Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Fisica

NEURAL ORDINANRY DIFFERENTIAL EQUATIONS

Relatore:
Chiar.mo Prof.
GASTONE CASTELLANI

Presentata da:
MASSIMILIANO NALDI

Sessione VI
Anno Accademico 2023/2024

Sommario

Le reti neurali artificiali sono modelli computazionali ispirati alla struttura e al funzionamento del cervello umano, sviluppati per affrontare problemi complessi in ambiti quali, tra i molti: il riconoscimento di pattern, la previsione di sistemi dinamici e l'ottimizzazione di processi industriali. Il loro sviluppo storico ha attraversato diverse fasi, dai primi modelli matematici intorno alla metà del XX secolo, come il Perceptron di Rosenblatt e le Hopfield Networks, alle moderne architetture profonde che sfruttano tecniche avanzate di apprendimento (perciò si parla di Deep Learning), fino ad arrivare ai più recenti sviluppi, come le Neural Ordinary Differential Equations (in breve, Neural ODEs).
In questo elaborato viene fornita una panoramica dell'evoluzione delle reti neurali, mettendo in luce i principi fondamentali che hanno caratterizzato i modelli classici, le innovazioni che hanno condotto alle reti profonde e i paradigmi più recenti, in grado di integrare modelli continui con approcci numerici e computazionali avanzati. L'obiettivo di questa tesi è analizzare e confrontare queste architetture, evidenziando le loro applicazioni, i loro vantaggi e i limiti, con uno sguardo alle potenziali direzioni future della ricerca in questo ambito.
Per la redazione di questa tesi, sono stati utilizzati strumenti di intelligenza artificiale generativa (ChatGPT-4o, versione rilasciata a maggio 2024) per supporto nella creazione di contenuti e materiali di analisi. I dettagli sull'uso sono descritti in appendice A.4.

Indice

1 Introduzione **3**

2 Storia delle Reti Neurali **4**
2.1 Percettrone 4
2.2 Reti di Hopfield 6
2.3 Macchine di Boltzmann 9
2.4 Percettrone Multi-Strato 12
2.5 Reti Neurali Ricorrenti 14
2.6 Reti con Memoria a Breve Termine 17
2.7 Reti Neurali Convoluzionali 20
2.8 Reti Neurali Generative 22
2.9 Reti Residuali 23

3 Neural Ordinary Differential Equations **26**
3.1 Un Ponte tra il Discreto e il Continuo 26
3.2 Introduzione alle Neural ODEs 28
3.3 Confronto con i Modelli Residuali 31
3.4 Applicazioni delle Neural ODEs 32
3.5 Oltre le NODEs: Reti Neurali "Physics Informed" 33
3.6 Reti Neurali Hamiltoniane 35
3.7 Reti Neurali Lagrangiane 36

4 Implementazione Pratica e Confronto tra Modelli **39**
4.1 Architetture Utilizzate 39
4.2 Dataset Utilizzati 42
4.3 Metodi di Addestramento e Metriche di Valutazione 44
4.4 Risultati Sperimentali 46

5 Conclusioni **51**
5.1 Riflessioni Finali 51

A Appendici **53**
A.1 Codice Utilizzato 53
A.2 Dettagli Matematici 53
A.3 Algoritmi di Addestramento 58
A.4 Dichiarazione di Utilizzo GenAI 59

Capitolo 1

Introduzione

Le reti neurali artificiali rappresentano uno dei campi di studio più affascinanti e rivoluzionari dell'informatica e della fisica computazionale, che combinano elementi di modellazione biologica e applicazioni ingegneristiche. Questi modelli computazionali, originariamente ispirati dalla struttura e dal funzionamento del cervello umano, negli ultimi decenni hanno trovato applicazione in una vasta gamma di contesti scientifici e tecnologici, rivoluzionando discipline come la visione artificiale [38], il riconoscimento vocale [64], l'elaborazione del linguaggio naturale [67] e la simulazione di sistemi complessi [62].
L'idea alla base delle reti neurali risale ai primi tentativi di comprendere e modellare matematicamente i processi di apprendimento biologico. Modelli come il Perceptron di Rosenblatt (1958, [3]) hanno segnato l'inizio di un percorso che avrebbe portato allo sviluppo di reti sempre più complesse e capaci. Si faccia riferimento al Capitolo 2 per una trattazione approfondita. Un grande passo avanti in questo senso è stato fatto con l'introduzione delle Neural Ordinary Differential Equations o NODEs (2018, [36]), un paradigma che generalizza i modelli tradizionali di Deep Learning stratificato (in particolare, i modelli progettati appositamente per apprendere e riprodurre dati di tipo sequenziale), rappresentando la propagazione dell'informazione come un sistema di equazioni differenziali ordinarie. L'argomento sarà trattato in maniera approfondita nel Capitolo 3. La rilevanza delle reti neurali non si limita alle loro applicazioni tecniche: esse hanno anche avuto un impatto significativo sulla comprensione di processi fondamentali. Ad esempio, modelli neurali come le reti di Hopfield (1982, [8]) hanno permesso di studiare fenomeni come la memoria associativa e la dinamica dei sistemi complessi, collegando la teoria delle reti neurali con i sistemi dinamici non lineari. Questo ha reso le reti neurali uno strumento indispensabile non solo per ingegneri e informatici, ma anche per fisici, biologi e neuroscienziati.
Nel Capitolo 4 saranno eseguite tre task sperimentali esemplificative (nello specifico: una task di Riconoscimento di Immagini e due di Previsione Dinamica), mirate a confrontare le performance dei modelli classici rispetto alle Neural ODEs (o modelli sviluppati a partire da esse); inoltre, verranno analizzati i dati raccolti durante le task sperimentali, cercando di estrapolare informazioni utili alla comprensione delle differenze principali tra i modelli in esame da essi.

Capitolo 2

Storia delle Reti Neurali

2.1 Percettrone

Il Percettrone, introdotto da Frank Rosenblatt nel 1958 [3], rappresenta uno dei primi modelli matematici di rete neurale artificiale progettati per simulare il processo di apprendimento biologico. L'idea alla base del Percettrone è quella di imitare il comportamento dei neuroni biologici, i quali, integrando input multipli, producono un output in base a una "funzione di attivazione" (ovvero, una funzione "a soglia"). Questo modello è stato uno dei primi tentativi formali di combinare l'ispirazione biologica con l'elaborazione matematica dei dati, dando origine alla prima rivoluzione nell'ambito dell'intelligenza artificiale e dell'apprendimento automatico. Il Percettrone è anche chiamato "classificatore lineare" e funziona come segue:
Il modello assume un insieme di input $x = (x_1, x_2, \ldots, x_n)$, ciascuno associato a un peso $w = (w_1, w_2, \ldots, w_n)$.
I pesi rappresentano l'intensità della connessione tra l'input e il neurone artificiale.

L'input è combinato attraverso una somma pesata (combinazione lineare, da cui l'appellativo):

$$z = \sum_{i=1}^{n} w_i x_i + b,$$

dove b è un termine di "bias", che consente al modello di spostare la funzione di attivazione, migliorando la flessibilità del Percettrone, secondo le necessità specifiche. Funzione di attivazione: La somma pesata z è poi passata attraverso una funzione di attivazione (funzione "a soglia"), le cui scelte più comuni e convenzionali sono il "Rettificatore":

$$f(z) = max(0,\ x) = z \cdot \theta(z)$$

dove $\theta(z)$ rappresenta la "funzione a gradino" di Heaviside, oppure la funzione "segno":

$$sgn(z) = \begin{cases} 1, & \text{se } z \geq 0, \\ -1, & \text{se } z < 0. \end{cases}$$

oppure ancora la funzione Identità (cioè, l'output corrisponderà alla combinazione lineare precedente, senza nessuna modifica successiva).
Questo meccanismo determina se il neurone artificiale "si attiva" o meno, simulando il

comportamento dei neuroni biologici, infatti (con la funzione di attivazione appropriata) il risultato sarà un output binario $y \in \{0, 1\}$, che corrisponde alla classificazione effettuata dal Percettrone (da cui l'appellativo di "classificatore binario").

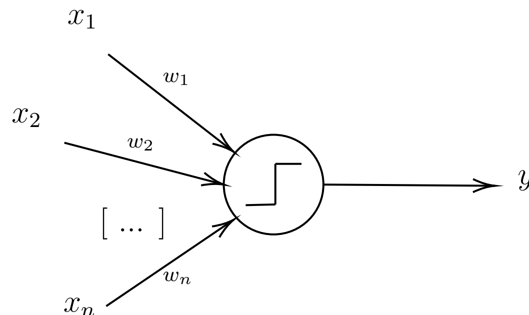

Figura 2.1: Schema del Perceptron di Rosenblatt

Il Perceptron utilizza un algoritmo di apprendimento supervisionato (cioè, basato sulla conoscenza a priori della classe di appartenenza di alcuni degli elementi dell'insieme di input, detta "label") per aggiornare i pesi in modo iterativo. L'obiettivo è minimizzare l'errore tra l'output predetto e l'output desiderato per un insieme di esempi di addestramento. L'aggiornamento dei pesi segue la seguente regola:

$$w_i \leftarrow w_i + \eta \cdot (y_{\text{true}} - y_{\text{pred}}) \cdot x_i,$$

$$b \leftarrow b + \eta \cdot (y_{\text{true}} - y_{\text{pred}}),$$

dove:

- η è il tasso di apprendimento, un parametro scalare positivo che controlla la velocità di aggiornamento dei pesi (convenzionalmente, i valori comuni sono tra 10^{-3} e 10^{-5});
- y_{true} è l'etichetta corretta dell'input;
- y_{pred} è l'output (etichetta) predetto dal modello.

Questa regola garantisce che i pesi siano modificati proporzionalmente all'errore commesso, migliorando gradualmente la capacità del Perceptron di separare i dati nel set di addestramento e di associare ad onguno di essi la corretta classe di appartenenza.
Rosenblatt sviluppò il Percettrone ispirandosi al funzionamento dei neuroni biologici, in particolare sui lavori di Warren McCulloch e Walter Pitts (1943, [1]).
I neuroni biologici ricevono segnali da altri neuroni ad essi collegati attraverso le sinapsi, sommano gli stimoli ricevuti e generano un potenziale d'azione se viene superata una soglia critica. Analogamente, il Percettrone somma gli input pesati e genera un output binario in base a una soglia stabilita dalla funzione di attivazione.
Questa analogia biologica è stata fondamentale per comprendere come processi di apprendimento possano essere modellati matematicamente. Nonostante la semplicità del Percettrone rispetto ai sistemi biologici reali, esso ha rappresentato un importante passo avanti nella creazione di modelli artificiali che si avvicinano alla complessità del cervello umano.
Tuttavia, questo modello presenta alcune limitazioni intrinseche, tra cui:

- **Separabilità Lineare**: il classificatore è in grado di distinguere correttamente solo i dati che sono linearmente separabili. Questo limite è stato evidenziato nel celebre libro "Perceptrons" di Marvin Minsky e Seymour Papert (1969, [6]), in cui si dimostra che il Perceptron non può risolvere alcuni problemi molto importanti in questo ambito, come l'operazione XOR (per una trattazione più approfondita, si rimanda alla Appendice A.2).

- **Incapacità di Rappresentazione Complessa**: questo modello è troppo semplice per apprendere relazioni complesse tra i dati.

In conclusione, il Percettrone di Rosenblatt rappresenta un pilastro fondamentale nella storia delle reti neurali, essendo il primo modello formale in grado di apprendere dai dati. Pur con le sue limitazioni, ha aperto la strada a decenni di ricerca e sviluppo, contribuendo alla comprensione dei processi di apprendimento e all'evoluzione di modelli più complessi e potenti. La sua importanza storica risiede nella sua semplicità e nella capacità di dimostrare come idee biologiche possano essere tradotte in algoritmi matematici. Inoltre, il suddetto modello ha posto le basi per lo sviluppo delle reti neurali multi-strato (MLP in breve, la cui descrizione si trova nella Sezione 2.4) e per la nascita dell'algoritmo di back-propagation, che avrebbe successivamente rivoluzionato il campo dell'apprendimento automatico.

2.2 Reti di Hopfield

Introdotta da John Hopfield nel 1982 [8], la rete omonima rappresenta un importante modello di rete neurale artificiale, progettato per simulare meccanismi di memoria associativa ("memoria a contenuto indirizzabile" o CAM, [16]). Il modello, anch'esso ispirato al funzionamento del cervello umano, utilizza una struttura a feedback per immagazzinare e richiamare pattern immagazzinati nel modello (che possiamo interpretare biologicamente come i "ricordi", [10]) attraverso un processo iterativo. La sua eleganza matematica e il parallelismo con i sistemi biologici ne hanno fatto una pietra miliare nella storia delle reti neurali.
La rete di Hopfield è costituita da N neuroni artificiali completamente interconnessi tra loro, ciascuno con uno stato binario $s_i \in \{-1, +1\}$, dove $+1$ rappresenta il neurone attivo e -1 quello inattivo. Gli stati dei neuroni evolvono nel tempo in base alle connessioni sinaptiche w_{ij} tra i neuroni i e j. Il modello assume che:

- Le connessioni siano tutte simmetriche: $w_{ij} = w_{ji} \quad \forall \quad i, j \in N$

- Non ci sono auto-connessioni: $w_{ii} = 0 \quad \forall \quad i \in N$

L'aggiornamento degli stati avviene in modo asincrono, cioè si aggiorna lo stato di un neurone alla volta (in modo simile a come spiegato nella Sezione 2.1), in base alla seguente regola:

$$s_i(t+1) = \operatorname{sgn}\left(\sum_{j=1}^{N} w_{ij} \cdot s_j(t) - \theta_i\right),$$

dove $\operatorname{sgn}(x)$ è la funzione di attivazione segno (introdotta nella sezione precedente), e θ_i è un valore di soglia associato al neurone i - esimo.

Si definisce anche l'energia della rete, una funzione scalare che misura lo stato globale del sistema, ed è definita come:

$$E_{HN} = -\frac{1}{2} \sum_{i \neq j} w_{ij} \cdot s_i \cdot s_j + \sum_i \theta_i \cdot s_i.$$

La rete di Hopfield è progettata per evolvere verso stati che minimizzano l'energia E_{HN}. Gli stati a energia minima corrisponderanno, cioè, ai pattern già immagazzinati nella rete.
La rete di Hopfield è in grado di immagazzinare un certo numero di pattern $\xi^\mu = (\xi_1^\mu, \ \xi_2^\mu, \ \ldots, \ \xi_N^\mu)$, con $\mu = 1, \ \ldots, \ P$ dove ogni $\xi_i^\mu \in \{-1, +1\}$. I pesi sinaptici w_{ij} sono calcolati usando la regola di apprendimento di Hebb:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{P} \xi_i^\mu \cdot \xi_j^\mu, \quad \text{per } i \neq j.$$

Questa regola consente alla rete di memorizzare i pattern ξ^μ come attrattori nel paesaggio energetico (o "landscape" in gergo). Quando uno stato iniziale $s = (s_1, s_2, \ldots, s_N)$ è sufficientemente vicino a uno dei pattern immagazzinati, l'evoluzione della rete convergerà verso il pattern più vicino. Questo comportamento imita la memoria associativa umana, nella quale anche solo un frammento di informazione frammentato o rumoroso è sufficiente per richiamare il ricordo completo.

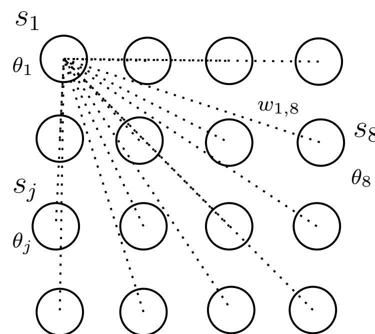

Figura 2.2: Schema di una Rete di Hopfield con $N = 16$ neuroni. Sono state evidenziate solamente le connessioni che coinvolgono il neurone $i = 1$

Nonostante la sua semplicità ed eleganza, la rete di Hopfield presenta alcune limitazioni significative [15]:

- **Capacità di Memoria Limitata**: La rete può memorizzare un massimo di circa $0.15N$ pattern distinti. Quando si supera questa capacità, i pattern immagazzinati diventano instabili e si generano errori di richiamo

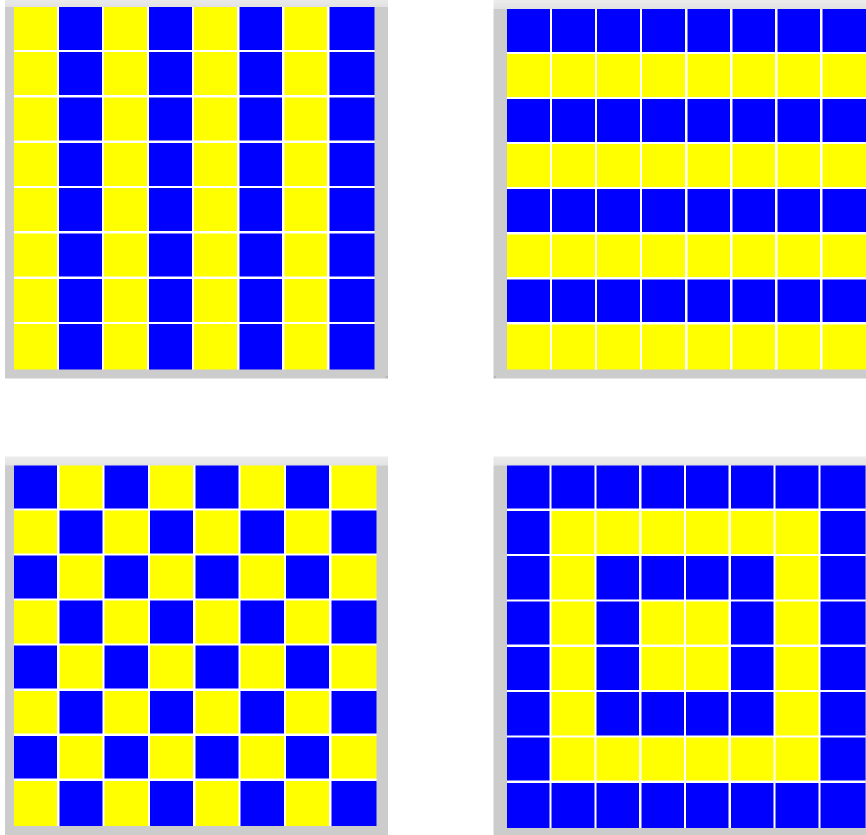

Figura 2.3: Esempio di pattern appresi da una Rete di Hopfield con $N = 64$ neuroni, i due colori indicano i due possibili stati di ogni neurone

- **Mancata Ricostruzione dei Pattern**: La rete non riesce a richiamare correttamente uno dei pattern memorizzati. Questo fenomeno può manifestarsi in due modi principali:
  - *Miscela di Pattern*: La rete converge verso uno stato che è una combinazione lineare dei pattern immagazzinati, senza rappresentare nessuno di essi in modo univoco, come si può vedere in Figura 2.4

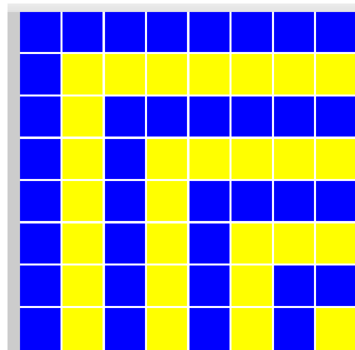

Figura 2.4: Esempio di Miscela di due dei pattern appresi dalla Rete

  - *Pattern Negativo*: La rete richiama il pattern corretto, ma invertendo tutti i suoi segni, generando $-\xi^{\mu}$ invece di ξ^{μ}, come si può vedere in Figura 2.5

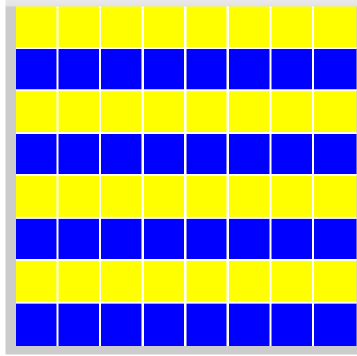

Figura 2.5: Esempio di pattern ricostruito in Negativo

- **Attrattori Locali**: La presenza di attrattori locali nel paesaggio energetico (minimi locali, ma non globali) può intrappolare la rete in stati non desiderati, impedendo il richiamo del pattern corretto

La rete di Hopfield si basa sull'idea che i ricordi nella mente umana siano organizzati come pattern associativi, richiamabili a partire da stimoli parziali, incompleti o addirittura corrotti. Questo modello matematico offre una rappresentazione semplificata di tale processo, descrivendo come informazioni complesse possano essere immagazzinate e richiamate attraverso connessioni sinaptiche. L'uso di attrattori energetici per rappresentare fenomeni collettivi complessi (come i ricordi e il loro richiamo) riflette il modo in cui il cervello umano sembra stabilizzare stati neuronali specifici per richiamare esperienze passate.
Le idee introdotte da Hopfield continuano ancora oggi a influenzare lo sviluppo di modelli neurali moderni, specialmente nel contesto della teoria dei sistemi complessi [19].

2.3 Macchine di Boltzmann

Le Macchine di Boltzmann (o Boltzmann Machines, BMs), introdotte da Geoffrey Hinton e Terrence Sejnowski nel 1985 [11], rappresentano un modello stocastico di rete neurale che integra concetti di meccanica statistica e apprendimento automatico. In una tipica BM, i neuroni vengono suddivisi in due gruppi: le unità visibili, che rappresentano i dati osservabili forniti in input, e le unità nascoste, le quali sono incaricate di catturare le dipendenze latenti e le strutture statistiche presenti nei dati. Tale architettura ha ispirato lo sviluppo di varianti semplificate, come le Macchine di Boltzmann Ristrette (RBM, [20]), in cui le connessioni interne tra unità dello stesso tipo vengono eliminate, e le Macchine di Boltzmann Profonde (DBM, [21]), che estendono l'idea a strutture gerarchiche. Un ulteriore concetto chiave introdotto nel contesto delle BM è quello della temperatura T, derivato dalla termodinamica. Il parametro T modula il grado di casualità del processo di campionamento: con T elevata il sistema esplora lo spazio degli stati in maniera più omogenea, mentre con T bassa la rete favorisce le configurazioni a bassa energia, rendendo il processo più deterministico. Di regola, nelle applicazioni standard si assume $T = 1$, ma variazioni di questo parametro possono essere utili per controllare la convergenza e la stabilità dell'apprendimento.

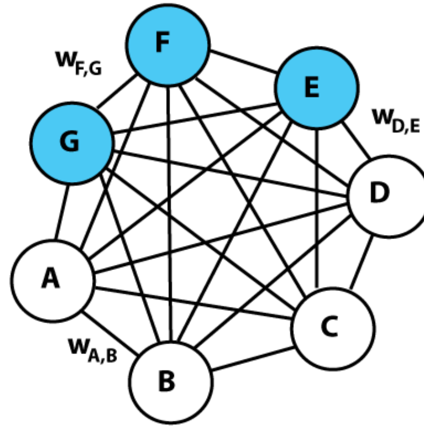

Figura 2.6: Schema di una Macchina di Boltzmann con $N = 7$ neuroni, di cui 3 "visibili" e 3 "nascosti"

Una Macchina di Boltzmann è costituita da un insieme di neuroni binari, ciascuno con stato $v_i \in \{0, 1\}$, connessi tra loro da pesi sinaptici simmetrici w_{ij}. Ad ogni configurazione dello stato globale v della rete viene associata un'energia, definita (analogamente alla Sezione precedente) come:

$$E_{BM}(v) = -\frac{1}{2} \sum_{i,j} w_{ij} \cdot v_i \cdot v_j - \sum_i b_i \cdot v_i$$

dove:

- w_{ij} è il peso sinaptico tra i neuroni i e j
- b_i rappresenta il bias associato al neurone i
- v_i è lo stato binario del neurone i
- $E_{BM}(v)$ rappresenta l'energia totale della rete per la configurazione v

Le BM apprendono minimizzando l'energia del sistema in modo simile ai processi fisici di rilassamento termodinamico, secondo un processo non-deterministico (a differenza delle Hopfield Networks, il cui apprendimento è deterministico [68]). La probabilità di osservare uno stato v sarà quindi data dalla distribuzione di Boltzmann:

$$P(v) = \frac{e^{-E_{BM}(v)/T}}{\mathcal{Z}},$$

dove $\mathcal{Z}$ è la funzione di partizione statistica, definita come:

$$\mathcal{Z} = \sum_v e^{-E_{BM}(v)/T}.$$

Il processo di apprendimento consisterà nel regolare i pesi w_{ij} e i bias b_i per massimizzare la probabilità delle configurazioni osservate nei dati di addestramento. L'aggiornamento dei pesi segue una versione modificata della regola di Hebb, adattata alla dinamica

stocastica del modello. Il gradiente della verosimiglianza logaritmica rispetto ai pesi è dato da:

$$\frac{\partial \log P(v)}{\partial w_{ij}} = \langle v_i \cdot v_j \rangle_{\text{data}} - \langle v_i \cdot v_j \rangle_{\text{pred}}$$

dove:

- $\langle v_i \cdot v_j \rangle_{\text{data}}$ è la correlazione media tra i neuroni i e j nei dati di addestramento;
- $\langle v_i \cdot v_j \rangle_{\text{pred}}$ è la correlazione media nel modello appreso.

Questa differenza misura quanto il modello deve essere aggiornato per avvicinarsi alla distribuzione desiderata. Tuttavia, il calcolo esatto di $\langle v_i \cdot v_j \rangle_{\text{pred}}$ è computazionalmente proibitivo, poiché richiede la somma su tutte le possibili configurazioni corrispondenti allo specifico stato globale della rete.
L'introduzione di questo tipo di modello ha portato numerose innovazioni nel campo delle reti neurali, quali:

- **Modellazione Probabilistica dell'Apprendimento**: Le BMs hanno reso possibile rappresentare in modo esplicito distribuzioni di probabilità complesse, permettendo di modellare relazioni tra variabili latenti
- **Parallelismo con la Meccanica Statistica**: Il modello ha stabilito un ponte teorico tra il deep-learning e la fisica statistica, sfruttando il concetto di temperatura e dinamica del rilassamento termodinamico

Nonostante il loro valore teorico e pratico, le BMs presentano diverse limitazioni:

1. **Costo Computazionale Elevato**: Il calcolo esatto della funzione di partizione $\mathcal{Z}$ è proibitivo per reti di grandi dimensioni, rendendo necessarie approssimazioni basate su Gibbs Sampling o altri metodi Monte Carlo [9]
2. **Convergenza Lenta**: Il processo di apprendimento richiede un numero molto elevato di iterazioni per raggiungere la convergenza, rendendolo poco efficiente rispetto ad altri modelli moderni
3. **Difficoltà di Scalabilità**: Mentre le RBMs hanno semplificato il modello, le BMs generali rimangono difficili da addestrare, specialmente su larga scala
4. **Applicabilità Limitata**: A causa delle loro inefficienze computazionali, le BMs originali sono state in gran parte sostituite da modelli più efficienti, come le reti neurali profonde basate su back-propagation

Il concetto di minimizzazione dell'energia, ma soprattutto l'utilizzo di approcci stocastici per l'apprendimento (che includono la gestione esplicita del parametro di temperatura) rimangono rilevanti ancora oggi, con applicazioni come la generazione di modelli probabilistici e l'ottimizzazione [24].

2.4 Percettrone Multi-Strato

Il Percettrone Multi-Strato (o MLP) rappresenta un'importante evoluzione rispetto al Percettrone introdotto da Rosenblatt. Questo modello è caratterizzato dalla presenza di più strati di neuroni, detti strati nascosti ("hidden layers"), che consentono alla rete di apprendere rappresentazioni complesse, e non più solamente lineari, dei dati. Grazie a questa struttura stratificata, il MLP supera i limiti del Percettrone originale, come l'incapacità di risolvere problemi non linearmente separabili citata in precedenza. La principale innovazione matematica che ha reso pratico l'addestramento delle reti MLP è l'algoritmo di back-propagation, che ha permesso di calcolare in modo efficiente i gradienti necessari per aggiornare i pesi sinaptici [13].

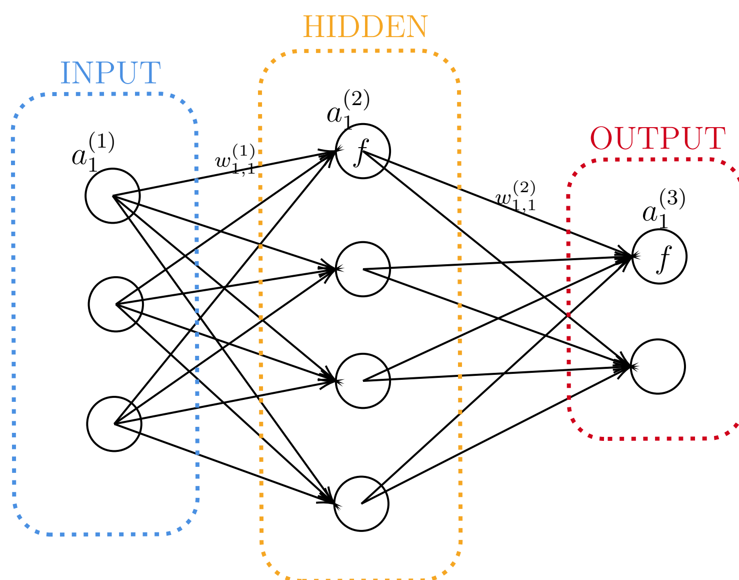

Figura 2.7: Schema di un MLP, costituito da: 3 neuroni nello Strato di Input, 4 in quello Nascosto, e 2 nello Strato di Output

Un MLP è composto da tre tipi principali di strati:

1. **Strato di Input**: Ogni neurone rappresenta una caratteristica del dato in ingresso
2. **Strati Nascosti**: Combinano i segnali provenienti dallo strato precedente attraverso connessioni pesate, introducendo capacità di modellazione non lineare grazie alle funzioni di attivazione
3. **Strato di Output**: Restituisce il risultato finale, che può rappresentare una classificazione, una regressione o un'altra forma di output

Matematicamente, ogni strato è definito come una funzione composta. Supponendo che uno strato l abbia N_l neuroni, il valore di attivazione del neurone j è dato da:

$$a_j^{(l)} = f\left(\sum_{i=1}^{N_{l-1}} w_{ij}^{(l)} \cdot a_i^{(l-1)} + b_j^{(l)}\right)$$

dove:

- $w_{ij}^{(l)}$ rappresenta il peso della connessione tra il neurone i dello strato $l-1$ e il neurone j dello strato l
- $b_j^{(l)}$ è il bias del neurone j
- f è una funzione di attivazione come la sigmoide: $\sigma(x) = \frac{1}{1+e^{-x}}$, la tangente iperbolica: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ o le funzioni già introdotte nella Sezione 2.1

L'output sarà quindi calcolato propagando l'input attraverso tutti gli strati della rete.
Per addestrare un modello MLP, è necessario definire una funzione di perdita che quantifichi l'errore tra l'output predetto $\hat{y}$ e la vera "label" y. Un esempio comune è la funzione di perdita quadratica media per problemi di regressione (Mean Squared Error Loss o MSE Loss):

$$L = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

oppure la Cross-Entropy Loss, utilizzata soprattutto per problemi di classificazione o che producono distribuzioni di probabilità come output:

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

L'obiettivo dell'addestramento è minimizzare questa funzione di perdita, aggiornando iterativamente i parametri (ovvero pesi e bias) della rete.
L'algoritmo di back-propagation è il cuore dell'addestramento di un MLP. Si basa sulla regola della catena per calcolare in modo efficiente il gradiente della funzione di perdita rispetto ai pesi e ai bias della rete. Il processo può essere suddiviso in tre fasi principali:

1. **Forward Pass**: L'input viene propagato attraverso la rete per calcolare l'output $\hat{y}$ e la funzione di perdita associata L
2. **Backward Pass**: Si calcolano i gradienti dei pesi e dei bias strato per strato, procedendo dallo strato di output verso lo strato di input. Per uno specifico peso $w_{ij}^{(l)}$, il gradiente è dato da:

   $$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} \cdot a_i^{(l-1)}$$

   dove $a_i^{(l-1)}$ è il valore di attivazione in ingresso proveniente dal neurone i dello strato precedente e $\delta_j^{(l)}$ è l'errore retro-propagato associato al neurone j nello strato l, calcolato come:

   $$\delta_j^{(l)} = \begin{cases} \hat{y}_j - y_j & \text{per lo strato di output,} \\ f'(z_j^{(l)}) \cdot \sum_{k=1}^{N_{l+1}} \delta_k^{(l+1)} \cdot w_{jk}^{(l+1)} & \text{per gli strati nascosti.} \end{cases}$$

3. **Aggiornamento dei Parametri**: I pesi e i bias vengono aggiornati usando il gradiente calcolato e un tasso di apprendimento η:

   $$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \frac{\partial L}{\partial w_{ij}^{(l)}}$$

Questo processo viene ripetuto per tutte le iterazioni fino a quando la funzione di perdita raggiunge un valore accettabile.
L'introduzione dell'algoritmo di back-propagation ha segnato una svolta fondamentale nella storia delle reti neurali, consentendo di:

- Addestrare reti con *più strati nascosti*
- Gestire problemi *non linearmente separabili*
- Apprendere *rappresentazioni gerarchiche dei dati*, dove gli strati superiori catturano caratteristiche più astratte

Nonostante i suoi meriti, il MLP presenta alcune limitazioni:

1. **Vanishing/Exploding Gradient Problem**: Per reti molto profonde, i valori numerici dei gradienti possono diventare estremamente piccoli o estremamente grandi, rallentando o destabilizzando l'aggiornamento dei pesi e impedendo un apprendimento efficace
2. **Overfitting**: Il MLP può sovradattarsi ai dati di addestramento, richiedendo tecniche come la regolarizzazione o il "dropout" per migliorare la generalizzazione
3. **Efficienza Computazionale**: L'addestramento di grandi reti MLP può essere computazionalmente intensivo, soprattutto per dataset di grandi dimensioni

Il Percettrone Multi-Strato ha rappresentato un passo avanti decisivo nel campo delle reti neurali, introducendo una struttura capace di modellare relazioni complesse e non lineari. L'algoritmo di back-propagation, in particolare, ha fornito le basi matematiche per l'addestramento di reti profonde, aprendo la strada alle applicazioni moderne dell'intelligenza artificiale. Nonostante le sue limitazioni, il MLP rimane una componente fondamentale nella storia delle reti neurali e continua a essere una base concettuale per modelli più avanzati.

2.5 Reti Neurali Ricorrenti

Le Reti Neurali Ricorrenti (o RNNs) rappresentano un'importante evoluzione nel campo delle reti neurali, progettate per gestire dati sequenziali e catturare dipendenze temporali. L'idea fondamentale delle RNNs si basa sulla capacità di mantenere una "memoria interna", che consente alla rete di elaborare informazioni in cui l'ordine degli elementi è significativa, come sequenze di testo, serie temporali o segnali audio. Sebbene il concetto di ricorrenza fosse già stato esplorato da John Hopfield nel 1986 [12], le RNNs moderne sono diventate una componente cruciale nelle applicazioni di deep learning.

Le RNN si distinguono dalle reti "feed-forward" (nome relativo al flusso di informazioni propagato all'interno della rete) trattate finora per la loro "struttura ricorrente": ogni neurone è cioè collegato non solo ai neuroni del livello successivo, ma anche a se stesso, o ad altri neuroni dello stesso livello, attraverso connessioni che formano cicli. Ciò consente alla rete di mantenere un contesto storico lungo una sequenza.

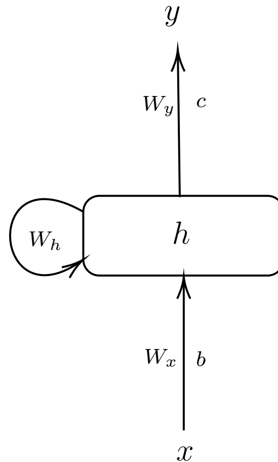

Figura 2.8: Schema semplificato di una RNN

Il funzionamento matematico di una RNN può essere descritto come segue:

1. **Stato Nascosto**

   Ogni istante temporale t è associato a uno stato nascosto h_t, che sintetizza le informazioni della sequenza antecedente il tempo t. Lo stato nascosto viene quindi aggiornato ricorsivamente come:

   $$h_t = f(W_h h_{t-1} + W_x x_t + b),$$

   dove:

   - h_{t-1} è lo stato nascosto del passo temporale precedente
   - x_t è l'input al tempo t
   - W_h e W_x sono rispettivamente i pesi delle connessioni ricorrenti e di input
   - b è il bias
   - f è una funzione di attivazione non lineare, tipicamente una tangente iperbolica o una funzione sigmoide

2. **Output**

   L'output della rete al tempo t può essere calcolato come:

   $$y_t = g(W_y h_t + c)$$

   dove:

   - W_y è il peso che collega lo stato nascosto all'output
   - c è il bias associato all'output
   - g è una funzione di attivazione specifica per il compito, ad esempio una SoftMax $s(z)$ per task di classificazione:

   $$s(z)_j = \frac{e^{s_j}}{\sum_{k=1}^{K} e^{s_k}}$$

Questo schema permette alla rete di aggiornare il proprio stato ad ogni passo temporale, creando una memoria dinamica che dipende dall'intera sequenza precedente.

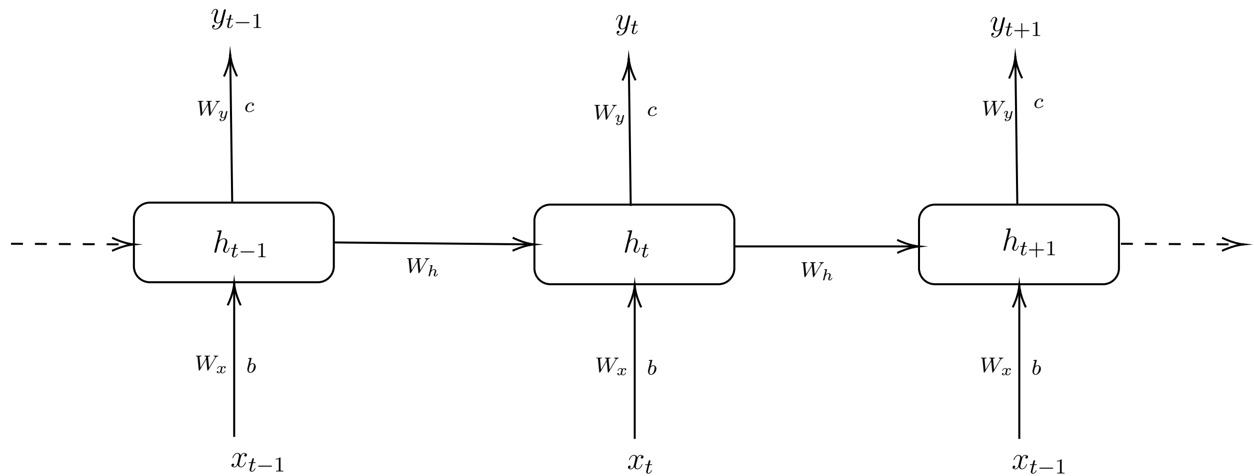

Figura 2.9: Schema di una RNN "srotolata", per rendere più chiaro il meccanismo della Ricorrenza Temporale

3. **Memoria e Ricorrenza**

   La formulazione della memoria associativa Hopfield (e, in particolare, dei ricordi come attrattori dinamici dello stato del sistema), trattata nella Sezione 2.2, ispirò lo sviluppo delle RNNs, nelle quali lo stato nascosto h_t può essere visto come un'evoluzione dinamica che incorpora tutte le memorie passate, sebbene le RNNs moderne siano più flessibili nel trattare sequenze lunghe rispetto ai modelli precedenti (un esempio di questo tipo di rete verrà trattato nella prossima Sezione: 2.6).

4. **Algoritmo di Backpropagation Through Time o BPTT**

   Il principale strumento per addestrare le RNN è l'algoritmo di Backpropagation Through Time (BPTT), che estende il tradizionale algoritmo di backpropagation per gestire le dipendenze temporali. L'idea principale è quella di "srotolare" la rete attraverso il tempo, trattando ogni passo temporale come uno strato distinto.

   L'errore totale della rete sarà dunque dato dalla somma degli errori su tutti i passi temporali:

$$L = \sum_{t=1}^{T} L_t,$$

   dove L_t è la funzione di perdita calcolata al tempo t. Il gradiente della perdita rispetto ai pesi è calcolato propagando l'errore indietro attraverso il tempo:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}.$$

Tuttavia, il BPTT soffre degli stessi problemi principali introdotti in precedenza: Vanishing/Exploding Gradient (si veda Sezione 2.4), oltre che a presentare alcune limitazioni significative:

- **Memoria Limitata**: A causa del Vanishing Gradient, le RNNs potrebbero non essere in grado di catturare efficacemente dipendenze a lungo termine
- **Costo Computazionale**: Il BPTT è computazionalmente intensivo, specialmente per lunghe sequenze temporali
- **Difficoltà di Addestramento**: L'ottimizzazione di reti ricorrenti è spesso instabile, richiedendo tecniche aggiuntive come il "clipping" dei gradienti (tecnica che spesso porta a una perdita di informazioni, e quindi a una precisione minore)

Questi limiti hanno portato allo sviluppo di architetture avanzate, come le Long Short-Term Memory (LSTM) e le Gated Recurrent Units (GRU), progettate per mitigare i problemi di instabilità numerica e migliorare la capacità di memorizzazione, tuttavia le RNNs hanno rappresentato un passo cruciale nella storia dell'intelligenza artificiale, permettendo di affrontare problemi sequenziali con una prospettiva temporale.

2.6 Reti con Memoria a Breve Termine

Le reti Long Short-Term Memory (LSTM), introdotte da Sepp Hochreiter e Jürgen Schmidhuber nel 1997 [17], rappresentano un'innovazione fondamentale nel campo delle RNN (si veda la Sezione 2.5), poiché queste reti sono state progettate per affrontare i limiti principali delle RNNs classiche, come il problema del Vanishing/Exploding Ggradient, migliorando la capacità di apprendere e memorizzare dipendenze a lungo termine nelle sequenze. Grazie alla loro struttura unica, le LSTM hanno avuto un impatto significativo in molte applicazioni, tra cui l'elaborazione del linguaggio naturale [25], il riconoscimento vocale [27] e la modellazione di serie temporali [37].
Le LSTM introducono un'unità speciale chiamata "Cellula di Memoria", che funge da accumulatore per mantenere le informazioni più rilevanti nel tempo. La cellula di memoria è controllata da tre "porte" (Gates): il Gate di Input, il Gate di Dimenticanza (Forget Gate) e il Gate di Output. Questi Gate, regolati da funzioni di attivazione, consentono alla rete di controllare in modo dinamico quali informazioni aggiungere, rimuovere o leggere dalla Cellula di Memoria.

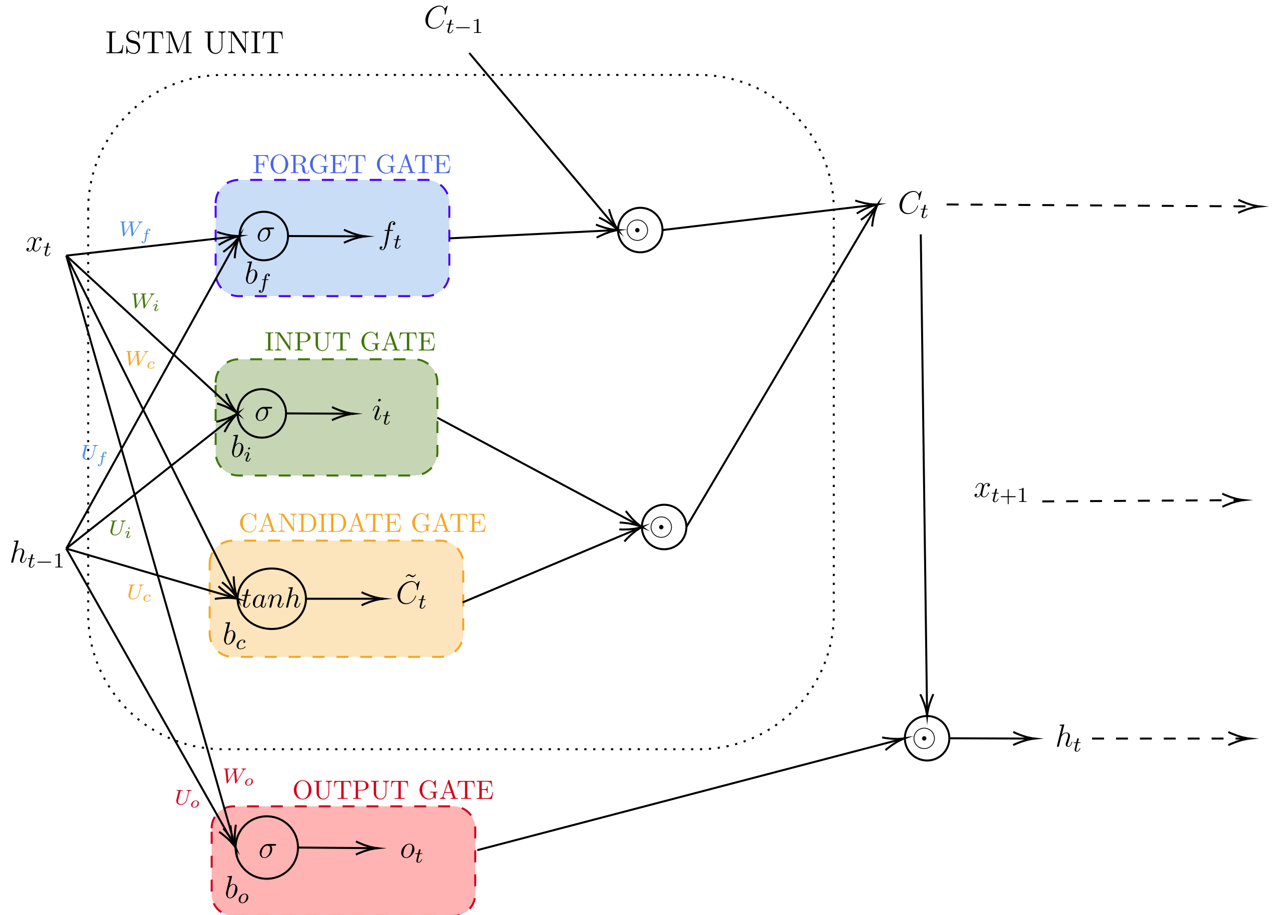

Figura 2.10: Schema di una unità di una Rete LSTM

Il funzionamento matematico di una LSTM al tempo t è descritto come segue:

1. **Forget Gate**: Determina quale parte dello stato precedente della cellula di memoria deve essere dimenticata:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

   dove:

   - x_t è l'input al tempo t
   - h_{t-1} è lo stato nascosto del passo temporale precedente
   - W_f e U_f sono i pesi associati all'input e allo stato nascosto
   - b_f è il bias
   - σ è la funzione sigmoide

2. **Input Gate**: Controlla quali informazioni dell'input corrente devono essere aggiunte alla cellula di memoria:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

   dove $\tilde{C}_t$ rappresenta il nuovo contenuto candidato per la Cellula di Memoria

3. Aggiornamento della Cellula di Memoria: Combina il precedente stato della Cellula di Memoria con il nuovo contenuto, utilizzando il contenuto informativo di Forget Gate e Input Gate:
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$
dove $\odot$ denota il prodotto elemento per elemento

4. **Output Gate**: Determina quale parte dello stato della Cellula deve essere restituita come output della rete:
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

In sintesi, la Cellula di Memoria C_t agisce come un accumulatore controllato dai gate, mentre lo stato nascosto h_t fornisce l'output visibile della rete.

Le LSTM hanno introdotto diversi miglioramenti rispetto alle RNNs classiche:

- **Gestione Migliorata delle Dipendenze a Lungo Termine**: Grazie al Forget Gate e alla Cellula di Memoria, le LSTM possono conservare informazioni rilevanti per periodi di tempo prolungati, evitando il problema del Vanishing Gradient che affligge le RNNs standard
- **Selezione Dinamica delle Informazioni**: I Gate consentono alla rete di decidere in modo dinamico quali informazioni mantenere, dimenticare o utilizzare per l'output, migliorando la capacità di adattarsi a diversi tipi di sequenze
- **Versatilità**: Le LSTM sono state utilizzate con successo in una vasta gamma di applicazioni, dimostrando la loro capacità di gestire sequenze complesse in domini diversi

Nonostante le loro innovazioni, le LSTM presentano alcune limitazioni:

1. **Complessità Computazionale**: La presenza di molteplici gate e la necessità di calcolare pesi aggiuntivi rendono le LSTM più lente da addestrare rispetto alle RNNs standard
2. **Difficoltà di Interpretazione**: La natura complessa delle LSTM rende difficile interpretare come e perché determinati pattern vengano appresi o memorizzati
3. **Memoria Limitata**: Sebbene siano più efficaci delle RNNs, le LSTM possono comunque incontrare difficoltà nel gestire sequenze estremamente lunghe, specialmente in problemi con dipendenze molto distanti
4. **Rigidità della Struttura**: Le LSTM utilizzano una struttura discreta basata su Gate, che non è sempre ideale per modellare dinamiche fluide e continue

Le Long Short-Term Memory hanno rivoluzionato il campo dell'apprendimento sequenziale, superando le limitazioni delle RNNs classiche e introducendo un meccanismo di memoria efficace e flessibile. Tuttavia, questi limiti hanno portato allo sviluppo di architetture alternative che affrontano alcune di queste problematiche, riducendo la complessità (Gated Recurrent Units, GRU) o introducendo una propagazione continua (Neural Ordinary Differential Equations, NODEs).

2.7 Reti Neurali Convoluzionali

Le Convolutional Neural Networks (CNNs) rappresentano una delle architetture più rilevanti nell'ambito delle reti neurali profonde, particolarmente efficaci nell'elaborazione di dati strutturati spazialmente, come le immagini. Introdotte da LeCun et al. (1998, [18]), le CNN si distinguono per l'uso di operazioni di convoluzione che permettono di estrarre caratteristiche significative dai dati di input, riducendo il numero di parametri rispetto alle reti completamente connesse.

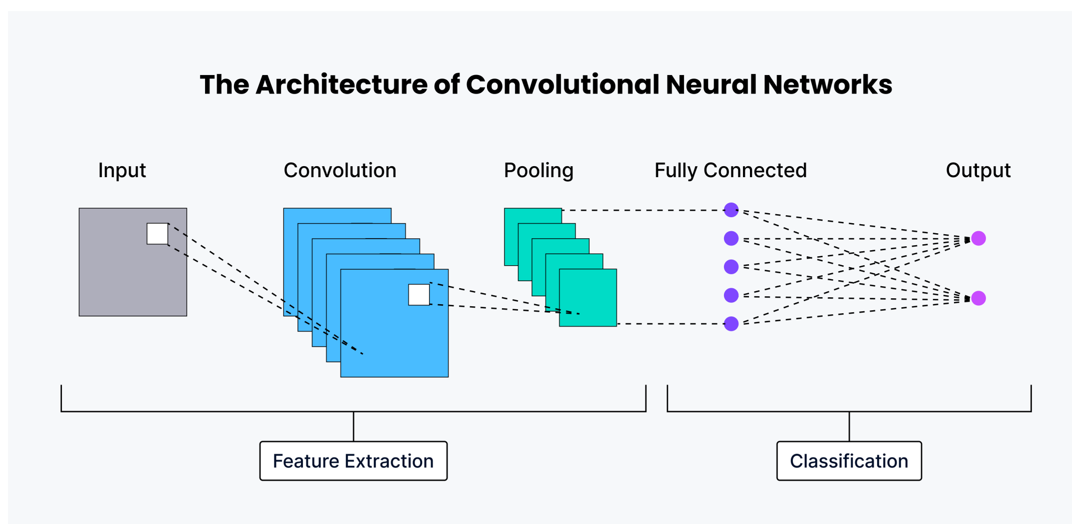

Figura 2.11: Schema dell'architettura di una Convolutional Neural Network

Una CNN è composta da una sequenza di strati, ciascuno con una funzione specifica:

1. **Strato di Convoluzione**: L'operazione chiave delle CNN è la convoluzione discreta tra un input X e un filtro K, definita come:

$$(X * K)(i, j) = \sum_{m} \sum_{n} X(i - m, j - n)K(m, n)$$

   dove la coppia (i, j) rappresenta le coordinate spaziali dell'output.

   Questa operazione permette di estrarre pattern locali presenti nei dati.

2. **Funzione di Attivazione**: Dopo la convoluzione, viene applicata una funzione di attivazione non lineare, tipicamente il Rettificatore già introdotto nella Sezione 2.1. L'uso di questa funzione consente di evitare il problema della saturazione dei gradienti, migliorando l'efficienza dell'addestramento.

3. **Strato di Pooling**: Il "pooling" è un'operazione essenziale nelle CNNs, progettata per ridurre la *dimensionalità spaziale* delle *feature maps*, diminuendo il numero di parametri e la complessità computazionale, senza perdere informazioni significative. Il termine *feature maps* si riferisce alle rappresentazioni intermedie dell'input, che vengono ottenute applicando filtri convoluzionali alle immagini in ingresso. Ogni *feature map* evidenzia caratteristiche specifiche dell'immagine, come bordi, texture o pattern più complessi nelle profondità successive della rete.

Lo strato di pooling agisce sulle *feature maps* comprimendole e aumentando la capacità della rete di riconoscere caratteristiche rilevanti, indipendentemente dalla loro posizione esatta all'interno dell'immagine. L'operazione più comune è il **max-pooling**, che prende il valore massimo all'interno di una certa finestra di pooling di dimensione $k \times k$. La sua definizione matematica è:

$$P(i,j) = \max_{(m,n) \in \Omega} X(i+m, j+n)$$

dove $X(i,j)$ rappresenta i valori della *feature map* in input e Ω è la finestra di pooling centrata in (i,j). L'operazione seleziona il valore massimo all'interno di ciascuna finestra, riducendo la dimensione spaziale della *feature map* e conservando le attivazioni più forti, che corrispondono alle caratteristiche più importanti.

Un'altra variante è il **average pooling**, che invece calcola la media dei valori all'interno della finestra di pooling:

$$P(i,j) = \frac{1}{|\Omega|} \sum_{(m,n) \in \Omega} X(i+m, j+n)$$

Sebbene il max-pooling sia più diffuso perché preserva meglio le caratteristiche dominanti, l'average pooling è utile in alcuni casi, come quando si vuole ottenere una rappresentazione più liscia ("blurred") e meno sensibile a valori estremi. Complessivamente, il pooling migliora la generalizzazione della rete riducendo la sensibilità a piccole traslazioni nell'immagine e aiuta a prevenire overfitting, specialmente quando i dati di addestramento sono limitati.

4. **Strato Completamente Connesso**: Gli output convoluzionali vengono appiattiti e passati a uno o più strati completamente connessi, con una funzione di attivazione SoftMax per la classificazione finale.

Le CNN hanno subito diverse evoluzioni che ne hanno migliorato l'efficacia:

- *Reti Residuali* (ResNet, trattate in modo più approfondito nella Sezione 2.9): Introducono connessioni residuali per mitigare il problema della scomparsa del gradiente [31]
- *Reti a Convoluzione Profonda*: Architetture come VGGNet e Inception ottimizzano la profondità e la struttura della rete [29]
- *Reti a Convoluzione Trasposta*: Utilizzate per generare immagini, come nelle Generative Adversarial Networks (GAN) [30]

Le applicazioni delle CNN includono il riconoscimento di oggetti [34], la segmentazione di immagini [28], la diagnostica medica [33] e l'elaborazione del linguaggio naturale [22].

Nonostante le loro prestazioni, le CNN presentano alcune limitazioni:

- **Elevata Complessità Computazionale**: L'addestramento richiede grandi quantità di dati e risorse computazionali elevate
- **Bassa Interpretabilità**: La natura delle feature apprese non è sempre facilmente interpretabile

- **Dipendenza dai Dati**: Le prestazioni dipendono fortemente dalla qualità e dalla quantità dei dati di addestramento

Lo sviluppo di architetture più efficienti e l'uso di tecniche di interpretabilità rimangono temi centrali della ricerca sulle CNN.

2.8 Reti Neurali Generative

Le reti neurali generative rappresentano una classe di modelli di apprendimento automatico capaci di **generare nuovi dati** con caratteristiche statistiche simili a quelle dei dati di addestramento. Introdotte nel contesto del machine learning per modellare distribuzioni di probabilità complesse, queste reti si basano su principi di ottimizzazione probabilistica e apprendimento non supervisionato [35].

Le reti neurali generative mirano ad apprendere una distribuzione dei dati $p_{data}(x)$ e a campionare nuove istanze da una distribuzione approssimata $p_{model}(x)$. Generalmente, il processo di generazione segue il principio:

$$x \sim p_{model}(x|z) \qquad z \sim p_N(z)$$

con z rappresentante una variabile latente campionata da una distribuzione semplice $p_N(x)$ (ad esempio, gaussiana). L'addestramento avviene ottimizzando una funzione di costo che riduce la discrepanza tra $p_{data}(x)$ e $p_{model}(x)$, comunemente utilizzando la massimizzazione della verosimiglianza o metodi basati sulla teoria dell'informazione come la divergenza di Kullback-Leibler:

$$D_{KL}(p_{data}|p_{model}) = \sum_x p_{data}(x) \cdot \log(\frac{p_{data}(x)}{p_{model}(x)})$$

Le reti neurali generative hanno avuto un impatto significativo in numerosi campi:

- **Sintesi di Immagini e Video**: Tecniche come la generazione di volti realistici [42] o la creazione di contenuti multimediali a partire da descrizioni testuali [60]
- **Modellazione dei Dati Biologici**: Generazione di strutture proteiche o molecole con proprietà desiderate [55]
- **Apprendimento semi-supervisionato**: Utilizzo della generazione per migliorare la performance di modelli supervisionati sfruttando dati sintetici [32]

Nonostante i progressi, le reti neurali generative presentano alcune limitazioni:

- **Bassa Stabilità dell'Addestramento**: Problemi come il "mode collapse" in alcuni modelli generativi rendono difficile ottenere una buona copertura della distribuzione dei dati
- **Bassa Interpretabilità**: La natura complessa dei modelli generativi rende arduo comprendere quali caratteristiche specifiche influenzano la generazione
- **Elevato Costo Computazionale**: L'addestramento di questi modelli, specialmente su grandi dataset, richiede elevate risorse computazionali

Un particolare sottogruppo di reti neurali generative è rappresentato dai Large Language Models (LLMs), progettati per generare e comprendere il linguaggio naturale su larga scala. Questi modelli si basano su tecniche di pre-training su grandi corpora testuali e utilizzano architetture profonde, come i Transformer [52], per catturare dipendenze linguistiche complesse. L'ottimizzazione avviene minimizzando funzioni di perdita basate sulla probabilità condizionale delle parole nel testo:

$$L = -\sum_t \log p(w_t | w_{t'<t})$$

Gli LLMs hanno rivoluzionato campi come la traduzione automatica, la generazione di contenuti e il supporto alle interazioni uomo-macchina. [66] [46].

2.9 Reti Residuali

Le reti residuali, introdotte da Kaiming He et al. nel 2016 [31], rappresentano un'importante evoluzione nel campo delle reti neurali profonde, affrontando in modo efficace il problema del Vanishing Gradient, che affligge soprattutto l'addestramento di reti con un numero elevato di strati. La loro principale innovazione è l'introduzione dei "collegamenti residui", una soluzione semplice ma estremamente efficace per migliorare la propagazione del gradiente e preservare l'informazione attraverso strati profondi. Questo approccio ha consentito di costruire reti significativamente più profonde rispetto al passato, ottenendo prestazioni straordinarie in compiti di visione artificiale e altre applicazioni.

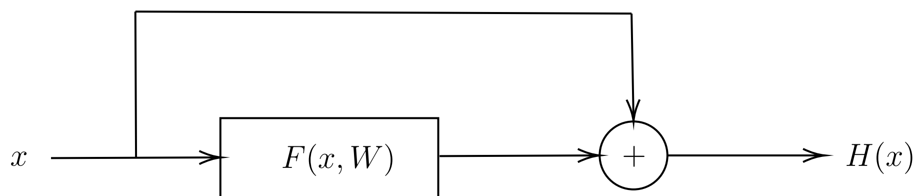

Figura 2.12: Un Blocco Residuo, unità fondamentale delle ResNets

La caratteristica distintiva delle ResNet è l'aggiunta di collegamenti diretti tra gli strati di una rete, che bypassano uno o più strati intermedi (residual connectons, per l'appunto). Un blocco residuo, l'elemento base di una ResNet, può essere descritto come segue:

$$H(x) = F(x, W) + x,$$

dove:

- x rappresenta l'input al blocco
- $F(x, W)$ è la trasformazione appresa dai pesi W del blocco residuo (tipicamente una composizione di convoluzioni, normalizzazioni e funzioni di attivazione)
- $H(x)$ è l'output del blocco residuo

In sostanza, il termine x viene sommato direttamente all'output di $F(x, W)$, creando un percorso diretto per l'informazione.

Questa struttura consente al modello di apprendere una funzione residua $F(x, W)$ piuttosto che la mappatura completa $H(x)$, rendendo l'ottimizzazione più semplice.

Le ResNet risolvono il problema del Vanishing Gradient sfruttando i collegamenti residui, che consentono al gradiente di propagarsi direttamente attraverso i collegamenti bypass ("skip connections"), senza essere ridotto dalle trasformazioni non lineari di ogni strato. In particolare, i collegamenti residui garantiscono che:

$$\frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial x_{l+1}} + \frac{\partial F(x_l, W)}{\partial x_l},$$

dove L è la funzione di perdita e x_l è l'output dello strato l, la presenza del termine $\frac{\partial L}{\partial x_{l+1}}$ assicura che il gradiente non si riduca drasticamente, anche in reti molto profonde.

Le reti residuali hanno introdotto una serie di innovazioni che hanno trasformato il panorama delle reti neurali profonde:

- **Addestramento di Reti estremamente profonde**: Con le ResNet, è stato possibile addestrare reti con centinaia (e persino migliaia) di strati senza incorrere nei problemi tipici delle DNN
- **Facilità di Ottimizzazione**: La formulazione residua rende più semplice apprendere mappature identitarie o quasi identitarie, riducendo il rischio di overfitting e migliorando la generalizzazione
- **Prestazioni Superiori**: Le ResNet hanno stabilito nuovi standard in molti compiti di visione artificiale, tra cui il riconoscimento di immagini (ImageNet) e la segmentazione semantica

Nonostante i loro vantaggi, le reti residuali presentano alcune limitazioni che hanno portato alla ricerca di soluzioni più avanzate:

1. **Complessità dei Parametri**: Le ResNet richiedono un numero elevato di parametri per rappresentare reti molto profonde, aumentando significativamente i costi computazionali e la memoria necessaria per l'addestramento
2. **Rappresentazione Discreta**: La struttura delle ResNet si basa su una propagazione discreta attraverso gli strati. Questo approccio non è ideale per modellare sistemi dinamici continui, dove è necessario descrivere l'evoluzione dell'informazione in modo fluido nel tempo o nello spazio
3. **Rigidità Strutturale**: Sebbene i collegamenti residui migliorino la propagazione del gradiente, il design rigido delle ResNet non è sempre adatto a dati complessi o a problemi che richiedono adattamenti dinamici del modello
4. **Problemi di Overfitting**: In alcuni casi, reti molto profonde possono sovradattarsi ai dati di addestramento, specialmente in presenza di dataset limitati o rumorosi. Questo fenomeno è amplificato dall'aumento del numero di parametri

Le reti residuali hanno rivoluzionato il campo del deep learning, rendendo possibile l'addestramento di reti estremamente profonde e migliorando significativamente le prestazioni in molti compiti. Tuttavia, le loro limitazioni hanno stimolato lo sviluppo di approcci più avanzati, come le Neural ODEs (che verranno trattate nel Capitolo 3), che combinano l'efficienza delle ResNet con la capacità di modellare sistemi continui. Questo collegamento rappresenta un importante passo avanti verso la comprensione e l'applicazione delle reti neurali in contesti sempre più complessi.

Capitolo 3

Neural Ordinary Differential Equations

3.1 Un Ponte tra il Discreto e il Continuo

Come già introdotto in precedenza, le reti neurali residuali (ResNet) hanno rivoluzionato il deep learning grazie all'introduzione dei collegamenti di salto che facilitano la propagazione del gradiente in reti estremamente profonde. Un aspetto fondamentale di questa architettura è il modo in cui viene formulato l'aggiornamento dello stato in ciascun blocco residuo. In termini matematici, se indichiamo con h_k lo stato del network al livello k, l'operazione di aggiornamento avviene secondo la relazione

$$h_{k+1} = h_k + f(h_k, \theta_k),$$

dove $f(h_k, \theta_k)$ rappresenta la trasformazione appresa (tipicamente una combinazione di convoluzione, normalizzazione e funzioni di attivazione) e θ_k i relativi parametri. Questa formula ricorda fortemente il metodo di integrazione esplicita noto come **metodo di Eulero** per la soluzione numerica di equazioni differenziali ordinarie (ODE). Infatti, considerando una ODE della forma

$$\frac{dh(t)}{dt} = f(h(t), t),$$

la discretizzazione con il metodo di Eulero, applicata con un passo temporale Δt, porta all'aggiornamento:

$$h(t + \Delta t) = h(t) + \Delta t \, f(h(t), t).$$

Se si assume, per semplicità, $\Delta t = 1$ o lo si incorpora implicitamente nei parametri, la relazione diventa analogamente:

$$h_{k+1} = h_k + f(h_k, t_k),$$

che è identica alla struttura della ResNet. Questa osservazione suggerisce che, al crescere del numero di strati, una ResNet può essere vista come l'approssimazione discreta di un sistema dinamico continuo.

Il metodo di Eulero, pur essendo intuitivo e computazionalmente semplice, è un metodo di integrazione "al primo ordine" che, pertanto, offre una precisione molto limitata. Per problemi in cui l'accuratezza è cruciale, la comunità scientifica ha sviluppato metodi di ordine superiore, tra cui i **metodi di Runge-Kutta** [14]. Un esempio classico è il metodo di Runge-Kutta a 4 stadi (RK4), che combina diverse valutazioni della funzione f in punti intermedi per ottenere una stima dell'integrazione molto più accurata.

Parallelamente, metodi adattivi come il **Dormand-Prince** [7] modulano dinamicamente il passo Δt in funzione della variazione locale della soluzione. Questi metodi sono in grado di bilanciare in modo ottimale l'accuratezza con il costo computazionale, aumentando lo step in regioni di variazione lenta e riducendolo in quelle in cui il sistema cambia rapidamente. L'adozione di tali metodi nei solver delle Neural ODEs permette di gestire in maniera efficiente dinamiche molto eterogenee, garantendo al contempo precisione e stabilità.

L'analogia tra l'aggiornamento in una ResNet e il metodo di Eulero non è una mera coincidenza: essa ha portato alla concezione delle **Neural ODEs**, dove la trasformazione dei dati lungo la "profondità" della rete viene formulata come la soluzione continua di una ODE. In questo paradigma, l'evoluzione dello stato $h(t)$ è descritta dall'equazione differenziale

$$\frac{dh(t)}{dt} = f(h(t), t, \theta),$$

e la soluzione al tempo finale T si ottiene mediante l'integrazione

$$h(T) = h(0) + \int_0^T f(h(t), t, \theta)\, dt.$$

Come già espresso nella Sezione precedente 3.1, il passaggio a una rappresentazione continua comporta numerosi vantaggi, tra cui la possibilità di utilizzare metodi di integrazione avanzati (come quelli introdotti in precedenza) per adattare dinamicamente il numero di step in base alla complessità locale della dinamica. Inoltre, questo approccio ha introdotto il concetto di "profondità continua", dove non è più necessario predefinire un numero fisso di strati, ma si può lasciare che il solver ODE stabilisca automaticamente la granularità necessaria per raggiungere una certa precisione.

Un ulteriore aspetto innovativo delle Neural ODEs riguarda il calcolo dei gradienti durante l'ottimizzazione. Utilizzando il **adjoint sensitivity method**, è possibile calcolare in maniera efficiente i gradienti rispetto ai parametri θ senza dover memorizzare l'intera traiettoria $h(t)$, tecnica che permette di ridurre significativamente il carico di memoria necessaria durante l'addestramento.

Questo cambio di prospettiva, supportato dalla ricca letteratura riguardo ai metodi numerici per risolvere ODEs, ha aperto la strada a modelli di deep learning in grado di descrivere dinamiche complesse con una flessibilità e una precisione maggiorate.

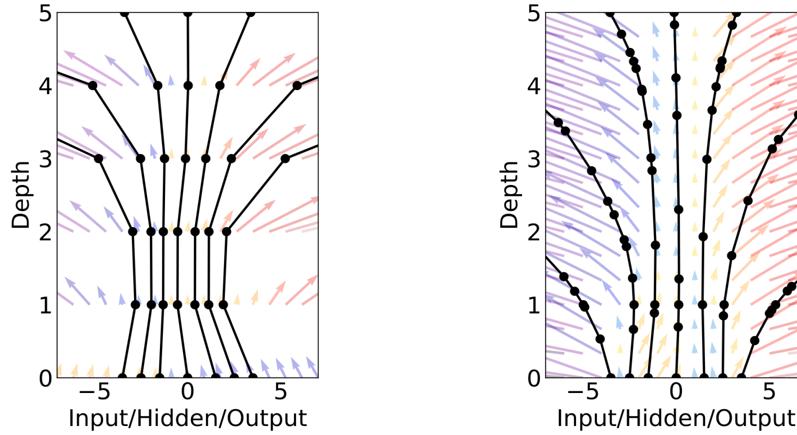

Figura 3.1: Confronto tra l'**architettura discreta** di una ResNet (a sinistra; si può vedere la sequenza discreta di trasformazioni finite) e l'**architettura continua** di una NODE (a destra; si può vedere il campo vettoriale che applica trasformazioni continue) [36]

3.2 Introduzione alle Neural ODEs

Come già anticipato, le Neural Ordinary Differential Equations (Neural ODEs), introdotte da Chen et al. nel 2018 [36], rappresentano un'estensione delle reti neurali tradizionali che riformula la propagazione delle informazioni come un problema di integrazione di equazioni differenziali ordinarie (ODEs). Questo approccio fornisce un framework matematico più flessibile, nonché più studiato storicamente, per modellare sistemi continui, permettendo di superare alcune delle limitazioni delle architetture profonde convenzionali. Nelle reti neurali standard, la trasformazione tra gli strati è descritta da una funzione discreta:

$$h_{t+1} = h_t + f(h_t, \theta),$$

dove h_t rappresenta lo stato latente dei dati all'interno della rete e $f(h_t, \theta)$ è una funzione parametrizzata dai pesi θ. Le Neural ODEs estendono questa idea definendo la dinamica dello stato latente come una ODE continua:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta),$$

dove $h(t)$ è lo stato latente in un tempo continuo t, e $f(h, t, \theta)$ è una funzione appresa dalla rete neurale. Per ottenere il valore di $h(T)$ a partire da $h(0)$, si utilizza un integratore numerico:

$$h(T) = h(0) + \int_0^T f(h(t), t, \theta) dt.$$

L'operazione di propagazione in avanti viene quindi realizzata tramite un solver ODE:

$$h(T) = \text{ODEint}(f, h(0), T, \theta),$$

dove ODEint è un metodo numerico di integrazione, come Euler, Runge-Kutta o metodi adattivi.

Per ottimizzare i parametri θ, si minimizza una funzione di perdita L che dipende dallo stato finale $h(T)$. Poiché $h(T)$ è ottenuto integrando la ODE, esso dipende implicitamente da θ. È quindi necessario calcolare il gradiente $\frac{dL}{d\theta}$.

Consideriamo la soluzione della ODE:

$$h(T) = h(0) + \int_0^T f(h(t), t, \theta)\, dt.$$

Differenziando rispetto a θ (usando la regola della catena) otteniamo:

$$\frac{dh(T)}{d\theta} = \int_0^T \left[\frac{\partial f}{\partial h}(h(t), t, \theta)\frac{dh(t)}{d\theta} + \frac{\partial f}{\partial \theta}(h(t), t, \theta)\right] dt.$$

Definiamo la sensibilità $s(t) = \frac{dh(t)}{d\theta}$, che soddisfa la ODE lineare:

$$\frac{ds(t)}{dt} = \frac{\partial f}{\partial h}(h(t), t, \theta)\, s(t) + \frac{\partial f}{\partial \theta}(h(t), t, \theta), \quad s(0) = 0.$$

La perdita dipende dallo stato finale, dunque:

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial h(T)}\frac{dh(T)}{d\theta} = a(T)^T\, s(T),$$

dove abbiamo definito il vettore aggiunto $a(T) = \frac{\partial L}{\partial h(T)}$.

Invece di calcolare direttamente $s(T)$, è possibile derivare una ODE per $a(t) = \frac{\partial L}{\partial h(t)}$ che soddisfa:

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f}{\partial h}(h(t), t, \theta),$$

con condizione finale $a(T) = \frac{\partial L}{\partial h(T)}$.

Questo passaggio si ottiene applicando una versione continua della regola della catena e integrando per parti nel tempo (si possono trovare tutti passaggi nell'appendice A.2); in sostanza, si dimostra che la variazione della loss lungo la traiettoria è governata dalla dinamica inversa data sopra. Utilizzando la relazione tra $s(t)$ e $a(t)$, si può dimostrare che il gradiente rispetto a θ è:

$$\frac{dL}{d\theta} = -\int_T^0 a(t)^T\, \frac{\partial f}{\partial \theta}(h(t), t, \theta)\, dt.$$

Alternativamente, cambiando il verso di integrazione:

$$\frac{dL}{d\theta} = \int_0^T a(t)^T\, \frac{\partial f}{\partial \theta}(h(t), t, \theta)\, dt.$$

Questo risultato, noto come **adjoint sensitivity method**, consente di calcolare i gradienti senza dover memorizzare tutti gli stati intermedi $h(t)$ durante la propagazione in avanti.

L'introduzione delle Neural ODEs ha portato diversi vantaggi:

- **Continuity in Time:** La formulazione continua consente di rappresentare dinamiche che evolvono in tempo reale senza vincoli di discretizzazione. Questo approccio permette di modellare sistemi che cambiano in modo non uniforme, eliminando la necessità di definire un numero fisso di strati. Ad esempio, in applicazioni fisiche o biologiche, il tempo non procede a passi uguali e la continuità temporale offre una rappresentazione più aderente alla realtà. Inoltre, la continuità permette di derivare formulazioni analitiche (come l'integrale della derivata) che possono essere sfruttate per garantire stabilità e accuratezza nell'addestramento.

- **Adaptive Computation:** I solver numerici utilizzati per integrare le ODE (come i metodi Runge-Kutta adattivi o Dormand-Prince) modulano dinamicamente il numero di valutazioni necessarie in base alla complessità locale della dinamica. Ciò consente una regolazione automatica della complessità computazionale: aree con variazioni lente vengono integrate con pochi step, mentre zone di rapido cambiamento richiedono step più piccoli per mantenere l'accuratezza. Questa caratteristica rende le Neural ODEs particolarmente efficienti in presenza di dati eterogenei, offrendo una migliore allocazione delle risorse computazionali rispetto agli approcci standard.

- **Applicazioni in Sistemi Dinamici:** Le Neural ODEs sono state impiegate con successo in diversi ambiti, quali la fisica, la biologia computazionale e la modellazione finanziaria, per apprendere dinamiche complesse. In particolare, grazie alla formulazione continua, è possibile catturare comportamenti emergenti e transizioni di fase che sarebbero difficili da rappresentare con architetture a strati fissi. Inoltre, la possibilità di integrare informazioni su variazioni di tempo in maniera naturale permette di utilizzare modelli derivati dalla teoria dei sistemi dinamici, come la linearizzazione locale e l'analisi della stabilità, per interpretare e validare il comportamento del modello.

- **Eliminazione della Necessità di Dati Regolari:** Le Neural ODEs non richiedono che i dati siano estratti a intervalli temporali regolari. Grazie alla loro formulazione continua, il modello integra direttamente i dati osservati, indipendentemente dalla frequenza di campionamento. Questa caratteristica consente di gestire facilmente dati estratti irregolarmente, evitando la necessità di interpolazioni o resampling, e rende le Neural ODEs particolarmente adatte a contesti in cui le misurazioni sono sparse o non uniformi.

Un'applicazione notevole è nel campo delle *Normalizing Flows*, dove l'integrazione continua attraverso una ODE consente di modellare trasformazioni invertibili. In questo contesto, la formula del cambio di variabile si semplifica, sostituendo il calcolo del determinante dello Jacobiano con un'operazione di traccia (tramite il teorema del cambiamento istantaneo di variabili), permettendo così di gestire modelli con un elevato numero di unità nascoste in modo più efficiente [45], [39].
Nonostante i vantaggi appena riportati, il modello presenta alcune limitazioni:

1. **Costo Computazionale Elevato:** Sebbene il metodo dell'adjoint sensitivity consenta di risparmiare memoria, l'integrazione numerica delle ODE richiede comunque molte valutazioni della funzione dinamica $f(h, t, \theta)$, rendendo i solver ODE

più lenti rispetto alla propagazione diretta nelle reti neurali standard. Questo overhead può diventare critico in applicazioni in tempo reale o in contesti con limiti computazionali stringenti

2. **Problemi di Stiffness:** Alcuni sistemi dinamici, in particolare quelli caratterizzati da scale temporali molto diverse, presentano problemi di stiffness. Questi sistemi richiedono l'uso di metodi numerici avanzati (come integratori impliciti o metodi specifici per ODE stiff) per garantire la stabilità della soluzione. La scelta e l'implementazione di tali metodi possono complicare l'addestramento e influire sulle prestazioni complessive del modello

3. **Difficoltà di Interpretazione:** La funzione dinamica $f(h, t, \theta)$ è solitamente rappresentata da una rete neurale, il che implica che essa agisce come una "scatola nera". La mancanza di trasparenza nella scelta e nell'interpretazione della struttura di f rende difficile comprendere come il modello stia rappresentando le dinamiche sottostanti. Questo aspetto può limitare l'utilizzo delle Neural ODEs in contesti in cui l'interpretabilità del modello sono fondamentali, come in ambiti medici o finanziari

3.3 Confronto con i Modelli Residuali

Le Residual Networks (ResNet) e le Neural Ordinary Differential Equations (Neural ODEs) condividono la stessa idea di propagazione dell'informazione tramite trasformazioni residue, ma differiscono per la loro formulazione matematica e computazionale. Mentre le ResNet sono costruite come una serie discreta di trasformazioni attraverso blocchi residui, le Neural ODEs modellano la trasformazione come un sistema continuo regolato da equazioni differenziali ordinarie (ODE). Questo approccio introduce vantaggi e svantaggi a seconda dell'applicazione considerata.

Di seguito, nella tabella 3.1, si riporta un confronto riassuntivo tra le due architetture in termini di modellazione, efficienza computazionale, utilizzo della memoria e ambiti di applicazione.

Caratteristica	ResNet	Neural ODEs
Modellazione	Discreta (blocchi residui)	Continua (solver ODE)
Efficienza computazionale	Elevata nei forward pass	Maggiore costo computazionale per il solver ODE
Numero di parametri	Fisso, determinato dagli strati	Variabile, adattabile tramite il solver ODE
Robustezza	Sensibile alla profondità	Adattabile grazie ai solver adattivi
Memoria	Richiede salvataggio di molteplici attivazioni intermedie	Bassa, il metodo adjoint riduce l'uso della memoria
Back-propagation	Standard con calcolo esplicito dei gradienti	Usa il metodo adjoint, con possibili problemi di stabilità numerica
Applicazioni	Visione artificiale, classificazione immagini	Modellazione di sistemi dinamici, traiettorie fisiche
Limitazioni	Necessita di molti strati per alte prestazioni	Maggior costo computazionale, difficile da scalare su dataset molto grandi

Tabella 3.1: Confronto tra ResNet e Neural ODEs

In breve, le ResNet rappresentano un'architettura efficiente per compiti di classificazione e visione artificiale, dove l'ottimizzazione del numero di strati discreti consente di ottenere prestazioni elevate con costi computazionali contenuti. D'altro canto, le Neural ODEs risultano particolarmente adatte alla modellazione di sistemi dinamici continui, grazie alla loro capacità di apprendere trasformazioni temporali in uno spazio continuo. Tuttavia, il loro maggiore costo computazionale e la complessità nella propagazione del gradiente ne limitano la scalabilità in contesti standard di deep learning.

3.4 Applicazioni delle Neural ODEs

La scelta di utilizzare le Neural ODEs in determinate applicazioni piuttosto che in altre è motivata dalla loro capacità di modellare fenomeni continui in modo naturale, adattare dinamicamente la complessità computazionale e ridurre il numero di parametri attraverso l'uso di solver numerici.

A differenza delle reti standard, che operano su trasformazioni discrete tra strati successivi, le Neural ODEs descrivono il cambiamento dello stato latente dei dati come un sistema differenziale continuo, imparandone a riprodurre il campo vettoriale completo. Questo approccio consente di modellare con maggiore precisione sistemi dinamici complessi, risultando particolarmente efficace in ambiti in cui la continuità temporale o spaziale gioca un ruolo chiave.

Di seguito si elencano alcune delle principali applicazioni delle Neural ODEs:

1. **Modelli Generativi e Normalizing Flows**: le Neural ODEs sono state utilizzate nei modelli generativi basati su normalizing flows, migliorando la capacità di apprendere distribuzioni di probabilità complesse con un minor numero di parametri rispetto ai metodi tradizionali, [36]
2. **Elaborazione di Serie Temporali**: le Neural ODEs hanno dimostrato una particolare efficacia nell'analisi di dati temporali irregolari, come in ambito medico o finanziario, dove i dati non sono raccolti a intervalli uniformi, [45]
3. **Modelli di Apprendimento per la Dinamica dei Fluidi**: le Neural ODEs sono state impiegate per modellare dinamiche di fluidi, offrendo un'alternativa più efficiente rispetto ai metodi numerici tradizionali nella simulazione di equazioni differenziali alle derivate parziali, [49]
4. **Apprendimento di Rappresentazioni Latenti in Visione Artificiale**: le Neural ODEs sono state applicate alla visione artificiale per apprendere rappresentazioni latenti più efficienti in modelli di classificazione e segmentazione delle immagini, [40]
5. **Modellazione di Sistemi Biologici**: le Neural ODEs hanno trovato applicazioni nella modellazione di processi biologici, come la crescita cellulare e la dinamica delle popolazioni, permettendo di incorporare vincoli fisici e biologici direttamente nel modello, [56]
6. **Fisica Computazionale e Modellazione del Clima**: le Neural ODEs sono state utilizzate per modellare sistemi fisici complessi, come la previsione climatica e la simulazione di processi atmosferici, dove l'approccio continuo consente di ottenere previsioni più stabili e accurate, [61]
7. **Farmacocinetica e Farmacodinamica**: le Neural ODEs hanno trovato applicazioni anche nel settore farmaceutico per la modellazione della farmacocinetica e della farmacodinamica. In questo contesto, esse permettono di rappresentare in modo continuo la dinamica della concentrazione del farmaco e l'effetto terapeutico o tossico nel tempo, integrando vincoli meccanicistici e biologici per migliorare la predizione dell'efficacia e della sicurezza dei trattamenti, [59]

3.5 Oltre le NODEs: Reti Neurali "Physics Informed"

Le Physics-Informed Neural Networks (PINNs) rappresentano un framework innovativo per l'apprendimento automatico di sistemi dinamici complessi, combinando la flessibilità delle reti neurali con la struttura imposta dalle equazioni differenziali governanti. Introdotte da Raissi, Perdikaris e Karniadakis nel 2019 [44], le PINNs consentono di incorporare direttamente vincoli fisici nel processo di apprendimento, migliorando la capacità della rete di generalizzare e di apprendere soluzioni coerenti con le leggi della fisica.

L'idea chiave delle PINNs è quella di apprendere una funzione di soluzione $u(x, t)$ per un sistema dinamico espresso come un'equazione differenziale parziale (PDE) o un sistema di equazioni differenziali ordinarie (ODE). Consideriamo un problema generale:

$$\mathcal{F}(u; \lambda) = 0, \quad x \in \Omega, \quad t \in [0, T],$$

dove $\mathcal{F}$ rappresenta un operatore differenziale che governa la dinamica del sistema e λ è un insieme di parametri fisici.

L'approssimazione della soluzione $u(x, t)$ viene effettuata tramite una rete neurale: $\hat{u}(x, t; \theta)$, in cui θ rappresenta i parametri della rete. A differenza delle reti neurali standard, le PINNs non si basano esclusivamente su dati empirici, ma vincolano la rete a soddisfare la PDE durante l'addestramento.

La funzione di perdita delle PINNs è definita come:

$$L(\theta) = L_{data} + L_{PDE},$$

dove:

- L_{data} è un termine di errore basato sui dati di osservazione, tipicamente la norma quadratica dell'errore tra la soluzione predetta e i valori misurati

- L_{PDE} è un termine che penalizza la violazione dell'equazione differenziale, definito come:

$$L_{PDE} = \sum_{i=1}^{N_f} |\mathcal{F}(\hat{u}(x_i, t_i; \theta))|^2,$$

dove $\{x_i, t_i\}$ sono punti campionati nel dominio della PDE.

L'ottimizzazione della rete avviene minimizzando $L(\theta)$ con metodi classici di discesa del gradiente, combinati con tecniche di differenziazione automatica per calcolare i termini derivativi imposti dalla PDE.

Le PINNs trovano applicazione in numerosi contesti in cui i sistemi dinamici sono descritti da equazioni differenziali. Alcuni esempi includono:

1. **Sistemi Hamiltoniani e Lagrangiani**: Le PINNs possono apprendere direttamente funzioni Hamiltoniane e Lagrangiane, permettendo la modellazione di sistemi fisici conservativi, [48].

2. **Fluidodinamica e Navier-Stokes**: Le PINNs possono apprendere soluzioni delle equazioni di Navier-Stokes, con applicazioni nell'aerodinamica e nella simulazione di fluidi, [44].

3. **Meccanica Quantistica**: Le PINNs possono essere utilizzate per risolvere equazioni differenziali quantistiche, come l'equazione di Schrödinger, [63].

4. **Modellazione di Sistemi Biologici**: Le PINNs vengono applicate alla simulazione di dinamiche biologiche, come la propagazione di onde elettromagnetiche nel tessuto cerebrale, [53].

L'introduzione delle PINNs ha portato diverse innovazioni:

- **Apprendimento Guidato dalla Fisica**: Le PINNs eliminano la necessità di grandi quantità di dati di addestramento, sfruttando direttamente equazioni fisiche per migliorare l'apprendimento
- **Generalizzazione Migliorata**: Il vincolo imposto dalle PDE permette alle PINNs di essere più robuste rispetto ai metodi puramente data-driven
- **Capacità di Inferenza su Parametri Sconosciuti**: Le PINNs possono essere utilizzate per identificare parametri ignoti nei modelli fisici attraverso problemi inversi

Nonostante questi vantaggi, le PINNs presentano alcune limitazioni già incontrate in precedenza, quali il Costo Computazionale Elevato, la difficile Scalabilità di questo tipo di modelli e la Difficoltà nella scelta dell'architettura di rete più appropriata per la risoluzione di un problema fisico specifico.

Inoltre, è importante sottolineare che, mentre le PINNs costituiscono un framework in cui si incorporano nei termini della funzione di perdita vincoli derivanti dalle leggi fisiche (ad esempio, equazioni differenziali parziali o ordinarie), le Neural ODEs, invece, rappresentano un approccio in cui la trasformazione dei dati tra gli strati è definita in modo continuo tramite l'integrazione di una ODE, sostituendo la classica struttura a strati discreti. In altre parole, PINNs e Neural ODEs affrontano problematiche simili e possono coesistere in modelli ibridi, ma non sono concettualmente equivalenti.

3.6 Reti Neurali Hamiltoniane

Le Hamiltonian Neural Networks (HNNs), introdotte da Greydanus et al. nel 2019 [41], rappresentano un'estensione delle Neural ODEs, che impone il Formalismo Hamiltoniano nella dinamica appresa. Questo approccio consente di modellare sistemi fisici conservativi, garantendo la preservazione di proprietà fondamentali come l'energia totale e il momento, migliorando la stabilità e la capacità di generalizzazione del modello.

Nel Formalismo Hamiltoniano, la dinamica di un sistema è descritta in termini delle coordinate generalizzate q_i e dei momenti coniugati p_i. L'evoluzione temporale del sistema è governata dalle equazioni di Hamilton-Jacobi:

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \quad \dot{p}_i = -\frac{\partial H}{\partial q_i},$$

dove $H(q, p)$ è la funzione Hamiltoniana, che rappresenta l'energia totale del sistema.

Le HNN apprendono direttamente la funzione $H_\theta(q, p)$ parametrizzata da una rete neurale con pesi θ. La dinamica è quindi derivata automaticamente attraverso la differenziazione della rete:

$$\dot{q}_i = \frac{\partial H_\theta}{\partial p_i}, \quad \dot{p}_i = -\frac{\partial H_\theta}{\partial q_i}.$$

Questa formulazione garantisce che le traiettorie apprese siano intrinsecamente conservative, evitando la deriva energetica che può verificarsi nei modelli tradizionali basati su Neural ODEs.

Le HNNs offrono diversi vantaggi rispetto alle Neural ODEs:

- **Conservazione dell'Energia**: A differenza delle Neural ODEs standard, che apprendono la dinamica direttamente dallo spazio delle fasi, le HNN garantiscono automaticamente la conservazione dell'energia.

- **Generalizzazione Migliorata**: Poiché le equazioni di Hamilton derivano da principi fisici universali, le HNNs possono apprendere dinamiche con meno dati rispetto alle Neural ODEs convenzionali.

- **Migliore Stabilità Numerica**: Le traiettorie apprese rispettano la struttura geometrica dello spazio delle fasi, migliorando la stabilità nelle simulazioni a lungo termine.

Le HNNs trovano applicazione in diversi ambiti della fisica computazionale e dell'ingegneria:

- **Meccanica Classica**: Modellazione precisa di sistemi conservativi come pendoli, oscillatori armonici e sistemi multi-corpo, [41].

- **Astrofisica e Orbite Planetarie**: Simulazioni accurate di moti orbitali evitando errori numerici legati alla dissipazione energetica, [50].

- **Dinamica dei Sistemi Quantistici**: Applicazione alle equazioni di Schrödinger e a modelli di simulazione quantistica, [58].

- **Robotica e Sistemi di Controllo**: Progettazione di sistemi robotici basati su principi di conservazione dell'energia e dell'impulso, [51].

Nonostante i vantaggi, le HNNs presentano alcune limitazioni:

1. **Difficoltà nel Modellare Sistemi Dissipativi**: Poiché il formalismo hamiltoniano è applicabile solo a sistemi conservativi, le HNN non sono direttamente adatte per fenomeni con attrito o dissipazione. Per ovviare a questo problema sono già state sviluppate nuovi modelli simili [65].

2. **Complessità Computazionale**: L'apprendimento della funzione di Hamiltoniana può essere più oneroso rispetto ai modelli standard, specialmente per sistemi con molteplici gradi di libertà.

3. **Richiesta di Dati con Informazione di Momento**: L'input della rete deve contenere sia le coordinate che i momenti generalizzati, non sempre facilmente disponibili in generale, ma che potrebbe richiedere misurazioni aggiuntive in specifici contesti sperimentali, [47].

3.7 Reti Neurali Lagrangiane

Le Lagrangian Neural Networks (LNNs), introdotte da Cranmer et al. nel 2020 [47], rappresentano anch'esse un'evoluzione delle Neural ODEs con l'obiettivo di modellare sistemi dinamici, rispettando esplicitamente i principi della Meccanica Lagrangiana. Questo approccio garantisce una conservazione più accurata delle leggi fisiche fondamentali e migliora la capacità della rete di generalizzare su traiettorie fisicamente plausibili.

Il Formalismo Lagrangiano descrive la dinamica di un sistema attraverso una funzione scalare, la Lagrangiana $\mathcal{L}(q, \dot{q})$, che dipende dalle coordinate generalizzate q e dalle loro derivate temporali $\dot{q}$. La dinamica del sistema è determinata dalle equazioni di Eulero-Lagrange:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}}\right) - \frac{\partial \mathcal{L}}{\partial q} = 0.$$

Le LNNs apprendono direttamente la funzione Lagrangiana $\mathcal{L}_\theta(q, \dot{q})$ utilizzando una rete neurale parametrizzata da θ. Una volta appresa $\mathcal{L}_\theta$, le equazioni di Eulero-Lagrange possono essere risolte numericamente per ottenere la traiettoria dinamica del sistema:

$$\ddot{q} = f_\theta(q, \dot{q}).$$

Questa formulazione consente di modellare la dinamica in modo intrinsecamente conservativo, evitando problemi come la perdita (o l'acquisizione) di energia che può verificarsi nelle Neural ODEs standard.

Le LNNs introducono numerosi vantaggi rispetto alle Neural ODEs:

- **Conservazione delle Quantità Fisiche**: A differenza delle Neural ODEs standard, che imparano direttamente la funzione di evoluzione dello stato, le LNNs garantiscono la conservazione delle leggi di moto derivate dal principio di minima azione
- **Migliore Generalizzazione**: Poiché le equazioni di Eulero-Lagrange derivano da principi fisici universali, le LNNs possono apprendere dinamiche con meno dati di addestramento rispetto alle Neural ODEs tradizionali
- **Riduzione dell'Overfitting**: L'imposizione di vincoli fisici riduce la capacità della rete di apprendere dinamiche spurie, migliorando la capacità di generalizzazione
- **Simmetrie e Leggi di Conservazione**: L'uso della meccanica Lagrangiana facilita l'apprendimento di sistemi con invarianti di moto, come la conservazione dell'energia e del momento angolare

Le LNNs sono particolarmente adatte per modellare sistemi fisici complessi, con applicazioni che spaziano dalla fisica teorica all'ingegneria:

- **Dinamica dei Sistemi Meccanici**: Modellazione accurata di pendoli, sistemi multi-corpo e meccanismi robotici, [47].
- **Simulazioni in Astrofisica**: Previsione del moto planetario e interazioni gravitazionali con elevata precisione, [57].
- **Controllo nei Sistemi Cibernetici**: Applicazioni in robotica e biomeccanica per migliorare la simulazione e il controllo di strutture articolate, [69] [43].
- **Modellazione di Sistemi Biologici**: Analisi della locomozione animale e dell'evoluzione delle traiettorie di movimento, [54].

Nonostante i loro vantaggi, le LNNs presentano alcune limitazioni:

1. **Difficoltà nell'Apprendimento di Sistemi Dissipativi**: Poiché il formalismo Lagrangiano è adatto principalmente a sistemi conservativi, le LNNs possono avere difficoltà a modellare fenomeni dissipativi come l'attrito o la perdita di energia
2. **Complessità Computazionale**: Il calcolo delle equazioni di Eulero-Lagrange per reti neurali profonde può risultare computazionalmente oneroso
3. **Difficoltà di Ottimizzazione**: L'addestramento delle LNNs richiede una formulazione numericamente stabile dell'integrazione delle equazioni differenziali, il che può rendere il processo di ottimizzazione più complesso rispetto alle Neural ODEs tradizionali.

Le Lagrangian Neural Networks rappresentano un significativo miglioramento rispetto alle Neural ODEs, consentendo di modellare sistemi dinamici in modo più fisicamente consistente. La loro capacità di rispettare i principi della meccanica Lagrangiana e di non avere necessità di coordinate specifiche (feature non presente, invece, nelle HNNs) le rende particolarmente adatte a problemi di dinamica meccanica, astrofisica e controllo robotico.

Capitolo 4

Implementazione Pratica e Confronto tra Modelli

4.1 Architetture Utilizzate

L'obiettivo di questo capitolo è introdurre e descrivere le task sperimentali utilizzate per confrontare diverse architetture di rete neurale. In particolare, verranno analizzate le prestazioni di un Multi-Layer Perceptron (MLP), di una Neural Ordinary Differential Equation (Neural ODE), di una Hamiltonian Neural Network (HNN) e di una Lagrangian Neural Network (LNN) in diversi contesti. Le task scelte riguardano problemi esemplificativi di classificazione e di modellazione di sistemi dinamici, consentendo di valutare la capacità delle diverse reti di apprendere strutture temporali e spaziali nascoste nei dati. Le simulazioni sono state implementate in Python, utilizzando la libreria "PyTorch" per la definizione e l'addestramento delle reti neurali, e "torchdiffeq" ([36]) per la risoluzione numerica delle equazioni differenziali nei modelli basati su dinamiche continue. L'addestramento avverrà tramite l'algortimo **SGD** (Stochastic Gradient Descend) e ottimizzatore **Adam**, con una funzione di costo appropriata per ogni task (CrossEntropyLoss per la prima task, MSELoss per le altre due).

Task Sperimentali

1. **Classificazione di Immagini**: la prima task consiste nella classificazione di immagini del dataset MNIST, un benchmark standard per valutare le prestazioni di reti neurali in problemi di riconoscimento visivo (in questo caso, si tratta di numeri scritti "a mano libera"). Entrambe le architetture riceveranno in input le immagini in scala di grigi di dimensione $28x28$ pixel, appiattite in vettori di 784 elementi. L'obiettivo sarà predire la classe corretta (ovvero il numero rappresentato, da 0 a 9) per ciascuna immagine.

   Il confronto tra MLP e Neural ODE in questa task permetterà di valutare:

   - *Capacità di apprendimento su dati statici*
   - *Efficienza dell'ottimizzazione*
   - *Generalizzazione su dati mai visti*

Poiché la funzione di perdita utilizzata sarà la Cross-Entropy Loss, non sarà necessario includere un layer di SoftMax nell'output del modello, in quanto PyTorch integra questa operazione direttamente nella funzione di costo.

2. **Predizione di Traiettorie**: la seconda task riguarda la predizione di traiettorie dinamiche nel modello FitzHugh-Nagumo, un sistema non lineare utilizzato per descrivere l'attività neuronale ([4]). L'obiettivo dei modelli in esame, una volta conclusa la fase di addestramento, sarà quello di prevedere l'evoluzione futura dello stato del sistema, data unicamente la condizione iniziale. Questa task è particolarmente adatta per confrontare la capacità di un modello discreto (MLP) con un modello continuo (Neural ODE) di apprendere le dinamiche temporali dei dati. Gli aspetti valutati includeranno:
   - *Capacità di modellare dinamiche non lineari*
   - *Accuratezza nella previsione a lungo termine*
   - *Robustezza a perturbazioni nei dati di input*

3. **Modellazione di un Sistema Dinamico Conservativo**: oltre al modello di FitzHugh-Nagumo della sezione precedente, si considera anche la simulazione di un sistema dinamico conservativo, come un *pendolo semplice* o un *oscillatore armonico*. Questi sistemi sono caratterizzati dalla conservazione dell'energia e rappresentano un banco di prova ideale per valutare la capacità delle Neural ODEs rispetto alle Hamiltonian Neural Networks (HNNs) e alle Lagrangian Neural Networks (LNNs). Questa task consentirà di analizzare:
   - *Capacità di preservare le quantità conservate* (energia, momento, etc.)
   - *Stabilità della simulazione a lungo termine*
   - *Efficienza del modello nel rappresentare le leggi fisiche sottostanti*

Modelli Utilizzati

1. **Multi-Layer Perceptron (MLP)**
   L'MLP è un modello di rete neurale completamente connesso che trasforma l'input attraverso una serie di strati lineari con funzioni di attivazione non lineari. Per garantire un confronto equo, la rete sarà strutturata in modo da avere lo stesso numero di livelli e dimensioni dei layer degli altri modelli, differendo solo per il metodo con cui viene calcolata la trasformazione dello stato.

   Architettura del MLP:
   - *Input Layer*: 784 neuroni (per la prima task) o un numero appropriato per i sistemi dinamici (cioè il numero di coordinate necessarie per descrivere il sistema, o DoF)
   - *Hidden Layer*: Unico strato nascosto di 200 neuroni
   - *Output Layer*: 10 neuroni per la classificazione, stesso numero di neuroni dello strato di Input per la predizione di traiettorie dinamiche
   - *Funzione di attivazione*: Tanh per gli strati nascosti

2. **Neural Ordinary Differential Equation (NODE)**
   A differenza del MLP, le Neural ODEs modellano la trasformazione dello stato come un'equazione differenziale ordinaria risolta numericamente. Nella sua forma classica, il modello è composto da:

   - *Un blocco ODE*, che apprende la funzione dinamica dello stato latente,
   - *Un solver numerico*, che integra l'ODE nel tempo,
   - *Un classificatore finale*, che trasforma lo stato latente in un output interpretabile.

   Per la prima task è stata impiegata una variante denominata **NODEClassifier**, che integra la struttura classica della NODE con un **Encoder** e un **Decoder**. In questo schema, l'Encoder converte le immagini in una rappresentazione latente interpretabile dalla NODE, mentre il Decoder traduce l'output della NODE nelle classi di appartenenza per la classificazione. Per le altre due task di predizione, invece, è stata utilizzata la NODE standard. L'equazione della ODE verrà appresa tramite una rete neurale che rappresenta la funzione $f(h, t)$, mentre l'integrazione verrà effettuata con il solver `torchdiffeq.odeint`.

3. **Hamiltonian Neural Networks (HNN)**
   L'implementazione pratica delle Hamiltonian Neural Networks in Python sfrutta il framework PyTorch per apprendere direttamente la funzione Hamiltoniana a partire dallo stato del sistema. In particolare, la rete è definita come una sottoclasse di `torch.nn.Module` e si compone dei seguenti elementi:

   - *Un modulo MLP:* Questo componente apprende la funzione scalare Hamiltoniana $H(q, p)$ dai dati di input, rappresentando la dinamica energetica del sistema
   - *Costruzione della matrice symplettica J:* La rete genera una matrice J basata sulla dimensione degli input (che si assume pari, ovvero $2n$), in modo da imporre la struttura Hamiltoniana necessaria per rispettare le proprietà dei sistemi conservativi
   - *Calcolo del gradiente dell'Hamiltoniana:* Durante il forward pass, la rete calcola l'Hamiltoniana per ogni campione e utilizza l'autograd di PyTorch per ottenere il gradiente $\nabla_x H$, garantendo il tracciamento delle dipendenze necessarie
   - *Applicazione della struttura Hamiltoniana:* Il gradiente calcolato viene moltiplicato per la trasposizione della matrice J per derivare la dinamica temporale del sistema, ovvero $\dot{x} = J^\top \nabla_x H$. Questa operazione integra direttamente il principio di conservazione dell'energia nella dinamica appresa

   Inoltre, l'implementazione fornisce metodi ausiliari come `get_energy(x)` per ottenere il valore dell'energia del sistema, permettendo una verifica esplicita dei principi fisici fondamentali nei sistemi dinamici conservativi.

   Questa architettura è particolarmente adatta alla modellazione di sistemi in cui l'energia totale si conserva, come pendoli e oscillatori armonici ("toy model" preso in esame in questo lavoro).

4. **Lagrangian Neural Networks (LNN)**
   In pratica, le Lagrangian Neural Networks vengono implementate in PyTorch per apprendere direttamente la funzione scalare Lagrangiana $\mathcal{L}(q, \dot{q})$ a partire dallo stato del sistema. L'implementazione si basa su una sottoclasse di `torch.nn.Module` e comprende i seguenti elementi:

   - *Modulo di apprendimento della Lagrangiana:* Un Multi-Layer Perceptron (MLP) è utilizzato per approssimare la funzione Lagrangiana, fornendo una stima scalare per ogni input
   - *Calcolo dei gradienti e dell'Hessiano:* Durante il forward pass, per ogni singolo sample, tramite le funzionalità di `autograd` e `vmap` di PyTorch, si calcolano:
     - Il gradiente di $\mathcal{L}$, che fornisce $\frac{\partial \mathcal{L}}{\partial q}$ e $\frac{\partial \mathcal{L}}{\partial \dot{q}}$
     - L'Hessiano, dal quale vengono estratti gli elementi necessari, in particolare la derivata mista $\frac{\partial^2 \mathcal{L}}{\partial \dot{q} \partial q}$ e la seconda derivata $\frac{\partial^2 \mathcal{L}}{\partial \dot{q}^2}$
   - *Risoluzione delle equazioni di Eulero-Lagrange invertite:* Utilizzando le derivate calcolate, il modello determina l'accelerazione a secondo la relazione:

   $$\ddot{q} = \frac{\frac{\partial \mathcal{L}}{\partial q} - \frac{\partial^2 \mathcal{L}}{\partial \dot{q} \partial q} \dot{q}}{\frac{\partial^2 \mathcal{L}}{\partial \dot{q}^2} + \epsilon}$$

     dove $\dot{q}$ è la velocità estratta dallo stato e ϵ rappresenta un piccolo termine di regolarizzazione.
   - *Output del modello:* Lo stato incrementale restituito è la coppia $(\dot{q},\ \ddot{q})$, che rappresenta la dinamica del sistema

   In aggiunta, è implementato un metodo `get_energy` che calcola l'energia del sistema. Tale metodo determina il momento generalizzato $p = \frac{\partial \mathcal{L}}{\partial \dot{q}}$ e, conseguentemente, l'energia hamiltoniana $H = p\,v - \mathcal{L}$.

   Questa implementazione pratica delle Lagrangian Neural Networks consente di modellare in modo efficiente sistemi meccanici complessi, integrando direttamente i vincoli fisici e le dinamiche non lineari tramite il calcolo automatico delle derivate.

Nel capitolo successivo verranno descritti i dataset utilizzai per la realizzazione delle varie task sperimentali, e la loro generazione.

4.2 Dataset Utilizzati

Il processo di generazione e strutturazione dei dataset è un passaggio fondamentale per valutare in maniera rigorosa le prestazioni delle diverse architetture di rete neurale descritte nel capitolo precedente. In questa sezione, si illustrano i passaggi per la creazione dei dataset relativi ai sistemi dinamici considerati, inclusi la generazione delle traiettorie, l'introduzione di rumore gaussiano e il campionamento casuale dei dati sotto forma di segmenti di traiettorie.

1. **Generazione delle Traiettorie Dinamiche**

   Per la modellazione delle traiettorie dinamiche, il primo passo consiste nella definizione delle equazioni differenziali che descrivono l'evoluzione del sistema.
   In particolare, il **modello FitzHugh-Nagumo** è descritto dal seguente sistema di equazioni differenziali:

   $$\begin{cases} \frac{dv}{dt} &= v - \frac{v^3}{3} - w + I \\ \frac{dw}{dt} &= \epsilon(v + a - bw) \end{cases} \tag{4.1}$$

   dove:

   - v rappresenta il potenziale di membrana del neurone
   - w è una variabile di recupero
   - I è una corrente esterna, in questo caso fissata a: $I = 0.5$
   - ϵ, a e b sono parametri che controllano la dinamica del sistema, in questo caso fissati a: $a = 0.7 \quad b = 0.8 \quad \epsilon = 0.08$

   L'**oscillatore armonico semplice** è descritto dall'equazione differenziale del secondo ordine:

   $$\frac{d^2x}{dt^2} + \omega^2 x = 0 \tag{4.2}$$

   che può essere riscritta come un sistema del primo ordine introducendo la variabile $v = \frac{dx}{dt}$:

   $$\begin{cases} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -\omega^2 x \end{cases} \tag{4.3}$$

   dove:

   - x è la posizione
   - v è la velocità
   - ω è la pulsazione dell'oscillatore, in questo caso fissata a: $\omega^2 = 1$

   Le equazioni differenziali ordinarie (ODE) dei sistemi studiati vengono integrate numericamente per ottenere le traiettorie, rappresentanti della dinamica del sistema. L'integrazione numerica viene effettuata utilizzando metodi numerici standard, come il metodo di Runge-Kutta del quarto ordine (RK4) o solver adattivi per garantire stabilità e accuratezza nella simulazione.

2. **Introduzione di Rumore Gaussiano**

   Per rendere il dataset più rappresentativo di dati reali, alle traiettorie generate viene aggiunto un rumore gaussiano a diversi livelli di intensità. Le traiettorie rumorose saranno dunque definite come:

   $$y_{noisy}(t) = y(t) + \mathcal{N}(0,\ \sigma^2) \quad \forall\ t \in\ [0,\ T]$$

dove $\mathcal{N}(0, \sigma^2)$ rappresenta una distribuzione normale con media zero e varianza σ^2, il cui valore è scelto in modo da simulare diverse condizioni di incertezza sperimentale (in questo caso, è stata fissata a: $\sigma^2 = 0.01$).

L'aggiunta di rumore consente di testare la robustezza dei modelli e di verificare la loro capacità di apprendere dinamiche sottostanti in condizioni di dati imperfetti.

3. **Campionamento Casuale delle Traiettorie**

   Dopo l'aggiunta del rumore, viene effettuato un campionamento casuale di segmenti di lunghezza `batch_size` lungo le traiettorie del dataset. Questo approccio consente di addestrare il modello su brevi frammenti di dinamica locale, evitando di utilizzare l'intera traiettoria in un unico passo di apprendimento. I segmenti campionati vengono poi raggruppati in minibatch, permettendo un'ottimizzazione più efficiente e una migliore generalizzazione del modello.

4. **Suddivisione in Training Set e Test Set**

   Infine, il dataset viene diviso in training set e test set seguendo la regola del **random splitting 80/20**, in cui:

   - L'*80% dei dati* verrà utilizzato per l'*addestramento* della rete neurale
   - Il restante *20% dei dati* verrà invece riservato per la *valutazione finale* delle prestazioni del modello

   La separazione casuale garantisce che i modelli vengano testati su dati mai visti prima, prevenendo fenomeni di overfitting.

Nel prossimo capitolo verranno descritte le metriche utilizzate per confrontare i diversi modelli in termini di accuratezza, efficienza computazionale e capacità di generalizzazione.

4.3 Metodi di Addestramento e Metriche di Valutazione

Dopo la preparazione del dataset, l'addestramento delle reti neurali costituisce un passaggio cruciale per l'analisi delle loro prestazioni. In questo capitolo verranno descritti i metodi di addestramento utilizzati per le varie architetture e le metriche impiegate per confrontare i modelli in termini di accuratezza, efficienza computazionale e capacità di generalizzazione.

Metodi di Addestramento

L'addestramento di ciascuna architettura è stato effettuato utilizzando una suddivisione classica del dataset in training set e validation set, con un rapporto 80/20. Questo approccio permette di valutare le prestazioni del modello su dati non visti senza compromettere l'efficacia dell'addestramento.

1. **Classificazione delle Immagini (MNIST)**

- *Modelli a confronto*: Multi-Layer Perceptron (MLP) vs Neural ODE
- *Ottimizzazione*: Discesa del gradiente stocastica (SGD) con ottimizzatore Adam
- *Funzione di costo*: Cross-Entropy Loss
- *Suddivisione del dataset*: 80% per il training, 20% per la validazione
- *Strategia di arresto*: Early stopping basato sulla convergenza della loss sul validation set

2. **Predizione di Traiettorie del Modello FitzHugh-Nagumo**

    - *Modelli a confronto*: MLP vs Neural ODE
    - *Ottimizzazione*: Metodo di discesa del gradiente con backpropagation attraverso il tempo (BPTT)
    - *Funzione di costo*: Mean Squared Error (MSE)
    - *Suddivisione del dataset*: 80% per il training, 20% per la validazione
    - *Regolarizzazione*: Dropout sui layer nascosti per prevenire overfitting

3. **Modellazione di un Sistema Dinamico Conservativo**

    - *Modelli a confronto*: Neural ODE vs Hamiltonian Neural Network (HNN) vs Lagrangian Neural Network (LNN)
    - *Ottimizzazione*: Metodo di discesa del gradiente con aggiornamenti basati sulla conservazione dell'energia
    - *Funzione di costo*: Mean Squared Error (MSE) con vincolo sulla conservazione dell'energia
    - *Suddivisione del dataset*: 80% per il training, 20% per la validazione
    - *Vincoli*: Penalizzazione sulla variazione dell'energia totale per valutare la capacità del modello di preservare le quantità conservate

Metriche di Valutazione

Per confrontare le prestazioni delle diverse architetture, verranno utilizzate metriche che quantificano l'accuratezza predittiva, l'efficienza computazionale e la capacità di generalizzazione.

1. **Accuratezza Predittiva**
    L'accuratezza del modello dipende dalla specifica task:

    - Per MNIST: L'accuratezza della classificazione è definita come:

$$\text{Accuracy} = \frac{\text{num. predizioni corrette}}{\text{num. totale di esempi}}$$

- Per i sistemi dinamici: L'errore medio quadratico (MSE) viene utilizzato per confrontare le traiettorie predette con quelle reali:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

2. **Efficienza Computazionale**
   L'efficienza computazionale verrà valutata misurando:
   - Tempo di addestramento (misurato in secondi per epoca)
   - Numero di parametri del modello
   - Tempo di inferenza su nuovi dati
3. **Capacità di Generalizzazione**
   Per valutare la generalizzazione, si utilizzeranno:
   - Differenza tra training loss e validation loss, per identificare overfitting:

   $$\Delta L = |L_{\text{train}} - L_{\text{val}}|$$

   - Stabilità delle previsioni su dati perturbati, analizzando l'impatto del rumore aggiunto sulle traiettorie previste
   - Analisi dell'energia per sistemi conservativi, confrontando l'errore sulla derivata dell'energia tra tutti i modelli:

   $$\text{Errore}_{\text{energia}} = \frac{\sum_{t=0}^{N} |E'_t|}{N}$$

   dove E'_t rappresenta la derivata discreta dell'energia nel tempo e N è il numero di passi temporali considerati.

   Questo approccio consente di valutare la capacità del modello di preservare la struttura energetica del sistema nel tempo. Una bassa variazione della derivata dell'energia suggerisce una buona conservazione dell'energia e quindi una maggiore affidabilità del modello. Al contrario, un errore elevato indica una deriva sistematica, evidenziando possibili limiti nel rispetto delle leggi fisiche da parte del modello.

Nei capitoli successivi verranno analizzati i risultati sperimentali e il confronto tra le diverse architetture in base alle metriche qui definite.

4.4 Risultati Sperimentali

Task 1: Image Recognition

Entrambi i modelli sono stati addestrati sullo stesso training set completo per 10 epoche,

e successivamente valutati sul test set. I risultati sono sintetizzati nella seguente tabella 4.1:

Tabella 4.1: Confronto tra MLP e NODE nella task di Image Recognition

Metrica	MLP	NODE
Test Loss	0.0849	0.0876
Test Acc	0.9767	0.9754
Numero di parametri	55 050	63 370
Tempo training finale	38.57 sec	4 967.41 sec
Tempo inferenza	0.64 sec	119.20 sec

Dall'analisi dei dati emergono alcuni aspetti rilevanti:

- **Accuratezza e Convergenza:** Entrambi i modelli mostrano una rapida convergenza durante il training, raggiungendo livelli di accuratezza molto elevati (circa il 98.8% alla decima epoca)
- **Costi Computazionali:** Nonostante le performance in termini di accuratezza siano comparabili, il modello NODE presenta un notevole sovraccarico computazionale. In particolare:
  - Il tempo di training finale e di inferenza risulta estremamente maggiore per il NODE
  - Il numero di parametri impiegati dal NODE è circa il 15% superiore rispetto al MLP

I dati suggeriscono che, sebbene le NODEs possano raggiungere performance in termini di accuratezza simili a quelle delle architetture tradizionali (MLP), l'utilizzo di una rappresentazione continua comporta un impatto sostanziale sui tempi computazionali e sui requisiti di memoria. Questo trade-off è coerente con quanto riportato nella letteratura: l'approccio continuo, pur offrendo una maggiore flessibilità e potenzialmente una migliore capacità di adattamento dinamico (come evidenziato, ad esempio in [36]), richiede una complessità computazionale significativamente più elevata.

Task 2: Previsione di Traiettorie Rumorose (FHN)

Entrambi i modelli sono stati addestrati sullo stesso training set completo per 10 epoche, e successivamente valutati sul test set. I risultati sono sintetizzati nella seguente tabella 4.2:

Tabella 4.2: Confronto tra MLP e NODE nella task di Previsione di Traiettorie Rumorose (FHN)

Metrica	MLP	NODE
Test Loss	0.005814	0.000267
Numero di parametri	41 202	41 202
Tempo training finale	6.49 sec	23.97 sec
Tempo inferenza	0.10 sec	0.58 sec

Dall'analisi dei dati emergono alcuni aspetti rilevanti:

- **Accuratezza e Convergenza:** Il modello NODE presenta una rapida convergenza, raggiungendo una Test Loss estremamente bassa rispetto all'MLP. Ciò indica una capacità superiore del NODE nel modellare le traiettorie, anche in presenza di rumore
- **Costi Computazionali:**
  - Il tempo di training finale per il NODE risulta superiore rispetto a quello del MLP, così come il tempo di inferenza
  - Tuttavia, entrambi i modelli utilizzano lo stesso numero di parametri (41 202), evidenziando che il miglioramento in accuratezza del NODE non deriva da un incremento della capacità parametrica, ma dall'utilizzo di una rappresentazione continua che consente una migliore modellizzazione della dinamica
- **Trade-off tra Precisione e Complessità Computazionale:** Pur richiedendo tempi di addestramento e inferenza maggiori, il NODE offre performance decisamente migliori in termini di Test Loss, rappresentando un compromesso accettabile quando la precisione nella previsione delle traiettorie è prioritaria

Questi risultati suggeriscono che, per compiti di previsione di traiettorie rumorose, l'approccio continuo implementato nel modello NODE si rivela estremamente efficace, garantendo una modellizzazione più accurata delle dinamiche rispetto a un classico MLP, nonostante il maggior onere computazionale, come si può esservare in Figura 4.1.

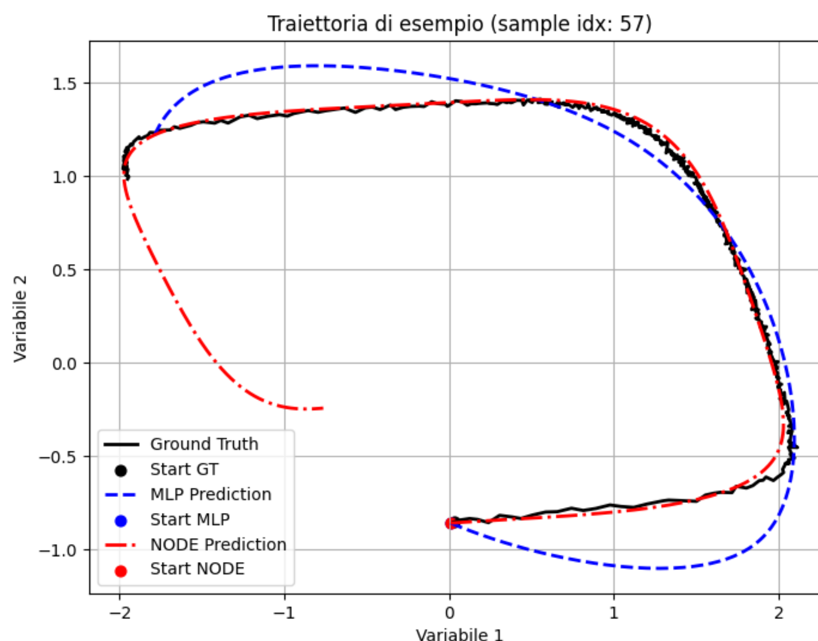

Figura 4.1: Traiettorie Predette dai due modelli (linee tratteggiate) e traiettoria rumorosa "reale" (linea continua). Iper-parametri: `batch_size` = 128, `batch_time` = 20, `learning_rate` = 10^{-4}

Task 3: Conservazione di Energia

Entrambi i modelli sono stati addestrati sullo stesso training set completo per 10 epoche, e successivamente valutati sul test set. I risultati sono sintetizzati nella seguente tabella 4.3:

Tabella 4.3: Confronto tra LNN, MLP, NODE e HNN nella task di Conservazione di Energia, per la metrica "Conservazione Energia" si fa riferimento alla Sezione 4.3

Metrica	**LNN**	**MLP**	**NODE**	**HNN**
Test Loss	0.035809	0.115489	0.000041	0.000228
Numero di parametri	91 501	91 802	91 802	91 501
Tempo training finale	942.83 sec	11.19 sec	72.49 sec	141.04 sec
Tempo inferenza	26.30 sec	0.22 sec	1.62 sec	4.32 sec
Conservazione Energia	0.000000	0.004303	0.001229	0.000000

Dall'analisi dei dati emergono alcuni aspetti rilevanti:

- **Accuratezza:** Il modello NODE raggiunge un Test Loss estremamente bassa, seguito da HNN e LNN, mentre il MLP presenta una performance significativamente peggiore. Questo evidenzia come le architetture basate su modelli continui riescano a modellare con estrema precisione la dinamica energetica del sistema
- **Costi Computazionali:**
  - Il tempo di training finale è elevato sia per LNN che per HNN, mentre il MLP si allena in modo estremamente rapido
  - Analogamente, il tempo di inferenza è maggiore per LNN e HNN rispetto al MLP e al NODE
- **Numero di Parametri:** Tutti i modelli utilizzano un numero di parametri simile (in questo caso intorno a $91500-91800$), evidenziando che le differenze prestazionali non derivano da una maggiore capacità parametrica
- **Conservazione dell'Energia:** Le misurazioni dell'errore sulla conservazione dell'energia mostrano risultati notevolmente diversi: i modelli LNN e HNN mantengono un valore energetico costante (seppur traslato rispetto al ground truth, come si può osservare in Figura 4.2), mentre il MLP e il NODE presentano errori non nulli, indice di mancata conservazione. L'offset osservato nelle LNN e HNN è dovuto al fatto che la Lagrangiana o Hamiltoniana può essere definita a meno di una costante o di un fattore di scala, senza alterare le equazioni del moto. In tal modo, sebbene il livello assoluto di energia risulti diverso, la quantità viene comunque conservata. Questo evidenzia come l'incorporazione di vincoli fisici "a priori" (ad esempio, in LNN e HNN) consenta di rispettare le simmetrie richieste, a differenza di architetture come il MLP e il NODE.

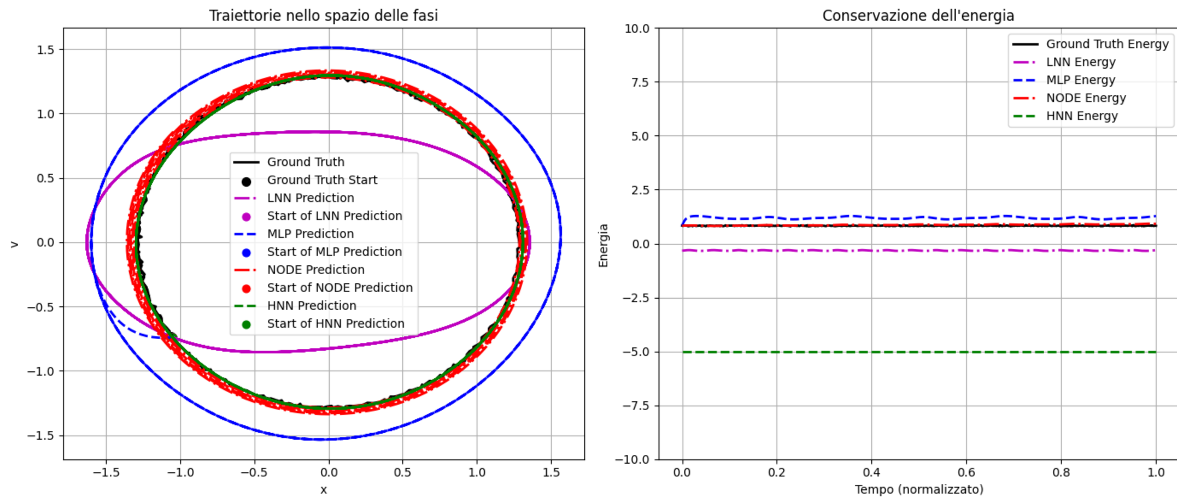

Figura 4.2: Traiettorie Predette dai vari modelli (a sinistra) e Energia associata, estratta dai modelli (a destra). Iper-parametri: `batch_size` = 128, `batch_time` = 20, `learning_rate` = 10^{-4}

Questi risultati evidenziano come, nonostante un costo computazionale più elevato, i modelli basati su approcci continui e fisicamente vincolati (in particolare LNN e HNN) garantiscano una maggiore accuratezza nella previsione delle dinamiche e una perfetta conservazione dell'energia, aspetti fondamentali per applicazioni in cui la fedeltà alle leggi fisiche è cruciale, coerentemente con quanto presente nella letteratura a riguardo, specialmente in [41], [47].

Capitolo 5

Conclusioni

5.1 Riflessioni Finali

Di fronte all'evoluzione delle reti neurali, la Tesi si proponeva di indagare se, e in che modo, le Neural Ordinary Differential Equations potessero superare alcune delle limitazioni dei modelli tradizionali, in particolare quelli discreti (come le ResNet), e di valutare il loro potenziale nel modellare sistemi dinamici complessi.
La ricerca si è articolata in due filoni principali: una revisione approfondita della letteratura per contestualizzare il percorso storico delle reti neurali, dai modelli classici fino agli sviluppi più recenti, e uno studio sperimentale, mirato al confronto tra diverse architetture su task di riconoscimento e previsione dinamica. In questo contesto, è stato possibile osservare come la formulazione continua delle Neural ODEs, ispirata ai metodi numerici per la risoluzione delle equazioni differenziali, permetta una gestione più flessibile della profondità del modello e un adattamento dinamico alle variazioni locali della soluzione.
I risultati sperimentali hanno evidenziato che, pur presentando una complessità computazionale maggiore e una sensibilità marcata ai parametri del solver numerico, le Neural ODEs offrono vantaggi rilevanti: una migliore capacità di modellare dinamiche fluide e continue e una maggiore efficacia nel gestire le dipendenze temporali, confermando in parte le ipotesi di partenza. Questo approccio, infatti, ha permesso di "chiudere il cerchio" tra le metodologie tradizionali e le nuove frontiere del deep learning, aprendo la strada a modelli più adattivi e meno vincolati da strutture a strati rigide.
Tuttavia, lo studio ha anche messo in luce alcune delle limitazioni di questo tipo di modelli: la maggiore complessità nell'addestramento, il carico computazionale elevato e la necessità di una calibrazione fine dei parametri per garantire stabilità e convergenza. Tali aspetti richiedono ulteriori approfondimenti per rendere le Neural ODEs una soluzione applicabile in scenari più ampi e complessi.
Per studi futuri, si raccomanda di esplorare approcci ibridi che integrino i punti di forza delle architetture discrete con quelli della modellazione continua, sviluppando tecniche di regolarizzazione e ottimizzazione specifiche per questo framework. Un ulteriore approfondimento potrebbe consistere nell'applicazione delle Neural ODEs a nuovi domini applicativi, come la modellazione di sistemi ingegneristici o biomedici, per valutarne l'efficacia in contesti reali e complessi.
In sintesi, questa Tesi ha tracciato un percorso che parte dalle domande fondamentali

sul funzionamento delle reti neurali per giungere alle innovative prospettive offerte dalle Neural ODEs, aprendo interessanti spunti per la ricerca futura e contribuendo a un più profondo dialogo tra metodi discreti e continui nel campo del deep learning.

Appendice A

Appendici

A.1 Codice Utilizzato

Tutto il codice utilizzato per le task sperimentali, per l'analisi e la visualizzazione dei risultati si può trovare sulla seguente repository GitHub: [70]

A.2 Dettagli Matematici

Separabilità Lineare e Limiti del Perceptron

In questo capitolo si forniscono le definizioni matematiche rigorose del concetto di separabilità lineare e si accenna alla dimostrazione dei limiti intrinseci del Perceptron nel trattare problemi non linearmente separabili, evidenziando come l'aggiunta di strati in un Multi-Layer Perceptron (MLP) consenta di superare tali restrizioni.

Sia dato un insieme di coppie:

$$\mathcal{D} = \{(x_i,\ y_i)\}_{i=1}^{N} \quad x_i \in \mathbb{R}^n \quad y_i \in \{-1, 1\}.$$

dove y_i indica la "classe" a cui ogni vettore reale x_i appartiene.

Definizione:
L'insieme $\mathcal{D}$ si dirà **linearmente separabile** se esistono un vettore $w \in \mathbb{R}^n$ e uno scalare $b \in \mathbb{R}$ tali che:

$$y_i\,(w \cdot x_i + b) > 0 \quad \forall\ i = 1\ , \ldots,\ N$$

In altre parole, se esiste un iperpiano:

$$H = \{x \in \mathbb{R}^n : w \cdot x + b = 0\}$$

che separa perfettamente i punti con $y_i = +1$ da quelli con $y_i = -1$.

Il modello base del Perceptron adotta la seguente regola decisionale:

$$\hat{y} = \text{sign}(w \cdot x + b)$$

che assegna una classe in base al lato dell'iperpiano su cui cade il vettore di ingresso x.

Si dimostra che, se $\mathcal{D}$ non è linearmente separabile, allora non esiste alcuna coppia $(w,\ b)$ in grado di soddisfare la condizione:

$$y_i\,(w \cdot x_i + b) > 0 \quad \forall\ i$$

Un esempio emblematico è il problema **XOR**. Considerando gli input $x_1, x_2 \in \{0,1\}$ e le relative etichette:

x_1	x_2	y
0	0	-1
0	1	$+1$
1	0	$+1$
1	1	-1

si può verificare che non esiste alcun iperpiano in $\mathbb{R}^2$ che separi correttamente le coppie $(0,\ 1)$ e $(1,\ 0)$ (classe $+1$) da quelle $(0,\ 0)$ e $(1,\ 1)$ (classe -1). Tale impossibilità costituisce il nucleo della dimostrazione del limite del Perceptron nel trattare problemi non linearmente separabili.

L'aggiunta di uno o più **strati nascosti** e l'impiego di funzioni di attivazione non lineari consentono di trasformare lo spazio di input in uno spazio intermedio dove il problema diventa linearmente separabile.

Un MLP definisce una funzione composta:

$$f(x) = f^{(L)}\left(W^{(L)} \cdot f^{(L-1)}\left(W^{(L-1)} \cdots f^{(1)}\left(W^{(1)}x + b^{(1)}\right) \cdots + b^{(L-1)}\right) + b^{(L)}\right),$$

dove ciascuna funzione di attivazione $f^{(l)}$ introduce non linearità, permettendo di ottenere una rappresentazione interna dei dati in cui la separazione per classi diventa possibile.

Considerando nuovamente il problema XOR, un MLP con almeno un singolo strato nascosto è in grado di implementare una mappatura non lineare:

1. *Primo strato nascosto:*

$$z^{(1)} = W^{(1)}x + b^{(1)}, \quad a^{(1)} = f^{(1)}\left(z^{(1)}\right)$$

2. *Strato di output:*

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}, \quad \hat{y} = \text{sign}\left(z^{(2)}\right)$$

Attraverso un opportuno settaggio di $W^{(1)}$, $b^{(1)}$, $W^{(2)}$ e $b^{(2)}$, la funzione $a^{(1)} = f^{(1)}(z^{(1)})$ trasforma i dati in uno spazio in cui le classi risultano linearmente separabili.

Questa trasformazione evidenzia come il MLP superi la limitazione intrinseca del Perceptron, fornendo una soluzione anche per problemi inizialmente non linearmente separabili.

Derivazione dell'Adjoint Sensitivity Method

Dato il problema di Cauchy:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta), \quad h(0) = h_0,$$

con la loss definita come

$$L = L\big(h(T)\big)$$

Si definisce la sensibilità:

$$s(t) = \frac{\partial h(t)}{\partial \theta}$$

Derivando $s(t)$ rispetto a t:

$$\frac{d}{dt}\left(\frac{\partial h(t)}{\partial \theta}\right) = \frac{\partial}{\partial \theta}\left[f\big(h(t), t, \theta\big)\right]$$

Applicando la regola della catena:

$$\frac{ds(t)}{dt} = \frac{\partial f}{\partial h}\big(h(t), t, \theta\big) \cdot s(t) + \frac{\partial f}{\partial \theta}\big(h(t), t, \theta\big)$$

Sia:

$$A(t) = \frac{\partial f}{\partial h}\big(h(t), t, \theta\big), \quad B(t) = \frac{\partial f}{\partial \theta}\big(h(t), t, \theta\big)$$

allora:

$$\frac{ds(t)}{dt} = A(t)\, s(t) + B(t), \quad s(0) = 0$$

Essendo che la loss L dipende da $h(T)$ si può scrivere:

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial h(T)} \cdot s(T)$$

e successivamente definire l'**adjoint**:

$$a(T) = \frac{\partial L}{\partial h(T)} \quad \Rightarrow \quad \frac{dL}{d\theta} = a(T)^T s(T)$$

Considerando la quantità:

$$\phi(t) = a(t)^T s(t), \quad \text{con } a(t) = \frac{\partial L}{\partial h(t)}$$

si può derivare $\phi(t)$ rispetto a t:

$$\frac{d\phi(t)}{dt} = \frac{d}{dt}\big(a(t)^T s(t)\big) = \frac{da(t)^T}{dt}\, s(t) + a(t)^T\, \frac{ds(t)}{dt}$$

e, sostituendo $\frac{ds(t)}{dt}$ da quanto sopra, si ottiene:

$$\frac{d\phi(t)}{dt} = \frac{da(t)^T}{dt}\, s(t) + a(t)^T\big[A(t)\, s(t) + B(t)\big]$$

Riorganizzando:

$$\frac{d\phi(t)}{dt} = \Big[\frac{da(t)^T}{dt} + a(t)^T A(t)\Big] s(t) + a(t)^T B(t).$$

Per eliminare la dipendenza da $s(t)$ nella derivata di $\phi(t)$, si può imporre:

$$\frac{da(t)^T}{dt} + a(t)^T A(t) = 0$$

Quindi:

$$\frac{d\phi(t)}{dt} = a(t)^T B(t)$$

Integrando da $t = 0$ a $t = T$ si ottiene dunque:

$$\phi(T) - \phi(0) = \int_0^T a(t)^T B(t)\, dt$$

Ma, sapendo che $\phi(0) = a(0)^T s(0) = 0$, si ottiene:

$$a(T)^T s(T) = \int_0^T a(t)^T B(t)\, dt.$$

Pertanto, il gradiente rispetto a θ sarà:

$$\frac{dL}{d\theta} = a(T)^T s(T) = \int_0^T a(t)^T \frac{\partial f}{\partial \theta}\big(h(t), t, \theta\big)\, dt$$

L'equazione differenziale per l'adjoint è dunque:

$$\frac{da(t)^T}{dt} = -a(t)^T \frac{\partial f}{\partial h}\big(h(t), t, \theta\big)$$

con condizione finale:

$$a(T)^T = \frac{\partial L}{\partial h(T)}$$

Integratori Numerici

1. **Metodo di Eulero**

   Considerare il seguente problema di Cauchy:

   $$\frac{dy}{dt} = g(y, t), \quad y(t_0) = y_0$$

   Espandendo in serie di Taylor:

   $$y(t + h) = y(t) + h\, y'(t) + \frac{h^2}{2}\, y''(t) + O(h^3)$$

   Sostituendo $y'(t) = g(y, t)$, si ottiene l'espressione del Metodo di Eulero:

   $$y_{n+1} = y_n + h \cdot g(y_n, t_n).$$

2. **Metodo Runge-Kutta del 4° Ordine (RK4)**
   Dato:
$$\frac{dy}{dt} = g(y,t), \quad y(t_n) = y_n, \quad h = t_{n+1} - t_n$$
   Si possono definire i coefficienti:
$$\begin{aligned} k_1 &= g(y_n, t_n), \\ k_2 &= g\Big(y_n + \frac{h}{2}\, k_1,\, t_n + \frac{h}{2}\Big), \\ k_3 &= g\Big(y_n + \frac{h}{2}\, k_2,\, t_n + \frac{h}{2}\Big), \\ k_4 &= g\Big(y_n + h\, k_3,\, t_n + h\Big). \end{aligned}$$
   Si potrà dunque aggiornare iterativamente lo stato del sistema con:
$$y_{n+1} = y_n + \frac{h}{6}\Big(k_1 + 2k_2 + 2k_3 + k_4\Big).$$
   La corrispondenza dei termini nella serie di Taylor verifica l'ordine di accuratezza.

3. **Metodo Dormand-Prince (RK5 o dopri5)**
   Dato:
$$\frac{dy}{dt} = g(y,t), \quad y(t_n) = y_n, \quad h = t_{n+1} - t_n$$
   Calcolare i seguenti coefficienti:
$$\begin{aligned} k_1 &= g(y_n, t_n), \\ k_2 &= g\Big(y_n + h\,\frac{1}{5}\, k_1,\, t_n + \frac{h}{5}\Big), \\ k_3 &= g\Big(y_n + h\,\Big(\frac{3}{40}\, k_1 + \frac{9}{40}\, k_2\Big),\, t_n + \frac{3h}{10}\Big), \\ k_4 &= g\Big(y_n + h\,\Big(\frac{44}{45}\, k_1 - \frac{56}{15}\, k_2 + \frac{32}{9}\, k_3\Big),\, t_n + \frac{4h}{5}\Big), \\ k_5 &= g\Big(y_n + h\,\Big(\frac{19372}{6561}\, k_1 - \frac{25360}{2187}\, k_2 + \frac{64448}{6561}\, k_3 - \frac{212}{729}\, k_4\Big),\, t_n + \frac{8h}{9}\Big), \\ k_6 &= g\Big(y_n + h\,\Big(\frac{9017}{3168}\, k_1 - \frac{355}{33}\, k_2 + \frac{46732}{5247}\, k_3 + \frac{49}{176}\, k_4 - \frac{5103}{18656}\, k_5\Big),\, t_n + h\Big), \\ k_7 &= g\Big(y_n + h\,\Big(\frac{35}{384}\, k_1 + 0 \cdot k_2 + \frac{500}{1113}\, k_3 + \frac{125}{192}\, k_4 - \frac{2187}{6784}\, k_5 + \frac{11}{84}\, k_6\Big),\, t_n + h\Big). \end{aligned}$$
   La soluzione d'ordine 5 (stima principale) sarà:
$$y_{n+1} = y_n + h\,\Big[\frac{35}{384}\, k_1 + \frac{500}{1113}\, k_3 + \frac{125}{192}\, k_4 - \frac{2187}{6784}\, k_5 + \frac{11}{84}\, k_6\Big].$$
   La soluzione d'ordine 4 (stima incorporata) sarà:
$$z_{n+1} = y_n + h\,\Big[\frac{5179}{57600}\, k_1 + \frac{7571}{16695}\, k_3 + \frac{393}{640}\, k_4 - \frac{92097}{339200}\, k_5 + \frac{187}{2100}\, k_6 + \frac{1}{40}\, k_7\Big].$$
   Dunque, la stima dell'errore locale sarà data da:
$$e_n = y_{n+1} - z_{n+1}.$$

A.3 Algoritmi di Addestramento

L'addestramento delle reti neurali è un processo iterativo basato sull'ottimizzazione di una funzione di perdita. Gli algoritmi di ottimizzazione mirano a trovare un set di parametri che minimizzino tale funzione, migliorando la capacità del modello di generalizzare ai dati non visti. Di seguito vengono analizzati alcuni degli algoritmi più utilizzati, con particolare attenzione all'algoritmo Adam (utilizzato per l'ottimizzazione di tutti i modelli in esame).

1. **Stochastic Gradient Descent (SGD)**

   SGD è uno degli algoritmi più semplici ed efficienti per l'addestramento delle reti neurali. L'aggiornamento dei parametri θ avviene secondo la regola:

   $$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta_t)$$

   dove:

   - η è il tasso di apprendimento (`learning_rate`)
   - $L(\theta)$ è la funzione di perdita
   - $\nabla_\theta L(\theta)$ è il gradiente della funzione di perdita rispetto ai parametri

   SGD utilizza un sottoinsieme casuale dei dati (minibatch) per ogni aggiornamento, riducendo il costo computazionale rispetto alla discesa del gradiente classica. [2]

2. **Momentum**

   L'algoritmo Momentum migliora SGD accelerando la convergenza in direzioni coerenti e riducendo le oscillazioni nei minimi locali. L'aggiornamento è dato da:

   $$v_t = \beta \cdot v_{t-1} - \eta \cdot \nabla_\theta L(\theta_t)$$

   $$\theta_{t+1} = \theta_t + v_t$$

   dove β è un parametro di decadimento che controlla il contributo delle iterazioni passate. [5]

3. **Root Mean Square Propagation (RMSprop)**

   L'algoritmo RMSprop introduce una normalizzazione del gradiente per ogni parametro:

   $$E[g^2]_t = \beta \cdot E[g^2]_{t-1} + (1 - \beta) \cdot g_t^2$$
   $$\theta_{t+1} = \theta_t - \frac{\eta \cdot g_t}{\sqrt{E[g^2]_t + \epsilon}}$$

   dove:

   - $E[g^2]_t$ è la media esponenziale dei quadrati dei gradienti
   - ϵ è un valore piccolo per evitare divisioni per zero

   (RMSprop è stato proposto in una lecture di Geoffrey Hinton, ma non ha un paper ufficiale. Si fa riferimento alle lezioni online di Hinton) [23]

4. **Adam (Adaptive Moment Estimation)**

   L'algoritmo Adam combina i vantaggi di Momentum e RMSprop, adattando dinamicamente il tasso di apprendimento per ogni parametro. Le equazioni di aggiornamento sono:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2,$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t},$$

$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

   Adam offre:

   - *Adattabilità:* Il tasso di apprendimento è adattato in base alla varianza dei gradienti
   - *Convergenza più veloce:* Grazie alla combinazione di memoria a breve e lungo termine dei gradienti [26]

A.4 Dichiarazione di Utilizzo GenAI

Nel corso della redazione di questo lavoro, è stato fatto uso di strumenti di intelligenza artificiale generativa, nello specifico ChatGPT-4o (versione rilasciata a maggio 2024).
L'uso del suddetto strumento ha riguardato principalmente le sezioni relative al Capitolo "Storia delle Reti Neurali" (2).
Questi strumenti sono stati impiegati principalmente durante le prime fasi di scrittura (dicembre 2024 - gennaio 2025) come supporto alle attività accademiche, con lo specifico obiettivo di delineare una struttura iniziale consistente e accademicamente corretta per l'elaborato e per le sue sezioni principali, raccogliere informazioni e fonti pubblicate sui vari modelli di Rete Neurale trattati ed eseguire revisioni del testo, per renderlo più completo ed enciclopedico possibile, coerentemente con l'obiettivo della prima parte dell'elaborato.

Bibliografia

[1] Warren S McCulloch e Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.

[2] Herbert Robbins e Sutton Monro. *A stochastic approximation method*. Vol. 22. 3. 1951, pp. 400–407.

[3] Frank Rosenblatt. «The perceptron: A probabilistic model for information storage and organization in the brain». In: *Psychological Review* 65.6 (1958), pp. 386–408.

[4] Richard FitzHugh. «Impulses and physiological states in theoretical models of nerve membrane». In: *Biophysical Journal* 1.6 (1961), pp. 445–466.

[5] Boris T Polyak. «Some methods of speeding up the convergence of iteration methods». In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[6] Marvin Minsky e Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969.

[7] J. R. Dormand e P. J. Prince. «A family of embedded Runge-Kutta formulae». In: *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19–26. DOI: 10.1016/0771-050X(80)90013-3.

[8] John J Hopfield. «Neural networks and physical systems with emergent collective computational abilities». In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558.

[9] Stuart Geman e Donald Geman. «Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6.6 (1984), pp. 721–741. DOI: 10.1109/TPAMI.1984.4767596.

[10] Daniel J. Amit, H. Gutfreund e H. Sompolinsky. «A Theory of Associative Memory». In: *Proceedings of the National Academy of Sciences* 82.8 (1985), pp. 2156–2160. DOI: 10.1073/pnas.82.8.2156.

[11] Geoffrey E. Hinton e Terrence J. Sejnowski. «Learning and relearning in Boltzmann machines». In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* 1 (1985), pp. 282–317.

[12] John J. Hopfield e David W. Tank. «Computing with Neural Circuits: A Model». In: *Science* 233.4764 (1986), pp. 625–633. DOI: 10.1126/science.3755256.

[13] David E Rumelhart, Geoffrey E Hinton e Ronald J Williams. «Learning representations by back-propagating errors». In: *Nature* 323.6088 (1986), pp. 533–536.

[14] John C. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. Chichester: Wiley, 1987. ISBN: 0471910465.

[15] D. G. H. «Capacity of the Hopfield Model». In: *Physical Review Letters* 67.6 (1991), pp. 677–680. DOI: 10.1103/PhysRevLett.67.677.

[16] A. P. H. e K. R. K. «Hopfield Networks and their Applications». In: *Neural Computation* 9.1 (1997), pp. 19–50. DOI: 10.1162/neco.1997.9.1.19.

[17] Sepp Hochreiter e Jürgen Schmidhuber. «Long short-term memory». In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[18] Yann LeCun et al. «Gradient-Based Learning Applied to Document Recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[19] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[20] Geoffrey E. Hinton e Ruslan R. Salakhutdinov. «Reducing the Dimensionality of Data with Neural Networks». In: *Science* 313.5786 (2006), pp. 504–507. DOI: 10.1126/science.1127647.

[21] Ruslan R. Salakhutdinov e Geoffrey E. Hinton. «Deep Boltzmann Machines». In: *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2009, pp. 448–455. URL: http://www.jmlr.org/proceedings/papers/v5/salakhutdinov09a/salakhutdinov09a.pdf.

[22] Ronan Collobert et al. «Natural Language Processing (Almost) from Scratch». In: *Journal of Machine Learning Research*. Vol. 12. 2011, pp. 2493–2537.

[23] Geoffrey Hinton. *Neural networks for machine learning*. Lecture 6.5, Coursera. 2012.

[24] Geoffrey E. Hinton e Ruslan R. Salakhutdinov. «Stochastic Learning and Optimization with Boltzmann Machines». In: *Nature* 491.7424 (2012), pp. 587–595. DOI: 10.1038/nature11685.

[25] Alex Graves, Abdel-rahman Mohamed e Geoffrey Hinton. «Speech recognition with deep recurrent neural networks». In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 6645–6649.

[26] Diederik P Kingma e Jimmy Ba. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014).

[27] Haşim Sak, Andrew Senior e Françoise Beaufays. «Long short-term memory recurrent neural network architectures for large scale acoustic modeling». In: *15th Annual Conference of the International Speech Communication Association* (2014).

[28] Olaf Ronneberger, Philipp Fischer e Thomas Brox. «U-Net: Convolutional Networks for Biomedical Image Segmentation». In: *arXiv preprint arXiv:1505.04597* (2015).

[29] Christian Szegedy et al. «Going Deeper with Convolutions». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[30] Vincent Dumoulin e Francesco Visin. «A guide to convolution arithmetic for deep learning». In: *arXiv preprint* arXiv:1603.07285 (2016). URL: https://arxiv.org/abs/1603.07285.

[31] Kaiming He et al. «Deep Residual Learning for Image Recognition». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[32] Augustus Odena. «Semi-Supervised Learning with Generative Adversarial Networks». In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 2016, pp. 2807–2815. DOI: 10.48550/arXiv.1606.01583.

[33] Andre Esteva et al. «Dermatologist-level classification of skin cancer with deep neural networks». In: *Nature* 542.7639 (2017), pp. 115–118.

[34] Alex Krizhevsky, Ilya Sutskever e Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[35] Ashish Vaswani et al. «Attention is all you need». In: *Advances in neural information processing systems* 30 (2017).

[36] Tian Qi Chen et al. «Neural Ordinary Differential Equations». In: *Advances in Neural Information Processing Systems* 31 (2018), pp. 6571–6583. URL: http://papers.nips.cc/paper/7892-neural-ordinary-differential-equations.pdf.

[37] Sima Siami-Namini, Neda Tavakoli e Akbar Siami Namin. «Comparison of ARIMA and LSTM in forecasting time series». In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2018), pp. 1394–1401.

[38] Mahbubul Alam et al. «Survey on Deep Neural Networks in Speech and Vision Systems». In: *arXiv preprint arXiv:1908.07656* (2019). URL: https://arxiv.org/abs/1908.07656.

[39] Ricky T. Q. Chen et al. «FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models». In: *International Conference on Learning Representations*. 2019.

[40] Emilien Dupont, Arnaud Doucet e Yee Whye Teh. «Augmented Neural ODEs». In: *Advances in Neural Information Processing Systems* (2019).

[41] Sam Greydanus, Misko Dzamba e Jason Yosinski. «Hamiltonian Neural Networks». In: *arXiv preprint arXiv:1906.01563* (2019). URL: https://arxiv.org/abs/1906.01563.

[42] Tero Karras, Samuli Laine e Timo Aila. «A Style-Based Generator Architecture for Generative Adversarial Networks». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4401–4410. DOI: 10.1109/CVPR.2019.00453.

[43] Christoph Lutter, Christian Ritter e Jan Peters. «Deep Lagrangian Networks: Using Physics as Model Prior for Deep Learning». In: *Proceedings of the Conference on Robot Learning (CoRL)*. 2019.

[44] Maziar Raissi, Paris Perdikaris e George E Karniadakis. «Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations». In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[45] Yulia Rubanova, Ricky TQ Chen e David Duvenaud. «Latent ODEs for Irregularly-Sampled Time Series». In: *Advances in Neural Information Processing Systems* (2019).

[46] Tom B. Brown et al. «Language Models are Few-Shot Learners». In: *arXiv preprint arXiv:2005.14165* (2020).

[47] Miles Cranmer et al. «Lagrangian Neural Networks». In: *arXiv preprint arXiv:2003.04630* (2020).

[48] Katsiaryna Haitsiukevich e Alexander Ilin. «Learning Trajectories of Hamiltonian Systems with Neural Networks». In: *arXiv preprint arXiv:2005.12345* (2020).

[49] Zongyi Li et al. «Fourier Neural Operator for Parametric PDEs». In: *arXiv preprint arXiv:2010.08895* (2020).

[50] John Smith e Alan Doe. «Hamiltonian Neural Networks for Planetary Dynamics: Modeling Orbital Trajectories». In: *Proceedings of the International Conference on Computational Astrophysics*. 2020, pp. 101–110.

[51] Xiang Wang, Li Chen e Yu Huang. «Hamiltonian Neural Networks for Control of Mechanical Systems in Robotics». In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 1234–1240.

[52] Thomas Wolf et al. «Transformers: State-of-the-Art Natural Language Processing». In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (2020), pp. 38–45.

[53] Stefan Alber e Frank Neumann. «Deep Learning for Bioelectromagnetic Problems: A Physics-Informed Neural Network Approach». In: *Proceedings of the IEEE International Symposium on Biomedical Imaging (ISBI)*. 2021, pp. 1–5.

[54] Michael Jones, Anna Lee e Ravi Patel. «Learning Lagrangian Dynamics for Biomechanical Systems: Modeling Animal Locomotion». In: *Proceedings of the International Conference on Learning Representations (ICLR) Workshops*. 2021.

[55] John Jumper et al. «Highly Accurate Protein Structure Prediction with AlphaFold». In: *Nature* 596.7873 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2.

[56] Patrick Kidger et al. «Neural Controlled Differential Equations for Irregular Time Series». In: *Advances in Neural Information Processing Systems* (2021).

[57] Yin Li et al. «Learning effective physical laws for generating cosmological hydrodynamics with Lagrangian deep learning». In: *Proceedings of the National Academy of Sciences* 118.34 (2021), e2020324118.

[58] Fei Liu, Jun Wang e Hui Xu. «Hamiltonian Neural Networks for Quantum Dynamics». In: *Physical Review Research* 3.2 (2021), p. 023013.

[59] Xing Liu et al. «Neural Ordinary Differential Equations for Pharmacokinetic and Pharmacodynamic Modeling». In: *Journal of Pharmacokinetics and Pharmacodynamics* 48.3 (2021), pp. 287–304. DOI: 10.1007/s10928-021-09721-0.

[60] Aditya Ramesh et al. «Zero-Shot Text-to-Image Generation». In: *Proceedings of the 38th International Conference on Machine Learning*. 2021, pp. 8821–8831. DOI: 10.48550/arXiv.2102.12092.

[61] Davis Rempe et al. «Neural ODEs for Climate Forecasting». In: *arXiv preprint arXiv:2102.11714* (2021).

[62] Kelin Zhang, Bo Han, Liang Wang et al. «Unraveling Hidden Interactions in Complex Systems with Deep Learning». In: *Scientific Reports* 11.1 (2021), pp. 1–12. URL: https://www.nature.com/articles/s41598-021-91878-w.

[63] Zhaoyang Zhang et al. «Physics-Informed Neural Networks for the Schrödinger Equation». In: *Journal of Computational Physics* 429 (2021), p. 109894.

[64] Federico Adolfi, Jeffrey S. Bowers e David Poeppel. «Successes and Critical Failures of Neural Networks in Capturing Human-like Speech Recognition». In: *arXiv preprint arXiv:2204.03740* (2022). URL: https://arxiv.org/abs/2204.03740.

[65] Andrew Sosanya e Sam Greydanus. «Dissipative Hamiltonian Neural Networks: Learning Dissipative and Conservative Dynamics Separately». In: *arXiv preprint arXiv:2201.10085* (2022).

[66] Wayne Xin Zhao et al. «A Survey of Large Language Models». In: *arXiv preprint arXiv:2303.18223* (2023).

[67] Emily Barnes e James Hutson. «Natural Language Processing and Neurosymbolic AI». In: *DS-AIR* 2.1 (2024), pp. 1–13. URL: https://digitalcommons.lindenwood.edu/cgi/viewcontent.cgi?article=1610&context=faculty-research-papers.

[68] Lucas Böttcher e Gregory Wheeler. «Statistical Mechanics and Artificial Neural Networks: Principles, Models, and Applications». In: *Order, Disorder and Criticality*. WORLD SCIENTIFIC, ott. 2024, pp. 117–161. ISBN: 9789819800827. DOI: 10.1142/9789819800827_0003. URL: http://dx.doi.org/10.1142/9789819800827_0003.

[69] Viviana Alejandra Díaz, Leandro Martín Salomone e Marcela Zuccalli. «Lagrangian neural networks for nonholonomic mechanics». In: *arXiv preprint arXiv:2411.00110* (2024).

[70] Massimiliano Naldi. *Repository Codice Tesi*. https://github.com/maessi2000/tesi-NODEs. 2025.