



**UNIVERSITÀ  
DEGLI STUDI  
DI TRIESTE**

---

Dipartimento di Fisica

CORSO DI LAUREA IN FISICA

TESI DI LAUREA

Reti Neurali come approssimatori universali in  
azione:

il caso della trasformata di Fourier discreta

*Laureando:*  
Cattai Alberto

*Relatore:*  
**Prof. Milotti Edoardo**  
*Co-Relatore:*  
**Dott. Principe Giacomo**

*Anno Accademico 2022/2023*



# Abstract

Negli ultimi decenni le reti neurali hanno suscitato un rinnovato interesse grazie ai progressi nell'ambito delle modalità di apprendimento di una macchina e alla disponibilità di ingenti quantità di dati e risorse computazionali. In particolare, molti studi hanno dimostrato come le reti multi-strato siano in grado di approssimare qualsiasi funzione con grado arbitrario di precisione.

Tuttavia, rimangono ancora molti aspetti poco chiari riguardo il meccanismo con cui le reti svolgono l'apprendimento e le proprietà computazionali delle singole architetture. In quest'ottica, questa tesi propone ed esamina un'architettura di rete neurale in grado di approssimare la Trasformata di Fourier discreta tramite l'impiego di una particolare funzione di attivazione, denominata *cosine squasher*.

Dopo un'introduzione ai fondamenti matematici della trasformata di Fourier e della teoria delle reti neurali, viene presentata nel dettaglio la funzione *cosine squasher*, l'architettura della rete e la teoria che ne supporta il successivo sviluppo. Con l'utilizzo di queste conoscenze, la rete viene quindi implementata in Python e addestrata su dataset sintetici, dimostrando la sua efficacia nell'approssimare la Trasformata di Fourier.

Attraverso una serie di esperimenti, si analizzano le proprietà di questo modello e le dinamiche del suo apprendimento. I risultati ottenuti permettono di comprendere meglio il meccanismo con cui le reti, sfruttando la struttura delle proprie funzioni di attivazione, riescono ad emulare trasformazioni complesse di segnali digitali.

Nel complesso, questo lavoro contribuisce ad una migliore comprensione del funzionamento delle reti neurali, apre nuove prospettive di ricerca: nel campo del segnale digitale, della intelligenza artificiale e della sua applicazione in campo scientifico.



# Indice

<b>Abstract</b>	<b>i</b>
<b>Indice</b>	<b>iii</b>
<b>1 Introduzione</b>	<b>1</b>
1.1 La trasformata di Fourier . . . . .	1
1.1.1 La FT nella computazione . . . . .	2
1.2 Le reti neurali . . . . .	3
1.3 Il metodo della Back-Propagation . . . . .	5
1.4 Stato dell'Arte . . . . .	6
1.5 Metodologia e Motivazioni . . . . .	7
<b>2 La Fourier Neural Network</b>	<b>9</b>
2.1 Il ruolo di approssimatore di funzioni . . . . .	9
2.1.1 La funzione <i>Squashing</i> . . . . .	9
2.2 L'approccio di Hornik . . . . .	10
2.3 Struttura della Rete . . . . .	11
<b>3 Implementazione Python della FNN</b>	<b>13</b>
3.1 Le classi . . . . .	13
<b>4 Studio delle proprietà della FNN</b>	<b>19</b>
4.1 Capacità di approssimazione . . . . .	19
4.2 Test Dimensionale . . . . .	20
4.3 Test Parametrici . . . . .	21
4.3.1 Learning Rate . . . . .	22
4.3.2 Numero di neuroni nello strato nascosto . . . . .	23
4.4 Test su segnali reali . . . . .	24
<b>5 Conclusione</b>	<b>27</b>
<b>6 Appendice</b>	<b>29</b>
6.1 Classe: <i>Punti</i> . . . . .	29
6.2 Funzioni d'onda . . . . .	29

6.3 Confronti tra le reti su funzioni d'onda . . . . .	30
<b>Elenco delle figure</b>	<b>33</b>
<b>Bibliografia</b>	<b>35</b>

# Capitolo 1

## Introduzione

Nei campi come la fisica e l'ingegneria l'interpretazione dei segnali è un tema molto importante per l'innovazione e la scoperta scientifica. Negli ultimi anni, lo sviluppo e l'utilizzo delle reti neurali ha aiutato numerosi ricercatori nello studio e interpretazione di fenomeni fisici noti e non.

In particolare, l'utilizzo delle reti neurali come approssimatori di funzioni ha portato a risolvere più velocemente calcoli complessi come: integrali di volume 2-dimensionali nel caso di "forward scattering" [5] e se implementate a reti più complesse (p.e. PINN- Physics Informed Neural Network), permette di risolvere, senza bias sullo spettro di soluzioni, equazioni differenziali come Navier-Stokes per la fluidodinamica[17].

L'avanzamento di queste tecnologie ci richiede dunque una comprensione sempre più mirata al loro funzionamento e ad uno studio matematico approfondito. Uno degli obiettivi di questa tesi è fornire una spiegazione di come la più semplice struttura di questa tipologia di rete funziona e quali sono i suoi limiti nei tipi di funzione da approssimare, incominciando l'analisi dall'origine teorica e matematica di queste reti [10][4].

Per una completa analisi è utile introdurre i concetti matematici su cui si fonda la teoria delle reti neurali e richiamare quelli della trasformata di Fourier.

### 1.1 La trasformata di Fourier

Nella sua descrizione più semplicistica la trasformata di Fourier è un operatore che trasforma una funzione in un'altra funzione mediante un'integrazione. Utilizziamo un rigore matematico per definirla. In molte pratiche fisiche si ha a che fare con una grandezza periodica nel tempo (p.e. un segnale sonoro, luminoso, di trasmissione ecc.) e si vuole risalire alle singole frequenze che la compongono e alle relative ampiezze, senza conoscerle in anticipo. La cosiddetta analisi di Fourier dato un segnale periodico, riesce a decomporlo come somma (sovraposizione) di segnali di tipo sinusoidale, ciascuno con una propria frequenza (multipla intera di una frequenza base) e una propria ampiezza. Definiamo questo operatore attraverso un linguaggio matematico, incominciando dalla definizione dello spazio delle funzioni dove esistono.

Uno spazio vettoriale è detto di **Hilbert** se è definito un prodotto scalare e lo spazio è completo rispetto alla norma indotta da tale prodotto scalare.[2]

Un particolare spazio di Hilbert detto  $L^p(K)$  è definito come l'insieme delle funzioni  $f$ , in un qualsiasi intervallo  $K$  o sottoinsieme misurabile di  $\mathbb{R}$ , tali che:

$$\int_K |f(x)|^p dx < +\infty \quad (1.1)$$

Si fa notare che possono essere considerate funzioni anche in  $\mathbb{R}^n$ . Nel dettaglio, risulta di particolare interesse per noi lo spazio  $L^2(K)$ , ovvero l'insieme delle funzioni:

$$\int_K |f(x)|^2 dx < +\infty$$

dove  $L^2(K)$  è uno spazio vettoriale:  $f, g \in L^2(K) \rightarrow f + g \in L^2(K) \wedge \alpha f \in L^2(K), \alpha \in \mathbb{C}$  e per il Teorema di Riesz-Fischer è completo rispetto alla norma. Utilizzando il Teorema di Fourier: [2]

Sia  $f = f(x) \in L^2(-L, L)$ , dove  $K = (-L, L)$  e siano  $a_n$  (*coefficienti di Fourier*) i prodotti scalari  $a_n = (f_n, f)$  fra  $f(x)$  e le funzioni  $f_n(x)$  del set ortonormale:

$$\frac{1}{\sqrt{L}} \left\{ \cos\left(\frac{n\pi}{L} x\right); \sin\left(\frac{n\pi}{L} x\right) \right\}_{n=0}^{\infty} \quad (1.2)$$

allora la serie (*di Fourier*):

$$f(x) = \sum_{n=0}^{\infty} a_n f_n(x) \quad (1.3)$$

così costruita risulta convergente alla funzione  $f$  nella norma di  $L^2$  (cioè la successione delle somme parziali approssima la funzione  $f$  “in media di ordine due”).

Grazie alla proprietà descritta dal teorema, il set ortonormale (1.2) è detto completo in  $L^2(K)$ .

Utilizzando il set completo in  $L^2(\mathbb{R}) \{e^{-i\omega_k t}\}_{k=-\infty}^{\infty}$ , allora la serie di Fourier può essere scritta come:

$$f(t) = \sum_{-\infty}^{\infty} f_k(\omega) e^{-i\omega_k t} \quad (1.4)$$

Così come i coefficienti  $f_k \propto \int_{-\infty}^{\infty} f(t) e^{-i\omega_k t} dt$  esprimono il contributo della componente  $e^{-i\omega_k t}$ , definiamo una funzione che rappresenterà il contributo della variabile reale  $\omega$ , supposta continua.

L'espressione di questa funzione sarà:

$$g(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt = \hat{f}(\omega) = \mathcal{F}(f) \quad (1.5)$$

Dunque la funzione  $g(\omega)$  è chiamata *trasformata di Fourier* o  $\mathcal{F}$ -trasformata.

### 1.1.1 La FT nella computazione

Per calcolare la Trasformata di Fourier, esistono diversi algoritmi computazionali, ciascuno con caratteristiche specifiche. Uno dei primi metodi è la Trasformata di Fourier Discreta (DFT), che rappresenta una funzione periodica campionata. L'algoritmo DFT coinvolge

una somma di prodotti complessi tra: gli N dati campionati, frazionati in sequenze finite  $x(s)$ , e le radici complesse dell'unità.

$$X(k) = \sum_{s=0}^{N-1} x(s)e^{-\frac{(2\pi i)ks}{N}} \quad (1.6)$$

L'equazione restituisce il coefficiente della componente di frequenza k. Questa operazione può essere computazionalmente costosa, con una complessità di  $O(n^2)$ , dove n è la lunghezza della sequenza. Tuttavia, è la forma più diretta di calcolo della Trasformata di Fourier. Per superare le limitazioni computazionali della DFT, è stata sviluppata la Fast Fourier Transform (FFT), un algoritmo più efficiente che sfrutta la simmetria nella matrice delle radici complesse dell'unità. La FFT riduce la complessità computazionale a  $O(n \log(n))$ , un notevole miglioramento rispetto alla DFT. Questo algoritmo è spesso implementato attraverso l'algoritmo di Cooley-Tukey[3], che sfrutta la ricorsione e la fattorizzazione per suddividere la DFT in sotto-DFT più piccole, attraverso la formula:

$$X(k) = \sum_{s=0}^{\frac{N}{2}-1} x(2s)e^{-\frac{(2\pi i)2ks}{N}} + e^{-\frac{(2\pi i)k}{N}} \sum_{s=0}^{\frac{N}{2}-1} x(2s+1)e^{-\frac{(2\pi i)k(2s+1)}{N}} \quad (1.7)$$

## 1.2 Le reti neurali

Le reti neurali hanno origine nei primi anni '40, quando il neurofisiologo Warren McCulloch e il matematico Walter Pitts svilupparono i primi modelli matematici per riprodurre il funzionamento dei neuroni biologici[12]. Nel 1958, l'ingegnere Frank Rosenblatt introdusse il Perceptron, un modello semplice di rete neurale ispirato allo studio dei circuiti neurali nel cervello[15].

Il perceptron è costituito da una serie di unità computazionali elementari dette neuroni, collegate tramite legami i cui pesi simulano la sinapsi neuronale. Ciascun neurone riceve attraverso le sue connessioni entranti (input) gli stimoli provenienti dall'ambiente esterno o dai neuroni ad esso collegato, li moltiplica per i relativi pesi sinaptici ed emette un segnale di output applicando una funzione di attivazione.

Di particolare importanza è la caratteristica di apprendimento del Perceptron, basata sul meccanismo dell'adattamento dei pesi sinaptici in funzione dell'errore commesso nel computare output corretti. Tramite l'algoritmo di addestramento proposto da Rosenblatt, una rete di Perceptron è in grado di modificare autonomamente i pesi per classificare correttamente nuovi stimoli in input [15].

Entriamo nel dettaglio del Perceptron introducendo del formalismo matematico.

Nel Perceptron proposto da Rosenblatt abbiamo un neurone che agisce come un *classificatore*, il cui output è dato da una funzione di trasferimento binaria:

$$y = sgn(\vec{w} \cdot \vec{x}_\nu) = \begin{cases} 1 & \text{se } \vec{w} \cdot \vec{x}_\nu > 0 \\ 0 & \text{se } \vec{w} \cdot \vec{x}_\nu = 0 \\ -1 & \text{altrimenti} \end{cases}$$

Insieme a gli m esempi  $\vec{x}_\nu$ ,  $\nu = 1, \dots, m$  a cui sono associati gli output desiderati  $\xi_\nu$  e quelli in realtà prodotti  $y_\nu$ .

Si definisce anche una funzione di errore  $E(\vec{w})$  dipendente dai pesi che indica il numero di risposte sbagliate (errori) del neurone:

$$E(\vec{w}) = \frac{1}{2} \sum_{\nu=1}^m (\xi_\nu - y_\nu)^2 = \frac{1}{2} \sum_{\nu=1}^m (\xi_\nu - sgn(\vec{x}_\nu \cdot \vec{w}))^2 \geq 0 \quad (1.8)$$

ed è valida l'uguaglianza *se e solo se*, per ogni esempio  $\vec{x}_\nu$  vale  $y_\nu = \xi_\nu$ , ovvero se:

$$\operatorname{sgn}(\vec{x}_\nu \cdot \vec{w}) = \xi_\nu \quad \nu = 1, \dots, m \quad (1.9)$$

che, essendo  $\xi_\nu \in \{-1, 1\}$ , porta alle seguenti condizioni:

$$\begin{cases} \vec{x}_\nu \cdot \vec{w} \geq 0 & \text{se } \xi_\nu = 1 \\ \vec{x}_\nu \cdot \vec{w} < 0 & \text{se } \xi_\nu = -1 \end{cases} \quad (1.10)$$

moltiplicandole entrambe per  $\xi_\nu$  otteniamo:

$$(\xi_\nu \vec{x}_\nu) \cdot \vec{w} := \vec{x}'_\nu \cdot \vec{w} \geq 0 \quad (1.11)$$

che ci dice che non vengono commessi errori se il vettore dei pesi  $\vec{w}$  ha un prodotto scalare non negativo con tutti gli *eempli modificati*  $\vec{x}'_\nu = \xi_\nu \vec{x}_\nu$ . Dunque l'insieme degli esempi  $\varepsilon = \{\vec{x}_\nu : \nu = 1, \dots, m\}$  e quello degli outputs desiderati  $\Xi := \{\xi_\nu : \nu = 1, \dots, m\}$  definiscono il *problema*, la cui *soluzione* consiste nel trovare un vettore dei pesi che soddisfi l'equazione 1.11.

Nel 1949 lo psicologo Donald Hebb, nel suo libro *Organization of Behaviour*[7] illustra il concetto di plasticità sinaptica, ovvero la capacità di un neurone biologico a variare l'intensità dei suoi collegamenti in base alla frequenza del suo utilizzo, associa questa variazione al processo di apprendimento. Hebb propose una teoria che prevede quando e come avviene la modifica delle sinapsi, questa teoria è oggi nota come la *regola di Hebb*. La sua teoria afferma che se due neuroni sono sistematicamente attivi allo stesso istante, o più precisamente hanno un'attività correlata, la sinapsi che li collega viene rafforzata aumentando così la loro correlazione.

In notazione matematica: dati due output  $y_k, y_j$  l'intensità  $w$  di una sinapsi che li collega viene modificata dalla regola:

$$w' = w + \varepsilon y_k y_j \quad (1.12)$$

dove il prodotto  $y_k y_j$  rappresenta la correlazione fra i due neuroni ed  $\varepsilon$  la costante di proporzionalità.

Rosenblatt introduce a sua volta una modifica nella regola di apprendimento:

$$w_i = w_i + \Delta w_i$$

dove la variazione del peso è:

$$\Delta w_i = \begin{cases} 0 & \text{se } y_\nu = \xi_\nu \\ 2\chi \xi_\nu x_{i,\nu} & \text{altrimenti} \end{cases} \quad (1.13)$$

dove  $\chi$  è una costante che serve a rendere graduale l'apprendimento.

Questa regola è chiaramente ispirata al lavoro di Hebb dato che il termine  $\xi_\nu x_{i,\nu}$  tende ad aumentare quei pesi (sinapsi) per i quali l'input e l'output desiderato sono correlati (uguali) mentre tende a ridurre i pesi per i quali sono scorrelati (opposti).

Utilizzando la forma vettoriale possiamo formularla in modo più compatto:

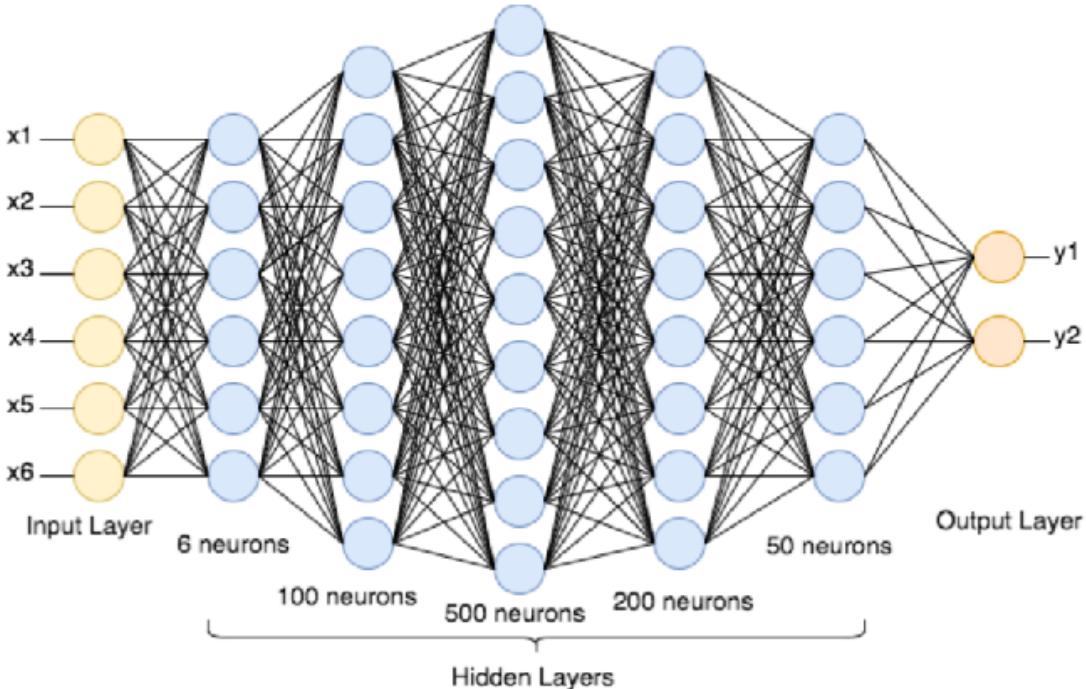
$$\Delta \vec{w} = \chi (1 - \xi_\nu y_\nu) \xi_\nu \vec{x}_\nu \quad (1.14)$$

per cui la regola di apprendimento completa è:

$$\vec{w} = \vec{w} + \chi (1 - \xi_\nu y_\nu) \xi_\nu \vec{x}_\nu \quad (1.15)$$

da cui l'interpretazione geometrica: se per un certo esempio il vettore dei pesi  $\vec{w}$  produce un risultato sbagliato esso viene modificato aggiungendoci l'esempio modificato che risultava mal classificato  $\xi_\nu \vec{x}_\nu$ .

A partire dagli anni '80, i progressi nel machine learning hanno portato allo sviluppo di architetture più complesse con più strati di unità, denominate reti neurali feed-forward o MLP (Multi-Layer Perceptron). In una rete feed-forward i segnali si propagano unicamente in avanti, dall'input all'output, attraverso i diversi strati nascosti, senza feedback. Ciascuno strato svolge un'elaborazione progressivamente più astratta dei dati, consentendo di apprendere rappresentazioni gerarchiche sempre più sofisticate.



**Figura 1.1:** Copyright: Deep Neural Network architecture used in "Deep Learning for Ligand-Based Virtual Screening in Drug Discovery" [1]

### 1.3 Il metodo della Back-Propagation

La back-propagation è un algoritmo di ottimizzazione utilizzato per minimizzare l'errore tra gli output predetti da una rete neurale e gli output desiderati. L'algoritmo si basa sul concetto di retropropagazione dell'errore attraverso la rete, identificando come il cambiamento dei pesi influisce sull'errore complessivo.

La back-propagation è ampiamente utilizzata nell'addestramento di reti neurali per compiti di apprendimento supervisionato. Può essere applicata a una varietà di problemi, come la classificazione, la regressione o altre attività di previsione. È importante notare che la back-propagation è solo una parte dell'intero processo di addestramento delle reti neurali, e spesso viene combinata con algoritmi di ottimizzazione come la discesa del gradiente stocastica per migliorare l'efficienza dell'addestramento.

Definiamo *rete feed-forward* una rete con  $n$  ingressi passati in parallelo ad  $h$  neuroni nello stato intermedio e convergenti ad un unico neurone nello strato finale. Ognuno degli  $h$  neuroni nell'hidden layer ha  $\vec{w}_j$  con  $j = 1, \dots, h$  e produce un output  $y_j = f(\vec{x} \cdot \vec{w}_j)$  con funzione di attivazione generica[9]. Questi output generati dallo strato nascosto sono convogliati in un unico neurone che con gli  $h$  pesi  $W_j$  a sua volta produce un out-

put  $Y = f(\vec{y} \cdot \vec{W})$ , dove il vettore  $\vec{y} \in H_h$ , ovvero appartiene all'iper cubo h-dimensionale  $H_h = \{\vec{x} \in \mathbb{R}^h : -1 \leq x_i \leq 1, i = 1, \dots, h\}$ . Definiamo l'insieme degli output desiderati (target) come  $\xi_\nu, \nu = 1, \dots, n$  e la funzione d'errore da minimizzare:

$$E(\vec{w}_1, \dots, \vec{w}_h, \vec{W}) = \frac{1}{2} \sum_{\nu=1}^n (\xi_\nu - Y_\nu)^2$$

dove  $Y_\nu$  può essere scritto come:

$$Y_\nu = Y(\vec{x}_\nu) = f(\vec{y}_\nu \cdot \vec{W}) = f\left(\sum_{j=1}^h y_{j,\nu} W_j\right) = f\left(\sum_{j=1}^h W_j f(\vec{x}_\nu \cdot \vec{w}_j)\right) \quad (1.16)$$

Per minimizzare la funzione di errore utilizziamo l'algoritmo di discesa lungo il gradiente (GD), ovvero calcoliamo per quali pesi la derivata della funzione d'errore rispetto al peso è nulla  $\frac{\partial E}{\partial w_{i,j}}$ . Questo metodo di minimizzazione prevede le seguenti regole di apprendimento:

$$\begin{cases} W_j = W_j - \chi \frac{\partial E}{\partial W_j} \\ w_{i,j} = w_{i,j} - \chi \frac{\partial E}{\partial w_{i,j}} \end{cases} \quad (1.17)$$

dove, ricordiamo,  $\chi$  è una costante che definisce la gradualità dell'apprendimento. Calcoliamo le derivate della funzione d'errore rispetto ai pesi, partendo da quella dell'ultimo strato:[9]

$$-\frac{\partial E}{\partial W_j} = \sum_{\nu=1}^n (\xi_\nu - Y_\nu) f'(\vec{y}_\nu \cdot \vec{W}) y_{j,\nu} \quad (1.18)$$

mentre per lo strato nascosto usiamo la regola della catena nella notazione di Leibniz, noto per calcolare derivate di funzioni composte:

$$-\frac{\partial E}{\partial w_{i,j}} = \sum_{\nu=1}^n (\xi_\nu - Y_\nu) \frac{\partial Y_\nu}{\partial w_{i,j}} = \sum_{\nu=1}^n (\xi_\nu - Y_\nu) \frac{\partial Y_\nu}{\partial y_{j,\nu}} \frac{\partial y_{j,\nu}}{\partial w_{i,j}} \quad (1.19)$$

ricaviamo dunque:

$$\frac{\partial Y_\nu}{\partial y_{j,\nu}} = f'(\vec{y}_\nu \cdot \vec{W}) W_j \quad \frac{\partial y_{j,\nu}}{\partial w_{i,j}} = f'(\vec{x}_\nu \cdot \vec{w}_j) x_{i,\nu} \quad (1.20)$$

$$\implies -\frac{\partial E}{\partial w_{i,j}} = \sum_{\nu=1}^n (\xi_\nu - Y_\nu) f'(\vec{y}_\nu \cdot \vec{W}) W_j f'(\vec{x}_\nu \cdot \vec{w}_j) x_{i,\nu} \quad (1.21)$$

Definendo  $\delta_\nu = (\xi_\nu - Y_\nu)$ , le regole di aggiornamento dei pesi diventano:

$$\begin{cases} W_j = W_j - \chi \sum_{\nu=1}^n \delta_\nu f'(\vec{y}_\nu \cdot \vec{W}) y_{j,\nu} \\ w_{i,j} = w_{i,j} - \chi \sum_{\nu=1}^n \delta_\nu f'(\vec{y}_\nu \cdot \vec{W}) W_j f'(\vec{x}_\nu \cdot \vec{w}_j) x_{i,\nu} \end{cases} \quad (1.22)$$

## 1.4 Stato dell'Arte

Le reti neurali hanno dimostrato nel tempo straordinarie capacità di approssimazione di funzioni, come verrà illustrato nei capitoli centrali. Un filone di ricerca interessante riguarda l'utilizzo della trasformata di Fourier all'interno delle architetture neurali.

I primi pionieri delle FNN (Fourier Neural Network) furono Jocelyn Sietsma e Robert J.F. Dow [18], che nel 1991 proposero una rete convoluzionale per l'approssimazione di serie temporali periodiche. Il loro modello implementava delle matrici di convoluzione ispirati alle

funzioni generatrici dello spazio della trasformata di Fourier, come coseni e seni discreti. Sie-  
tsma e Dow dimostrarono come queste matrici fossero in grado di estrarre automaticamente  
informazioni nel dominio della frequenza, velocizzando l'addestramento.

Nel 1993, Pao e Takefuji introdussero il concetto di *Fourier Neural Network* vero e proprio [13]. Definirono un modello multi-strato con neuroni aventi come funzione di attivazione sinusoidi e coseni al posto delle classiche sigmoide o tangente iperbolica. Presentarono anche un nuovo algoritmo di *back-propagation* adattato a questa architettura. Applicarono la rete al riconoscimento di forme, ottenendo risultati promettenti.

Il punto di svolta si ebbe nel 1999 quando Adrian Silvescu propose una novità concettuale che estendeva i precedenti modelli neurali al riconoscimento di pattern spazio-frequenziali. Silvescu intodusse una rete convoluzionale con filtri parametrizzati mediante trasformate di Fourier discreta 2D. In particolare, i parametri dei filtri corrispondevano ai coefficienti della DFT (Discrete Fourier Transform), che venivano appresi tramite backpropagation. [19] Grazie a questa innovazione, la rete era in grado di apprendere automaticamente pattern ricorrenti sia nel dominio spaziale che in quello della frequenza, dai livelli inferiori a quelli superiori. Riusciva in sostanza ad affinare i filtri per captare al meglio correlazioni complesse tra posizione e frequenza. Silvescu applicò con successo il suo modello al compito del completamento di immagini deteriorate, ottenendo performance superiori rispetto alle CNN (Convolutional Neural Network) classiche. Questo lavoro diede quindi un contributo importante nell'estendere le FNN al riconoscimento congiunto di relazioni spazio-frequenziali all'interno delle immagini. Tuttavia, la scarsa potenza computazionale dell'epoca limitò l'approfondimento di queste architetture. Con l'avvento del deep learning, le FNN hanno ricominciato a suscitare interesse da parte della comunità scientifica.

Un'innovazione degna di nota è quella pubblicata nel 2015 da Ripple et al. i quali proposero le *Complex Spectral CNN*. Reti convoluzionali dove le convoluzioni vengono estese ai numeri complessi per estrarre relazioni di fase nel dominio della frequenza [14]. Questa ricerca ebbe negli anni a seguire dei risultati importanti nell'ambito medico. [20]

In questo contesto, la tesi propone una architettura di FNN basata su una funzione di attivazione ad-hoc ispirata alla FFT (Fast Fourier Transform). Tale modello verrà addestrato ed analizzato al fine di dimostrarne le potenzialità nel campo dell'approssimazione di funzioni e del segnale digitale, aprendo nuove prospettive di ricerca.

## 1.5 Metodologia e Motivazioni

All'interno di questa tesi verrà esposto la forma più semplice di rete neurale in grado di approssimare qualsiasi funzione continua appartenente ad uno spazio di Schwarz. Il modello si baserà su una architettura di rete molto semplice a tre strati: input, hidden, output. Dove, a differenza del primo ed ultimo strato a cui è applicata una funzione di attivazione lineare, nell'unico strato nascosto presente si utilizzerà una funzione di attivazione definita come *cosine squasher*.

L'intento della tesi è di mostrare come sia possibile ottenere un algoritmo che approssimi funzioni continue con spesa computazionale minima. Verranno illustrati in seguito inferenze sui tempi e sul numero di nodi dello strato nascosto per minimizzare il tempo di calcolo. Viene reso noto al lettore che un tema chiave non affrontato durante l'analisi della rete è l'inferenza sulla dimensione del dataset sintetico, ovvero come varia lo sforzo computazionale cambiando il numero di punti generati dal dataset.



# Capitolo 2

## La Fourier Neural Network

Nel 1988, anno di pubblicazione del loro articolo[4], Ronald Gallant e Halbert White teorizzarono il primo concetto di rete che "non commette errori evitabili", ovvero un algoritmo che riesce ad approssimare con qualsivoglia accuratezza qualsiasi funzione a quadrato integrabile utilizzando sufficienti neuroni nello strato nascosto. Il loro lavoro fu ispirato da quello di Kolmogorov [11] e Hecht-Nielsen[8], i quali trattarono mappature di spazi 1-dimensionali attraverso singoli layer di Perceptron.

Gallant e White implementarono in un'architettura pre-esistente chiamata "RHW Network" [16] (Rumelhart, Hinton e Williams) una nuova funzione di attivazione nello strato nascosto chiamata *cosine squasher*. In questo lavoro si è deciso di costruire l'algoritmo di Gallant e White in linguaggio Python, senza utilizzo di librerie specifiche per le reti neurali (Pytorch, TensorFlow, etc.).

Come illustrato dall'articolo in questione [4] si è costruito una rete feed-forward con un singolo neurone d'input, output ed  $n$  neuroni nello strato nascosto. La novità all'interno di questa rete è l'utilizzo di una nuova funzione di attivazione. Illustriamo l'architettura con l'aiuto di un rigore matematico.

### 2.1 Il ruolo di approssimatore di funzioni

Fino alla fine degli anni '80, nonostante i primi risultati promettenti, mancava ancora una dimostrazione rigorosa della capacità delle reti neurali multilayer di approssimare qualsiasi funzione in modo universale. Alcuni ricercatori avevano citato il teorema di superposizione di Kolmogorov[11] per sostenere questa tesi, ma esso lasciava indefiniti aspetti cruciali come la funzione di attivazione e il numero massimo di unità ausiliarie.

In questo contesto si collocano i fondamentali lavori di Hornik et al. (1989) e Gallant & White (1988), che per primi fornirono una dimostrazione matematica rigorosa del fatto che le reti neurali feedforward sono in grado di approssimare qualsiasi funzione in modo universale. I due articoli, pur portando risultati complementari, concorsero insieme a gettare le basi della concezione delle reti multi-strato come approssimatori universali.

#### 2.1.1 La funzione *Squashing*

All'interno del suo articolo[10], Hornik definisce, con precisione matematica, come si presenta una funzione *squashing*.

Riportiamo la definizione 2.8 mostrata nell'articolo:

Una funzione  $\psi : R \rightarrow [0, 1]$  è una funzione *squashing* se è non decrescente, ovvero se

$$\lim_{\lambda \rightarrow +\infty} \psi(\lambda) = 1 \quad \text{e} \quad \lim_{\lambda \rightarrow -\infty} \psi(\lambda) = 0 \quad (2.1)$$

Esempi noti di funzione *squashing* sono: la funzione di Gallant e White *cosine squasher* e la funzione di soglia  $\psi(\lambda) = 1$  per  $\lambda \geq 0$ , in particolare questa funzione è stata esposta precedentemente sotto il nome di theta di Heaviside, essa è chiamata anche funzione a scalino.

## 2.2 L'approccio di Hornik

Gli autori definiscono formalmente le classi di funzioni approssimabili  $\Sigma\Pi^r$ ,  $\Sigma^r$  e l'insieme delle funzioni continue  $C^r$  tali che  $\{f \in C^r \mid f : \mathbb{R}^r \rightarrow \mathbb{R}\}$ , sottoinsieme delle funzioni misurabili  $M^r$ , secondo la definizione di Borel. (Definizione 2.5)

Hornik et al. riuscirono a dimostrare che è possibile approssimare le funzioni appartenenti a  $C^r$ . Essi definiscono una metrica  $\rho$  nello spazio  $C^r$  in modo da poter introdurre il concetto di *denso rispetto alla metrica*  $\rho$ , ovvero: un elemento, cioè una funzione, di  $S$ , sottoinsieme di uno spazio metrico con metrica  $\rho$ , può approssimare qualsiasi elemento di  $T$ , sottoinsieme dello stesso spazio metrico, con qualsivoglia grado di accuratezza, cioè se la misura, secondo la metrica definita  $\rho$ , tra i singoli elementi è minore di una quantità definita a priori. (Definizione 2.6)

Gli autori applicano il teorema di Stone-Weierstrass per dimostrare che la classe  $\Sigma\Pi^r(G)$  generata da una qualsiasi funzione di attivazione non costante  $G : \mathbb{R}^r \rightarrow \mathbb{R}$  è densa (Definizione 2.7) secondo la misura  $\rho$  nello spazio  $K$  in  $C^r(K)$ , ovvero approssima funzioni continue su insiemi compatti  $K$  (Teorema 2.1).

$\Sigma\Pi^r(G)$  è un'algebra che separa punti e non è mai nulla: ciò permette di soddisfare le ipotesi del teorema e affermare che le reti in  $\Sigma\Pi$  sono capaci di approssimare, con precisione arbitraria, qualsiasi funzione reale continua in un intervallo compatto. Una caratteristica interessante del risultato è che la funzione di attivazione  $G$  della rete può essere qualsiasi funzione continua non costante.

Nell'ordine di espandere il risultato da  $C^r$  a  $M^r$ , gli autori introducono una misura di probabilità  $\mu$  che descrive le frequenze relative rispetto all'avvenimento di un determinato pattern degli input. Definendo la  $\mu$ -equivalenza come: la probabilità che l'uguaglianza tra due funzioni appartenenti a  $M^r$ ,  $f$  e  $g$ , valutate nello stesso punto  $x$  (input), sia uguale a 1. Nella terminologia utilizzata da White [21],  $\mu$  è la "misura dell'ambiente dello spazio degl'input".

Introducono quindi la metrica  $\rho_\mu$  che misura la convergenza di funzioni nello spazio  $M^r$  delle funzioni misurabili, cercando di espandere l'insieme delle funzioni approssimabili. Tale metrica corrisponde alla convergenza in probabilità, equivalente alla convergenza in misura. Ricordando che la notazione  $\mathbb{B}^r$  richiama il campo- $\sigma$  di Borel di  $\mathbb{R}^r$ , ovvero la classe di insiemi più piccola (campo- $\sigma$ ), chiusi rispetto unioni e intersezioni numerabili, che contiene tutti gli insiemi aperti di  $\mathbb{R}^r$ .

Definizione 2.9: Data una probabilità di misura  $\mu$  in  $(\mathbb{R}^r, \mathbb{B}^r)$  si definisce la metrica

$$\rho_\mu : M^r \times M^r \rightarrow \mathbb{R}^r$$

$$\rho_\mu(f, g) = \infty \{ \epsilon > 0 : \mu \{ x : |f(x) - g(x)| > \epsilon \} < \epsilon \}$$

In questa definizione di metrica, due funzioni si dicono vicine *se e solo se* c'è una piccola probabilità ( $\mu$ ) che differiscano significativamente. Uno dei casi limiti è la  $\mu$ -equivaleanza delle funzioni  $f$  e  $g$ , solo in questo caso la metrica è nulla.

Grazie a questa definizione Hornik et al. riescono a collegare la convergenza uniforme su insieme compatto alla convergenza in termini della metrica  $\rho_\mu$  (Lemma 2.2).

Dunque, il Lemma 2.2 risulta fondamentale (insieme al Teorema 2.1) per riuscire a dimostrare che

**Teorema 2.2:** Per ogni funzione continua non costante  $G$ , per ogni  $r$  e per ogni misura di probabilità  $\mu$  su  $(\mathbb{R}^r, \mathbb{B}^r)$ ,

$$\Sigma\Pi^r(G) \text{ è } \rho_\mu\text{-denso in } M^r.$$

Questo risultato amplia il teorema 2.1 a qualsiasi funzione  $G$  utilizzata, ottenendo una rete a singolo strato nascosto  $\Sigma\Pi$  capace di approssimare qualsiasi funzione misurabile con precisione arbitraria, indipendentemente da la dimensione dello spazio di input  $r$  e dell'ambiente dello spazio degl'input  $\mu$ .

I corollari estendono i risultati a: reti vettoriali (**Teorema 2.6**); reti con più hidden layer (**Teorema 2.3**); funzioni misurabili anziché solo continue (**Teorema 2.2**); casi discreti/-finiti (**Corollari 2.4-2.5**). Questi dimostrano la portata universale dei risultati ottenuti, indipendente dall'architettura della rete e dal tipo di funzione approssimata.

I due studi fornirono dunque una solida base teorica alle capacità computazionali delle reti neurali, dimostrando che i successi osservati negli esperimenti non erano casuali ma riflettevano proprietà intrinseche di questi modelli. Essi permisero quindi di concepire le reti neurali a strati multipli come approssimatori universali in senso matematico, indirizzando gli sviluppi successivi sia della teoria che delle applicazioni pratiche. I risultati ottenuti risultano tuttora fondamentali e sono alla base della concezione moderna del potere rappresentativo delle reti neurali artificiali.

## 2.3 Struttura della Rete

Supponiamo di avere  $N$  unità sensoriali, dunque il nostro neurone iniziale riceverà un vettore N-dimensionale dove ad ogni dimensione corrisponde un'informazione dell'impulso. In analogia con i sistemi biologici, assumiamo che l'output emesso dal neurone iniziale e quello finale, siano nello stesso intervallo dell'impulso ricevuto in ingresso. Questo comportamento dei neuroni biologici è ben approssimabile da una funzione di attivazione lineare:

$$y_l = H\left(\sum_{j=1}^N w_{lj}x_j + \gamma_l\right) = H(w_l x_l + \gamma_l) = w_l x_l + \gamma_l \quad (2.2)$$

Nel momento in cui gli  $y_l$ , con  $l = 1, \dots, h$  numero di neuroni nello strato nascosto, entrano nei rispettivi neuroni dello strato hidden, viene utilizzata una funzione di *squashing*, nello specifico:

$$Y_l = F(y_l) = \begin{cases} 0 & -\infty < y_l < -\frac{\pi}{2} \\ \frac{\cos(y_l + \frac{3\pi}{2}) + 1}{2} & -\frac{\pi}{2} \leq y_l \leq \frac{\pi}{2} \\ 1 & \frac{\pi}{2} < y_l < +\infty \end{cases} \quad (2.3)$$

detta *cosine squashing*.

I valori processati dalla funzione d'attivazione dello strato nascosto vengono trasferiti al

neurone finale come un vettore colonna  $h$ -dimensionale. L'unità finale utilizza una funzione lineare (Equazione 2.1) come quella del primo strato, in modo tale che la rete produca un output della stessa dimensione dell'input:

$$O_k = G\left(\sum_{l=1}^h Y_l \beta_l\right) = G\left(\sum_{l=1}^h \beta_l F(w_l x_l + \gamma_l)\right) \quad (2.4)$$

La parte più importante in una rete neurale è rendere efficiente l'apprendimento della rete. Questo significa trovare un vettore dei pesi che ci permette di riprodurre il risultato che si vuole ottenere. Nel nostro caso è l'andamento di una funzione continua a quadrato integrabile.

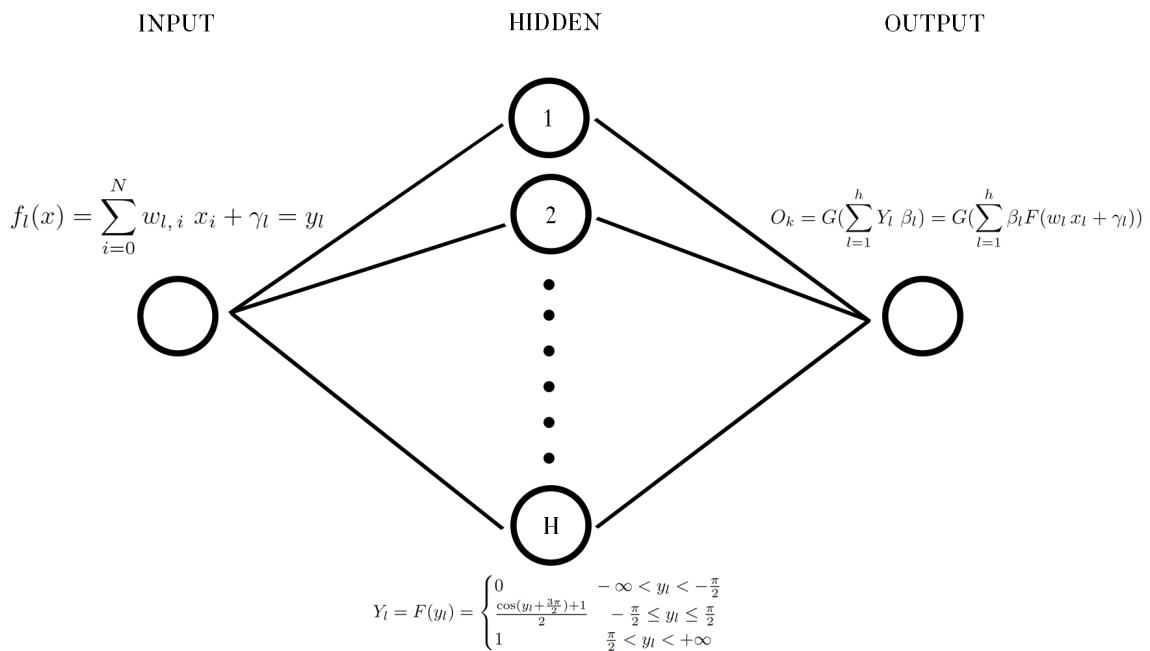
Per utilizzare un'analogia biologica, i pesi del neurone corrispondono alla "forza" della connessione tra le unità neuronali ma, data l'incapacità di comprendere a pieno il "ragionamento" negli strati nascosti a causa della loro definizione, è ragionevole utilizzare una spiegazione statistica: i pesi rappresentano la probabilità che quello specifico neurone abbia più importanza di un altro, come una sorta di media, per l'appunto, pesata.

Per "incapacità di comprendere il ragionamento" s'intende l'impossibilità di trovare una correlazione tra il risultato della rete e la tipologia di informazioni estratta dalla rete e processata negli strati intermedi.

Esattamente come illustrato da Gallant e White, i pesi finali tenderanno ad avere lo stesso valore dei coefficienti di Fourier della serie rappresentata di ordine  $N$ , ovvero della stessa dimensione del vettore impulso [4].

L'apprendimento in questo caso è il metodo della *Back-Propagation* definito precedentemente, con funzione d'errore:

$$E(w) = \frac{1}{N} \sum_{k=1}^N (\epsilon_k - O_k)^2 = \frac{1}{N} \sum_{k=1}^N (\epsilon_k - G\left(\sum_{l=1}^h Y_l \beta_l\right))^2 \quad (2.5)$$



**Figura 2.1:** Struttura della rete di Gallant & White. Copyright: Nicolò Gazzetta, 2023

# Capitolo 3

## Implementazione Python della FNN

Per la stesura della rete ho deciso di utilizzare il linguaggio di programmazione Python che grazie alla sua versatilità e alla vastità di librerie accessibili, risulta essere uno dei linguaggi più utilizzati nella programmazione di Reti Neurali, algoritmi di Machine Learning, etc.. Una delle pecurialità di questo linguaggio è la possibilità di utilizzare diversi paradigmi di programmazione, Python supporta cioè sia la programmazione procedurale (che fa uso delle funzioni), sia la programmazione funzionale (includendo iteratori e generatori), sia la programmazione ad oggetti dove gli *oggetti* svolgono la funzione di racchiudere in un'unica unità organizzativa sia i dati che il comportamento. Proprio su quest'ultimo è ricaduta la mia scelta per la creazione della logica del codice.

Il codice della rete è stato scritto autonomamente con il solo utilizzo della libreria *NumPy* [6], fondamentale per la manipolazione di dati multidimensionali, e alcune librerie di base come: *math* per l'implementazione di funzioni matematiche e *os* per poter spostarsi all'interno del disco rigido e salvare i dati più importanti. Successivamente ho ricorso all'utilizzo di *Matplotlib* per creare i grafici.

Ho deciso di strutturare la rete in diverse *classi* per definire le caratteristiche degli oggetti, ovvero i loro attributi e i loro metodi.

A causa della lunghezza del codice, riportato per intero all'interno dell'appendice, andremo ad illustrarlo partendo da una visione d'insieme fino a definire ogni funzione utilizzata.

### 3.1 Le classi

**Punti** : questa classe permette di generare dei punti equidistanti all'interno di un intervallo chiuso  $[a, b] \in \mathbb{R}$  e costituirà il nostro dataset "sintetico", ovvero sarà l'intervallo a cui verrà applicata una funzione a quadrato integrabile, definita come *target*. Il codice è riportato in appendice (Appendice 1.1).

**Network** : questa classe è il corpo centrale dell'algoritmo e comprende quasi 200 linee di codice, risulta dunque più utile, per comprenderne al meglio il funzionamento, illustrarla attraverso i diversi metodi inseriti:

`__init__` : è un tipo di metodo speciale detto *costruttore*, lo scopo dei costruttori consiste nell'inizializzare (assegnare valori) alle variabili date della classe quando viene

creato un oggetto della classe. In questo metodo vengono definite le variabili fondamentali per definire l'architettura della rete.

```

1 class Network:
2     def __init__(self, x, yy, num_input, num_hidden, num_output,
3                  epoc, lr):
4         self.x = x
5         self.yy = yy
6         self.num_input = num_input
7         self.num_hidden = num_hidden
8         self.num_output = num_output
9         self.EPOCHS = epoc
10        self.lr = lr
11        self.weight = self.pesi
12        self.cos = self.cosine_squasher
13        self.cos_deriv = self.cosine_squasher_deriv
14        self.log = self.logistic_function
15        self.log_deriv = self.logistic_function_deriv

```

**Listing 3.1:** Il metodo speciale permette di definire gli "input" che entreranno nella classe quando l'istanza verrà creata

dove:  $x$  sono i punti creati da *Punti*,  $yy$  sono i punti della funzione che la rete deve approssimare,  $num\_input$ ,  $num\_hidden$ ,  $num\_output$  sono il numero di neuroni nei diversi strati,  $EPOCHS$  è il numero di cicli di apprendimento che la rete deve compiere prima di fornire il risultato finale,  $lr$  è il valore del *learning rate*, corrisponde alla costante  $\chi$  vista nell'equazione 1.13.

*logistic\_func* : viene definita una delle funzione d'attivazione che servirà per il confronto con la *cosine squasher* dall'algoritmo di Gallant e White. La funzione "logistica" ha forma:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (3.1)$$

```

1     def logistic_function_deriv(self, k):
2         y = np.zeros(k.shape)
3         for i in range(k.shape[0]):
4             for j in range(k.shape[1]):
5                 y[i, j] = np.exp(-k[i, j]) / (1 + np.exp(-k[i, j]))
6         return y
7
8     def logistic_function(self, k):
9         y = np.zeros(k.shape)
10        for i in range(k.shape[0]):
11            for j in range(k.shape[1]):
12                y[i, j] = 1 / (1 + np.exp(-k[i, j]))
13
14        return y

```

**Listing 3.2:** Funzione logistica e la sua derivata

*cosine\_squasher* : è la funzione di attivazione applicata allo strato nascosto, definita nell'articolo di Gallant e White [4] e descritta nella Sezione 2.1.1. La derivata della funzione sarà necessaria per implementare l'algoritmo di back-propagation (Sez. 1.3).

```

1     def cosine_squasher(self, k): #funzione di attivazione
2         y = np.zeros(k.shape)
3         for i in range(k.shape[0]):
4             for j in range(k.shape[1]):

```

```

5             if k[i, j] < -np.pi/2:
6                 y[i, j] = 0
7             elif k[i, j] > np.pi/2:
8                 y[i, j] = 1
9             else:
10                y[i, j] = (np.cos(k[i, j]
11                           + 3/2 * np.pi)
12                           + 1) / 2
13
14
15     def cosine_squasher_deriv(self, k): #derivata della funzione
16         di attivazione
17         y = np.zeros(k.shape)
18         for i in range(k.shape[0]):
19             for j in range(k.shape[1]):
20                 if k[i, j] < -np.pi/2 or k[i, j] > np.pi/2:
21                     y[i, j] = 0
22                 else:
23                     y[i, j] = -0.5 * np.sin(k[i, j]
24                                     + 3/2 * np.pi)
25
26
27     return y

```

**Listing 3.3:** cosine squasher e la sua derivata

*pesi* : metodo che inizializza il valore dei pesi secondo una distribuzione normale standard con dimensione 2 (input,output)

```

1     def pesi(self, num_i, num_h): #inizializza casualmente i pesi
2         per la rete neurale
3         w = np.random.randn(num_i, num_h)
4
4     return w

```

**Listing 3.4:** Inizializzazione dei pesi

*NN\_cos* : questo metodo è, nella sua semplicità, l'architettura della rete neurale con funzione di attivazione di Gallant e White. Nella sua struttura viene considerato un singolo punto dell'intervallo e poi processato all'interno della rete. Inoltre, sono compresi l'algoritmo di aggiornamento dei pesi tramite back-propagation e la funzione d'errore, definita come la distanza quadratica tra l'output ottenuto e quello desiderato, illustrata nel capitolo introduttivo (Sezione 1.4).

```

1     def NN_cos(self, w1, b1, w2, b2, i): #Rete Neurale con cosine
2         squasher
3
4         hidden_input = np.dot(self.x[i], w1) + b1 #input layer
5         hidden_output = self.cos(hidden_input)
6         prediction_F = (np.dot(hidden_output, w2) + b2)
7
8
9         delta = prediction_F - self.yy[i]
10        deriv = self.cos_deriv(hidden_input)
11        grad_hidden = np.dot(delta, w2.T)
12        grad_w2 = np.dot(hidden_output.T, delta)
13        grad_b2 = np.sum(delta, axis=0, keepdims=True)
14        grad_w1 = np.dot(self.x[i].T, grad_hidden * deriv)
15        grad_b1 = np.sum(grad_hidden * deriv, axis=0, keepdims=True)
16
17
18        w1 = w1 - self.lr * grad_w1
19        b1 = b1 - self.lr * grad_b1
20        w2 = w2 - self.lr * grad_w2
21        b2 = b2 - self.lr * grad_b2

```

```

19
20     return prediction_F, w1, b1, w2, b2

```

**Listing 3.5:** Fourier Neural Network secondo Gallant & White

*NN\_log* : viene riprodotto la stessa architettura precedente variando la funzione di attivazione dello strato nascosto. Risulterebbe pleonastico riportare il codice in questo caso.

*Cosine\_Network* : a questo punto viene definito l'addestramento vero e proprio della rete. Dopo aver inserito ogni punto in modo ordinato all'interno della rete, e quindi prodotto un output per la coordinata inserita, si confronta globalmente i punti di output generati dalla rete e quelli della funzione da approssimare, in modo da avere un errore non solo sulla singola coordinata ma anche sull'intero vettore. Viene calcolata l'efficienza della rete ed introdotte delle regole di fuoriuscita dal ciclo.

```

1 def Cosine_Network(self): #Algoritmo di apprendimento
2     losses = np.array([], dtype=np.float32)
3     start = timeit.default_timer()
4     timer = np.array([], dtype=np.float32)
5     R = np.array([], dtype=np.float32)
6     w1 = self.weight(self.num_input, self.num_hidden)
7     b1 = np.zeros((1, self.num_hidden))
8     w2 = self.weight(self.num_hidden, self.num_output)
9     b2 = np.zeros((1, self.num_output))
10
11     for epoch in range(self.EPOCHS):
12         pred_array = np.array([], dtype=np.float32)
13         for i in range(len(self.x)):
14             pred, w1, b1, w2, b2 = self.NN_cos(w1, b1, w2, b2
15 , i)
16             pred_array = np.append(pred_array, pred[0][0] )
17             reshape(-1,1)
18
19             sst = np.sum((self.y - np.mean(self.y))**2)
20             ssr = np.sum((self.y - pred_array)**2)
21             r2 = 1 - (ssr / sst)
22             R = np.append(R, round(r2,6)*100)
23             loss = 1/len(self.x)*np.sum((pred_array - self.y)
24 **2)
25             losses = np.append(losses, loss)
26             stop_epoch = timeit.default_timer()
27             timer = np.append(timer, stop_epoch - start)
28
29             if epoch % 10 == 0 :
30                 sst = np.sum((self.y - np.mean(self.y))**2)
31                 ssr = np.sum((self.y - pred_array)**2)
32                 r2 = 1 - (ssr / sst)
33                 stop = timeit.default_timer()
34                 print(f'Epoch: {epoch} | Loss: {loss} |
35                 Efficienza: {round(r2,6)*100}%')
36                 if r2 > 0.99:
37                     stop = timeit.default_timer()
38                     print(f'Epoch: {epoch}| Loss: {loss} |
39                     Efficienza: {round(r2,6)*100}%')
40                     break

```

```
38     return pred_array, losses, timer, R, epoch
```

**Listing 3.6:** Algoritmo di apprendimento

*Logistic\_Network* : il codice si presenta uguale al precedente, la differenza sta nell'utilizzo della rete neurale con funzione di attivazione logistica *NN\_log*.



# Capitolo 4

## Studio delle proprietà della FNN

In questo capitolo verranno esposti ed analizzati i vari test effettuati sulla rete in modo da ottimizzare quelli che nel campo del Machine Learning vengono definiti come *Hyper-parametri*. Questi parametri "speciali" si differenziano dai parametri "normali" (pesi e bias) perchè definiscono in maniera "rigida" l'architettura della rete neurale. A differenza dei pesi e dei bias che vengono modificati ad ogni ciclo di apprendimento, gli *hyper-parametri* rimangono immutati e non possono essere modificati se non a costo di fare un *reset* della rete. Questi parametri "speciali" servono a controllare come una rete impara e risolve un compito specifico. Nel nostro caso possiamo definire 3 hyper-parametri: numero di neuroni dello strato nascosto, *learning rate*, numero di *epoch*.

Sono stati effettuati diversi tipi di test per caratterizzare la rete. Possiamo dividere i test eseguiti in due tipologie: parametrici e dimensionali.

La prima tipologia comprende i test in cui sono stati fatti variare il valore degli hyper-parametri, mentre la seconda il numero delle dimensioni del dataset sintetico, ovvero il numero delle dimensioni del vettore input (impulso) entrante nel neurone.

Entrambe le tipologie di test sono state eseguite confrontando l'andamento con la rete neurale logistica, ovvero la rete che ha come funzione di attivazione del layer nascosto la funzione logistica. Nella prossima sezione verranno mostrati i plot delle efficienze e dell'errore in funzione della variazione dei parametri esposti precedentemente.

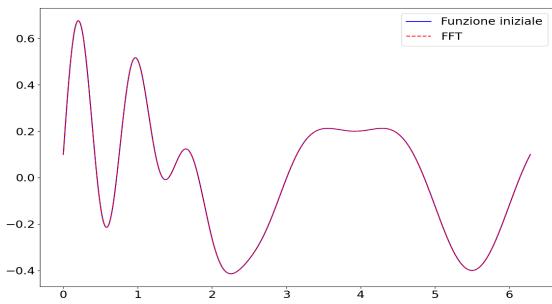
### 4.1 Capacità di approssimazione

Come predetto dall'articolo di Gallant & White [4], la rete riesce a superare il 99.9% di accuratezza nell'approssimazione, risultando dunque un approssimatore di funzioni a quadrato integrabile. I seguenti risultati sono stati ottenuti mediando i valori su 30 cicli di apprendimento, in particolare, le capacità di approssimazione delle reti vengono esposti nella Figura 4.1.

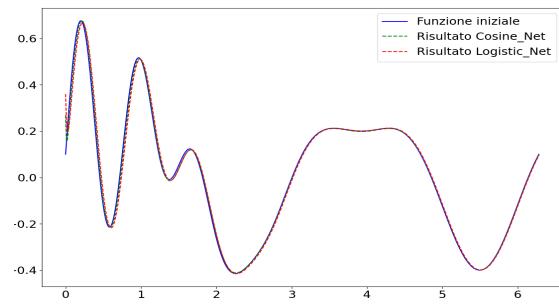
Questo risultato è stato ricavato con dei valori di hyper-parametri fissati:

- numero di neuroni nello strato nascosto = 128
- learning rate =  $1 * 10^{-2}$
- numero di epoch di addestramento = 50

e un vettore di input di dimensione 600.



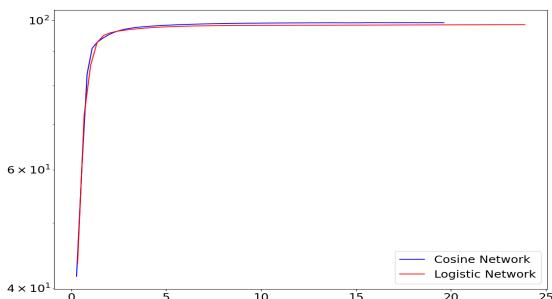
(a) Trasformata di Fourier della funzione target



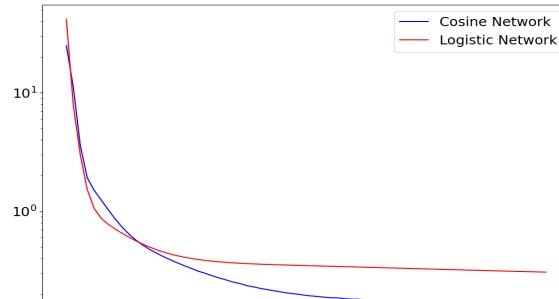
(b) Approssimazione della rete della funzione target

**Figura 4.1:** Confronto tra trasformata e rete neurale

Come si nota nella Fig. 4.2, la *Cosine\_Network* risulta essere più conveniente in termini di efficienza rispetto al tempo computazionale ed errore rispetto alle epoche di addestramento. La rete con funzione di attivazione *cosine squasher* raggiunge in minor tempo un'alta efficienza (oltre il 99.9%) rispetto all'altra rete testata.



(a) Andamento dell'Efficienza in funzione del tempo computazionale



(b) Andamento della Loss in funzione delle epoche

**Figura 4.2:** Confronto tra Logistic e Cosine Network

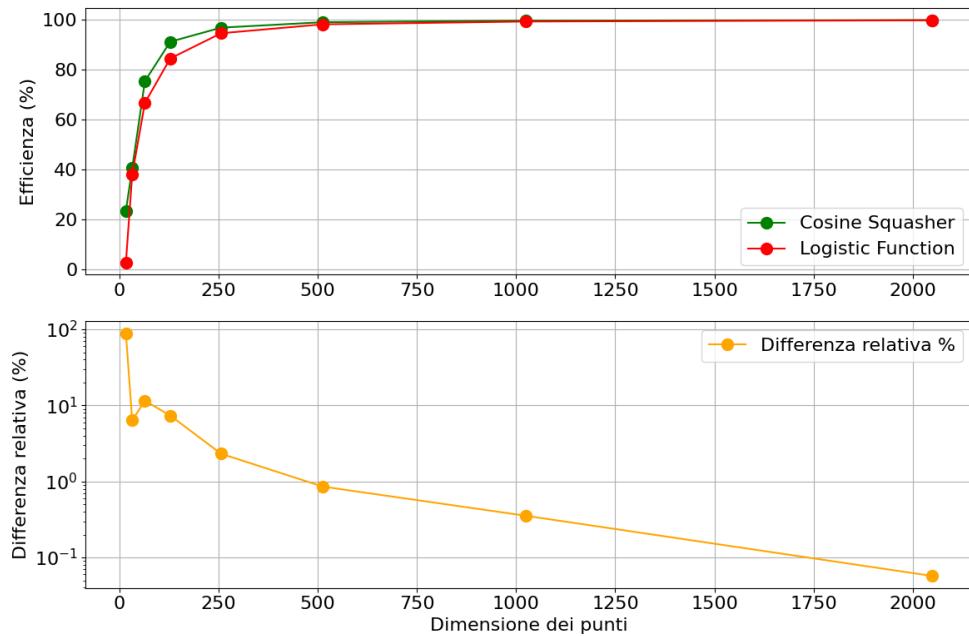
## 4.2 Test Dimensionale

Vengono proposte in questa sezione, dei confronti tra le due tipologie di rete in funzione della variazione della dimensione del vettore input, ovvero in funzione della variazione della quantità di punti generati all'interno dell'intervallo chiuso. I test sono stati effettuati con i seguenti valori fissati per gli hyper-parametri:

- numero di neuroni nello strato nascosto = 128
- learning rate =  $1 * 10^{-2}$
- numero di epoche di addestramento = 50

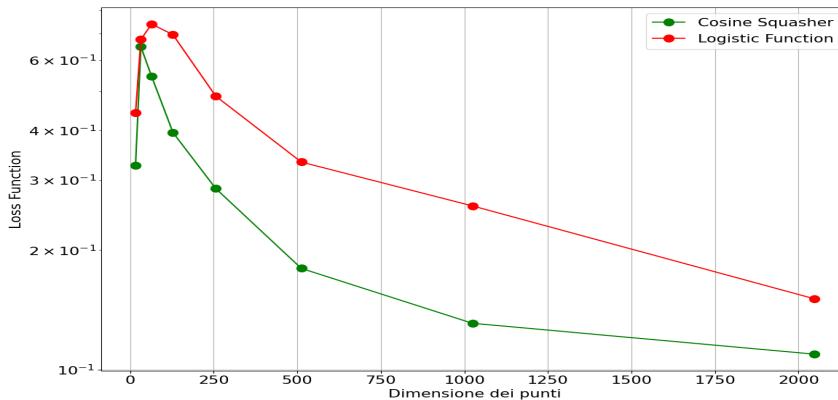
E' stato iterato l'addestramento facendo variare le dimensioni tra i valori:(16, 32, 64, 128, 256, 512, 1024, 2048). Il ciclo è stato ripetuto 50 volte e i risultati sono stati mediati sul numero totale di cicli.

Dalla Figura 4.3 si può notare come la rete con la funzione *cosine squasher* di Gallant & White ottenga una maggiore efficienza anche con una quantità ridotta di punti a disposizione, a differenza della controparte che ha bisogno di dimensioni maggiori per raggiungere un'efficienza ragionevolmente alta.



**Figura 4.3:** Efficienza in funzione della dimensione con grafico della differenza relativa tra le efficienze

Analizzando la Figura 4.4 notiamo l'errore calcolato dalla rete. Risulta evidente che per la FNN l'errore sia molto minore per tutte le dimensioni dei punti. Questo risultato ci può portare ad affermare l'ottima compatibilità della *Cosine\_Network* a dataset ristretti con intervalli tra i punti del dominio anche molto distanti tra loro.



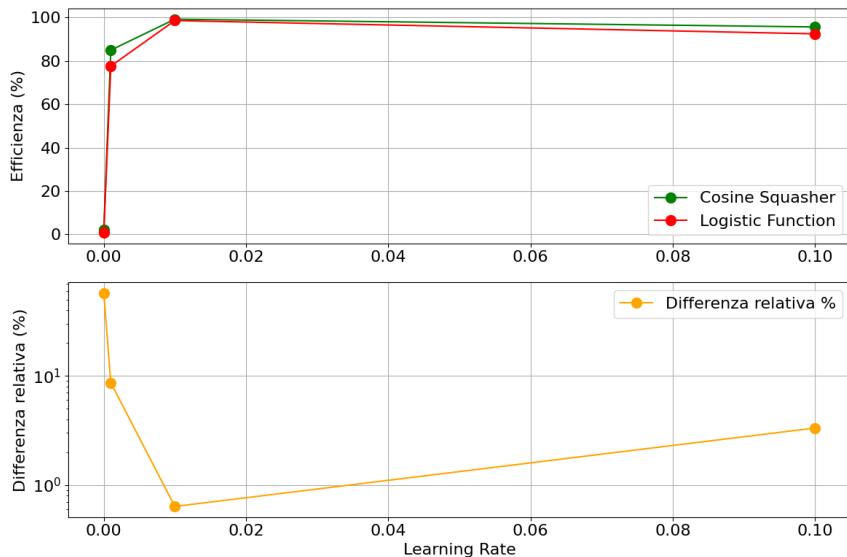
**Figura 4.4:** Errore in funzione della dimensione dell'input

### 4.3 Test Parametrici

In questa sezione andremo ad analizzare i risultati ottenuti dalle reti facendo variare due hyper-parametri: learning rate e numero di neuroni nello strato nascosto. Si è deciso di non modificare il terzo hyper-parametro (numero di epoche), avendo ottenuto un risultato determinante già nella sezione precedente, ed è stato impostato il valore dimensionale dell'input a 600.

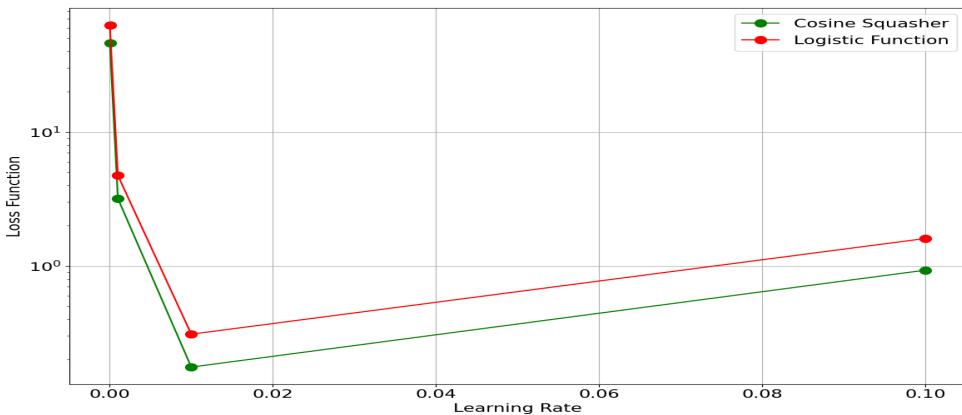
### 4.3.1 Learning Rate

Il primo test parametrico è stato effettuato variando il learning rate tra i valori:  $(1 * 10^{-1}, 1 * 10^{-2}, 1 * 10^{-3}, 1 * 10^{-4}, 1 * 10^{-5})$ , mantenendo fissati i restanti hyper-parametri. Si nota come, già con un learning rate molto basso (i.e.  $1 * 10^{-5}$ ), la FNN ottenga un'efficienza molto alta. Va sottolineato che in Figura 4.5 è possibile notare una diminuzione nell'efficienza a valori alti di *learning rate*, dovuto ad un'incapacità della rete di fermarsi all'interno di un minimo locale. Si ricordi che il learning rate è la costante che indica la gradualità dell'apprendimento, ovvero quanto è incidente l'aggiornamento del peso sul peso stesso, dunque un basso learning rate significa apprendimento più lento ma con più probabilità di individuare e rimanere "bloccato" in un minimo locale della funzione, viceversa per un valore alto.



**Figura 4.5:** Andamento dell'efficienza in funzione del learning rate

Ancora una volta la rete di Gallant & White risulta essere la più efficiente a parità di learning rate. Seppur il confronto delle *loss* non evidenzi una differenza sostanziale tra le

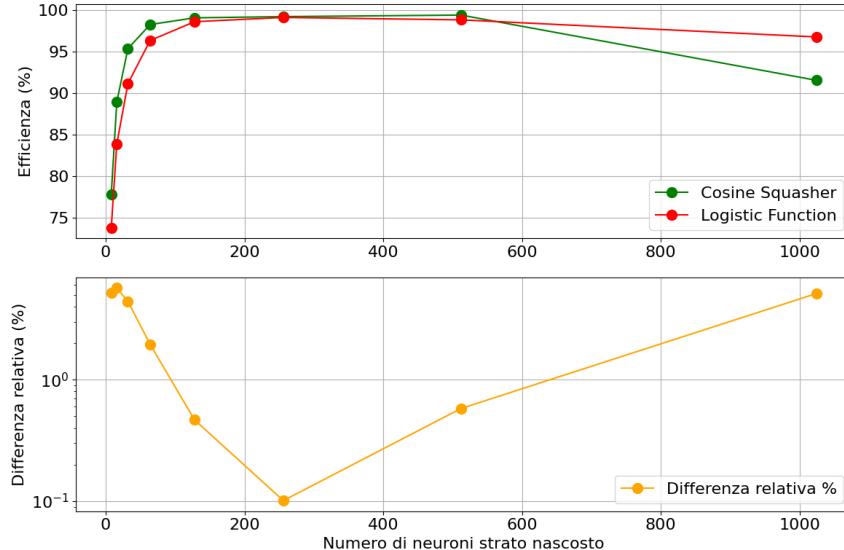


**Figura 4.6:** Andamento della loss dell'ultima epoca d'addestramento in funzione del learning rate

reti, è apprezzabile la rapida discesa della loss da parte della FNN anche nei learning rate più bassi.

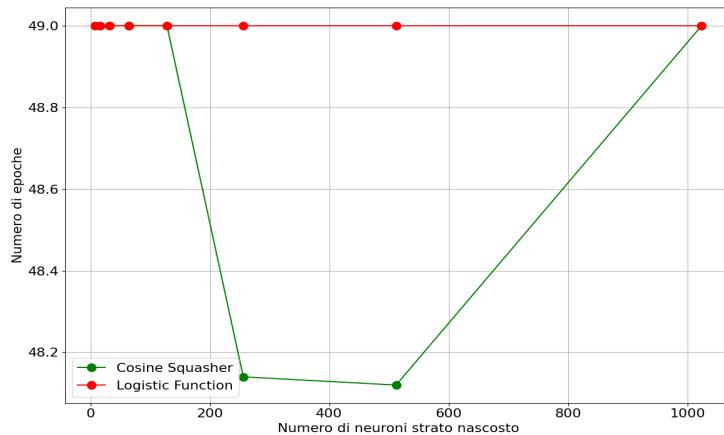
### 4.3.2 Numero di neuroni nello strato nascosto

Il secondo test parametrico viene effettuato facendo variare il numero di neuroni nello strato nascosto  $l$  tra i valori: (8, 16, 32, 64, 128, 256, 512, 1024). Esattamente come per il learning rate, anche per il numero di neuroni del *hidden layer* è apprezzabile una maggiore efficienza da parte della *Cosine Network* nella zona con un numero di neuroni molto basso. Ottenere



**Figura 4.7:** Andamento dell'efficienza della rete in funzione del numero di neuroni nello strato nascosto

un alto valore dell'efficienza anche con un numero basso di neuroni nello strato nascosto significa che la rete riesce ad estrapolare una grande quantità di informazioni anche con poche unità per manipolarle ovvero, minore è la quantità di neuroni e maggior sarà l'informazione contenuta in ogni unità, dunque ottenere un'alta efficienza significa che le informazioni "grossolane" estratte dalla rete sono comunque abbastanza efficienti da ottenere una buona approssimazione della funzione. Per quanto riguarda l'andamento ad alti valori di unità neuronali nello strato nascosto, la rete risulta meno efficiente a causa di una quantità eccessiva di informazione immagazzinata, avendo un numero di neuroni molto maggiore della dimensione del vettore input, ovvero 600. Un altro grafico molto utile è la Figura 4.8, a differenza della *Logistic Network* che non soddisfa mai il raggiungimento di un efficienza del 99.9%, condizione di uscita anticipata dall'addestramento, la rete di Gallant & White riesce ad ottenere a soddisfare la condizione di uscita in un numero inferiore di epoche prefissate per l'addestramento, nel nostro caso il valore fissato è di 50 epoche. Si ricordi che il numero di epoche è un hyper-parametro e dunque è un valore fissato che influenza direttamente sulla qualità dell'apprendimento. Si specifica che in Figura 4.8 il numero di epoche massime sia 49, questo conteggio è dovuto al fatto che Python numera gli elementi partendo dall'elemento 0.



**Figura 4.8:** Andamento del numero di epoche d'addestramento in funzione del numero di neuroni nello strato nascosto

## 4.4 Test su segnali reali

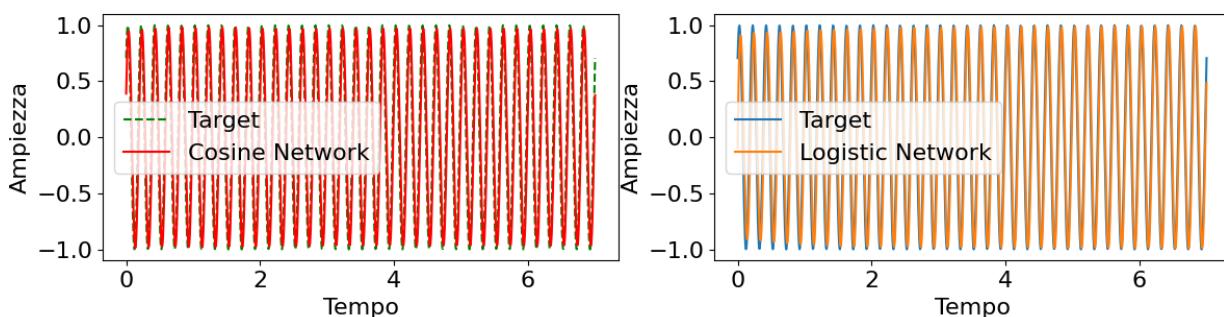
In questa sezione conclusiva riportiamo i vari test eseguiti sulla capacità di approssimare funzioni assomiglianti ad onde (impulsi) reali. Ancora una volta la rete di Gallant & White risulta essere la più efficiente e con minor richiesta di spesa computazionale.

I test sono stati eseguiti con la seguente scelta degli hyper-parametri:

- numero di neuroni nello strato nascosto: 256
- numero di epoche d'addestramento: 100
- learning rate:  $1 * 10^{-2}$
- dimensione del vettore input: 800

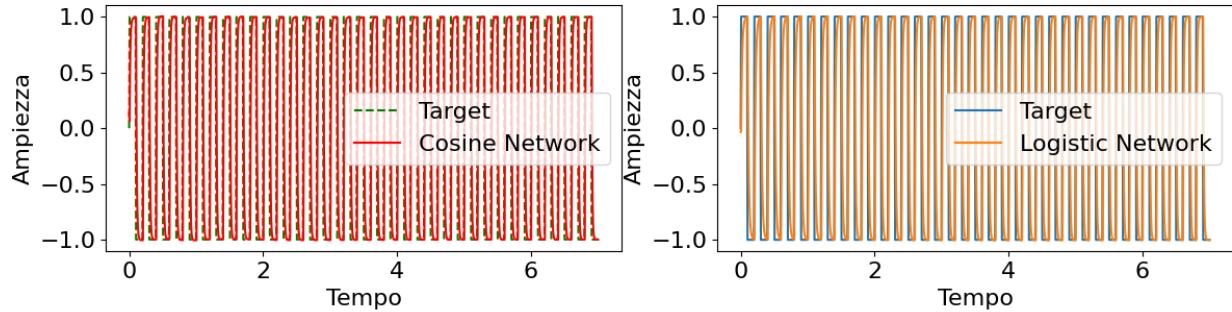
In appendice viene riportato il codice con cui sono state create le funzioni d'onda.

*Sinusoidale* : entrambe le funzioni mostrano un'ottima capacità di approssimazione. Si noti che nel caso della funzione logistica, sembra necessitare di un maggior numero di periodi per riuscire ad approssimare correttamente l'andamento della funzione.



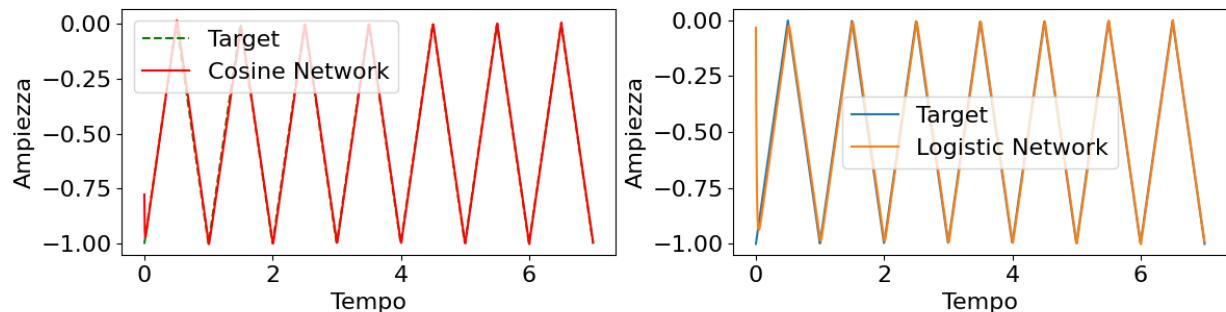
**Figura 4.9:** Confronto tra le approssimazioni di un'onda sinusoidale

*Quadra* : anche in questa casistica la *Logistic\_Network* trova maggiore difficoltà ad approssimare i picchi e le gole di un'onda quadra, mentre la rete con funzione *cosine squasher* riscontra maggior successo.



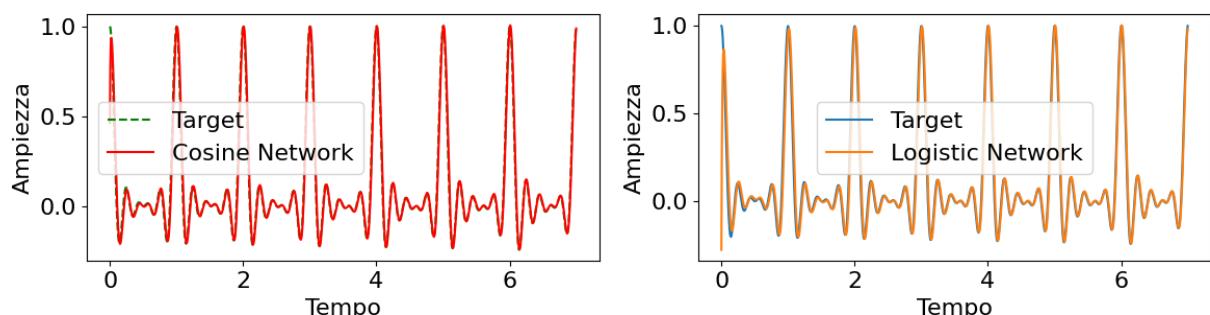
**Figura 4.10:** Confronto tra le approssimazioni di un'onda quadra

*Triangolare* : in questo caso, entrambe le reti offrono un'ottima approssimazione



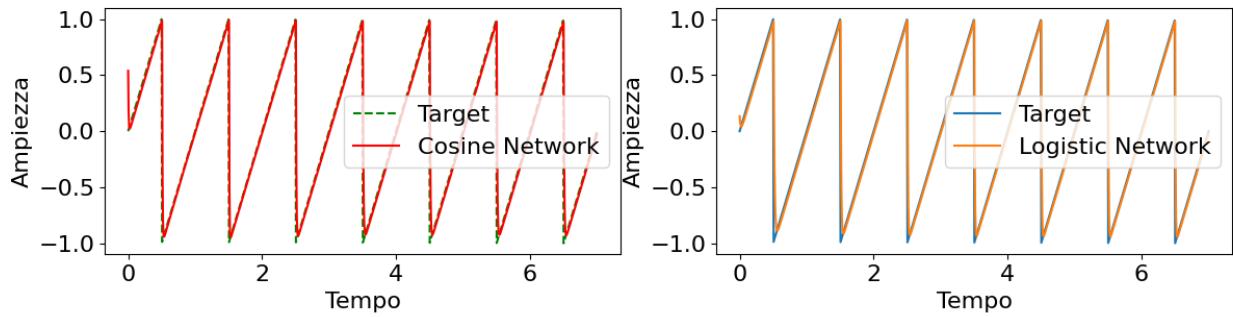
**Figura 4.11:** Confronto tra le approssimazioni di un'onda triangolare

*Treno d'impulsi* : per un treno d'impulsi la rete di Gallant & White offre un'ottima approssimazione della funzione, anche la sua controparte, nonostante si possa notare nei primi intervalli un leggero errore, propone una buona approssimazione della funzione d'onda



**Figura 4.12:** Confronto tra le approssimazioni di un treno d'impulsi

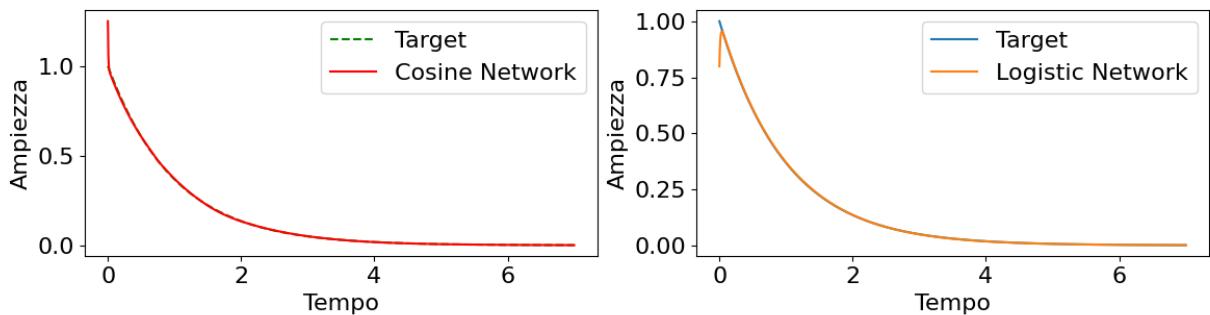
*Dente di sega* : come per la Fig. 4.10, anche in questo caso la rete con funzione logistica ha maggiore difficoltà ad approssimare le gole ed i picchi di un segnale a dente di sega.



**Figura 4.13:** Confronto tra le approssimazioni di dente di sega

### Scarica

*Condensatore* : questa è l'unica delle casistiche proposte in cui la *Logistic Network* è effettivamente più efficace ad approssimare la funzione scelta. Questo è dovuto alla mancanza di picchi e gole da parte dell'onda, se avessimo considerato anche la carica del condensatore il risultato sarebbe stato diverso.



**Figura 4.14:** Confronto tra le approssimazioni di un processo di scarica di un condensatore

Gli andamenti dell'efficienza e della funzione errore vengono riportate in ordine nell'Appendice, caso per caso.

# Capitolo 5

## Conclusione

In questa tesi sono state prese in considerazione e studiate delle reti neurali, in particolare, è stata implementata con successo la rete neurale, descritta nell'articolo di Gallant & White [4], nel linguaggio Python e addestrata su una funzione campione, ottenendo un'approssimazione simile ad una trasformata di Fourier come previsto dalla teoria.

Si è dimostrato sia teoricamente che sperimentalmente, che le FNN, pur avendo un'architettura peculiare, sono in grado di ottenere le proprietà di approssimazione delle serie di Fourier. Attraverso i test eseguiti si è dimostrato inoltre, l'efficacia di una rete feed-forward con funzione di attivazione *cosine squasher* in confronto con un'altra funzione di attivazione nello strato nascosto, la funzione logistica.

La FNN conferma quanto dimostrato da Gallant& White, ovvero l'esistenza di reti neurali capaci di approssimare funzioni in modo universale. Come è stato dimostrato dai risultati del capitolo precedente, la loro rete risulta più performante nei tempi di computazione ed efficienza, a parità di hyper-parametri selezionati e dimensione del vettore input rispetto ad una rete con funzione logistica, che invece risulta più adatta nei modelli di classificazione binaria.

L'implementazione è stata effettuata su diversi tipi di funzioni *target* dimostrando, in tutti i casi, una buona capacità di approssimazione. Avrebbe avuto maggior utilità utilizzare dei dataset reali per comprendere il reale valore della rete. I test dimensionali e parametrici sono stati ricavati iterando, e mediando, i risultati dell'apprendimento 50 volte. Sebbene il limitato numero di esecuzioni possa non essere sufficiente a dimostrare in maniera statisticamente robusta le considerazioni aprioristiche sull'architettura di rete in questione, questo lavoro tratta uno studio preliminare e si rimanda a lavori futuri ulteriori indagini delle reti qui analizzate.

Per gli sviluppi futuri sarebbe utile estendere l'approccio ad altri tipi di reti e funzioni; sviluppare l'apprendimento statistico delle FNN come suggerito da lavori successivi.

Le reti neurali possiedono effettivamente capacità computazionali intrinseche che giustificano il loro successo. L'implementazione di una FNN conferma validità e importanza dei risultati teorici pionieristici.



# Capitolo 6

## Appendice

### 6.1 Classe: *Punti*

```
1 class Punti():
2     def __init__(self, dom, epsilon):
3         self.dom = dom
4         self.epsilon = epsilon
5
6     #genera un numero di punti equidistanti nell'intervallo [epsilon, dom-epsilon]
7     def generatore(self, num):
8         step = (self.dom - self.epsilon) / num
9         result = [self.epsilon + i * step for i in range(num)]
10        return np.array([result])
```

Listing 6.1: Classe Punti

### 6.2 Funzioni d'onda

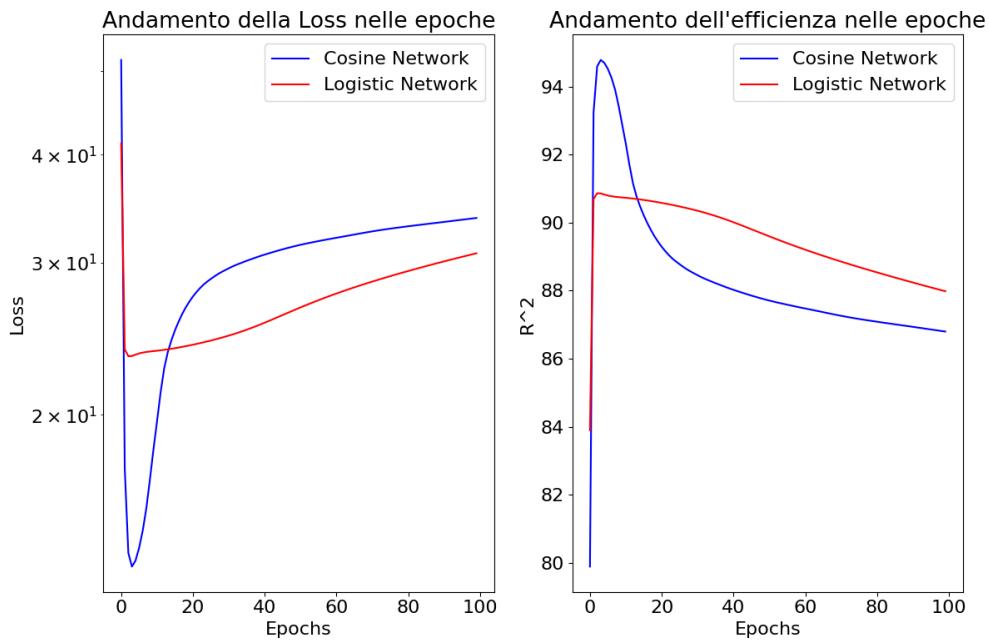
```
1 def onda_sin(t, A, f, phi):
2     return A * np.sin(2 * np.pi * f * t + phi)
3
4 def onda_quad(t, f):
5     return np.sign(np.sin(2 * np.pi * f * t))
6
7 def onda_triang(t):
8     return 2 * np.abs(t - np.floor(t + 0.5)) - 1
9
10 def treno_impulsi(t, T, tau):
11     signal = np.zeros_like(t)
12     for n in range(-10, 11):
13         signal += np.sinc((t - n * T) / tau)
14     return signal
15
16 def dente_Sega(t, T):
17     return 2 * (t / T - np.floor(0.5 + t / T))
18
19 def scarica_cond(t, R, C, V0):
20     return V0 * np.exp(-t / (R * C))
21
22 T = 1.0      # Periodo del treno di impulsi
23 tau = 0.1    # Larghezza degli impulsi
24 A = 1.0      # Ampiezza dell'onda sinusoidale
25 f = 5.0      # Frequenza dell'onda sinusoidale
```

```

26 phi = np.pi/4 # Fase dell'onda sinusoidale
27 x0 = 2.0      # Posizione dell'impulso unitario
    
```

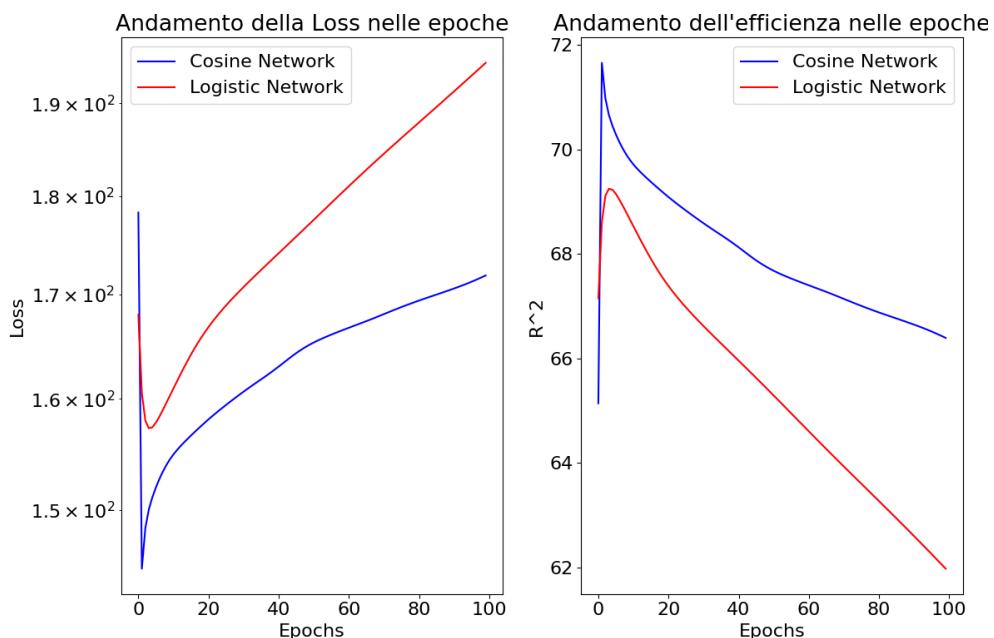
### 6.3 Confronti tra le reti su funzioni d'onda

*Sinusoidale :*



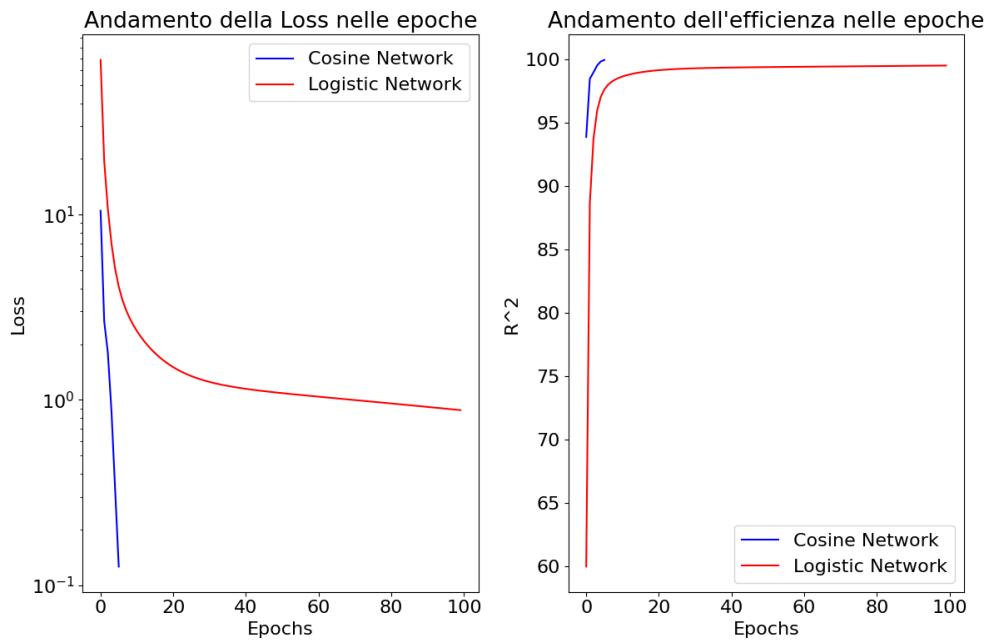
**Figura 6.1:** confronto tra l'andamento dell'efficienza e della funzione errore di un'onda sinusoidale

*Qudra :*



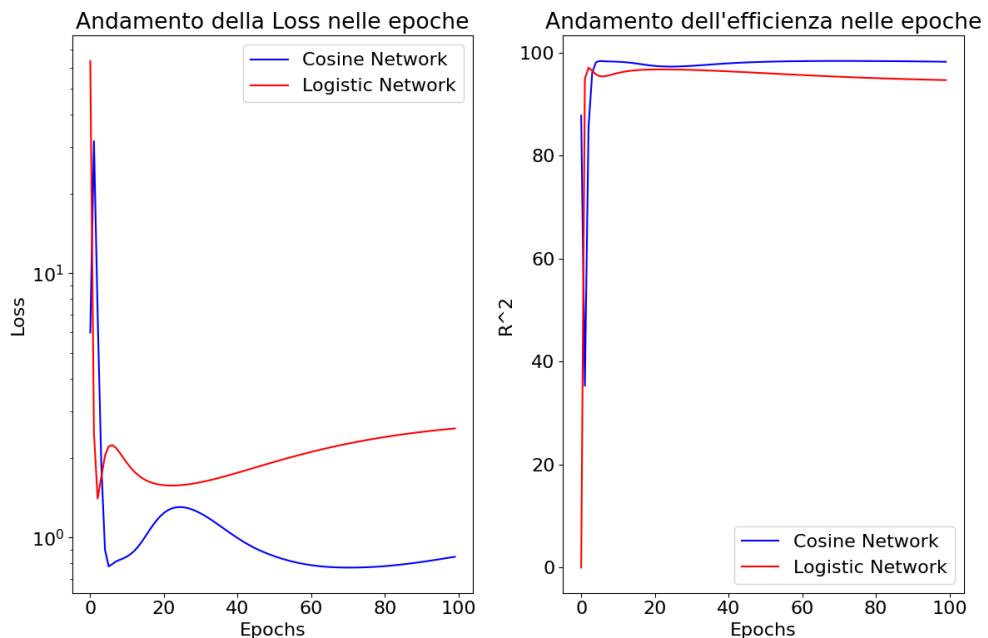
**Figura 6.2:** confronto tra l'andamento dell'efficienza e della funzione errore di un'onda quadra

*Triangolare* :



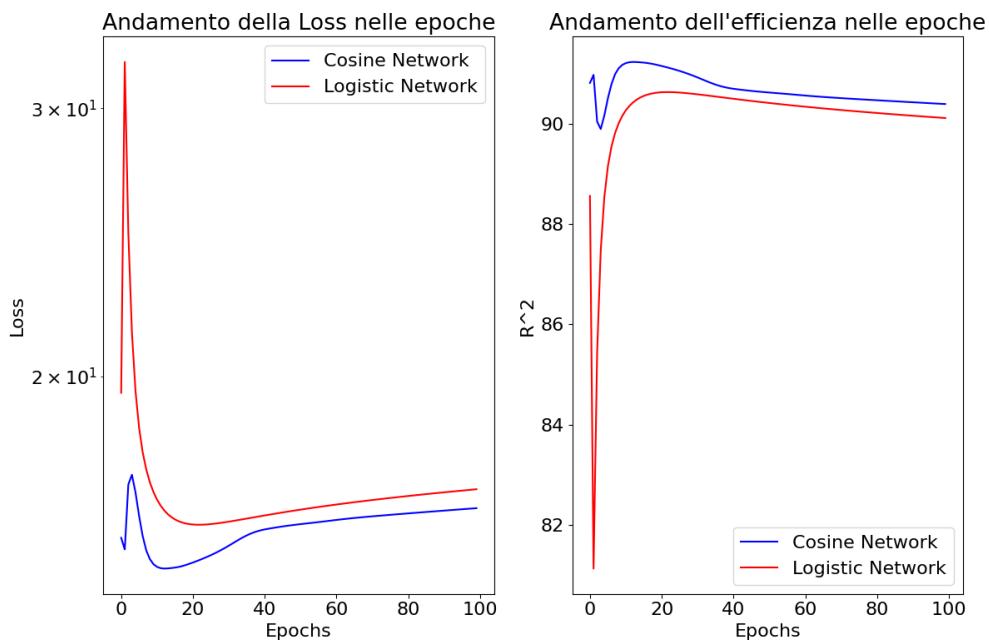
**Figura 6.3:** Confronto tra l'andamento dell'efficienza e della funzione errore di un'onda triangolare

*Treno d'impulsi* :

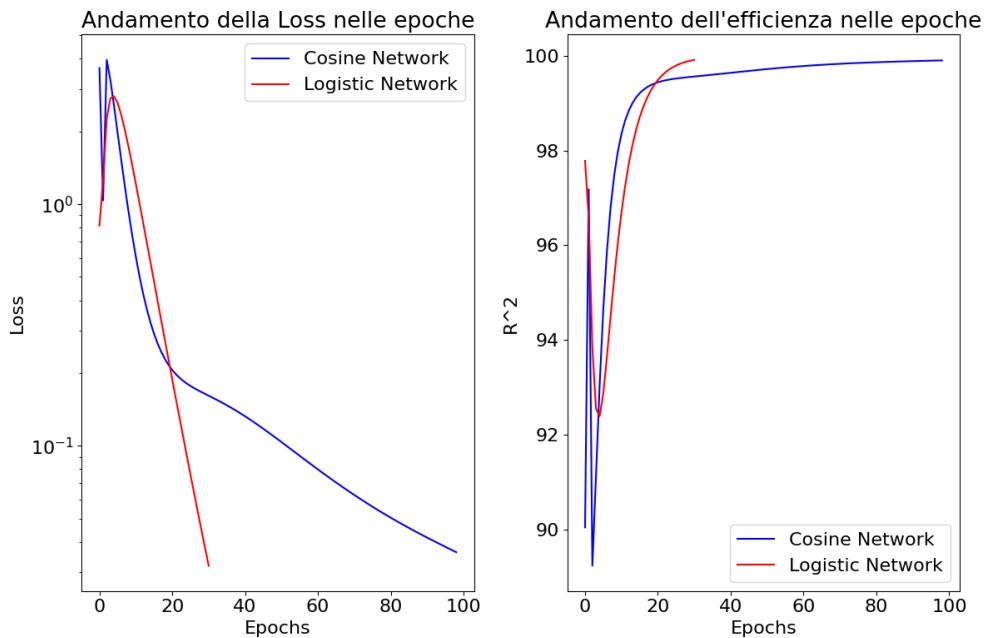


**Figura 6.4:** confronto tra l'andamento dell'efficienza e della funzione errore di un treno d'impulsi

*Dente di sega :*



**Figura 6.5:** Confronto tra l'andamento dell'efficienza e della funzione errore di un segnale a dente di sega



**Figura 6.6:** Confronto tra l'andamento dell'efficienza e della funzione errore di un processo di scarica di un condensatore

*Scarica*

# Elenco delle figure

1.1	<i>Copyright:</i> Deep Neural Network architecture used in "Deep Learning for Ligand-Based Virtual Screening in Drug Discovery" [1]	5
2.1	Struttura della rete di Gallant & White. <i>Copyright: Nicolò Gazzetta, 2023</i>	12
4.1	Confronto tra trasformata e rete neurale	20
4.2	Confronto tra Logistic e Cosine Network	20
4.3	Efficienza in funzione della dimensione con grafico della differenza relativa tra le efficienze	21
4.4	Errore in funzione della dimensione dell'input	21
4.5	Andamento dell'efficienza in funzione del learning rate	22
4.6	Andamento della loss dell'ultima epoca d'addestramento in funzione del learning rate	22
4.7	Andamento dell'efficienza della rete in funzione del numero di neuroni nello strato nascosto	23
4.8	Andamento del numero di epoche d'addestramento in funzione del numero di neuroni nello strato nascosto	24
4.9	Confronto tra le approssimazioni di un'onda sinusoidale	24
4.10	Confronto tra le approssimazioni di un'onda quadra	25
4.11	Confronto tra le approssimazioni di un'onda triangolare	25
4.12	Confronto tra le approssimazioni di un treno d'impulsi	25
4.13	Confronto tra le approssimazioni di dente di sega	26
4.14	Confronto tra le approssimazioni di un processo di scarica di un condensatore	26
6.1	confronto tra l'andamento dell'efficienza e della funzione errore di un'onda sinusoidale	30
6.2	confronto tra l'andamento dell'efficienza e della funzione errore di un'onda quadra	30
6.3	Confronto tra l'andamento dell'efficienza e della funzione errore di un'onda triangolare	31
6.4	confronto tra l'andamento dell'efficienza e della funzione errore di un treno d'impulsi	31

6.5 Confronto tra l'andamento dell'efficienza e della funzione errore di un segnale a dente di sega . . . . .	32
6.6 Confronto tra l'andamento dell'efficienza e della funzione errore di un processo di scarica di un condensatore . . . . .	32

# Bibliografia

- [1] M. Bahi e M. C. Batouche. «Deep Learning for Ligand-Based Virtual Screening in Drug Discovery». In: *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)* (2018), pp. 1–5. URL: <https://api.semanticscholar.org/CorpusID:57377648>.
- [2] G. Cicogna. *Metodi matematici della Fisica*. UNITEXT for Physics. Springer Milan, 2014. ISBN: 9788847056848. URL: <https://books.google.it/books?id=CeskBAAQBAJ>.
- [3] J. W. Cooley e J. W. Tukey. «An algorithm for the machine calculation of complex Fourier series». In: *Mathematics of computation* 19.90 (1965), pp. 297–301.
- [4] Gallant e White. «There exists a neural network that does not make avoidable mistakes». In: *IEEE 1988 International Conference on Neural Networks*. 1988, 657–664 vol.1. DOI: 10.1109/ICNN.1988.23903.
- [5] R. Guo et al. «Physics Embedded Deep Neural Network for Solving Volume Integral Equation: 2-D Case». In: *IEEE Transactions on Antennas and Propagation* 70.8 (2022), pp. 6135–6147. DOI: 10.1109/TAP.2021.3070152.
- [6] C. R. Harris et al. «Array programming with NumPy». In: *Nature* 585.7825 (set. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [7] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Psychology Press, 2002. ISBN: 9781410612403.
- [8] R. Hecht-Nielsen. «Kolmogorov's Mapping Neural Network Existence Theorem». In: 1987. URL: <https://api.semanticscholar.org/CorpusID:118526925>.
- [9] J. Hertz, A. Krogh e R. Palmer. *Introduction To The Theory Of Neural Computation*. Addison-Wesley Computation and Neural Systems Series. Avalon Publishing, 1991. ISBN: 9780201515602. URL: [https://books.google.it/books?id=dI2rDnN\\_eZEC](https://books.google.it/books?id=dI2rDnN_eZEC).
- [10] K. Hornik, M. Stinchcombe e H. White. «Multilayer feedforward networks are universal approximators». In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8). URL: <https://www.sciencedirect.com/science/article/pii/0893608089900208>.
- [11] A. N. Kolmogorov. «On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition». In: *Doklady Akademii Nauk*. Vol. 114. 5. Russian Academy of Sciences. 1957, pp. 953–956.

- [12] W. S. McCulloch e W. Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133. DOI: 10.1007/BF02478259.
- [13] Y.-H. Pao e Y. Takefuji. «Functional-link net computing: theory, system architecture, and functionalities». In: *Computer* 25.5 (1992), pp. 76–79. DOI: 10.1109/2.144401.
- [14] O. Rippel, J. Snoek e R. P. Adams. *Spectral Representations for Convolutional Neural Networks*. 2015. arXiv: 1506.03767 [stat.ML].
- [15] F. Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain». In: *Psychological review* (1958), pp. 386–408.
- [16] D. E. Rumelhart, G. E. Hinton e R. J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323 (1986). DOI: 10.1038/323533a0.
- [17] O. Sallam e M. Fürth. «On the use of Fourier Features-Physics Informed Neural Networks (FF-PINN) for forward and inverse fluid mechanics problems». In: *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment* 237.4 (2023), pp. 846–866. DOI: 10.1177/14750902231166424.
- [18] J. Sietsma e R. J. Dow. «Creating artificial neural networks that generalize». In: *Neural Networks* 4.1 (1991), pp. 67–79. ISSN: 0893-6080. DOI: [https://doi.org/10.1016/0893-6080\(91\)90033-2](https://doi.org/10.1016/0893-6080(91)90033-2). URL: <https://www.sciencedirect.com/science/article/pii/0893608091900332>.
- [19] A. Silvescu. «Fourier neural networks». In: *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*. Vol. 1. 1999, 488–491 vol.1. DOI: 10.1109/IJCNN.1999.831544.
- [20] W. D. Tan K. «Learning Complex Spectral Mapping with Gated Convolutional Recurrent Networks for Monaural Speech Enhancement.» In: *PubMed* 28 (2020), pp. 380–390. DOI: 10.1109/taslp.2019.2955276.
- [21] H. White. «The case for conceptual and operational separation of network architectures and learning mechanisms». In: *Discussion Paper* (1988).