0:wg3:1", "0:dc2:1", "0:wg4:1", 1)
190
191 ### Activation Function
192 class ActivationFunction(pt.Network):
193     def __init__(self):
194         super().__init__()
195         self.soi1 = pt.AgrawalSoa(I = 0.0, L=5.e-3 ,W=1e-6)
196         self.soi2 = pt.AgrawalSoa(I=1.7,W=4e-6)
197         self.link(0,"0:soi1:1", "0:soi2:1",1)
198
199 print(torch.where(ActivationFunction().free_ports_at)[0])
200
201
202 ### Network - Circuit definition
203 #### Circuit - Definition
204 class Layer4x4(pt.Network):
205     def __init__(self):
206         super().__init__()
207
208         # termination - empty ports on mzis
209         self.trm1 = pt.Term()
210         self.trm2 = pt.Term()
211         self.trm3 = pt.Term()
212         self.trm4 = pt.Term()
213         self.trm5 = pt.Term()
214         self.trm6 = pt.Term()
215         self.trm7 = pt.Term()
216         self.trm8 = pt.Term()
217         self.trm9 = pt.Term()
218         self.trm10 = pt.Term()
219         self.trm11 = pt.Term()
220         self.trm12 = pt.Term()
221
222         #phases of the training - import if you have already done the training
223         #otherwise set the phases at random
224         #mzi- core
225         self.mz1 = MZI(wga = pt.Waveguide(trainable=True, phase= 5.511269214786659),
226                         wgb = pt.Waveguide(trainable=True, phase= 4.471076212304134))
227         self.mz3 = MZI(wga = pt.Waveguide(trainable=True, phase=2.0540326066925103),
228                         wgb = pt.Waveguide(trainable=True, phase= 6.58561905573238))
229         self.mz2 = MZI(wga = pt.Waveguide(trainable=True, phase= 0.7786257630925012),
230                         wgb = pt.Waveguide(trainable=True,phase= 2.109536543368054))
231         self.mz4 = MZI(wga = pt.Waveguide(trainable=True, phase= 6.04023710764202),
232                         wgb = pt.Waveguide(trainable=True, phase= 1.65570692282401))
233         self.mz5 = MZI(wga = pt.Waveguide(trainable=True, phase= 2.2982816097813865),
234                         wgb = pt.Waveguide(trainable=True, phase= 0.45367019892674343))
235         self.mz6 = MZI(wga = pt.Waveguide(trainable=True, phase= 4.434690336164388),
236                         wgb = pt.Waveguide(trainable=True, phase= 5.31515594359596))
237
238         #mzi - sum
239         self.su1 = MZI(wga = pt.Waveguide(trainable=True, phase= 3.1433636854403333),
240                         wgb = pt.Waveguide(trainable=True, phase= 3.759899227156775))
241         self.su2 = MZI(wga = pt.Waveguide(trainable=True, phase= 3.1423183089782545),
242                         wgb = pt.Waveguide(trainable=True, phase= 5.185065570446145))
243
244         #mzi - out
245         self.out1 = MZI(wga = pt.Waveguide(trainable=True, phase= 3.141592684253872),
246                         wgb = pt.Waveguide(trainable=True, phase= 2.7383028848699373))
247         self.out2 = MZI(wga = pt.Waveguide(trainable=True, phase= 3.141592995276882),
248                         wgb = pt.Waveguide(trainable=True, phase= 0.33552349002188975))
249         self.out3 = MZI(wga = pt.Waveguide(trainable=True, phase= 3.141592622894886),
250                         wgb = pt.Waveguide(trainable=True, phase= 2.3353542077570455))
251         self.out4 = MZI(wga = pt.Waveguide(trainable=True, phase= 3.0655330408990333),
252                         wgb = pt.Waveguide(trainable=True, phase= -0.2715611340760071))
253
254         self.link(0,"3:mz1:1", "3:mz2:2", "0:mz3:1", "3:mz5:2", "0:mz6:2", "3:out1:2",4)
255         self.link(1,"0:mz1:2", "0:su1:1", "3:mz3:2", "0:su2:1", "3:mz6:1", "3:out2:2",5)

```

```

252         self.link(2,"0:mz2:1","3:mz4:2","0:mz5:1","3:out3:2",6)
253         self.link(3,"0:mz4:1","3:out4:2",7)
254         self.link("trm1:0","3:su1:2","0:trm2")
255         self.link("trm3:0","3:su2:2","0:trm4")
256         self.link("trm5:0","0:out1:1","0:trm6")
257         self.link("trm7:0","0:out2:1","0:trm8")
258         self.link("trm9:0","0:out3:1","0:trm10")
259         self.link("trm11:0","0:out4:1","0:trm12")
260
261     # print(self.connections)
262
263 print(torch.where(Layer4x4().free_ports_at)[0])
264
265 class Circuit(pt.Network):
266     def __init__(self, length=1e-5, neff=2.34, ng=3.4, loss=1000):
267         super().__init__()
268         #input
269         self.src1 = pt.Source()
270         self.src2 = pt.Source()
271         self.src3 = pt.Source()
272         self.src4 = pt.Source()
273         #output
274         self.det1 = pt.Detector()
275         self.det2 = pt.Detector()
276         self.det3 = pt.Detector()
277         self.det4 = pt.Detector()
278         #layer
279         self.layer1 = Layer4x4()
280         #activation function
281         self.act1 = ActivationFunction()
282         self.act2 = ActivationFunction()
283         self.act3 = ActivationFunction()
284         self.act4 = ActivationFunction()
285
286         #connections
287         self.link("src1:0","0:layer1:4","0:act1:1","0:det1")
288         self.link("src2:0","1:layer1:5","0:act2:1","0:det2")
289         self.link("src3:0","2:layer1:6","0:act3:1","0:det3")
290         self.link("src4:0","3:layer1:7","0:act4:1","0:det4")
291
292 print(torch.where(Circuit().free_ports_at)[0])
293
294 ### Network - Circuit inizialization
295 circuit = Circuit(length=1.2e-5, neff=2.84, ng=3.2, loss=3.e4)
296
297 ### Environment
298 # create simulation environment
299 env = pt.Environment(dt=1.500e-13, t1=5.0e-10, wl=1.55e-6,freqdomain=False, grad=True)
300         #important to "turn on" the gradient (grad)
301
302
303 ## Caratteristics at the beginnig
304 ### Plot cunfusion matrix
305 from sklearn.metrics import confusion_matrix
306 from sklearn.metrics import ConfusionMatrixDisplay
307
308 #prepare data
309 confusion_matrix_value = [] #format is [[real class],[predicted class]]
310 with env:
311     for i in tqdm(range(len(test))):
312         source = test[i][0]
313         source = torch.tensor(source, device=circuit.device, dtype=torch.get_default_dtype()).rename("s")
314         target = test[i][1]
315         target = torch.tensor(target, device=circuit.device, dtype=torch.get_default_dtype())
316         detected = circuit(source=source)
317         confusion_matrix_value.append([(torch.argmax(target)).item() + 1,

```

```

318         torch.argmax((detected[-1,0,:,:0])).item()+1)
319     print(confusion_matrix_value[-1])
320
321 #plot
322 confusion_matrix_value = np.array(confusion_matrix_value)
323
324 confusion_matrix_graph = ConfusionMatrixDisplay.from_predictions(confusion_matrix_value[:,0].tolist(),
325                                         confusion_matrix_value[:,1].tolist(), labels=[1, 2, 3, 4])
326
327
328 circuit.plot(detected=detected)
329 plt.show()
330 # Training - successive approximations of the gradient
331 to test the network assign the input to the source and run. <br>
332 each cycle the gain of the gradients are decreased to fine tuning the phases.
333 from sklearn.metrics import confusion_matrix
334 from sklearn.metrics import ConfusionMatrixDisplay
335 learning_rate = 1.5
336 for i in range(2):
337     #initial settings
338     learning_rate = learning_rate/2 # multiplication factor for the gradients during optimization.
339     lossfunc = torch.nn.MSELoss() #type of loss
340     optimizer = torch.optim.SGD(circuit.parameters(), learning_rate) #type of optimizazion
341
342     with env:
343         for epoch in tqdm(range(data_size)): #
344             optimizer.zero_grad()
345             source = data[epoch][0]
346             source = torch.tensor(source, device=circuit.device,
347                                 dtype=torch.get_default_dtype()).rename("s")
348             target = data[epoch][1]
349             target = torch.tensor(target, device=circuit.device, dtype=torch.get_default_dtype())
350             detected = circuit(source=source)[-1,0,:,:0] # get the last timestep,
351                                         #the only wavelength, all detectors, the only batch
352             loss = lossfunc(detected, target) # calculate the loss (error) between detected and target
353             loss.backward() # calculate the resulting gradients for all the parameters of the network
354             optimizer.step() # update the networks parameters with the gradients
355             del detected, loss, source, target # free up memory (important for GPU)
356
357 confusion_matrix_value = [] #format is [<real class>, <predicted class>]
358 with env:
359     for i in range(len(test)):
360         source = test[i][0]
361         source = torch.tensor(source, device=circuit.device,
362                               dtype=torch.get_default_dtype()).rename("s")
363         target = test[i][1]
364         target = torch.tensor(target, device=circuit.device, dtype=torch.get_default_dtype())
365         detected = circuit(source=source)
366         confusion_matrix_value.append([(torch.argmax(target)).item()+1,
367                                         torch.argmax((detected[-1,0,:,:0])).item()+1])
368         del detected, source, target
369 #plot
370 confusion_matrix_value = np.array(confusion_matrix_value)
371
372 confusion_matrix_graph = ConfusionMatrixDisplay.from_predictions(confusion_matrix_value[:,0].tolist(),
373                                         confusion_matrix_value[:,1].tolist(), labels=[1, 2, 3, 4])
374
375 #print the value of the phases. You should save them if the training is okay
376 for p in circuit.parameters():
377     print(p.item())

```
