UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

"Equazioni differenziali ordinarie applicate alle reti neurali"

Relatore: Prof. Alessandro Chiuso

Laureando: Stefano Califano

ANNO ACCADEMICO 2021 – 2022

Data di laurea 25-11-2022

Abstract

Le reti neurali sono da alcuni decenni un argomento di grande interesse nel machine learning per la loro capacità di approssimare qualsiasi funzione. Nel tempo sono stati proposti vari modelli con diversi scopi, raggiungendo così numerosi campi d'applicazione e discipline di interesse. Solo recentemente le conoscenze sulle reti neurali si sono unite alla matematica delle equazioni differenziali ordinarie (ODE).

L'obiettivo di questo elaborato è quello di presentare il modello di rete neurale introdotto nel 2019 da Ricky T. Q. Chen et al. denominato *Neural ODE*, analizzandone i vantaggi, le limitazioni e alcune sue applicazioni. Si intende, inoltre, esaminare l'impatto e l' evoluzione che tale modello ha avuto dalla sua introduzione ad oggi.

Indice

Capitolo 1

Introduzione

1.1 Apprendimento

Il termine *apprendimento*, come il termine *intelligenza*, è talmente ampio e complesso che è difficile dare una definizione rigorosa. L'enciclopedia Treccani [1] fornisce la seguente definizione: "*Nella ricerca sia psicologica sia etologica, acquisizione persistente di modificazioni del comportamento, dal semplice condizionamento di riflessi primari fino a forme complesse di organizzazione delle informazioni, determinate dall'esperienza del soggetto, piuttosto che da un controllo genetico. [...]*".

É fuori dubbio che le metodologie che governano l'apprendimento nel mondo animale e umano hanno ispirato le ricerche e gli studi nell'ambito del *machine learning* fondato sull'assunto che una macchina impara quando cambia la propria struttura grazie a input esterni allo scopo di migliorarne le prestazioni nel tempo. [2]

Traducendo direttamente la definizione proposta da Tom Mitchell [3] si può esprimere l'apprendimento nel modo seguente:

"*Un algoritmo apprende dall'esperienza E con riferimento ad alcune classi di compiti T e con misurazione della performance P, se con le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E.*"

Quindi l'apprendimento è il processo di conversione dell'esperienza in competenza e conoscenza. Si può schematizzare il processo di apprendimento di un algoritmo come riportato in figura 1.1. Le varie componenti di tale processo verranno analizzate nei capitoli seguenti, formalizzando matematicamente il concetto di apprendimento.

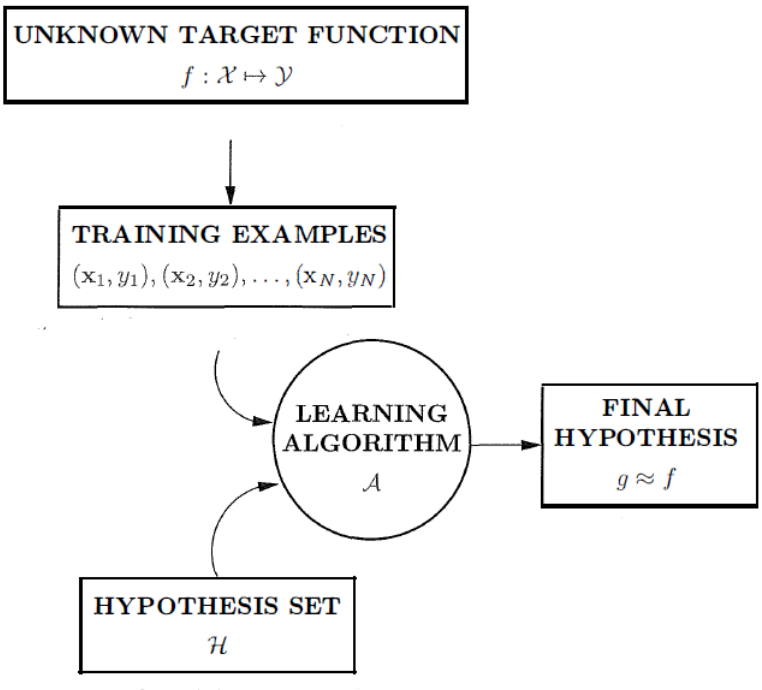


Figura 1.1: Schema del processo di apprendimento [4]

Nella figura si indica viene indicato con g quella che nel testo appare come h; inoltre nel riquadro intitolato *Training examples* (che corrisponde al *training set*) sono rappresentate le coppie $(\mathbf{x} \in X, y \in Y)$. Questo perché in alcuni problemi di apprendimento i pattern del training set sono accompagnati dal valore dell'output corretto (nei capitoli successivi si chiarirà il concetto).

1.1.1 Componenti dell'apprendimento

Le componenti principali del modello matematico di un problema di apprendimento sono le seguenti [2] [4]:

- ***Prediction rule*** : l'obiettivo di un algoritmo di apprendimento A è quello di produrre una regola, ovvero una funzione $h : X \to Y$ che meglio approssimi una funzione target ideale $f : X \to Y$. É importante precisare che la funzione f è sconosciuta.

- ***Input*** : insieme X di *pattern* $\mathbf{x} = \{x_1, x_2, ..., x_d\}$ di input, ognuno dei quali ha d componenti chiamate *features*. Il valore delle features può essere principalmente di tre tipi: numeri reali, numeri interi e valori categorici (ovvero valori non numerici presi da un insieme finito, come stringhe oppure variabili booleane). Un pattern, le cui features possono essere di tipi diversi, può rappresentare anche un oggetto molto complesso come un'immagine o un grafo. L'insieme dei pattern che viene utilizzato per addestrare l'algoritmo prende il nome di *training set* D.
  Dal punto di vista statistico si assume che tale insieme derivi da una qualche distribuzione di probabilità *P*; è importante precisare che tale distribuzione è sconosciuta durante il processo di apprendimento.

- ***Output*** : l'output dell'algoritmo $y = h(\mathbf{x})$ (con $y \in Y$ e $\mathbf{x} \in X$) può essere di due tipi: se è un valore numerico, la funzione h prende il nome di *fuction estimator* e $h(\mathbf{x})$ è il valore stimato; alternativamente $h(\mathbf{x})$ può essere un valore categorico e in tal caso prende il nome di etichetta o classe, mentre h è definito classificatore.

Generalmente la funzione h viene scelta da un insieme H di possibili funzioni; tale insieme viene chiamato *hypothesis class*. Ad esempio H può essere l'insieme di tutte le possibili formule lineari: l'algoritmo A sceglierà la funzione lineare con la migliore corrispondenza rispetto ai dati di training D forniti in input. La funzione h (non la funzione f che si ricorda essere sconosciuta) potrà essere utilizzata per definire il valore $y \in Y$ di un nuovo pattern $\mathbf{x} \in X$ (e in particolare $\mathbf{x} \notin D$); questo processo prende il nome di generalizzazione.
Per un dato problema di apprendimento, la funzione target f e il training set D sono determinati dal problema stesso; invece il particolare algoritmo di apprendimento A e la classe di ipotesi H devono essere scelti. Tali scelte costituiscono il modello di apprendimento. [4]

1.1.2 Paradigmi di apprendimento

Si vogliono ora introdurre i diversi paradigmi di apprendimento che sono stati sviluppati per affrontare diversi problemi di apprendimento. [4]

- ***supervised learning*** : l'apprendimento è supervisionato quando per ogni pattern **x** nel training set D è specificato qual è l'output esatto y (*labeled data*).

- ***unsupervised learning*** : l'apprendimento è definito non supervisionato quando i pattern del training set D non contengono informazioni sull'output (*unlabeled data*). Nonostante la mancanza di tali informazioni è comunque possibile creare nuova conoscenza: si può interpretare questo paradigma come la ricerca di strutture "interessanti" interne ai dati; si parla infatti di *knowledge discovery*.

- ***reinforcement learning*** : si parla di reinforcement learning quando l'algoritmo (nello specifico si usa il termine agente) deve compiere delle scelte in un determinato ambiente per raggiungere un obiettivo. Ogni volta che l'agente opera una scelta lo si può ricompensare o penalizzare in base a quanto la scelta fatta sia concorde o meno rispetto all'obiettivo finale. [3]

1.1.3 Problemi di apprendimento

In base al paradigma di apprendimento utilizzato e all'insieme Y, si possono definire quattro tipi di problemi di apprendimento considerati canonici, definiti in figura 1.2. In particolare, la classificazione e la regressione ricadono nel caso di apprendimento supervisionato; invece il problema di *clustering* e della riduzione di dimensionalità appartengono al caso non supervisionato. Segue una breve descrizione di questi problemi: [5]

- ***Classification*** : lo scopo della classificazione è quello classificare un pattern di input **x** con (almeno) una delle possibili C classi $y \in Y$. Di fatto si vuole approssimare una funzione f tale che $y = f(\mathbf{x})$ partendo dai pattern del training set di cui si conoscono le classi di appartenenza. Tale funzione sarà utile per fare predizioni su nuovi pattern sconosciuti. Nel caso in cui $C = 2$ si parla di *binary classification*; se $C > 2$ si parla di *multiclass classification*. Se le classi non sono mutualmente esclusive, cioè se è possibile assegnare più classi a uno stesso pattern, allora il problema prende il nome di *multi-label classification*.

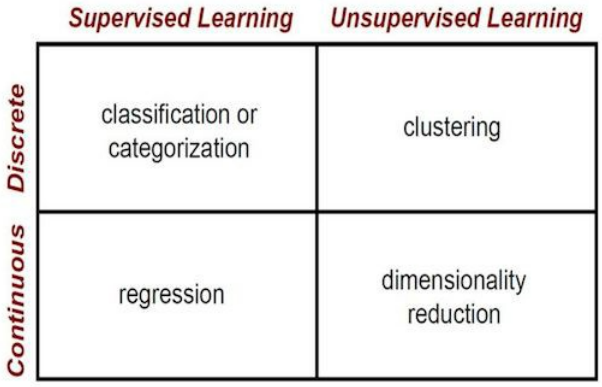


Figura 1.2: Tabella dei principali problemi di apprendimento

- ***Regression*** : il caso della regressione è molto simile al problema della classificazione con l'importante differenza che $Y \subseteq \mathbb{R}$. Ad esempio dai dati del training set $(\mathbf{x}, y) \in D$ si può voler trovare la funzione lineare che costituisce il *fit* migliore tra i dati di X e di Y.

- ***Clustering*** : il problema del clustering consiste nel suddividere i pattern disponibili nel training set D in più gruppi in modo che pattern simili siano nello stesso cluster mentre pattern dissimili si trovino in cluster diversi. Essendo un problema non supervisionato il numero delle classi e le classi stesse sono sconosciute. [6]

- ***Dimensionality reduction*** : nel momento in cui bisogna gestire pattern con un elevato numero di features è spesso utile ricorrere alla riduzione della dimensionalità, proiettando i dati in uno spazio con dimensionalità minore. Concentrando l'analisi dei dati sulle features principali e più significative, si ottengono miglioramenti del modello predittivo. Inoltre la riduzione della dimensionalità permette la visualizzazione grafica di dati ad alta dimensionalità.

1.1.4 Esempio di modello di apprendimento: perceptron

Si vuole illustrare ora un semplice modello di apprendimento utile anche per introdurre i concetti fondamentali delle reti neurali che verranno approfondite nei capitoli successivi. [4]
Viene dato $X = \mathbb{R}^d$ lo spazio dei pattern di input, dove $\mathbb{R}^d$ è lo spazio euclideo *d*-dimensionale, e $Y = \{+1, -1\}$ i possibili output. Il problema posto corrisponde quindi ad una classificazione binaria. Volendo fornire l'esempio concreto di una persona che richiede un prestito ad una

banca, ogni pattern **x** in input potrebbe essere un vettore contente dati relativi allo stipendio, al tipo di lavoro, al reddito e altre informazioni di carattere economico. I due output possibili corrisponderebbero rispettivamente all'approvazione e al rifiuto del prestito. Inoltre, è fornito un training set D di pattern con il relativo output $(\mathbf{x}, y)$, ovvero esempi passati di persone a cui è stato concesso o meno il prestito in base ai propri parametri.
Il processo di addestramento consiste nell'assegnare un coefficiente reale, chiamato peso, alle varie coordinate di **x** rispetto alla loro importanza nella decisione da prendere (concedere o meno il prestito). La determinazione dei pesi deve basarsi sull'analisi dei dati del *training set*.
La forma funzionale di $h(\mathbf{x})$ consiste nella combinazione lineare delle coordinate con i relativi pesi, formando un punteggio che è confrontato con una soglia prefissata: se tale punteggio supera la soglia il prestito viene concesso, in caso contrario è respinto.

$$\sum_{i=1}^{d} w_i x_i > soglia \Rightarrow \text{prestito concesso}$$

$$\sum_{i=1}^{d} w_i x_i < soglia \Rightarrow \text{prestito non concesso}$$

In maniera più compatta:

$$h(\mathbf{x}) = sign\left(\left(\sum_{i=1}^{d} w_i x_i\right) + b\right)$$

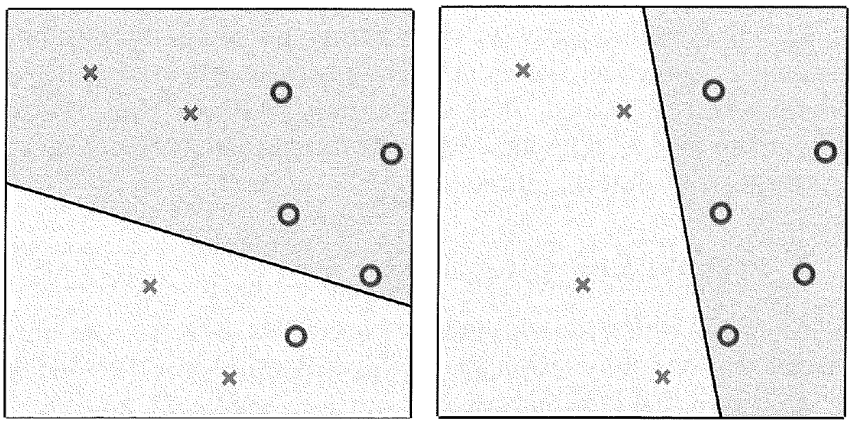
dove $x_1, x_2, ..., x_d$ sono le features del pattern **x**; $w_1, w_2, ..., w_d$ sono i pesi relativi alle feature dei pattern; il valore b è chiamato *bias* e corrisponde di fatto allo soglia di cui sopra. Il prestito è quindi concesso se $h(\mathbf{x}) = +1$ ovvero se $sign(\cdot) > 0$; altrimenti il prestito non viene concesso. Questa classe di ipotesi H viene denominata *perceptron*.

Lo scopo dell'algoritmo di apprendimento sarà quindi quello di determinare i valori ottimi dei singoli pesi e definire la funzione $h \in H$ con cui decidere se concedere o meno futuri prestiti.

In figura 1.3 è riportato un esempio di apprendimento di un perceptron nel caso in cui $d = 2$: lo spazio dei pattern viene diviso in due regioni che corrispondono ai due possibili output dell'algoritmo. La retta di separazione $w_1 x_1 + w_2 x_2 + b = 0$ è determinata dai diversi valori che possono assumere i pesi w_1, w_2 e b.

Per semplificare la notazione, il termine b può essere interpretato come un ulteriore peso $w_0 = b$. É possibile inoltre modificare anche il vettore **x** aggiungendo la componente $x_0 = 1$. Si può quindi scrivere:

$$h(\mathbf{x}) = sign\left(\mathbf{w}^T \mathbf{x}\right) = sign\left(\sum_{i=0}^{d} w_i x_i\right)$$



(a) Pattern pre-addestramento [4] (b) Pattern post-addestramento

Figura 1.3: Classificazione di un perceptron [4]

In figura sono mostrati dei pattern di training: $\times$ corrisponde a -1 (prestito non concesso) mentre $\circ$ corrisponde a +1 (prestito concesso). Si può notare come, in seguito all'addestramento dell'algoritmo, si sia determinata una retta che separa tali pattern. Un nuovo pattern verrà classificato in base alla regione in cui ricade.

1.2 Reti Neurali Artificiali (ANN)

Una rete neurale artificiale (*Artificial Neural Network* ANN o semplicemente *neural network*) è un modello computazionale ispirato alla struttura della rete neurale del cervello umano. [3]

Le reti neurali sono impiegate in diversi ambiti di studio e della ricerca: in ingegneria e *data science* sono usate come modelli per l'analisi statistica di dati per la regressione non lineare e per problemi di clustering; in neuroscienze e in psicologia costituiscono delle valide astrazioni del cervello umano e animale aiutando a comprendere e studiare l'elaborazione delle informazioni nelle reti nervose biologiche; infine, in matematica e fisica, le reti neurali sono degli strumenti per lo studio dei sistemi dinamici non lineari, della meccanica statistica e dell'automazione. [7]

In generale una rete neurale è una interconnessione di semplici elementi di elaborazione, detti nodi, il cui funzionamento è ispirato a quello dei neuroni umani. L'abilità di elaborazione della rete risiede nei pesi delle connessioni inter-unità, ottenuti attraverso un processo di apprendimento a partire da un training set. [7]

In sintesi, l'aggiornamento dei pesi delle connessioni (e quindi l'apprendimento della rete) avviene attraverso un processo iterativo: la rete elabora un pattern del training set; al termine dell'elaborazione, in base al confronto tra l'output della rete e quello desiderato, avviene l'ag-

giornamento dei pesi; si passa quindi al pattern successivo. La logica con cui vengono aggiornati i pesi è definita *learning rule*. L'intero processo, compresa la scelta dell'ordine con cui elaborare i pattern, della *learning rule*, etc., costituisce il *training algorithm*. [7]

Una rete neurale può essere descritta da un grafo $G = (V, E)$, dove l'insieme dei vertici V corrisponde ai neuroni della rete, mentre i rami E sono le connessioni tra tali neuroni. [6] Ad ogni ramo del grafo è associato un numero reale denominato peso. Due nodi sono connessi se esiste un ramo del grafo che li collega.

L'input di un nodo n è definito *network input* di n, ed è ottenuto calcolando la somma pesata di tutti gli output dei neuroni connessi al nodo n; tale somma prende il nome di *propagation function*. L'output del nodo n è dato dalla successiva elaborazione del network input attraverso la *activation function* σ. [8] [6]

Esistono tre principali varianti della *activation function*:

- funzione segno : $\sigma(a) = sign(a)$
- funzione soglia (*threshold*) : $\sigma(a) = \mathbb{1}_{[a>0]}$
- funzione sigmoide : $\sigma(a) = \frac{1}{1+exp(-a)}$

Il neurone costruito con la funzione soglia corrisponde al perceptron, visto precedentemente. Lo sviluppo di tale modello risale agli anni '50 e '60 ad opera dello scienziato Frank Rosenblatt, ispirato dai lavori precedenti di Warran McCulloch e Walter Pitts. Attualmente è più comune utilizzare modelli non lineari come quelli che implementano la funzione sigmoide. [9] [10]

In base alle caratteristiche delle connesioni del grafo possono essere definite alcune tipologie o *design* di reti:

- se il grafo è aciclico allora si parla di *feedforward neural network* (FNN)
- se il grafo contiene cicli si parla di *recurrent neural network* (RNN)
- se sono presenti *shortcut connection* allora si parla di *residual neural network* (ResNet)

1.2.1 Cost Function

L'obiettivo di una rete neurale è quello di trovare il valore dei pesi dei rami ottimale in modo che, per ogni pattern di training $(\mathbf{x}, y)$, venga restituito un output $h(\mathbf{x})$ che meglio approssimi

il valore target y. Per quantificare quanto è affidabile l'approssimazione, si definisce una *cost function* (a volte denominata anche *loss* o *objective function*):

$$C(w) = \frac{1}{2N} \sum_{\forall \mathbf{x} \in D} \|y - h(\mathbf{x})\|^2 \tag{1.1}$$

Dove w rappresenta l'insieme di tutti i pesi della rete, N è la dimensione del training set D e la notazione $\|v\|$ indica la lunghezza del vettore v. Per semplificare la notazione non è stata indicata la dipendenza dell'output delle rete $h(\mathbf{x})$ dai pesi w. La cost function sopra definita prende il nome di *quadratic cost function* o *mean squared error* (MSE), che tuttavia non è l'unica scelta possibile.

Si può notare che la funzione $C(w)$, e di conseguenza la sommatoria, è non-negativa. Inoltre, il valore $C(w)$ tende a zero, cioè $C(w) \approx 0$, quando $h(\mathbf{x})$ si avvicina al valore target y per ogni pattern del training set.

L'obiettivo dell'algoritmo di training sarà perciò quello di minimizzare la funzione $C(w)$ trovando i valori ottimi di w. L'algoritmo che verrà utilizzato è conosciuto come *gradient descent*. [9]

1.2.2 Gradient descent

Questo paragrafo non è strettamente legato al contesto delle reti neurali: infatti, per semplicità, è possibile dimenticare la forma specifica della cost function e immaginare una qualsiasi funzione in più variabili $C(v)$. Per aiutare l'intuizione e senza perdita di generalità, si può ridurre il problema a una funzione in due variabili, cioè con $v = (v_1, v_2)$.

Si vuole trovare il minimo globale della funzione C. Un primo approccio potrebbe essere quello di trovare il minimo analiticamente. Ma quando la funzione dipende da molte variabili (e nel caso delle reti neurali bisogna gestire anche migliaia di variabili) questo metodo diventa impraticabile.

L'intuizione è quella di pensare alla funzione come ad una valle e ad una pallina che scende lungo il versante, come mostrato in figura 1.4. Se la pallina si sposta di Δv_1 nella direzione di v_1 e di Δv_2 nella direzione di v_2, allora la funzione C cambia come:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \tag{1.2}$$

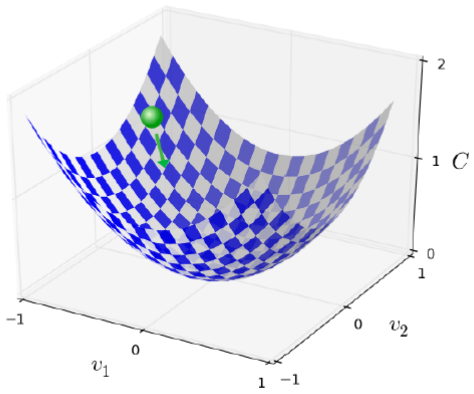


Figura 1.4: *Gradient descent* per una funzione in due variabili [9]

Se si definisce il vettore Δv come $\Delta v = (\Delta v_1, \Delta v_2)^T$ e il gradiente di C come il vettore delle derivate parziali $\nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$, allora:

$$\Delta C \approx \nabla C \cdot \Delta v \tag{1.3}$$

É possibile scegliere Δv in modo che ΔC sia negativo. In particolare, si sceglie

$$\Delta v = -\eta \nabla C \tag{1.4}$$

dove η è un numero reale positivo arbitrariamente piccolo, chiamato *learning rate*. Allora dalla equazione (1.3):

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 \tag{1.5}$$

Siccome $\|\nabla C\|^2 \geq 0$ è garantito che $\Delta C \leq 0$, ovvero che C diminuirà sempre. Il movimento della pallina, detto *update rule*, può essere definito come:

$$v \rightarrow v' = v - \eta \nabla C \tag{1.6}$$

Di fatto, l'*update rule* definisce l'algoritmo per gradient descent: è la regola con cui ripetutamente si cambia la posizione di v per trovare il minimo della funzione C.
La scelta del valore del *learning rate* è importante: deve essere sufficientemente piccolo in modo che l'equazione (1.3) sia una buona approssimazione ma grande abbastanza per evitare che l'algoritmo sia troppo lento.

La funzione C può essere generalizzata a m variabili: in questo caso $\Delta v = (\Delta v_1, ..., \Delta v_m)^T$ e $\nabla C = (\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_m})^T$; le formule viste precedentemente rimangono invariate.

Nel contesto delle reti neurali, la funzione da minimizzare è la cost function che dipende dai pesi della rete. Perciò l'*update rule* diventa:

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \tag{1.7}$$

Esiste un problema pratico che è importante menzionare: l'equazione della cost function (1.1) è nella forma $C = \frac{1}{n} \sum_{\mathbf{x}} C_{\mathbf{x}}$, ovvero è una media dei singoli costi $C_{\mathbf{x}} = \frac{1}{2}\|y - h(\mathbf{x})\|$ associati ai pattern del training set. Nella pratica significa calcolare il gradiente $\nabla C_{\mathbf{x}}$ di ogni input $\mathbf{x}$. Quando il numero dei pattern di input è molto grande questo può portare a un rallentamento eccessivo del processo di apprendimento.

Per velocizzare il processo si utilizza una variante dell'algoritmo appena visto, conosciuto come *stochastic gradient descent* (SGD). L'idea è semplicemente quella di calcolare il gradiente della cost function per un numero ridotto m di pattern, generalmente scelti in modo randomico. Ci si riferisce ai pattern scelti come *mini-batch*. Il valore m deve essere scelto in modo che la media calcolata sul *mini-batch* sia una buona approssimazione del valore che si calcolerebbe considerando tutti i pattern. In realtà non è necessario che la stima sia perfetta: è sufficiente che la pallina si muova in una direzione tale da far diminuire il valore di C, non serve che sia l'esatta direzione del gradiente. [9]

1.2.3 Reti feedforward

In una rete feedforward, ogni ramo del grafo collega l'output di un neurone con l'input di un altro. L'input totale di un neurone è ottenuto dalla combinazione lineare di tutti gli output dei neuroni collegati ad esso, moltiplicati per i rispettivi pesi determinati dalla funzione w.

Un rete può essere organizzata in *layers*: ovvero, si possono suddividere i neuroni in insiemi non vuoti e disgiunti $V = \bigcup_{t=0}^{T} V_t$ tali che ogni ramo in E connette un nodo in V_{t-1} con un nodo V_t. Ci si riferisce a T come la profondità (*depth*) della rete; V_0 è definito *input layer*, V_T è definito *output layer*, mentre i layer $V_1, ..., V_{T-1}$ sono definiti *hidden*. Quando una rete neurale è composta da più *hidden layer* allora prende la denominazione di rete neurale profonda (*deep neural network* DNN) [11].

Si denota con n_k^t il k-esimo neurone del layer t; mentre con o_k^t l'output di n_k^t. Il peso del ramo che connette il neurone n_k^{t-1} con il neurone n_j^t è indicato con $w_{j,k}^t$.
Conoscendo il valore dell'output dei neuroni del layer $t-1$ si può calcolare l'output dei neuroni

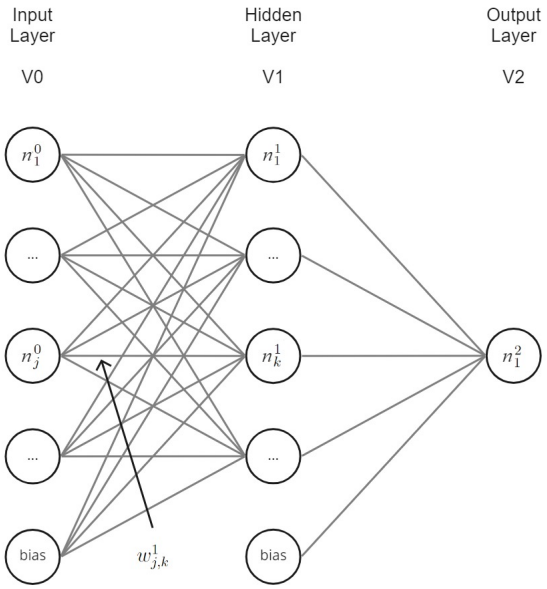


Figura 1.5: Rete feedforward

del layer t. In particolare, per un nodo $n_j^t \in V_t$, l'input è dato da:

$$z_j^t = \sum_{\forall k} w_{j,k}^t o_k^{t-1} \tag{1.8}$$

mentre l'output:

$$o_j^t = \sigma(z_j^t) \tag{1.9}$$

In figura 1.5 è rappresentata una rete neurale feedforward con tre livelli di cui uno hidden. [6]

1.2.4 Backpropagation

L'algoritmo di *backpropagation* è uno dei metodi che si utilizza per calcolare il gradiente della cost function nell'algoritmo del gradient descent. Si vuole quindi ora calcolare la derivata parziale $\frac{\partial C}{\partial w}$, ovvero come la cost function varia rispetto ai pesi della rete.

Per poter applicare l'algoritmo di backpropagation è necessario che vengano fatte due assunzioni sulla cost function: la prima è che deve poter essere scrivibile come media $C = \frac{1}{n}\sum_{\mathbf{x}} C_{\mathbf{x}}$ di cost function $C_{\mathbf{x}}$ riferite ai singoli pattern **x** del training set; in questo modo si può considerare la cost function di uno solo dei pattern di input e alleggerire la notazione omettendo il pedice **x**; si scriverà quindi $C_{\mathbf{x}}$ come C. La seconda assunzione è che la cost function deve poter scritta

come funzione degli output della rete. Nel caso particolare della *quadratic cost function*:

$$C = \frac{1}{2}\|y - o^T\|^2 = \frac{1}{2}\sum_j (y_j - o_j^T)^2 \tag{1.10}$$

Per calcolare la variazione della cost function rispetto a uno dei pesi della rete, ovvero $\partial C/\partial w_{j,k}^t$, è preferibile introdurre una quantità intermediaria δ_j^t che rappresenta una misura dell'errore nel neurone v_j^t. Si definisce l'errore δ_j^t del neurone j-esimo del livello t come:

$$\delta_j^t = \frac{\partial C}{\partial z_j^t} \tag{1.11}$$

L'algoritmo di backpropagation si basa su tre equazioni fondamentali. Nel loro insieme consentono di calcolare δ_j^t per arrivare alla quantità di interesse $\partial C/\partial w_{j,k}^t$. Si osservi che non è specificato quale *activation function* utilizzare. Le equazioni sono le seguenti:

- **Equazione per l'errore nell'output layer** :

$$\delta_j^T = \frac{\partial C}{\partial o_j^T}\sigma'(z_j^T) \tag{1.12}$$

Il termine $\partial C/\partial o_j^T$ misura quanto velocemente la cost function varia rispetto alla *activacion function* del j-esimo neurone del layer di output. Il termine $\sigma'(z_j^T)$ misura, invece, quanto velocemente la funzione di attivazione varia rispetto a z_j^T. *dimostrazione* : applicando la regola della catena all'equazione (1.11), si può esprimere la derivata parziale rispetto alle *activation function* dei neuroni dell'output layer:

$$\delta_j^T = \sum_k \frac{\partial C}{\partial o_k^T}\frac{\partial o_k^T}{\partial z_j^T} \tag{1.13}$$

Dove la sommatoria è su tutti i neuroni del livello T. Il valore dell'*activation function* o_k^T dipende solamente dal *network input* z_j^T quando $k = j$. Quindi il termine della sommatoria è pari a zero quando $k \neq j$.

$$\delta_j^T = \frac{\partial C}{\partial o_j^T}\frac{\partial o_j^T}{\partial z_j^T} \tag{1.14}$$

Ricordando che $o_j^T = \sigma(z_j^T)$ e osservando che il secondo termine a destra dell'uguaglianza corrisponde a $\sigma'(z_j^T)$ si ottiene:

$$\delta_j^T = \frac{\partial C}{\partial o_j^T}\sigma'(z_j^T) \tag{1.15}$$

- **Equazione per l'errore δ_j^t rispetto all'errore del layer successivo δ_j^{t+1} :**

$$\delta_j^t = \sum_k w_{k,j}^{t+1} \delta_k^{t+1} \sigma'(z_j^t) \tag{1.16}$$

Si può pensare intuitivamente a questa equazione come alla propagazione "all'indietro" dell'errore. Ovvero, supponendo di conoscere l'errore δ^{t+1} del $t+1$-esimo layer, si risale ad una misura dell'errore del t-esimo layer. Combinando (1.12) con (1.16) è possibile calcolare l'errore δ per ogni livello della rete.

dimostrazione : si vuole riscrivere $\delta_j^t = \partial C / \partial z_j^t$ in termini di $\delta_k^{t+1} = \partial C / \partial z_k^{t+1}$. Applicando la regola della catena all'equazione (1.11):

$$\delta_j^t = \sum_k \frac{\partial C}{\partial z_k^{t+1}} \frac{\partial z_k^{t+1}}{\partial z_j^t} \tag{1.17}$$

Sostituendo la definizione di $\delta_k^{t+1} = \partial C / \partial z_k^{t+1}$:

$$\delta_j^t = \sum_k \delta_k^{t+1} \frac{\partial z_k^{t+1}}{\partial z_j^t} \tag{1.18}$$

Ricordando che

$$z_k^{t+1} = \sum_h w_{k,h}^{t+1} o_h^t = \sum_h w_{k,h}^{t+1} \sigma(z_h^t) \tag{1.19}$$

e differenziando, si ottiene

$$\begin{aligned} \frac{\partial z_k^{t+1}}{\partial z_j^t} &= \frac{\partial}{\partial z_j^t} \left(\sum_h w_{k,h}^{t+1} \sigma(z_h^t) \right) \\ &= \frac{\partial}{\partial z_j^t} (w_{k,j}^{t+1} \sigma(z_j^t)) \\ &= w_{k,j}^{t+1} \sigma'(z_j^t) \end{aligned} \tag{1.20}$$

siccome la derivata di $\sigma(z_h^t)$ rispetto a z_j^t è diversa da zero solo se $h = j$. Sostituendo (1.20) in (1.18) si ottiene:

$$\delta_j^t = \sum_k w_{k,j}^{t+1} \delta_k^{t+1} \sigma'(z_j^t) \tag{1.21}$$

- **Equazione per il cambiamento della cost function rispetto a un qualsiasi peso della rete** :

$$\frac{\partial C}{\partial w_{j,k}^t} = o_k^{t-1} \delta_j^t \tag{1.22}$$

Questa equazione definisce come calcolare $\partial C / \partial w_{j,k}^t$ rispetto alle quantità note o_k^{t-1} e δ_j^t. Anche in questo caso la dimostrazione si basa sulla regola della catena in modo simile alle precedenti, perciò viene omessa.

Riassumendo, le tre equazioni viste consentono di calcolare il gradiente della cost function per un singolo pattern del training set, $C = C_{\mathbf{x}}$. Nella pratica, si combina l'algoritmo di backpropagation con un *training algorithm* come SGD. In particolare, dato un *mini-batch* di m pattern, l'algoritmo completo consiste nei seguenti passi:

1. Input del *mini-batch*
2. Per ogni pattern del *mini-batch* inizializza l'*input layer* e esegui le seguenti operazioni:
   - feedforward: per ogni nodo j di ogni layer $t = 2, .., T$ calcola $z_j^t(\mathbf{x})$ e $o_j^t(\mathbf{x})$
   - *Output error*: per ogni nodo j dell'output layer calcola l'errore $\delta_j^T(\mathbf{x})$ con (1.12)
   - backpropagation: per ogni nodo j di ogni layer $t = T-1, T-2, ..., 2$ calcola l'errore $\delta_j^t(\mathbf{x})$ con (1.16)
3. *Gradient descent*: per ogni nodo j di ogni layer $t = T, T-1, ..., 2$ aggiorna i pesi secondo la *learning rule* (1.7)

1.2.5 Universal approximation theorem

Uno dei risultati più importanti nella teoria delle reti neurali è che tali modelli possono calcolare qualsiasi funzione: il teorema è noto come *Universal Approximation Theorem*. Non importa quanto sia complessa la funzione f, è garantita l'esistenza di una rete neurale che per qualsiasi input x restituisca il valore $f(x)$. Questo risultato è valido anche per funzioni in più variabili e che restituiscono più output, ovvero funzione del tipo $f : \mathbb{R}^n \to \mathbb{R}^m$. Inoltre si può restringere la tipologia delle reti considerando quelle aventi solamente un singolo hidden layer.

É fondamentale fare alcune precisazioni: la prima è che il teorema garantisce l'esistenza di una rete ma non è costruttivo, ovvero non è detto che si abbiano le tecniche adatte per costruire o riconoscere la rete. In secondo luogo, non è corretto sostenere che "una rete può calcolare qualsiasi funzione": è più appropriato dire che una rete neurale può restituire una approssimazione arbitrariamente precisa della funzione. In ultimo, il teorema è valido solo per le funzioni continue: se una funzione è discontinua non è in generale possibile approssimare tale funzione con una rete neurale. Una formulazione più corretta dell'*Universal Approximation Theorem* è quindi la seguente: una rete neurale con un singolo hidden layer può essere utilizzata per approssimare in modo arbitrariamente preciso qualsiasi funzione continua. [9]

Le dimostrazioni formali possono essere molto tecniche, come nel caso della dimostrazione fornita nel *paper* di G. Cybenko [12]; è possibile dimostrare il teorema anche in modo grafico [9].

1.2.6 Recurrent e Residual Neural Network

Si vogliono ora introdurre due varianti principali delle reti feedforward. Il termine *recurrence* si riferisce alla proprietà attraverso la quale un neurone può influenzare se stesso tramite un ciclo nel grafo che rappresenta la rete. Le reti che presentano tali cicli (che vengono chiamate anche connessioni di *feedback*) prendono il nome di recurrent neural network (RNN) e sono predisposte per gestire sequenze di dati nel tempo (*time-series data*). [13] [8] [14]

Se un gruppo di layer di un rete feedforward approssima una funzione $f(\mathbf{x})$ (dove $\mathbf{x}$ è il pattern di input della rete), una residual neural network produce un *residual mapping*, ovvero approssima esplicitamente la funzione $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$. La funzione originale è poi riottenuta calcolando: [15]

$$f(\mathbf{x}) = g(\mathbf{x}) + \mathbf{x} \tag{1.23}$$

La formulazione $g(\mathbf{x}) + \mathbf{x}$ può essere realizzata introducendo delle *shortcut connection* in una rete feedforward. Una *shortcut connection* è una connessione tra due neuroni della rete che ”salta” uno o più layer. In altri termini, connette due neuroni che non appartengono a layer consecutivi. [8] [16]

Un esempio di blocco di una residual network (chiamato anche *residual block*) è mostrato in figura 1.6. In questo caso la *shortcut connection* corrisponde semplicemente alla funzione identità (*identity shortcut connection*): l'input x del blocco è lo stesso che viene sommato algebricamente alla funzione $g(x)$ prodotta dal blocco stesso. Le *identity shortcut connection* non aggiungono altri parametri o complessità computazionale rispetto alla rete feedforward ottenuta rimuovendo tali connessioni. Inoltre, l'intera residual network può essere ancora addestrata utilizzando l'SGD con la backpropagation. [15]

Può essere vantaggioso addestrare questo tipo di reti rispetto alle feedforward: estremizzando, se la funzione $f(x)$ ottimale è la funzione identità, sarà più facile portare $g(x) = 0$ piuttosto che fare in modo che un rete feedforward approssimi esattamente la funzione identità. In scenari reali è improbabile che la funzione ottima sia proprio l'identità; ma anche nel caso in cui

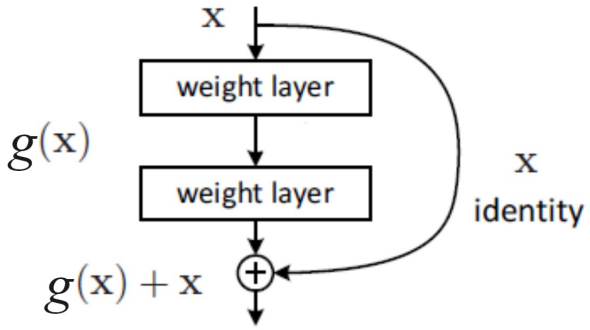


Figura 1.6: Blocco di una residual network [15]

In questo esempio la connessione salta due layer della rete

la funzione ottima sia prossima all'identità, è comunque più facile trovare una perturbazione dell'identità rispetto ad approssimare la funzione senza alcun riferimento. [15]

1.3 Analisi di time-series e modelli generativi

1.3.1 Time-series forecasting

Uno dei più rilevanti argomenti di ricerca nell'ambito dell'intelligenza artificiale e del machine learning è l'analisi e l'elaborazione delle *time-series*. Una time-series è una collezione di dati (i pattern visti precedentemente) raccolti ripetutamente nel tempo. Gli intervalli temporali con i quali vengono registrati i dati possono essere regolari o irregolari [17]. Data la loro capacità di descrivere l'evoluzione temporale di un fenomeno, queste sequenze di dati possono essere impiegate in ambiti di finanza, di *management* e di *healthcare* [18]. Lo studio delle time-series permette di fare delle previsioni sull'evoluzione del fenomeno di interesse (*time-series forecasting*) basandosi sugli schemi interni alla sequenza di dati raccolta nel passato [17].

Esistono due approcci per il time-series forecasting: il primo è il machine learning "tradizionale". In questo caso i principali metodi sono la regressione, l'*exponential smoothing* e l'*Autoregressive Integred Moving Average* (ARIMA) [19]. I metodi classici sono però limitati e spesso non possono essere applicati a problemi reali in quanto essi non riescono a gestire dati mancanti o corrotti [17]. Il secondo approccio si basa sul *deep learning*, impiegando quindi reti neurali specifiche. Le architetture più utilizzate sono le *Recurrent Neural Networks* (RNNs) e le relative varianti, che prendono il nome di *Long Short-Term Memory* (LSTM) e *Gated Recurrent*

Unit (GRU) [17].

La gestione di time-series irregolari, laddove i dati non sono raccolti a intervalli regolari, come accade ad esempio nell'ambito medico e in altre applicazioni reali, è complessa anche nel caso si utilizzino reti neurali. Una possibile soluzione è la generazione artificiale di nuovi dati attraverso modelli generativi (*generative models*) [20].

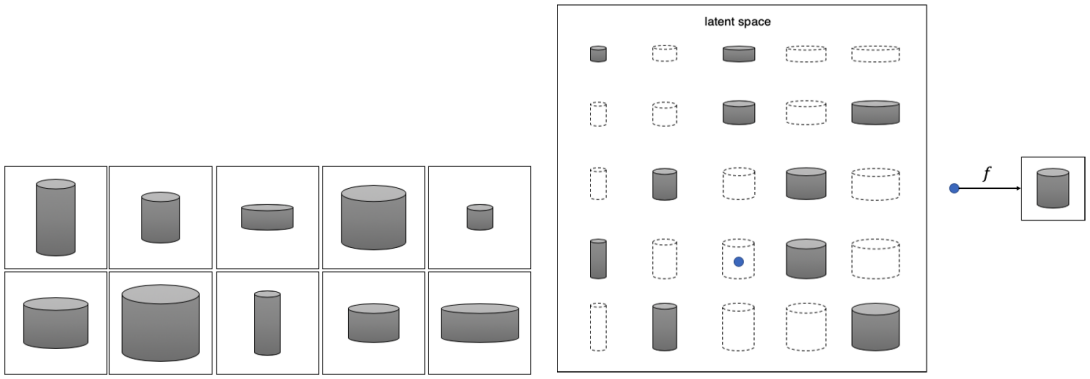
Anche in questo caso i metodi per generare *fake data* possono essere divisi in due famiglie: i metodi statistici e, anche in questo caso, i metodi di deep learning [18]. Il principale metodo statistico prende il nome di *data imputation* ed è il processo di generazione di dati tramite una semplice stima: ad esempio il dato artificiale può essere il valore medio tra i dati precedenti e successivi (*mean value imputation*), oppure essere generato in base ad un modello predittivo (*regression imputation*) [19]. La seconda classe impiega invece le reti neurali le quali, grazie alla proprietà di approssimazione universale, possono replicare qualsiasi tipo di distribuzione, costituendo così una metodologia più versatile e flessibile rispetto a quelle fornite dalla prima classe. In questo caso si parla di *deep learning generative models*.

1.3.2 Deep learning generative models

In generale, un modello generativo fornisce una descrizione di come i dati osservati sono generati in termini probabilistici. Analizzando le principali features dei dati è possibile generare nuovi dati seguendo le stesse regole. In altri termini, immaginando che i dati disponibili seguano una certa distribuzione di probabilità sconosciuta, l'obiettivo del modello generativo sarà quello di approssimare il meglio possibile tale distribuzione; in questo modo si potranno generare nuovi dati come se fossero stati inclusi in quelli disponibili inizialmente.[11]

La controparte dei modelli generativi sono i modelli discriminativi che vengono impiegati per i problemi classici del machine learning (come si è visto nei capitoli precedenti). Per chiarire la differenza, un modello discriminativo può essere addestrato a riconoscere i dipinti di Van Gogh, mentre un modello generativo può essere addestrato a riprodurli [11]. In termini matematici:

- un modello discriminativo stima la probabilità $p(y|\mathbf{x})$, ovvero la probabilità che dato un pattern **x** appartenga alla classe y;
- un modello generativo stima la probabilità $p(\mathbf{x}|y)$ nel caso sia supervisionato o $p(\mathbf{x})$ nel caso non lo sia.



(a) Training set di immagini [11] (b) Latent Space

Figura 1.7: Representation Learning [11]

A sinistra il training set di immagini in input per il modello generativo; *A destra* è rappresentato il latent space: scelto un punto dello spazio è possibile ricavare un nuovo cilindro con features diverse dai precedenti.

Il paradigma sui cui si basano i *deep learning generative models* prende il nome di *representation learning*: piuttosto di operare direttamente sullo spazio dei pattern del training set, ogni pattern viene ridotto ad uno spazio a bassa dimensionalità (*latent space*); in seguito le reti neurali vengono addestrate a rappresentare la mappa tra tale spazio e quello del dominio originale. Nella pratica ciò significa che una rete impara sia ad individuare le features più importanti per descrivere i *pattern* forniti sia a generare nuovi dati a partire dalle stesse features.

Supponendo ad esempio di avere a disposizione un trainig set di immagini bidimensionali di cilindri (figura 1.7a), l a rete dovrà essere in grado di riconoscere le due features fondamentali e caratterizzanti, ovvero l'altezza e la larghezza dei cilindri. Tali features saranno le dimensioni del latent space. Sarà quindi possibile scegliere un punto del latent space per generare un nuovo cilindro (1.7b) [13].

Come accennato sopra, modello generativo mira ad approssimare la distribuzione di probabilità p_{data} dei dati osservati con una distribuzione p_{model}, a partire da una distribuzione nota p_z. In particolare, la distribuzione p_{model} è ottenuta mappando campioni z estratti randomicamente dalla distribuzione p_z in *fake data* che vengono poi confrontati con i dati originali per migliorare la qualità del modello generativo [18].

La distribuzione p_{model} può essere calcolata implicitamente nel caso sia *intractable* o esplicitamente nel caso sia *tractable*. Nella prima categoria ricade una delle architetture più utilizzate

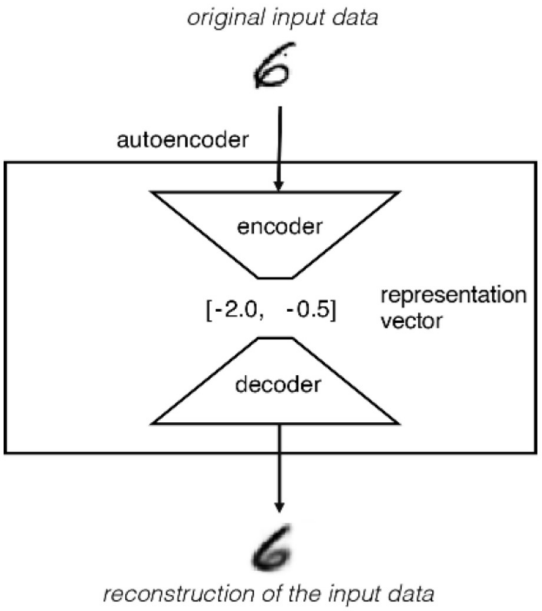


Figura 1.8: Autoencoder [11]

che prende il nome di *Generative Adversarial Networks* (GAN); essa si basa sulla ricerca di un equilibro tra una componente generatrice e una discriminatoria [18]. La seconda categoria si divide ulteriormente nei modelli che cercano di calcolare l'esatta distribuzione di probabilità e in quelli che invece ricercano una sua approssimazione. Un'altra importante architettura, nominata *Variational Autoencoders* (VAE), ricade in quest'ultima casistica. Spesso ci si riferisce a tali modelli con il nome di *latent variable models* [21].

1.3.3 Autoencoders e Variational AE

Al fine di introdurre gli argomenti necessari per comprendere al meglio l'articolo di Chen et al. [20], si procede ad una descrizione più dettagliata del modello VAE. Esso è una variante dell' *Autoencoder* (AE) [22] che è una rete neurale composta da due parti [11]:

- un rete di codifica (*encoder network*) che comprime gli input ad alta dimensionalità in rappresentazioni vettoriali a dimensionalità minore;
- una rete di decodifica (*decoder network*) che decomprime le rappresentazioni vettoriali ritornando nel dominio di origine.

Il processo è mostrato in figura 1.8.

La rete è addestrata in modo da trovare i pesi della codifica e della decodifica per far si che il valore della cost function tra il dato originale e la sua ricostruzione sia minimo. La rappresentazione vettoriale è una compressione del dato originale nel latent space. Scegliendo un punto qualsiasi del latent space è possibile generare un nuovo dato attraverso la rete di decodifica [11].

Rispetto ad un *autoencoder*, un *variational autoencoder* presenta due differenze fondamentali: la prima rispetto alla rete di codifica e la seconda rispetto alla cost function. Rispetto all'*encoder network* dell'AE, il quale mappa un dato direttamente nel latent space, quello di un VAE mappa il dato in una distribuzione normale multivariata di media μ e matrice di covarianza Σ, come mostrato in figura 1.9. Siccome nei modelli VAE si assume che non ci sia correlazione tra le dimensioni del latent space, la matrice di covarianza risulta diagonale, e quindi ogni input viene trasformato in due vettori: il vettore della media μ e quello della varianza σ. In genere si sceglie di lavorare con il logaritmo della varianza. Si può campionare un elemento del latent space tramite l'equazione

$$z = \mu + \sigma\epsilon \tag{1.24}$$

dove ϵ è un coefficiente campionato da una distribuzione normale. Questa variante rispetto all'AE garantisce che i campioni estratti in un'area attorno a μ producano una decodifica simile tra di loro.

La seconda importante differenza riguarda la cost function. Nel modello AE la *loss* consiste nell'errore quadratico medio (RSME) tra il dato originale e la sua ricostruzione. Nel modello si estende tale funzione aggiungendo una componente che prende il nome di *Kullback-Lebiler divergence* (KL). La KL *divergence* è una misura di quanto due distribuzioni di probabilità differiscano tra loro. Nel caso delle VAE si vuole misurare la differenza tra la distribuzione normale di parametri μ e σ rispetto alla distribuzione normale standard.

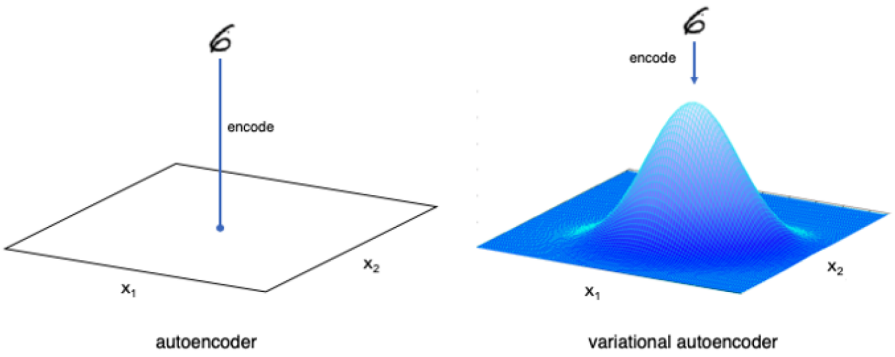


Figura 1.9: Confronto tra la codifica AE e VAE [11]

Capitolo 2

Neural Ordinary Differential Equation

Nel giugno 2018, in occasione della 32esima conferenza dell'*Neural Information Processing Systems (NeurIPS)*, viene pubblicato un articolo intitolato *Neural Odrinary Differential Equation* ad opera dei ricercatori dell'Università di Toronto Ricky Chen, Yulia Rubanova, Jesse Bettencourt e David Duvenaud.

L'articolo introduce un nuovo modello di rete neurale in cui, invece di specificare una sequenza discreta di hidden layer come nelle reti "tradizionali", si parametrizza la dinamica degli stati di tali layer usando una equazione differenziale ordinaria (*ordinary differential equation* ODE) [20]. In altri termini, l'idea principale dell'articolo è che alcuni tipi di reti neurali (come ad esempio le residual neural networks) possano essere interpretati come la discretizzazione di una equazione differenziale e, di conseguenza, l'utilizzo di risolutori di ODE allo stato dell'arte possa portare a risultati migliori. In realtà la vera novità non è aver individuato una relazione tra reti neurali e equazioni differenziali, in quanto studi precedenti alla pubblicazione di questo articolo avevano già presentato un'idea simile (si veda [23] [24] [25] [26]); Chen, Rubanova, Bettencoutr e Duvenaud sono però riusciti a trovare un metodo efficiente per calcolare il gradiente necessario per la backpropagation [27].

Nei prossimi capitoli si intende approfondire tale modello, analizzando come, partendo dalle residual neural network, sia possibile arrivare ad individuare la relazione tra equazioni differenziali e reti neurali, definendo quindi le *neural ordinary differential equation* (o Neural ODE); si evidenzieranno alcuni vantaggi, il funzionamento dell'algoritmo di addestramento e la principale applicazione del nuovo modello.

2.1 Dalle ResNet alle Neural ODEs

Per quanto detto nei capitoli introduttivi, una rete neurale è una serie discreta di layer, ognuno dei quali prende il vettore $\mathbf{h}_t$ che rappresenta lo stato del layer precedente, e produce un nuovo stato $\mathbf{h}_{t+1} = g(\mathbf{h}_t, w)$. La funzione g tipicamente è nella forma $g(\mathbf{x}, w) = \sigma(\sum_i w_i x_i)$, dove σ è la *activation function* e w è il vettore di pesi (o parametri) da ottimizzare durante l'addestramento. Alcuni ricerche testimoniano l'importanza della profondità della rete per le prestazioni [28] [29]. Ma il numero di layer non può aumentare in modo indefinito: mentre l'aggiunta di layer dovrebbe aumentare le prestazioni della rete, quello che succede è che le prestazioni calano [30] [15]. Il problema si risolve utilizzando il modello delle residual neural network introdotto da un gruppo di ricercatori presso Microsoft. Lo stato del layer è ora dato da:

$$\mathbf{h}_{t+1} = g(\mathbf{h}_t, w) + \mathbf{h}_t \tag{2.1}$$

quindi la rete deve imparare solo la differenza tra due stati successivi; questo permette di aumentare ulteriormente il numero di layer [15].

Qual è la relazione tra le residual neural network e le equazioni differenziali? Sia data la seguente equazione differenziale

$$\mathbf{h}' = \mathbf{g}(t, \mathbf{h}), \qquad t \geq t_0, \qquad \mathbf{h}(t_0) = \mathbf{h}_0 \tag{2.2}$$

dove $\mathbf{g} : [t_0, \infty) \times \mathbb{R}^d \to \mathbb{R}^d$ e rispetta la condizione di *Lipschitz*; mentre la condizione iniziale $\mathbf{h}_0 \in \mathbb{R}^d$ è un vettore noto. Si vuole stimare il valore di $\mathbf{h}(t)$ usando l'approssimazione $\mathbf{g}(t, \mathbf{h}(t)) \approx \mathbf{g}(t_0, \mathbf{h}(t_0))$ per $t \in [t_0, t_0 + k]$ dove $k > 0$ è arbitrariamente piccolo. Integrando (2.2):

$$\mathbf{h}(t) = \mathbf{h}(t_0) + \int_{t_0}^{t} \mathbf{g}(\tau, \mathbf{h}(\tau))d\tau \approx \mathbf{h}_0 + (t - t_0)\mathbf{g}(t_0, \mathbf{h}_0) \tag{2.3}$$

Data una sequenza $t_0, t_1 = t_0 + k, t_2 = t_0 + 2k, ...$ dove $k > 0$ prende il nome di *time step*, si denota con $\mathbf{h}_n$ la stima numerica della soluzione esatta $\mathbf{h}(t_n)$. Ad esempio,

$$\mathbf{h}_1 = \mathbf{h}_0 + (t - t_1)\mathbf{g}(t_0, \mathbf{h}_0) \tag{2.4}$$

$$= \mathbf{h}_0 + k\mathbf{g}(t_0, \mathbf{h}_0) \tag{2.5}$$

In generale, si ottiene il seguente schema ricorsivo:

$$\mathbf{h}_{n+1} = \mathbf{h}_n + k\mathbf{g}(t_n, \mathbf{h}_n), \qquad n = 0, 1, ... \tag{2.6}$$

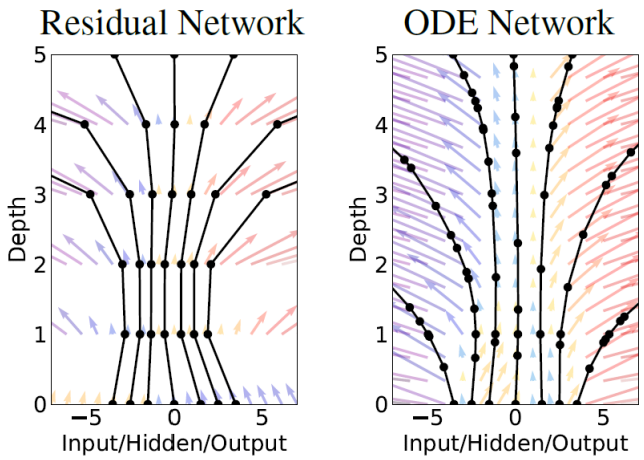


Figura 2.1: Confronto tra *residual netowork* e Neural ODE [20]

I punti neri rappresentano l'esecuzione di una stima. *A sinistra*: una *residual netowrk* definisce un numero finito di trasformazioni*A destra*: una Neural ODE definisce un campo vettoriale, che trasforma in maniera continua lo stato; in base alla complessita del campo, la Neural ODE si adatta eseguendo più o meno trasformazioni (computazione adattiva).

Questo metodo numerico per approssimare la soluzione di una equazione differenziale prende il nome di metodo di Eulero. Si noti come, per $k = 1$, l'equazione (2.6) sia identica a (2.1). Questo esplicita la relazione di cui sopra: la sequenza di passi iterativi di una residual neural network può essere interpretata come la discretizzazione con il metodo di Eulero di una equazione differenziale. [31]

Percorrendo il ragionamento a ritroso, aggiungendo infiniti layer in una residual neural network e compiendo passi infitesimi tra un layer e il successivo ($k \to 0$), si arriva a parametrizzare la dinamica continua della rete che porta dall'input all'output con una equazione differenziale ordinaria:

$$\frac{d\mathbf{h}_t}{dt} = g(\mathbf{h}(t), t, w) \tag{2.7}$$

Quindi partendo dell'input della rete $\mathbf{h}(0)$ si può definire l'output $\mathbf{h}(T)$ come la soluzione di tale ODE per una qualche condizione iniziale.

Non è immediato ridefinire il concetto di "profondità" nel caso di una Neural ODE. Una quantità connessa è il numero di stime richieste dalla dinamica degli *hidden state*, che dipende dal *solver* utilizzato e dalla condizione iniziale.

2.1.1 Vantaggi

Definire la rete come una equazione differenziale comporta i seguenti vantaggi:

- **Utilizzo di memoria**: il metodo di *training* proposto dagli autori permette di calcolare il gradiente della cost function senza propagare l'errore attraverso le operazioni del risolutore e quindi senza aver bisogno di memorizzare tutti gli stati intermedi della rete. Questo permette di addestrare la rete con un utilizzo di memoria costante rispetto alla profondità della rete. Il particolare *ODE solver* utilizzato nel seguito non è importante, ed è quindi considerato come una *black box*.

- **Computazione adattiva**: il metodo di Eulero è probabilmente il modo più semplice per risolvere una equazione differenziale; ma non è efficiente e non è accurato. Fortunatamente ci sono circa 120 anni di studio e di ricerca per trovare risolutori per equazioni differenziali (*ODE solver*). I risolutori più recenti forniscono controllo sull'errore di approssimazione e permettono di adattare la qualità della stima in base al livello di precisione richiesta.

- **Continuous Time-Series Models**: rispetto ad altri modelli, le Neural ODE permettono di incorporare naturalmente dati forniti in maniera irregolare.

2.1.2 Adjoint sensitivity method

La principale difficoltà nell'addestrare reti neurali in cui la profondità è una grandezza continua è applicare l'algoritmo di backpropagation, poiché può essere computazionalmente costoso, richiedendo elevate quantità di memoria e introducendo errori numerici.

Nell'articolo, gli autori presentano un approccio alternativo per calcolare il gradiente della cost function usando il metodo chiamato *adjoint sensitivity method* introdotto da Pontryagin nel 1962 [32]. Questo metodo scala linearmente con la dimensione del problema, richiede poca memoria e consente di controllare l'errore. Si può pensare all'*adjoint method* come l'analogo "istantaneo" della regola della catena.

Si consideri una cost function $C \in \mathbb{R}$, il cui input è il risultato di un *ODE solver*:

$$C(\mathbf{h}(t_1)) = C\left(\mathbf{h}(t_0) + \int_{t_0}^{t_1} g(\mathbf{h}(t), t, w)dt\right) = C(\text{ODESolve}(\mathbf{h}(t_0), g, t_0, t_1, w)) \tag{2.8}$$

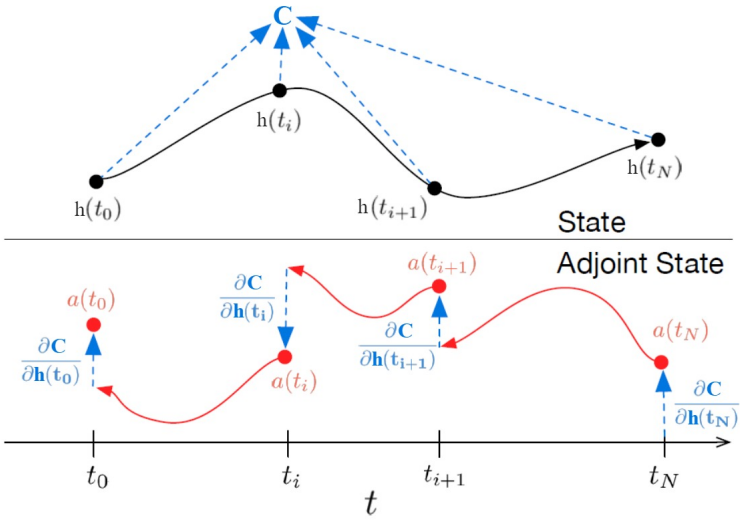


Figura 2.2: Adjoint sensitivity method [20]

L'*adjoint sensitivity method* risolve l'*augmented ODE* a ritroso rispetto al tempo. L'*augmented state* contiene sia lo stato originale sia la sensibilità della cost function rispetto allo stato.

Per ottimizzare C è necessario calcolarne il gradiente rispetto allo stato del sistema $\mathbf{h}(t)$, alla variabile t (che può essere vista come un analogo continuo della profondità; nel caso discreto corrisponde all'output del t-esimo layer) e ai pesi w.

A tal fine si introduce una nuova quantità, chiamata *adjoint*, definita come

$$\mathbf{a}(t) = \frac{\partial C}{\partial \mathbf{h}(t)} \tag{2.9}$$

che rappresenta come la cost function cambia con lo stato $\mathbf{h}(t)$ della rete allo specifico istante t. Si può definire una seconda ODE che rappresenta la dinamica dell'*adjoint*:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial g(\mathbf{h}(t), t, w)}{\partial \mathbf{h}(t)} \tag{2.10}$$

Anche questa ODE può essere risolta da una chiamata all'*ODE solver*. Il risolutore deve partire dalla condizione iniziale $\frac{\partial C}{\partial \mathbf{h}(t_1)}$ per calcolare $\frac{\partial C}{\partial \mathbf{h}(t_0)}$. Calcolare il gradiente della cost function rispetto ai parametri w richiede di risolvere una terza ODE:

$$\frac{\partial C}{\partial w} = -\int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial g(\mathbf{h}(t), t, w)}{dw} dt \tag{2.11}$$

Dimostrazione. Sia data l'equazione differenziale

$$\frac{d\mathbf{h}(t)}{dt} = g(\mathbf{h}(t), t, w)$$

Definendo l'*adjoint state* come

$$\mathbf{a}(t) = \frac{dC}{d\mathbf{h}(t)} \tag{2.12}$$

si vuole dimostrare che segue la seguente equazione differenziale:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial g(\mathbf{h}(t), t, w)}{\partial \mathbf{h}(t)} \tag{2.13}$$

Ai fini di semplificare la notazione, i vettori sono ora considerati vettori riga.

Nelle reti neurali "standard", applicando la regola della catena, si può esprimere il gradiente rispetto al layer $\mathbf{h}_t$ in funzione del layer $\mathbf{h}_{t+1}$ come

$$\frac{dC}{d\mathbf{h}_t} = \frac{dC}{d\mathbf{h}_{t+1}} \frac{d\mathbf{h}_{t+1}}{d\mathbf{h}_t} \tag{2.14}$$

Nel caso delle Neural ODE, si può scrivere la trasformazione dopo un tempo ϵ come

$$\mathbf{h}(t+\epsilon) = \mathbf{h}(t) + \int_t^{t+\epsilon} g(\mathbf{h}(t), t, w) = T_\epsilon(\mathbf{h}(t), t) \tag{2.15}$$

La dimostrazione segue dalla definizione di derivata:

$$\begin{aligned}
\frac{d\mathbf{a}(t)}{dt} &= \lim_{\epsilon \to 0^+} \frac{\mathbf{a}(t+\epsilon) - \mathbf{a}(t)}{\epsilon} && (2.16)\\
&= \lim_{\epsilon \to 0^+} \frac{\mathbf{a}(t+\epsilon) - \mathbf{a}(t+\epsilon)\frac{\partial}{\partial \mathbf{h}(t)} T_\epsilon(\mathbf{h}(t))}{\epsilon} && (2.17)\\
&= \lim_{\epsilon \to 0^+} \frac{\mathbf{a}(t+\epsilon) - \mathbf{a}(t+\epsilon)\frac{\partial}{\partial \mathbf{h}(t)} \big(\mathbf{h}(t) + \epsilon g(\mathbf{h}(t), t, w) + \mathcal{O}(\epsilon^2)\big)}{\epsilon} && (2.18)\\
&= \lim_{\epsilon \to 0^+} \frac{\mathbf{a}(t+\epsilon) - \mathbf{a}(t+\epsilon)\big(I + \epsilon \frac{\partial g(\mathbf{h}(t), t, w)}{\partial \mathbf{h}(t)} + \mathcal{O}(\epsilon^2)\big)}{\epsilon} && (2.19)\\
&= \lim_{\epsilon \to 0^+} \frac{-\epsilon \mathbf{a}(t+\epsilon) \frac{\partial g(\mathbf{h}(t), t, w)}{\partial \mathbf{h}(t)} + \mathcal{O}(\epsilon^2)}{\epsilon} && (2.20)\\
&= \lim_{\epsilon \to 0^+} -\mathbf{a}(t+\epsilon) \frac{\partial g(\mathbf{h}(t), t, w)}{\partial \mathbf{h}(t)} + \mathcal{O}(\epsilon) && (2.21)\\
&= -\mathbf{a}(t) \frac{\partial g(\mathbf{h}(t), t, w)}{\partial \mathbf{h}(t)} && (2.22)
\end{aligned}$$

Dove per ottenere l'equazione (2.17) si è applicata (2.15); mentre per ottenere l'equazione (2.18) si è usata la serie di Taylor alla funzione T_ϵ attorno a $\mathbf{h}(t)$. □

In modo analogo alla backpropagation, l'equazione differenziale per l'*adjoint state* deve essere risolta a ritroso rispetto al tempo:

$$\mathbf{a}(t_N) = \frac{dL}{d\mathbf{h}(t_N)} \quad \mathbf{a}(t_0) = \mathbf{a}(t_N) + \int_{t_N}^{t_0} \frac{d\mathbf{a}(t)}{dt} dt = \mathbf{a}(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t) \frac{\partial g(\mathbf{h}(t), t, w)}{\partial \mathbf{h}(t)} \tag{2.23}$$

Si vuole generalizzare (2.13) per ottenere il gradiente della cost function rispetto ai parametri w. É conveniente interpretare w e t come due funzioni della variabile t:

$$\frac{\partial w(t)}{\partial t} = \mathbf{0} \qquad\qquad \frac{\partial t(t)}{\partial t} = 1 \tag{2.24}$$

Si possono combinare le equazioni (2.24) con $\mathbf{h}(t)$, costruendo così l'*augmented state* con la rispettiva equazione differenziale:

$$\frac{d}{dt}\begin{bmatrix}\mathbf{h}\\ w\\ t\end{bmatrix}(t) = g_{aug}([\mathbf{h}, w, t]) = \begin{bmatrix}g([\mathbf{h}, \theta, t])\\ \mathbf{0}\\ 1\end{bmatrix} \tag{2.25}$$

L'*augmented adjoint state* è dato quindi da:

$$\mathbf{a}_{aug} = \begin{bmatrix}\mathbf{a}\\ \mathbf{a}_w\\ \mathbf{a}_t\end{bmatrix} \qquad \mathbf{a} = \frac{dC}{d\mathbf{h}(t)} \qquad \mathbf{a}_w = \frac{dC}{dw(t)} \qquad \mathbf{a}_t = \frac{dC}{dt(t)} \tag{2.26}$$

La matrice jacobiana di g ha la forma

$$\frac{\partial g_{aug}}{\partial[\mathbf{h}, w, t]} = \begin{bmatrix}\frac{\partial g}{\partial \mathbf{h}} & \frac{\partial g}{\partial w} & \frac{\partial g}{\partial t}\\ \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0}\end{bmatrix}(t) \tag{2.27}$$

Dove $\mathbf{0}$ è la matrice di zeri con la dimensione appropriata. Sostituendo nell'equazione (2.13) si ottiene

$$\frac{d\mathbf{a}_{aug}(t)}{dt} = -\begin{bmatrix}\mathbf{a}(t) & \mathbf{a}_w(t) & \mathbf{a}_t(t)\end{bmatrix}\frac{\partial g_{aug}}{\partial[\mathbf{h}, w, t]}(t) = -\begin{bmatrix}\mathbf{a}\frac{\partial g}{\partial \mathbf{h}} & \mathbf{a}\frac{\partial g}{\partial \theta} & \mathbf{a}\frac{\partial g}{\partial t}\end{bmatrix}(t) \tag{2.28}$$

Il primo elemento corrisponde all'equazione differenziale dell'*adjoint* (2.13). Il secondo elemento può essere utilizzato per ottenere il gradiente rispetto ai parametri, integrando nell'intervallo $[t_N, t_0]$ e ponendo $\mathbf{a}_\theta(t_N) = \mathbf{0}$:

$$\frac{dC}{dw} = \mathbf{a}_w(t_0) = -\int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial g(\mathbf{h}(t), t, w)}{\partial w}dt \tag{2.29}$$

Infine, dal terzo termine si ricava il gradiente rispetto a t_0 e t_N, ovvero rispetto agli estremi di integrazione:

$$\frac{dC}{dt_N} = \mathbf{a}(t_N)g(\mathbf{h}(t_N), t_N, w) \qquad \frac{dC}{dt_0} = \mathbf{a}_t(t_0) = \mathbf{a}_t(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial g(\mathbf{h}(t), t, w)}{\partial t}dt \tag{2.30}$$

Input: dynamics parameters w, start time t_0, stop time t_1, final state $\mathbf{h}(t_1)$, loss gradient $\partial C/\partial \mathbf{h}(t_1)$

$\frac{\partial C}{\partial t_1} = (\frac{\partial C}{\partial \mathbf{h}(t_1)})^T g(\mathbf{h}(t_1), t_1, w)$ — Compute gradient w.r.t. t_1

$s_0 = \left[\mathbf{h}(t_1), \frac{\partial C}{\partial \mathbf{h}(t_1)}, \mathbf{0}_{|w|}, -\frac{\partial L}{\partial t_1}\right]$ — Define initial augmented state

def aug_dynamics $([\mathbf{h}(t), \mathbf{a}(t), \cdot, \cdot], t, w)$ — Define dynamics on augmented state

  return $\left[g(\mathbf{h}(t), t, w), -\mathbf{a}(t)^T \frac{\partial g}{\partial \mathbf{h}}, -\mathbf{a}(t)^T \frac{\partial g}{\partial w}, -\mathbf{a}(t)^T \frac{\partial g}{\partial t}\right]$ — Compute vector-Jacobian products

$\left[\mathbf{h}(t_0), \frac{\partial C}{\partial \mathbf{h}(t_0)}, \frac{\partial C}{\partial w}, \frac{\partial C}{\partial t_0}\right] = ODESolve(s_0, aug\_dynamics, t_1, t_0, w)$ — Solve reverse-time ODE

return $\frac{\partial C}{\partial \mathbf{h}(t_0)}, \frac{\partial C}{\partial w}, \frac{\partial C}{\partial t_0}, \frac{\partial C}{\partial t_1}$ — Return all gradients

Figura 2.3: Pseudocodice dell'algoritmo di backpropagation [20]

2.1.3 Implementazione dell'adjoinnt method

Nell'appendice D dell'articolo è fornita una possibile implementazione dell'*adjoint method* usando la libreria Autograd di python. Viene riportato il codice come appendice dell'elaborato.

2.2 Time-series Model

Le Neural ODE sono adatte alla gestione di dati nel tempo e godono di molti vantaggi per la natura continua delle ODE si cui si basano. Possono essere infatti usate come *latent generative model*. In questo contesto sono utili a estrapolare dati o interpolare quelli mancanti in una *time-series*. Tali caratteristiche sono fondamentali nelle applicazioni reali poiché permettono di addestrare le reti anche nei casi in cui le *time-sries* del *training set* sono irregolari. [33]

In particolare, le Neural ODE possono essere integrate nei modelli VAE per generare gli stati nel *latent space*. Lo schema del modello è mostrato in figura 2.4, mentre la varie componenti possono essere riassunte come segue [34]:

$$\begin{cases} \text{RNN} & : \{x(t_i) : t_i \in [t_0, t_n]\} \rightarrow \mathbf{h}(t_0) \\ \text{N-ODE} & : \mathbf{h}(t_0) \rightarrow \mathbf{h}(t_0), ..., \mathbf{h}(t_{fore}) \\ \text{Decoder} & : \mathbf{h}(t_{fore}) \rightarrow x(t_{fore}) \end{cases}$$

In questo nuovo approccio, chiamato dagli autori *latent ODE*, ogni *time-series* è rappresentata da una *latent trajectory* determinata univocamente da uno stato iniziale (*latent initial state*)

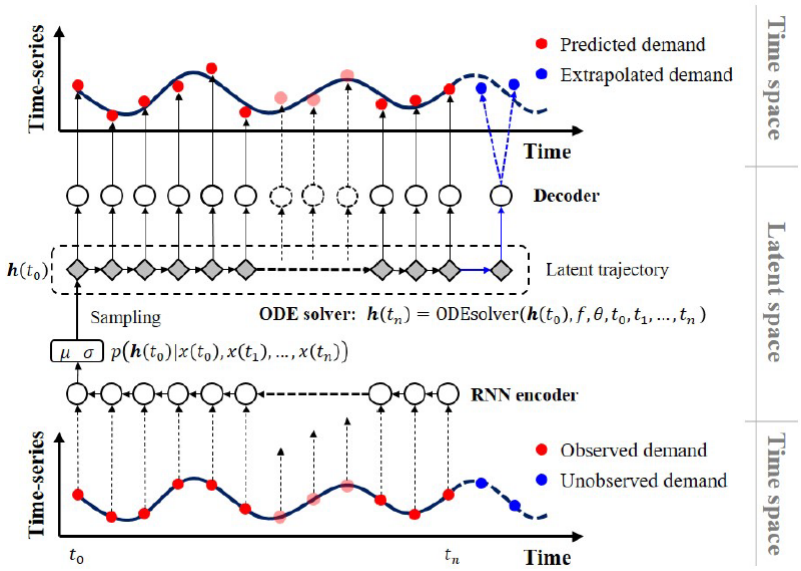


Figura 2.4: Schema di un modello per le *time-series forecasting* basato su Nueral ODE [34]

$\mathbf{h}(t_0)$ [34].

L'input è costituito dalla sequenza dei dati osservati $X = x(t_0), x(t_1), ..., x(t_n)$ agli istanti $t_0, t_1, ..., t_N$. Una RNN elabora ogni sequenza X a ritroso nel tempo, ovvero dall'istante t_n a t_0. In questo modo la rete può approssimare la probabilità a posteriori $q_\phi(\mathbf{h}(t_0)|x(t_0), x(t_1), ..., x(t_n))$, dove $\mathbf{h}(t_0)$ rappresenta lo stato iniziale per generare la traiettoria nel *latent space* e ϕ sono i parametri della rete RNN di *encoding*. Si assume che lo stato $\mathbf{h}(t_0)$ sia campionato da una distribuzione normale con media $\mu_{\mathbf{h}(t_0)}$ e varianza $\sigma_{\mathbf{h}(t_0)}$. Partendo dallo stato iniziale $\mathbf{h}(t_0)$ la traiettoria $\mathbf{h}(t_1), \mathbf{h}(t_2), ..., \mathbf{h}(t_n), \mathbf{h}(t_{fore})$ nel *latent space* può essere univocamente determinata utilizzando un *ODE solver*.

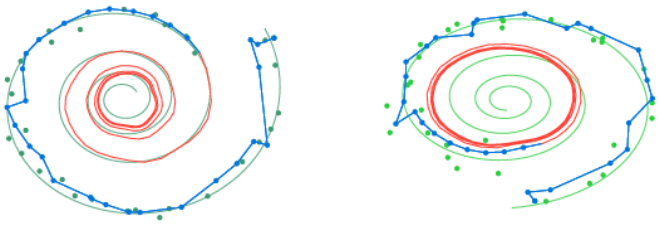
$$\frac{\partial \mathbf{h}(t)}{\partial t} = f(\mathbf{h}(t), \theta_f) \qquad \mathbf{h}_{t_1}, \mathbf{h}_{t_2}, ..., \mathbf{h}_{t_n} = ODESolver(\mathbf{h}(t_0), f, \theta_f, t_0, ..., t_n) \tag{2.31}$$

L'impiego di tali risolutori permettono di fare predizioni ($t_i \in [t_0, t_n]$) e estrapolazioni ($t_{fore} \in [t_n, +\infty]$) per un punto arbitrario del tempo. Infine la rete di decodifica ricostruisce $\hat{x}(t_0), ..., \hat{x}(t_n), \hat{x}(t_{fore})$ partendo dalla traiettoria nel *latent space* [34].

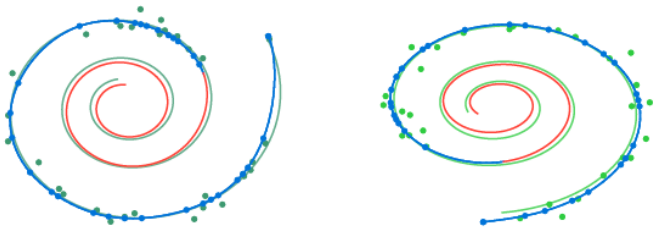
Gli autori dell'articolo [20] hanno confrontato le prestazioni di un modello *latent ODE* con quelle di una RNN nell'estrapolare le *time-series*. Il training set del *toy problem* è un dataset di 1000 spiarli bidimensionali, metà orientate in senso orario e metà in senso antiorario. Ogni spirale è campionata a intervalli casuali e ad ogni osservazione è aggiunto del rumore gaussiano per rendere il problema più realistico. I risultati sono mostrati in figura 2.5 [20].

Figura 2.5: Confronto tra RNN e Latent ODE [20]

(a) Recurrent Neural Network



(b) Latent ODE



Capitolo 3

Dal 2018 ad oggi

La pubblicazione dell'articolo di Chen et al. ha da subito suscitato grande interesse, al punto da essere tra i quattro vincitori del *Best Paper Award* del NeurIPS nel 2018. Nel dicembre 2018 sulla rivista *MIT Technology Review* del *Massachusetts Institute of Technology* viene pubblicato un articolo intitolato *"A radical new neural network design could overcome big challenges in AI"*. [35]

Sulla rivista viene brevemente spiegato come le Neural ODEs possano costituire uno strumento importante per le applicazione di *healthcare*, un tema particolarmente caro al *Vector Institute* di Toronto di cui David Duvenaud (coautore dell'articolo sulle Neural ODEs) è attualmente membro. [36]

Nell'articolo dell'*MIT* è scritto: *"Like any initial technique proposed in the field, it still needs to fleshed out, experimented on, and improved until it can be put into production. But the method has the potential to shake up the field"*. É riportata anche una dichiarazione di Richard Zemel, professore di *Computer Science* presso il *Vector Institute*: *"The paper will likely spur a whole range of follow-up work, particularly in time-series models, which are foundational in AI applications such as healthcare"*. Effettivamente i lavori pubblicati dal 2018 al 2022 in occasione delle conferenze più importanti nell'ambito dell'intelligenza artificiale, come NeurIPS, ICRL, ICML, etc., sono molti e puntano a colmare alcune lacune di questa nuova tecnologia.

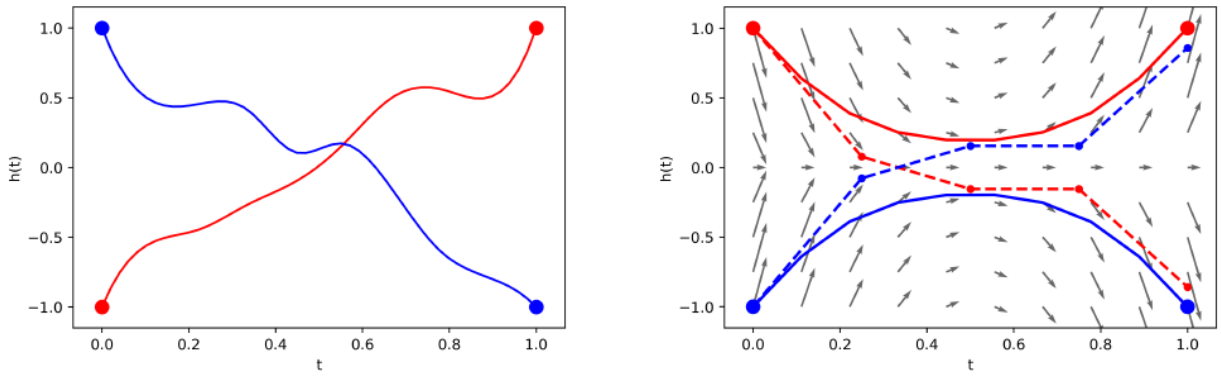


Figura 3.1: Confronto tra Neural ODE e Residual network [27]

In alto a sinistra: le traiettorie da -1 a 1 (in rosso) e da 1 a -1 (in blu) devono intersecarsi, rendendo la funzione impossibile da rappresentare per una NODE. *In alto a destra*: le soluzioni dell'NODE sono disegnate con una linea continua mentre le soluzioni ottenute con il metodo di Eulero (che corrisponde ad una ResNet) sono disegnate con linee tratteggiate. É possibile notare che la discretizzazione permette alle traiettorie di incrociarsi.

3.1 Oltre le Neural ODE

Un articolo dell'aprile 2019 pubblicato presso il NeurIPS e intitolato "*Augmented Neural ODEs*" ha evidenziato l'esistenza di classi di funzioni che non possono essere rappresentate dalle Neural ODE (NODEs). Per affrontare questa importante limitazione, gli autori dell'articolo propongono un'estensione del modello del 2018 chiamata, appunto, *Augmented Neural ODE* (ANODEs) che presenta alcuni vantaggi: oltre ad essere più espressiva risulta anche più stabile, ha performance di generalizzazione migliori con un costo computazionale minore rispetto alle Neural ODE. [27]

Sia data ad esempio la funzione $g_{1d} : \mathbb{R} \to \mathbb{R}$ tale che $g_{1d}(-1) = 1$ e $g_{1d}(1) = -1$. Le traiettorie della funzione che vanno da -1 a 1 e da 1 a -1 devono intersecarsi ma le traiettorie delle soluzioni di una ODE non possono intersecarsi quindi una ODE non può rappresentare $g_{1d}(x)$. Per inciso, una residual network può rappresentare $g_{1d}(x)$ proprio perché è una discretizzazione di una Neural ODE, permettendo alle traiettorie di saltare da un punto al successivo. Il confronto tra i due modelli è mostrato in figura 3.1. [27] Un lavoro successivo del giugno 2020, intitolato *On Second Order Behaviour in Augmented Neural ODEs*, ha esteso l'idea delle NODE alle equazioni differenziali del secondo ordine, concetto di fondamentale importanza per i processi dinamici della fisica classica. Oltre ad introdurre il modello, che prende il nome di *Second*

Order Neural ODEs (SONODEs), gli autori dell'articolo mostrano come anche il modello delle ANODE possa essere adattato per gestire dinamiche più complesse. In generale, le SONODEs riescono a trovare soluzioni più semplici per il problema considerato (figura 3.2). [37] Gli autori

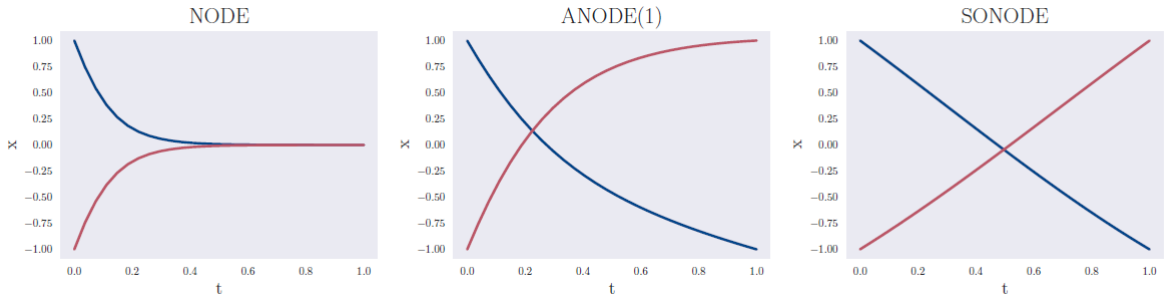


Figura 3.2: Confronto tra NODE, ANODE e SONODE [37]

In figura sono mostrate le traiettorie della funzione g_{1d} generate rispettivamente da una NODE, da una ANODE e da una SODNODE.

dell'articolo pubblicato nel dicembre 2020 e intitolato *Bayesian Neural Ordinary Differential Equations* hanno invece preso un'altra direzione, integrando con successo le Neural ODE con i metodi di inferenza bayesiana per incrementarne la robustezza [38]. Un altro importante limite delle Neural ODE è la loro insufficiente affidabilità, fattore invece fondamentale per le applicazioni ad alto rischio come quelle di *healthcare*. Su questo aspetto viene pubblicato nel giugno 2022, in occasione della 36esima conferenza sull'intelligenza artificiale (AAAI-22), un articolo intitolato *Latent Time Neural Ordinary Differential Equations* (LT-NODEs). L'approccio proposto considera T (la variabile tempo/profondità nelle Neural ODE) come una variabile latente e applica i metodi di inferenza bayesiana per incrementare l'incertezza e l'affidabilità della rete [39].

3.2 Oltre l'*adjoint method*

Un articolo del 2020 intitolato *Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE* ha sottolineato come le performance empiriche delle Neural ODE siano significativamente inferiori rispetto alle reti con layer discreti. Viene dimostrato come tale inefficienza sia dovuta all'insufficiente accuratezza del metodo di calcolo del gradiente, e in particolare come l'*adjoint method* sia particolarmente sensibile agli errori numerici. Gli autori propongono quindi un'alternativa chiamata *Adaptive Checkpoint Adajoint* (ACA) in grado di ottenere la metà

dell'*error rate* in metà del tempo di *trainig*, superando di gran lunga anche le performance delle residual network [40].
Un anno dopo, in occasione del ICLR 2021, viene pubblicato l'articolo *"MALI: A memory efficient and reverse accurate integrator for Neural ODEs"* (MALI) in cui gli autori evidenziano come l'utilizzo di memoria del metodo ACA cresca linearmente con il tempo di integrazione. Similmente all'*adjoint method*, MALI presenta un utilizzo di memoria costante rispetto al numero di *step* del *solver* e inoltre garantisce precisione e accuratezza nel calcolo del gradiente, unendo i vantaggi dei due precedenti metodi [41].

3.3 Oltre le *Latent ODE*

L'anno successivo alla pubblicazione dell'articolo, gli stessi autori hanno rivisto il modello proposto per la gestione delle *time-series*. In particolare, nell'articolo "*Latent ODEs for Irregularly-Sampled Time Series*" viene proposto un nuovo modello che generalizza quello delle RNN dove le dinamiche degli hidden layer sono definite da una equazione differenziale ordinaria. Tale modello prende infatti il nome di ODE-RNNs. Gli esperimenti condotti dagli autori dimostrano come le prestazioni dei modelli che si basano sulle ODE superino quelle delle controparti basate sulle RNN nel caso di *irregular time-series* [42].

Un successivo lavoro del 2021 ha rilevato come le Neural ODE presentino due importanti svantaggi nella trattazione delle *time-series*: il primo è che non sono in grado di gestire dati che arrivano in tempo reale, rendendo di fatto tale modello inaffidabile nelle *real-time applications*; il secondo svantaggio è che non tengono conto del fatto che le evoluzioni delle *time-series* passono essere descritte con dinamiche diverse. Gli autori notano come queste lacune possano essere colmate da un'altra famiglia di reti neurali, presentate in un lavoro del 2018 e intitolato *Conditional Neural Processes* (NPs) [43]. Tale modello è studiato per stimare l'incertezza delle descrizioni e adattarsi velocemente ai cambiamenti dei dati osservati. Viene quindi proposto un nuovo modello chiamato *Neural ODE Processes* (NPDs) che riesce a unire i vantaggi delle Neural ODEs con quelli delle NPs. [44]

La stessa problematica sull'incertezza delle dinamiche delle *time-series* è stata riscontrata in un articolo del 2022 dal titolo *Latent Time Neural Ordinary Differential Equations*. L'articolo puntualizza anche come le Neural ODE pecchino di robustezza, caratteristica cruciale nelle applicazioni reali come quelle di guida autonoma e nei contesti medici. Il modello suggerito,

omonimo al titolo dell'articolo (LT-NODEs), considera la variabile T (che implicitamente definisce la profondità della rete) come una *latent-variable* e sfrutta l'inferenza statistica bayesiana [39].

Gli articoli che sono stati menzionati sono una minima parte nel panorama di tutti quelli pubblicati negli ultimi anni sulle applicazioni delle reti neurali; questo vivace proliferare dimostra come l'interesse per una loro implementazione in ambiti industriali, commerciali o medici sia particolarmente sentito tra i ricercatori e gli studiosi del settore. L'applicazione di sistemi sempre più affidabili a tecnologie avanzate mira ad una progressiva precisione nelle prestazioni degli strumenti in cui vengono utilizzate; l'auspicio è che ciò porti ad un innalzamento della qualità della vita per chiunque, in maniera sempre più globale, abbia la possibilità di godere dei benefici della tecnologia.

Bibliografia

[1] Enciclopedia treccani. https://www.treccani.it/enciclopedia/apprendimento/. Ultimo accesso: 01-10-2022.

[2] N. Nilsson. *Introduction to Machine Learning*. MIT Press, 1998.

[3] Tom M Mitchell. *Machine learning*. McGraw-hill New York, 1997.

[4] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[5] Kevin P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, 2013.

[6] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[7] Kevin Gurney. *An Introduction to Neural Networks*. Taylor and Francis, Inc., 1997.

[8] David Kriesel. A brief introduction to neural networks. https://www.dkriesel.com/en/science/neural_networks, 2007.

[9] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018.

[10] F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, 1959.

[11] David Foster. *Generative deep learning*. O'Reilly Media, Sebastopol, CA, July 2019.

[12] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

[13] L. Fausett and L.V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall, 1994.

[14] Fathi M. Salem. *Recurrent Neural Networks*. Springer International Publishing, 2022.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[16] Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[17] Amal Mahmud and Ammar Mohammed. A survey on deep learning for time-series forecasting, 01 2021.

[18] Federico Gatta, Fabio Giampaolo, Edoardo Prezioso, Gang Mei, Salvatore Cuomo, and Francesco Piccialli. Neural networks generative models for time series. *Journal of King Saud University - Computer and Information Sciences*, 34(10, Part A):7920–7939, 2022.

[19] Douglas C Montgomery, Cheryl L Jennings, and Murat Kulahci. *Introduction to time series analysis and forecasting*. Wiley Series in Probability and Statistics. John Wiley & Sons, Nashville, TN, 2 edition, April 2015.

[20] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2018.

[21] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[22] Zhuoyue Lyu, Safinah Ali, and Cynthia Breazeal. Introducing variational autoencoders to high school students, 2021.

[23] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, 2017.

[24] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations, 2017.

[25] Weinan Ee. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5:1–11, 2017.

[26] Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations, 2018.

[27] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes, 2019.

[28] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[30] Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[31] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge University Press, 2008.

[32] L.S. Pontryagin, V.G. Boltânskij, K.N. Trirogoff, L.W. Neustadt, and R.V. Gamkrelidze. *The Mathematical Theory of Optimal Processes*. Interscience Publishers, 1962.

[33] M. L. Garsdal, V. Søgaard, and S. M. Sørensen. Generative time series models using neural ode in variational autoencoders, 2022.

[34] Xiang Xie, Ajith Kumar Parlikad, and Ramprakash Puri. A neural ordinary differential equations based approach for demand forecasting within power grid digital twins, 10 2019.

[35] Karen Hao. A radical new neural network design could overcome big challenges in ai, 2018.

[36] Vector institute. `https://vectorinstitute.ai/`. Ultimo accesso: 01-10-2022.

[37] Alexander Norcliffe, Cristian Bodnar, Ben Day, Nikola Simidjievski, and Pietro Liò. On second order behaviour in augmented neural odes, 2020.

[38] Raj Dandekar, Karen Chung, Vaibhav Dixit, Mohamed Tarek, Aslan Garcia-Valadez, Krishna Vishal Vemula, and Chris Rackauckas. Bayesian neural ordinary differential equations, 2020.

[39] Srinivas Anumasa and P. K. Srijith. Latent time neural ordinary differential equations. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(6):6010–6018, Jun. 2022.

[40] Juntang Zhuang, Nicha Dvornek, Xiaoxiao Li, Sekhar Tatikonda, Xenophon Papademetris, and James Duncan. Adaptive checkpoint adjoint method for gradient estimation in neural ode, 2020.

[41] Juntang Zhuang, Nicha C. Dvornek, Sekhar Tatikonda, and James S. Duncan. Mali: A memory efficient and reverse accurate integrator for neural odes, 2021.

[42] Yulia Rubanova, Ricky T. Q. Chen, and David Duvenaud. Latent odes for irregularly-sampled time series, 2019.

[43] Marta Garnelo, Dan Rosenbaum, Chris J. Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo J. Rezende, and S. M. Ali Eslami. Conditional neural processes, 2018.

[44] Alexander Norcliffe, Cristian Bodnar, Ben Day, Jacob Moss, and Pietro Liò. Neural ode processes, 2021.

Appendice A

Implementazione Autograd

```
import scipy.integrate

import autograd.numpy as np
from autograd.extend importprimitive, defvjp_argnumsfrom aut
ograd import make_vjp
from autograd.misc import flatten
from autograd.builtins import tuple

odeint = primitive(scipy.integrate.odeint)


def grad_odeint_all(yt, func, y0, t, func_args,       **kwargs):
    # Extended from "Scalable Inference of Ordinary Differential# Equ
    ation Models of Biochemical Processes", Sec.2.4.2
    # Fabian Froehlich, Carolin Loos, Jan Hasenauer, 2017# htt
    ps://arxiv.org/pdf/1711.08079.pdf

    T, D = np.shape(yt)
    flat_args, unflatten = flatten(func_args)

    def flat_func(y, t, flat_args):
        return func(y, t, *unflatten(flat_args))

    def unpack(x):
        #           y,       vjp_y,       vjp_t,     vjp_args
        return x[0:D], x[D:2 * D], x[2 * D], x[2 * D + 1:]

    def augmented_dynamics(augmented_state, t, flat_args):
        # Orginal system augmented with vjp_y, vjp_t and vjp_args.y,
        vjp_y, _, _ = unpack(augmented_state)
        vjp_all, dy_dt = make_vjp(flat_func, argnum=(0, 1, 2))(y, t, flat_args)vjp_y, vj
        p_t, vjp_args=vjp_all(vjp_y)          _
        return np.hstack((dy_dt, vjp_y, vjp_t, vjp_args))

    defvjp_all(g,**kwargs):vjp_y = g[

        1,:]          _
        vjp_t0 = 0
        time_vjp_list = []
        vjp_args = np.zeros(np.size(flat_args))

        for i in range(T - 1, 0, -1):
```

```
            # Compute effect of moving current time.
            vjp_cur_t = np.dot(func(yt[i, :], t[i],      *func_args), g[i, :])
            time_vjp_list.append(vjp_cur_t)
            vjp_t0 = vjp_t0 - vjp_cur_t

            #Run augmented system backwards to the previous observation.aug_y0=
            np.hstack((yt[i,:], vjp_y,vjp_t0,vjp_args))
            aug_ans = odeint(augmented_dynamics, aug_y0,
                             np.array([t[i], t[i      - 1]]), tuple((flat_args,)), **kwargs)
            _, vjp_y, vjp_t0, vjp_args = unpack(aug_ans[1])

            #Add gradient from current output.vjp_y =
            vjp_y + g[i - 1, :]

        time_vjp_list.append(vjp_t0)
        vjp_times = np.hstack(time_vjp_list)[::-1]

        return None, vjp_y, vjp_times, unflatten(vjp_args)retur
    n vjp_all


def grad_argnums_wrapper(all_vjp_builder):
    # A generic autograd helper function.       Takes a function that
    # builds vjps for all arguments, and wraps it to return only required vjps.def build_sele
    cted_vjps(argnums, ans, combined_args, kwargs):
        vjp_func = all_vjp_builder(ans,      *combined_args, **kwargs)def chose
        n_vjps(g):
            # Return whichever vjps were asked for.all_vjps=vjp_
            func(g)
            return [all_vjps[argnum] for argnum in argnums]return chosen
        _vjps
    return build_selected_vjps

defvjp_argnums(odeint, grad_argnums_wrapper(grad_odeint_all))
```