UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Magistrale in Fisica e Astronomia: Fisica Teorica dei Sistemi Complessi

Parametrizzazione Spettrale Generalizzata per Reti Neurali Feedforward

Generalized Spectral Parametrization for Feedforward Neural Networks

Candidato

Gianluca Peri (Matricola 7075549)

Relatore

Prof. Duccio Fanelli

Correlatore

Dott. Lorenzo Giambagli

Anno Accademico 2023/2024

Parametrizzazione Spettrale Generalizzata per Reti Neurali Feedforward
Master Thesis. Università degli Studi di Firenze

Questa tesi è stata composta con LaTeX e la classe UniFiTh.

Versione: 1 giugno 2025

Sito web: https://github.com/gianluca-peri/Spectral_Learning_WSLC_Thesis

Email dell'autore: gianluca.peri@edu.unifi.it

Dedicato
al Prof. Duccio Fanelli e al Dott. Lorenzo Giambagli, per avermi supportato,
alla mia famiglia, per avermi sopportato,
e ad Alessandro e Gabriele, per aver fatto entrambe le cose.

Indice

Introduzione

In questa tesi tratteremo temi inerenti la branca dell'intelligenza artificiale: un settore di studio multidisciplinare che ha come obbiettivo ultimo l'automazione delle attività intellettuali. Il respiro di ciò che può essere considerato attività intellettuale umana è senza dubbio ampio, ed abbraccia ambiti diversi e variegati, dalla gestione di banali pratiche quotidiane, come far di conto alla cassa, fino ai più complessi esercizi di astrazione, che hanno come oggetto d'indagine lo studio dei fenomeni naturali. Circa settant'anni fa l'avvento dei moderni calcolatori programmabili (*computer*) rese plausibile la possibilità di automatizzare alcuni di tali processi, codificando nella macchina una dettagliata lista di operazioni da svolgere per portare a termine il compito assegnato. Questo approccio algoritmico nel corso del tempo si è tuttavia rivelato inefficace per l'automazione di attività intellettuali complesse, ed al giorno d'oggi il metodo più utilizzato per sviluppare nuove forme di intelligenza artificiale consiste nel lasciare che la macchina apprenda in autonomia, sulla base della sua esperienza; questo approccio è noto in letteratura come *apprendimento automatico* (*machine learning*) [29]. Le intelligenze artificiali frutto di questo processo si sono spesso rivelate estremamente efficienti, e nel corso degli ultimi anni hanno permeato l'ambito accademico ed industriale. Di seguito citiamo alcuni esempi di applicazioni.

- **Individuazione di nuovi esopianeti:** *ExoMiner* è un intelligenza artificiale addestrata a riconoscere i segnali di transito di un pianeta davanti alla sua stella madre, il suo impiego ha permesso di validare 301 nuovi esopianeti. La capacità di impiegare un sistema automatico per l'analisi di dati astronomici è cruciale, poiché i dati raccolti sono spesso troppi per essere analizzati manualmente nella loro interezza [23, 4].

- **Predizione di struttura proteica:** *AlphaFold* è un altro esempio di intelligenza artificiale, questa volta addestrata a predire la struttura tridimensionale di una proteina sulla base della sua sequenza di amminoacidi. Questo compito riveste una cruciale importanza in biologia, poiché la struttura di una proteina determina la sua funzione [18].

- **Creazione di assistenti artificiali:** *GPT-4* è addestrato a generare testi, e può essere impiegato per scrivere articoli, rispondere a domande, o svolgere compiti di traduzione. Intelligenze generative di questo tipo hanno ampie potenzialità: ad esempio *Copilot*, un'intelligenza artificiale addestrata per aiutare i programmatori nello sviluppo di codice, è basata proprio su *GPT-4* [42].

Figura I: Lee Se-dol, uno dei migliori giocatori al mondo di Go nel 2016, dopo la famigerata mossa 37 di AlphaGo (precursore di AlphaZero). Se-dol perse la partita, ed a seguire commentò: *"I thought AlphaGo was based on probability calculation, and it was merely a machine. But when I saw this move I changed my mind. Surely AlphaGo is creative. This move was really creative and beautiful"* [11].

- **Ottenimento di performance sovrumane in ambiti ristretti:** *AlphaZero* è un giocatore artificiale di scacchi, go e shōgi. Esso ha ottenuto performance sovrumane in tutti e tre i giochi semplicemente tramite l'esperienza ottenuta giocando contro se stesso, ed attualmente non esiste un giocatore umano in grado di batterlo con frequenza apprezzabile [21].

- **Sviluppo di sistemi di guida autonoma:** *Autopilot* è un'intelligenza artificiale addestrata a guidare un'automobile in autonomia, e viene impiegata da *Tesla* per i suoi veicoli. Sebbene attualmente essa non sia completamente funzionale, la sua parziale viabilità è stata dimostrata da numerosi test su strada. Si capisce come un completo sviluppo di questa tecnologia potrebbe rivoluzionare l'interezza del settore dei trasporti [5].

Nel presente lavoro tratteremo una delle metodologie attualmente più utilizzate per consentire ad una macchina di apprendere: l'implementazione al calcolatore di una rete neurale artificiale. Questa tecnica consiste nel simulare all'interno di un computer un modello di rete neurale biologica, che è in grado di modulare le attività neurali sulla base di input esterni, confrontare i prodotti della propria elaborazione con i risultati desiderati, e modificare di conseguenza il flusso d'informazione che si propaga al suo interno (si veda la Figura II) [29]. Queste reti sono alla base di tutte le applicazioni menzionate in precedenza. Date le enormi potenzialità la teoria sottende al funzionamento delle reti neurali è in costante sviluppo. Si ricercano in particolare nuovi metodi per migliorare le performance delle reti addestrate, cercando al contempo di evidenziare i meccanismi fondamentali responsabili per il loro funzionamento. In questo contesto si colloca il *formalismo spettrale*, un approccio allo studio di queste reti basato sulla teoria dei grafi e degli operatori lineari, che nel corso degli ultimi anni ha portato ad alcuni interessanti sviluppi teorici [16].
In questa tesi ci proponiamo di generalizzare il formalismo spettrale ad architetture neurali complesse, e ponendo le basi per nuove tecniche di addestramento per reti neurali profonde. Nello specifico la tesi è organizzata come di seguito illustrato.
Nel primo capitolo è fornita un'introduzione all'apprendimento automatico ed alle reti neurali artificiali. Questa trattazione si conclude con l'esposizione del formalismo spettrale come attualmente descritto in letteratura. Nel corso del secondo capitolo il formalismo spettrale viene parzialmente generalizzato, consentendo l'applicazione dello stesso ad architetture neurali più complesse, e discutendo i vantaggi connessi alla generalizzazione proposta. Alla fine del capitolo vengono riportati i risultati

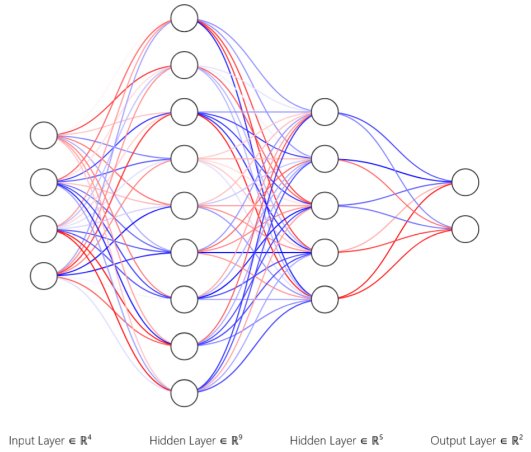

Figura II: Rappresentazione grafica di una semplice rete neurale artificiale feedforward a 4 strati (*layers*). I cerchi rappresentano i neuroni, mentre le linee le sinapsi tra di essi; la diversa colorazione delle linee indica la diversa intensità delle connessioni. Il flusso d'informazione attraverso la rete scorre da sinistra verso destra. (Immagine generata con *NN-SVG* [7]).

di test numerici svolti: essi supportano le considerazioni teoriche svolte nel corso del capitolo e dimostrano la funzionalità del nuovo approccio proposto. Nel terzo capitolo il formalismo spettrale viene completamente generalizzato, consentendo la descrizione di reti con profondità ed architettura arbitraria. Nel quarto ed ultimo capitolo si riassumono i principali risultati ottenuti, e si indicano possibili futuri sviluppi.

Capitolo 1

Reti Neurali Artificiali

1.1 Apprendimento automatico

1.1.1 Introduzione all'intelligenza artificiale

I propose to consider the question, 'Can machines think?' Con queste parole Alan Turing nel 1950 apre il suo articolo seminale *Computing Machinery and Intelligence*, e sebbene a più di settant'anni di distanza non sia stato ancora raggiunto un consenso su questo tema al giorno d'oggi il settore dell'AI (*Artificial Intelligence*) è in rapidissimo sviluppo [44]. Basti pensare che *ChatGPT*, il celebre chatbot rilasciato da *OpenAI* il 30 novembre 2022 è considerato come il prodotto software dalla più rapida crescita della storia, accumulando una *userbase* di circa 100 milioni di utenti in meno di due mesi [41].
Il percorso che ha condotto dall'articolo di Turing ai moderni modelli GPT (*Generative Pre-Trained Transformer*) è lungo e tortuoso, ed è di fatto impossibile ripercorrerlo in questa sede con dovizia di dettagli. Ci limiteremo nel seguito a proporre alcune note storiche fondamentali per inquadrare il tema nella cornice di contesto.
Inizialmente l'intelligenza artificiale mirava ad affrontare e risolvere problemi difficili da trattare per gli esseri umani ma relativamente semplici per un computer, nello specifico problemi che possono essere descritti formalmente in maniera chiara e completa [29]. Un esempio di questo tipo di problemi è il gioco degli scacchi: l'universo del gioco, completamente avulso dall'ambiente esterno, è composto da 64 caselle albergate da un set preciso di pezzi, che si muovono seguendo regole ben specifiche. Contesti di questo tipo si prestano particolarmente bene all'analisi di un sistema automatico, e difatti nel 1997 *Deep Blue*, celebre supercomputer della IBM,[1] battè Garry Kasparov, il campione del mondo di scacchi dell'epoca. Deep Blue è un classico esempio di quella che oggi viene chiamata GOFAI (*Good Old Fashioned AI*), ovvero una forma di intelligenza artificiale che per funzionare si basa su un set di nozioni e regole di inferenza espresse logicamente in qualche tipo di linguaggio formale. Nello specifico Deep Blue è stato programmato con un set di regole atte alla valutazione della bontà di una posizione, regole sviluppate dai programmatori stessi (e da un team di scacchisti) [33]. Tuttavia questo approccio all'intelligenza artificiale,

[1] *Supercomputer* per gli standard dell'epoca.

noto anche in letteratura come *symbolic AI*, fallisce quando applicato ad altre classi di problemi; ed ironicamente ha difficoltà proprio con quei problemi che tendono ad essere banali da risolvere per un essere umano. Un esempio particolarmente illuminante è quello di *Cyc*: un sistema di intelligenza artificiale sviluppato a partire dal 1984 presso MCC (*Microelectronics and Computer Technology Corporation, USA*); Cyc consiste in un *inference engine* ed in un database di proposizioni logiche espresse in linguaggio formale. L'idea alla base del progetto è quella di equipaggiare Cyc con un insieme di proposizioni logiche riguardo il nostro mondo che sia abbastanza esteso da consentire all'*inference engine* di iniziare a trarre le giuste conclusioni quando posto innanzi a situazioni di varia natura, creando una sorta di *buon senso* artificiale. Sebbene questo approccio allo sviluppo dell'intelligenza artificiale nel corso degli anni abbia fornito qualche risultato promettente le difficoltà derivanti da esso sembrano spesso superare gli aspetti positivi: Cyc spesso fallisce nel comprendere semplici storie di vita quotidiana, ad esempio è celebre l'episodio riferito all'analisi, da parte di Cyc, di un contesto ordinario di un uomo intento a farsi la barba con un rasoio elettrico. L'inference engine trovava inconsistente la scena. Il sistema sapeva, dal database di proposizioni, che gli esseri umani non contengono componenti elettriche, ma dato che l'uomo stava tenendo in mano un rasoio elettrico Cyc era anche giunto alla conclusione che l'uomo, nel mentre che si stava radendo, potesse essere considerato un entità composta anche dal rasoio elettrico, il che era in contrasto con le sue conoscenze sulla natura umana (un uomo non contiene componenti elettriche) [29, 36].

Un approccio GOFAI a questo tipo di contesti *quotidiani* sembra richiedere una completa risoluzione della branca dell'ontologia per iniziare ad essere abbordabile, eppure un essere umano non ha bisogno di simili competenze filosofiche per comprendere storie di vita quotidiana, il che ci porta a sospettare che possa esistere un metodo migliore, o quantomeno un metodo più semplice, per trattare questo tipo di problemi. Un paradigma promettente in questo senso è l'*apprendimento automatico*, noto nella letteratura italiana anche con il termine inglese *machine learning*; questo è l'approccio che sta dietro al recente nuovo *boom* dell'intelligenza artificiale.

Per introdurre il machine learning facciamo un altro esempio canonico di problema difficile da risolvere per un computer ma ironicamente facile da risolvere per un essere umano: il riconoscimento di immagini. Nello specifico concentriamoci sul dataset MNIST, che contiene immagini 28×28 di cifre scritte a mano [25].

Figura 1.1. Alcuni esempi di immagini contenute in MNIST (Modified National Institute of Standards and Technology database). Esso contiene decine di migliaia di immagini, in bianco e nero, di cifre scritte a mano da studenti delle scuole superiori ed impiegati dell'ufficio censimento USA (immagine estratta da [6]).

Per un essere umano classificare correttamente queste immagini è solitamente semplice, con performance che si aggirano attorno al 99.8%, tuttavia risulta estremamente difficile scrivere un algoritmo GOFAI che funzioni bene per questo task: non solo

una cifra può essere scritta ad altezze diverse nell'immagine ma soprattutto cose come la forma della cifra, gli angoli fra i tratti, la dimensione degli elementi grafici, possono variare di molto fra immagini appartenenti alla stessa categoria [26]. Queste difficoltà sono un sintomo della natura del problema: il riconoscimento di immagini opera in un contesto che non è descritto da un piccolo numero di regole rigide, e questo rende l'approccio GOFAI di fatto impraticabile.
Abbiamo proposto un esempio di difficile *problema di classificazione*, facciamo adesso un esempio di difficile *problema di regressione*; un problema di regressione ha come obbiettivo la predizione di un valore numerico invece che di una categoria. Nello specifico concentriamoci sul celebre dataset *California Housing*, che contiene informazioni inerenti appartamenti situati nell'area californiana (USA), fra cui il loro valore di mercato (prezzo) [30]. L'obbiettivo in questo contesto è solitamente quello di predire il prezzo di un immobile sulla base di tutte le altre informazioni presenti nel dataset, come ad esempio

- La densità demografica in prossimità dell'immobile;
- Il numero di stanze;
- Il numero di camere da letto;
- Etc.

Abbiamo quindi una collezione di input numerici ed il nostro obbiettivo è trovare la funzione f (l'algoritmo) che sulla base di essi fornisca una buona stima del prezzo della casa

$$f : I \to \mathbb{R}^+ \tag{1.1}$$

dove con I vogliamo indicare l'*input space*, ovvero l'insieme di tutti i possibili input che la nostra funzione può ricevere. In questo contesto l'approccio GOFAI al problema sarebbe quello di cercare direttamente di trovare la forma di una funzione f che abbia buone performance, magari basando la ricerca della forma corretta sulle opinioni di esperti del settore o su studi presenti in letteratura sul tema; approccio analogo a quello che è stato effettivamente implementato per sviluppare Deep Blue. Tuttavia ci sono ragioni per crede che questa strategia non sia ottimale se applicata ad un problema come quello posto da *California Housing*, e si preferisce in questo contesto, come anche in MNIST, utilizzare tecniche di *machine learning*. Il machine learning è un approccio *indiretto* all'AI, in esso non si cerca di trovare direttamente la funzione (algoritmo) f, bensì si cerca di implementare un *algoritmo di apprendimento* A che funzioni sulla base del dataset D che sottende il problema in esame; nello specifico l'algoritmo A si comporterà come una funzione che colleghi lo spazio di tutti i possibili dataset $\mathcal{D}$ inerenti il nostro problema all'insieme di tutte le possibili funzioni f ($\mathcal{F}$) tali da poter trattare il problema

$$A : \mathcal{D} \to \mathcal{F}, \tag{1.2}$$

lo spazio $\mathcal{F}$ è noto in letteratura con il nome di *spazio delle ipotesi*. A prima vista trovare un buon algoritmo di apprendimento A può sembrare più arduo che trovare

direttamente la forma di f, tuttavia all'atto pratico per la maggior parte dei problemi che hanno a che fare con un contesto ampio e realistico è vero il contrario: l'approccio del *machine learning* è di gran lunga il più facile.

1.1.2 Algoritmi di apprendimento

Per iniziare la trattazione di questo tema operiamo nel contesto dei problemi di **regressione**. L'idea che sta alla base di ogni algoritmo di apprendimento è innanzitutto quella di considerare una famiglia di funzioni la cui forma è caratterizzata da un set di parametri $w_i \in \mathbb{R}$, solitamente chiamati *pesi*:

$$f(\boldsymbol{w}, \boldsymbol{x}) : I \to \mathbb{R} \tag{1.3}$$

dove $\boldsymbol{x}$ è il vettore degli input, e si capisce che I è sempre lo spazio di tutti i possibili input. La funzione f, scelto un set di pesi $\boldsymbol{w}$, connetterà il vettore degli input al valore in output, che costituirà la nostra predizione. Questo *framework* sposta il problema dalla corretta scelta della funzione f alla corretta scelta del valore dei pesi w_i; il prezzo da pagare per questa concretizzazione del problema è la riduzione dello spazio delle ipotesi esplorabile: la forma di $f(\boldsymbol{w}, \boldsymbol{x})$ al variare di $\boldsymbol{w}$ esprime lo spazio di tutte le funzioni che potremo prendere in considerazione come possibili soluzioni del problema di regressione; ad esempio se scegliessimo una f del tipo:

$$f(\boldsymbol{w}, w_0, \boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 \tag{1.4}$$

ci staremmo limitando a considerare solo modelli lineari (questa scelta prende il nome di *regressione lineare*). Scelta la nostra famiglia di funzioni consideriamo l'output prodotto dalla funzione f a fronte di uno specifico input x. L'output così ottenuto verrà confrontato, ricorrendo ad una opportuna metrica, al valore in uscita atteso. Questo approccio ovviamente è perseguibile solo se sappiamo quale y dovrebbe essere prodotta da uno specifico $\boldsymbol{x}$; fortunatamente datasets come *California Housing* ci forniscono questa informazione: essi hanno come elementi delle coppie $(\boldsymbol{x}_i, y_i)$, coppie di input e corretto output associato, con i che enumera gli elementi del dataset. Questo tipo di datasets in letteratura vengono chiamati *datasets per l'apprendimento supervisionato* (*supervised learning*). Per confrontare la nostra *predizione* $\hat{y}_i$ al valore y_i che l'output dovrebbe avere definiamo una misura di distanza fra i due valori; questa misura è nota come *funzione di loss* L. La scelta più banale per la funzione di loss è semplicemente il *valore assoluto della differenza fra* $\hat{y}_i$ *e* y_i:

$$L(\hat{y}_i, y_i) = |\hat{y}_i - y_i|; \tag{1.5}$$

tuttavia per ragioni legate alla teoria dell'apprendimento, in cui non ci addentreremo in questa sede,[2]. Spesso si preferisce prendere come loss la *Mean Square Error loss* (*MSE loss*):

[2]Di seguito una brevissima esposizione delle ragioni a cui abbiamo fatto cenno: innanzitutto la (1.5) presenta un punto di non derivabilità in zero; di contro la (1.6) è derivabile in ogni punto, e risulta quindi preferibile (molte tecniche di *machine learning* richiedono che la loss sia derivabile). In secondo luogo si può dimostrare che sotto opportune condizioni la *MSE loss* emerga spontaneamente dall'applicazione del principio di massima verosimiglianza, il che la rende maggiormente desiderabile e funzionale in diversi contesti. In ultimo luogo la MSE loss gode dei risultati esposti nel *Teorema di Gauss-Markov* [9]

$$L(\hat{y}_i, y_i) = (\hat{y}_i - y_i)^2; \tag{1.6}$$

ed ancora, per ragioni analitiche, alle volte è addirittura preferita la seguente

$$L(\hat{y}_i, y_i) = \frac{1}{2}(\hat{y}_i - y_i)^2. \tag{1.7}$$

Il punto cruciale sul quale qui ci soffermiamo è la possibilità di misurare la distanza fra y e $\hat{y}$. Tale distanza può essere ridotta selezionando il miglior set di pesi $\boldsymbol{w}$, ovvero determinando la funzione f che risolve il problema in questione.
Per essere più precisi una volta che abbiamo definito la nostra funzione di loss quello che vogliamo fare è sfruttarla per definire una nuova quantità: il *rischio* $\mathcal{R}$ associato alla $f(\boldsymbol{w}, \boldsymbol{x})$, ovvero il *valore d'aspettazione* della loss quando si utilizzi la f per effettuare le predizioni $\hat{y}$:

$$\mathcal{R} \doteq E_p\Big[L(f(\boldsymbol{w}, \boldsymbol{x}), y)\Big] = \int L(f(\boldsymbol{w}, \boldsymbol{x})), y) p(\boldsymbol{x}, y) d\boldsymbol{x} dy. \tag{1.8}$$

La minimizzazione di questo rischio consentirebbe di trovare i migliori possibili valori per i pesi $\boldsymbol{w}$, e dunque la miglior possibile funzione soluzione del problema di regressione (all'interno del nostro spazio delle ipotesi). Questo approccio tuttavia richiederebbe la conoscenza della distribuzione di probabilità $p(\boldsymbol{x}, y)$ che sottende il problema, funzione ignota nella maggior parte dei casi. Questo problema può essere aggirato, con qualche compromesso, sfruttando il dataset di apprendimento supervisionato D a disposizione: invece di considerare il rischio $\mathcal{R}$, e dunque il valore di aspettazione della loss, possiamo considerare il *rischio empirico* R, definito non su tutte le possibili coppie $(\boldsymbol{x}, y)$ ma bensì su tutte le coppie contenute nello specifico dataset D, e definito come:

$$R_f = \frac{1}{N} \sum_{i=1}^{N} L(f(\boldsymbol{w}, \boldsymbol{x}_i), y_i) \tag{1.9}$$

dove N è il numero di elementi del dataset D. Questo ci consente di mettere in pratica la nostra idea per trovare i pesi; il prezzo da pagare è che in generale la bontà dei pesi che troveremo dipenderà da quanto il dataset è rappresentativo del contesto in esame. All'atto pratico la stima dei corretti pesi sarà data da:

$$\hat{w} = \underset{w}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L(f(\boldsymbol{w}, \boldsymbol{x}_i), y_i). \tag{1.10}$$

Si capisce che la procedura appena descritta equivale a operare una sorta di *fit* sui dati contenuti nel dataset; stiamo apprendendo la corretta forma della funzione risolutiva sulla base delle sue performance sui dati a nostra disposizione. All'atto pratico tuttavia rimane il problema di come trovare i pesi $\hat{w}$ tali da minimizzare R sul dataset: esistono diversi metodi per svolgere l'$argmin$, ma quello che viene in generale più spesso utilizzato è il **metodo della discesa del gradiente**. Con questo metodo l'idea è partire considerando una funzione $\tilde{f} \equiv f(\tilde{\boldsymbol{w}}, \boldsymbol{x})$ dove i pesi $\tilde{w}_i$ sono stati scelti a caso (il più ragionevolmente possibile), armati di $\tilde{f}$ possiamo calcolare il gradiente del rischio empirico rispetto ai pesi

$$\Big[\boldsymbol{\nabla}_{\boldsymbol{w}} R_f\Big]_{f\equiv\tilde{f}} = \begin{pmatrix} \frac{\partial}{\partial w_1} R_{\tilde{f}} \\ \frac{\partial}{\partial w_2} R_{\tilde{f}} \\ \vdots \\ \frac{\partial}{\partial w_N} R_{\tilde{f}} \end{pmatrix} \tag{1.11}$$

e con questo gradiente troviamo una versione migliorata dei pesi rispetto al *guess* iniziale $\tilde{w}$ tramite la seguente:

$$w = \tilde{w} - \gamma\Big[\boldsymbol{\nabla}_{\boldsymbol{w}} R_f\Big]_{f\equiv\tilde{f}} \tag{1.12}$$

dove γ è un parametro che ci dice quanto ampio dev'essere questo passo di ottimizzazione; questo parametro è noto come *tasso di apprendimento* (*learning rate*). Si capisce che questo processo può essere *iterato*, ed a ogni iterazione, dato che stiamo letteralmente *discendendo il gradiente*, ci avvicineremo al valore dei pesi $\hat{w}$ desiderato. Questo processo di ottimizzazione iterativa nel contesto del machine learning viene spesso chiamato *addestramento* (*training*).
Dato che il metodo funziona ugualmente con o senza il fattore $1/N$ nell'espressione del rischio empirico possiamo anche omettere la normalizzazione, assumendo quindi di minimizzare il valore della loss calcolata su tutto il dataset.

MSE loss e Maximum Likelihood Estimation
In questo inserto vogliamo approfondire le motivazioni che sottendono la scelta della *Mean Square Error loss* come funzione di loss per problemi di regressione. Dato un modello $f(\boldsymbol{w}, \boldsymbol{x})$ e un dataset D di apprendimento supervisionato potremmo identificare i valori ottimali dei parametri $\boldsymbol{w}$ come quei valori tali da massimizzare la probabilità di osservare i dati in D sotto il modello f. Questo approccio, concettualmente diverso da quello esposto nelle precedenti pagine, è noto in letteratura come *Maximum Likelihood Estimation* (*MLE*). Si noti che la *MLE* non fa uso della *loss function*, e dunque rimuove il problema di dover scegliere per essa una forma appropriata. I parametri ottimali $\boldsymbol{w}$ saranno quindi dati da:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmax}} \prod_{i=1}^{N} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}) \tag{1.13}$$

questa affermazione fa leva sulle seguenti assunzioni:

- il target y_i è generato da una distribuzione di probabilità condizionata $p(y_i|\boldsymbol{x}_i, \boldsymbol{w})$, non stiamo quindi a priori assumendo che y_i sia perfettamente determinato da $\boldsymbol{x}_i$;
- le osservazioni $(\boldsymbol{x}_i, y_i)$ presenti nel dataset D sono *iid* (*independent and identically distributed*).

L'argomento dell'argmax viene spesso chiamato *likelihood function* $L_D(\boldsymbol{w})$:

$$L_D(\boldsymbol{w}) = \prod_{i=1}^{N} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}) \tag{1.14}$$

Per fare progressi sotto questa prospettiva occorre ipotizzare una forma per la distribuzione condizionata $p(y_i|\boldsymbol{x}_i, \boldsymbol{w})$. Una scelta comune è quella di assumere che y_i sia distribuito gaussianamente attorno alla predizione del nostro modello $f(\boldsymbol{w}, \boldsymbol{x}_i)$, ovvero:

$$p(y_i|\boldsymbol{x}_i, \boldsymbol{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(- \frac{(y_i - f(\boldsymbol{w}, \boldsymbol{x}_i))^2}{2\sigma^2} \right). \tag{1.15}$$

Questa ipotesi spesso si rivela ragionevole; notiamo ad esempio che nel caso in cui i dati presenti nel dataset siano medie di osservazioni multiple l'ipotesi di gaussianità è sorretta dal *Teorema del Limite Centrale* [10].
Possiamo interpretare il fatto di dover ipotizzare una forma per la distribuzione di probabilità $p(y_i|\boldsymbol{x}_i, \boldsymbol{w})$ come il prezzo da pagare per non doversi preoccupare della scelta della funzione di loss.
Sotto l'ipotesi di distribuzione di probabilità gaussiana per y_i possiamo riscrivere la *likelihood function* come:

$$\begin{aligned} L_D(\boldsymbol{w}) &= \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(- \frac{(y_i - f(\boldsymbol{w}, \boldsymbol{x}_i))^2}{2\sigma^2} \right) = \\ &= \frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left(- \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - f(\boldsymbol{w}, \boldsymbol{x}_i))^2 \right) \end{aligned} \tag{1.16}$$

da ora in avanti, per alleggerire la notazione, indicheremo la predizione del nostro modello $f(\boldsymbol{w}, \boldsymbol{x}_i)$ con $\hat{y}_i$. Adesso possiamo notare che massimizzare la likelihood è equivalente a massimizzare il logaritmo della likelihood, spesso chiamato *log-likelihood*:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmax}} \log L_D(\boldsymbol{w}) = \underset{\boldsymbol{w}}{\operatorname{argmax}} \left(-\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \right). \tag{1.17}$$

Il termine $1/2\sigma^2$ davanti alla sommatoria è irrilevante ai fini della massimizzazione, ed inoltre dato il segno meno possiamo interpretare l'argmax come il seguente argmin:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2. \tag{1.18}$$

Ma da quest'ultima espressione si evince immediatamente che la massimizzazione della likelihood è equivalente alla minimizzazione della *Mean Square Error loss*! Questo motiva la scelta di *MSE* come funzione di loss per problemi di regressione nei casi in cui sia ragionevole assumere che i target siano distribuiti gaussianamente attorno alle predizioni del modello.

Alcuni problemi tuttavia non si prestano bene all'approccio descritto nelle precedenti

pagine semplicemente per il fatto di non essere *regressivi* ma bensì **classificativi**, come il problema esposto da MNIST. In questi problemi ad un input $\boldsymbol{x}_i$ non è associato un valore $y_i \in \mathbb{R}$ ma bensì una *classe*. In un dataset per apprendimento supervisionato si dice anche che i dati $\boldsymbol{x}_i$ sono *etichettati* da un'etichetta (*label*) y_i. Potremmo pensare di ricondurre il problema della classificazione ad un problema di regressione semplicemente numerando le classi, in questa maniera predire l'appartenenza ad una classe equivarrebbe a predire un numero $n \in \mathbb{N} \subset \mathbb{R}$. Questo approccio tuttavia non funziona affatto bene poiché prevede l'introduzione di un ordinamento arbitrario delle classi, assunzione a priori che può riflettersi negativamente sull'intero meccanismo di training.
Per risolvere questo problema nel contesto del machine learning esistono numerose strategie: una di queste, applicabile alla **classificazione binaria**, prevede di interpretare $f(\boldsymbol{w}, \boldsymbol{x})$ come la funzione che descrive la *superficie di separazione* fra le classi [31]. Questo approccio può poi essere promosso alla classificazione multi-classe utilizzando *più di una funzione*, ovvero più di una superficie di separazione. La teoria che sottende questo argomento è ampia e non banale, dunque in questa sede preferiamo sorvolare sull'argomento per concentrarci su una strategia di costruzione dei classificatori più semplice da esporre e più facile da generalizzare ai contesti di nostro interesse, ovvero la classificazione multi-classe e la classificazione per mezzo di *reti neurali artificiali*, che vedremo in seguito.
Nel contesto della classificazione possiamo prendere in esame un set di funzioni non più del tipo:

$$f : I \to \mathbb{R} \tag{1.19}$$

ma nella forma:

$$f : I \to \mathbb{R}^K \tag{1.20}$$

con K numero delle classi nel contesto in esame. Fornito l'input $\boldsymbol{x}$ la funzione produrrà dunque un vettore[3] $\hat{\boldsymbol{y}}' \in \mathbb{R}^K$, vettore che verrà poi fatto passare attraverso una particolare funzione

$$S : \mathbb{R}^K \to \mathbb{R}^K$$

denominata *funzione di softmax*:

$$\hat{y}_i = S(\hat{\boldsymbol{y}}')_i = \frac{e^{\hat{y}'_i}}{\sum_{j=1}^{K} e^{\hat{y}'_j}}. \tag{1.21}$$

Questa funzione ha l'effetto di normalizzare il vettore $\hat{\boldsymbol{y}}'$ in maniera tale che possa essere interpretato come una *distribuzione di probabilità discreta* sulle K classi. L'output di $f(\boldsymbol{w}, \boldsymbol{x}_i)$ potrà quindi essere facilmente confrontato con la label y_i, considerando che anche y_i può essere interpretata come una distribuzione di probabilità discreta sulle classi: se ad esempio l'input $\boldsymbol{x}_5$ avesse nel dataset una label y_5 che indicasse la sua appartenenza alla *terza categoria* questo potrebbe essere rappresentato pensando alla label come a un vettore $\boldsymbol{y}_5 \in \mathbb{R}^K$ con la seguente forma:

[3] In letteratura, per ragioni che non abbiamo il tempo di approfondire in questa sede, il vettore $\hat{\boldsymbol{y}}'$ pre-normalizzazione è anche noto come *vettore dei logits*.

$$\boldsymbol{y}_5 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{1.22}$$

ovvero come un vettore normalizzato che associ probabilità 1 alla classe corretta e dunque probabilità 0 a tutte le altre classi. Questa rappresentazione della categoria di appartenenza prende anche il nome di *one hot encoding*. Per misurare la distanza fra l'output di f e il *target* $\boldsymbol{y}$ come nel caso precedente abbiamo bisogno di definire una funzione di loss, che questa volta dovrà avere come argomenti due vettori appartenenti a $\mathbb{R}^K$ anziché due valori scalari. Una delle loss più utilizzate in questo contesto è la *log-loss*, anche chiamata *cross entropy loss*:

$$L(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i) = -\frac{1}{K} \sum_{j=1}^{K} y_j \log \hat{y}_j \tag{1.23}$$

o più semplicemente:

$$L(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i) = -\sum_{j=1}^{K} y_j \log \hat{y}_j. \tag{1.24}$$

Si noti che l'ordine delle distribuzioni di probabilità discreta all'interno della formula è importante: nello specifico è la nostra predizione della distribuzione di probabilità

$$\hat{\boldsymbol{y}}_i = S(f(\boldsymbol{w}, \boldsymbol{x}_i)) \tag{1.25}$$

che dovrebbe apparire nella formula per la log-loss all'interno del logaritmo, e la label $\boldsymbol{y}$ dovrebbe stare all'esterno, mai il contrario. Ovviamente avremmo anche potuto scegliere forme differenti per la funzione di loss: ad esempio avremmo potuto scegliere di misurare la distanza fra il vettore target $\boldsymbol{y}_i$ e il vettore predetto da f attraverso l'applicazione della *MSE loss* a tutti gli elementi dei vettori; questa scelta sarebbe stata funzionale ma probabilmente non ottimale: esiste un corpo di teoria che indica come ottimale la scelta della *cross entropy loss* per problemi di classificazione, sotto opportune condizioni [29]. Approfondiremo il tema nel seguente inserto.

Softmax e Cross Entropy Loss

Questo inserto mira a motivare la scelta di utilizzare *softmax* e *cross entropy loss* per problemi di classificazione multi-classe come esposto nelle precedenti pagine. Le ragioni teoriche che sottendono questa scelta hanno a che fare, come nell'inserto precedente, con la prospettiva della *Maximum Likelihood Estimation*. Dato il contesto in cui ci troviamo, ovvero quello di classificazione multi-classe (K classi), possiamo interpretare il target (*label*) y_i come un vettore $\boldsymbol{y}_i \in \mathbb{R}^K$ normalizzato, o più precisamente come un vettore avente una singola entrata pari ad 1 e tutte le altre pari a 0 (*one hot encoding*). Alla luce di questo

sceglieremo un modello $f(\boldsymbol{w}, \boldsymbol{x})$ che produca in output un vettore $\hat{\boldsymbol{y}}' \in \mathbb{R}^K$; inoltre è spontaneo richiedere, data la forma del target, che l'output di f sia normalizzato; questo può essere ottenuto includendo a valle del modello la funzione *softmax*:

$$S(\boldsymbol{y})_i \doteq \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}. \tag{1.26}$$

L'output del nostro modello avrà quindi forma:

$$\hat{\boldsymbol{y}}_i = S(\hat{\boldsymbol{y}}')_i = S\big(f(\boldsymbol{w}, \boldsymbol{x}_i)\big). \tag{1.27}$$

Passiamo adesso alla stima dei parametri $\boldsymbol{w}$ del modello attraverso la massimizzazione della *likelihood function* $L_D(\boldsymbol{w})$:

$$L_D(\boldsymbol{w}) = \prod_{i=1}^{N} p(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{w}) \tag{1.28}$$

si noti che esistono solo K possibili vettori $\boldsymbol{y}_i$ (si ricordi la codifica *one hot encoding*).
Come nell'inserto precedente per fare progressi dobbiamo adesso ipotizzare una forma per la distribuzione di probabilità condizionata $p(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{w})$. Dato che stiamo trattando un problema di classificazione multi-classe la distribuzione di probabilità condizionata sarà discreta, ed è ragionevole aspettarsi che essa, per i K possibili vettori $\boldsymbol{y}_i$, sia una *distribuzione di Bernoulli multivariata (distribuzione categorica)*:

$$p(\boldsymbol{y}_i = \boldsymbol{h}_k | \boldsymbol{x}_i, \boldsymbol{w}) = p_k \quad , \quad p_k \mid \sum_{i=1}^{K} p_k = 1. \tag{1.29}$$

dove con $\boldsymbol{h}_k$ vogliamo indicare la codifica *one hot encoding* della classe k-esima. Ricordiamo adesso che nell'inserto precedente abbiamo assunto che i target y_i fossero distribuiti *attorno alla predizione del modello* $\hat{y}_i$; in analogia con questo assumeremo adesso che i target $\boldsymbol{y}_i$ siano distribuiti attorno alla predizione normalizzata del modello, ovvero:

$$p(\boldsymbol{y}_i = \boldsymbol{h}_k | \boldsymbol{x}_i, \boldsymbol{w}) = [\hat{\boldsymbol{y}}_i]_k \tag{1.30}$$

dove $[\hat{\boldsymbol{y}}_i]_k$ vuole indicare il k-esimo elemento del vettore normalizzato $\hat{\boldsymbol{y}}_i$, output del modello sotto input $\boldsymbol{x_i}$; con questo stiamo di fatto interpretando l'output del modello come una distribuzione di probabilità discreta sulle K classi.
Notiamo che la distribuzione di Bernoulli multivariata può adesso essere riscritta nella seguente forma più comoda:

$$p(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{w}) = \prod_{k=1}^{K} [\hat{\boldsymbol{y}}_i]_k^{[\boldsymbol{y}_i]_k}. \tag{1.31}$$

difatti tutti i fattori provenienti da classi diverse da quella corretta saranno 1 a causa dell'*one hot encoding*. Questa riscrittura ci permette di scrivere comodamente la likelihood function come:

$$L_D(\boldsymbol{w}) = \prod_{i=1}^{N} \prod_{k=1}^{K} [\hat{\boldsymbol{y}}_i]_k^{[\boldsymbol{y}_i]_k}. \tag{1.32}$$

I parametri ottimali $\hat{\boldsymbol{w}}$ saranno dunque dati da:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmax}}\, L_D(\boldsymbol{w}) = \underset{\boldsymbol{w}}{\operatorname{argmax}} \prod_{i=1}^{N} \prod_{k=1}^{K} [\hat{\boldsymbol{y}}_i]_k^{[\boldsymbol{y}_i]_k}. \tag{1.33}$$

Passiamo adesso alla massimizzazione della *log-likelihood*:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmax}} \log L_D(\boldsymbol{w}) = \underset{\boldsymbol{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \sum_{k=1}^{K} [\boldsymbol{y}_i]_k \log[\hat{\boldsymbol{y}}_i]_k \tag{1.34}$$

E possiamo adesso passare all'argmin con l'aggiunta di un segno meno:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(- \sum_{k=1}^{K} [\boldsymbol{y}_i]_k \log[\hat{\boldsymbol{y}}_i]_k \right). \tag{1.35}$$

Nell'ultima espressione possiamo riconoscere la minimizzazione della *cross entropy loss* (1.23) su tutto il dataset D.
Abbiamo quindi mostrato che l'uso della *cross entropy loss* è motivato dalla *Maximum Likelihood Estimation* sotto l'assunzione che i target siano distribuiti secondo una distribuzione di Bernoulli multivariata attorno alla predizione del modello, predizione che dovrà dunque essere normalizzata attraverso la funzione *softmax*. Si noti che altri metodi di normalizzazione, diversi dalla *softmax*, potrebbero causare problemi: ad esempio normalizzare attraverso una *hard-max*, ovvero ponendo ad 1 nel vettore di output la posizione corrispondente alla massima entrata e a 0 tutte le altre, potrebbe causare problemi durante la fase di training a causa della comparsa di punti di non derivabilità.

Arrivati a questo punto ci possiamo finalmente riconnettere con la teoria illustrata con riferimento al caso regressivo. Con la loss possiamo calcolare il rischio empirico sul dataset, e minimizzarlo attraverso il metodo della discesa del gradiente per determinare i pesi ottimali.

1.1.3 Train, Test, Validation

Dobbiamo notare che una volta concluso l'addestramento del nostro modello è necessario testarlo per valutarne le performances; difatti non ci possiamo accontentare di ottenere un basso valore della loss sul dataset utilizzato per l'addestramento, in quanto il modello (ovvero la f) potrebbe semplicemente aver appreso come riprodurre i dati presenti nel dataset, ovvero potrebbe aver *memorizzato* il dataset, senza esser diventato idoneo ad un utilizzo generale per il *task* di interesse; in particolare ci si deve guardare dal problema del *sovra-adattamento* (*overfitting*), condizione che comporta l'aver acquisito una struttura idonea alla minimizzazione della loss

associata allo specifico dataset usato per l'addestramento, senza però disporre di adeguate caratteristiche di generalizzazione [9, 22, 31]. Per verificare che il modello abbia appreso correttamente il suo compito solitamente i datasets sono muniti di una porzione non adibita al training, che viene chiamata *porzione di test*; una volta conclusa la fase di training si smette di ottimizzare i pesi del modello e lo si fa girare sulla porzione di test, mai elaborata durante il train, per verificare che il modello sia correttamente in grado di operare su situazioni che non ha mai *visto prima*. Come misura delle performances in fase di test solitamente si usa semplicemente il valore della loss sul dataset di test (*test loss*).
Chiarito questo punto dobbiamo infine notare che esiste un'altra porzione separata dei datasets per machine learning di cui non abbiamo ancora parlato: la porzione di validazione (*validation set*). Questa occorre per trattare un importante problema in machine learning, ovvero la scelta degli iperparametri del modello, come ad esempio il *learning rate*. Per cercare di ottenere il modello migliore possibile è necessario cercare anche valori ottimali per gli iperparametri, non ci possiamo permettere di sceglierli a caso, e dunque solitamente quello che si fa è addestrare tanti modelli sullo stesso *training set*, ognuno con diversi valori degli iperparametri; una volta in possesso dei modelli addestrati si passa alla misura delle loro performances sul *validation set* (e non sul *test set* come detto prima), ovvero su un insieme di esempi mai visti in fase di train; il migliore dei modelli sul validation set viene infine testato sul test set, ovvero su altri esempi mai visti, per ottenere una misura finale delle performances.
Si capisce come questo procedimento possa risultare computazionalmente dispendioso, e difatti quando possibile si evita di mettere in atto la fase di validazione, limitandosi ad effettuare train e test.

1.2 Reti Neurali

1.2.1 Introduzione alle Reti Neurali

Nel corso della precedente sezione non abbiamo affrontato il tema di come scegliere la forma della funzione $f(\boldsymbol{w}, \boldsymbol{x})$, ovviamente la sua forma esplicita dipenderà in parte dal valore dei pesi, ma si capisce che essa dipenderà anche da come scriviamo f, ad esempio scegliendo:

$$f(\boldsymbol{w}, \boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} \tag{1.36}$$

f sarà necessariamente una funzione lineare, a prescindere dal valore che i pesi w_i assumeranno durante la fase di training. In machine learning è spesso consigliabile scegliere una forma di f tale da poter esprimere una funzione non lineare, in quanto questo estende di molto lo spazio delle ipotesi del modello, e permette quindi di risolvere una classe più ampia e complessa di problemi. Esistono molti modi per scrivere una funzione f non lineare, ad esempio uno dei più gettonati prende il nome di GLM (*Generalized Linear Models*); anche in questo caso la teoria che sottende l'argomento è estremamente ampia, quindi in questa sede scegliamo di sacrificare l'esposizione del contesto generale per giungere direttamente a quello che ci interessa davvero, ovvero la trattazione delle reti neurali artificiali. Scopriremo che nel contesto

del machine learning una rete neurale rappresenta nient'altro che un buon modo per scrivere la funzione $f(\boldsymbol{w}, \boldsymbol{x})$.

Il primo esempio di rete neurale artificiale può essere considerato il **percettrone** [40]. La sua prima formulazione fu sviluppata nel 1943 da *Warren McCulloch & Walter Pitts*, e nel nostro contesto moderno consiste nella scelta della seguente funzione $f(\boldsymbol{w}, \boldsymbol{x})$:

$$f(\boldsymbol{w}, w_0, \boldsymbol{x}) = \theta(\boldsymbol{w}^T \boldsymbol{x} + w_0) \tag{1.37}$$

dove spesso w_0 è chiamato *bias* b, e dove θ rappresenta la *funzione theta di Heaviside*. Possiamo notare che la funzione θ rende lo spazio delle ipotesi non lineare, e dunque la sua presenza è cruciale, tuttavia vediamo anche che questo limita i possibili output del modello a $\{0, 1\}$, e dunque questa struttura diviene utilizzabile al più per la risoluzione di problemi di classificazione binaria. Inoltre si capisce come la θ non si presti molto bene alle tecniche di addestramento che abbiamo esposto nella sezione precedente, in quanto ha derivata nulla ovunque eccetto che in zero, dove la derivata non è definita; questo ovviamente non è ottimale per l'implementazione del metodo della discesa del gradiente, o per altri simili metodi di ottimizzazione iterativi. Per queste ragioni nelle implementazioni moderne di strutture simili a quella del percettrone si preferisce scrivere la f facendo uso di un *rilassamento* della non-linearità θ, ed il più banale è la *funzione sigmoide* (anche chiamata *funzione sigmoidea*, o *funzione logistica*):

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{1.38}$$

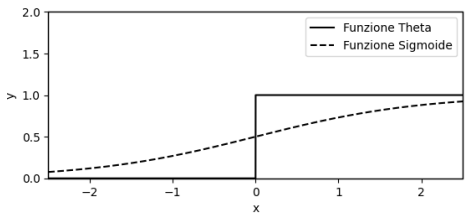

Figura 1.2. Grafico che mette a confronto la forma della funzione θ di Heaviside con la funzione sigmoide. Si noti come la seconda possa essere interpretata come un *rilassamento* della prima.

Ad ogni modo in questo contesto quello che davvero ci interessa è capire in quale senso la funzione (1.37) possa essere vista come l'implementazione di una rete neurale artificiale: ebbene la struttura della (1.37) può essere interpretata come esposto in Figura 1.3.

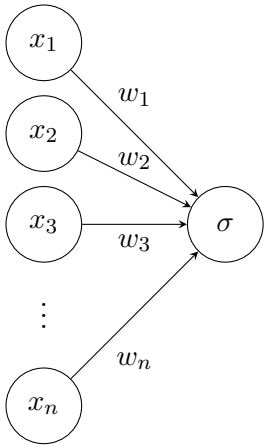

Figura 1.3. Rappresentazione grafica di un percettrone. Possiamo notare come il vettore degli input $\boldsymbol{x}$ sia interpretabile come le *attivazioni neurali* del primo strato (*layer*) di neuroni in figura; si può poi pensare che le attivazioni neurali si propaghino attraverso la rete, in maniera *feedforward*, passando per i canali di connessione pesati rappresentati in figura. Quando la somma di tutti i trasferimenti delle attivazioni arriva sul neurone del secondo layer esso applica una non-linearità al segnale, nota come *funzione di attivazione* σ (ed opzionalmente somma un *bias* $b \equiv w_0 \in \mathbb{R}$).

Giunti a questo punto potremmo dilungarci nell'esporre le similitudini e le differenze fra strutture del tipo (1.37) e quello che realmente sappiamo accadere in una rete neurale biologica; tuttavia questo argomento non è davvero rilevante per gli scopi di questa tesi; quello che ci interessa è semplicemente notare che *esiste* una similitudine fra strutture del tipo (1.37) e la struttura di una rete neurale biologica, e che molta della nomenclatura presente in letteratura fa leva su questo fatto.

1.2.2 Deep Learning

La struttura esposta nella (1.37) ed in Figura 1.3 è una rete neurale a due layers, per ovvie ragioni chiamati *layer di input* e *layer di output*, ma dovrebbe apparire a questo punto ovvio che questa struttura possa essere estesa per ammettere più strati neurali densamente connessi; inoltre si capisce che anche il numero dei neuroni di output può essere aumentato, permettendo di generalizzare ad una f capace di classificazione multi-classe. Una generica rete neurale artificiale avrà dunque la struttura esposta nella Figura 1.2.2.

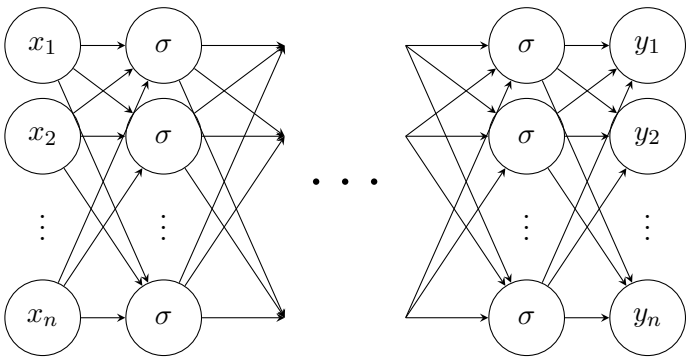

Figura 1.4. Rappresentazione di una generica rete neurale *feedforward*. Ogni neurone può essere associato ad un valore $a \in \mathbb{R}$ noto come *attivazione del neurone*. I neuroni del primo layer vengono attivati da una codifica vettoriale dell'input, successivamente le attivazioni si propagano attraverso i canali pesati.

In Figura 1.2.2 abbiamo rappresentato una generica rete neurale *feedforward*. Il suo funzionamento è quasi completamente analogo alla rete a soli due layers trattata nella sezione precedente: le attivazioni del primo layer, che indicheremo con $a_i^{[1]}$ sono fornite dall'input $\boldsymbol{x}$, e tutte le attivazioni degli strati neurali successivi sono date dalle attivazioni del layer neurale immediatamente più a monte dalla seguente:

$$a_i^{[j]} = \sigma\left(\sum_{k=1}^{N_{j-1}} w_{ik} a_k^{[j-1]} + b_i^{[j]}\right) \tag{1.39}$$

dove con N_{j-1} vogliamo indicare il numero di neuroni nel $j-1$-esimo layer, con w_{ik} il peso sul canale che connette il neurone k-esimo nel layer $j-1$ al neurone i-esimo nel layer j, e con $b_i^{[j]}$ il bias dell'i-esimo neurone del j-esimo layer. Dato che le connessioni sono in generale assunte *dense*, ovvero ogni neurone è connesso ad ogni altro neurone immediatamente a valle, capiamo che in questo contesto generale è conveniente pensare ai pesi non più in termini di vettore, ma usando piuttosto il formalismo matriciale. Una singola matrice di pesi W codificherà per le connessioni fra due layer neurali adiacenti. Notiamo inoltre che in questa struttura generale i layer di input ed output sono in qualche modo layers *speciali*, nel senso che a nessuno dei due è associata una funzione σ, che invece è associata ai neuroni di ogni altro layer (*hidden layers*); in letteratura la funzione σ prende il nome di *funzione di attivazione*. Esistono più funzioni di attivazione comunemente utilizzate, e nel seguito sceglieremo di lavorare con una delle più celebri, ovvero la funzione *ReLU* (*Rectified Linear Unit*), esposta in Figura 1.5.

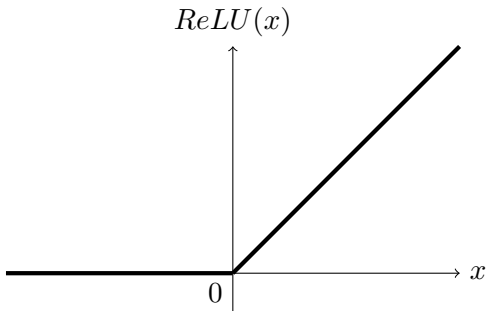

Figura 1.5. Grafico della funzione ReLU (Rectified Linear Unit).

Analiticamente:

$$ReLU(x) = \begin{cases} x & se\ x > 0 \\ 0 & se\ x \leq 0 \end{cases} \tag{1.40}$$

Le attivazioni di un intero strato neurale *profondo* (*hidden*), che indicheremo con $\boldsymbol{a}_i$, saranno date da:

$$\boldsymbol{a}_i = \sigma\Big(W_{i,i-1}\boldsymbol{a}_{i-1} + \boldsymbol{b}_i\Big) \tag{1.41}$$

dove con $W_{i,i-1}$ vogliamo indicare la matrice di pesi associata alle connessioni fra l'i-esimo strato neurale e quello immediatamente precedente; e dove con $\boldsymbol{b}_i$ vogliamo indicare il vettore dei bias associati allo strato neurale i-esimo (in generale ogni neurone può avere il suo bias b). Capiamo dunque che la funzione $f(\boldsymbol{w}, \boldsymbol{x})$ associata

ad una generica rete neurale profonda feedforward ad H *hidden layers*, priva di biases, può esser scritta nella seguente maniera:

$$f(\boldsymbol{w}, \boldsymbol{x}) \equiv \hat{\boldsymbol{y}} = W_{H+2,H+1} \circ \left(\bigcirc_{i=1}^{H} \sigma \circ W_{i+1,i} \right) \boldsymbol{x} \tag{1.42}$$

dove con $\bigcirc$ si indica la composizione di funzioni

$$\bigcirc_{i=1}^{N} f_i \doteq f_1 \circ f_2 \circ \ldots \circ f_N. \tag{1.43}$$

La (1.42) definisce il nostro spazio delle ipotesi, esplorabile al variare delle matrici $W_{i+1,i}$; compreso questo il machine learning con reti neurali profonde (*deep learning*) è concettualmente del tutto analogo a quanto trattato nei casi più semplici della sezione scorsa.
Si noti un dettaglio importante: nel caso in cui la rete sia adibita alla classificazione l'output della rete verrà sempre fatto passare attraverso la già menzionata *funzione softmax*; dunque in questo caso scriviamo che:

$$f(\boldsymbol{w}, \boldsymbol{x}) \equiv \hat{\boldsymbol{y}} = S \left[W_{H+2,H+1} \circ \left(\bigcirc_{i=1}^{H} \sigma \circ W_{i+1,i} \right) \boldsymbol{x} \right]; \tag{1.44}$$

mentre nel caso in cui la rete si occupi di regressione la predizione $\hat{y}$ sarà semplicemente fornita dalla (1.42), dunque senza l'applicazione della softmax.
Arrivati a questo punto tuttavia non abbiamo chiarito le motivazioni che stanno dietro a questa scelta di f; ci stiamo basando unicamente su una flebile similitudine della struttura scelta ad un cervello biologico o c'è altro a motivare? Ebbene innanzitutto imitare la struttura di un cervello biologico potrebbe di per sé essere considerata una buona idea: dopotutto il cervello umano è capace di produrre manifestazioni di vera *intelligenza generale*, fra l'altro con estensioni spaziali e costi energetici ridotti. In secondo luogo si dimostra che una rete neurale artificiale di adeguate dimensioni è un *interpolatore universale*, ovvero è capace di riprodurre una qualsiasi funzione continua con precisione arbitraria (formalmente si richiede anche che la funzione sia definita su un insieme compatto); questo è valido addirittura per reti con un solo hidden layer, ammesso che esso abbia un numero adeguato di neuroni [19]. Dato questo fatto viene da chiedersi come mai si scelga spesso di lavorare con reti neurali profonde, ovvero reti con un numero significativo di hidden layers; ebbene questo ha a che fare con il terzo punto cruciale, ovvero che le reti neurali permettono di fare **feature learning**: nel seguito intendiamo chiarire il significato di questa affermazione.
Le performances di un processo di learning dipendono non solo da scelte come la forma di f o l'algoritmo di ottimizzazione ma anche dal modo in cui sono presentati i dati da cui il sistema deve apprendere. Si pensi alla classificazione di MNIST: il compito sarebbe molto più semplice per un computer se fosse fornita una *descrizione* dell'immagine anziché i pixel dell'immagine stessa; informazioni come la presenza di loops, la presenza di tratti diagonali, il numero di tratti, etc, permetterebbero di classificare le immagini di MNIST molto più agevolmente. Allo stesso modo si capisce come datasets quali California Housing sarebbero molto più ardui da apprendere se al posto di una descrizione tabulare degli immobili fosse fornito il

plico di tutto il *paperwork* burocratico inerente ad essi. In generale dato un input x, contenente tutta l'informazione necessaria per la classificazione (o la regressione), le informazioni cruciali spesso tendono a non essere esplicitamente presenti, e vanno invece ricostruite; queste caratteristiche determinanti dell'input, come ad esempio il numero di tratti orizzontali in un'immagine di MNIST, in letteratura prendono il nome di *features*. Dato un problema come quello posto da MNIST dunque ci potremmo inventare una qualche sorta di metodo automatico per estrarre le features dall'immagine, ed in effetti questa strategia può funzionare, applicando per esempio delle *maschere* (vedi ad esempio *Viola & Jones*) [45]. Tuttavia ci possiamo rendere conto che questo approccio, sebbene applicato nel contesto del machine learning, ha le stesse problematiche che avevamo identificato con riferimento all'approccio GOFAI: sviluppare un set di regole per capire quali features sono rilevanti per un processo di classificazione, e capire come estrarle in generale, è spesso così complicato da rendere impossibile una specifica formale da parte di un essere umano. In linea con l'approccio del machine learning dovremmo lasciare che sia il modello a determinare *se* e *come* estrarre *quali* features. L'estrazione delle features deve quindi essere perseguita con un modello di machine learning, il cui output sarà dato in pasto ad un altro modello di machine learning per la classificazione vera e propria, ma questo è esattamente quello che si ottiene mettendo insieme 3 strati neurali! Il passaggio dal primo al secondo estrae le features, mentre il passaggio dal secondo al terzo classifica. Potremmo però imbatterci in problemi complessi nei quali è molto ottimistico aspettarsi che un modello di machine learning riesca ad estrarre immediatamente le features rilevanti, magari perché esse probabilmente sono molto complicate; ebbene in questo caso possiamo pensare alle features rilevanti come a loro volta estraibili da features meno rilevanti, e dunque il processo di estrazione di features è iterabile, e questo spiega come mai in molte situazioni si preferisca usare reti neurali profonde: ogni passaggio delle attivazioni neurali idealmente serve ad estrarre features, via via sempre più complesse, fino ad arrivare a features abbastanza rilevanti da poter essere utilizzate direttamente per la classificazione (o la regressione) [29].

Questa è una delle idee chiave che stanno alla base dell'utilizzo delle reti neurali profonde. Tuttavia questo processo non sempre si svolge come desiderato: in particolare, per fare un esempio rilevante, alle volte l'input del modello contiene implicitamente una sequenzialità, si pensi ad esempio al testo scritto; questa sequenzialità rischia di non essere ben catturata da un processo di estrazione delle features che avviene in parallelo su tutto l'input, e dunque per alcuni tipi di problemi è conveniente non tanto utilizzare reti molto profonde quanto usare *reti neurali ricorrenti*. Avremo modo di approfondire ulteriormente quest'argomento in una delle prossime sezioni.

Si noti in coda che all'atto pratico sarà necessario scegliere il numero di hidden layers della rete e la dimensione di questi ultimi. Queste scelte definiscono l'architettura di una rete densamente feedforward, ed attualmente non esiste un singolo metodo per determinare quale possa essere la scelta migliore. Spesso nelle *toy applications* questi valori sono essenzialmente fissati a caso, ma esistono alcune tecniche che mirano alla determinazione di valori ottimali per questi parametri d'architettura, per esempio lo *spectral pruning* [14].

1.2.3 Come Addestrare una Rete Neurale

Avendo capito che impiegare reti neurali profonde sostanzialmente equivale ad operare una particolare scelta per la funzione $f(\boldsymbol{w}, \boldsymbol{x})$ capiamo anche che l'addestramento della stessa può avvenire con i metodi descritti nella sezione precedente. Tuttavia all'atto pratico lavorare con reti neurali profonde ci mette di fronte ad un problema che in precedenza poteva essere ignorato, ovvero le difficoltà del calcolo del gradiente, necessario per applicare il metodo della discesa del gradiente. Dato che il numero di parametri di una rete neurale è in generale ingente, e visto che la struttura della stessa tende ad essere intricata, il calcolo del gradiente rischia di divenire computazionalmente dispendioso, rendendo il training delle reti arduo in pratica. Fortunatamente è possibile procedere con il calcolo in *CPU times* ragionevoli operando oculate scelte di metodo: nel nostro contesto il modo migliore di calcolare le derivate parziali che compongono il gradiente è ricorrere alla *differenziazione automatica*. In particolare è stato sviluppato un algoritmo di differenziazione molto efficiente per i nostri scopi noto come *backpropagation*, che consente il calcolo del gradiente in tempi adeguati; essenzialmente si differenzia costruendo il *grafo computazionale* corrispondente alla funzione f, e questo permette di mettere in atto una serie di scorciatoie nel calcolo delle derivate parziali che riducono il costo computazionale necessario. In questa sede scegliamo di non approfondire ulteriormente questo tema, basti sapere che questo efficiente metodo di calcolo del gradiente è in pratica una componente fondamentale del processo di training [29, 12]. Un approfondimento su questo argomento è tuttavia presente in Appendice A.

Potendo calcolare il gradiente potremmo semplicemente applicare il metodo della discesa del gradiente per ottimizzare i pesi della rete neurale. Tuttavia all'atto pratico, se come spesso accade il dataset di train ha dimensioni ingenti, il calcolo del gradiente anche utilizzando *backpropagation* rischia comunque di esser troppo dispendioso. Questo è uno dei principali motivi che stanno alla base dell'adozione di un'altra tecnica di ottimizzazione per le reti neurali: nota come SGD (**Stochastic Gradient Descent**). Il suo funzionamento si basa sulla divisione del dataset di train in porzioni, dette *batches*. Il *batch size* di queste ultime va considerato come un iperparametro. L'idea è quella di calcolare il gradiente della loss (del rischio empirico dunque) unicamente su uno dei batches, per poi operare uno step di ottimizzazione con quel gradiente *"limitato"*. Si opera un numero di passi di ottimizzazione (1.12) pari al numero di *batches*, cambiando ogni volta il batch di riferimento per il calcolo del gradiente; una volta compiuti tutti questi *steps* diremo di aver raggiunto *un'epoca di train*; il training può andare avanti anche per più epoche, eseguendo un *reshuffle* dei batches fra epoca e epoca [29].

SGD è comunemente utilizzato per il training di reti neurali. Tuttavia esso non è privo di problemi, ed uno dei più fastidiosi è il rischio di fare *overshooting*: ovvero avere nelle fasi finali del training un valore del learning rate troppo alto per consentire ulteriore ottimizzazione, il valore dei pesi *rimbalza* attorno alla regione ottimale, saltandola ad ogni step di ottimizzazione, come mostrato nella Figura 1.6.

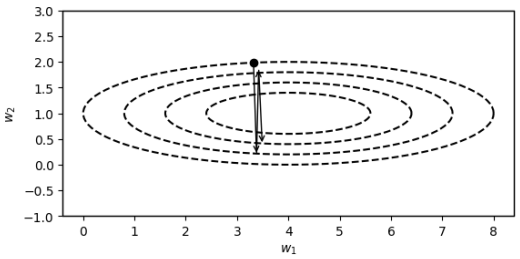

Figura 1.6. Grafico della curve di livello di una tipica loss $L(w)$ problematica per il metodo del gradiente. Si vede bene che il peso w_1 sull'asse delle ascisse è associato ad una variabile rumorosa, ovvero che non è molto rilevante per il problema in esame; possiamo notare difatti come la loss al variare di questa vari poco. L'irrilevanza della x_1 fa sì che le curve di livello siano molto schiacciate, e questo graficamente ci fa subito intuire che gli step di ottimizzazione non saranno probabilmente ottimali in questa regione. Nello specifico il valore dei pesi è prono a *"rimbalzare"* da lato a lato come mostrato in figura, avvicinandosi al contempo pochissimo al vero punto di ottimo.

Potremmo pensare di risolvere il problema diminuendo il valore del learning rate, questa scelta non è però percorribile dato che oltre un certo livello un learning rate troppo basso allunga di troppo il processo di ottimizzazione nella sua fase di avvicinamento alla regione di ottimo.
Per cercare di ovviare a questo problema quindi sono state essenzialmente sviluppate diverse versioni di SGD: una delle principali è **Adam**. Adam modifica SGD introducendo essenzialmente due idee: la prima è modificare dinamicamente il valore effettivo del learning rate per ridurlo in prossimità della zona d'ottimo, e la seconda è tener conto della direzione del gradiente agli step di ottimizzazione precedenti per creare una sorta di *inerzia* nel processo di ottimizzazione. Queste due meccaniche portano a buone performance con ridotte necessità di *hyperparameter tuning* [34].

Approfondimento sull'ottimizzatore Adam
Nel capitolo successivo l'ottimizzatore che verrà utilizzato per il training sarà proprio *Adam*. Risulta quindi doveroso inserire un inserto che esponga più in dettaglio il suo funzionamento.
Il contesto iniziale è analogo a quello di SGD: vogliamo minimizzare il rischio empirico $\mathcal{R}_D(\boldsymbol{w})$, che dipenderà dal dataset D e dai parametri del modello $\boldsymbol{w}$. Il processo di ottimizzazione inizia con la scelta di un punto di partenza per i parametri da ottimizzare (come solito per un processo di ottimizzazione iterativa); chiameremo i parametri iniziali $\boldsymbol{w}_0$. Inoltre inizialmente fisseremo un valore γ per il learning rate. Se volessimo operare l'ottimizzazione con il metodo della discesa del gradiente l'esposizione del contesto iniziale sarebbe terminata, tuttavia con *Adam* dobbiamo anche fissare due ulteriori parametri

$$\beta_1, \beta_2 \in [0, 1)$$

che serviranno per la modifica dinamica del learning rate effettivo. Tutte le quantità appena esposte vanno considerate come iperparametri del processo di learning con *Adam*. Giunti a questo punto definiamo inoltre 3 importanti variabili:

- Il *time step* t; inizialmente posto a 0.
- Il *first moment estimate* $\boldsymbol{m}_t$; inizialmente posto a 0.
- Il *second raw moment estimate* $\boldsymbol{v}_t$; inizialmente posto a 0.

Il *time step* conta semplicemente i passi dell'ottimizzazione iterativa. $\boldsymbol{m}_t$ e $\boldsymbol{v}_t$ sono vettori di dimensione pari a quella dei pesi $\boldsymbol{w}$, e servono per tenere traccia della direzione del gradiente e della sua varianza (*raw: varianza non centrata*). Nel seguito il valore del gradiente allo step t sarà indicato con $\boldsymbol{g}_t$:

$$\boldsymbol{g}_t \doteq \nabla_{\boldsymbol{w}} \mathcal{R}_D(\boldsymbol{w}_{t-1}). \tag{1.45}$$

Fissiamo infine il valore di un piccolo parametro ϵ, servirà per evitare problemi di divisione per zero; solitamente si sceglie:[34]

$$\epsilon = 10^{-8}. \tag{1.46}$$

L'esposizione del contesto iniziale con questo è terminata; il processo di ottimizzazione con *Adam* si svolge come segue:

1. Aggiornare il *time step*: $t \leftarrow t + 1$;
2. Calcolare il gradiente $\boldsymbol{g}_t$;
3. Aggiornare $\boldsymbol{m}_t$ (*biased*): $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$;
4. Aggiornare $\boldsymbol{v}_t$ (*biased*): $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$.
   Si faccia attenzione al fatto che qui $\boldsymbol{g}_t^2$ è da intendersi come il quadrato elemento per elemento di $\boldsymbol{g}_t$ (*hadamard product*);
5. Correggere $\boldsymbol{m}_t$ (*unbiased*): $\hat{\boldsymbol{m}}_t \leftarrow \frac{\boldsymbol{m}_t}{1-\beta_1^t}$;
6. Correggere $\boldsymbol{v}_t$ (*unbiased*): $\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1-\beta_2^t}$;
7. Aggiornare i pesi: $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \gamma \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}$.
   Si noti che qui ogni operazione è da intendersi come un'operazione elemento per elemento.
8. Ripetere fino al numero di time steps desiderato.

Nel seguito si mira ad elucidare il funzionamento dell'algoritmo appena esposto *step by step*. I *primi due steps* non hanno bisogno di ulteriori spiegazioni. Il *terzo step* di fatto aggiorna la direzione del gradiente, e nello specifico lo calcola come una media pesata fra il valore precedente ed il valore del gradiente attuale, quanto contare il gradiente attuale rispetto al precedente è determinato dal parametro β_1; si capisce come questo crei una sorta di *inerzia*. Il *quarto step* è analogo al terzo, ma calcola la varianza, non centrata, del gradiente; ed è pesato da β_2. Capiremo fra poco l'utilità di questa variabile. Per comprendere

il *quinto step* dobbiamo notare che il valore di $\boldsymbol{m}_t$ è inizializzato a zero, e questo ovviamente crea un bias verso zero per il suo valore. Per correggere questo bias si divide $\boldsymbol{m}_t$ per un fattore $1 - \beta_1^t$: questo ha l'effetto di spostare lontano da zero il valore di $\boldsymbol{m}_t$ nelle fasi iniziali del training, dove il bias è più presente e dannoso, e di divenire poi gradualmente meno rilevante grazie alla potenza t-esima. Il *sesto step* è perfettamente analogo al quinto, ma per $\boldsymbol{v}_t$.
Siamo infine giunti al *settimo*, ed ultimo, *step*: questo è il più importante in quanto mette insieme quanto abbiamo fatto e esprime come aggiornare il valore dei pesi. La formula di aggiornamento dei pesi è in parte analoga a quella classica della discesa del gradiente (o SGD), l'importante differenza è che al posto di g_t è presente un espressione contenente $\hat{\boldsymbol{m}}_t$: si ha quindi un effetto di inerzia, come già detto; la variabile ϵ è ignorabile in quanto serve unicamente ad evitare un'eventuale divisione per zero, quello che è davvero importante è la divisione per la radice di $\hat{\boldsymbol{v}}_t$, che serve a normalizzare il gradiente in base alla sua varianza. Questo è un passaggio cruciale in quanto permette di mitigare il problema dell'overshooting: $\hat{\boldsymbol{v}}_t$ essenzialmente tiene conto dei moduli di tutti i gradienti passati, e dunque se il gradiente improvvisamente diminuisce in modulo, come accade spesso nei pressi di una regione di ottimo, il learning rate effettivo verrà soppresso. Questo in linea di principio permette di iniziare a compiere piccoli steps di ottimizzazione in prossimità della regione di ottimo, mitigando il problema dell'overshooting. Si noti infine che essendo l'aggiornamento dei parametri un'operazione *elementwise* il learning rate effettivo può a priori essere diverso per ogni parametro; questa differenziazione rappresenta un ulteriore vantaggio rispetto ad ottimizzatori più semplici come SGD [34].

1.2.4 Alcune Interessanti Tipologie di Reti Neurali

Fino ad ora ci siamo concentrati su un unico tipo di rete neurale, ovvero reti feedforward densamente connesse. Tuttavia esistono molte altre tipologie di reti neurali, adatte a diversi contesti, e in questa sezione ne esporremo brevemente alcune, le più rilevanti per la nostra trattazione.

Reti Neurali Residuali

Iniziamo questa sezione parlando delle *reti neurali residuali*, anche note come *ResNets*. L'idea di base è che architetture profonde sono necessarie per una corretta estrazione di features complesse, e dunque sorge spontaneo chiedersi se un aumento di profondità dell'architettura sia sempre associato a performances migliori; la risposta a questo quesito è negativa a causa del celebre *vanishing/exploding gradient problem*: essenzialmente un grande numero di layers in sequenza rende il processo di training ingestibile a causa dell'azione ripetuta dei pesi, che tende a spingere il gradiente della funzione di loss sul batch ad annullarsi o ad esplodere. In letteratura si mostra che si può rimediare al problema attraverso l'aggiunta di opportuni *layers di normalizzazione*, tuttavia queste procedure non sono completamente risolutive in quanto le reti neurali profonde possono comunque andare incontro ad un processo di *degradazione*: si verifica empiricamente che spesso un'eccessiva profondità riduce le performance della rete [32].

Le reti neurali residuali nascono con l'obbiettivo di rendere possibile il train di reti neurali profonde, lenendo le problematiche esposte sopra. L'idea di base è estremamente semplice, e consiste nell'aggiunta di *connessioni skip layer*, ovvero connessioni fra strati neurali non adiacenti. Nello specifico nelle ResNets le connessioni skip layer tendono ad essere *identity skip connections*, ovvero connessioni non pesate (o se si preferisce con $w_{ij} = 1\ \forall i, j$). Queste connessioni permettono al gradiente di *"saltare"* i layer neurali, e dunque di mitigare il problema del vanishing/exploding gradient. Inoltre si mostra in letteratura che le ResNets sono in grado di lenire anche il problema della *degradation*, ovvero il fatto che l'aggiunta di layer profondi ad una rete possa ridurne le performances: questo è dovuto al fatto che i blocchi ResNet (vedi la Figura 1.7) sono facilmente in grado di approssimare la funzione identità, e dunque in generale non dovrebbero fare peggio di una rete neurale meno profonda. Nella figura sottostante riportiamo la struttura di un classico *blocco residuale* [32].

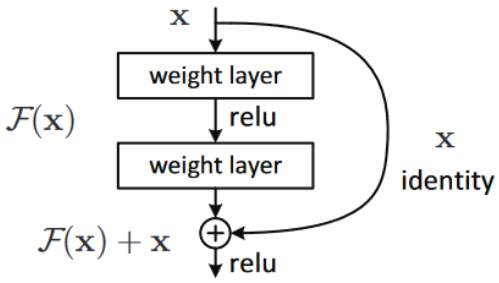

Figura 1.7. Rappresentazione grafica di un blocco residuale. Si noti come il blocco sia composto da due canali in parallelo: il primo formato da due layer neurali, ed il secondo formato da un semplice *identity mapping*. (Immagine estratta dall'articolo originale di *He et al.* [32]).

Per un blocco residuale *a due layers* come quello in Figura 1.7 dato un input $\boldsymbol{x}$ l'elaborazione che porta all'output $\boldsymbol{y}$ è data da:

$$\boldsymbol{y} = \sigma(\mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}) \tag{1.47}$$

dove $\mathcal{F}(\boldsymbol{x})$ indica l'azione dei due layer neurali, ovvero:

$$\mathcal{F}(\boldsymbol{x}) = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2. \tag{1.48}$$

Notiamo che nel caso in cui l'input x non abbia le stesse dimensioni dell'output y il formalismo può essere esteso includendo una matrice di proiezione W_P:

$$\boldsymbol{y} = \sigma(\mathcal{F}(\boldsymbol{x}) + W_P\boldsymbol{x}). \tag{1.49}$$

Questi blocchi residuali compongono una ResNet quando messi in serie; ed è anche possibile prendere in esame architetture con un numero di layers neurali in $\mathcal{F}$ maggiore di 2.

Reti Neurali Ricorrenti

Le reti neurali ricorrenti (RNN) sono un'altra tipologia di rete neurale molto utilizzata in letteratura. Queste reti sono particolarmente adatte per problemi in cui l'input è sequenziale, ovvero in cui l'ordine degli elementi dell'input è rilevante. Un esempio di problema in cui le RNN sono particolarmente adatte è la previsione di serie temporali, contesto in cui l'input è una sequenza di valori temporali e l'output è un valore futuro della serie. Un semplice esempio di RNN è mostrato in Figura 1.8.

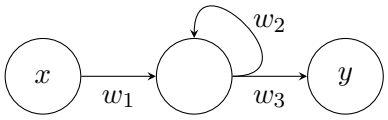

Figura 1.8. Diagramma rappresentativo della più semplice architettura possibile per una rete neurale ricorrente. La rete presenta un solo neurone di input ed un solo neurone di output; l'hidden layer è composto da un singolo neurone con connessione ricorrente. Si noti che alle volte l'output y della rete ricorrente è in letteratura chiamato *hidden state* h.

La caratteristica distintiva delle RNN è la presenza di *loops*; questi permettono di mantenere una memoria dello stato passato della rete. Questa proprietà è particolarmente utile in contesti in cui l'input è sequenziale, in quanto permette alla rete di tenere conto dell'ordine degli elementi dell'input.
Il modo migliore di acquisire una chiara comprensione del funzionamento di una RNN è *"srotolarla nel tempo"*, ovvero considerare la rete come una sequenza di reti neurali feedforward, una per ogni elemento dell'input. Si faccia riferimento alla Figura 1.9 per una rappresentazione grafica.

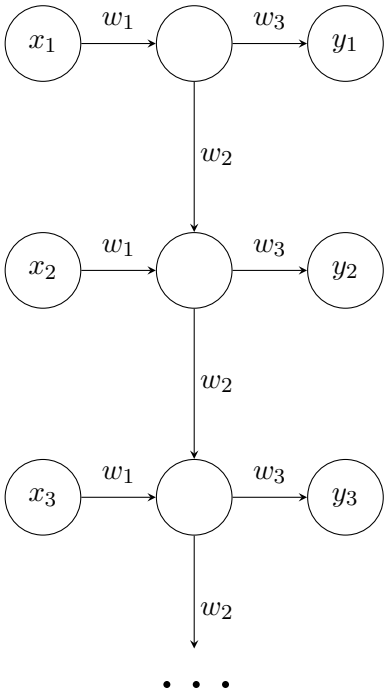

Figura 1.9. Rappresentazione grafica della rete neurale ricorrente in Figura 1.8, srotolata nel tempo. Gli x_i rappresentano gli elementi della sequenza in input alla rete, gli y_i di contro rappresentano gli output della rete, essi vengono prodotti a mano a mano che la rete riceve elementi in input. Si noti che questa struttura consente di gestire sequenze di lunghezza arbitraria L. All'ultimo elemento della sequenza corrisponderà l'ultimo output y_L, che sarà l'output finale dell'elaborazione della rete. La rete ricorrente potrebbe ad esempio essere allenata per predire il prossimo elemento della sequenza, in maniera tale quindi che y_L sia la predizione dell'elemento successivo a x_L.

Dunque, nel nostro semplicissimo esempio di RNN, y_1 sarà dato da:

$$y_1 = w_3\big(\sigma(w_1 x_1 + b)\big) \tag{1.50}$$

dove con σ come al solito indichiamo la funzione di attivazione, e con b l'eventuale bias neurale. Di contro y_2 inizierà a sentire gli effetti dell'architettura ricorrente, e sarà dunque:

$$y_2 = w_3\Big(\sigma\big(w_1 x_2 + w_2\sigma(w_1 x_1 + b) + b\big)\Big) \tag{1.51}$$

possiamo quindi facilmente notare come gli output successivi siano determinati anche dal valore degli elementi in input precedenti.
Questa tipologia di rete tuttavia, a prescindere dai dettagli d'architettura, soffre del *vanishing/exploding gradient problem*, cosa che rende il training assai difficoltoso. Si noti infatti che sulle auto-connessioni, ovvero sui *loops*, saranno necessariamente presenti dei pesi w (come per tutte le connessioni di una rete neurale artificiale); ma dunque ogni nuovo elemento della sequenza immesso nell'elaborazione implica un'azione ripetuta dei pesi, che si concretizza in una moltiplicazione ripetuta dei pesi stessi. In definitiva elaborare una sequenza di lunghezza L comporta la presenza nell'output di una potenza L-esima dei pesi w presenti sulle auto-connessioni: nel caso in cui i w siano minori di 1 si ha quindi il problema della scomparsa del gradiente, mentre nel caso essi siano maggiori di 1 si ha il problema dell'esplosione del gradiente. Concretamente questo implica che le reti RNN abbiano difficoltà a mantenere una memoria a lungo termine [8, 29].
Le reti ricorrenti *LSTM (Long Short Term Memory)* sono state sviluppate per ovviare a questo problema. Accenneremo al loro funzionamento nella prossima sezione.

Reti Neurali LSTM

Come già accennato le reti neurali LSTM sono state sviluppate per ovviare al problema del *vanishing/exploding gradient* nelle reti neurali ricorrenti. L'architettura di una LSTM è solitamente rappresentata come una *cella* [17]. In figura 1.10 è riportato un esempio di cella LSTM.

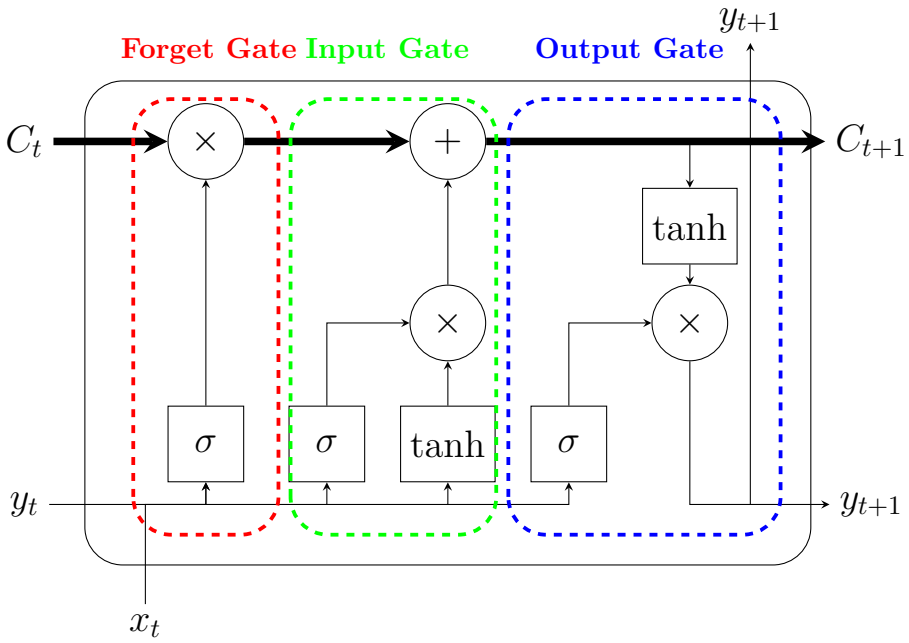

Figura 1.10. Rappresentazione grafica di una cella LSTM.

Si noti come la cella LSTM (Figura 1.10) sia composta da 3 blocchi principali: il *forget gate*, l'*input gate*, e l'*output gate*. Alle volte in letteratura si preferisce pensare all'*input gate* come composto da due parti distinte: una che si occupa di creare

una nuova memoria a lungo termine (*destra*), e l'altra che si occupa di decidere quanto di questa nuova memoria immettere nella memoria a lungo termine della cella (*sinistra*); queste due vengono rispettivamente chiamate *control gate* e *input gate*. Tuttavia noi sceglieremo di adottare la nomenclatura alternativa esposta in figura, chiamando semplicemente la totalità del gate di mezzo *input gate*. Infine si faccia attenzione al fatto che, per come è disegnata, la cella LSTM rappresentata in figura si presta ad essere *"srotolata"* in direzione orizzontale; e non in direzione verticale come fatto in Figura 1.9.

Nel seguito esporremo a grandi linee il funzionamento di una cella LSTM, sempre basandoci sulla sua rappresentazione in Figura 1.10.

In primo luogo osserviamo che una cella LSTM è un tipo di rete neurale ricorrente: possiamo notare come essa riceva un input x_t ed anche il risultato della precedente computazione y_t, esattamente come accade in una RNN. Il fatto che la cella riceva in input anche y_t e non solo x_t è ciò che la rende *ricorrente*.

Possiamo inoltre notare come la cella riceva in input, attraverso un canale separato, anche una quantità C_t, chiamata *cell state*; questa è la *memoria a lungo termine* della cella, inizialmente inizializzata a zero (come del resto è posto a zero anche y_0). La presenza di questo canale rappresenta la principale differenza fra una cella LSTM e una classica RNN.

Concentriamoci adesso sulla prima parte della cella, ovvero il *forget gate*: questo si occupa di determinare quanta memoria a lungo termine ricordare. Fatta eccezione per il canale della memoria a lungo termine tutti i canali di una cella LSTM sono pesati, esattamente come accade in una rete neurale tradizionale; x_t e y_t vengono dunque moltiplicati per i pesi e combinati, e poi il segnale risultante viene dato in pasto ad una funzione sigmoide (σ), che produrrà un output compreso in $(0, 1)$; questo viene poi moltiplicato alla memoria a lungo termine, e determina quindi quanta di essa verrà *"ricordata"* e quanta verrà *"dimenticata"*.

La prossima porzione della cella è l'*input gate*: questo si occupa di creare nuove memorie a lungo termine sulla base di x_t, y_t. Possiamo notare che esso è composto da due parti: una fa uso di una funzione di attivazione (solitamente la tangente iperbolica) per creare una memoria a lungo termine potenziale, l'altra ha di nuovo una sigmoide, e determina quanta della potenziale nuova memoria a lungo termine effettivamente immettere nel canale di C_t.

Infine abbiamo l'*output gate*: questo si occupa di combinare la memoria a lungo termine C_t con x_t e y_t per produrre l'output della rete y_{t+1}. Si noti come l'output y_{t+1} venga passato sia in output (nel canale di uscita in alto a destra nella figura) che alla prossima applicazione ricorrente della cella (attraverso il canale in basso a destra in figura).

Come abbiamo già menzionato la principale differenza fra una rete neurale ricorrente e una cella LSTM è la presenza della memoria a lungo termine, ovvero la presenza di un canale diretto che connette diverse applicazioni della cella. Dalla trattazione delle *ResNets* sappiamo che canali di connessione a lungo raggio mitigano il problema della scomparsa/esplosione del gradiente, e nel caso delle LSTM si osserva la stessa cosa: il canale a lungo raggio della memoria a lungo termine sana l'*exploding/vanishing gradient problem* [29, 39].

1.3 Reti Neurali Spettrali

Nella presente sezione esporremo un formalismo descrittivo per reti neurali profonde diverso da quello esposto nelle scorse sezioni, questo sarà centrale per la trattazione dei capitoli successivi.
Per iniziare ad esporre questo tema notiamo che ogni coppia di layers di una rete neurale feedforward può essere interpretata come un grafo orientato bipartito, e dunque sarà ben rappresentabile da una *matrice di adiacenza* A.

Inserto: Matrice di Adiacenza
Si definisce *matrice di adiacenza* (o *matrice delle adiacenze*) una matrice quadrata A atta a rappresentare un grafo finito. L'elemento a_{ij} rappresenterà l'eventuale connessione fra il nodo j ed il nodo i, l'assenza di connessione verrà indicata con uno zero; si noti che l'ordine è importante per la definizione, ed alle volte in letteratura la matrice A viene definita *al contrario*, nel seguito useremo sempre la convenzione appena introdotta. Per esempio il grafo orientato esposto nella Figura 1.11 è rappresentato dalla matrice di adiacenza in (1.52).

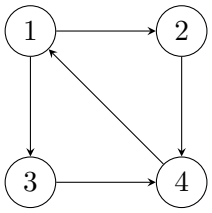

Figura 1.11. Grafo orientato rappresentato dalla matrice di adiacenza (1.37).

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}. \tag{1.52}$$

Nel caso in cui invece il grafo sia caratterizzato da *connessioni pesate*, ovvero ogni arco fra due nodi j, i è associato ad un peso $w_{ij} \in \mathbb{R}$, la matrice di adiacenza rappresentativa conterrà i pesi degli archi al posto di elementi unitari per indicare le connessioni. Per esempio il grafo pesato esposto in Figura (1.12) è rappresentato dalla matrice di adiacenza in (1.53).

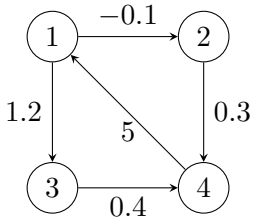

Figura 1.12. Grafo pesato rappresentato dalla matrice di adiacenza (1.53).

$$A = \begin{pmatrix} 0 & 0 & 0 & 5 \\ -0.1 & 0 & 0 & 0 \\ 1.2 & 0 & 0 & 0 \\ 0 & 0.3 & 0.4 & 0 \end{pmatrix}. \tag{1.53}$$

Nello specifico se i due strati in esame hanno un numero complessivo di neuroni pari a N la matrice di adiacenza della sotto-rete in esame, ovvero della rete composta dai due strati, sarà una matrice $N \times N$ tale da avere diversi da zero solo gli elementi sotto la diagonale principale. Ovviamente N sarà pari alla somma del numero di neuroni del primo layer N_1 con il numero di neuroni del secondo layer N_2.
Notiamo inoltre che i *links* fra i neuroni in una rete sono *pesati*, e dunque per descrivere completamente la rete l'elemento a_{ij} della matrice di adiacenza dovrà contenere il valore del peso che caratterizza la connessione dal neurone j al neurone i (si faccia attenzione a non confondere l'ordine degli indici). Il blocco sotto-diagonale della matrice di adiacenza contiene tutte le specifiche sul passaggio di informazione fra i due layers della rete. Date queste semplici premesse il metodo spettrale (standard) si propone di addestrare e studiare i pesi delle connessioni neurali nello *spazio reciproco*; avremo modo di vedere come questo approccio conduca a molti vantaggi.
Concretamente l'approccio spettrale parte considerando una *matrice di connessione spettrale* Φ caratterizzata dalla forma esposta in Figura 1.13.

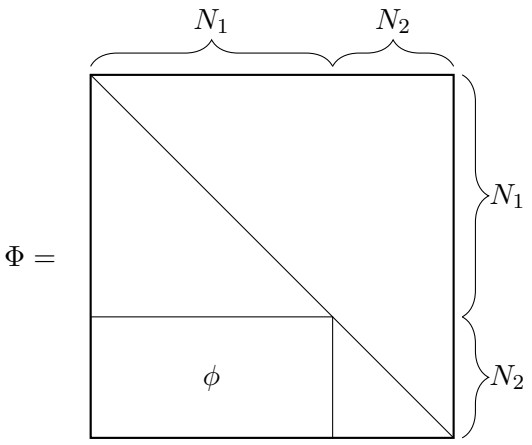

Figura 1.13. Matrice di connessione spettrale Φ_1. Il blocco ϕ è un generico blocco di valori reali, e la linea diagonale sta a significare che gli elementi ivi ospitati sono identicamente uguali ad uno. Le restati entrate della matrice sono fissate a zero.

Per chiarezza notiamo che la matrice Φ può anche essere espressa, in una forma a blocchi più tradizionale, come:

$$\Phi = \begin{pmatrix} \mathbb{I}_{N_1 \times N_1} & \mathbb{O}_{N_1 \times N_2} \\ \phi_{N_2 \times N_1} & \mathbb{I}_{N_2 \times N_2} \end{pmatrix}_{N \times N} \tag{1.54}$$

dove con $\mathbb{O}$ vogliamo indicare una generica matrice rettangolare di zeri e con $\mathbb{I}$ una generica matrice identità. Questa matrice verrà interpretata come la matrice degli *autovettori* della matrice di adiacenza del grafo bipartito. La nostra intenzione dunque è quella di esprimere la matrice di adiacenza A come:

$$A = \Phi\Lambda\Phi^{-1} \tag{1.55}$$

con Λ opportuna matrice degli autovalori, la cui struttura è esposta in Figura 1.14

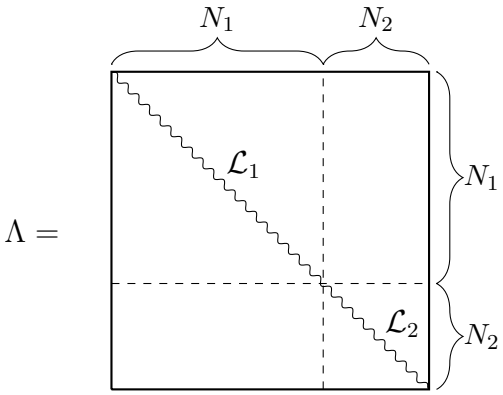

Figura 1.14. Matrice degli autovalori Λ. La linea diagonale ondulata sta a significare che gli elementi ivi ospitati hanno in generale valore diverso da 0 o 1.

Scegliamo di interpretare la matrice degli autovalori di A come una matrice diagonale a blocchi (quadrati), questo sarà utile nell'analisi che segue, data la struttura della (1.55).
Si può dimostrare, notando che la matrice inversa di Φ è data da

$$\Phi^{-1} = 2\mathbb{I} - \Phi, \tag{1.56}$$

che la matrice di adiacenza che questo formalismo produce attraverso la (1.55) ha la struttura riportata in Figura 1.15.

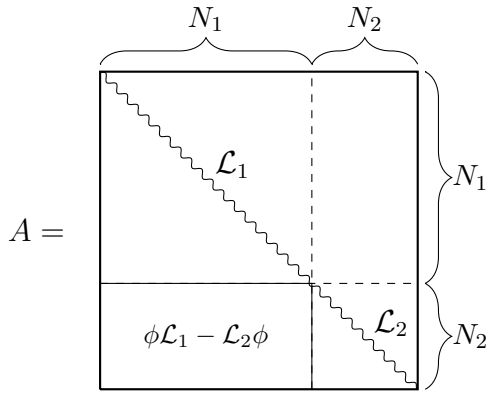

Figura 1.15. Matrice di adiacenza A prodotta dal metodo spettrale.

La matrice di adiacenza risultante rappresenta quindi un grafo dalla topologia esposta in Figura 1.16.

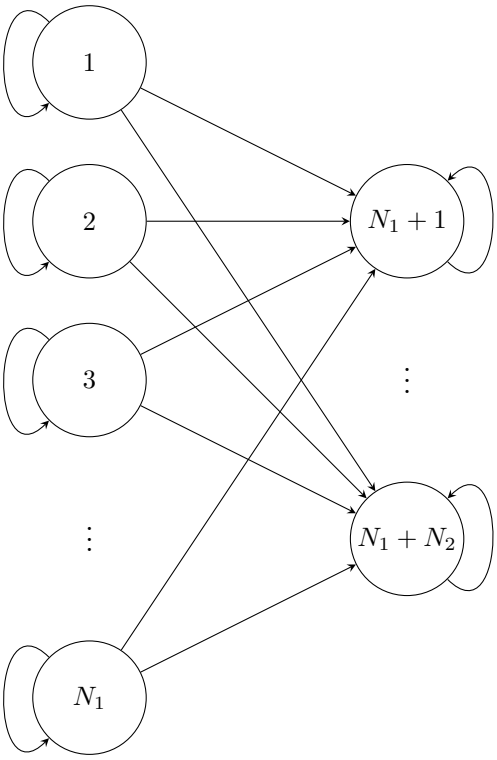

Figura 1.16. Grafo rappresentato dalla matrice di adiacenza A prodotta dal metodo spettrale. La connessione fra il j esimo neurone del primo layer e l'i esimo neurone del secondo layer è pesata dall'elemento w_{ij} del blocco sotto-diagonale della matrice di adiacenza. I self-loops sul primo layer sono pesati dagli elementi diagonali della sotto-matrice $\mathcal{L}_1$, mentre i self-loops del secondo layer sono pesati dagli elementi diagonali della sotto-matrice $\mathcal{L}_2$.

Il formalismo spettrale porta quindi a grafi bipartiti feedforward, ma non *strettamente* feedforward: difatti possiamo notare la presenza di *self-loops* su ogni neurone. La matrice A, che rappresenta la topologia della rete nella sua interezza, nel formalismo spettrale viene anche utilizzata come operatore di evoluzione delle attivazioni dei neuroni della rete. In particolare si definisce un vettore delle attivazioni neurali $\boldsymbol{a} \in \mathbb{R}^N$ tale da avere come primi N_1 elementi le attivazioni del primo layer e come ultimi N_2 elementi le attivazioni del secondo layer. Dato che stiamo lavorando nel contesto di una rete feedforward le attivazioni saranno presenti sul primo layer neurale e dovranno fluire sul secondo; dunque inizialmente il vettore delle attivazioni avrà solo i primi N_1 elementi non nulli

$$\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{N_1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \tag{1.57}$$

Nel seguito chiameremo la parte inizialmente non nulla del vettore delle attivazioni *input* $\boldsymbol{x}$.
La propagazione delle attivazioni attraverso la rete viene dunque calcolata aggiornando il vettore delle attivazioni con la matrice di adiacenza A:

$$A\boldsymbol{a} = \begin{pmatrix} \mathcal{L}_1 \boldsymbol{x} \\ W\boldsymbol{x} \end{pmatrix}. \tag{1.58}$$

Gli ultimi N_2 elementi del vettore risultante rappresenteranno le attivazioni del secondo layer dopo la propagazione delle attivazioni dal primo layer, e possono dunque essere estratti per ottenere il risultato dell'elaborazione dell'informazione operata dalla rete, che nel seguito chiameremo *output* $\boldsymbol{y}$:

$$\boldsymbol{y} = W\boldsymbol{x}. \tag{1.59}$$

Possiamo notare che i self-loops, pur essendo presenti nella topologia, non contribuiscono direttamente all'elaborazione dell'informazione della rete.
L'elaborazione dell'informazione può volendo esser resa non lineare semplicemente introducendo una non linearità σ, un filtro che agisce selettivamente su ogni componente del vettore delle attivazioni:

$$\sigma \;\Big|\; \sigma(\boldsymbol{v}) = \begin{pmatrix} f(v_1) \\ f(v_2) \\ \vdots \\ f(v_D) \end{pmatrix} \;\; \forall v \in \mathbb{R}^D \tag{1.60}$$

dove $f : \mathbb{R} \to \mathbb{R}$ è una funzione non lineare ad hoc, identificabile con la *ReLU*.
L'output non lineare ha quindi semplicemente forma:

$$\boldsymbol{y} = \sigma(W\boldsymbol{x}). \tag{1.61}$$

Concatenando multipli di questi *blocchi spettrali* si capisce come sia possibile costruire una generale rete feedforward densa. Infatti l'azione che dà luogo all'output $\boldsymbol{y}$ è proprio quella che ci aspettiamo dal trasferimento di informazione da uno strato all'altro di una rete neurale standard.[4] Quello che cambia rispetto ad una classica rete neurale non è solo l'interpretazione della stessa, che adesso viene vista dalla prospettiva della teoria dei grafi, ma anche la parametrizzazione della rete. Difatti nel formalismo spettrale risulta facile dimostrare che il blocco di connessione W è dato dalla seguente espressione:

$$W = \phi\mathcal{L}_1 - \mathcal{L}_2\phi \tag{1.62}$$

e dunque che i singoli pesi dei link neurali sono dati da:

$$w_{ij} = \varphi_{ij}[\mathcal{L}_1]_j - [\mathcal{L}_2]_i\varphi_{ij}. \tag{1.63}$$

dove con φ_{ij} indichiamo gli elementi del blocco ϕ. Il training della rete dunque non avviene manipolando direttamente il valore dei pesi w_{ij}, come si fa nelle reti neurali tradizionali, bensì il training avviene *nello spazio reciproco*, manipolando il valore degli elementi della sotto-matrice ϕ e della diagonale di Λ. Questo ha

[4] Si potrebbe lamentare l'assenza del *bias neurale*, tuttavia esso può esser facilmente reintrodotto nel nostro formalismo, per esempio semplicemente aggiungendo un *neurone di bias*, ad attivazione costante pari ad 1, sul primo layer del grafo bipartito. Questo è un trucco molto comune nel contesto delle reti neurali artificiali.

importanti conseguenze sul processo di ottimizzazione, in quanto una differente parametrizzazione comporta a priori una diversa esplorazione dello spazio delle ipotesi. In coda notiamo che avere accesso diretto agli autovalori ed autovettori della rete permette di operare un controllo fine sul trasferimento di informazione che avviene nella rete, e questo si presta ad essere sfruttato in molti modi, daremo alcuni esempi dei vantaggi di questo approccio nella prossima sezione [16].

1.3.1 Noti Vantaggi del Formalismo Spettrale

Nel corso degli ultimi anni sono emersi diversi vantaggi nell'utilizzo del formalismo spettrale per la costruzione e l'addestramento di reti neurali profonde. A seguire una succinta esposizione di due dei principali.

- **Riduzione dello spazio dei parametri sottoposti all'addestramento:** tramite l'adozione del formalismo spettrale lo spazio dei parametri della rete si bipartisce nella porzione degli autovalori e la porzione degli autovettori. In letteratura viene mostrato come sia possibile addestrare unicamente gli autovalori, lasciando dunque le matrici Φ inalterate, ed ottenere performances comparabili con quelle di un addestramento tradizionale. Questo consente di comprimere notevolmente lo spazio dei parametri sottoposti ad ottimizzazione, diminuendo considerevolmente il costo computazionale associato al training [15].
- **Capacità di pruning dell'architettura neurale:** si dimostra in letteratura che sotto la parametrizzazione spettrale l'attività neurale è intimamente legata al valore degli autovalori. Questo consente di determinare l'importanza dei neuroni per la computazione della rete per mezzo degli autovalori, e rimuovere quindi quelli meno rilevanti, con un impatto trascurabile sulle performance nel task in esame. Questa procedura, denominata *spectral pruning*, consente di compattare la dimensine della rete, il che comporta notevoli vantaggi in fase di *deployment* [14].

Capitolo 2

Reti Spettrali a Tre Layers con Skip Connections

2.1 Introduzione

Nello scorso capitolo abbiamo avuto modo di vedere che la matrice spettrale Φ, se interpretata come matrice degli autovettori, codifica per una matrice di adiacenza A che rappresenta un tipo particolare di rete neurale: una rete feedforward a due strati. Nello specifico possiamo dire che il blocco sotto-diagonale di Φ rappresenta le connessioni fra i due layers, e difatti ricordiamo che è possibile scrivere:

$$\mathcal{W} = \phi\mathcal{L}_1 - \mathcal{L}_2\phi.$$

Data questa osservazione sorge spontaneo il seguente quesito: se un blocco sotto-diagonale nella matrice Φ rappresenta le connessioni fra due strati di neuroni allora due blocchi sotto-diagonali in Φ cosa rappresenterebbero? Connessioni feedforward fra tre strati di neuroni?

L'obbiettivo di questo capitolo sarà rispondere esaurientemente a questa domanda.

Prendiamo quindi in esame una matrice spettrale $\Phi_{N\times N}$, con

$$N = N_1 + N_2 + N_3 \tag{2.1}$$

Esattamente come nel capitolo precedente gli elementi diagonali della matrice Φ saranno unitari, ma questa volta sotto la diagonale considereremo una struttura del tipo esposto in Figura 2.1.

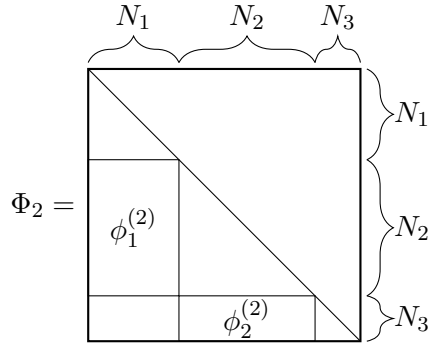

Figura 2.1. Struttura di una matrice spettrale Φ_2 con due blocchi sotto-diagonali.

Notiamo che abbiamo effettuato un sottile cambiamento nella nomenclatura: da ora in avanti indicheremo a pedice in Φ il numero di blocchi sotto diagonali; inoltre i blocchi ϕ avranno da ora in avanti sia un apice che un pedice. L'apice indicherà il numero di blocchi sotto-diagonali della matrice di appartenenza ed il pedice indicherà di quale blocco si tratta (il pedice scorre dall'alto verso il basso, da sinstra verso destra).
Dobbiamo inoltre notare che abbiamo scelto di aggiungere il secondo blocco sotto la diagonale seguendo un determinato criterio: i due blocchi sotto-diagonali sono *concatenati*. L'altezza del primo coincide con la larghezza del secondo, ed essi insieme coprono nella matrice Φ_2 tutto lo spazio orizzontale associato alla dimensione $N_1 + N_2$, e tutto lo spazio verticale associato alla dimensione $N_2 + N_3$ (si veda la figura precedente). Abbiamo scelto questo criterio di estensione per il numero dei blocchi sotto-diagonali ispirandoci al comportamento della matrice Φ_1, dove il singolo blocco sotto diagonale $N_2 \times N_1$ implica nella matrice di adiacenza connessioni fra un primo strato di N_1 neuroni ed un secondo strato di N_2 neuroni; la speranza che si cela dietro alla scelta di estendere il numero dei blocchi sotto-diagonali in maniera concatenata è che si mantengano le stesse implicazioni topologiche nello spazio diretto: ovvero che il blocco ϕ_1 con dimensioni $N_2 \times N_1$ codifichi per connessioni fra un primo strato di N_1 neuroni ed un secondo strato di N_2 neuroni, e che il secondo blocco, avendo le dimensioni concatenate $N_2 \times N_3$, codifichi nello spazio diretto per connessioni dal secondo strato (con appunto sempre N_2 neuroni) al terzo strato con N_3 neuroni. Se questo fosse effettivamente il caso avremmo trovato il modo di estendere il formalismo spettrale a reti feedforward a 3 strati di neuroni, e questo potrebbe aprire la strada ad un'ulteriore generalizzazione a n strati di neuroni, aumentando quindi di moltissimo la capacità descrittiva del formalismo. Tuttavia per adesso queste sono solo ipotesi: è necessario prendere in esame la matrice Φ_2 e da essa dedurre la forma della matrice di adiacenza per poter effettivamente verificare che la struttura concatenata proposta porti a qualcosa di interessante.

2.2 Struttura della Matrice di Adiacenza

Per capire che grafo viene rappresentato da Φ_2 sarà sufficiente ricavare la matrice di adiacenza ad essa associata dal formalismo spettrale; nello specifico sappiamo che essa è data dalla seguente formula:

$$A_2 = \Phi_2 \Lambda_2 \Phi_2^{-1}, \tag{2.2}$$

con Λ_2 matrice diagonale degli autovalori; il pedice 2 sta a significare che ci si riferisce ad una matrice spettrale con due blocchi sotto-diagonali. Il primo passo dunque sarà quello di comprendere la forma di Φ_2^{-1}, e per farlo è conveniente partire descrivendo la struttura dell'inversa di una generica matrice a blocchi M:

$$M = \begin{bmatrix} A_{n\times n} & \mathbb{O}_{n\times m} \\ C_{m\times n} & D_{m\times m} \end{bmatrix}, \tag{2.3}$$

per una matrice invertibile di questa forma, con A, D matrici quadrate tali da ammettere inversa, è noto che [38]

$$M^{-1} = \begin{bmatrix} A^{-1} & O \\ -D^{-1}CA^{-1} & D^{-1} \end{bmatrix} \tag{2.4}$$

come del resto si può verificare attraverso il calcolo diretto:

$$\begin{aligned} &\begin{bmatrix} A & \mathbb{O} \\ C & D \end{bmatrix} \begin{bmatrix} A^{-1} & \mathbb{O} \\ -D^{-1}CA^{-1} & D^{-1} \end{bmatrix} = \begin{bmatrix} AA^{-1} & \mathbb{O} \\ CA^{-1} - DD^{-1}CA^{-1} & DD^{-1} \end{bmatrix} = \\ &= \begin{bmatrix} \mathbb{I} & \mathbb{O} \\ CA^{-1} - CA^{-1} & \mathbb{I} \end{bmatrix} = \begin{bmatrix} \mathbb{I}_{n\times n} & \mathbb{O} \\ \mathbb{O} & \mathbb{I}_{m\times m} \end{bmatrix} = \mathbb{I} \end{aligned} \tag{2.5}$$

Possiamo dunque applicare la (2.4) in *due step* per ottenere l'inversa di Φ_2. Innanzitutto ci occupiamo del blocco di Φ_2 (in alto a sinistra, che denomineremo nel seguito $\tilde{\Phi}_2$):

$$\tilde{\Phi}_2 = \begin{bmatrix} \mathbb{I}_{N_1\times N_1} & \mathbb{O}_{N_1\times N_2} \\ \phi_1^{(2)} & \mathbb{I}_{N_2\times N_2} \end{bmatrix}_{N_1+N_2, N_1+N_2} \tag{2.6}$$

questo blocco di Φ_2 corrisponde al blocco A della matrice M; possiamo subito ricavare con la (2.4) che il blocco $\tilde{\Phi}_2$ ha inversa:

$$\tilde{\Phi}_2^{-1} = \begin{bmatrix} \mathbb{I}_{N_1\times N_1} & \mathbb{O}_{N_1\times N_2} \\ -\phi_1^{(2)} & \mathbb{I}_{N_2\times N_2} \end{bmatrix} \tag{2.7}$$

difatti è ovvio che i blocchi diagonali che compongono $\tilde{\Phi}_2$ siano invertibili (sono matrici identità), e risulta al contempo chiaro che anche $\tilde{\Phi}_2$ debba ammettere inversa in quanto, data la sua struttura, l'insieme dei suoi vettori colonna non può che formare una base del suo spazio vettoriale di riferimento.

Da questo segue immediatamente, applicando nuovamente la (2.4) ma questa volta all'interezza della matrice Φ_2, che Φ_2 ha inversa con struttura esposta in Figura 2.2.

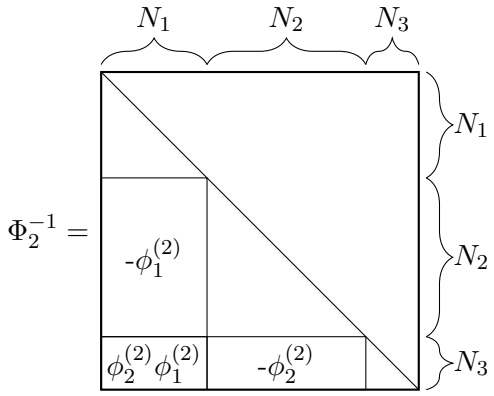

Figura 2.2. Struttura dell'inversa di una matrice spettrale Φ_2 con due blocchi sotto-diagonali.

In Figura 2.2 possiamo notare come in alto a sinistra sia presente l'inversa di $\tilde{\Phi}_2$, in basso a destra l'inversa dell'identità, ed infine il blocco sotto-diagonale sia fornito dal prodotto $-D^{-1}CA^{-1}$. Notiamo inoltre che l'invertibilità delle matrici in esame non rappresenta un problema nemmeno in questo secondo step: difatti le due matrici sulla diagonale di Φ_2 sono chiaramente invertibili, ed al contempo Φ_2 stessa lo è sicuramente data la forma dei suoi vettori colonna.
Avendo trovato la forma di Φ_2^{-1} possiamo subito svolgere i prodotti in (2.2) per determinare la struttura della matrice di adiacenza codificata da Φ_2 e Λ_2. Notiamo che data la natura dei prodotti matriciali risulta conveniente pensare a Λ_2 come ad una matrice composta da 3 blocchi quadrati sulla diagonale, come mostrato in Figura 2.3.

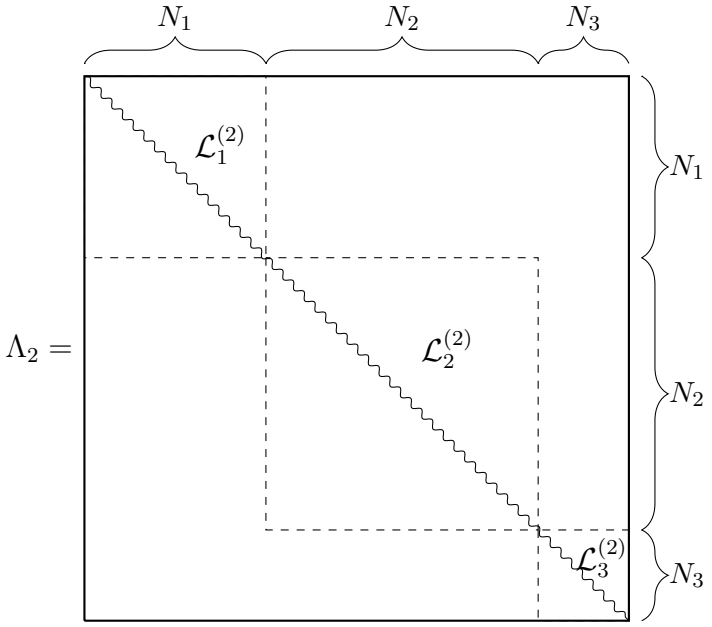

Figura 2.3. Struttura di una matrice diagonale Λ_2 con tre blocchi sulla diagonale.

Si noti che nella figura sovrastante la linea ondulata vuole sempre indicare che gli elementi sulla diagonale sono in generale diversi da zero e diversi da 1; abbiamo tratteggiato il contorno delle matrici sulla diagonale per indicare che esse non sono matrici *"piene"*, bensì matrici diagonali.

Svolgendo (2.2) otteniamo quanto riportato in Figura 2.4.

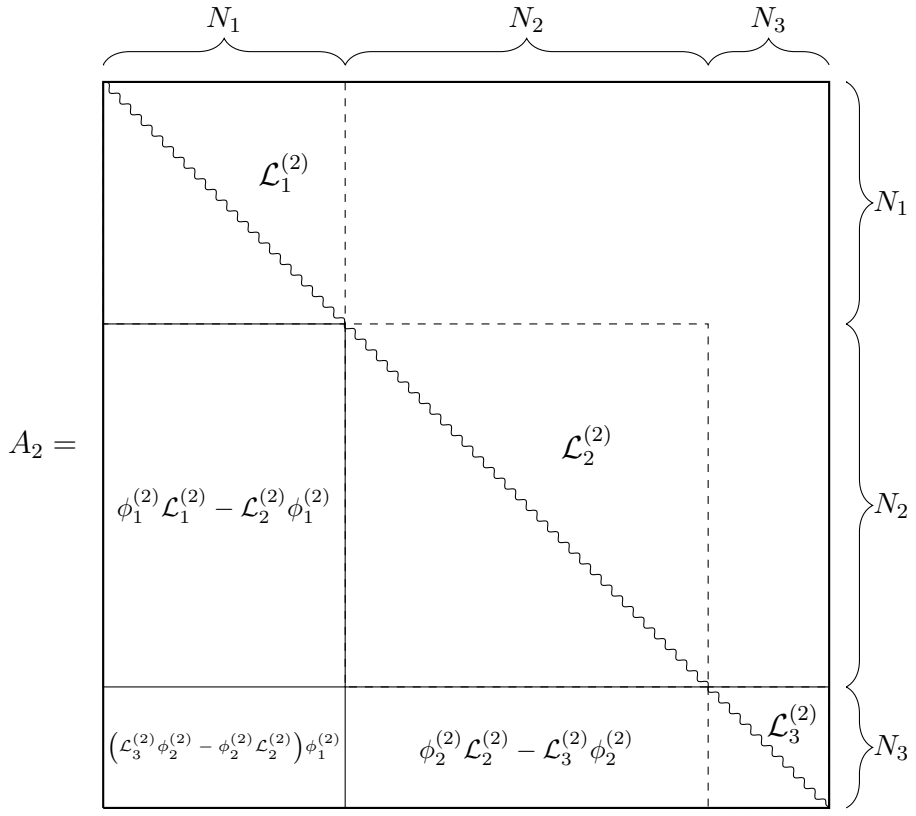

Figura 2.4. Struttura della matrice di adiacenza A_2 codificata da una matrice spettrale Φ_2 con due blocchi sotto-diagonali.

Nel seguito ci riferiremo ai blocchi sotto-diagonali di A_2 con la nomenclatura esposta in Figura 2.5.

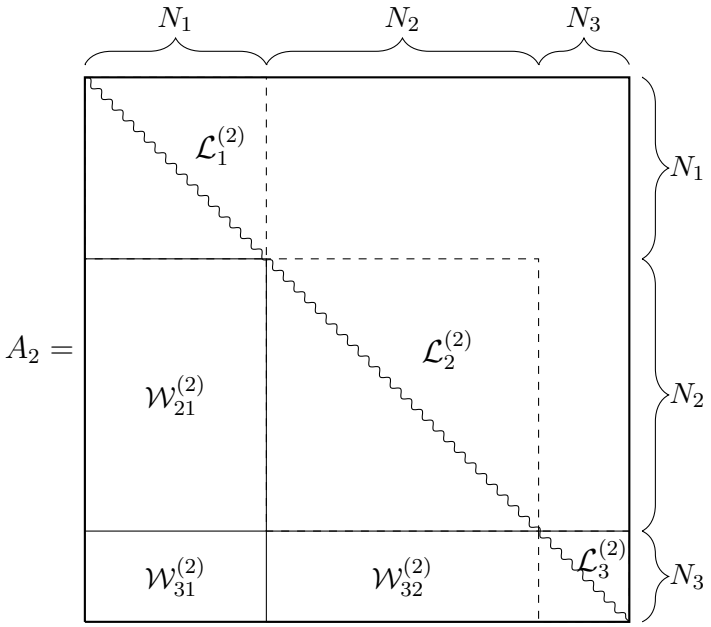

Figura 2.5. Nomenclatura dei blocchi sotto-diagonali della matrice di adiacenza A_2.

Dunque:

$$\mathcal{W}_{21}^{(2)} = \phi_1^{(2)}\mathcal{L}_1^{(2)} - \mathcal{L}_2^{(2)}\phi_1^{(2)} \tag{2.8}$$

$$\mathcal{W}_{32}^{(2)} = \phi_2^{(2)}\mathcal{L}_2^{(2)} - \mathcal{L}_3^{(2)}\phi_2^{(2)} \tag{2.9}$$

$$\mathcal{W}_{31}^{(2)} = \left(\mathcal{L}_3^{(2)}\phi_2^{(2)} - \phi_2^{(2)}\mathcal{L}_2^{(2)}\right)\phi_1^{(2)} \tag{2.10}$$

Già da subito possiamo intuire che la matrice Φ_2 descrive una struttura più ricca di quanto inizialmente previsto: essa è compatibile con 3 strati di neuroni, auto-connessi, ed al contempo caratterizzati anche da 3 blocchi di connessioni feedforward: il primo, $\mathcal{W}_{21}^{(2)}$, connette il primo strato al secondo strato; il secondo, $\mathcal{W}_{32}^{(2)}$, connette il secondo strato al terzo; ed infine $\mathcal{W}_{31}^{(3)}$ connette il primo al terzo.
La topologia delle connessioni fra neuroni è completamente descritta dalla matrice di adiacenza A_2 (a patto di interpretare i nodi del grafo come neuroni), tuttavia A_2 non fornisce alcuna prescrizione esplicita su come la geometria descritta debba essere utilizzata per elaborare informazione. Nel corso delle prossime pagine sarà dunque necessario studiare la topologia neurale derivante dal formalismo spettrale Φ_2 più in dettaglio, con l'intento ultimo di comprendere come sia meglio sfruttare A_2 per connettere l'input $\boldsymbol{x}$ all'output $\boldsymbol{y}$, cercando per quanto possibile di mantenere anche l'analogia con il formalismo spettrale standard Φ_1.

2.3 Interpretazione della Matrice di Adiacenza

Cerchiamo di comprendere in dettaglio la natura del grafo descritto dalla matrice di adiacenza A_2. Abbiamo adesso a che fare con una topologia che si presta ad essere interpretata come un *grafo tripartito*: possiamo pensare i nodi del grafo come distribuiti in 3 strati, dove il primo strato conterà N_1 nodi, il secondo N_2, ed il terzo N_3. Ai fini della rappresentazione per mezzo della matrice di adiacenza i nodi sono numerati a partire dal primo strato, esattamente come nel formalismo spettrale standard. Scendendo nel dettaglio i nodi del primo strato saranno numerati da 1 a N_1, quelli del secondo da $N_1 + 1$ a N_2, ed infine quelli del terzo da $N_2 + 1$ a N_3.
Sempre seguendo la filosofia del formalismo spettrale vorremo interpretare i nodi del grafo come i neuroni di una rete neurale feedforward, questa volta *a 3 layers*. Le attivazioni dei neuroni verranno dunque descritte da un vettore delle attivazioni $\boldsymbol{a} \in \mathbb{R}^N$, dove però questa volta:

$$N = N_1 + N_2 + N_3. \tag{2.11}$$

Si capisce dunque che questo vettore delle attivazioni si presta bene alla descrizione delle attività neurali di una rete a tre layers, esattamente come speravamo. Nello specifico possiamo semplicemente affermare che l'attività dell'i-esimo neurone della rete sarà descritta dall'i-esimo elemento del vettore $\boldsymbol{a}$.
Dato quanto appena esposto se vogliamo tentare di utilizzare il nostro formalismo per descrivere l'attività di una rete neurale feedforward si capisce che inizialmente il vettore delle attivazioni dovrà avere solo i primi N_1 suoi elementi non nulli

$$\boldsymbol{a} = \begin{pmatrix} \boldsymbol{x} \\ \mathbf{0} \\ \tilde{\mathbf{0}} \end{pmatrix} \text{ , dove } \boldsymbol{x} \in \mathbb{R}^{N_1}, \mathbf{0} \in \mathbb{R}^{N_2}, \tilde{\mathbf{0}} \in \mathbb{R}^{N_3} \tag{2.12}$$

il che equivale a dire che inizialmente solo i neuroni nel layer di input sono attivi. Sappiamo che nel formalismo spettrale standard (Φ_1) è l'azione della matrice di adiacenza a propagare le attivazioni del primo layer attraverso la rete; mantenendo questo approccio nel contesto presente ricaviamo il seguente aggiornamento del vettore delle attivazioni:

$$A\boldsymbol{a} = \begin{pmatrix} \mathcal{L}_1^{(2)} \boldsymbol{x} \\ \mathcal{W}_{21}^{(2)} \boldsymbol{x} \\ \mathcal{W}_{31}^{(2)} \boldsymbol{x} \end{pmatrix}. \tag{2.13}$$

Notiamo che l'attività del primo strato di neuroni fluisce attraverso la rete, attivando tutti i neuroni della stessa. Nello specifico gli N_3 neuroni di output acquistano attivazioni $\mathcal{W}_{31}^{(2)} \boldsymbol{x}$. Tuttavia se leggessimo immediatamente l'output della rete dagli ultimi N_3 elementi di $\boldsymbol{a}$, come si fa nel formalismo spettrale standard, staremmo di fatto ignorando l'elaborazione dell'informazione codificata dai blocchi di connessioni feedforward $\mathcal{W}_{21}^{(2)}$ e $\mathcal{W}_{32}^{(2)}$: queste connessioni non hanno avuto ancora modo di propagare il segnale in input alla rete fino all'output, in un certo senso *non hanno fatto in tempo*, in quanto una singola evoluzione dell'attività neurale, codificata da

A_2, porta fino all'output il segnale solo attraverso le connessioni $\mathcal{W}_{31}^{(2)}$, che collegano direttamente il primo strato neurale al terzo strato neurale.

La situazione in esame si presta bene ad una descrizione grafica, si veda la Figura 2.6.

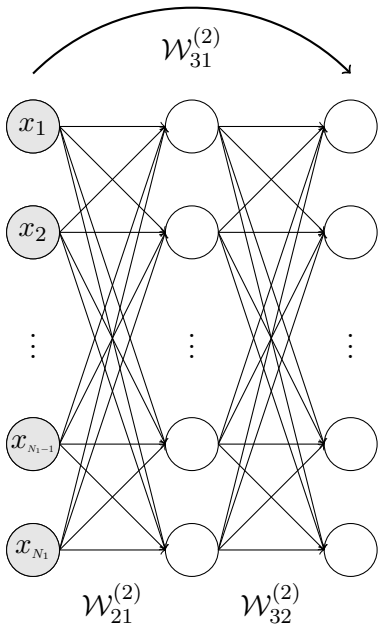

Figura 2.6. Rete neurale (nello stato iniziale) rappresentata dalla matrice di adiacenza A_2. Si noti che per chiarezza pittorica abbiamo scelto di non rappresentare le auto-connessioni, ma esse sarebbero presenti su ogni neurone. Si noti inoltre che, sempre per chiarezza pittorica, abbiamo scelto di rappresentare le connessioni dense dal primo al terzo strato con una singola freccia. Possiamo immediatamente notare come questa struttura, emersa naturalmente dal formalismo spettrale, ricordi quella di un blocco di una *ResNet* [32].

Dalla figura (2.6) si capisce bene che la rete neurale rappresentata da A_2, frutto della matrice spettrale Φ_2, non è una semplice rete neurale feedforward a tre layers come ci potevamo inizialmente aspettare, ma è bensì una rete neurale dove ogni neurone in generale è collegato ad ogni altro neurone *a valle* dello stesso. Se volessimo coniare un nuovo termine per l'occasione potremmo dire che Φ_2 da luogo ad una rete *feedforward superdensamente connessa*, connessioni dense su ogni strato, provenienti da ogni strato precedente. Chiameremo queste reti: *reti spettrali WSLC (With Skip Layer Connections).*

Dalla figura possiamo anche notare bene quello a cui accennavamo qualche paragrafo addietro in (2.13): il segnale in input alla rete ha due modi per giungere fino all'output, il primo è prendere la strada diretta codificata dalla matrice $\mathcal{W}_{31}^{(2)}$, ed il secondo è filtrare attraverso i neuroni dell'hidden layer, attraverso le connessioni pesate da $\mathcal{W}_{21}^{(2)}$ e $\mathcal{W}_{32}^{(2)}$. Tuttavia il secondo percorso, essendo indiretto, non è attuabile da una singola evoluzione delle attivazioni codificata da A_2.

Volendo studiare l'elaborazione dell'informazione codificata dall'interezza della rete esistono diversi modi per ovviare al problema appena esposto: il primo, e forse anche il più elegante, consiste nell'interpretare la matrice di adiacenza come un vero e proprio operatore di evoluzione temporale, in tempo discreto, dello stato delle attivazioni neurali. In questa prospettiva chiameremo le attivazioni iniziali della rete (2.12):

$$\boldsymbol{a}(t=0) = \begin{pmatrix} \boldsymbol{x} \\ \mathbf{0} \\ \tilde{\mathbf{0}} \end{pmatrix}. \tag{2.14}$$

Interpretando i rami pesati del grafo come canali in cui l'attivazione neurale si propaga nel tempo, ed assumendo che l'attivazione impieghi sempre un tempo unitario per propagarsi in un singolo canale, si capisce che lo stato delle attivazioni della rete al tempo $t = 1$ sarà:

$$\boldsymbol{a}(1) = A_2\boldsymbol{a}(0) = \begin{pmatrix} \mathcal{L}_1^{(2)}\boldsymbol{x} \\ \mathcal{W}_{21}^{(2)}\boldsymbol{x} \\ \mathcal{W}_{31}^{(2)}\boldsymbol{x} \end{pmatrix} \tag{2.15}$$

e lo stato delle attivazioni al tempo $t = 2$ sarà:

$$\boldsymbol{a}(2) = A_2\boldsymbol{a}(1) = \begin{pmatrix} \mathcal{L}_1^{(2)}\mathcal{L}_1^{(2)}\boldsymbol{x} \\ \mathcal{W}_{21}^{(2)}\mathcal{L}_1^{(2)}\boldsymbol{x} + \mathcal{L}_2^{(2)}\mathcal{W}_{21}^{(2)}\boldsymbol{x} \\ \mathcal{W}_{31}^{(2)}\mathcal{L}_1^{(2)}\boldsymbol{x} + \mathcal{W}_{32}^{(2)}\mathcal{W}_{21}^{(2)}\boldsymbol{x} + \mathcal{L}_3^{(2)}\mathcal{W}_{31}^{(2)}\boldsymbol{x} \end{pmatrix} \tag{2.16}$$

notiamo quindi che al tempo $t = 2$ il segnale in input si è propagato in output *percorrendo tutti i possibili canali di connessione strettamente feedforward*, ed ha quindi sfruttato tutta la capacità di elaborazione feedforward della rete; questo sembra quindi un buon momento per estrarre l'output dall'ultimo strato neurale, le cui attivazioni, come si vede da (2.16), risultano essere:

$$\boldsymbol{y} = \mathcal{W}_{31}^{(2)}\mathcal{L}_1^{(2)}\boldsymbol{x} + \mathcal{W}_{32}^{(2)}\mathcal{W}_{21}^{(2)}\boldsymbol{x} + \mathcal{L}_3^{(2)}\mathcal{W}_{31}^{(2)}\boldsymbol{x}. \tag{2.17}$$

I tre addendi nell'output (2.17) corrispondono ai 3 possibili percorsi che le attività neurali del primo strato possono sfruttare per propagarsi fino all'output e permanerci fino al tempo $t = 2$;[1] nello specifico dunque le attivazioni $\boldsymbol{x}$ possono

- propagarsi istantaneamente attraverso il canale diretto $\mathcal{W}_{31}^{(2)}$ per poi sostare nello strato di output compiendo un self-loop pesato da $\mathcal{L}_3^{(2)}$;
- oppure propagarsi prima attraverso il canale $\mathcal{W}_{21}^{(2)}$ per poi passare attraverso il canale $\mathcal{W}_{32}^{(2)}$ e giungere nello strato di output;
- oppure inizialmente sostare nello strato di input con il self-loop dato da $\mathcal{L}_1^{(2)}$ per poi proiettarsi direttamente in output attraverso il canale $\mathcal{W}_{31}^{(2)}$.

Una rappresentazione grafica di quanto appena esposto è fornita in Figura 2.7, dove si prende ad esempio una rete neurale concreta codificata da una matrice Φ_2, 9×9, con due blocchi sotto-diagonali concatenati.

[1] Qualche paragrafo addietro avevamo affermato che il segnale in input alla nostra rete potesse propagarsi fino all'output seguendo solo 2 possibili percorsi: uno diretto ed uno indiretto. Tuttavia in realtà, come vedremo adesso, il percorso indiretto è composto da due possibili sotto-percorsi, e dunque gli addendi nell'output si rivelano essere 3. Questo fenomeno è conseguenza diretta del fatto di aver scelto di interpretare la matrice di adiacenza come operatore di evoluzione temporale delle attivazioni.

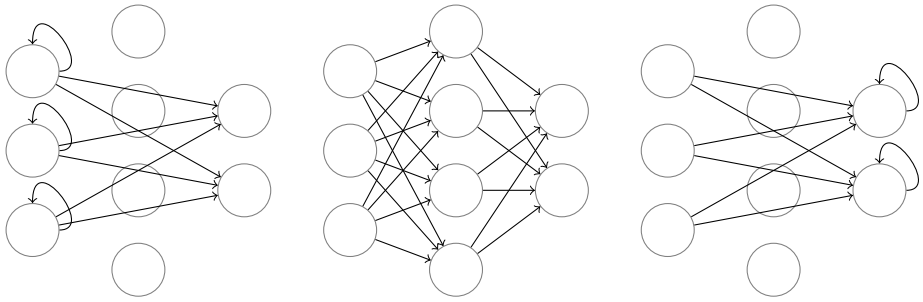

Figura 2.7. In questa figura è riportata una rappresentazione grafica della produzione dell'output espressa in (2.17), e descritta nel precedente elenco puntato. Nello specifico abbiamo scelto di esemplificare la produzione dell'output per mezzo di una rete a 9 neuroni, rappresentata globalmente dunque da una matrice $\Phi_{9\times 9}$ con due blocchi sotto-diagonali concatenati, con dimensioni rispettive 4×3 e 2×4. Si noti come le attività neurali del primo strato possono propagarsi fino all'ultimo seguendo 3 possibili percorsi, corrispondenti ai 3 addendi nell'output.

Chiameremo questo approccio all'utilizzo della matrice di adiacenza A_2 ALLIN, acronimo di *ALl Links In the Network*, in quanto nel ricavare l'evoluzione delle attivazioni vengono considerati tutti i canali codificati dal formalismo spettrale, compresi i self-loops.
Tuttavia se la nostra intenzione è quella di sfruttare il formalismo spettrale per descrivere una rete feedforward appare evidente un altro possibile approccio: ovvero semplicemente interpretare la matrice di adiacenza A_2 come una mappa della rete, da cui estrarre quello che ci interessa, ovvero le connessioni feedforward. Con questa filosofia si capisce che l'output della rete può essere immediatamente scritto come:

$$\boldsymbol{y} = \mathcal{W}_{31}^{(2)}\boldsymbol{x} + \mathcal{W}_{32}^{(2)}\mathcal{W}_{21}^{(2)}\boldsymbol{x}. \tag{2.18}$$

Chiameremo questo approccio FAST, acronimo di *Feedforward Approach to Spectral Topology*.

Approccio SFAST
All'inizio di questo capitolo la nostra intenzione era quella di estendere il formalismo spettrale alla descrizione di reti neurali feedforward con tre layers; portando avanti il nostro studio abbiamo notato come in realtà il formalismo spettrale sia adeguato alla descrizione di una struttura neurale *superdensamente connessa*, descritta nelle pagine precedenti. Nel seguito tuttavia avremo modo di vedere come questa struttura possa essere manipolata, ed eventualmente ricondotta a quella di una classica rete neurale feedforward a tre layers attraverso opportuni vincoli sui valori degli autovalori in Λ_2. Tuttavia possiamo anche notare che l'approccio concettuale esposto per introdurre FAST possa essere applicato anche in altra maniera per ottenere la struttura che andavamo ricercando all'inizio di questo capitolo: in particolare se fossimo interessati unicamente ad architetture feedforward classiche, e dunque prive sia di self-loops che di connessioni skip layer, potremmo semplicemente estrarre dalla matrice di adiacenza unicamente i blocchi di connessione di nostro interesse, ovvero i blocchi $\mathcal{W}_{21}$ e $\mathcal{W}_{32}$, per poi

utilizzarli come matrici dei pesi come si farebbe in una normale rete feedforward. In questa maniera il formalismo spettrale cesserebbe di avere implicazioni sulla topologia della rete e diverrebbe unicamente un modo per introdurre una *parametrizzazione spettrale* dei pesi di una normale rete feedforward a 3 layers. Chiameremo questo approccio SFAST, acronimo di *Strictly Feedforward Approach to Spectral Topology*. Si potrebbe pensare che questo approccio sia equivalente alla parametrizzazione dei due blocchi di connessione per mezzo del metodo spettrale standard esposto nel precedente capitolo, tuttavia questo non è vero, in quanto la descrizione tramite una singola matrice Φ_2 (e dunque anche una singola matrice Λ_2) di tutta la rete che si ha in SFAST implica una *interconnessione* nella struttura dei pesi che non avremmo parametrizzando con due separate matrici Φ_1. Nello specifico con la parametrizzazione per mezzo di due Φ_1 (Φ_1', Φ_1'') le due matrici dei pesi sarebbero date da:

$$\begin{aligned} \mathcal{W}_{21} &= \phi' \mathcal{L}_1' - \mathcal{L}_2' \phi' \\ \mathcal{W}_{32} &= \phi'' \mathcal{L}_1'' - \mathcal{L}_2'' \phi''. \end{aligned} \tag{2.19}$$

Di contro descrivendo la rete con la singola Φ_2 le due matrici sono date da (2.8) e (2.9), dove possiamo notare la dipendenza intrecciata data dalla presenza di $\mathcal{L}_2^{(2)}$ in entrambe le espressioni. Questa osservazione era dovuta, per motivi di consistenza nella trattazione. Nel seguito abbiamo scelto di non approfondire ulteriormente la questione se non limitatamente alla caratterizzazione delle scelte sui parametri che possono essere operate per ricreare una struttura di tipo SFAST.

2.3.1 Introduzione delle non linearità

Se concludessimo qui la nostra trattazione avremmo descritto una rete neurale capace di esplorare uno spazio delle ipotesi strettamente lineare, cosa ovviamente non ottimale.
Nell'approccio FAST possiamo rapidamente ovviare al problema come si fa normalmente nella letteratura inerente le reti neurali, ovvero introducendo una non linearità *elementwise* σ dopo ogni blocco di connessioni feedforward. In FAST possiamo dunque semplicemente scrivere l'output della rete come:

$$\boldsymbol{y} = \mathcal{W}_{31}^{(2)} \boldsymbol{x} + \mathcal{W}_{32}^{(2)} \sigma(\mathcal{W}_{21}^{(2)} \boldsymbol{x}). \tag{2.20}$$

Nell'approccio ALLIN la questione è più sottile: possiamo pensare di trattare il problema modificando leggermente ciò che consideriamo l'operatore di evoluzione temporale della rete. Nello specifico possiamo definire l'operatore di evoluzione temporale come:

$$\sigma \circ A_2 \tag{2.21}$$

con σ non linearità applicata *elementwise*, come abbiamo già detto. L'elaborazione dell'informazione in input alla rete può essere allora scritta semplicemente come:

$$\boldsymbol{a}(2) = \left[\sigma \circ A_2\right]^{\circ 2} \boldsymbol{a}(0) \tag{2.22}$$

oppure, se si preferisce la scrittura più esplicita

$$\boldsymbol{y} = \sigma\Big(\mathcal{W}_{31}^{(2)}\sigma(\mathcal{L}_1^{(2)}\boldsymbol{x}) + \mathcal{W}_{32}^{(2)}\sigma(\mathcal{W}_{21}^{(2)}\boldsymbol{x}) + \mathcal{L}_3^{(2)}\sigma(\mathcal{W}_{31}^{(2)}\boldsymbol{x})\Big) \tag{2.23}$$

Per chiarezza: in (2.22) con l'apice $\circ 2$ vogliamo indicare *la composizione* della funzione fra parentesi con se stessa.
L'approccio all'introduzione della non linearità appena esposto è ottimale se si ha l'intenzione di usare la topologia di A_2 come un blocco di una rete neurale più grande, creando architetture affini a quelle di una *ResNet* [32]. Se invece si avesse intenzione di usare A_2 come una rete neurale *standalone*, da cui leggere direttamente l'output, allora la seconda non linearità è ovviamente di troppo[2], e l'output della rete dovrà essere scritto come:

$$\boldsymbol{a}(2) = A_2 \circ \sigma \circ A_2\boldsymbol{a}(0), \tag{2.24}$$

oppure nella forma esplicita come

$$\boldsymbol{y} = \mathcal{W}_{31}^{(2)}\sigma(\mathcal{L}_1^{(2)}\boldsymbol{x}) + \mathcal{W}_{32}^{(2)}\sigma(\mathcal{W}_{21}^{(2)}\boldsymbol{x}) + \mathcal{L}_3^{(2)}\sigma(\mathcal{W}_{31}^{(2)}\boldsymbol{x}). \tag{2.25}$$

Se vogliamo trattare quest'ultimo caso mantenendo il framework concettuale dell'operatore di evoluzione temporale siamo costretti a fare una distinzione fra l'evoluzione temporale delle attivazioni *in fase di elaborazione*, ovvero prima dell'estrazione dell'output, e l'evoluzione temporale corrispondente *all'estrazione* di $\boldsymbol{y}$, ovvero l'ultima. L'unica differenza fra queste due evoluzioni è la presenza della non linearità.
Il fatto che l'evoluzione delle attivazioni subito prima dell'output debba essere distinta dalle altre non dovrebbe sorprendere: invero possiamo notare che anche il modo in cui vengono attivati i neuroni di input all'inizio del processo di elaborazione dell'informazione è distinto da quello di tutti gli altri neuroni. All'inizio ai neuroni di input vengono associate attivazioni corrispondenti ad un qualche *encoding* dell'oggetto d'interesse (*e.g.* immagine), e questo processo di attribuzione delle attivazioni non può ovviamente essere descritto per mezzo della matrice di adiacenza della rete. Durante la fase di estrazione dell'output le attivazioni non si propagano ulteriormente nella rete, e vengono invece collezionate ed utilizzate (ad esempio attraverso una *softmax*). Si può capire che ci si debba aspettare un comportamento anomalo agli estremi del processo di elaborazione descritto dalla rete, ed il fatto che sia necessario definire un operatore differente per l'ultimo step dell'evoluzione delle attivazioni rispecchia questo fatto.
Per non esporci a possibili ambiguità nel seguito sceglieremo di lavorare considerando la rete prodotta dal formalismo spettrale come una rete *standalone*, e non come un blocco di una rete più grande; dunque in futuro ometteremo sempre l'ultima non linearità, e scriveremo l'output della rete come fatto in (2.20) o (2.25). Inoltre nel seguito indicheremo l'operatore di evoluzione temporale del formalismo ALLIN con $\mathcal{E}$, e la sua versione di output priva di non linearità con $\tilde{\mathcal{E}}$.

[2] In letteratura è comune non associare una non linearità ai neuroni di output. Se associassimo una *ReLU* all'output staremmo imponendo la positività di tutti gli elementi di $\boldsymbol{y}$, cosa alle volte non desiderabile. Per fare un esempio concreto si pensi di voler utilizzare la rete neurale per problemi di regressione: vorremmo un singolo neurone di output la cui attivazione verrebbe interpretata come l'output del modello regressivo, in questo caso è palese che associare una *ReLU* all'output infici il funzionamento della rete.

2.3.2 Introduzione del bias

Come sappiamo in generale le reti neurali possono avere un termine di bias associato ad ogni neurone. Per introdurre questo termine nel nostro formalismo potremmo semplicemente ammettere la presenza di un vettore di bias $\boldsymbol{b}_i$ per ognuno degli strati di neuroni non di input, e scrivere l'output della rete tenendo conto della presenza della somma con il vettore di bias prima dell'applicazione della non linearità; questo approccio concettualmente si presta molto bene a FAST:

$$\boldsymbol{y} = \mathcal{W}_{31}^{(2)}\boldsymbol{x} + \mathcal{W}_{32}^{(2)}\sigma(\mathcal{W}_{21}^{(2)}\boldsymbol{x} + \boldsymbol{b}_2) + \boldsymbol{b}_3. \tag{2.26}$$

Per ALLIN ancora una volta la questione è più complicata in quanto vogliamo rappresentare l'azione della rete per mezzo dell'operatore di evoluzione delle attivazioni. Una prima idea potrebbe semplicemente essere quella di ridefinire l'operatore $\mathcal{E}$ come segue:

$$\mathcal{E}\boldsymbol{a}(t) \doteq \sigma \circ (A_2\boldsymbol{a}(t) + \boldsymbol{b}) \quad , \quad t \in \mathbb{N} \tag{2.27}$$

e dunque anche:

$$\tilde{\mathcal{E}}\boldsymbol{a}(t) \doteq A_2\boldsymbol{a}(t) + \boldsymbol{b} \quad , \quad t \in \mathbb{N} \tag{2.28}$$

con $\boldsymbol{b} \in \mathbb{R}^N$, dove $N = N_1 + N_2 + N_3$ è come al solito il numero di neuroni presenti nella rete. Fissando i primi N_1 elementi di $\boldsymbol{b}$ a zero, e chiamando $\boldsymbol{b}_2$ gli elementi da $N_1 + 1$ a N_2, e $\boldsymbol{b}_3$ gli elementi da $N_2 + 1$ a N_3, la forma dell'output della rete potrà come al solito essere dedotta considerando i due step di evoluzione temporale:

$$\boldsymbol{a}(t=1) = \sigma \circ (A_2\boldsymbol{a}(t=0) + \boldsymbol{b}) = \sigma \begin{pmatrix} \mathcal{L}_1^{(2)}\boldsymbol{x} \\ \mathcal{W}_{21}^{(2)}\boldsymbol{x} + \boldsymbol{b}_2 \\ \mathcal{W}_{31}^{(2)}\boldsymbol{x} + \boldsymbol{b}_3 \end{pmatrix} \tag{2.29}$$

ed applicando poi $\tilde{\mathcal{E}}$ per l'ultima evoluzione temporale:

$$\boldsymbol{a}(t=2) = \begin{pmatrix} \mathcal{L}_1^{(2)}\sigma(\mathcal{L}_1^{(2)}\boldsymbol{x}) \\ \mathcal{L}_2^{(2)}\sigma(\mathcal{W}_{21}^{(2)}\boldsymbol{x} + \boldsymbol{b}_2) + \mathcal{W}_{21}^{(2)}\sigma(\mathcal{L}_1^{(2)}\boldsymbol{x}) + \boldsymbol{b}_2 \\ \mathcal{L}_3^{(2)}\sigma(\mathcal{W}_{31}^{(2)}\boldsymbol{x} + \boldsymbol{b}_3) + \mathcal{W}_{32}^{(2)}\sigma(\mathcal{W}_{21}\boldsymbol{x} + \boldsymbol{b}_2) + \mathcal{W}_{31}^{(2)}\sigma(\mathcal{L}_1^{(2)}\boldsymbol{x}) + \boldsymbol{b}_3 \end{pmatrix}. \tag{2.30}$$

Possiamo adesso estrarre l'output come al solito:

$$\boldsymbol{y} = \mathcal{L}_3^{(2)}\sigma(\mathcal{W}_{31}^{(2)}\boldsymbol{x} + \boldsymbol{b}_3) + \mathcal{W}_{32}^{(2)}\sigma(\mathcal{W}_{21}\boldsymbol{x} + \boldsymbol{b}_2) + \mathcal{W}_{31}^{(2)}\sigma(\mathcal{L}_1^{(2)}\boldsymbol{x}) + \boldsymbol{b}_3. \tag{2.31}$$

Notiamo dunque che la modifica apportata all'operatore di evoluzione temporale equivale all'associare ad ogni neurone della rete (non di input) un bias $b_i \in \mathbb{R}$, lasciando tutto il resto inalterato.

Tuttavia se volessimo mantenere tutta l'informazione sulla dinamica della rete racchiusa nella matrice di adiacenza potremmo, come già detto nello scorso capitolo, adattare al formalismo spettrale un trucco molto utilizzato in machine learning, ovvero l'aggiunta di un neurone costante, con attivazione sempre unitaria. Questo

neurone costante attraverso le sue connessioni pesate agisce come un effettivo termine di bias su tutti i neuroni a cui è connesso.
Nel seguito, per non complicare la trattazione del tema, ci occuperemo unicamente di reti neurali prive di bias.

2.4 Alcune Interessanti Inizializzazioni

Le performance di una rete neurale durante l'addestramento sono fortemente influenzate da come i pesi vengono inizializzati, ed il formalismo spettrale in questo non fa eccezione. La cosa non dovrebbe sorprendere in quanto i metodi di ottimizzazione iterativi come SGD o Adam hanno naturalmente bisogno di un punto di partenza ragionevole per avere speranza di collassare su una regione ottimale in tempi brevi. La scelta di appropriate inizializzazioni per i valori dei parametri spettrali sarà dunque cruciale al fine di ottenere performance rappresentative del reale potenziale del nostro formalismo.
Il tema della scelta di una corretta inizializzazione dei parametri per una rete neurale è notoriamente complesso, ed oggetto di ricerca attiva. In questa sede non cercheremo di fornire delle inizializzazioni dimostrabilmente eccelse per il nostro contesto di learning, e ci limiteremo ad esporne alcune particolarmente interessanti, senza portare avanti alcuna pretesa di ottimalità.
L'inizializzazione di un'architettura descritta con il formalismo spettrale è naturalmente bipartita: dovremo specificare sia come inizializzare la matrice degli autovettori Φ sia come inizializzare la matrice degli autovalori Λ; dato che alcuni elementi di queste matrici sono fissati dal formalismo nel concreto questo si traduce nel dover specificare i valori all'interno dei blocchi sotto-diagonali ϕ_i di Φ ed i valori all'interno della diagonale della matrice Λ (che verrà naturalmente pensata come composta dalle diagonali delle $\mathcal{L}_i$).
Dato che le conseguenze derivanti da una particolare inizializzazione saranno dipendenti dalla scelta dell'approccio utilizzato per l'interpretazione della matrice di adiacenza divideremo questa sezione in due parti, una per ciascun approccio.

2.4.1 Inizializzazioni per ALLIN

Iniziamo notando che se poniamo:

$$\begin{cases} \mathcal{L}_1^{(2)} = \mathbb{O} \\ \mathcal{L}_2^{(2)} = \mathbb{I} \\ \mathcal{L}_3^{(2)} = \mathbb{O} \end{cases} \tag{2.32}$$

otteniamo, come possiamo facilmente verificare dalla (2.25), un output che ha struttura

$$\boldsymbol{y} = \mathcal{W}_{32}\sigma(\mathcal{W}_{21}\boldsymbol{x}). \tag{2.33}$$

Possiamo notare come l'inizializzazione proposta di fatto converta la struttura *super-densa* della rete spettrale A_2 nella forma tipica per un normale blocco feedforward;

in alternativa possiamo dire che (2.32) trasmuta un blocco di tipo *ResNet* in un normale blocco di connessione a 3 layers.

Notiamo che avremmo potuto ottenere la struttura esposta in (2.33) inizializzando la matrice diagonale $\mathcal{L}_2^{(2)}$ in tante diverse maniere: quello che conta è unicamente l'azzeramento delle matrici $\mathcal{L}_1^{(2)}, \mathcal{L}_3^{(2)}$; tuttavia inizializzare $\mathcal{L}_2^{(2)} = \mathbb{I}$ è preferibile a causa di quanto segue. Ricordando la struttura dei blocchi di pesi nel formalismo spettrale (2.8),(2.9), e tenendo conto che $\mathcal{L}_2^{(2)} = \mathbb{I}$, possiamo esplicitare la (2.33) come

$$\boldsymbol{y} = \mathcal{W}_{32}\sigma(\mathcal{W}_{21}\boldsymbol{x}) = \phi_2^{(2)}\sigma(-\phi_1^{(2)}\boldsymbol{x}) \tag{2.34}$$

Questo è interessante in quanto ci consente di creare un ponte con la normale teoria delle reti neurali feedforward: sappiamo che normalmente i parametri della rete sono gli elementi delle matrici dei pesi, e dunque la maggior parte della teoria in letteratura su come sia meglio inizializzare i parametri di rete è pensata per operare direttamente sulle matrici delle connessioni pesate; questo normalmente implica che essa non sia applicabile al nostro formalismo, in quanto i parametri della nostra rete sono i blocchi delle matrici Φ e Λ, e non direttamente le matrici $\mathcal{W}$. Tuttavia con (2.32) la rete spettrale si comporta (inizialmente) esattamente come una rete neurale feedforward standard con matrici dei pesi $\phi_1^{(2)}$ e $\phi_2^{(2)}$, e dunque la teoria standard sull'inizializzazione dei pesi è (almeno parzialmente) applicabile anche al nostro formalismo. Alla luce di questo sceglieremo di inizializzare i blocchi sotto-diagonali della matrice degli autovettori Φ_2 con le normali tecniche di inizializzazione per i blocchi di connessione delle reti neurali presenti in letteratura; nello specifico tenderemo ad utilizzare la celebre *Xavier normal*: dove i valori dei singoli elementi della matrice vengono inizializzati secondo una distribuzione gaussiana con media 0 (e varianza dettata dalle dimensioni del blocco) [28].

Dato che (2.32) è un'inizializzazione ci chiediamo se il processo di addestramento possa accendere le connessioni skip layer inizialmente spente nel caso in cui ciò si riveli utile alla minimizzazione della loss; tuttavia la situazione è più complessa di quanto sembri: l'aver posto alcuni parametri a zero potrebbe annullare gli elementi corrispondenti del gradiente, rendendo impossibile l'evoluzione dei parametri a valori diversi da zero. Se avessimo a che fare con una normale rete neurale feedforward addestrata nello spazio diretto, con attivazioni $ReLU$, sicuramente andremmo incontro alla problematica descritta sopra: il processo di addestramento non riuscirebbe mai a far evolvere pesi inizialmente fissati a zero; tuttavia nel nostro caso di addestramento spettrale il responso non è così scontato.

Approfondimento sulla funzione di attivazione ReLU
Nel capitolo precedente abbiamo già esposto la struttura della funzione *ReLU* in (1.40); in questa sede notiamo che essa è anche equivalentemente esprimibile come segue:

$$\sigma(x) = \max(0, x) \tag{2.35}$$

e si può subito notare che questa funzione presenta un punto di non derivabilità in $x = 0$. Tuttavia la presenza di questo punto in una funzione di attivazione può creare problemi in fase di ottimizzazione iterativa. Per questo motivo nelle moderne implementazioni è comune modificare leggermente la definizione della *ReLU* complementando la (1.40) con la specifica:

$$\sigma'(0) = 0. \tag{2.36}$$

Il punto di non derivabilità viene cioè *"rimosso manualmente"*, ponendo per definizione a zero il valore della derivata di σ in $x = 0$. Nel seguito ci adegueremo a questa definizione modificata [13].

Per fare chiarezza su questo tema prendiamo in esame il metodo della discesa del gradiente:

$$p_i = \tilde{p}_i - \gamma \left[\frac{\partial}{\partial p_i} R_f \right]_{\tilde{f}} \tag{2.37}$$

abbiamo scritto la regola di aggiornamento per un singolo parametro p_i del modello f (il modello f è la legge che connette l'input all'output); con γ *learning rate*, R rischio empirico, e dove la tilde vuole indicare i valori prima dell'aggiornamento.
Affinché la derivata parziale del rischio empirico non si annulli è necessario che la derivata dell'output y del nostro modello non si annulli; prendendo in esame uno dei parametri d'interesse, ad esempio il primo elemento di $\mathcal{L}_1^{(2)}$ inizialmente posto a zero, che chiameremo λ_1, la quantità rilevante è allora:

$$\left[\frac{\partial}{\partial \lambda_1} \boldsymbol{y} \right]_{\lambda_1=0} = \left[\frac{\partial}{\partial \lambda_1} \left[\mathcal{W}_{31}^{(2)} \sigma(\mathcal{L}_1^{(2)} \boldsymbol{x}) + \mathcal{W}_{32}^{(2)} \sigma(\mathcal{W}_{21}^{(2)} \boldsymbol{x}) + \mathcal{L}_3^{(2)} \sigma(\mathcal{W}_{31}^{(2)} \boldsymbol{x}) \right] \right]_{\lambda_1=0} \tag{2.38}$$

possiamo notare una derivata di tre addendi: la prima si annulla senz'altro in quanto il valore della derivata della *ReLU* in zero è nullo (si veda il precedente inserto),[3] l'ultima è irrilevante in quanto non contiene alcuna dipendenza da $\mathcal{L}_1^{(2)}$, dunque ci rimane da studiare solo:

$$\left[\frac{\partial}{\partial \lambda_1} \boldsymbol{y} \right]_{\lambda_1=0} = \left[\frac{\partial}{\partial \lambda_1} \mathcal{W}_{32}^{(2)} \sigma(\mathcal{W}_{21}^{(2)} \boldsymbol{x}) \right]_{\lambda_1=0} = \mathcal{W}_{32}^{(2)} \left[\frac{\partial}{\partial \lambda_1} \sigma(\mathcal{W}_{21}^{(2)} \boldsymbol{x}) \right]_{\lambda_1=0} \tag{2.39}$$

[3] Questo sarebbe stato esattamente il problema in cui saremmo incappati se avessimo avuto a che fare con una rete nello spazio diretto, tuttavia nel nostro caso spettrale dobbiamo considerare altri due termini nella derivata.

ovviamente $\mathcal{W}_{32}^{(2)}$ è irrilevante per l'annullamento, ed esplicitando $\mathcal{W}_{21}^{(2)}$ la quantità d'interesse è:

$$\left[\frac{\partial}{\partial\lambda_1}\sigma\Big(\phi_1^{(2)}\mathcal{L}_1^{(2)}\boldsymbol{x} - \mathcal{L}_2^{(2)}\phi_1^{(2)}\boldsymbol{x}\Big)\right]_{\lambda_1=0} \tag{2.40}$$

E possiamo notare che questa derivata può essere diversa da zero; questo ragionamento può essere chiaramente esteso ad ogni parametro λ_i di $\mathcal{L}_1^{(2)}$. Se avessimo invece preso in esame un elemento diagonale di $\mathcal{L}_3^{(2)}$, che chiameremo λ_i', avremmo potuto notare dalla struttura esposta in (2.38) che l'ultima derivata sarebbe stata potenzialmente diversa da zero:

$$\frac{\partial}{\partial\lambda_i'}\left[\mathcal{L}_3^{(2)}\sigma\big(\mathcal{W}_{31}^{(2)}\boldsymbol{x}\big)\right]_{\lambda_1=0} \neq 0 \quad \forall i \tag{2.41}$$

L'insieme di queste osservazione fa capire come il paradigma dell'apprendimento spettrale sia in grado di attivare le connessioni skip layer anche se i valori degli autovalori sono inizialmente fissati a zero.

Per completezza notiamo che in letteratura non è solo sconsigliato fissare parametri a zero, ma è anche sconsigliato assegnare a diversi parametri lo stesso identico valore, in quanto questo potrebbe creare simmetrie indesiderate nel processo di ottimizzazione. Tuttavia anche in questo caso il formalismo spettrale sembra essere resistente a questa problematica, in quanto gli autovalori sono sempre accoppiati agli autovettori, che vengono inizializzati gaussianamente; questo sembra rompere la simmetria, evitando il problema.

Chiameremo l'inizializzazione esposta in (2.32) CHAIN,[4] in quanto con questa inizialmente il flusso di informazione attraverso la rete percorre tutta la *catena* di strati neurali dall'input fino all'output, senza utilizzare le connessioni skip layer.

Se volessimo utilizzare il formalismo spettrale per effettuare training esattamente come si fa nel formalismo standard basterebbe *fissare* quanto specificato in (2.32) anziché porlo come condizione iniziale; questo sembra suggerire che il formalismo spettrale sia un'estensione del formalismo standard, che appare difatti racchiuso in esso ed utilizzabile attraverso l'applicazione di opportuni vincoli ai parametri spettrali.

Notiamo in coda che il meno nella (2.32) può essere quasi sempre ignorato in quanto le inizializzazioni tendono ovviamente ad essere simmetriche per inversione di segno; siamo quindi di fatto autorizzati a scrivere:

$$\boldsymbol{y} = \phi_2^{(2)}\sigma\big(\phi_1^{(2)}\boldsymbol{x}\big). \tag{2.42}$$

[4]Questo nome vuole essere un acronimo ricorsivo: CHAIN sta per *CHAin INitialization*.

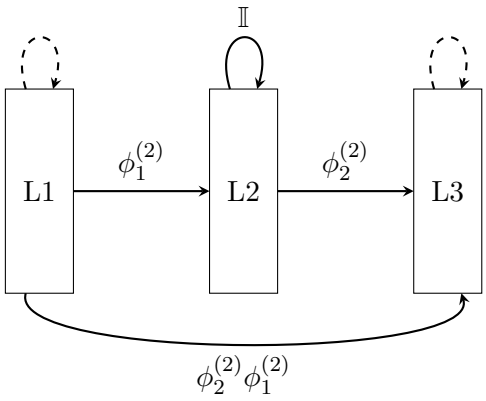

Figura 2.8. Rappresentazione dell'inizializzazione CHAIN; si noti come la connessione skip layer sia attiva ma tuttavia non contribuisca all'output finale della rete (2.42) in quanto le auto-connessioni dei neuroni coinvolti sono nulle: il segnale si propaga direttamente dall'input all'output durante la prima evoluzione temporale, ma non ha modo di permanere fino alla fase di estrazione, che avviene dopo la seconda evoluzione temporale.

Per ALLIN vogliamo anche trattare l'inizializzazione GROWTH,[5] che consiste in quanto segue:

$$\begin{cases} \mathcal{L}_1^{(2)} = \mathbb{O} \\ \mathcal{L}_2^{(2)} = \mathbb{O} \\ \mathcal{L}_3^{(2)} = \mathbb{I} \end{cases} \tag{2.43}$$

Con questa, come si evince dalla (2.25), otteniamo un output iniziale con la seguente struttura:

$$\boldsymbol{y} = \sigma(\mathcal{W}_{31}\boldsymbol{x}) = \sigma(\phi_2^{(2)}\boldsymbol{x}). \tag{2.44}$$

Si noti che anche in questo caso i blocchi sotto-diagonali della matrice degli autovettori saranno inizializzati in maniera standard (con *Xavier normal* ad esempio). Come già detto la presenza di una non linearità sull'output non è ideale, tuttavia l'addestramento potrà subito modificare gli autovalori in $\mathcal{L}_3^{(2)}$, rendendo ad esempio possibile un output a valori negativi.

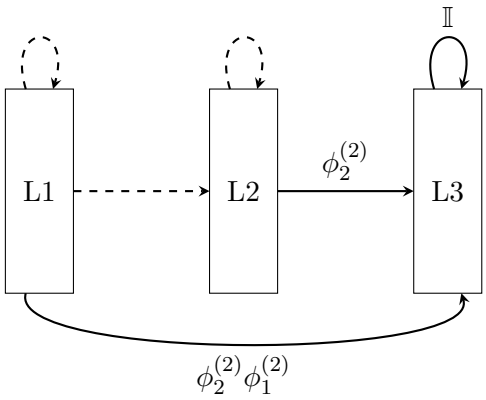

Figura 2.9. Rappresentazione dell'inizializzazione GROWTH. Si noti come inizialmente l'input sia collegato all'output unicamente attraverso la connessione diretta pesata da $\mathcal{W}_{31}^{(2)}$, e come quindi la topologia sia di fatto quella di una rete a due layers. Tuttavia l'addestramento dei parametri spettrali potrà modificare la topologia effettiva della rete, attivando gli elementi di $\mathcal{W}_{21}^{(2)}$. In questo senso l'architettura di rete *cresce* durante la fase di train.

[5] Anche questo nome vuole essere un acronimo ricorsivo: *GRowth Of Weights Through Hierarchies*; in riferimento al fatto che, come vedremo, con questa inizializzazione le connessioni della rete partono con una struttura minimale che viene poi ampliata gerarchicamente durante il processo di addestramento.

Come nella scorsa sezione sorge spontaneo chiedersi se l'annullamento dei parametri spettrali in $\mathcal{L}_1^{(2)}$ e $\mathcal{L}_2^{(2)}$ infici il processo di addestramento degli stessi a causa dell'annullamento degli elementi del gradiente associati. Ebbene possiamo effettivamente notare dalla struttura riportata in (2.38) che le derivate rispetto agli elementi diagonali di $\mathcal{L}_1^{(2)}$ sono tutte nulle, e questo sembra implicare che i λ_i rimarranno fissi a 0. Tuttavia considerando le derivate rispetto agli elementi diagonali di $\mathcal{L}_2^{(2)}$ si vede subito che esse non si annullano (nello specifico l'ultimo dei 3 addendi ha derivata diversa da zero). Da questo segue che gli elementi di $\mathcal{L}_2^{(2)}$ verranno ottimizzati in fase di train, e si può facilmente verificare che questo rende anche il valore delle derivate rispetto agli elementi di $\mathcal{L}_1^{(2)}$ diverso da 0.

Dal ragionamento appena esposto si evince che durante l'addestramento saranno inizialmente attivati gli autovalori corrispondenti a $\mathcal{L}_2^{(2)}$, e solo successivamente si potranno accendere quelli di $\mathcal{L}_1^{(2)}$. Questo fenomeno motiva il nome che abbiamo scelto per questa inizializzazione, e mostra che nonostante l'annullamento di alcuni parametri anche in questo caso il formalismo spettrale riesce a modificare l'architettura di rete.

Maggiori dettagli su questa inizializzazione saranno forniti nelle prossime sezioni.

2.4.2 Inizializzazioni per FAST

Nel caso di FAST l'inizializzazione CHAIN non porta al risultato desiderato da cui prendeva il nome, ovvero le connessioni skip layer non vengono inizialme difatti l'output della rete con la (2.32) risulta essere

$$\boldsymbol{y} = \mathcal{W}_{32}\sigma(\mathcal{W}_{21}\boldsymbol{x}) + \mathcal{W}_{31}\boldsymbol{x} \tag{2.45}$$

e dunque le connessioni skip layer non vengono inizialmente annullate.

Questo accade poiché nel caso precedente era la particolare struttura dell'aggiornamento delle attivazioni a consentire la soppressione delle connessioni skip layer, tramite l'annullamento delle matrici $\mathcal{L}_1^{(2)}, \mathcal{L}_3^{(2)}$. Le matrici dei pesi $\mathcal{W}$ non erano mai propriamente nulle, ma il loro contributo veniva soppresso dalle $\mathcal{L}_i$, cosa che in FAST non può accadere.

Data la struttura dell'output di FAST e la struttura delle matrici $\mathcal{W}$ si capisce come questo ragionamento possa essere facilmente generalizzato per affermare che non possa esistere un'inizializzazione attuabile in FAST che consenta di ottenere una struttura *a catena* come quella che avevamo in ALLIN: difatti questo richiederebbe inizializzare $\mathcal{L}_i, \phi_i$ in maniera tale che $\mathcal{W}_{31}$ sia la sola matrice nulla, ma dalle (2.8),(2.9), (2.10) si capisce che questo implica annullare o una delle ϕ_i o entrambi i blocchi $\mathcal{L}_2, \mathcal{L}_3$, ed ognuna di queste opzioni annulla almeno una delle matrici $\mathcal{W}_{21}, \mathcal{W}_{32}$.

Sebbene riprodurre il funzionamento di una classica rete feedforward in FAST sia impossibile nulla ci vieta di ottenere la struttura caratteristica di GROWTH, e nello specifico essa è ottenibile esattamente come in ALLIN:

$$\begin{cases} \mathcal{L}_1^{(2)} = \mathbb{O} \\ \mathcal{L}_2^{(2)} = \mathbb{O} \\ \mathcal{L}_3^{(2)} = \mathbb{I} \end{cases}$$

Possiamo difatti facilmente verificare che questo porta a:

$$\boldsymbol{y} = \mathcal{W}_{31}\boldsymbol{x} = \phi_2^{(2)}\boldsymbol{x}.$$

Notiamo che anche in questo caso la derivata rispetto agli elementi di $\mathcal{L}_2^{(2)}$, inizialmente nulli, è diversa da zero. Nello specifico il mancato annullamento è causato dalla presenza del termine lineare in (2.45). Lo spostamento della diagonale di $\mathcal{L}_2^{(2)}$ a valori diversi da zero renderà poi possibile anche l'evoluzione degli elementi di $\mathcal{L}_1^{(2)}$, in maniera analoga a quanto già visto per ALLIN-GROWTH.

2.4.3 Approfondimento sull'inizializzazione GROWTH

Come abbiamo notato nelle scorse pagine l'inizializzazione GROWTH è applicabile ad entrambe le interpretazioni della matrice di adiacenza (ALLIN e FAST), e già per questo risulta comoda. Tuttavia la ragione principale che rende questa inizializzazione particolarmente interessante è che essa in un certo senso permette al processo di train di scegliere non solo il valore dei pesi ma anche la struttura e la profondità della topologia di rete.
Una rete spettrale A_2 con GROWTH ha inizialmente attivo unicamente il canale diretto che connette l'input all'output ($\mathcal{W}_{31}$); guardando la struttura delle matrici dei pesi nel formalismo spettrale si potrebbe argomentare che con (2.43) anche $\mathcal{W}_{32}$ è diversa da zero, tuttavia essa inizialmente non può influenzare l'output in quanto il suo contributo è sempre dipendente dalla presenza di $\mathcal{W}_{21}$, che tuttavia è inizialmente fissata a zero. Il fatto che $\mathcal{W}_{32}$ sia diversa da zero è tuttavia cruciale in quanto essa è in un certo senso *pronta* a ricevere il flusso d'informazione proveniente da $\mathcal{W}_{21}$ nel momento in cui il processo di train porti all'attivazione di alcuni suoi elementi. Questo è importante in quanto l'attivazione di $\mathcal{W}_{21}$ comporterà un immediato contributo sull'output del modello, permettendo quindi al processo di ottimizzazione iterativo di operare meglio. Si noti come questo processo di fatto modifichi l'architettura di rete: inizialmente i neuroni nel layer di mezzo (*hidden layer*) non sono effettivamente presenti nella rete, in quanto non hanno alcun effetto sull'output, e potrebbero quindi benissimo essere omessi; durante l'addestramento l'attivazione di elementi della matrice $\mathcal{W}_{21}$ (attraverso l'attivazione dei parametri spettrali corrispondenti) di fatto include nell'output l'attività di alcuni dei neuroni dell'hidden layer, che non potranno più essere ignorati, modificando così l'effettiva topologia di rete.
Si noti infine che è presente un'importante differenza fra ALLIN-GROWTH e FAST-GROWTH: il primo inizialmente include una non linearità, mentre il secondo presenta unicamente un termine lineare. Da questo segue che FAST-GROWTH si comporti inizialmente come un modello lineare, includendo termini non lineari solo se vantaggioso in fase di ottimizzazione. Questo effetto può essere intensificato includendo una *regolarizzazione esplicita* sui valori delle diagonali si $\mathcal{L}_1^{(2)}$ e $\mathcal{L}_2^{(2)}$.

Regolarizzazione Esplicita
In *machine learning* con il termine *regolarizzazione esplicita* si intende solitamente l'inserire nella *loss* un termine che penalizzi la complessità del modello: questo solitamente si concretizza in un termine additivo che penalizza un eccessivo valore della somma delle norme (quadre) dei parametri, di fatto spingendo i valori dei parametri a zero. Possiamo interpretare il processo di regolarizzazione come una cruda messa in atto del principio filosofico del *rasoio di Occam*: a parità di potenza esplicativa la soluzione migliore è quella meno complessa; e per un modello in *machine learning* molti parametri a zero sono solitamente sinonimo di semplicità [29, 31].

2.5 Risultati numerici

Questa sezione si pone i seguenti due obbiettivi:

- verifica empirica del funzionamento, e delle performance, del metodo spettrale con Φ_2, nello specifico attraverso ALLIN-CHAIN;
- verifica del funzionamento dell'inizializzazione GROWTH, assicurandosi che effettivamente l'addestramento sia in grado di attivare gli autovalori posti a zero (ovvero di far crescere l'architettura di rete nello spazio diretto).

Partendo dal primo punto sono state addestrate due reti neurali a 3 layers densamente connessi, e con connessioni *skip* dal primo all'ultimo layer, con l'obbiettivo di classificare le immagini contenute nel dataset MNIST, già descritto nel precedente capitolo. Nello specifico la dimensione del layer di input e del layer di output è fissata dal *task* di classificazione, ovvero 784 e 10 rispettivamente, mentre la dimensione dell'hidden layer è stata scelta pari a 160 per entrambe le reti. La prima rete è stata addestrata nello spazio diretto, mentre la seconda è stata parametrizzata spettralmente, con inizializzazione ALLIN-CHAIN. Per l'ottimizzazione abbiamo suddiviso la porzione di dataset di train in *batches* da 128 elementi, ed abbiamo poi portato avanti l'addestramento per numerose epoche, testando periodicamente le performances delle reti sulla porzione di test. Specifichiamo inoltre di aver utilizzato il paradigma *softmax-cross entropy loss* per l'implementazione dei classificatori, e di aver scelto l'ottimizzatore *Adam*. Al fine di poter fornire una misura dell'errore associato a questa presa dati abbiamo allenato 5 reti per ognuno dei due modelli proposti, registrando poi media e deviazione standard. I risultati di questo processo sono esposti in Figura 2.10.
Possiamo notare come la rete spettrale risulti funzionante: essa è in grado di apprendere il task di classificazione con un'accuratezza che si attesta attorno al 98%, risultato in linea con le performance di una rete neurale standard di simili dimensioni. Inoltre la nostra presa dati sembra suggerire che la rete spettrale sia caratterizzata da un migliore rendimento per allenamenti brevi (poche epoche di train), come si può vedere in Figura 2.10.
Siamo inoltre interessati a verificare se il processo di addestramento ha attivato le connessioni skip layer, ovvero se i valori delle diagonali di $\mathcal{L}_1^{(2)}$ e $\mathcal{L}_3^{(2)}$ si sono

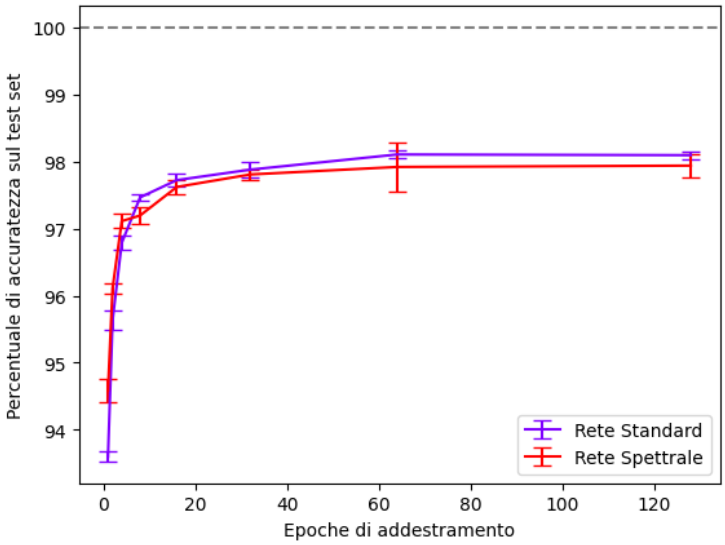

Figura 2.10. Risultati sperimentali per ALLIN-CHAIN. In viola è riportata la curva d'apprendimento per la rete neurale allenata nello spazio diretto (ovvero che ha come parametri direttamente i valori dei pesi sugli archi di connessione), mentre in rosso è riportata la curva d'apprendimento per la rete spettrale con inizializzazione ALLIN-CHAIN. Si noti come sul lungo periodo la rete spettrale abbia performance in linea con la rete neurale standard, e come inoltre sembri avere un rendimento migliore per allenamenti brevi.

allontanati da zero. Abbiamo quindi estratto i valori delle diagonali dei modelli allenati, ed abbiamo provveduto a riportare questi valori, per uno dei modelli addestrati con 100 epoche, in istogramma nella Figura 2.11. Si può verificare che i valori delle diagonali di $\mathcal{L}_1^{(2)}$ e $\mathcal{L}_3^{(2)}$ hanno avuto modo di evolvere durante la fase di train, in linea con le nostre predizioni teoriche. Da questo si conclude anche che l'addestramento spettrale è effettivamente in grado di modificare la topologia di rete, attivando le connessioni skip layer inizialmente spente.
Passando al secondo punto, ovvero la verifica numerica dell'inizializzazione GROWTH, sceglieremo nello specifico di prendere in esame l'inizializzazione FAST-GROWTH. Il contesto sperimentale in questo caso è perfettamente analogo a quello descritto sopra, e non è stato implementato alcun processo di regolarizzazione esplicita; l'unica differenza è appunto la diversa inizializzazione per la rete spettrale. I risultati sono riportati in Figura 2.12.
Possiamo notare che anche con questa inizializzazione la rete spettrale risulta funzionante, e in grado di apprendere il task di classificazione con una buona accuratezza. Nelle fasi iniziali dell'apprendimento questa volta la rete spettrale è caratterizzata da performance inferiori rispetto alla rete addestrata nello spazio diretto: questo non dovrebbe sorprendere in quanto come già discusso la rete spettrale con FAST-GROWTH si comporta inizialmente come un modello lineare di dimensioni estremamente ridotte, e solo successivamente può evolvere in una rete profonda con canali non lineari. Come è già stato menzionato siamo interessati a verificare esplicitamente che il processo di addestramento abbia spinto fuori da 0 gli autovalori contenuti in $\mathcal{L}_1^{(2)}$ e $\mathcal{L}_2^{(2)}$, sinonimo di attivazione dei canali indiretti non lineari.

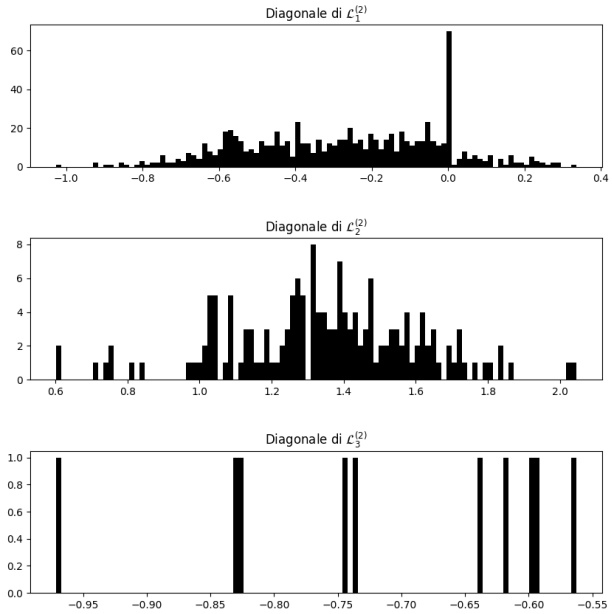

Figura 2.11. Istogramma dei valori delle diagonali di $\mathcal{L}_i^{(2)}$ per ALLIN-CHAIN, dopo l'addestramento (100 epoche). Si ricorda che inizialmente tutti i valori sulle diagonali di $\mathcal{L}_1^{(2)}$ e $\mathcal{L}_3^{(2)}$ erano fissati a zero. Si noti come i valori delle diagonali siano evoluti durante il processo di train, dimostrando il corretto funzionamento del formalismo spettrale e la capacità di attivare le connessioni skip layer.

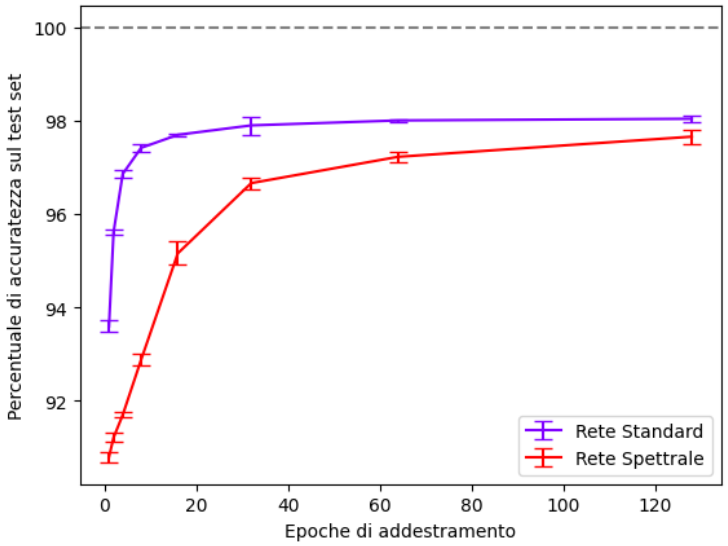

Figura 2.12. Risultati sperimentali per FAST-GROWTH. In viola è riportata la curva d'apprendimento per la rete neurale allenata nello spazio diretto, mentre in rosso è riportata la rete spettrale (con inizializzazione FAST-GROWTH).

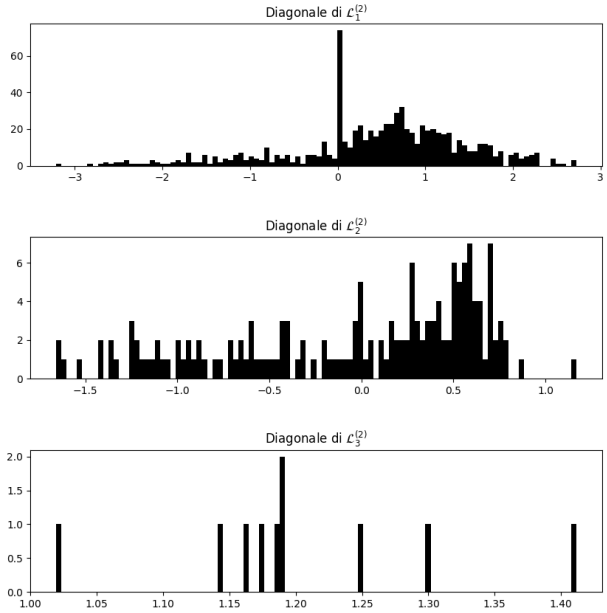

Figura 2.13. Istogramma dei valori delle diagonali di $\mathcal{L}_i^{(2)}$ per FAST-GROWTH, dopo l'addestramento (100 epoche). Si ricorda che inizialmente tutti i valori sulle diagonali di $\mathcal{L}_1^{(2)}$ e $\mathcal{L}_2^{(2)}$ erano fissati a zero. Si noti come i valori delle diagonali siano evoluti durante il processo di addestramento, aumentando quindi la profondità dell'architettura di rete e trasformando il modello da lineare a non lineare.

Abbiamo quindi, come nel caso precedente, estratto i valori delle diagonali di $\mathcal{L}_i^{(2)}$ per uno dei modelli addestrati con 100 epoche, e abbiamo riportato questi valori in istogramma, nella Figura 2.13.
Possiamo ottenere una verifica ulteriore del corretto funzionamento del formalismo spettrale ponendo in istogramma i valori delle matrici dei pesi con FAST-GROWTH: difatti i valori di $\mathcal{W}_{21}^{(2)}$ sono tutti inizialmente fissati a zero, e risulta interessante verificare direttamente la loro attivazione. Abbiamo riportato l'istogramma in questione nella figura 2.14.
Anche in questo caso i risultati della presa dati confermano le deduzioni teoriche: alcuni degli elementi di $\mathcal{W}_{21}^{(2)}$ si sono allontanati da zero durante l'addestramento, manifestando il legame che ci aspettavamo fra parametri spettrali e pesi sugli archi di connessione nello spazio diretto.

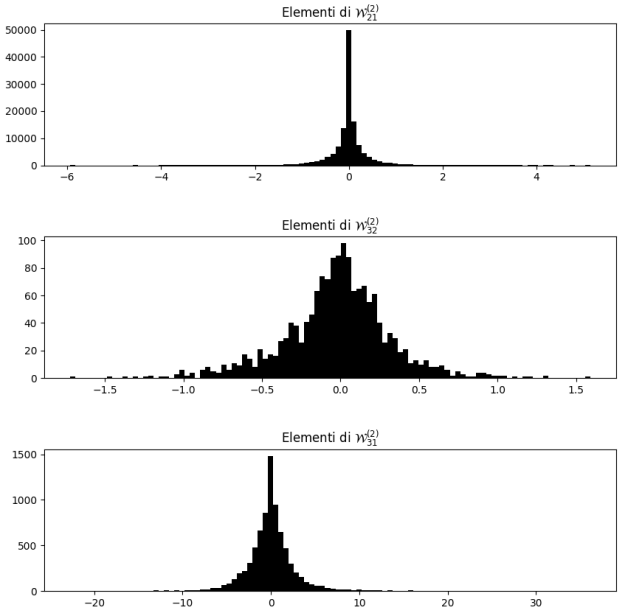

Figura 2.14. Istogramma dei valori di $\mathcal{W}_{ij}^{(2)}$ per FAST-GROWTH, dopo l'addestramento (100 epoche).

Capitolo 3

Reti Spettrali Generalizzate

3.1 Introduzione

Nel precedente capitolo abbiamo mostrato come il formalismo spettrale generalizzato a due blocchi sotto-diagonali codifichi per una matrice di adiacenza A_2 che in generale descrive connessioni fra tre strati di neuroni, con una topologia che si presta ad un utilizzo feedforward della rete. Alla luce di questo risultato ci concentriamo qui di seguito sulla classe di matrici spettrali Φ_B, con B blocchi sotto-diagonali, nella speranza che esse codifichino per una matrice di adiacenza generalizzata A_B che possa essere utilizzata per descrivere una rete completa a $B+1$ strati di neuroni.
Analizziamo quindi la situazione partendo dalla matrice spettrale Φ_3, con dimensione

$$N = N_1 + N_2 + N_3 + N_4.$$

dove, in analogia con quanto discusso in precedenza, N_i vuole essere la taglia del layer $i-esimo$ della rete. Nel seguito, per comodità tipografica, disegneremo queste matrici con blocchi sotto-diagonali quadrati, e sarà dato per scontato che i blocchi $\varphi_i^{(B)}$ potranno in generale essere rettangolari, con dimensioni $N_{i+1} \times N_i$.

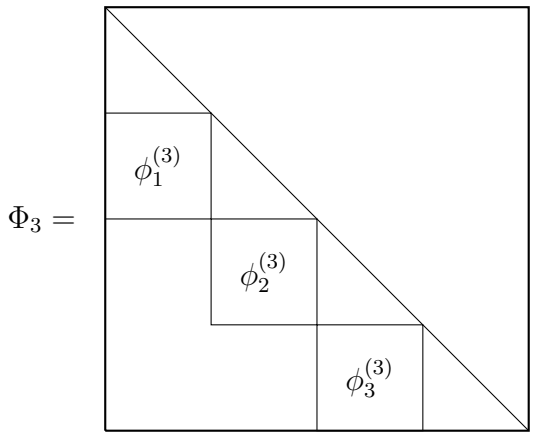

Figura 3.1. Matrice spettrale Φ_3.

3.2 Struttura dell'inversa spettrale

Come nel precedente capitolo la prima cosa di cui ci dobbiamo occupare è determinare la forma della matrice Φ_3^{-1}; iterando il metodo già utilizzato si arriva facilmente a determinare che l'inversa avrà la struttura esemplificata in figura 3.2.

$\Phi_3^{-1} =$

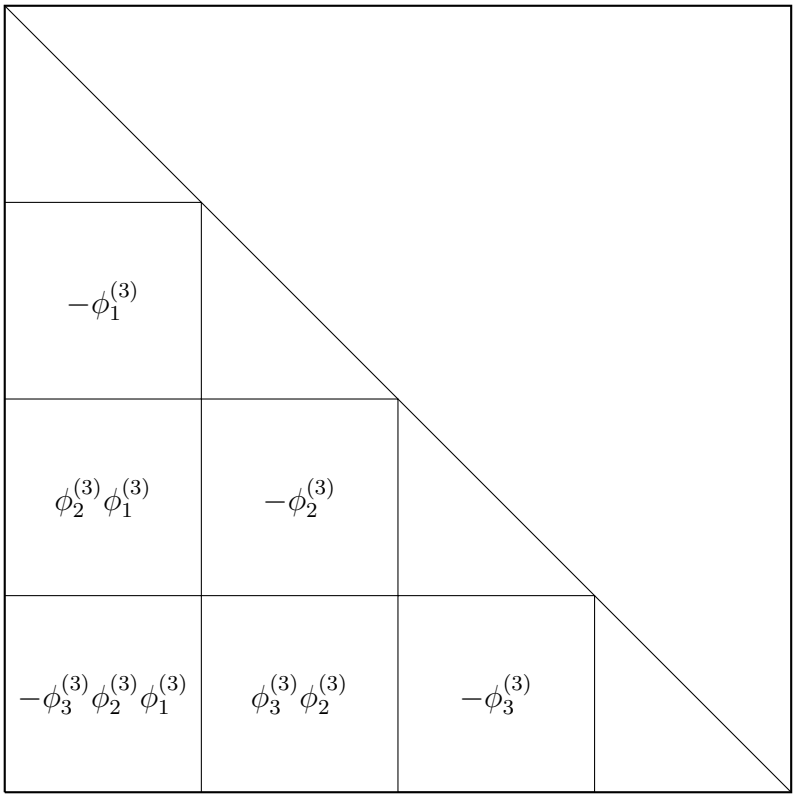

Figura 3.2. Matrice inversa Φ_3^{-1}.

Già dal caso $B = 3$ è dunque possibile notare un pattern: la struttura dei blocchi sotto-diagonali di Φ_3^{-1} è molto simile alla struttura dei blocchi di Φ_2^{-1}; confrontando con il risultato del capitolo precedente possiamo infatti notare la stessa forma dei blocchi per i primi 3 strati diagonali della matrice inversa, l'unica novità è l'ultimo blocco in basso a sinistra, che tuttavia ha una forma affine agli altri blocchi. Esattamente come nello scorso capitolo possiamo adesso considerare la forma delle potenze della matrice Φ_3, e così facendo si arriva facilmente a determinare che la matrice Φ_3^{-1} ammette la seguente espressione algebrica:

$$\Phi_3^{-1} = -\Phi_3^3 + 4\Phi_3^2 - 6\Phi + 4\mathbb{I} \tag{3.1}$$

Anche qui è subito possibile notare un pattern: la struttura analitica di Φ_B sembra consistere in una somma di $B + 1$ termini, ciascuno con la sua crescente potenza di Φ_B.
L'intuizione appena riportata si rivela corretta, difatti è possibile dimostrare quanto segue.

Teorema di inversione spettrale
Sia Φ_B una matrice spettrale a B blocchi sotto-diagonali concatenati. La sua inversa Φ_B^{-1} ha allora forma:

$$\Phi_B^{-1} = \sum_{i=0}^{B} (-1)^i \Phi_B^i \binom{B+1}{i+1} \tag{3.2}$$

Dimostrazione:
Notiamo che dimostrare la validità della (3.2) equivale a mostrare che:

$$\Phi_B \sum_{i=0}^{B} (-1)^i \Phi_B^i \binom{B+1}{i+1} = \mathbb{I}. \tag{3.3}$$

Quest'ultima equazione può essere riscritta più comodamente come segue:

$$\Phi_B \sum_{i=0}^{B} (-1)^i \Phi_B^i \binom{B+1}{i+1} - \mathbb{I} = \mathbb{O} \;\Rightarrow\; \sum_{i=0}^{B} (-1)^i \Phi_B^{i+1} \binom{B+1}{i+1} - \mathbb{I} = \mathbb{O}. \tag{3.4}$$

Il lato sinistro di quest'ultima equazione ha una struttura che ricorda quella del *binomio di Newton*:

$$(a+b)^n = \sum_{k=0}^{n} a^{n-k} b^k \binom{n}{k} \tag{3.5}$$

In particolare al caso in cui $a = -1$:

$$(b-1)^n = \sum_{k=0}^{n} (-1)^{n-k} b^k \binom{n}{k} \tag{3.6}$$

e quest'ultima può essere riscritta come:

$$(b-1)^n = \begin{cases} \sum_{k=0}^{n} (-1)^k b^k \binom{n}{k} & \text{se} \quad \text{mod } (n,2) = 0 \\ -\sum_{k=0}^{n} (-1)^k b^k \binom{n}{k} & \text{se} \quad \text{mod } (n,2) = 1 \end{cases}. \tag{3.7}$$

Nel seguito cercheremo di manipolare la forma dell'ultima espressione in (3.4) per rendere esplicita la corrispondenza ipotizzata. Con questo intento è conveniente introdurre la variabile

$$k \doteq i + 1 \tag{3.8}$$

questa permette di riscrivere l'ultima espressione in (3.4) come segue:

$$\sum_{k=1}^{B+1} (-1)^{k-1} \Phi_B^k \binom{B+1}{k} - \mathbb{I} = \mathbb{O}. \tag{3.9}$$

Ma adesso ci possiamo rendere conto che l'espressione nella sommatoria per $k = 0$ è proprio uguale a $-\mathbb{I}$, dunque possiamo assorbire l'addendo $-\mathbb{I}$ nella sommatoria, ottenendo

$$\sum_{k=0}^{B+1}(-1)^{k-1}\Phi_B^k\binom{B+1}{k}=\mathbb{O} \tag{3.10}$$

e portando infine fuori un -1:

$$-\sum_{k=0}^{B+1}(-1)^{k}\Phi_B^k\binom{B+1}{k}=\mathbb{O}\ \Rightarrow\ \sum_{k=0}^{B+1}(-1)^{k}\Phi_B^k\binom{B+1}{k}=\mathbb{O} \tag{3.11}$$

Possiamo a questo punto notare come, a meno di un eventuale segno meno, quest'ultima espressione sia equivalente a quella del binomio di Newton in (3.7), dunque la possiamo riscrivere come:

$$(\Phi_B-\mathbb{I})^{B+1}=\mathbb{O}. \tag{3.12}$$

Si noti che questo passaggio è legittimato anche dal fatto che tutte le matrici commutano con l'identità.
Con questo abbiamo dimostrato che la (3.2) è valida per ogni Φ_B se e solo se la (3.12) è valida per ogni Φ_B, cosa che adesso ci apprestiamo a dimostrare.
Possiamo adesso procedere per induzione: per $B=1$ si ha che Φ_B ha forma:

$$\Phi_1=\begin{pmatrix}\mathbb{I}_{N_1\times N_1} & \mathbb{O}_{N_1\times N_2}\\ \phi_{N_2\times N_1} & \mathbb{I}_{N_2\times N_2}\end{pmatrix}_{N\times N} \tag{3.13}$$

ma da questo possiamo immediatamente verificare che:

$$(\Phi_1-\mathbb{I})=\begin{pmatrix}\mathbb{O}_{N_1\times N_1} & \mathbb{O}_{N_1\times N_2}\\ \phi_{N_2\times N_1} & \mathbb{O}_{N_2\times N_2}\end{pmatrix}_{N\times N}\ \Rightarrow\ (\Phi_1-\mathbb{I})^2=\mathbb{O} \tag{3.14}$$

e questo conclude la dimostrazione per $B=1$. Per terminare la dimostrazione per induzione della (3.12) dobbiamo adesso mostrare che la validità di:

$$(\Phi_B-\mathbb{I})^{B+1}=\mathbb{O} \tag{3.15}$$

implica la validità di:

$$(\Phi_{B+1}-\mathbb{I})^{B+2}=\mathbb{O} \tag{3.16}$$

Per fare questo conviene indagare la struttura di $(\Phi_{B+1}-\mathbb{I})^i$: data la forma a blocchi sotto-diagonali concatenati di Φ_{B+1} possiamo scrivere:

$$\Phi_{B+1}=\begin{pmatrix}\Phi_B & \mathbb{O}\\ \alpha & \mathbb{I}\end{pmatrix} \tag{3.17}$$

dove con α vogliamo indicare la matrice rettangolare che contiene una matrice di zeri sulla sinistra e l'ultimo blocco sotto-diagonale di ϕ_{B+1} sulla destra. Ma dunque:

$$(\Phi_{B+1}-\mathbb{I})=\begin{pmatrix}\Phi_B-\mathbb{I} & \mathbb{O}\\ \alpha & \mathbb{O}\end{pmatrix} \tag{3.18}$$

arrivati a questo punto ci chiediamo che forma abbia $(\Phi_{B+1} - \mathbb{I})^2$:

$$(\Phi_{B+1} - \mathbb{I})^2 = \begin{pmatrix} (\Phi_B - \mathbb{I})^2 & \mathbb{O} \\ \alpha(\Phi_B - \mathbb{I}) & \mathbb{O} \end{pmatrix} \tag{3.19}$$

data la struttura dell'operazione di moltiplicazione matriciale si capisce dunque subito che:

$$(\Phi_{B+1} - \mathbb{I})^{B+1} = \begin{pmatrix} (\Phi_B - \mathbb{I})^{B+1} & \mathbb{O} \\ \alpha(\Phi_B - \mathbb{I})^B & \mathbb{O} \end{pmatrix} \tag{3.20}$$

ma adesso possiamo far valere la nostra ipotesi induttiva (3.15), che ci permette di scrivere:

$$(\Phi_{B+1} - \mathbb{I})^{B+1} = \begin{pmatrix} \mathbb{O} & \mathbb{O} \\ \alpha(\Phi_B - \mathbb{I})^B & \mathbb{O} \end{pmatrix} \tag{3.21}$$

e dunque procedendo con l'ultima moltiplicazione per $(\Phi_{B+1} - \mathbb{I})$:

$$(\Phi_{B+1} - \mathbb{I})^{B+2} = \begin{pmatrix} \mathbb{O} & \mathbb{O} \\ \alpha(\Phi_B - \mathbb{I})^B & \mathbb{O} \end{pmatrix} \begin{pmatrix} \Phi_B - \mathbb{I} & \mathbb{O} \\ \alpha & \mathbb{O} \end{pmatrix} = \begin{pmatrix} \mathbb{O} & \mathbb{O} \\ \alpha(\Phi_B - \mathbb{I})^{B+1} & \mathbb{O} \end{pmatrix} \tag{3.22}$$

ed applicando ancora una volta l'ipotesi induttiva (3.15):

$$(\Phi_{B+1} - \mathbb{I})^{B+2} = \mathbb{O} \tag{3.23}$$

Questo conclude la dimostrazione per induzione della (3.12), e dunque la dimostrazione del teorema.

Per il seguito risulterà conveniente indagare esplicitamente le espressioni per i blocchi diagonali e sotto-diagonali della matrice Φ_B^{-1}; chiameremo questi blocchi $S_{ij}^{(B)}$, con $i, j = 1, ..., B+1$ e $j \leq i$. Dalla Figura 3.2 si può intuire che essi avranno la seguente struttura:

$$S_{ij}^{(B)} = \begin{cases} \mathbb{O} & \text{se } \; j > i \\ \mathbb{I} & \text{se } \; j = i \quad \forall \; B \\ (-1)^{i-j} \prod_{k=1}^{i-j} \phi_{i-k}^{(B)} & \text{se } \; j < i \end{cases} \tag{3.24}$$

Nel seguito ci proponiamo di dimostrare esplicitamente questa relazione. Il modo migliore per procedere è considerare la seguente proposizione.

Proposizione: espressione per i blocchi sotto-diagonali di Φ_B^n
Sia Φ_B una matrice spettrale con B blocchi sotto-diagonali concatenati e sia $n \in \mathbb{N}^+$. Chiameremo $T_{ij}^{(B,n)}$ il blocco i, j della matrice Φ_B^n, con $i, j = 1, ..., B+1$ e $j \leq i$. Allora $T_{ij}^{(B,n)}$ avrà la seguente espressione:

$$T_{i,j}^{(B,n)} = \begin{cases} \mathbb{O} & \text{se } \; j > i \\ \mathbb{I} & \text{se } \; j = i \\ \tau(i-j,n)\prod_{k=1}^{i-j}\phi_{i-k}^{(B)} & \text{se } \; j < i \end{cases} \quad \forall\ B \tag{3.25}$$

con:

$$\tau(i-j,n) \doteq \begin{cases} 0 & \text{se } \; n < i-j \\ \binom{n}{i-j} & \text{se } \; n \geq i-j \end{cases} \tag{3.26}$$

Dimostrazione:
Procederemo per induzione: partiamo dunque dal caso $n = 1$: per definizione si ha che

$$T_{ij}^{(B,1)} = [\Phi_B]_{i,j} = \begin{cases} \mathbb{O} & \text{se } \; j > i \\ \mathbb{I} & \text{se } \; j = i \\ \phi_j^{(B)} & \text{se } \; j = i-1 \\ \mathbb{O} & \text{se } \; j < i-1 \end{cases} \quad \forall\ B \tag{3.27}$$

E possiamo immediatamente verificare che la (3.25) riproduce la (3.27): infatti l'uguaglianza è ovvia per $j > i$ e $j = 1$, e per quanto riguarda gli ultimi due casi il fattore τ permette di riprodurre correttamente quello che abbiamo in (3.27). Adesso possiamo procedere con il passo induttivo: ipotizzando che la (3.25) sia vera per n dobbiamo dimostrare che essa vale per $n+1$. Possiamo sicuramente scrivere che:

$$T_{i,j}^{(B,n+1)} = \sum_{l=1}^{B+1} T_{i,l}^{(B,n)}[\Phi_B]_{l,j} \tag{3.28}$$

ma data la struttura del blocco $[\Phi_B]_{l,j}$ esplicitata in (3.27) si capisce immediatamente che sopravviveranno unicamente 2 termini della sommatoria in (3.28):

$$T_{i,j}^{(B,n+1)} = T_{i,j}^{(B,n)}[\Phi_B]_{j,j} + T_{i,j+1}^{(B,n)}[\Phi_B]_{j+1,j} \tag{3.29}$$

Sfruttando la definizione di Φ_B possiamo scrivere:

$$T_{i,j}^{(B,n+1)} = T_{i,j}^{(B,n)}\mathbb{I} + T_{i,j+1}^{(B,n)}\phi_j^{(B)} \tag{3.30}$$

Ed adesso possiamo verificare che caso per caso si ottenga quello che vogliamo, facendo valere la nostra ipotesi induttiva (3.25):

- Per $j > i$ l'ipotesi induttiva fa annullare la somma in (3.30), e dunque otteniamo $T_{i,j}^{(B,n+1)} = \mathbb{O}$ come desiderato.

- Per $j = i$ otteniamo:

$$T_{i,j}^{(B,n+1)} = \mathbb{III} + \mathbb{O}\phi_j^{(B)} = \mathbb{I} \tag{3.31}$$

come desiderato.

- Per $j < i$ otteniamo:

$$T_{i,j}^{(B,n+1)} = \tau(i-j,n)\prod_{k=1}^{i-j}\phi_{i-k}^{(B)}\mathbb{I} + \tau(i-j-1,n)\prod_{k=1}^{i-j-1}\phi_{i-k}^{(B)}\phi_j^{(B)} \tag{3.32}$$

da questo vediamo che se $n+1 < i-j$ otteniamo $\mathbb{O}$ come desiderato, mentre se $n+1 \geq i-j$ otteniamo:

$$T_{i,j}^{(B,n+1)} = \binom{n}{i-j}\prod_{k=1}^{i-j}\phi_{i-k}^{(B)} + \binom{n}{i-j-1}\prod_{k=1}^{i-j-1}\phi_{i-k}^{(B)}\phi_j^{(B)} \tag{3.33}$$

Adesso possiamo notare che la produttoria nel secondo termine corre fino a $j+1$, e dunque moltiplicando a destra per $\phi_j^{(B)}$, che è quello che abbiamo, otteniamo esattamente la produttoria presente nel primo termine, e dunque possiamo raccogliere:

$$T_{i,j}^{(B,n+1)} = \left[\binom{n}{i-j} + \binom{n}{i-j-1}\right]\prod_{k=1}^{i-j}\phi_{i-k}^{(B)} \tag{3.34}$$

Adesso possiamo notare che il coefficiente binomiale soddisfa la celebre relazione:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{3.35}$$

ovvero, chiamando $n \to n+1$ e $k \to i-j$:

$$\binom{n+1}{i-j} = \binom{n}{i-j} + \binom{n}{i-j-1} \tag{3.36}$$

e dunque la (3.34) si riduce a:

$$T_{i,j}^{(B,n+1)} = \binom{n+1}{i-j}\prod_{k=1}^{i-j}\phi_{i-k}^{(B)} \tag{3.37}$$

che è esattamente quello che vogliamo per il caso $n+1$, e con questo dunque la dimostrazione per induzione è conclusa.

Dato il lemma appena esposto possiamo dimostrare la relazione per i blocchi di Φ_B^{-1} (3.24) semplicemente applicando la precedentemente dimostrata formula (3.2) per

l'inversa di Φ_B:

$$S_{ij}^{(B)} = \sum_{k=0}^{B} (-1)^k T_{ij}^{(B,k)} \binom{B+1}{k+1} \tag{3.38}$$

dal lemma sappiamo che tutti i termini con $i < j$ si annullano, dunque la prima parte della relazione (3.24) è già verificata. Per $i = j$ otteniamo invece:

$$S_{ii}^{(B)} = \sum_{k=0}^{B} (-1)^k \mathbb{I}^k \binom{B+1}{k+1} = \mathbb{I} \sum_{k=0}^{B} (-1)^k \binom{B+1}{k+1} \tag{3.39}$$

e per proseguire ci occorre la seguente proposizione.

Proposizione: somma di binomiali
Sia $n \in \mathbb{N}^+$, allora vale la seguente identità:

$$\sum_{k=0}^{B} (-1)^k \binom{B+1}{k+1} = 1 \quad \forall\ B \tag{3.40}$$

Dimostrazione:
Per dimostrare questa identità dobbiamo tenere a mente la seguente proprietà dei coefficienti binomiali (verrà dimostrata nel lemma successivo):

$$\sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0 \quad \forall\ n \in \mathbb{N}^+ \tag{3.41}$$

Mettiamoci adesso nel caso in cui B **sia pari**: in questo caso possiamo scrivere

$$\begin{aligned} &\sum_{k=0}^{B} (-1)^k \binom{B+1}{k+1} = \sum_{k=0}^{B-1} (-1)^k \binom{B+1}{k+1} + (-1)^B \binom{B+1}{B+1} = \\ &= \sum_{k=0}^{B-1} (-1)^k \binom{B+1}{k+1} + 1. \end{aligned} \tag{3.42}$$

Adesso possiamo sfruttare la relazione di ricorrenza (3.35) per scrivere:

$$\sum_{k=0}^{B-1} (-1)^k \binom{B+1}{k+1} + 1 = \sum_{k=0}^{B-1} (-1)^k \binom{B}{k} + \sum_{k=0}^{B-1} (-1)^k \binom{B}{k+1} + 1. \tag{3.43}$$

Ma adesso ci possiamo rendere conto che

$$1 = \binom{B}{B} \tag{3.44}$$

e dunque possiamo accorpare il primo e l'ultimo termine:

$$\begin{aligned}&\sum_{k=0}^{B-1}(-1)^k\binom{B}{k}+\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}+\binom{B}{B}=\\&=\sum_{k=0}^{B}(-1)^k\binom{B}{k}+\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}\end{aligned}\tag{3.45}$$

ma quindi per la (3.41):

$$\sum_{k=0}^{B}(-1)^k\binom{B}{k}+\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}=\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}.\tag{3.46}$$

Arrivati a questo punto possiamo effettuare il cambiamento di variabile $k'=k+1$:

$$\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}=\sum_{k'=1}^{B}(-1)^{k'-1}\binom{B}{k'}=-\sum_{k'=1}^{B}(-1)^{k'}\binom{B}{k'}\tag{3.47}$$

Adesso possiamo aggiungere e togliere 1 nel seguente modo:

$$-\sum_{k'=1}^{B}(-1)^{k'}\binom{B}{k'}=-\sum_{k'=1}^{B}(-1)^{k'}\binom{B}{k'}-\binom{B}{0}+\binom{B}{0}\tag{3.48}$$

ed infine possiamo notare che i primi due termini di quest'ultima espressione possono essere accorpati per ottenere proprio la (3.41), e dunque possono essere eliminati, e quello che rimane dunque è:

$$\binom{B}{0}=1\tag{3.49}$$

e questo conclude la dimostrazione per il caso B pari. Nel caso in cui B **sia dispari** la dimostrazione si svolge in maniera simile:

$$\begin{aligned}&\sum_{k=0}^{B}(-1)^k\binom{B+1}{k+1}=\sum_{k=0}^{B-1}(-1)^k\binom{B+1}{k+1}+(-1)^B\binom{B+1}{B+1}=\\&=\sum_{k=0}^{B-1}(-1)^k\binom{B+1}{k+1}-1.\end{aligned}\tag{3.50}$$

Adesso possiamo sfruttare la relazione di ricorrenza (3.35):

$$\sum_{k=0}^{B-1}(-1)^k\binom{B+1}{k+1}-1=\sum_{k=0}^{B-1}(-1)^k\binom{B}{k}+\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}-1.\tag{3.51}$$

Ma adesso ci possiamo rendere conto che

$$-1=-\binom{B}{B}\tag{3.52}$$

ed anche in questo caso possiamo accorpare il primo e l'ultimo termine: il segno è corretto in quanto questa volta B è dispari e dunque l'ultimo termine avrà un meno. Dunque si arriva a:

$$\begin{aligned}&\sum_{k=0}^{B-1}(-1)^k\binom{B}{k}+\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}+\binom{B}{B}=\\&=\sum_{k=0}^{B}(-1)^k\binom{B}{k}+\sum_{k=0}^{B-1}(-1)^k\binom{B}{k+1}\end{aligned} \tag{3.53}$$

che possiamo riconoscere essere la stessa espressione che avevamo trovato nel caso pari, la (3.50). Da qui dunque la dimostrazione procede esattamente nello stesso modo, e troviamo dunque anche in questo caso l'uguaglianza con 1. Questo conclude la dimostrazione di questo lemma.

Lemma: azzeramento della somma di binomiali
Sia $n \in \mathbb{N}^+$, allora vale la seguente identità:

$$\sum_{k=0}^{n}(-1)^k\binom{n}{k}=0 \quad \forall\ n \tag{3.54}$$

Dimostrazione:
Si ricordi l'espressione del binomio di Newton:

$$(x+y)^n=\sum_{k=0}^{n}\binom{n}{k}x^{n-k}y^k. \tag{3.55}$$

Notiamo che ponendo $x=1, y=-1$ si ottiene quindi:

$$(1-1)^n=\sum_{k=0}^{n}\binom{n}{k}1^{n-k}(-1)^k=\sum_{k=0}^{n}(-1)^k\binom{n}{k} \tag{3.56}$$

dove l'ultima espressione è proprio (3.54), e dove la prima espressione è nulla per ogni n; con questo la dimostrazione è conclusa.

Alla luce di quest'ultima proposizione la (3.39) implica subito che:

$$S_{ii}^{(B)}=\mathbb{I} \quad con\ i=j \quad \forall B \tag{3.57}$$

Dunque per concludere la dimostrazione della (3.24) non ci resta altro che trattare il caso $j<i$:

$$S_{ij}^{(B)}=\sum_{k=0}^{B}(-1)^k T_{ij}^{(B,k)}\binom{B+1}{k+1}=\sum_{k=0}^{B}(-1)^k\tau(i-j,k)\prod_{l=1}^{i-j}\phi_{i-l}^{(B)}\binom{B+1}{k+1} \tag{3.58}$$

data la forma di τ possiamo subito eliminare tutti i termini con $k<i-j$, ed esplicitare τ:

$$S_{ij}^{(B)} = \sum_{k=i-j}^{B} (-1)^k \binom{k}{i-j} \prod_{l=1}^{i-j} \phi_{i-l}^{(B)} \binom{B+1}{k+1}. \tag{3.59}$$

Dato che la produttoria non ha dipendenze da k possiamo estrarla:

$$S_{ij}^{(B)} = \prod_{l=1}^{i-j} \phi_{i-l}^{(B)} \sum_{k=i-j}^{B} (-1)^k \binom{k}{i-j} \binom{B+1}{k+1}. \tag{3.60}$$

Ma dunque per concludere la dimostrazione della (3.24) ci basta mostrare che:

$$\sum_{k=i-j}^{B} (-1)^k \binom{k}{i-j} \binom{B+1}{k+1} = (-1)^{i-j} \quad \forall\, B \in \mathbb{N}^+ \quad \forall\, i-j \mid 1 \leq i-j \leq B \tag{3.61}$$

Per comodità conviene porre $\rho \doteq i-j$, e riscrivere la (3.61) come:

$$\sum_{k=\rho}^{B} (-1)^k \binom{k}{\rho} \binom{B+1}{k+1} = (-1)^{\rho} \quad \forall\, B \in \mathbb{N}^+ \quad \forall\, \rho \mid 1 \leq \rho \leq B \tag{3.62}$$

Procediamo per induzione: per $B = 1$ l'unico valore possibile per ρ è $\rho = 1$, in questo caso otteniamo:

$$\sum_{k=1}^{1} (-1)^k \binom{k}{1} \binom{2}{k+1} = (-1)^1 \binom{1}{1} \binom{2}{2} = -1 = (-1)^1 \tag{3.63}$$

passando allo step induttivo ipotizziamo la validità della (3.62) per B e per ogni $\rho \leq B$, e dimostriamo che vale anche per $B+1$ $\forall\ \rho \leq B+1$. Per fare questo possiamo scrivere:

$$\begin{aligned} &\sum_{k=\rho}^{B+1} (-1)^k \binom{k}{\rho} \binom{B+2}{k+1} = \sum_{k=\rho}^{B} (-1)^k \binom{k}{\rho} \binom{B+2}{k+1} + (-1)^{B+1} \binom{B+1}{\rho} \binom{B+2}{B+2} = \\ &= \sum_{k=\rho}^{B} (-1)^k \binom{k}{\rho} \binom{B+2}{k+1} + (-1)^{B+1} \binom{B+1}{\rho} \end{aligned} \tag{3.64}$$

adesso possiamo sfruttare la relazione di ricorrenza per i coefficienti binomiali:

$$\sum_{k=\rho}^{B} (-1)^k \binom{k}{\rho} \binom{B+1}{k+1} + \sum_{k=\rho}^{B} (-1)^k \binom{k}{\rho} \binom{B+1}{k} + (-1)^{B+1} \binom{B+1}{\rho} \tag{3.65}$$

ma adesso possiamo riconoscere nella prima somma l'ipotesi induttiva (3.62), e dunque possiamo scrivere:

$$(-1)^{\rho} + \sum_{k=\rho}^{B} (-1)^k \binom{k}{\rho} \binom{B+1}{k} + (-1)^{B+1} \binom{B+1}{\rho} \tag{3.66}$$

possiamo notare che l'ultimo termine è esattamente il termine $B+1$ della sommatoria, e dunque lo possiamo accorpare ad essa:

$$(-1)^{\rho} + \sum_{k=\rho}^{B+1} (-1)^k \binom{k}{\rho} \binom{B+1}{k} \tag{3.67}$$

Dunque per concludere la nostra dimostrazione dobbiamo provare la seguente proposizione.

Proposizione: azzeramento di somma di prodotti di binomiali
Siano $n, \rho \in \mathbb{N}$. Allora vale la seguente identità:

$$\sum_{k=\rho}^{n} (-1)^k \binom{k}{\rho} \binom{n}{k} = 0 \quad \forall n, \rho \in \mathbb{N} \tag{3.68}$$

Dimostrazione:
Innanzitutto possiamo sfruttare la definizione di coefficiente binomiale

$$\binom{k}{\rho} = \frac{k!}{\rho!(k-\rho)!} \tag{3.69}$$

nella seguente maniera:

$$\sum_{k=\rho}^{n} (-1)^k \binom{k}{\rho} \binom{n}{k} = \sum_{k=m}^{n} (-1)^k \frac{k!}{(k-\rho)!\rho!} \frac{n!}{k!(n-k)!} = \tag{3.70}$$

possiamo notare che $k!$ si elide, e moltiplicando sopra e sotto per $(n-\rho)!$ otteniamo:

$$\begin{aligned} &= \sum_{k=\rho}^{n} (-1)^k \frac{n!}{(n-\rho)!\rho!} \frac{(n-\rho)!}{(n-k)!(k-\rho)!} = \\ &= \sum_{k=\rho}^{n} (-1)^k \binom{n}{\rho} \binom{n-\rho}{k-\rho} = \end{aligned} \tag{3.71}$$

Questo è molto utile in quanto ci permette di estrarre un fattore fuori dalla sommatoria:

$$= \binom{n}{\rho} \sum_{k=\rho}^{n} (-1)^k \binom{n-\rho}{k-\rho}. \tag{3.72}$$

Adesso poniamo $\alpha = k - \rho$ e riscriviamo la sommatoria (3.72):

$$\binom{n}{\rho} \sum_{\alpha=0}^{n-\rho} (-1)^{\alpha+\rho} \binom{n-\rho}{\alpha} = (-1)^{\rho} \binom{n}{\rho} \sum_{\alpha=0}^{n-\rho} \binom{n-\rho}{\alpha} (-1)^{\alpha} \tag{3.73}$$

ma quest'ultima sommatoria è nulla per il lemma dimostrato in precedenza, e dunque la (3.68) è dimostrata.

Con questo lemma la dimostrazione di (3.62) è conclusa, e dunque abbiamo anche finalmente dimostrato la relazione (3.24) per Φ_B^{-1}.

3.3 Struttura della matrice di adiacenza

Trovata la struttura generale dell'inversa di Φ_B ci possiamo concentrare sulla struttura generale di A_B; analiticamente la questione è di facile gestione, in quanto ovviamente la formula per A_B sarà sempre:

$$A_B = \Phi_B \Lambda_B \Phi_B^{-1} \tag{3.74}$$

con Λ_B matrice diagonale degli autovalori associata alla matrice spettrale Φ_B. Dato che abbiamo ricavato la formula generale per l'inversa la formula generale per la matrice di adiacenza sarà semplicemente:

$$A_B = \Phi_B \Lambda_B \sum_{i=0}^{B} (-1)^B \Phi_B^i \binom{B+1}{i+1}. \tag{3.75}$$

Tuttavia questa espressione generale per A_B non ci soddisfa appieno, in quanto quello a cui siamo davvero interessati è la forma dei singoli blocchi di connessione di A_B, ed una scrittura *globale* della matrice di adiacenza oscura in parte questa informazione.
Analizziamo quindi più nel dettaglio la struttura di A_B. Per iniziare possiamo ricavare la struttura di A_3

$$A_3 = \Phi_3 \Lambda_3 \Phi_3^{-1} \tag{3.76}$$

con ovviamente Λ_3 matrice diagonale degli autovalori associati alla matrice Φ_3; e data la struttura dei prodotti matriciali utilizzati per ricavare A_3 è conveniente pensare a Λ_3 come matrice a 4 blocchi diagonali quadrati, come mostrato in Figura 3.3.

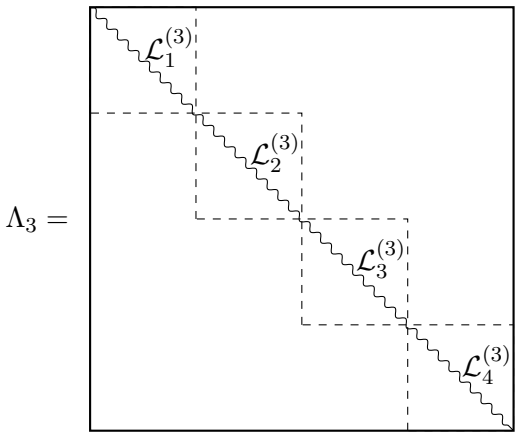

Figura 3.3. Struttura a blocchi di Λ_3.

Svolgendo i prodotti in (3.76) si arriva ad una matrice di adiacenza con struttura esposta in Figura 3.4.

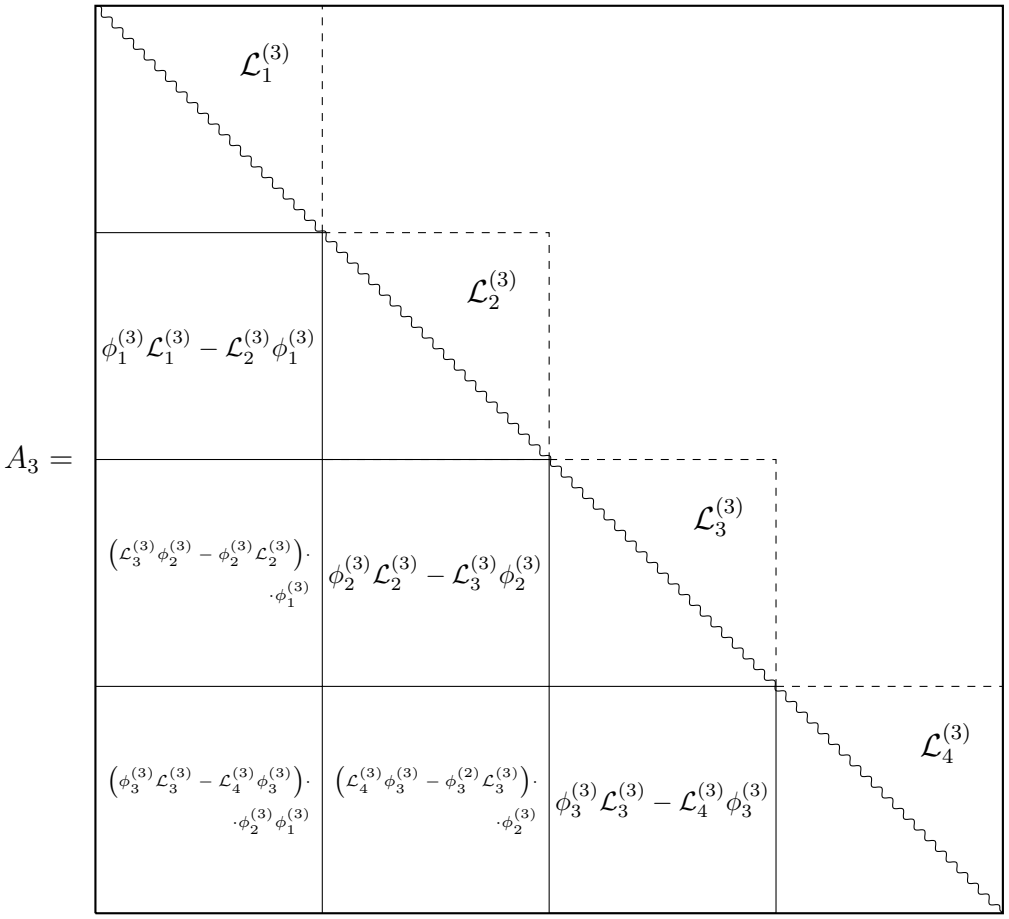

Figura 3.4. Struttura della matrice di adiacenza A_3.

In generale per riferirci alla struttura di A è comodo utilizzare la nomenclatura per i blocchi di connessione introdotta nel precedente capitolo, che in questo caso viene estesa a quanto riportato in Figura 3.5.

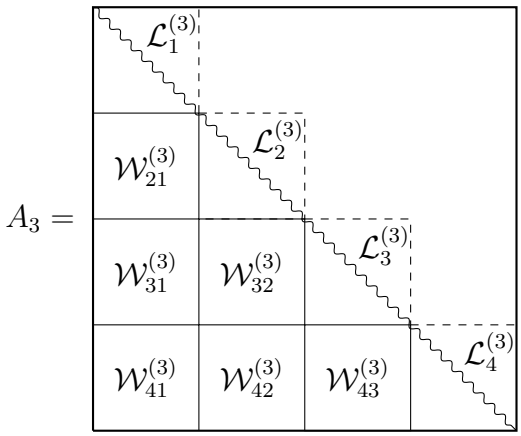

Figura 3.5. Struttura a blocchi di A_3.

Possiamo anche qui notare un pattern confrontando con i risultati del precedente capitolo: i blocchi della matrice di adiacenza sembrano forniti da una differenza di prodotti con la stessa struttura, e moltiplicati poi per una sequenza di blocchi spettrali.
Ancora una volta l'intuizione si rivela corretta, in quanto è possibile dimostrare quanto segue.

Teorema dei blocchi di connessione
Sia Φ_B una matrice spettrale con B blocchi sotto-diagonali concatenati

$$\phi_i^{(B)} \ , \ i = 1, ..., B,$$

e sia Λ_B la matrice degli autovalori associata, con $B + 1$ blocchi diagonali quadrati

$$\mathcal{L}_i^{(B)} \ , \ i = 1, ..., B + 1.$$

Allora la matrice di adiacenza A_B associata a Φ_B, Λ_B dalla formula:

$$A_B = \Phi_B \Lambda_B \Phi_B^{-1}$$

Avrà un numero di blocchi sotto-diagonali pari al B-esimo *numero triangolare*:

$$T_B = \frac{B(B+1)}{2} \tag{3.77}$$

enumerabili come

$$W_{ij}^{(B)} \ , \ i = 1, ..., B + 1 \ , \ j < i$$

e forniti dalla seguente formula:

$$\mathcal{W}_{ij}^{(B)} = (-1)^{i-1-j}\left[\phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)} - \mathcal{L}_{i}^{(B)}\phi_{i-1}^{(B)}\right]\prod_{k=1}^{i-1-j}\phi_{i-1-k}^{(B)} \tag{3.78}$$

inoltre i blocchi diagonali di A_B saranno esattamente gli stessi di Λ_B, mentre la parte superiore della matrice sarà nulla.
Dimostrazione
La matrice A_B è costruita come $A_B = \Phi_B\Lambda_B\Phi_B^{-1}$, dove Λ_B è una matrice diagonale, che pensiamo come una matrice a blocchi diagonale (le matrici dei blocchi saranno matrici diagonali), e dunque i blocchi di A rispettano:

$$A_{ij} = \sum_{k=1}^{B+1}\Phi_{ik}\Lambda_k\Phi_{kj}^{-1} \tag{3.79}$$

Tuttavia sappiamo anche che la forma dei blocchi della matrice spettrale Φ_B è:

$$\Phi_{ij} = [\Phi_B]_{i,j} = \begin{cases}\mathbb{O} & \text{se } \; j > i \\ \mathbb{I} & \text{se } \; j = i \\ \phi_j^{(B)} & \text{se } \; j = i-1 \\ \mathbb{O} & \text{se } \; j < i-1\end{cases} \qquad \forall\ B \tag{3.80}$$

da questo si deduce immediatamente che solo due termini della sommatoria sopravviveranno, e dunque la (3.79) si riduce a:

$$A_{ij} = \Phi_{i,i}\Lambda_i\Phi_{i,j}^{-1} + \Phi_{i,i-1}\Lambda_{i-1}\Phi_{i-1,j}^{-1} \tag{3.81}$$

e mettendo insieme la (3.80) con il fatto che Λ_B ha forma a blocchi:

$$[\Lambda_B]_{ij} = \begin{cases}\mathcal{L}_i^{(B)} & \text{se } \; j = i \\ \mathbb{O} & \text{se } \; j \neq i\end{cases} \tag{3.82}$$

possiamo scrivere la (3.81) come:

$$A_{ij} = \mathbb{I}\mathcal{L}_i^{(B)}\Phi_{i,j}^{-1} + \phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)}\Phi_{i-1,j}^{-1} \tag{3.83}$$

adesso dobbiamo ricordare che nella scorsa sezione abbiamo dimostrato la (3.24), ovvero che la matrice Φ_B^{-1} ha forma a blocchi:

$$[\Phi_B^{-1}]_{ij} = \begin{cases}\mathbb{O} & \text{se } \; j > i \\ \mathbb{I} & \text{se } \; j = i \\ (-1)^{i-j}\prod_{k=1}^{i-j}\phi_{i-k}^{(B)} & \text{se } \; j < i\end{cases} \qquad \forall\ B \tag{3.84}$$

E dunque mettendo insieme la (3.83) con la (3.84) si vede subito che per $j > i$:

$$A_{ij} = \mathbb{I}\mathcal{L}_i^{(B)}\mathbb{O} + \phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)}\mathbb{O} = \mathbb{O} \tag{3.85}$$

esattamente come previsto; mentre per $j = i$:

$$A_{ii} = \mathbb{I}\mathcal{L}_i^{(B)}\mathbb{I} + \phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)}\mathbb{O} = \mathcal{L}_i^{(B)}. \tag{3.86}$$

Non ci resta altro che dimostrare che per $j < i$ la forma dei blocchi è proprio la (3.78). Mettendo insieme (3.80) con (3.83) e (3.84) possiamo scrivere, per $j < i$:

$$\begin{aligned}\mathcal{W}_{ij}^{(B)} = A_{ij} = \mathcal{L}_i^{(B)}(-1)^{i-j}\prod_{k=1}^{i-j}\phi_{i-k}^{(B)} + \phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)}(-1)^{i-1-j}\prod_{k=1}^{i-1-j}\phi_{i-1-k}^{(B)} = \\ = -\mathcal{L}_i^{(B)}\phi_{i-1}^{(B)}(-1)^{i-j-1}\prod_{k=2}^{i-j}\phi_{i-k}^{(B)} + \phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)}(-1)^{i-1-j}\prod_{k=1}^{i-1-j}\phi_{i-1-k}^{(B)}\end{aligned} \tag{3.87}$$

adesso nella prima produttoria possiamo effettuare un cambiamento di variabile: $k \to k-1$, quello che si ottiene è

$$\mathcal{W}_{ij}^{(B)} = -\mathcal{L}_i^{(B)}\phi_{i-1}^{(B)}(-1)^{i-j-1}\prod_{k=1}^{i-j-1}\phi_{i-1-k}^{(B)} + \phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)}(-1)^{i-1-j}\prod_{k=1}^{i-1-j}\phi_{i-1-k}^{(B)} \tag{3.88}$$

e finalmente possiamo raccogliere, ottenendo:

$$\mathcal{W}_{ij}^{(B)} = (-1)^{i-1-j}\left[\phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)} - \mathcal{L}_i^{(B)}\phi_{i-1}^{(B)}\right]\prod_{k=1}^{i-1-j}\phi_{i-1-k}^{(B)} \tag{3.89}$$

come volevasi dimostrare.

3.4 Interpretazione della matrice di adiacenza

Alla luce di quanto dimostrato nella scorsa sezione possiamo affermare che la matrice di adiacenza generica $A_B \doteq \Phi_B\Lambda_B\Phi_B^{-1}$ effettivamente descrive le connessioni fra $B+1$ strati di neuroni, in particolare la struttura delle connessioni è comunque riconducibile alla tipica struttura di una rete neurale feedforward (ma non *strettamente* feedforward). È tuttavia necessario scendere più nel dettaglio della struttura di A_B, così come calcolata in precedenza, per chiarire meglio la topologia associata a questa classe di reti spettrali generalizzate, e per comprendere come la propagazione delle attivazioni ne sia influenzata.

Muovendo dalle osservazioni del precedente capitolo, e tenendo presenti i risultati del teorema appena esposto, risulta facile comprendere che ciascuno dei blocchi sotto-diagonali $W_{ij}^{(B)}$ di A_B descrive le connessioni pesate dal j-esimo strato neurale all'i-esimo, di contro i blocchi quadrati $\mathcal{L}_i^{(B)}$, sulla diagonale di A_B, descrivono le connessioni dell'i-esimo strato con se stesso.

Risulta inoltre opportuno mettere l'accento sul fatto che, per costruzione, A_B sarà una matrice $N \times N$ (esattamente come Φ_B e Λ_B), con N numero di neuroni della rete. Chiameremo N_i il numero di neuroni dell'i-esimo layer, e dunque:

$$N = \sum_{i=1}^{B+1} N_i. \tag{3.90}$$

Osserviamo inoltre che le matrici $\mathcal{L}_i^{(B)}$ saranno sempre matrici quadrate $N_i \times N_i$, e che i blocchi $W_{ij}^{(B)}$ saranno $N_i \times N_j$. La parte sovrastante la diagonale superiore della matrice A_B non sarà invece mai popolata, e questo implica l'assenza di connessioni *all'indietro* nella rete.
Riassumendo possiamo affermare che per rappresentare una rete neurale feedforward, con connessioni ricorrenti e connessioni skip layer, avente N neuroni organizzati in L layers, ciascuno con N_i neuroni $(0 < i < L + 1)$, dobbiamo partire da una matrice spettrale Φ_B, $N \times N$, con un numero di blocchi sotto-diagonali ϕ pari a $L - 1$; la dimensione del blocco sotto-diagonale i-esimo dovrà essere $N_{i+1} \times N_i$. La matrice di adiacenza A_B viene poi costruita come $A \doteq \Phi_B \Lambda_B \Phi_B^{-1}$, con Λ_B matrice diagonale degli autovalori, che pensiamo come composta da L blocchi sulla diagonale, di dimensione $N_i \times N_i$ (questi blocchi sono proprio le matrici diagonali $\mathcal{L}_i$). Gli elementi dei blocchi $\phi_i^{(B)}$ e gli elementi diagonali di $\mathcal{L}_i^{(B)}$ rappresentano i parametri della rete neurale in questo formalismo. L'ultima cosa che ci resta da comprendere, in analogia a quanto fatto nel precedente capitolo, è come utilizzare le informazioni sulla topologia della rete contenute in A_B per evolvere il vettore delle attivazioni $\boldsymbol{a} \in \mathbb{R}^N$, dai cui ultimi N_{B+1} elementi estrarremo l'output.
Per chiarezza in Figura 3.6 è mostrato un esempio di possibile rete neurale ottenibile attraverso questo formalismo spettrale generalizzato.
A questo punto siamo indirizzati verso due possibili interpretazioni (ALLIN e FAST): possiamo appunto considerare il vettore delle attivazioni globale $\boldsymbol{a} \in \mathbb{R}^N$, che inizialmente presenterà attivazioni non nulle unicamente nei primi N_1 elementi, e analizzare l'elaborazione svolta dalla rete interpretando la matrice di adiacenza A_B come una matrice di evoluzione temporale in tempo discreto (approccio ALLIN)

$$\boldsymbol{a}(t+1) = A\boldsymbol{a}(t) \ , \ t \in \mathbb{N}^+. \tag{3.91}$$

Come sappiamo per il caso $B > 1$ una singola evoluzione temporale non basta per far fluire l'informazione attraverso tutti i possibili canali di connessione della rete: in generale per far fluire l'informazione contenuta nelle attivazioni del primo strato neurale *attraverso tutti i possibili canali di connessione feedforward descritti da A* saranno necessarie B evoluzioni temporali. L'output della rete potrà poi essere letto negli ultimi N_{B+1} elementi del vettore delle attivazioni.
Scendendo maggiormente nel dettaglio nel caso dell'approccio ALLIN la legge per l'aggiornamento del vettore delle attivazioni può esser scritta con la seguente relazione ricorsiva.

$$\boldsymbol{a}_i(t+1) = \sum_{k \leq i} \mathcal{A}_{ik}^{(B)} \boldsymbol{a}_k(t) \quad i = 1, ..., B+1 \tag{3.92}$$

dove con $\boldsymbol{a}_i$ si intende l'i-esima porzione del vettore delle attivazioni a (ovvero le attivazioni dell'i-esimo strato neurale), e dove con $\mathcal{A}_{ij}^{(B)}$ si intende il blocco i, j della matrice A:

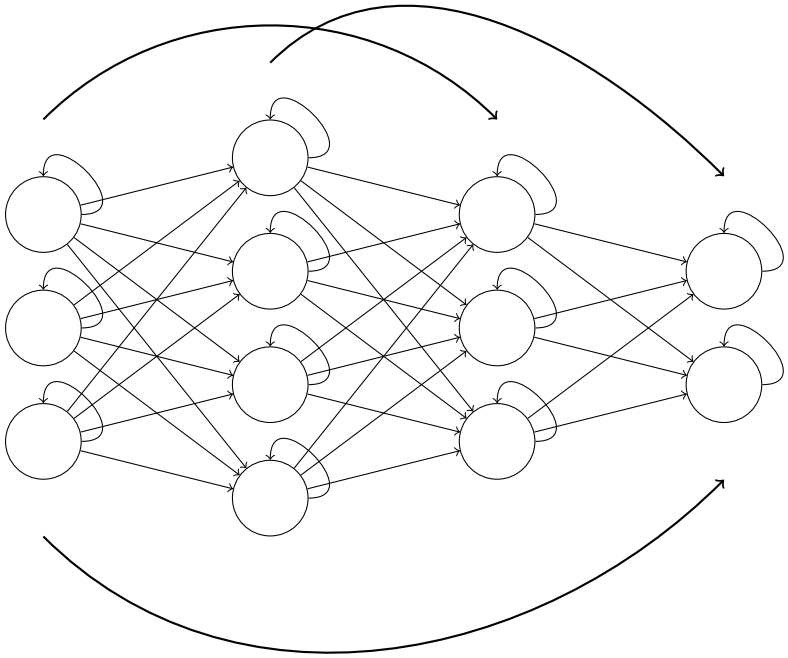

Figura 3.6. Esempio di rete neurale feedforward ottenibile da una matrice spettrale generalizzata. La matrice spettrale generatrice di questa particolare rete sarebbe una matrice Φ, 12×12, con 3 blocchi sotto-diagonali concatenati, e la matrice di adiacenza $A_{12\times 12}$ associata avrebbe 6 blocchi sotto-diagonali: 3 per le connessioni feedforward fra layer adiacenti e 3 per le connessioni skip layer; le autoconnessioni sarebbero codificate dalle matrici $\mathcal{L}_i$ sulla diagonale di A. Si vedano le figure 3.3, 3.4 e 3.5.

$$\mathcal{A}_{ij}^{(B)} = \begin{cases} \mathcal{W}_{ij}^{(B)} & \text{se } \; i > j \\ \mathcal{L}_i^{(B)} & \text{se } \; i = j \end{cases} \tag{3.93}$$

e si ricorda che la parametrizzazione spettrale delle matrici $\mathcal{W}_{ij}^{(B)}$ è data dalla (3.78). Si noti che la formula (3.92) appena riportata non include la presenza della funzione di attivazione, che tuttavia può essere facilmente inserita come segue:

$$\boldsymbol{a}_i(t+1) = \sigma\left(\sum_{k\leq i} \mathcal{A}_{ik}^{(B)} \boldsymbol{a}_k(t)\right) \quad i = 1, ..., B+1. \tag{3.94}$$

Tuttavia si ricordi che è bene rimuovere la non linearità σ nel calcolo delle attivazioni dell'ultimo layer $\boldsymbol{a}_{B+1}$ nel caso in cui la rete spettrale non sia un blocco di un'architettura più grande. Notiamo inoltre che, come già menzionato nel precedente capitolo, è possibile volendo inserire il bias neurale attraverso l'aggiunta di opportuni neuroni di bias nell'architettura spettrale.
In alternativa possiamo estrarre i blocchi di connessione feedforward dalla matrice di adiacenza ed usarli direttamente (approccio FAST, SFAST). Sotto questo approccio

la propagazione delle attivazioni avviene con un solo *forward pass*: e nello specifico le attivazioni $\boldsymbol{a}_i$ del layer neurale i-esimo sono date da:

$$\boldsymbol{a}_i = \sigma\left(\sum_{k<i} \mathcal{W}_{ik}^{(B)} \boldsymbol{a}_k\right), \quad \boldsymbol{a}_1 = \boldsymbol{x}, \quad \boldsymbol{a}_{B+1} = \boldsymbol{y} \tag{3.95}$$

dove è sempre bene ricordare che spesso nell'ultimo layer è conveniente scegliere di sopprimere la non linearità. Questa interpretazione della matrice di adiacenza di fatto ignora le connessioni ricorrenti che emergono dal formalismo spettrale.
Quanto appena riportato è valido in assenza di vincoli sui parametri spettrali, ma è immediato da quanto esposto nel capitolo precedente che sia possibile modificare l'architettura di rete, semplificandola a piacere, attraverso l'imposizione di opportuni *bounds* sulle matrici spettrali Φ_B e Λ_B. Per un esempio concreto si veda la Figura 3.7.

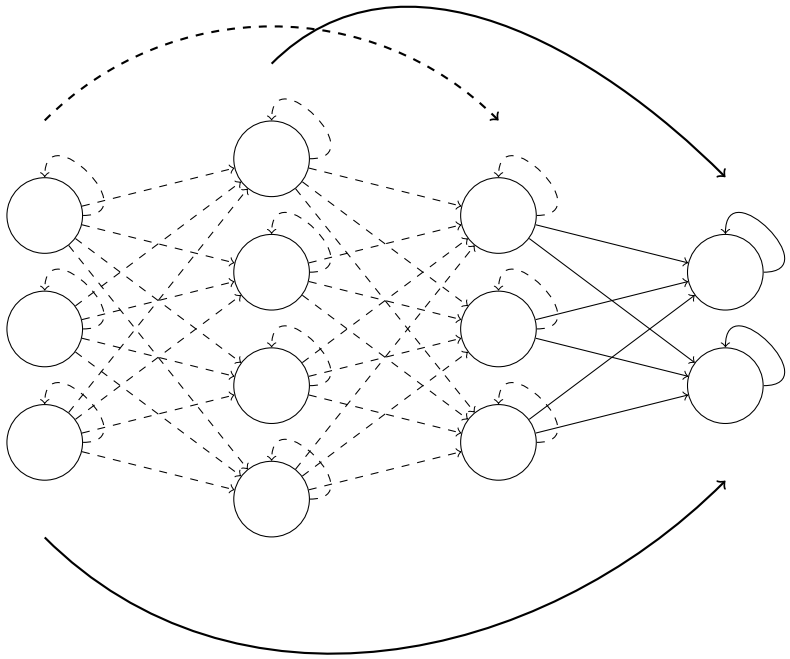

Figura 3.7. Esempio di architettura neurale ottenibile vincolando $\mathcal{L}_1^{(3)} = \mathcal{L}_2^{(3)} = \mathcal{L}_3^{(3)} = \mathbb{O}$ nella matrice degli autovalori Λ_3. Le connessioni tratteggiate sarebbero presenti in assenza di vincoli, come mostrato in Figura 3.6, ma dati i vincoli imposti esse sono assenti. Si noti come $\mathcal{L}_4^{(3)} \neq 0$ mantenga attive tutte le connessioni feedforward dirette all'ultimo layer neurale; nel caso in cui il vincolo sia posto come un'inizializzazione risulta ragionevole ipotizzare che l'attività di queste connessioni dirette all'output faciliti la formazione di connessioni profonde.

La rete in figura 3.6 muta nella struttura esposta in Figura 3.7 se si impone che i blocchi diagonali di Λ_3 siano nulli eccetto che per il blocco $\mathcal{L}_4^{(3)}$; la rete vincolata è

essenzialmente una rete a soli due strati, dato che solo la connessione skip layer fra il primo ed il quarto strato permette il passaggio di informazione.
Data la (3.78) è facile capire che in generale, per una rete con $B+1$ layer neurali, l'architettura effettiva minimale sarà ottenibile semplicemente vincolando tutti i blocchi diagonali di Λ_B a zero eccetto che per il blocco $\mathcal{L}^{(B)}_{B+1}$.
Notiamo che, come visto anche nel precedente capitolo, questi vincoli possono essere trattati come *inizializzazioni*, lasciando quindi al processo di training la libertà di modificare l'architettura neurale effettiva. Nello specifico inizializzando come mostrato in Figura 3.7 si sceglie l'architettura di partenza più minimale possibile. Il processo spettrale appena descritto si presta ad essere utilizzato per la ricerca di architetture neurali ottimali, fornendo un alternativa al già menzionato processo di *spectral pruning* presente in letteratura. Notiamo inoltre che il presente metodo permette di gestire architetture più ricche rispetto ai normali metodi spettrali, in quanto il formalismo spettrale generalizzato descrive reti che potenzialmente presentano non solo connessioni feedforward, ma anche connessioni skip layer e connessioni ricorrenti.
Con questo sono state gettate le basi per la generalizzazione del metodo spettrale a reti complesse con un numero arbitrario di layers. Ulteriori approfondimenti numerici sul tema vengono rimandati a future analisi dedicate.

Capitolo 4

Conclusioni

Giunti alla fine di questo lavoro di tesi appare doveroso spendere qualche paragrafo per fare il punto dei principali risultati ottenuti nel corso di questa trattazione, e per suggerire possibili sviluppi futuri.

4.1 I principali risultati

Prima di questo lavoro di tesi il formalismo spettrale era limitato a quanto esposto nel primo capitolo: le uniche matrici di adiacenza trattate erano quelle associate a network neurali con soli due layers. In letteratura si parlava lo stesso di reti neurali spettrali profonde, ma la descrizione di queste ultime era sempre ottenuta scindendo la rete globale in coppie di layers consecutivi, parametrizzando ogni trasformazione in termini spettrali indipendenti.
Il primo importante risultato di questa tesi consiste nell'aver mostrato che, con minime modifiche, il formalismo spettrale è anche adatto a descrivere matrici di adiacenza di reti con più layers, in sostanza è dunque possibile operare una descrizione spettrale unificata di una rete neurale complessa, in termini di una sola matrice di trasferimento che tenga simultaneamente conto di tutti i layers attivi.
Il secondo risultato rilevante ottenuto in questa tesi è strettamente legato al primo, e consiste nell'aver mostrato che il formalismo spettrale, quando generalizzato ad un numero di layers maggiore di 2, descriva naturalmente reti con architettura feedforward, equipaggiate con connessioni ricorrenti (non ignorabili, diversamente dal caso a 2 layers), e connessioni skip layer. Nello specifico all'inizio del secondo capitolo abbiamo esplorato la possibilità di generalizzare il formalismo spettrale con un'idea semplicissima: se nel formalismo spettrale standard un singolo blocco sotto-diagonale nella matrice degli autovettori Φ codifica per un bundle di connessioni feedforward fra due strati di neuroni si può forse ipotizzare che 2 blocchi sotto-diagonali codifichino per 2 bundles di connessioni feedforward fra 3 strati di neuroni. Questa intuizione si è rivelata solo parzialmente corretta: effettivamente una matrice Φ con 2 blocchi non nulli sotto la diagonale principale (Φ_2) produce una matrice di adiacenza A (A_2) che può essere interpretata come associata ad una rete con tre layers neurali. Tuttavia la matrice A_2 non contiene unicamente connessioni feedforward fra strati neurali adiacenti, come inizialmente atteso, bensì essa include anche un bundle di connessioni skip layer, che collegano direttamente il primo e

l'ultimo layer della rete, e 3 gruppi di connessioni ricorrenti, uno per ognuno dei layers neurali. Questa scoperta indica che il formalismo spettrale, quando generalizzato, è adatto a descrivere architetture anche più complesse di una semplice rete neurale feedforward, architetture come *ResNets* o *RNNs*.
Sempre nel capitolo due, trattando reti spettrali generalizzate a tre layers, abbiamo scoperto che vincolare alcuni parametri spettrali comporta una modifica dell'architettura effettiva della rete. Tramite opportuni vincoli sugli autovalori, ad esempio, è possibile trasformare l'architettura di rete codificata spettralmente in un'architettura priva di connessioni skip layer; tramite altri bounds invece è possibile ridurre la topologia neurale effettiva da 3 a 2 layers. Quello che appare ancor più interessante è la possibilità di trattare questi bounds come inizializzazioni: possiamo quindi inizializzare la rete con una certa architettura e lasciare poi che il processo di training, che agisce sui parametri spettrali, determini non solo il valore dei pesi sugli archi di connessione (come accade normalmente nel train di una rete neurale) ma anche l'architettura effettiva di rete. Questa potenzialità del formalismo spettrale era ignota prima della sua generalizzazione in questa tesi, e rappresenta un possibile vantaggio rispetto all'allenamento standard di reti neurali. In letteratura sono presenti tecniche per l'ottimizzazione della topologia di rete, ma nessuna di queste ha completamente preso piede, e spesso l'architettura neurale viene semplicemente fissata a priori, prima della fase di addestramento, e non è quindi un prodotto della procedura di ottimizzazione applicata ai dati a disposizione. Inoltre le tecniche per la ricerca d'architettura presenti in letteratura, come NEAT o DARTS [43, 20], sono spesso frutto di paradigmi ad hoc, e non emergono naturalmente dall'ottimizzazione dei parametri della rete neurale come nel caso del formalismo spettrale generalizzato. Riguardo a quest'ultimo tema apriamo una piccola parentesi per notare una coincidenza interessante: mentre NEAT (*NeuroEvolution of Augmenting Topologies*) è un algoritmo di ricerca d'architettura che si basa su un paradigma evolutivo, DARTS (*Differentiable Architecture Search*) cerca di determinare la migliore topologia di rete tramite un'ottimizzazione iterativa basata su metodi di discesa del gradiente; a grandi linee l'idea alla base di DARTS è rilassare la descrizione della topologia di rete: ad esempio invece di pensare alla presenza di un neurone, o alla presenza di un link, come una variabile binaria $0, 1$, presente o assente, DARTS assegna a queste proprietà delle variabili ad hoc continue, che lascia poi libere per l'ottimizzazione sotto il processo di learning. Questo è più o meno quello che accade anche nel formalismo spettrale generalizzato, ma una delle cruciali differenze è che le variabili d'architettura sono naturalmente presenti fin dall'inizio, e non devono quindi essere *"aggiunte a mano"* [43, 20].
L'ultimo prodotto di questo lavoro è riportato nel terzo capitolo: il formalismo spettrale viene completamente generalizzato, consentendo la descrizione di reti con un numero arbitrario di layers, e connessioni complesse, tramite una singola codifica spettrale, in termini quindi di un'unica matrice di adiacenza. In particolare sono presenti due importanti risultati teorici. Il primo consiste nell'aver formalmente mostrato che effettivamente una matrice degli autovettori Φ_B, con B blocchi sottodiagonali concatenati, codifica per una matrice di adiacenza che descrive una rete con $B + 1$ layers, e con una ricca topologia di connessione (connessioni feedforward standard, connessioni ricorrenti sui neuroni, connessioni skip layers). Il secondo è aver dimostrato in generale la forma della parametrizzazione spettrale per le connessioni

della rete. Nello specifico abbiamo dimostrato che le connessioni ricorrenti sui neuroni dell'i-esimo layer sono parametrizzate dalla matrice quadrata diagonale $\mathcal{L}_i^{(B)}$, l'i-esima matrice diagonale della matrice degli autovalori Λ_B; e abbiamo inoltre dimostrato che le connessioni feedforward dal layer j-esimo al layer i-esimo sono parametrizzate da una matrice rettangolare $\mathcal{W}_{ij}^{(B)}$:

$$\mathcal{W}_{ij}^{(B)} = (-1)^{i-1-j}\left[\phi_{i-1}^{(B)}\mathcal{L}_{i-1}^{(B)} - \mathcal{L}_i^{(B)}\phi_{i-1}^{(B)}\right]\prod_{k=1}^{i-1-j}\phi_{i-1-k}^{(B)}$$

con $\phi_i^{(B)}$ i-esimo blocco sotto-diagonale della matrice degli autovettori Φ_B (contando dall'alto). Aver trovato la forma esplicita di queste parametrizzazioni non è solo potenzialmente fruttuoso da un punto di vista teorico, ma permette in molti casi un'implementazione più semplice e efficiente del formalismo al calcolatore; se non avessimo trovato quest'espressione generale saremmo stati costretti ad implementare il formalismo spettrale generalizzato operando il calcolo diretto della matrice di adiacenza, secondo la formula:

$$A_B = \Phi_B \Lambda_B \Phi_B^{-1}.$$

Questo approccio, comunque percorribile, è alquanto dispendioso dal punto di vista dell'implementazione numerica: trovare l'inversa di grandi matrici può risultare costoso in termini di risorse computazionali. Le matrici con le quali abbiamo a che fare contengono inoltre grandi porzioni nulle, e dunque lavorando con il prodotto fra matrici occupiamo spazio di memoria semplicemente per salvare degli zeri.
Notiamo che alla fine del terzo capitolo abbiamo anche avuto modo di mostrare come il paradigma spettrale per la ricerca d'architettura esposto nel secondo capitolo per reti a tre layers si estenda naturalmente anche al formalismo spettrale completamente generalizzato, e dunque a reti con un numero arbitrario di layers.

4.2 Futuri sviluppi

La presente tesi, come esposto nella precedente sezione, ha portato allo sviluppo di una promettente generalizzazione del formalismo spettrale. Tuttavia per ottenere una reale comprensione delle potenzialità del formalismo sviluppato sarà necessaria una batteria di test numerici che non abbiamo avuto modo di realizzare durante l'orizzonte limitato di questo lavoro. A seguire 4 linee sperimentali di possibile sviluppo che appaiono particolarmente promettenti.

- Come già detto questo formalismo descrive spontaneamente reti equipaggiate con connessioni skip layers, e sappiamo dalla letteratura inerente le *ResNets* che queste connessioni sono particolarmente utili in reti dalla profondità elevata. Appare dunque ragionevole ipotizzare che il formalismo spettrale generalizzato possa portare ad ottime performance quando utilizzato per descrivere reti pensate per la risoluzione di problemi complessi, e che necessitano dunque di un numero elevato di layers. Sarebbe per questo interessante sperimentare le performances del formalismo su datasets più complessi di MNIST, quali ad esempio FASHION-MNIST o CIFAR 10 [24, 1].

- Il formalismo spettrale generalizzato presenta connessioni ricorrenti e fa uso della matrice di adiacenza come operatore di evoluzione temporale in tempo discreto. Queste caratteristiche, emerse spontaneamente, lo rendono un candidato ideale per operare sulle serie temporali; inoltre la presenza di connessioni a lungo raggio potrebbe in parte mitigare il problema della scomparsa/esplosione del gradiente, in maniera simile a quanto accade con la connessione di memoria a lungo termine nelle celle LSTM. Sfortunatamente in questo lavoro di tesi non abbiamo avuto modo di confrontarci con problemi di learning applicati a serie temporali, ma ci ripromettiamo di approfondire questo tema nel prossimo futuro. Nello specifico un semplice approccio potrebbe essere quello di caricare nuovi elementi della serie nelle attivazioni del primo layer della rete dopo ogni evoluzione temporale, un approccio simile dunque a quello usato nelle *RNNs*; tuttavia potrebbe anche essere opportuno aspettare B evoluzioni temporali prima di caricare un nuovo elemento da processare. Ulteriori indagini su questo tema appaiono necessarie.

- Sebbene nel corso del corso del secondo e terzo capitolo si sia mostrato come il formalismo spettrale generalizzato sia capace di effettuare ricerca d'architettura molte considerazioni sono rimaste del tutto teoriche: abbiamo sperimentato questo processo unicamente su reti a 3 layers, e senza alcun tipo di regolarizzazione, o test su altre diverse possibili inizializzazioni a minima architettura. In futuro sarà cruciale testare il meccanismo di ricerca d'architettura di questo formalismo su reti potenzialmente molto più profonde, e sarà inoltre necessario portare avanti una serie sistematica di tests per mettere a confronto le performances del metodo sotto diversi contesti d'apprendimento.

- L'implementazione del formalismo spettrale standard, come attualmente presente in letteratura, ha molteplici vantaggi. Particolarmente interessante è la possibilità di ottenere performances di un certo livello semplicemente allenando gli autovalori ad autovettori fissi, con una drastica riduzione dunque dello spazio dei parametri sottoposti ad ottimizzazione. Sembrerebbe ragionevole ipotizzare quindi che questo vantaggio computazionale possa estendersi al formalismo spettrale generalizzato. Anche questa prospettiva d'indagine verrà approfondita nel prossimo futuro.

Appendice A

Backpropagation

A.1 Introduzione

La *backpropagation*, come abbiamo già detto nel primo capitolo di questo scritto, non è altro che un algoritmo efficiente per il calcolo del gradiente. In un contesto ideale dove si abbia accesso a risorse computazionali illimitate potremmo usare qualunque altro metodo per calcolare il gradiente, come ad esempio la *differenziazione numerica* o la *differenziazione simbolica*. Tuttavia in un contesto reale, dove si devono tenere in conto gli effetti della limitatezza delle risorse computazionali, la differenziazione numerica soffre di problemi di instabilità, mentre la differenziazione simbolica soffre di problemi di complessità computazionale; backpropagation si rivela la scelta migliore all'atto pratico.
Backpropagation è un caso particolare di *differenziazione automatica* (*Automatic Differentiation*), nello specifico di quella che viene chiamata *Reverse Mode Automatic Differentiation* (*RM-AD*). Consideriamo una funzione F di cui si voglia calcolare le derivate parziali:

$$F : \mathbb{R}^n \to \mathbb{R}^m \tag{A.1}$$

La *RM-AD* è un algoritmo che si rivela estremamente efficace per il calcolo delle derivate nel caso in cui $n > m$, ovvero quando il numero di inputs della funzione è maggiore del numero di outputs. Nel caso in cui si abbia a che fare con $n < m$ si rivela più conveniente un'altra forma di differenziazione automatica, nota come *Forward Mode Automatic Differentiation* (*FM-AD*). Nel contesto di nostro interesse, ovvero il *machine learning*, la funzione di cui si vogliono calcolare le derivate parziali è il *rischio empirico* (la *loss* di fatto) sul *batch* di riferimento, dunque una funzione ad output *scalare* ($m = 1$). Di contro solitamente la dimensione dell'input $\boldsymbol{x}$ è molto maggiore di 1; si pensi ad esempio al caso di MNIST, dove già il singolo input al modello ha dimensione $28 \times 28 = 784$, e dove inoltre un *batch* di input conterrà decine o centinaia di immagini. Si capisce dunque non solo che nel contesto di nostro interesse sarà sempre conveniente utilizzare la *RM-AD*, ma anche che finiremo sempre nel caso particolare in cui $m = 1$.
Il termine *backpropagation* è nato per indicare l'implementazione dell'algoritmo *RM-AD* a funzioni con molti input ed un solo output, e nello specifico in letteratura viene quasi esclusivamente utilizzato per parlare dell'implementazione al calcolo del

gradiente di una loss su un batch, per il *train* di un modello di *machine learning* (come una rete neurale).
La cosa importante da trarre è che *backpropagation* non è altro che *RM-AD* nel caso $m = 1$. Nel seguito scenderemo in maggiori dettagli su questo tema [29, 12].

A.2 Differenziazione automatica

La differenziazione automatica consiste essenzialmente nel pensare ogni funzione come un'intricata funzione composta di funzioni elementari, la cui derivata è nota, per poi applicare in maniera opportuna la *chain rule* per calcolare la derivata della funzione complessiva.
Per trattare questa tecnica di calcolo delle derivate è fondamentale dunque ripassare le regole di derivazione per funzioni composte (*chain rule*). Date $f : \mathbb{R} \to \mathbb{R}$, $r : \mathbb{R} \to \mathbb{R}$

$$\frac{d}{dx} f \circ r = \frac{d}{dx} f(r(x)) = f'(g(x))r'(x) \tag{A.2}$$

questa è la classica regola di derivazione per le funzioni composte. Il punto cruciale è che questa può essere estesa a classi di funzioni più generali di quelle sovra esposte. Per fare un esempio consideriamo adesso le funzioni $f : \mathbb{R}^n \to \mathbb{R}$, $r : \mathbb{R} \to \mathbb{R}^n$, e definiamo per comodità $g = f \circ r$, si ha:

$$g'(x) = \frac{d}{dx} f(\boldsymbol{r}(x)) = \nabla_{\boldsymbol{r}} f(\boldsymbol{r}(x)) \cdot \boldsymbol{r}'(x) \tag{A.3}$$

questa non è altro che un'ovvia generalizzazione.
La *chain rule* è importante in quanto ci consente di mettere in atto un approccio *divide et impera:* possiamo calcolare separatamente le derivate esterne e quelle interne senza aver bisogno di sapere come sono fatti gli altri pezzi, e questo significa anche che sono possibili percorsi di ottimizzazione del calcolo nel caso in cui la stessa struttura appaia più volte nel corso della derivazione. Possiamo riutilizzare le derivate già calcolate, magari con un'altra funzione in input, senza doverle ricalcolare. L'idea di dividere il problema di derivazione per poi calcolare i vari pezzi separatamente, e ottimizzare il conto quando possibile, è il *cuore* del processo di derivazione (differenziazione) automatica, ed è uno degli aspetti fondamentali che lo distingue dagli altri due metodi di differenziazione sovra citati.
Possiamo completamente generalizzare la *chain rule* nella seguente maniera: supponiamo di avere a che fare con le funzioni $f : \mathbb{R}^n \to \mathbb{R}^m$, $r : \mathbb{R}^p \to \mathbb{R}^n$, allora la loro composizione $g = f \circ r$ avrà la struttura:

$$g : \mathbb{R}^p \to \mathbb{R}^m$$

A questa g può essere applicata la *chian rule* come segue:

$$g' = f' \cdot r' \tag{A.4}$$

dove con queste tre quantità non si intendono numeri o vettori, bensì *matrici* (quindi non stiamo facendo una derivata direzionale, stiamo facendo *tutte* le derivate direzionali rispetto agli assi contemporaneamente). Per usare la corretta terminologia

stiamo calcolando le *matrici jacobiane*, ovvero le matrici con le derivate parziali prime corrette in ogni posizione [27, 37].
Avendo compreso che la *chain rule* ha generale applicabilità possiamo procedere con la trattazione della *differenziazione automatica*.
Supponiamo, per esempio, di aver a che fare con la funzione $y : \mathbb{R}^2 \to \mathbb{R}$:

$$y(\boldsymbol{x}) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \cdot \left[\frac{x_1}{x_2} - e^{x_2}\right] \quad , \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{A.5}$$

Il nostro obbiettivo è ricavare il valore della derivata di questa funzione in un certo punto $\tilde{\boldsymbol{x}}$.
Il primo step nel processo di differenziazione automatica consiste nel creare una struttura *ad hoc* per la funzione in esame, che la rappresenti: ovvero il *grafo computazionale* della funzione; che nel caso di (A.5) sarà fatto nella seguente maniera.

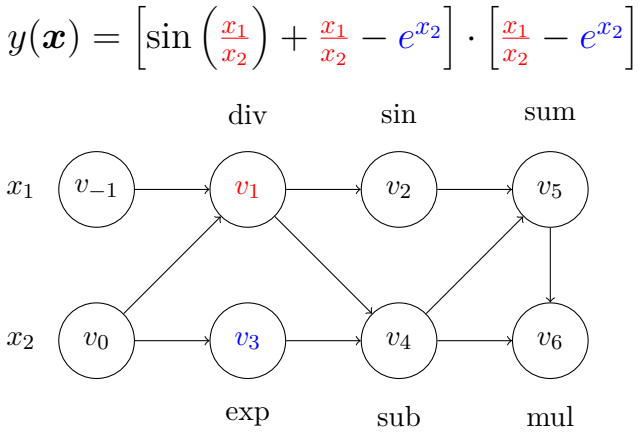

Figura A.1. Rappresentazione grafica del grafo computazionale (correttamente ottimizzato) della funzione (A.5). Possiamo notare come il grafo (che *non* è una rete neurale) riceva gli inputs (a sinistra), che poi filtrano attraverso i nodi del grafo, aventi ciascuno la sua operazione, fino a produrre l'output (nel nodo v_6). Questo ci permette di pensare ad ogni funzione come ad un'enorme funzione composta di funzioni elementari. I nodi sono organizzati in maniera tale da non ripetere operazioni inutilmente. Abbiamo usato la nomenclatura di *Griewank & Walther (2008)* [46].

Come vediamo in Figura A.1 il grafo computazionale di una funzione è una sorta di riassunto schematizzato e ottimizzato della funzione in esame, gli algoritmi che permettono di costruire simili grafi a partire dalle funzioni possono diventare anche molto complessi, dunque in questa sede non li descriveremo, tuttavia dovrebbe essere chiaro come sia sempre possibile almeno in linea di principio costruire il grafo computazionale di una funzione data.
Utilizzando il grafo computazionale di una funzione anziché la funzione stessa si ottengono enormi vantaggi, in quanto moltissime funzioni di interesse, trattando quantità e fenomeni reali, presentano simmetrie e strutture ridondanti, che sono proprio le strutture che il grafo computazionale riesce ad ottimizzare.
A questo punto dovrebbe esser chiaro che possiamo usare questi grafi computazionali non solo per calcolare i valori delle funzioni per certi input, ma anche per operare

sulle funzioni in generale dato che sono costruiti per essere analoghi alle funzioni rappresentate in tutto e per tutto. Ed in particolare possiamo usare le nostre regole di derivazione composta sui grafi ottimizzati, derivando nodo per nodo e facendo poi il prodotto, per ottenere le derivate parziali della funzione in esame.
Tramite il grafo computazionale possiamo calcolare la derivata della funzione in esame attraverso una procedura denominata **forward mode AD (FM-AD):** scelto il punto $\tilde{\boldsymbol{x}}$ in cui calcolare le derivate parziali si parte dall'inizio del grafo computazionale, calcolando per i primi nodi *il loro valore* e *le loro derivate parziali rispetto agli input*, e poi si procede in avanti, propagando entrambe queste infomazioni ai nodi successivi. Data la struttura della *chain rule* conoscere valore e derivata dei nodi precedenti è tutto quello che ci serve per conoscere il valore e la derivata del nodo corrente.
Nello specifico nel nodo i-esimo saremo interessati a calcolare la derivata rispetto ad una delle variabili in input, ad esempio x_1:

$$\frac{\partial v_i}{\partial x_1} \tag{A.6}$$

ma non dobbiamo dimenticarci che stiamo eseguendo i calcoli per mezzo di un grafo computazionale, e che quindi il valore che il nodo assume sarà una certa funzione g dei nodi che vengono prima di lui, solitamente chiamati i **padri** (Pa) del nodo in esame:

$$v_i = g(Pa_1, ..., Pa_n) \tag{A.7}$$

ma dunque possiamo scrivere, secondo le regole di derivazione composta:

$$\frac{\partial v_i}{\partial x_1} = \sum_i \frac{\partial g}{\partial Pa_i} \frac{\partial Pa_i}{\partial x_1}$$

dove la prima derivata sul lato destro è ovviamente facile da calcolare sul momento sapendo l'operazione g contenuta nel nodo i-esimo in esame, mentre la seconda derivata, quella dei padri rispetto alla variabile in input (x_1), è appunto un *messaggio dei padri*, che arriva al nodo dai suoi padri. In questo passaggio è cruciale che al nodo in esame arrivi sia l'informazione sul valore della derivata dei padri sia l'informazione sul valore della funzione calcolata fino al nodo, difatti le prime derivate nella sommatoria vanno ovviamente calcolate proprio su quest'ultimo valore. Una volta calcolata la derivata del nodo i-esimo esso manderà ai suoi *figli* il valore della sua derivata rispetto alla variabile in input ed il valore della funzione fino a lui, imitando quello che i suoi padri prima di lui hanno fatto.
Questo metodo risulta estremamente efficiente per calcolare la derivata in molte situazioni, tuttavia **non** è adatto all'implementazione per il calcolo del gradiente in *machine learning*: difatti si noti che questo metodo permette di calcolare in un passaggio (un passaggio attraverso tutto il grafo computazionale) la derivata parziale rispetto ad una delle variabili in input (nel nostro esempio x_1), questo significa che se il numero dei valori in output per la y supera il numero dei valori in input questo è un ottimo metodo. Tuttavia nell'implementazione per *machine learning* siamo ovviamente interessati al caso dove la y è proprio la *loss*, dunque una funzione con potenzialmente molti input x ma un singolo output. Dunque questo metodo di

forward mode AD costringe a ripetere il calcolo della derivata tante volte quanti sono gli input della funzione in esame per poter ottenere il gradiente. Si capisce dunque che questo non possa essere un buon metodo per il calcolo del gradiente della loss. Fortunatamente esiste anche il **reverse mode AD (RM-AD)**.
Nel *reverse mode* per ogni nodo vogliamo calcolare una nuova quantità: l'*adjoint* $\bar{v}_i$

$$\bar{v}_i = \frac{\partial y}{\partial v_i} \tag{A.8}$$

si noti la differenza fra questa quantità di interesse in *reverse mode* e (A.6), quella che avevamo in *forward mode*. Non siamo più interessati nodo per nodo alla derivata della funzione y calcolata fino al nodo, bensì adesso siamo interessati nodo per nodo al calcolo della derivata della funzione y in toto rispetto al valore contenuto nel nodo. Con le nostre regole di derivazione composta si può dimostrare che questo valore è uguale a:

$$\bar{v}_i = \frac{\partial y}{\partial v_i} = \sum_{k \in Ch_i} \frac{\partial v_k}{\partial v_i} \frac{\partial y}{\partial v_k} \tag{A.9}$$

dove Ch_i sono tutti i *figli* del nodo i-esimo, ovvero tutti i nodi in cui il nodo i-esimo manda il suo output nel grafo computazionale. Il calcolo della prima derivata parziale in (A.9) è complesso: ogni nodo in esame sa bene chi sono i suoi figli, e sarebbe facilmente in grado di calcolare la derivata dei figli rispetto a se stesso, se solo sapesse il valore della funzione fino a lui, ovvero se solo sapesse in che punto calcolare la sua derivata. Vedremo fra poco come questa informazione possa essere ottenuta con un primo *forward pass*. La seconda derivata parziale in (A.9) è quella che riguarda la derivata della funzione y tutta rispetto ai figli, e questa è la parte che viene trasmessa all'indietro dai figli fino al padre in esame. Per questo si chiama *reverse mode!* Concretamente come dicevamo quello che si fa è prima calcolare il grafo computazionale in avanti (*forward pass*), in maniera tale da ricavare il valore della funzione ad ogni nodo, e poi forti di questa conoscenza, che risolve il problema sovra menzionato, si fa una passata indietro (*backward pass*), con i figli che via via trasmettono indietro ai padri la loro derivata (e questa volta non c'è bisogno di trasmettere all'indietro il valore della funzione). Si capisce che alla fine del percorso all'indietro avremo proprio la derivata della y rispetto agli inputs iniziali. Questo processo di *Reverse Mode AD* è quello che da luogo all'algoritmo di **backpropagation**, che fa esattamente la procedura appena descritta per differenziare, applicando il tutto ad una speciale funzione y: la loss di un modello di *machine learning*.
Il vantaggio della *reverse mode* è che come possiamo notare da (A.8) si calcola direttamente la derivata totale, e non la derivata parziale, e dunque possiamo gestire tutti gli input x con una sola passata; lo svantaggio di questo metodo è che calcoliamo la derivata di uno specifico output y, e quindi se la nostra funzione di interesse ha più output dovremo fare tante passate quant'è il numero degli output della funzione in esame [12, 29].

Ottimizzazione al calcolatore ed Ordinamento Topologico
In questa trattazione abbiamo sorvolato su una serie di dettagli tecnici, che però all'atto pratico si rivelano cruciali. In particolare la struttura *DAG* (*Directed Acyclic Graph*) del grafo computazionale ci permette, a mano a mano durante il calcolo, di *de-allocare* molte variabili già utilizzate per ricavare le quantità d'interesse, e che non sono quindi più necessarie. Tuttavia affinché questo processo vada a buon fine è necessario determinare in quale ordine i nodi del grafo vadano calcolati. Questo ordinamento, che di fatto *stira* il grafo computazionale in una sequenza lineare di calcoli, è detto *ordinamento topologico*. Per grafi complessi determinare l'ordinamento topologico non è banale, e richiede l'uso di algoritmi specifici. Questo è un aspetto che non abbiamo trattato in questa sede, ma che è fondamentale per un'implementazione efficiente della differenziazione automatica. La necessità di implementare una moltitudine di diversi algoritmi e processi di ottimizzazione per ottenere un'efficiente differenziazione automatica è uno dei motivi per cui la comunità scientifica si affida sempre più a librerie e pacchetti standard per questi scopi, come ad esempio *TensorFlow* o *PyTorch*, invece di implementare ogni volta da zero l'intera mole algoritmica [35, 2, 3].

Bibliografia

[1] Cifar-10. https://www.cs.toronto.edu/~kriz/cifar.html.

[2] Pytorch. https://pytorch.org/.

[3] Tensorflow. https://www.tensorflow.org/.

[4] New deep learning method adds 301 planets to kepler's total count, 2023. https://www.nasa.gov/missions/kepler/new-deep-learning-method-adds-301-planets-to-keplers-total-count/.

[5] Autopilot e funzionalità di guida autonoma al massimo potenziale, 2024. https://www.tesla.com/it_it/support/autopilot.

[6] Mnist database, 2024. https://en.wikipedia.org/wiki/MNIST_database.

[7] Nn svg, 2024. https://alexlenail.me/NN-SVG/.

[8] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[10] G. Cowan. *Statistical data analysis*. Oxford University Press, USA, 1998.

[11] Google DeepMind. Alphago - the movie | full award-winning documentary, 2020. https://www.youtube.com/watch?v=WXuK6gekU1Y.

[12] Baydin et al. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 2017.

[13] Bertoin et al. Numerical influence of relu'(0) on backpropagation. *Advances in Neural Information Processing Systems*, 2021.

[14] Buffoni et al. Spectral pruning of fully connected layers. *Scientific Reports*, 2022.

[15] Chicchi et al. Training of sparse and dense deep neural networks: Fewer parameters, same performance. *Physical Review E*, 2021.

[16] Giambagli et al. Machine learning in spectral domain. *Nature communications*, 2021.

[17] Hochreiter et al. Long short-term memory. *Neural computation*, 1997.

[18] Jumper et al. Highly accurate protein structure prediction with alphafold. *Nature*, 2021.

[19] Kurt Hornik et al. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989.

[20] Liu et al. Darts: Differentiable architecture search. *2019 International Conference on Learning Representations*, 2019.

[21] Silver et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 2018.

[22] Srivastava et al. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15, 2014.

[23] Valizadegan et al. Exominer: A highly accurate and explainable deep learning classifier that validates 301 new exoplanets. *The Astrophysical Journal*, 2022.

[24] Xiao et al. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv e-prints*, 2017.

[25] Yann Lecun et al. The mnist database.

[26] Yann Lecun et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *The Statistical Mechanics Perspective*, 2000.

[27] E. Giusti. *Analisi Matematica 2.* Bollati Boringhieri, 3nd edition, 2003.

[28] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research*, 2010.

[29] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.

[30] Aurélien Géron. California housing prices.

[31] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning. Data Mining, Inference, and Prediction.* Springer, 2nd edition, 2009.

[32] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

[33] Feng hsiung Hsu. *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion.* Princeton University Press, 2002.

[34] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

[35] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 3rd edition, 1997.

[36] N. Linde. The machine that changed the world, 1992.

[37] Lynn H. Loomis and Shlomo Sternberg. *Advanced Calculus*. Jones and Bartlett Publishers, revised edition, 1990.

[38] Tzon-Tzer Lu and Sheng-Hua Shiou. Inverses of 2×2 block matrices. *Computers & Mathematics with Applications*, 43(1):119–129, 2002.

[39] Junling Luo, Zhongliang Zhang, Yao Fu, and Feng Rao. Time series prediction of covid-19 transmission in america using lstm and xgboost algorithms. *Results in Physics*, 2021.

[40] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 1943.

[41] D. Milmo. Chatgpt reaches 100 million users two months after launch. *The Guardian*, 2023.

[42] OpenAI. Gpt-4 technical report. *ArXiv*, 2023.

[43] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 2002.

[44] A. M. TURING. I.—computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

[45] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, 2001.

[46] Andreas Griewank & Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.