Alma Mater Studiorum - - Università di Bologna

Dipartimento di Fisica e Astronomia

Corso di Laurea in Fisica

Neural Ordinary Differential Equations e le loro applicazioni

Relatore:
Chiar.mo Prof.
Gastone Castellani

Correlatore:
Dr. Federico Magnani

Presentata da:
Martino Pasqualotto

Sessione ottobre 2024

Anno Accademico 2024/2025

Ringraziamenti

Innanzitutto vorrei ringraziare la mia famiglia che ha sempre creduto in me e mi ha sempre aiutato in questi anni di esperienza universitaria a Bologna. Un caloroso ringraziamento va anche a tutti i miei amici che ho conosciuto durante questo percorso che lo hanno reso meno difficile e più piacevole. Infine vorrei ringraziare il prof. Gastone Castellani e Federico Magnani che mi hanno aiutato nella scrittura di questa tesi.

Abstract

L'elaborato propone una breve introduzione alle *neural ordinary differential equations*.
Esso si compone di un'introduzione ai *feedforward neural networks*, passando per la definizione di neurone artificiale, dall'enunciato dell'*universal approximation theorem* fino al *training* del modello attraverso la *backpropagation*.
In seguito si approfondiscono alcuni elementi fondamentali dell'*automatic differentiation*, studiando la *forward mode* e la *reverse mode*.
Successivamente si arriva alla definizione di *neural ordinary differential equation*, analizzando qualche esempio semplice e le possibili modalità di *training*.
Infine viene presentata più nel dettaglio un'applicazione delle *neural ordinary differential equations* alla *precision medicine*.

Indice

Capitolo 1

Feedforward Neural Networks

1.1 Introduzione

I *neural networks* si sono imposti negli ultimi anni al centro della ricerca nel dominio dell'intelligenza artificiale, date le loro notevoli prestazioni in vari campi, tra cui il riconoscimento di immagini, di suoni, la traduzione automatica e molti altri.
Essi si inseriscono all'interno dei metodi di *supervised learning*, dove dati di input e output contribuiscono all'allenamento di un modello, costruendo una funzione che mappa nuovi dati di input in valori di output attesi, che potranno essere comparati con dati sperimentali valutando così la *performance* del modello.
I neural networks sono il cuore di quello che viene definito *deep learning*, ovvero l'insieme dei modelli dove i dati passano attraverso diversi *layers* di trattamento, dove ogni *layer* calcola i valori per quello successivo affinché l'informazione venga elaborata in maniera sempre più completa.
Nei *neural networks* ogni *layer* è composto da neuroni artificiali.

1.1.1 Neurone artificiale

Il concetto di neurone artificiale è stato creato verso la metà degli anni '50 cercando di modellizzare il funzionamento di un neurone biologico. In modo molto semplice, possiamo dire che un neurone biologico è una cellula in grado di rilevare segnali da parte di altri neuroni e rinviare un segnale in uscita che non sia funzione lineare del segnale di entrata [9].
Possiamo modellizare questo comportamento in due tappe:

- a partire da una variabile di entrata $x = (x_1, \dots, x_d)$, se ne fa una combinazione lineare

$$u(x) = w_0 + w_1 x_1 + \cdots + w_d x_d = \langle w, x \rangle$$

- applichiamo in seguito una trasformazione non lineare σ a questo risultato

$$z(x) = \sigma(w_0 + w_1 x_1 + \cdots + w_d x_d)$$

I coefficienti w_i sono chiamati **pesi** del neurone e σ è detta **funzione d'attivazione**. Alcuni esempi di funzioni d'attivazione sono

- funzione sigmoidea:
$$sig(u) = \frac{1}{1+e^{-u}}$$
- tangente iperbolica:
$$tanh(u)$$
- ReLU:
$$f(u) = \begin{cases} u & \text{se } u \geq 0 \\ 0 & \text{se } u < 0 \end{cases}$$

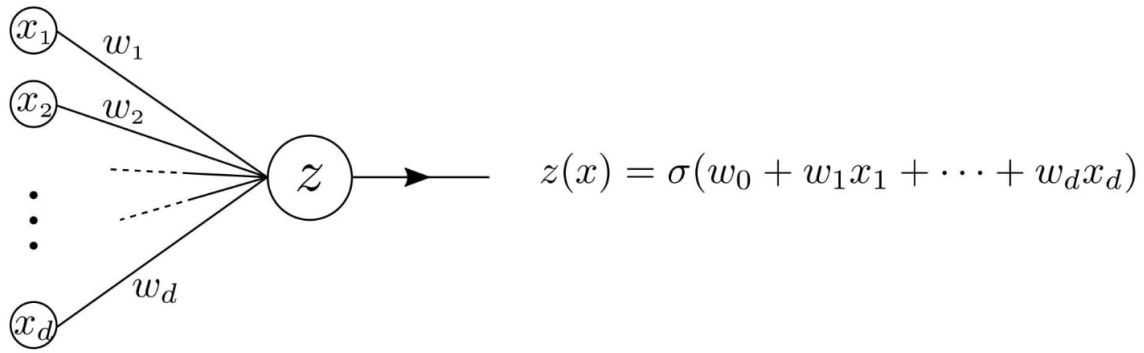

Figura 1.1: Rappresentazione schematica di un neurone artificiale [9].

1.2 Definizione

Un *feedforward neural network* è composto da Q *hidden layers*, ognuna delle quali ha un numero variabile di neuroni artificiali, che qui di seguito chiameremo per semplicità neuroni. Ogni neurone riceve in ingresso l'insieme dei segnali del *layer* precedente, mentre l'entrata del primo *layer*, chiamato *input layer*, è costituita dalle d componenti della variabile di entrata $x = (x_1, \ldots, x_d)$. Dopo l'ultimo *hidden layer*, si ha un *output layer*.

Definiamo le seguenti quantità:

- m_q il numero di neuroni in ciascun *hidden layer*, con $1 \leq q \leq Q$
- $z_i^{(q)}$ il segnale di uscita del neurone i del *layer* q, con $1 \leq i \leq m_q$
- $w_{ij}^{(q)}$ i pesi di uscita del neurone i del *layer* $q-1$ nell'entrata del neurone j del *layer* q, dove $1 \leq i \leq m_{q-1}$ e $1 \leq j \leq m_q$

Quindi le espressioni degli output dei neuroni sono

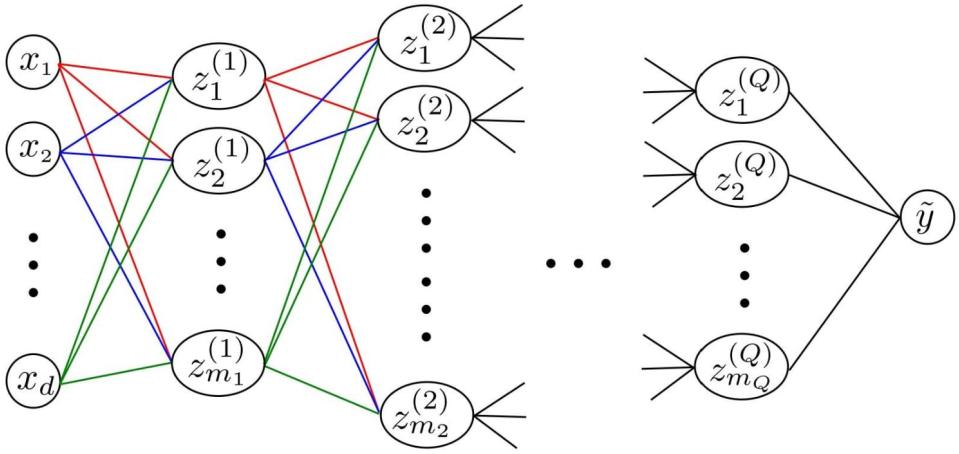

Figura 1.2: Rappresentazione di un *feedforward neural network* per un problema di regressione [9].

$$u_j^{(q)} = w_{0j}^{(q)} + \sum_{i=1}^{m_{q-1}} w_{ij}^{(q)} z_i^{(q-1)} \tag{1.1}$$

$$z_j^{(q)} = \sigma(u_j^{(q)}) \tag{1.2}$$

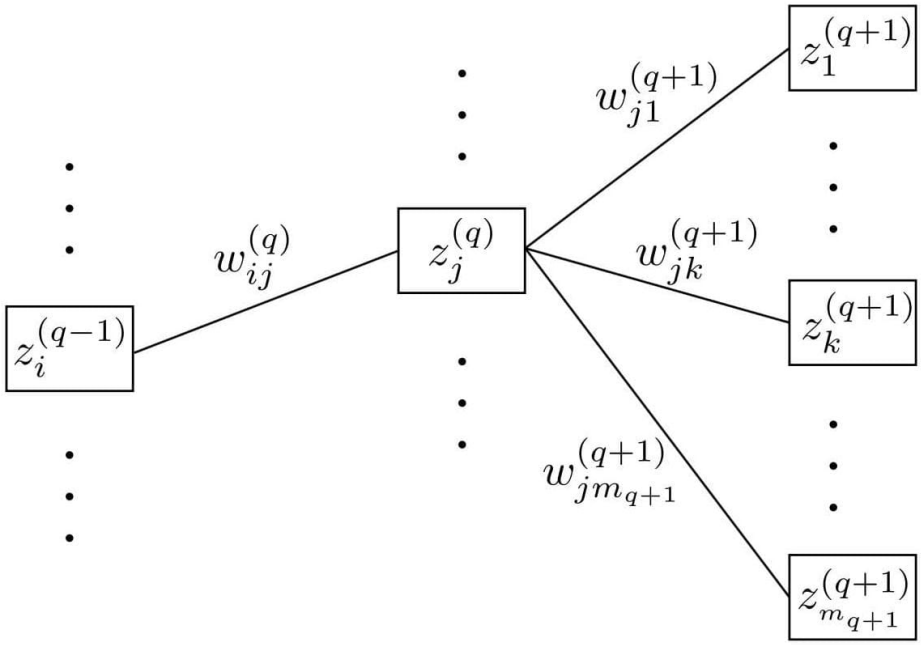

Figura 1.3: Rappresentazione di 3 *hidden layers* consecutivi di un *neural network* [9].

1.2.1 Universal Approximation Theorem

In teoria, finché abbiamo abbastanza dati, energia e tempo a disposizione, possiamo aspettarci di riuscire ad allenare un *feedforward network* che approssima una data funzione con una precisione arbitraria. Questo principio è dimostrato ed è noto come Universal Approximation

Theorem [10], che afferma che un *feedforward network* con un numero arbitrario di neuroni è un approssimatore universale delle funzioni continue.

Definizione 1.1. *Sia $\rho : \mathbb{R} \to \mathbb{R}$ una qualsiasi funzione continua. Sia $\mathcal{N}_d^\rho$ l'insieme di feedforward neural networks con funzione di attivazione ρ, con d neuroni nell'input layer e un singolo hidden layer con un numero arbitrario di neuroni.*

Teorema 1.1. *Sia $K \subseteq \mathbb{R}^d$ compatto. Allora $\mathcal{N}_d^\rho$ è denso in $C(K)$[1] se e solo se ρ è non polinomiale.*

Esistono enunciati analoghi di questo teorema. Ad esempio, anche un *feedforward network* con un numero di *layers* arbitrario è un approssimatore universale delle funzioni continue.

Definizione 1.2. *Sia $\rho : \mathbb{R} \to \mathbb{R}$ e $d_x, d_y, d_w \in \mathbb{N}$. Sia $\mathcal{NN}_{d_x,d_y,d_w}^\rho$ l'insieme di feedforward neural networks con d_x neuroni nell'input layer, d_y neuroni nell'output layer e con un arbitrario numero di hidden layers con d_w neuroni e con funzione di attivazione ρ.*

Teorema 1.2. *Sia $\rho : \mathbb{R} \to \mathbb{R}$ una qualsiasi funzione continua non affine*[2]*, che ha derivata continua e non nulla in almeno un punto del dominio. Sia $K \subseteq \mathbb{R}^{d_x}$ compatto. Allora $\mathcal{NN}_{d_x,d_y,d_w}^\rho$ è denso in $C(K; \mathbb{R}^{d_y})$.*[3]

1.3 Training

L'allenamento dei *neural networks* avviene generalmente tramite la minimizzazione del rischio empirico. Questo metodo è basato sull'idea di minimizzare la differenza, calcolata attraverso una *loss function* [4], tra l'output predetto dal modello e quello di un certo *dataset*. Esso si effettua attraverso l'ottimizzazione dei parametri del modello calcolando il gradiente della *loss function* rispetto ad essi.

1.3.1 Algoritmo di backpropagation del gradiente

Al fine di ottimizzare un *neural network*

- si inizializzano i pesi di ciascun neurone in modo aleatorio [5]
- si fa evolvere i coefficienti in modo iterativo:

[1] Con $C(K)$ si intende lo spazio delle funzioni continue $f : K \to \mathbb{R}$.

[2] Con funzione affine si intende una funzione nella forma $f(x) = ax + b$. Le funzioni affini rappresentano rette o, nel caso di più variabili, iperpiani.

[3] Con $C(K; \mathbb{R}^{d_y})$ si intende lo spazio delle funzioni continue $f : K \to \mathbb{R}^{d_y}$.

[4] Un esempio molto comune di *loss function* è la norma euclidea, definita in questo caso come $C\left(y, f(x)\right) = \frac{1}{2}\left[y - f(x)\right]^2$, dove $f(x)$ è l'output predetto dal *neural network* mentre y è quello presente nel *dataset*, corrispondente all'input x.

[5] Esistono diverse strategie di inizializzazione la cui efficacia dipende particolarmente dalla funzione di attivazione utilizzata. Normalmente si utilizza una distribuzione normale con media nulla e deviazione standard della forma $\sqrt{2/n_{in}}$, dove n_{in} è la dimensione del segnale d'entrata di ogni *layer*.

1. si calcola la derivata della *loss function* rispetto a ciascun coefficiente del modello : $\frac{\partial C}{\partial w_{ij}^{(q)}}$
2. si aggiornano i coefficienti nel modo seguente

$$w_{ij}^{(q)} \to w_{ij}^{(q)} - \eta_k \frac{\partial C}{\partial w_{ij}^{(q)}}$$

dove η_k è detto *learning rate* della k-esima iterazione

Per calcolare $\frac{\partial C}{\partial w_{ij}^{(q)}}$, consideriamo 3 *layers* consecutivi come quelli in fig.1.3, aventi indici $q-1$, q e $q+1$.

Innanzitutto, si calcolano, a partire dai dati di input, tutti i valori delle funzioni $u_j^{(q)}$ e $z_j^{(q)}$, dal primo all'ultimo *layer*, in un processo chiamato **forward pass**.
In seguito, consideriamo la *loss function* C, che dipende dai valori di $u_j^{(q)}$, che sono funzione a loro volta dei $w_{ij}^{(q)}$:

$$C = C\left[u_j^{(q)}\left(w_{ij}^{(q)}\right)\right]$$

Possiamo quindi scrivere

$$\frac{\partial C}{\partial w_{ij}^{(q)}} = \frac{\partial C}{\partial u_j^{(q)}} \frac{\partial u_j^{(q)}}{\partial w_{ij}^{(q)}}$$

Dalle equazioni 1.1 e 1.2 si ha

$$\frac{\partial u_j^{(q)}}{\partial w_{0j}^{(q)}} = 1 \qquad \frac{\partial u_j^{(q)}}{\partial w_{ij}^{(q)}} = z_i^{(q-1)}$$

Da cui

$$\boxed{\frac{\partial C}{\partial w_{0j}^{(q)}} = \frac{\partial C}{\partial u_j^{(q)}} \qquad \frac{\partial C}{\partial w_{ij}^{(q)}} = \frac{\partial C}{\partial u_j^{(q)}}\ z_i^{(q-1)}} \tag{1.3}$$

Se adesso consideriamo C come una funzione di $z_j^{(q)}$, che dipende a sua volta da $u_j^{(q)}$,

$$\frac{\partial C}{\partial u_j^{(q)}} = \frac{\partial C}{\partial z_i^{(q)}} \frac{\partial z_i^{(q)}}{\partial u_j^{(q)}} = \frac{\partial C}{\partial z_i^{(q)}}\ \sigma'\left(u_j^{(q)}\right)$$

dove σ' è la derivata della funzione di attivazione $\sigma(u)$. Possiamo vedere dalla fig.1.3 che tutti i termini $z_k^{(q+1)}$, per $1 \le k \le m_{q+1}$, dipendono da $z_k^{(q)}$, come anche tutti i $u_k^{(q+1)}$. Possiamo quindi scrivere

$$C = C\left[u_1^{(q+1)}(z_j^{(q)}), \ldots, u_{m_{q+1}}^{(q+1)}(z_j^{(q)})\right]$$

$$\frac{\partial C}{\partial z_j^{(q)}} = \sum_{k=1}^{m_{q+1}} \frac{\partial C}{\partial u_k^{(q+1)}} \frac{\partial u_k^{(q+1)}}{\partial z_j^{(q)}} = \sum_{k=1}^{m_{q+1}} \frac{\partial C}{\partial u_k^{(q+1)}}\ w_{jk}^{(q+1)}$$

Inserendo questo risultato nell'espressione precedente si ottiene

$$\boxed{\frac{\partial C}{\partial u_j^{(q)}} = \left(\sum_{k=1}^{m_{q+1}} \frac{\partial C}{\partial u_k^{(q+1)}} \; w_{jk}^{(q+1)} \right) \sigma' \left(u_j^{(q)} \right)} \tag{1.4}$$

Quest'ultima equazione ci mostra che se è possibile calcolare $\frac{\partial C}{\partial u_k^{(q+1)}}$ per tutti i neuroni del *layer* $q+1$, allora si può calcolare anche $\frac{\partial C}{\partial u_k^{(q)}}$ per tutti i neuroni del *layer* q. È dunque sufficiente saper calcolare questi termini per l'ultimo *layer* del *neural network* per poterli determinare per ogni *layer*. Ciò costituisce il processo di **backpropagation** del gradiente.
Ad ogni iterazione (forward pass $\rightarrow$ backpropagation $\rightarrow$ aggiornamento dei pesi $w_{ij}^{(q)}$) il valore dei pesi evolve in modo da diminuire il rischio empirico. Questo processo continua finché non si raggiunge un determinato criterio stabilito dall'utilizzatore.

Capitolo 2

Automatic Differentiation

L'*automatic differentiation* è un insieme di tecniche per il calcolo automatico delle derivate di una funzione matematica implementata da un programma informatico. Essa si basa sul fatto che l'implementazione di una funzione, indipendentemente dalla sua complessità, si riduce all'esecuzione di una serie di operazioni aritmetiche e funzioni elementari, le cui derivate sono note [2].
Le derivate degli output del programma (variabili dipendenti) rispetto ai suoi input (variabili indipendenti) sono calcolate applicando la regola della catena. Ad esempio, data

$$y(x) = w_n(w_{n-1}(w_{n-2}(\dots(w_1(x))\dots)))$$

la sua derivata è

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}w_n}{\mathrm{d}w_{n-1}}\frac{\mathrm{d}w_{n-1}}{\mathrm{d}w_{n-2}}\frac{\mathrm{d}w_{n-2}}{\mathrm{d}w_{n-3}}\cdots\frac{\mathrm{d}w_1}{\mathrm{d}x}$$

Se la catena di operazioni è computata a partire dall'input si parla di *forward mode*, eseguita calcolando ricorsivamente

$$\frac{\mathrm{d}w_i}{\mathrm{d}x} = \frac{\mathrm{d}w_i}{\mathrm{d}w_{i-1}}\frac{\mathrm{d}w_{i-1}}{\mathrm{d}x}$$

Se invece è computata a partire dall'output si parla di *reverse mode*, eseguita calcolando ricorsivamente

$$\frac{\mathrm{d}y}{\mathrm{d}w_i} = \frac{\mathrm{d}y}{\mathrm{d}w_{i+1}}\frac{\mathrm{d}w_{i+1}}{\mathrm{d}w_i}$$

2.1 Le due trasformazioni dell'automatic differentiation: JVPs and VJPs

L'*automatic differentiation* è basata su due trasformazioni: Jacobian-vector products (JVPs) e vector-Jacobian products (VJPs) [8].

Matematicamente, JVP è la funzione che associa

$$(x, v) \to (f(x), Df(x)v)$$

dove $f : \mathbb{R}^n \to \mathbb{R}^m$, $x \in \mathbb{R}^n$, $v \in \mathbb{R}^n$ e $Df(x)$ è il jacobiano di f calcolato in x. Il JVP serve quindi a calcolare, data una variazione nell'input v, quanto varia l'output nello sviluppo di Taylor al primo ordine. Inoltre, se poniamo $v = (0, \dots, 1, \dots, 0)$, possiamo valutare una colonna del jacobiano alla volta.

Al contrario, il VJP permette di calcolare il jacobiano una riga alla volta. Infatti è la funzione che associa

$$(x, w) \to (f(x), w^T Df(x))$$

dove $f : \mathbb{R}^n \to \mathbb{R}^m$, $x \in \mathbb{R}^n$, $w \in \mathbb{R}^m$.

Esistono varie librerie software che hanno già implementate queste due funzioni, tra cui *JAX* e *pytorch*.

2.2 Forward mode

Un modo molto diffuso di rappresentare funzioni matematiche in un programma è attraverso un *directed acyclic graph*, un grafo che non ha cicli diretti, ovvero, dato un vertice, non è possibile tornare ad esso percorrendo gli archi del grafo. I vertici rappresentano le variabili mentre gli archi rappresentano le operazioni matematiche.

Data una funzione scalare $L : \mathbb{R}^p \to \mathbb{R}$, possiamo chiamare $v_{-p+1}, v_{-p+2}, \dots, v_0 = \theta_1, \theta_2, \dots, \theta_p$ gli input della funzione, $v_1, \dots, v_{m-1}$ le variabili intermedie e $v_m = L(\theta)$ l'output finale. Ciò può essere fatto in un modo in cui ogni variabile v_i è calcolata come funzione delle variabili precendenti v_j, con $j < i$. Una volta composto il grafo, possiamo calcolare la derivata di ogni vertice rispetto ad ogni altro vertice da cui esso dipende ricorsivamente attraverso la formula

$$\frac{\partial v_i}{\partial v_j} = \sum_{w \text{ tale che } w \to v_i} \frac{\partial v_i}{\partial w} \frac{\partial w}{\partial v_j} \tag{2.1}$$

con $j = -p+1, \dots, m$. Ciò è possibile in *forward mode* poiché il termine $\frac{\partial w}{\partial v_j}$ è già stato calcolato nell'iterazione precedente.

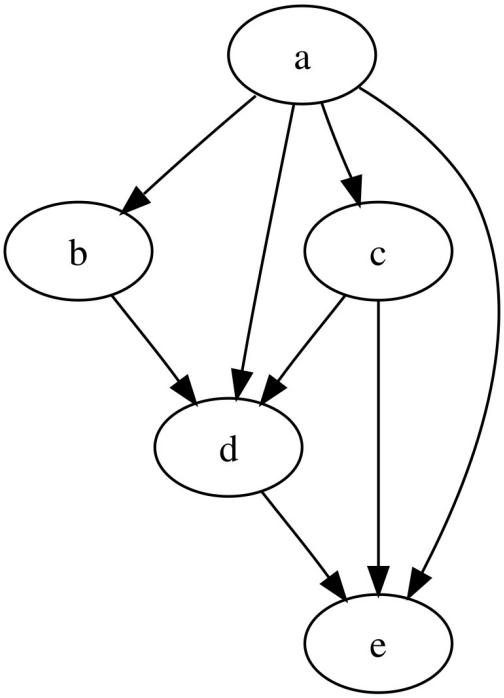

Figura 2.1: Esempio di *directed acyclic graph* [2].

Questo calcolo è implementato attraverso la computazione in sequenza di JVPs. Ipotizzando che la funzione $L(\theta)$ sia una composizione di k funzioni intermedie

$$L(\theta) = l \circ g_k \circ \dots \circ g_2 \circ g_1(\theta)$$

con $g_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$, dove $d_0 = p$, e $l : \mathbb{R}^{d_k} \to \mathbb{R}$. Se perturbiamo il parametro $\theta \to \theta + \delta\theta$, ciò causa una variazione $L(\theta) \to L(\theta) + \delta L$ nell'output che può essere calcolata al primo ordine come

$$\delta L = \nabla_\theta L \cdot \delta\theta = \nabla l \cdot Dg_k \cdot \ldots \cdot Dg_2 \cdot Dg_1 \cdot \delta\theta \tag{2.2}$$

Possiamo implementare questo calcolo defininedo la perturbazione intermedia δg_j, calcolata attraverso il JVP.

$$\begin{aligned} \delta g_0 &= \delta\theta \\ \delta g_j &= Dg_j \cdot \delta g_{j-1} \qquad j = 1, 2, \ldots, k \\ \delta L &= \nabla l \cdot \delta g_k \end{aligned}$$

Per $\|\delta\theta\|_2 = 1$, si ottiene la derivata direzionale di L in θ nella direzione $\delta\theta$. Al fine di calcolare il gradiente totale della funzione $\nabla L \in \mathbb{R}^p$, occorre eseguire questa operazione p volte, una per ogni dimensione dell'input. Questa operazione ha quindi un costo computazionale proporzionale a kp.
È importante notare che anche per funzioni non scalari, ad esempio $f : \mathbb{R}^p \to \mathbb{R}^q$ il numero di operazioni eseguite è pressoché identico, dato che il JVP calcola il jacobiano colonna per colonna. La *forward mode* è dunque conveniente se $p \ll q$.

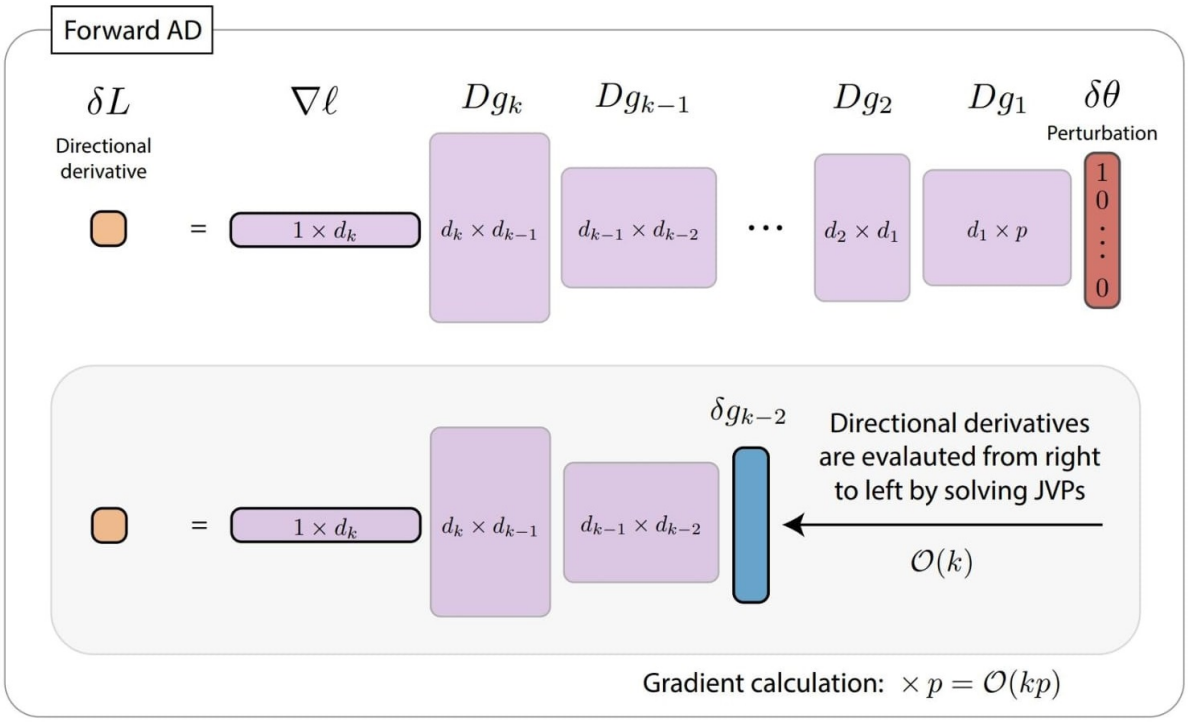

Figura 2.2: Rappresentazione schematica della computazione di ∇L nella *forward mode* [2].

2.3 Reverse mode

La *reverse mode*, conosciuta come *adjoint method* o come *backpropagation* nel campo del *machine learning*, analoga al metodo di *training* dei *neural networks* di cui abbiamo già parlato nella

sezione 1.3.1, si basa sul fatto che possiamo calcolare il gradiente di una qualsiasi funzione, ad esempio $L(\theta)$ già definita nel paragrafo precedente, nello stesso modo espresso nell'equazione 2.1, ma procedendo in ordine decrescente, con $j = m, m-1, \ldots, -p+1$.

$$\bar{v}_i = \frac{\partial l}{\partial v_i} = \sum_{w \text{ tale che } w \to v_i} \bar{w} \frac{\partial w}{\partial v_i}$$

La quantità $\bar{w}$ è detta *adjoint* ed è la derivata dell'output rispetto alle variabili di input o a quelle intermedie.

Dato che in *reverse mode* i valori $\bar{w} = \frac{\partial L}{\partial w}$ vengono calcolati in ordine inverso, abbiamo bisogno di tenere traccia di tutti i valori delle variabili intermedie per poter poi sostituirli nel calcolo delle derivate. Ciò fa in modo che rispetto alla *forward mode*, dove tutti i valori delle variabili intermedie e delle loro derivate parziali vengono calcolate nello stesso passaggio, evitando così di doverle immagazzinare in memoria, questa tecnica richieda più memoria disponibile.

L'idea della **reverse mode** è di calcolare i termini dell'equazione 2.2 procedendo da sinistra verso destra. È possibile calcolare $\nabla L \in \mathbb{R}^p$ applicando ripetutamente il VJP, ovvero la funzione che associa $\bar{y} \to \bar{y}^T \cdot Dg_j$, a tutte le funzioni intermedie.

$$\begin{aligned} \bar{g}_k^T &= \nabla l \\ \bar{g}_{j-1}^T &= \bar{g}_j^T \cdot Dg_j \qquad j = k, k-1, \ldots, 1 \\ \nabla L &= \bar{g}_0 \end{aligned}$$

In pratica, attraverso la quantità $\bar{g}_j$ stiamo calcolando ∇L secondo l'equazione 2.2 partendo da sinistra, aggiungendo al calcolo via via i vari jacobiani delle trasformazioni intermedie.

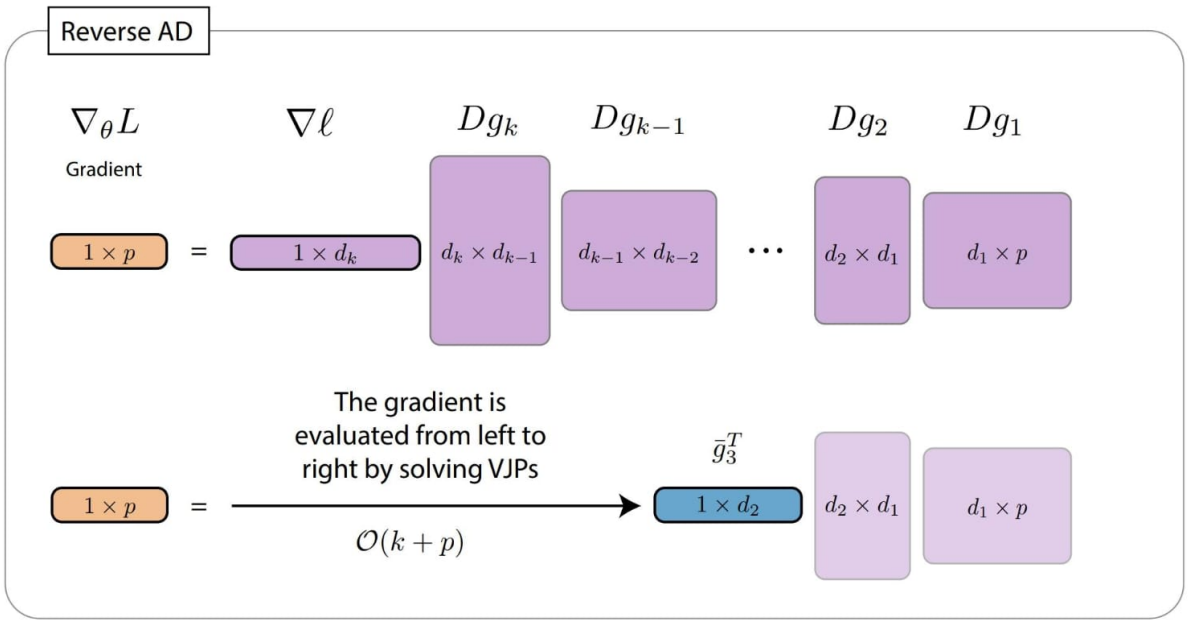

Figura 2.3: Rappresentazione schematica della computazione di ∇L nella *reverse mode* [2].

Nel nostro caso L è una funzione scalare e basta eseguire questo procedimento una sola volta. Se invece avessimo una funzione a valori vettoriali $f : \mathbb{R}^p \to \mathbb{R}^q$ dovremmo ripeterlo q volte, una per ogni dimensione dell'output, dato che componiamo il jacobiano di f una riga alla volta. Il costo computazionale della *reverse mode* è quindi proporzionale a kq.

Nel caso dei **neural networks**, la *loss function* è scalare, è molto più efficiente in questo caso utilizzare la *reverse mode* piuttosto che la *forward mode*.

Capitolo 3

Neural Ordinary Differential Equations

3.1 Definizione

Un'*equazione differenziale ordinaria neurale*, o *neural ordinary differential equation*, è un'equazione differenziale ordinaria che usa un *neural network* per parametrizzare la derivata di una funzione [10]. È espressa nella forma

$$y(0) = y_0 \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)) \tag{3.1}$$

dove $f_\theta : \mathbb{R} \times \mathbb{R}^{d_1 \times \ldots \times d_k} \to \mathbb{R}^{d_1 \times \ldots \times d_k}$ è un qualsiasi *neural network*, θ è il vettore dei parametri del *neural network* e $y : [0, T] \to \mathbb{R}^{d_1 \times \ldots \times d_k}$ è la soluzione. La variabile y_0 è l'input, mentre $y(T)$ è l'output.

Come esempio, possiamo immaginare di osservare un'immagine $\in \mathbb{R}^{3 \times 32 \times 32}$ (a colori con 32 pixels) che vorremmo classificare come immagine di un gatto o di un cane. L'immagine è l'input y_0, che facciamo evolvere integrando numericamente il *neural network* fino ad ottenere l'output $y(T) \in \mathbb{R}^{3 \times 32 \times 32}$, al quale applichiamo una trasformazione affine l_θ, per avere un vettore $\in \mathbb{R}^2$, seguita dalla funzione *softmax*, che contiene le due probabilità che l'immagine rappresenti un gatto o un cane.

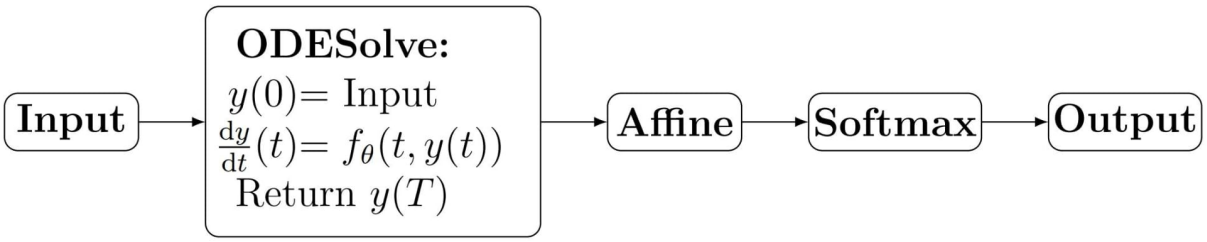

Figure 3.1 Rappresentazione schematica di un possibile utilizzo di una *neural ODE* [10].

Matematicamente, questo processo può essere espresso come

$$\mathrm{softmax}\left(l_\theta\left(y(0) + \int_0^T f_\theta(t, y(t))dt\right)\right)$$

Una rappresentazione pratica del procedimento spiegato sopra è mostrata in Fig. 3.2, in cui due gruppi di dati non separabili al tempo di input sono trasformati, tramite l'evoluzione definita da un neural ODE utilizzato proprio a scopo di classificazione, in modo da essere linearmente separabili al tempo di output. Così, sarà sufficiente applicare un classificatore anche molto semplice in questo nuovo spazio.

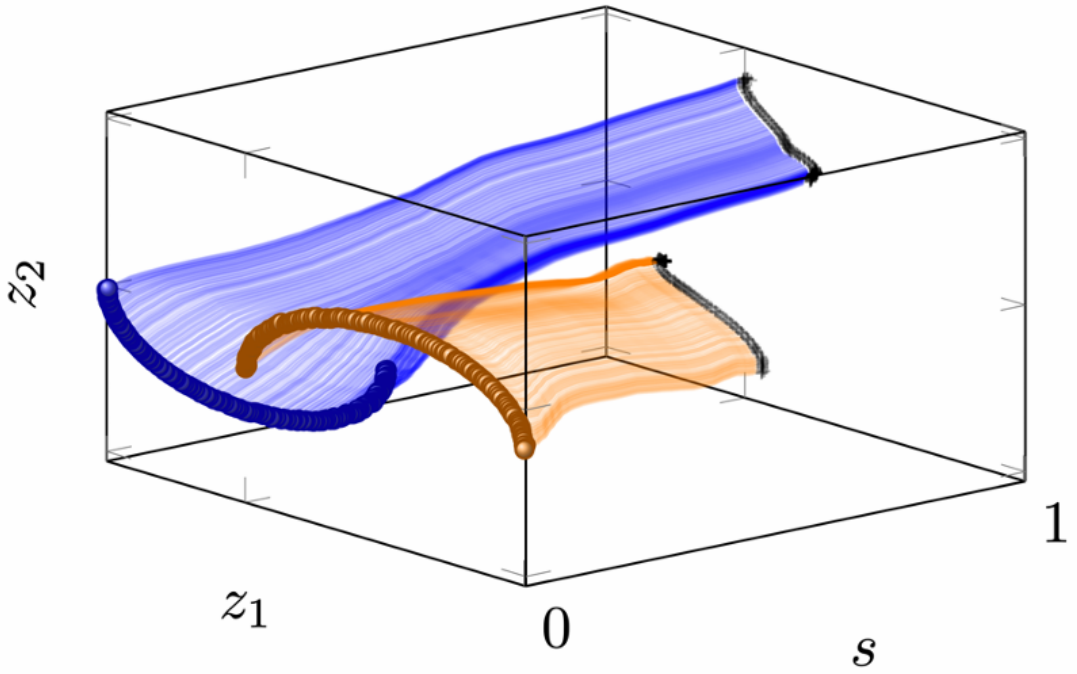

Figure 3.2 Esempio di come due gruppi di dati non separabili linearmente in input possono essere trasformati tramite un campo vettoriale in modo da essere linearmente separabili in output. Dopodiché, si potrebbe applicare un classificatore molto semplice per distinguere i due gruppi. Figura presa da [6].

3.1.1 Continuous-depth neural networks

Le *neural ordinary differential equations*, che di seguito chiameremo *neural ODEs* per semplicità, possono essere viste anche come il passaggio al continuo dei *residual networks*.

In un *residual neural network*, invece di apprendere direttamente una mappatura complessa $y(x)$, il modello apprende una funzione residua $f_\theta(x)$, che è la differenza tra l'output desiderato e l'input stesso. Un *residual network* è formato da *residual blocks*, composti da un numero variabile di *layers*. Il *residual block* può essere formalizzato come

$$y(x) = f_\theta(x) + x$$

dove $y(x)$ è la funzione che la rete dovrebbe apprendere, $f_\theta(x)$ è il residuo, ossia la parte che il modello effettivamente impara, e x è l'input al *residual block*. Dato che l'output di ogni *block*

è l'input del *block* successivo, chiamando y_{j+1} l'output e y_j l'input possiamo scrivere

$$y_{j+1} = y_j + f_\theta(j, y_j)$$

dove $f_\theta(j, \cdot)$ è il $j-$esimo *residual block*.
Ritornando adesso alla definizione di *neural ODE*

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t))$$

discretizzandola a vari tempi t_j separati da intervalli temporali Δt si ottiene

$$\frac{y(t_{j+1}) - y(t_j)}{\Delta t} \approx \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t))$$

Incorporando il termine costante Δt in $f_\theta(t, y(t))$ riotteniamo la definizione di *residual neural network*.

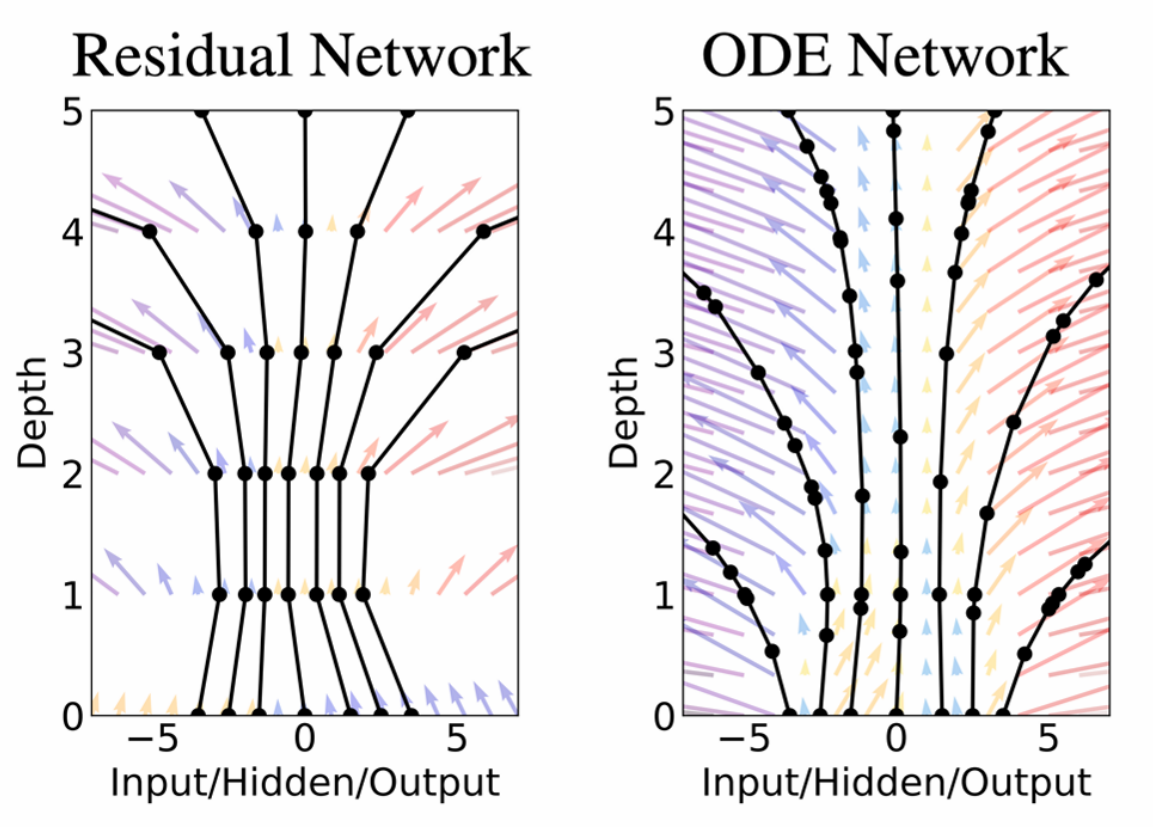

Figure 3.3 Evoluzione definita da un Residual Network (sinistra) ed evoluzione definita da un neural ODE (destra). Un Residual Network applica recursivamente una trasformazione finita, mentre il neural ODE definisce un campo vettoriale. In entrambi i casi le traiettorie indicano l'evoluzione di vari input ai rispettivi output. Figura presa da [7].

Un'idea della differenza tra il processo definito da un residual network e quello definito da una neural ODE è data in Fig. **??**. L'evoluzione definita da un Residual Network, rappresentata a sinistra, corrisponde a una sequenza discreta di trasformazioni finite. Un neural ODE corrisponde a un campo vettoriale che trasforma in modo continuo lo stato, tramite incrementi

infinitesimali. Il modo in cui si valutano gli stati intermedi, nel caso del neural ODE, dipende dall'algoritmo di integrazione scelto. Nella figura le valutazioni sono più frequenti per le traiettorie più complesse, un esempio di *adaptive quadrature*.
Una *neural ODE* può essere quindi vista come il passaggio al continuo di un *residual network*.

3.1.2 Esistenza e unicità

Data una *neural ODE* nella forma definita precedentemente (equazione 3.1), l'esistenza e unicità della soluzione dell'equazione differenziale è garantita dal Teorema di Esistenza di Picard.

Teorema 3.1. *Sia* $f : [0,T] \times \mathbb{R}^d \to \mathbb{R}^d$ *funzione continua in* t *e lipschitziana in* y[1]. *Sia* $y_0 \in \mathbb{R}^d$. *Allora esiste un'unica funzione differenziabile* $y : [0,T] \to \mathbb{R}^d$ *che risolve il sistema*

$$y(0) = y_0 \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t))$$

3.2 Modello Lotka-Volterra

Un'applicazione delle *neural ODEs* è quella di "correggere" delle equazioni teoriche che però nella pratica non sono perfettamente confermate dai dati sperimentali. Un esempio è il modello *Lotka-Volterra*, che descrive l'interazione tra due specie di prede e predatori:

$$\frac{\mathrm{d}x}{\mathrm{d}t}(t) = \alpha x(t) - \beta x(t)y(t) \quad \in \mathbb{R} \tag{3.2}$$
$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = -\gamma x(t) + \delta x(t)y(t) \quad \in \mathbb{R} \tag{3.3}$$

dove $x(t), y(t)$ rappresentano il numero di esemplari della specie rispettivamente di prede e predatori, per ogni $t \in [0,T]$. Tuttavia, previsioni teoriche e osservazioni differiscono di una quantità non trascurabile. Per rimediare a ciò, possiamo introdurre due *neural networks* f_θ, g_θ e considerare invece il modello

$$\frac{\mathrm{d}x}{\mathrm{d}t}(t) = \alpha x(t) - \beta x(t)y(t) + f_\theta(x(t), y(t)) \quad \in \mathbb{R}$$
$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = -\gamma x(t) + \delta x(t)y(t) + g_\theta(x(t), y(t)) \quad \in \mathbb{R}$$

In questo modo, la differenza tra le equazioni 3.2 e 3.3 e i dati osservati viene descritta dai *neural networks* f_θ, g_θ, che permettono di ottenere un modello che descrive il processo considerato in modo molto più accurato.

[1] Ovvero, esiste $C > 0$ tale che $\forall\, t, y_1, y_2,\ \|f_\theta(t, y_1) - f_\theta(t, y_2)\| \leq C \|y_1 - y_2\|$

3.2.1 Loss function e training

Supponiamo di osservare dati sperimentali $x_i(t_j) \in \mathbb{R}$, $y_i(t_j) \in \mathbb{R}$, dove $i = 1, \dots, N$ indica osservazioni indipendenti del processo preso in considerazione (con diverse condizioni iniziali) e $j = 1, \dots, M$ corrispondono a diversi istanti $t_j \in [0, T]$ in cui vengono effettuate le misure. Indichiamo con $x_{x_0,y_0}(t)$, $y_{x_0,y_0}(t)$ le soluzioni con condizioni iniziali $x(0) = x_0$, $y(0) = y_0$.

Il processo di training viene usualmente effettuato attraverso la *stochastic gradient descent*[2] rispetto alla *loss function*

$$\frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} (x_{x_0,y_0}(t_j) - x_i(t_j))^2 + (y_{x_0,y_0}(t_j) - y_i(t_j))$$

In questo caso, l'utilizzo di un *neural network* è un ammissione della nostra incapacità di comprendere appieno il funzionamento del processo considerato, tuttavia possiamo sfruttare l'alta capacità di modellizzazione del *network* sia per comprendere meglio il funzionamento teorico del processo, sia per ottenere delle previsioni più accurate di quelle che otteremmo utilizzando solamente le equazioni di *Lotka-Volterra*.

3.3 Applicazioni

Le *neural ODEs* trovano applicazioni in moltissimi campi, data la loro capacità di descrivere dinamicamente funzioni e andamenti molto complessi.
Alcuni esempi di applicazioni sono:

- **Modelli generativi**. Lo scopo è quello di modellizzare una distribuzione *target* ν a partire da elementi estratti da quella distribuzione. Solitamente si procede a partire da una distribuzione ben conosciuta (ad esempio quella normale nel caso del *continuous normalizing flow*) che viene fatta evolvere utilizzando una *neural ODE* in modo da ottenere una distribuzione che approssima ν.

- **Serie temporali**. Dati irregolari o mancanti sono molto comuni nelle serie temporali discrete. Trattare queste serie discrete in modo continuo è possibile trattare questi dati allo stesso modo di dati regolari presi ad intervalli di tempo costanti. Questo è particolarmente utile in campi come *data science* o il *machine learning*, dove i dati irregolari possono essere difficili da trattare.

- **Modelli fisici, biologici, finanziari,** ... . Modelli teorici descritti da equazioni differenziali sono presenti in moltissimi campi. Tuttavia, spesso questi modelli non riescono a descrivere perfettamente la realtà. Combinando modelli esistenti con il *deep learning*, possiamo ridurre il *gap* tra teoria e osservazioni.

[2] *Stochastic gradient descent* opera similmente a *gradient descent* ma, ad ogni iterazione, sostituisce il valore esatto del gradiente della *loss function* con una stima ottenuta valutando il gradiente solo su un sottinsieme di dati

3.4 Backpropagation per le neural ODEs

Allenare una *neural differential equation* vuol dire eseguire la *backpropagation* attraverso la soluzione. Ci sono due modi principali per effettuare questo processo:

- **Discretise-then-optimise**: Richiede più memoria disponibile, però è veloce e accurato
- **Optimise-then-discretise**: Richiede meno memoria disponibile, tuttavia è un po' lento e meno accurato

3.4.1 Discretise-then-optimise

Questa opzione richiede semplicemente di eseguire la *backpropagation* attraverso le operazioni interne effettuate nel calcolo della soluzione numerica della *neural ODE*.
Dato che la soluzione $y(T)$ viene calcolata come

$$y(T) = y(0) + \int_0^T f_\theta(t, y(t))dt$$

l'algoritmo di risoluzione divide questo calcolo in numerose operazioni elementari, come somme e prodotti, ognuna delle quale è differenziabile. Dato che la soluzione è la composizione di queste operazioni elementari, è anch'essa differenziabile. Per allenare la *neural ODE* si calcola il gradiente della *loss function* rispetto ai parametri θ del *neural network* computando i VJPs di tutte queste operazioni intermedie, come spiegato nella sezione 2.3. In seguito si esegue l'aggiornamento dei parametri e questo processo viene ripetuto finché non si raggiunge un criterio prestabilito.
Dal momento che per performare la *backpropagation* tramite *reverse mode* serve l'intero grafo computazionale della funzione che trasforma l'input nell'output, che in questo caso è dato da tutte le iterazioni dell'integratore numerico, questo metodo è estremamente inefficiente in termini di memoria richiesta. In compenso, in termini di tempo è efficiente ed è anche molto accurato.

3.4.2 Optimise-then-discretise

Questo metodo invece si basa sul differenziare un modello continuo. Dalla *neural ODE* si ottiene un sistema di equazioni differenziali che viene risolto numericamente, la cui soluzione fornisce il gradiente della *loss function* rispetto ai parametri θ del *neural network*.
Rispetto al metodo precedente di discretize-then-optimize, questo approccio richiede molta meno memoria. Tuttavia, è più dispendioso a livello di tempo e senza applicare delle tecniche apposite potrebbe risultare inaccurato.
Una rappresentazione del metodo è data in Fig. **??**, in cui si mostra l'evoluzione "in avanti" (rispetto al tempo che parametrizza l'evoluzione) dell'input fino all'output e l'evoluzione "all'indietro" dell'*aggiunta*, la quantità che, una volta calcolata al tempo 0, rappresenta il gradiente della *Loss Function* rispetto ai parametri dell'evoluzione.

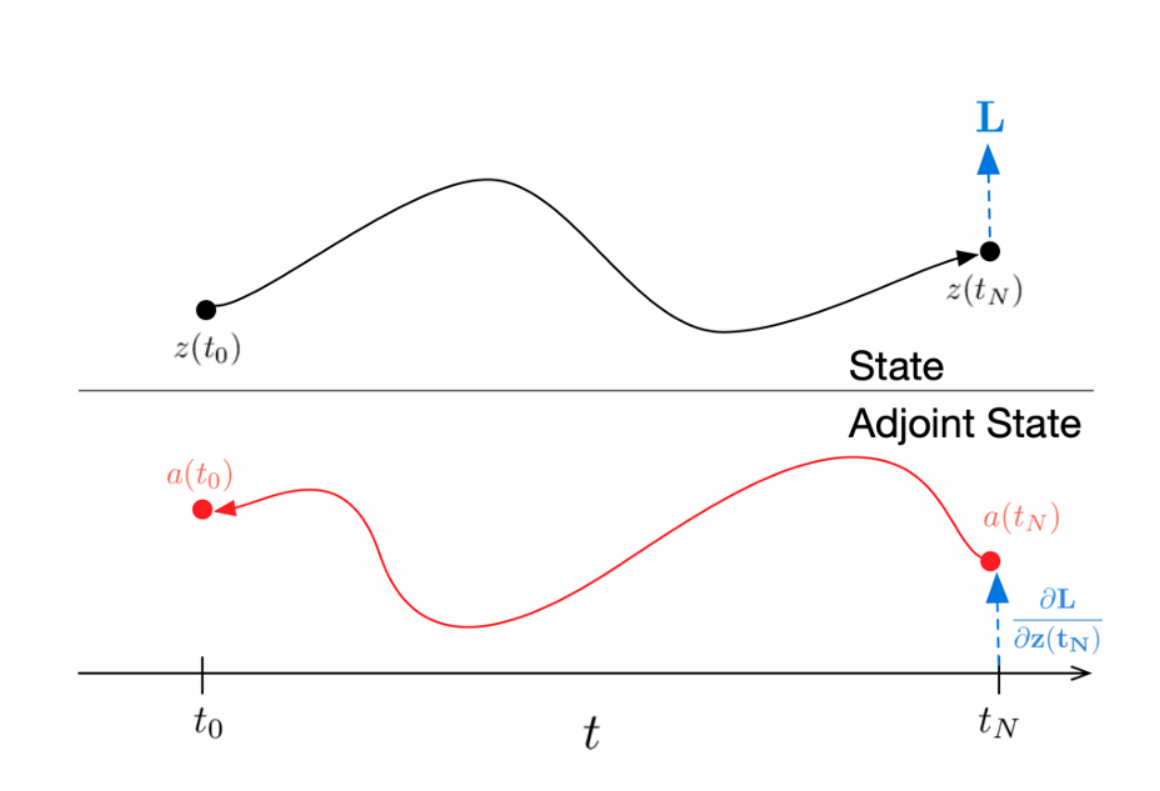

Figure 3.4 Rappresentazione dell'evoluzione dell'input definita da un neural ODE (sopra) e dell'evoluzione dell'aggiunta (sotto), integrata in direzione opposta. Il sistema comprendente entrambe le grandezze ed evoluzioni viene chiamato *augmented system*. Figura presa da [7].

Teorema 3.2. *Siano $y_0 \in \mathbb{R}^d$, $\theta \in \mathbb{R}^m$. Sia $f_\theta : [0,T] \times \mathbb{R}^d \to \mathbb{R}^d$ continua in t, lipschitziana in y e con derivata prima rispetto a y continua. Sia $y : [0,T]$ soluzione dell'equazione differenziale*

$$y(0) = y_0 \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t))$$

Sia $L = L(y(T))$ una funzione scalare.[3] *Allora $\frac{dL}{dy(y)} = a_y(t)$ e $\frac{dL}{d\theta} = a_\theta(0)$, dove $a_y : [0,T] \to \mathbb{R}^d$ e $a_\theta : [0,T] \to \mathbb{R}^m$ risolvono il seguente sistema di equazioni differenziali*

$$a_y(T) = \frac{\mathrm{d}L}{\mathrm{d}y(T)}, \qquad \frac{\mathrm{d}a_y}{\mathrm{d}t}(t) = -a_y(t)^T \frac{\partial f_\theta}{\partial y}(t, y(t)), \tag{3.4}$$

$$a_\theta(T) = 0, \qquad \frac{\mathrm{d}a_\theta}{\mathrm{d}t}(t) = -a_y(t)^T \frac{\partial f_\theta}{\partial \theta}(t, y(t)). \tag{3.5}$$

Dimostrazione. La dimostrazione completa e rigorosa di questo noto teorema si può trovare in varie fonti, indicate in bibliografia ([5], [12]). Qui diamo solamente un'idea della dimostrazione, derivando l'equazione per $a_y(t)$.

Dato che y è continua e la derivata di f_θ rispetto a y è continua, quindi la funzione $t \to \frac{\partial f_\theta}{\partial y}(t, y(t))$ è continua sul dominio $[0,T]$ e limitata da un numero $C > 0$.
Analogamente, dato $a \in \mathbb{R}^d$, la funzione $(t,a) \to -a^T \frac{\partial f_\theta}{\partial y}(t, y(t))$ è lipschitziana in a con costante di Lipschitz C indipendente da t. Dunque per il teorema di esistenza di Picard (3.1) la soluzione

[3] Nelle nostre applicazioni $L(y(T))$ è la *loss function*

a_y all'equazione 3.4 esiste ed è unica.

Dati due istanti temporali $s, t \in [0, T]$, con $s < t$, si ha che

$$y(t) = y(s) + \int_s^t f_\theta(u, y(u))\mathrm{d}u$$

e quindi

$$\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} = I_{d\times d} + \int_s^t \frac{\partial f_\theta}{\partial y}(u, y(u))\frac{\mathrm{d}y(u)}{\mathrm{d}y(s)}\mathrm{d}u$$

Adesso, differenziando rispetto a t la quantità $a_y(t)^T \frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}$ si ottiene

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\left(a_y(t)^T\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}\right) &= \frac{\mathrm{d}a_y}{\mathrm{d}t}(t)^T\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} + a_y(t)^T\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)}\right) \\
&= \frac{\mathrm{d}a_y}{\mathrm{d}t}(t)^T\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} + a_y(t)^T\frac{\partial f_\theta}{\partial y}(t, y(t))\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} \\
&= \left(\frac{\mathrm{d}a_y}{\mathrm{d}t}(t)^T + a_y(t)^T\frac{\partial f_\theta}{\partial y}(t, y(t))\right)\frac{\mathrm{d}y(t)}{\mathrm{d}y(s)} \\
&= 0
\end{aligned}$$

dove la seconda riga si ottiene differenziando rispetto a t l'espressione trovata precedentemente. Quindi

$$a_y(t) = a_y(t)^T\frac{\mathrm{d}y(t)}{\mathrm{d}y(t)} = a_y(T)^T\frac{\mathrm{d}y(T)}{\mathrm{d}y(t)} = \frac{\mathrm{d}L}{\mathrm{d}y(t)}$$

e in particolare $\frac{\mathrm{d}L}{\mathrm{d}y_0} = \frac{\mathrm{d}L}{\mathrm{d}y(0)} = a_y(0)$ □

Capitolo 4

Un applicazione: neural ODEs e farmacocinetica

Un'applicazione particolarmente interessante è rappresentata dall'utilizzo dei neural ODEs in farmacologia. In problemi di farmacocinetica, questi modelli hanno il potenziale di migliorare gli approcci attuali permettendo maggiore personalizzazione della terapia, aiutando per esempio i clinici a gestire il dosaggio di un farmaco in base alle caratteristiche di ogni paziente.

4.1 Farmacocinetica

I modelli farmacocinetici sono utilizzati per semplificare i processi che avvengono durante la somministrazione di un farmaco, che includono l'assorbimento, la distribuzione all'interno dell'organismo e l'eliminazione [11]. Un esempio sono i modelli compartimentali, nei quali la concentrazione del farmaco all'interno di un ipotetico compartimento, ad esempio un organo, è assunta uniforme. Ovviamente il corpo umano è molto più complesso di come descritto in questi modelli, che tuttavia risultano molto utili per studiare come agiscono e vengono smaltiti i farmaci.

Un modello comunemente utilizzato è quello a due compartimenti, dove sono presenti il compartimento centrale, ovvero il flusso sanguigno presente in tutto il corpo, e il compartimento tessutale, dove il farmaco ha il suo effetto. Infatti sappiamo che la concentrazione, intesa come concentrazione media all'interno di uno stesso compartimento, di un farmaco nel tessuto dove agisce segue un andamento biesponenziale diviso in due fasi (fig. 4.1):

- fase di distribuzione, che rappresenta l'iniziale declino della concentrazione del farmaco nel sangue a favore del tessuto, fino a raggiungere un equilibrio tra le due concentrazioni.
- fase di eliminazione, ovvero quando avviene lo smaltimento del farmaco da parte del corpo, che può avere luogo nel compartimento centrale, in quello tissutale o in entrambi. In ogni caso le due concentrazioni, quella nel sangue e quella tissutale, diminuiscono parallelamente.

Questo andamento può essere descitto da un sistema di equazioni differenziali, come indicato in fig. 4.1, che ha come paramentri costanti k_{12} e k_{21}, che rappresentano rispettivamente

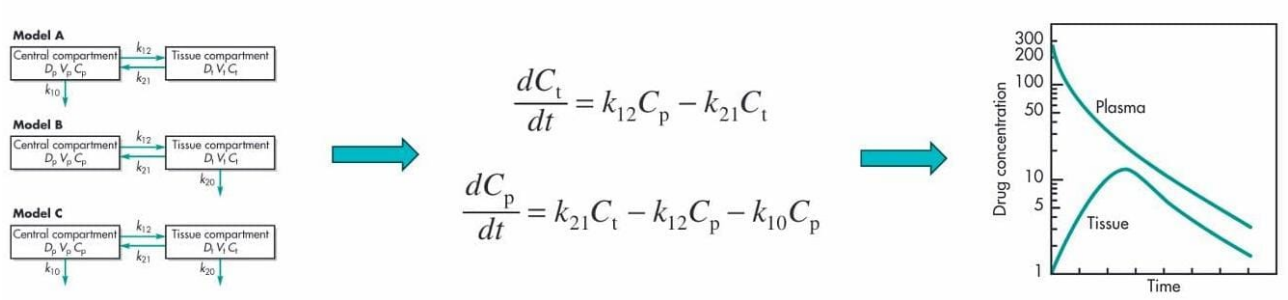

Figure 4.1 Andamento nel tempo delle concentrazioni tessutale C_t e sanguigna C_p di un farmaco nel corpo umano [11] .

il *rate* di trasferimento del farmaco dal compartimento centrale a quello tessutale e viceversa.

Tuttavia questo modello analitico non riesce a cogliere tutte le complessità del corpo umano e per ottenere un modello più preciso possiamo integrarlo con una *neural ODE*. Ci sono differenti opzioni per realizzare ciò. Ad esempio possiamo sostituire tutto il sistema con una *neural ODE*, avendo solamente come input le concentrazioni inziali, oppure possiamo integrarle parzialmente, trattando con un *neural network* parte delle variabili in input e attraverso un modello analitico l'output del *network* insieme al resto delle variabili di input (fig. 4.2).

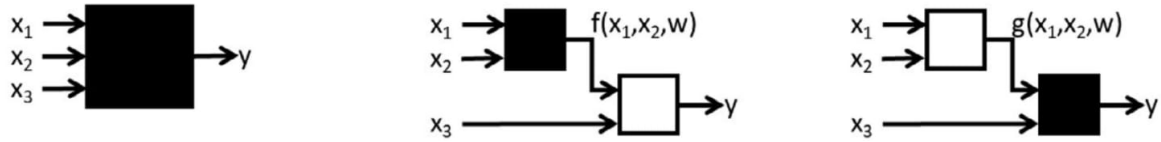

Figure 4.2 Rappresentazione dei possibili utilizzi di una *neural ODE* in questo caso. Nel primo caso abbiamo solo un *neural network*, rappresentato dal quadrato nero, che prende tutte le variabili in input, negli altri due casi si integra la *neural ODE* con un modello analitico, rappresentato dal quadrato bianco [1].

In questo modo possiamo riuscire a ottenere un modello più accurato, capace di adattarsi alle specificità di ogni paziente, espresse da variabili di input del *network*.

4.2 Precision Medicine

La stessa dose di un farmaco può portare a diverse concentrazioni e risposte in funzione della persona a cui viene somministrata. Trovare la dose giusta per ogni paziente è al centro della farmacometrica, che studia come varia la concentrazione di una determinata dose di farmaco nell'organismo e come esso reagisce a diverse concentrazioni del farmaco. Questi studi stanno diventando sempre più importanti nell'ambito della *precision medecine* [4], che richiede una modellizzazione individuale della malattia e della dinamica del farmaco, che è resa possibile grazie ai metodi di *machine learning*, che stanno diventando sempre più utilizzati in questo ambito.
Una compresione maggiore della relazione tra la quantità di un farmaco e l'effetto dello stesso

aiuterebbe molto nel trovare il miglior rapporto rischio beneficio in funzione di ogni paziente. Misurare direttamente la quantità del farmaco nei vari organi aiuterebbe in questo problema, tuttavia lo sviluppo di biosensori per misure in tempo reale all'interno del corpo umano è promettente, ma ancora lontano da un utilizzo su larga scala. Infatti questi dati possono soltanto essere stimati a partire da concentrazioni più accessibili, ad esempio le concentrazioni nel sangue, che li rendono però meno accurati. Tuttavia le concentrazioni sanguigne sono anche molto variabili ed presentano altre molecole che possono influire negativamente sulla precisione della misurazione.
Inoltre, questi dati possono variare significativamente anche da una persona a un'altra, dipendentemente dalle sue caratteristiche fisiche, ad esempio età, peso o altezza.

4.3 Un caso di studio

Per risolvere questi problemi sono creati dei modelli che, attraverso l'impiego di *neural ODEs*, riescono ad adattarsi alle specificità di ogni paziente e a trovare il percorso ottimale per la cura. Questo è quello che hanno fatto alcuni ricercatori , prendendo dati di un ciclo di cura effettuato somministrando *trastuzumab emtansine* su un campione composto da 675 persone [3]. I ricercatori hanno utilizzato 3 diversi metodi di *machine learning*, ovvero *LightGBM*, *LSTM* e *neural ODEs*, per predire nuovi regimi di dosaggio in base alle caratteristiche del paziente. I dati dei pazienti che sono stati presi come input dei modelli sono: etnia, continente di nascita, sesso, età, altezza e peso. I dati relativi relativi al ciclo di cura presi come input del modello sono: TFDS, ovvero il tempo in ore tra una dose e quella successiva, TIME, ovvero il tempo espresso in ore dall'inizio del trattamento, AMT, ovvero la quantità di farmaco espressa in *mg* e CYCL, ovvero quale regime di dosaggio si sta analizzando. In pratica, i modelli sapevano quando e in che quantità sono state effettuate le somministrazioni per ogni paziente.
I 3 modelli, dopo essere stati allenati su un *training set* con dati relativi a una parte dei pazienti, hanno predetto le concentrazioni del farmaco durante l'intero ciclo di cura per un paziente i cui dati non erano stati usati per l'allenamento (fig. 4.3). In questo caso, sono stati presi come input i dati delle concentrazioni del farmaco nel periodo tra la prima e la seconda somministrazione, corrispondenti ai punti blu in fig. 4.3. Infine questi 3 modelli sono stati confrontati con le previsioni di un modello analitico già esistente, chiamato *NLME* (*Non-Linear Mixed Effect*). Il *test* è stato effettuato sotto 3 condizioni di input diverse, chiamate in fig. 4.3 con A, B, C. Nel primo caso, le somministrazioni sono continuate normalmente fino alla fine della cura. Nel secondo caso, le somministrazioni del farmaco sono state artificialmente interrotte a metà del trattamente, ovvero i modelli non ricevevano più in input i dati relativi alle somministrazioni. Nel terzo caso, oltre ad interrompere i dati relativi alle somministrazioni, è stata fermata anche l'evoluzione temporale, ovvero i modelli non ricevevano più in input i dati sul tempo trascorso dall'inizio del trattamento.

Come possiamo osservare dai risultati, i modelli *LightGBM* e *LSTM* non hanno prodotto risultati soddisfacenti, infatti si notano discrepanze significative tra le loro previsioni e i dati sperimentali. Addirittura nel secondo caso, nonostante sia stato interrotta la somministrazio-

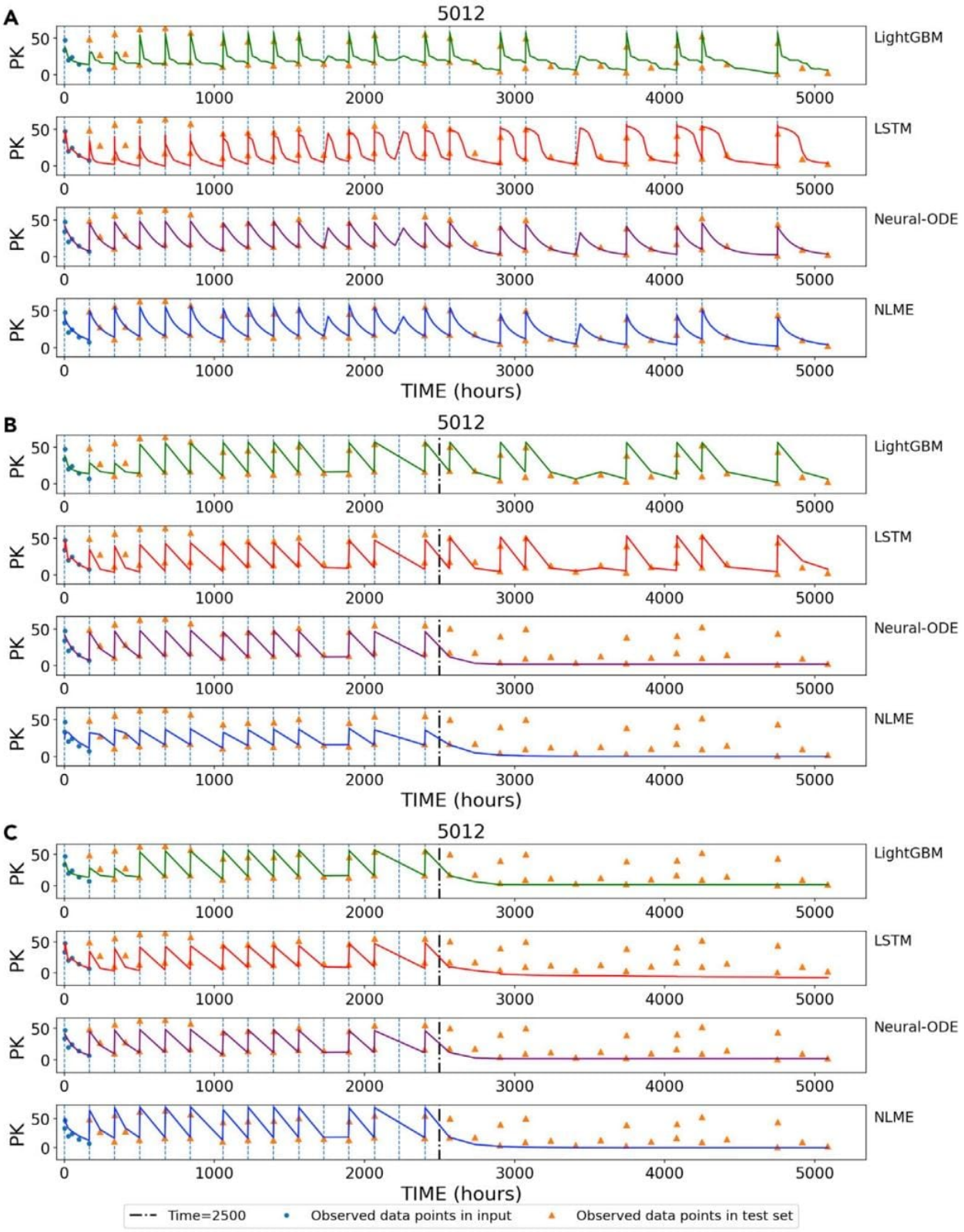

Figure 4.3 Confronto tra dati sperimentali e previsioni dei modelli nei 3 regimi di dosaggio diversi (A, B, C). 5012 si riferisce al numero del paziente i cui dati sono stati utilizzati per il *test* [3].

ne, essi continuano a predire lo stesso *pattern* appreso durante la prima metà del ciclo di cura, poiché non sono stati capaci di modellare gli eventi di somministrazione. Invece, la *neural ODE* e il modello *NLME* hanno saputo incorporare nella dinamica l'input riguardante l'evento di

somministrazione, infatti le concentrazioni previste vanno a zero molto rapidamente. I primi due modelli riescono a riprodurre questo comportamento solamente quando viene fermata anche l'evoluzione temporale, indicando che sono riusciti ad imparare i *pattern* solo perché riccorrono temporalmente e non perché riescono a ricollegarli alle somministrazioni.
Dunque i primi due modelli, *LightGBM* e *LSTM*, sono molto diversi come dinamica dal modello analitico, mentre la *neural ODE* effettua previsioni simili a livello cinetico.
Questo rende le *neural ODEs* uno strumento potenzialmente molto efficace in ambito farmacologico, dove non è solo importante prevedere i punti del *dataset*, ma soprattutto saperli interpolare bene con la rappresentazione continua, perché le grandezze cruciali per questo campo sono caratteristiche della curva, per esempio concentrazione massima o area sotto la curva. Inoltre, le *neural ODEs* sono promettenti anche per operazioni di ibridazione tra modelli di *machine learning* e modelli analitici, data la loro affinità concettuale.

Conclusioni

Per concludere, le neural ODEs offrono un modo nuovo e interessante di affrontare la modellizzazione di sistemi dinamici complessi. L'idea di combinare il deep learning con la struttura delle equazioni differenziali ha aperto la strada a una serie di applicazioni molto promettenti, in particolare quando si tratta di descrivere fenomeni continui e dinamici, come nel caso della precision medicine e della farmacocinetica.

Durante questo lavoro, abbiamo esplorato le caratteristiche principali delle neural ODEs, mettendo in evidenza la loro capacità di adattarsi a dati irregolari e di modellare processi complessi in modo più efficace rispetto ai metodi tradizionali. Un aspetto particolarmente interessante è la possibilità di vedere le neural ODEs come una versione continua dei residual networks, che permette di gestire in modo naturale le variazioni nel tempo e nelle dinamiche. Questo le rende adatta per affrontare problemi dove i dati non sono raccolti a intervalli regolari o dove i fenomeni evolvono in modo non prevedibile.

Le applicazioni pratiche che abbiamo visto, come la modellizzazione del modello Lotka-Volterra e l'analisi della dinamica dei farmaci, dimostrano quanto queste tecniche possano fare la differenza. Soprattutto nel campo della precision medicine, dove ogni paziente ha caratteristiche uniche, le neural ODEs possono essere utilizzate per creare trattamenti personalizzati e ottimizzare il dosaggio dei farmaci. Questo significa non solo migliorare l'efficacia delle terapie, ma anche ridurre gli effetti collaterali e aumentare la sicurezza dei trattamenti.

Naturalmente, ci sono ancora alcune sfide da affrontare. L'allenamento di questi modelli richiede molta potenza di calcolo e i risultati dipendono molto dai dati disponibili, che devono essere numerosi per ottenere previsioni affidabili. Per questo motivo, è importante continuare a lavorare su tecniche di ottimizzazione più efficienti e su metodi che possano garantire una buona precisione anche quando i dati sono limitati.

Guardando al futuro, le possibilità delle neural ODEs sono molto ampie. La loro capacità di combinare la teoria con il machine learning potrebbe aiutare a sviluppare modelli più precisi in tanti campi, dalla fisica alla biologia, fino all'economia. Integrare le neural ODEs con modelli matematici tradizionali, come abbiamo visto, potrebbe permettere di ottenere soluzioni che riescono a descrivere meglio la realtà e ad adattarsi alle esigenze dei diversi contesti.

In sintesi, questo lavoro rappresenta un primo passo per comprendere e utilizzare le neural ODEs, evidenziandone le potenzialità. Proseguire con la ricerca in questo ambito potrà portare a sviluppare tecniche di allenamento migliori e architetture più pratiche, rendendo le neural ODEs uno strumento ancora più utile per la ricerca e le applicazioni pratiche.

Bibliografia

[1] Artur M. Schweidtmann et al. "A review and perspective on hybrid modeling methodologies". In: *Elsevier* (2024), pp. 1–5.

[2] Facundo Sapienza et al. "Differentiable Programming for Differential Equations: A Review". In: *Science* (2024), pp. 18–22.

[3] James Lu et al. "Neural-ODE for pharmacokinetics modeling and its advantage to alternative machine learning models in predicting new dosing regimens". In: *iScience* (2021), pp. 1–8.

[4] Kamilė Stankevičiūtė1 et al. "Bridging the Worlds of Pharmacometrics and Machine Learning". In: *Clin Pharmacokinet* (2023), pp. 1–5.

[5] Leon Lettermann et al. "Tutorial: a beginner's guide to building a representative model of dynamical systems using the adjoint method". In: *communications physics* (2024), pp. 3–6.

[6] Stefano Massaroli et al. "Dissecting Neural ODEs". In: *ArXiv* abs/2002.08071 (2020). URL: `https://api.semanticscholar.org/CorpusID:211171558`.

[7] Tian Qi Chen et al. "Neural Ordinary Differential Equations". In: *CoRR* abs/1806.07366 (2018). arXiv: `1806.07366`. URL: `http://arxiv.org/abs/1806.07366`.

[8] Zico Kolter et al. *Implicit functions and automatic differentiation*. https://implicit-layers-tutorial.org/implicit$_f$*unctions*/, 2023.

[9] Romain Bernard. *IA pour la Physique : introduction aux méthodes d'apprentissage statistique*. Sorbonne University. Paris, France, 2023.

[10] Patrick Kidger. "On Neural Differential Equations". Tesi di dott. University of Oxford, 2021.

[11] Andrew B.C. Yu Leon Shargel. *Applied Biopharmaceutics and Pharmacokinetics*. McGraw-Hill Education, 2016, pp. 97–105.

[12] Pau Baldillou Salse. "An Introduction To Neural Differential Equations". Tesi di laurea mag. Universitat de Barcelona, 2024, pp. 31–33.