UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Legendre Reservoir Memory Unit: LRMU

Relatori:
Prof. Claudio Gallicchio
Dott. Andrea Ceni

Candidato:
Vito Barra

Anno Accademico 2023/2024

Contents

1 Introduzione **5**

2 Background **7**
2.1 Machine Learning 7
2.1.1 Apprendimento supervisionato 9
2.2 Reti neurali 9
2.2.1 Reti neurali Feed-forward 11
2.2.2 Reti neurali ricorrenti 13
2.3 Reservoir computing 15
2.3.1 Echo State Network 15
2.4 Legendre Memory Unit 18
2.5 Metodologia 19
2.5.1 Concetti generali 19
2.5.2 TensorFlow e Keras 21

3 Modelli **26**
3.1 Legendre Reservoir Memory Unit 26
3.2 Legendre Memory Unit con ESN 31
3.3 Legendre Reservoir Memory Unit con ESN 32
3.4 Legendre Reservoir Memory Unit con ESN e Ridge come Readout 33

4 Sperimentazione e Risultati **36**
4.1 Caratterizzazione generale degli esperimenti 36
4.1.1 Obiettivi delle sperimentazioni 36
4.1.2 Selezioni dei modelli 37
4.2 psMNIST 37
4.3 Mackey-Glass 40

5 Conclusioni **45**

List of Figures

2.1 Schema di un neurone artificiale. 10
2.2 Schema di perceptron network. 11
2.3 Schema di una multi layer perceptron network. 12
2.4 Schema di una rete simple ricorrente. 14
2.5 Schema Echo State Network. 16
2.6 Schema compatto di una layer LMU. 19

4.1 Il grafico a sinistra mostra un elemento del MNIST con label 8 mentre il grafico a destra mostra lo stesso elemento trasformato in pMNIST. 37
4.2 Illustrazione dello stesso elemento della figura 4.1, nel dataset ***psMNIST*** visualizzato come un rettangolo 8×98 pixel. 38
4.3 Mackey-Glass con $\tau = 17$. 41
4.4 Mackey-Glass con $\tau = 30$. 41

Listings

1 Codice python per costruire un modello con api funzionali. Nello specifico questo modello è composto da layer di input da 784 unità una per ogni pixel di un'immagine del dataset, un hidden layer da 64 unità con *funzione di attivazione* ReLu e layer di output da 10 unità con *funzione di attivazione* Softmax, utile per fare la classificazione multi classe.222 Codice python per allenare un modello testarlo e farci inferenza. Continuazione del listato 2.233 Codice python per costruire un custom layer.234 Codice python per costruire HyperModel.245 Codice python per avviare una ricerca randomica degli iperparametri sul hypermodel.256 Codice python dell'inizializzazione del Custom Layer LRMU.277 Codice python del metodo build del Custom Layer LRMU.298 Codice python del metodo call del Custom Layer LRMU.319 Codice python per costruire un modello utilizzando il Custom Layer LRMU.3110Codice python per costruire un modello utilizzando il layer LMU e una ESN.3211Codice python per costruire un modello utilizzando il Custom Layer LRMU.3312Codice python per la definizione del modello LRMU-ESN-R tramite subclassing dell'oggetto Model di Keras. 35

Chapter 1

Introduzione

Il *Machine Learning* moderno si concentra sull'utilizzo delle *Reti Neurali*, in particolare sul *Deep Learning*. Questo sfrutta pienamente la capacità di composizione di layer di reti neurali per formarne una capace di esprimere relazioni più complesse tra input e output. Un esempio facilmente visualizzabile sono le reti neurali Feed-Forward che sono formate da più layer di neuroni organizzati sequenzialmente.

L'algoritmo principale impiegato per l'apprendimento è la back propagation, algoritmo fondamentale, ma che spesso richiede tempi di esecuzione molto lunghi. Per esempio, si stima che per il training di modelli molto complessi, come GPT-4, sono stati impiegati circa 90 giorni [7]. Questi tempi così prolungati dipendono non solo da questo algoritmo ma anche dall'hardware utilizzato, con il quale si può fornire una stima del consumo di energia elettrica necessario per completare il training. Proprio il consumo di corrente è uno dei grandi problemi del Machine Learning applicato e, al fine di ridurlo, è necessario ricercare modelli più efficienti sia in termini di parametri totali, il che riduce il tempo di inferenza, che in termini di tempo di training. Inoltre, se il modello è di più piccole dimensioni, è più facile istanziarlo su meno GPU, riducendo così l'hardware necessario.

Nel contesto delle Reti Neurali Ricorrenti, modelli che lavorano con serie temporali come input, per ridurre i tempi di training si può usare il paradigma del *Reservoir Computing*. Questo racchiude varie metodologie che condividono un'idea di base: dividere il modello in due parti. La prima parte è chiamata *reservoir*, ed è la parte del modello che viene inizializzata e mantenuta fissa per tutto il processo di training, e la seconda parte, invece, è il *read-out*, che diversamente dall'altra viene allenata e si occupa di leggere i valori del *reservoir* in modo da poter calcolare il valore da predire o la classe corretta, a seconda del tipo di problema. L' allenamento di questo tipo di modello, rispetto ad altri, come le *Long Short Term Memory* e le *Simple RNN* di dimensioni comparabili, è molto più veloce poiché il numero di parametri da allenare è inferiore, riducendo così il tempo di training e, di conseguenza, il

consumo di corrente.

Un'altra problematica comune delle *reti neurali ricorrenti* è la difficoltà nel mantenere della memoria a lungo termine, che è utile a risolvere problemi dove sono presenti dipendenze temporali tra time step molto distanti tra loro, ovvero tra eventi distanti nel tempo. Per poter affrontare questo problema esistono molteplici approcci, tra i quali spicca il *Legendre Memory Unit* [15], un'architettura che divide il modello in modo esplicito in due componenti: una parte lineare, che massimizza la capacità di memoria, e una parte non lineare, che permette al modello di esprimere relazioni complesse tra i dati e la memoria. Questo modello ha dimostrato di essere capace di mantenere una memoria anche oltre a 100.000 time step, motivo per cui è stato preso sotto analisi per questa tesi.

Questa tesi, infatti, si concentra sull'implementazione e la sperimentazione di alcune varianti della *Legendre Memory Unit* basate sul paradigma del *Reservoir Computing*, ponendo enfasi sul confronto tra i modelli proposti e il modello originale. Questo approccio ha l'obiettivo di unire la capacità di memoria della *Legendre Memory Unit* e la velocità di addestramento delle reti reservoir.

Struttura della tesi

Questa tesi si apre con il capitolo 2, in cui vengono forniti alcuni approfondimenti teorici e alcune informazioni metodologiche, utilizzate per l' implementazione dei modelli e per la sperimentazione. Successivamente, nel capitolo 3, vengono esposti i vari modelli sviluppati e implementati come progetto di tesi. Nel capitolo 4 vengono poi presentati gli esperimenti eseguiti per misurare le performance dei modelli utilizzando alcuni problemi di benchmark. Infine, nel capitolo 5, vengono tratte le conclusioni finali della sperimentazione e i possibili sviluppi futuri che si potrebbero fare partendo da questi modelli.

Chapter 2

Background

In questo capitolo vengono presentate le basi teoriche necessarie per comprendere il contenuto sperimentale di questa tesi. La sezione 2.1 introduce i concetti fondamentali del *Machine Learning*, con un particolare focus sull' *apprendimento supervisionato*, spiegando le sue caratteristiche principali e i tipi di problemi che può risolvere.

Successivamente, nella sezione 2.2, vengono approfondite le *reti neurali*, con una distinzione tra le reti *Feed-forward* e le *reti ricorrenti* (RNN), evidenziando le loro architetture e modalità di funzionamento.

Mentre la sezione 2.2 si focalizza sul *Reservoir Computing*, un paradigma utile per risolvere problemi comuni delle reti neurali ricorrenti. Nel dettaglio vengono descritte le *Echo State Network*, uno degli approcci classici di questo paradigma e oggetto di questa tesi.

A seguire, nella sezione 2.4, si introduce la *Legendre Memory Unit* (LMU), un tipo architettura di rete neurale ricorrente che si è dimostrato capace di gestire meglio, rispetto alle normali RNN, le dipendenze temporali lontane. L'architettura LMU rappresenta l'oggetto principale della modifica e sperimentazione svolte in questa tesi.

Infine, la sezione 2.5 fornisce una panoramica su concetti generali circa i framework di macchine learning per reti neurali e Keras, il framework utilizzato per l'implementazione di ciò che verrà poi presentato nel capitilo dell'analisi e dei risultati.

2.1 Machine Learning

L'***Intelligenza Artificiale*** (***IA***) è un campo dell'informatica che mira a creare sistemi capaci di svolgere compiti che, se eseguiti da esseri umani, richiederebbero intelligenza.

Il ***machine Learning*** (***ML***) è un sotto insieme dell'*IA* ed è termine ombrello per gli algoritmi che permettono a una macchina di imparare. Non ha una unica definizione precisa. Tra le definizioni che ha assunto negli anni si può

citare:

1. Arthur Samuel con "***A field of study that gives computers the ability to learn without being explicitly programmed***" [14]

2. Tom Mitchell con "***A computer program is said to learn from experience (E) with respect to some class of tasks (T) and performance measure (P), if its performance at tasks in T, as measured by P, improves with experience E***" [10]

3. Ethem Alpaydin con "***Programming computers to optimize a performance criterion using example data or past experience***" [1].

In generale possiamo dire che il machine learning si occupa di algoritmi che danno la capacità a un computer di apprendere dalle osservazioni che può fare.

Questi algoritmi si possono racchiudere in 3 categorie principali [13]:

1. ***Reinforcement learning***: in cui l'algoritmo impara attraverso dei feedback interpretati come un punteggio. Il designer deve definire prima del learning le regole con cui vengono assegnati i punti. Queste regole, chiamate policy, influenzano ciò che l'algoritmo imparerà in quanto questo adatta il suo comportamento per massimizzare il punteggio. Si utilizza in situazioni in cui si devono prendere delle decisioni come ad esempio nel gioco degli scacchi.

2. ***Supervisionato***: in cui l'algoritmo impara da un *training set* di dati etichettati. Ognuno di questi dati è una coppia input-output (x_i, y_i). L'algoritmo impara un'approssimazione della funzione che mappa gli input agli output che viene utilizzata per predire l'output y corrispondente a un nuovo input x non visto durante l'addestramento. È utilizzato per i compiti di *classificazione* e *regressione*.

3. ***Non supervisionato***: in cui l'algoritmo cerca di trovare modelli o strutture all'interno di dati non etichettati. L'obiettivo principale è scoprire somiglianze o gruppi nei dati, ed è spesso utilizzato per compiti come il clustering.

Per comprendere al meglio il contenuto di questa tesi è importante approfondire gli algoritmi di *apprendimento supervisionato* e i problemi affrontabili con essi.

2.1.1 Apprendimento supervisionato

L'***apprendimento supervisionato*** è definito come segue:
dato un *Training set*, definito come una serie di punti

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

dove ogni punto è generato da una funzione $y_i = f(x_i)$, trovare una funzione h, detta *ipotesi*, che è una approssimazione della funzione vera f.

L'apprendimento è definito come la ricerca di un' *ipotesi* h nello *spazio delle ipotesi* $\mathcal{H}$ in grado di *generalizzare* [13]. Dove lo *spazio delle ipotesi* $\mathcal{H}$ è definito come tutte le possibili ipotesi che si possono esprimere con il modello usato e con *generalizzare* si intende la capacità di performare bene su dati non presenti nel *training set* rispetto alle metriche scelte.

Formalmente, una metrica è una funzione chiamata *funzione di loss* l e si definisce il *valore di loss* come

$$Loss = \sum_{i}^{n} l(f(x_i), h(x_i)) \tag{2.1}$$

Il valore di *Loss* quantifica quanto il modello erra nella predizione dell' output. Si deve scegliere una funzione di loss adatta al problema che si vuole affrontare in modo che un basso valore di *Loss* correli con un miglior risultato. Infatti, un algoritmo di *apprendimento supervisionato* minimizza la loss scelta con l'obiettivo di selezionare un'ipotesi h tra tutte le ipotesi possibili in $\mathcal{H}$.

I problemi affrontabili con l'*apprendimento supervisionato* si dividono in due classi:

1. **Classificazione**: dove l'input deve essere mappato in una o più delle $n \in \mathbb{N}$ categorie disgiunte; se le categorie sono due la classificazione viene detta binaria altrimenti ennaria.

2. **Regressione**: dove l'input viene mappato in uno o più numeri continui.

Per queste due classi di problemi si conoscono delle funzioni di loss adatte alla loro risoluzione. Ad esempio, per la *regressione*, è comunemente usata la *Mean Squared Error* (**MSE**): $\frac{\sum_{i=1}^{n}(\hat{y}-y_i)^2}{n}$, mentre, per la classificazione ennaria, si usa la *sparse categorical crossentropy* o la *categorical crossentropy* a seconda dell'encoding delle lable.

2.2 Reti neurali

Le ***reti neurali*** sono una classe di *modelli parametrici* di machine learning la cui struttura è ispirata a quella del cervello umano. Queste sono composte da multiple unità di calcolo connesse attraverso dei pesi adattivi [11]. Le

unità di calcolo sono entità che accettano degli input e restituiscono degli output. Nelle reti neurali, queste unità sono i neuroni artificiali, da cui deriva il nome. I neuroni sono connessi tra loro, ovvero l'output di un neurone è una parte dell' input dei neuroni a cui è connesso e, a ogni connessione, è associato un certo peso che ne definisce l'importanza. L'organizzazione delle connessioni definisce l'architettura della rete neurale. Il funzionamento dei neuroni è ispirato a quello dei neuroni biologici ed è formalizzato nel seguente modo:

dati un vettore di input $\mathbf{x}$, un valore numerico $\mathbf{b}$ chiamato *bias*, un vettore di pesi $\mathbf{w}$ e una funzione di attivazione $\sigma(x)$, l'output del neurone, detto *valore di attivazione*, è calcolato come

$$y = \sigma(\mathbf{x} \cdot \mathbf{w} + b) = \sigma\left(\sum_{i=1}^{n} x_i w_i + b\right) \tag{2.2}$$

ovvero è una somma degli input $\mathbf{x}$ pesata con i pesi delle connessioni $\mathbf{w}$. Il *bias* funge da input extra opzionale. I pesi $\mathbf{w}$ e il *bais* b sono detti *parametri* del modello e vengono adattati durante il processo di allenamento.

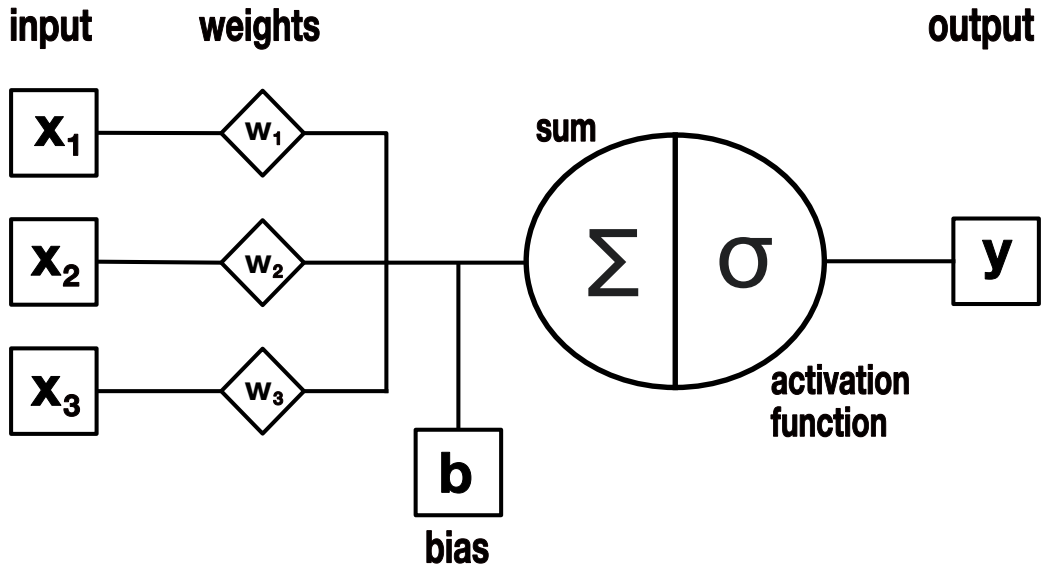

Figure 2.1: Schema di un neurone artificiale.

La scelta della funzione di attivazione $\sigma(x)$ è fondamentale, poiché questa, insieme all'architettura, definisce l'*espressività* della rete neurale dove, per *espressività*, si intende la grandezza dello spazio delle ipotesi $\mathcal{H}$. La funzione

$\sigma(x)$ può essere un hard threshold o una qualche altra funzione non lineare come $ReLU(x)$ o $Sigm(x)$.

Comunemente le *reti neurali* vengono allenate con algoritmi di *apprendimento supervisionato* e apprendere significa adattare i *parametri* del modello in modo da minimizzare la *funzione di loss*, ciò si ottiene tramite *gradiend descent*.

2.2.1 Reti neurali Feed-forward

Le ***reti neurali Feed-Forward*** (***FF***) sono delle *reti neurali* dove l'architettura è organizzata a stack di neuroni chiamati *layer*. In un layer sono presenti un numero arbitrario di neuroni e tra questi non ci sono connessioni. I *layer* sono anch'essi in numero arbitrario e sono connessi tra loro con delle sole connessioni "in avanti", ossia l'output di un *layer* è l'input del successivo. Non ci sono connessioni all'indietro o cicliche e quindi la rete risulta essere un grafo diretto aciclico. Una volta fissati i parametri della rete, questa diventa una funzione, ovvero ogni volta che le viene presentato lo stesso input si ottiene sempre lo stesso output.

La rete *Feed-Forward* più semplice che si può costruire è chiamata ***perceptron network***. Questa è composta da un *layer di input* e un *layer di output* e le connessioni sono fatte in modo da collegare direttamente l'*input* all'*output*.

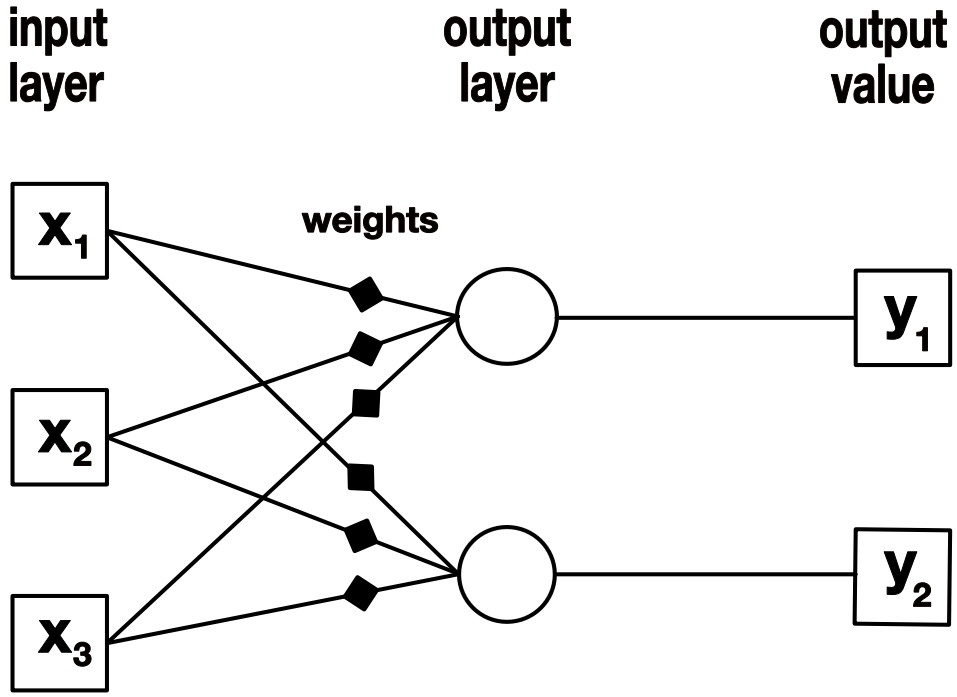

Figure 2.2: Schema di perceptron network.

In questo tipo di rete la modifica di un solo peso influisce su un solo output infatti, nel caso in cui il *layer* di output è composto da m neuroni, la rete si può decomporre in m reti separate e perciò possono essere eseguiti m processi di apprendimento indipendenti. Per l'apprendimento si utilizza la *regola del percettrone* o la *regressione logistica* a seconda che la *funzione di attivazione* sia rispettivamente un *hard threshold* o la *funzione sigm*. Una *perceptron network* funziona solo nel caso in cui i dati siano linearmente separabili. Per migliorare tale aspetto vengono utilizzate le ***multi layer perceptron*** (***MLP***), reti neurali Feed-Forward più complesse in cui vengono utilizzati n *layer*. In questo tipo di rete, oltre ai layer di *input* e *output*, si hanno altri layer interni, detti *hidden layer*.

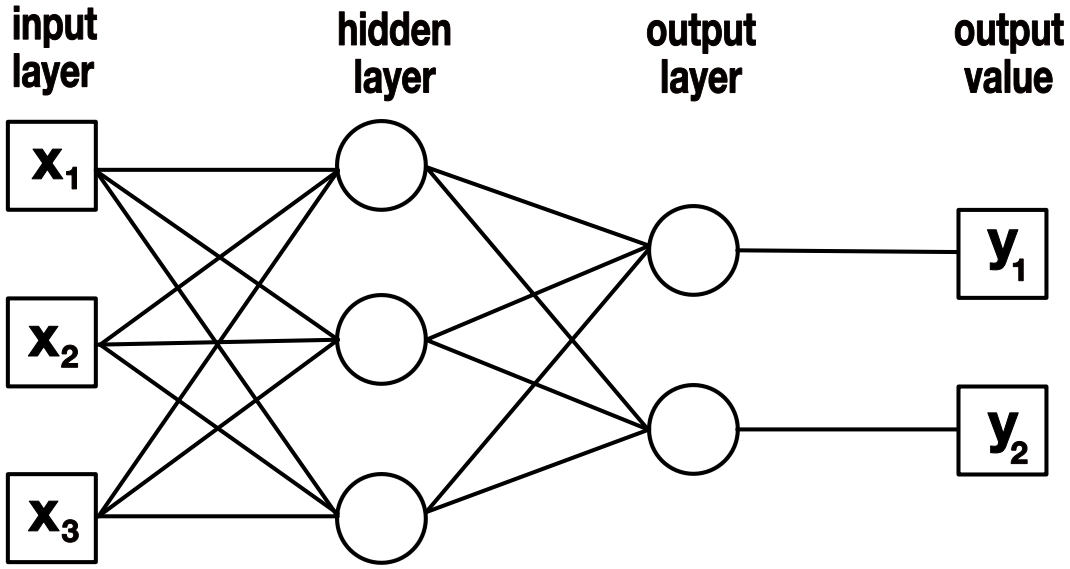

Figure 2.3: Schema di una multi layer perceptron network.

In questo tipo di rete non si può dividere il processo di apprendimento in processi indipendenti. Questo perché, anche nel caso semplice con solo un *hidden layer*, adattare i pesi per correggere un output influirebbe anche sugli altri. Per risolvere il problema si utilizza l'algoritmo di *back propagation* che prende in considerazione queste dipendenze e funziona in generale con n *hidden layer*.

Le reti neurali feed-forward costituiscono la base per il deep learning, approccio dove si utilizzano più *hidden layer* per aumentare il potere espressivo

della rete. Ogni *hidden layer* aumenta la non linearità e, componendo più layer si possono esprimere dipendenze complesse. Queste reti, infatti, vanno viste come strumenti con cui fare *regressione non lineare*.

Il funzionamento di questo tipo di reti neurali si basa sull'esistenza del *teorema di approssimazione universale*, che le rende in grado di approssimare qualsiasi funzione continua.

2.2.2 Reti neurali ricorrenti

Le ***reti neurali ricorrenti*** (***RNN***) sono una classe di *reti neurali* in cui l' architettura presenta dei cicli. In questo tipo di rete gli input sono delle sequenze di dati composte da *data point* che vengono presentati alla rete uno alla volta in ordine. L'ordine della serie è rilevante poiché può esprimere una dipendenza temporale o contestuale. Presentare un *data point* alla rete scandisce un *time step*.

Grazie alla presenza dei cicli, questa classe di reti conserva memoria, ossia la capacità della rete di generare un output che dipende non solo dall'input al time step corrente ma anche dai precedenti. Ciò avviene tramite il mantenimento di uno stato che viene aggiornato ad ogni time step e utilizzato in quello successivo per il calcolo del nuovo output e del nuovo stato.

Per via della presenza dello stato le reti ricorrenti non sono funzioni come le *Feed-Forward* ma sono dei *sistemi dinamici*, ossia sistemi che evolvono nel tempo. La loro evoluzione avviene tra un time step e l'altro e questa loro natura rende più complessa la comprensione del comportamento della rete.

Questo tipo di reti si sono dimostrate utili per risolvere problemi come il riconoscimento vocale e il riconoscimento della scrittura a mano.

A causa della definizione generica queste reti possono avere architetture variegate e, tra queste, l'architettura più semplice è la *simple RNN*. Questa è simile a una rete *Feed-Forward* con un solo *hidden layer* connesso anche a se stesso.

Per comprendere e visualizzare la rete si possono usare due rappresentazioni: quella *compatta* e la versione "*unfolded*", in cui si decompone la rete in n reti quanti i time step [3], come illustrato in figura 2.4.

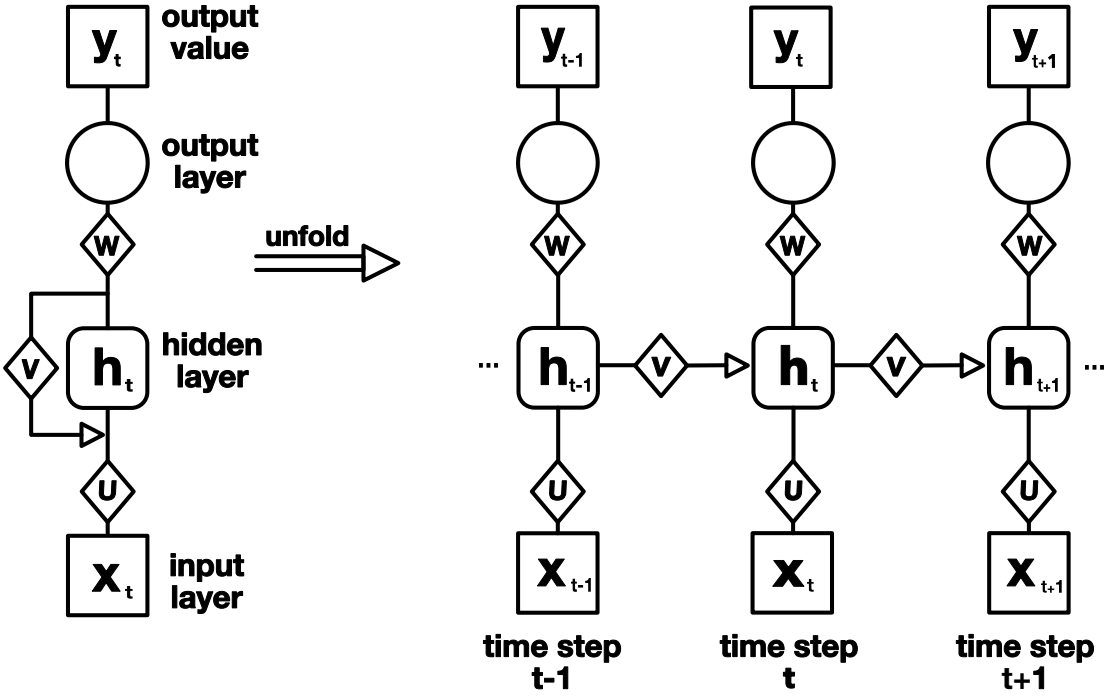

Figure 2.4: Schema di una rete simple ricorrente.

La simple RNN può essere espressa dalle equazioni:

$$\begin{aligned} \mathbf{h_t} &= \sigma_h(\mathbf{U x_t} + \mathbf{V h_{t-1}} + \mathbf{b_h}) \\ \mathbf{y_t} &= \sigma_y(\mathbf{W h_t} + \mathbf{b_y}) \end{aligned} \tag{2.3}$$

dove:

- $\mathbf{h}_t$ è lo stato calcolato dall' hidden layer al time step t .
- $\mathbf{x_t}$ è l'input e $\mathbf{y_t}$ è l'output al time step t.
- $\mathbf{V}, \mathbf{U}, \mathbf{W}$ sono le matrici dei pesi delle connessioni rispettivamente: ricorrente sull' hidden layer, tra l'input layer e l'hidden layer, e tra l'hidden layer e l'output layer.
- $\mathbf{b_h}$ e $\mathbf{b_y}$ sono i vettori di bias rispettivamente per l'hidden layer e l'output layer
- σ_h e σ_y sono le *funzioni di attivazione* rispettivamente per l'hidden layer e l'output layer.

Per allenare le *simple RNN* si usa la *back propragation through time* che è essenzialmente la *back propagation* applicata alla rete neurale ricorrente.

Nella pratica le simple RNN si sono dimostrate difficili da usare in quanto soffrono del problema del *gradient Vanishing/Exploding*, ovvero il gradiente viene moltiplicato per un numero sempre più grande o sempre più piccolo, rendendo l'apprendimento rispettivamente instabile o con step infinitesimali. Inoltre non sono adatte a problemi con dipendenze temporali remote. Questi problemi sono stati mitigati da altre architetture ricorrenti come la *Long Short Term Memory* (*LSTM*) e la *Gated Recurent Unit* (*GRU*).

2.3 Reservoir computing

Il ***reservoir computing*** (***RC***) [8] è un paradigma di *reti ricorrenti* in cui viene creata una divisione tra la parte dinamica della rete e la parte non dinamica, la quale si può esprimere con una funzione, come nelle *feed foward*. Ciò viene fatto dividendo la rete in due parti: *reservoir* e *readout*, dove la prima è la parte dinamica e la seconda quella non dinamica.

Il reservoir è un layer di neuroni, inteso diversamente da quello delle *feed foward* in quanto presenta delle connessioni interne e ricorrenti. Il reservoir non viene allenato ma se ne inizializzano i parametri randomicamente in base ad una distribuzione statistica scelta e i valori una volta selezionati non vengono mai modificati. Nella rete, il *reservoir*, ha il compito di proiettare gli input in un spazio a più alta dimensionalità in cui è più facile separarli.

Il *readout* è un layer di output che viene allenato per imparare a "interpretare" i valori del reservoir. Questo layer può essere di tipo variegato: si potrebbe usare una *feed foward* o un singolo neurone. Più comunemente però si sceglie di utilizzare una semplice *regressione lineare* con MSE come *funzione di loss*, in quanto per la minimizzazione della loss esiste una soluzione in forma chiusa.

Questo paradigma si è dimostrato utile per imparare a predire processi dinamici, come la generazione di segnali, e per la modellazione di sistemi biologici.

Tre approcci classici del reservoir computing sono le *Liquid State Machine*, le *Echo State Network* e l'algoritmo di *Backpropagation-decorrelation* [8]. Le *Echo State Network* sono parte dell'oggetto della tesi, motivo per cui è importante approfondirle.

2.3.1 Echo State Network

Le ***Echo State Networks*** (***ESN***) sono un tipo di modello che appartiene alla famiglia del *Reservoir Computing* ed è caratterizzato da un reservoir di grandi dimensioni in cui i neuroni sono connessi in modo sparso. Nella versione più generale del'*ESN* sono permessi anche i collegamenti diretti tra il layer di input e il layer di output, nonché tra i valori di output e il reservoir [4].

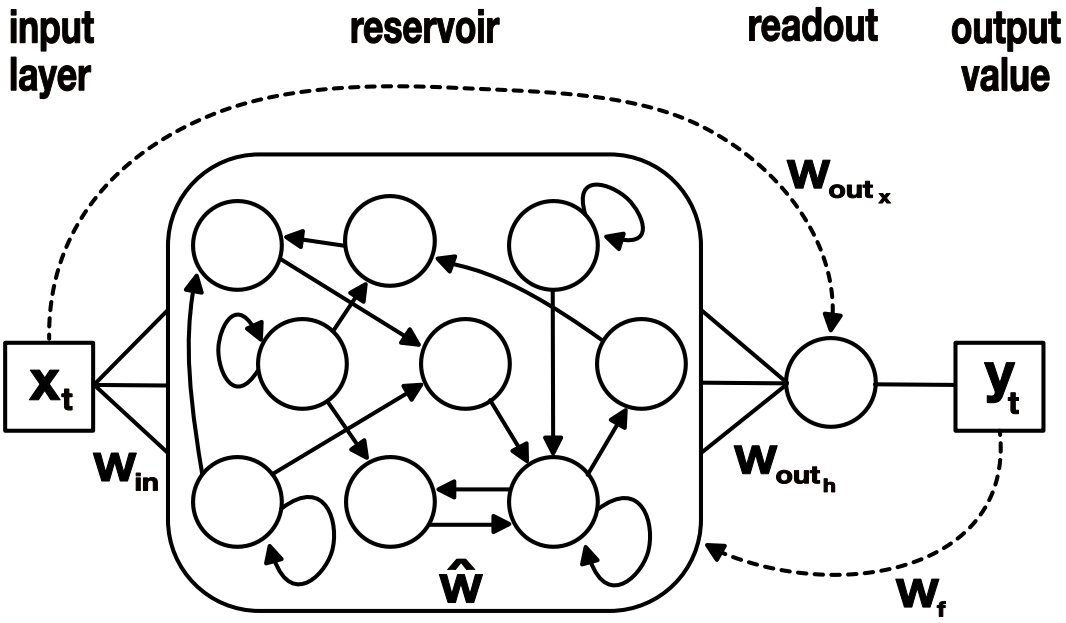

Figure 2.5: Schema Echo State Network.

Una ***Echo State Network*** può essere descritta con le seguenti equazioni:

$$\begin{aligned} \mathbf{z}_t &= \mathbf{W}_{in}\mathbf{x}_t + \hat{\mathbf{W}}\mathbf{h}_{t-1} + \mathbf{W}_f\mathbf{y}_{t-1} \\ \mathbf{h}_t &= \tanh(\mathbf{z}_t w + \mathbf{b})leaky + \mathbf{h}_{t-1}leaky \\ \mathbf{y}_t &= \mathbf{W}_{out_h}\mathbf{h}_t + \mathbf{W}_{out_x}\mathbf{x}_t \end{aligned} \tag{2.4}$$

dove:

- $\mathbf{h}_t$ è lo stato del reservoir al time step t.
- $\mathbf{x}_t$ è l'input e $\mathbf{y}_t$ è l'output al time step t.
- $\mathbf{b}$ è il bias.
- $\mathbf{W}_{in}, \hat{\mathbf{W}}, \mathbf{W}_{out_h}, \mathbf{W}_{out_x}, \mathbf{W}_f$ sono le matrice dei pesi delle connessioni rispettivamente: tra input e reservoir, ricorrenti per il reservoir, tra reservoir e output, tra input e output e tra ouput e reservoir.
- *leaky* è un iperparametro del modello.

Il *reservoir*, essendo inizializzato in modo randomico, potrebbe presentare una dinamica instabile, ovvero valori che possono risultare caotici, poco coerenti tra loro e difficili da prevedere. Per evitare ciò, è fondamentale per il corretto funzionamento delle ESN la ***Echo State Property*** (***ESP***) [5] che

assicura che, dopo un certo numero di time step, lo stato del reservoir dipenda principalmente dagli input più recenti piuttosto che dalle condizioni iniziali. Questo fenomeno, noto anche come *fading memory*, implica che le influenze degli stati passati svaniscano gradualmente nel tempo, permettendo alla rete di "dimenticare" gli input remoti e dare più importanza a quelli recenti. Ciò è vantaggioso per stabilizzare la rete e impedire l'accumulo di errori. Per via della *ESP*, le *ESN* soffrono di limitazioni nei casi in cui sono presenti dipendenze remote e per cercare di risolvere questo problema sono state provate delle architetture ibride.

Gli **iperparametri** di un modello ESN che influenzano sulla performance della rete sono i seguenti:

- **Raggio spettrale**: corrisponde al raggio spettrale della matrice ricorrente del reservoir $\rho(\hat{\mathbf{W}})$. Comunemente è impostato a poco meno di 1 siccome si è visto sperimentalmente che questo assicura la *ESP* nella maggior parte dei casi.
- **Input scaling**: un numero reale $scaler_{in}$ che influisce sull'inizializzazione della matrice di connessione tra input e reservoir $\mathbf{W}_{in}$. Questa viene inizializzata prendendo valori da una distribuzione uniforme tra $[-scaler_{in}, scaler_{in}]$
- **Bias scaling**: un numero reale $scaler_b$ che influisce sull'inizializzazione del vettore di bias che viene inizializzato prendendo anch'esso i valori da una distribuzione uniforme tra $[-scaler_b, scaler_b]$
- **Leaky**: un numero reale tra 0 e 1 che controlla la velocità di reazione della ESN agli input; un valore basso implica una risposta lenta mentre con 1 si ottiene una risposta immediata.

Readout

Il ***readout*** è la parte del modello che estrae l'output dal reservoir ed è rappresentata dalla matrice dei pesi $\mathbf{W_{out}}$ che collega il reservoir all'output. $\mathbf{W_{out}}$ è l'unica matrice di pesi che viene effettivamente allenata. È pratica comune usare come funzione di loss una last squere definendo quindi il problema di minimizzazione dell'errore come:

$$\min_{\mathbf{W}_{out}} \|\mathbf{W}_{out}\mathbf{H} - \mathbf{Y}\|_2^2 \tag{2.5}$$

Per risolverlo si utilizza un modello lineare, come la *ridge regression*, che rende l'addestramento molto efficiente siccome, per minimizzare l'errore, esiste la forma chiusa [8]:

$$\mathbf{W}_{out} = \mathbf{Y}\mathbf{H}^T(\mathbf{H}\mathbf{H}^T + \alpha^2\mathbf{I})^{-1} \tag{2.6}$$

dove:

- $\mathbf{H}$ e $\mathbf{Y}$ sono rispettivamente vettori: degli stati e dei valori target a ogni time step t, quindi $\mathbf{H} = [h_1, \ldots, h_n]$ e $\mathbf{Y} = [y_1, \ldots, y_n]$. Il primo valore h_0 viene scartato in quanto contaminato dai valori iniziali del reservoir; di conseguenza anche y_0 viene scartato.
- $\mathbf{I}$ è la matrice identità
- α è un coefficiente di regolarizzazione, iperparametro del modello. Un valore più alto implica un *readout* più regolarizzato, ovvero più penalizzato all'aumentare della dimensione del modello lineare.

2.4 Legendre Memory Unit

Le ***Legendre Memory Unit*** (LMU) sono un tipo di architettura di *rete neurale ricorrente* definita da due parti, in cui una ha il ruolo di "memoria" mentre l'altra quello di fare la "computazione" non lineare. La parte di memoria è implementata attraverso una cella lineare utilizzando i polinomi di Legendre, mentre la computazione non lineare può essere realizzata in vari modi tra cui usando un singolo neurone, una simpleRNN o una LSTM. Questa strategia permette alla rete di catturare dipendenze su gli input anche con time step $T > 100.000$ [15].

Un layer LMU accetta un vettore di input $\mathbf{x_t}$ e genera uno stato nascosto $\mathbf{h}_t \in \mathbb{R}^n$ e una memoria $\mathbf{m}_t \in \mathbb{R}^d$. Questi saranno usati per computare funzioni non lineari dell'input. Il layer LMU è descritto dalle seguenti equazioni:

$$\begin{aligned} \mathbf{h}_t &= \sigma(\mathbf{W}_x \mathbf{x}_t + \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_m \mathbf{m}_t) \\ \mathbf{u}_t &= \mathbf{e}_x^\top \mathbf{x}_t + \mathbf{e}_h^\top \mathbf{h}_{t-1} + \mathbf{e}_m^\top \mathbf{m}_{t-1} \\ \mathbf{m}_t &= \overline{\mathbf{A}} \mathbf{m}_{t-1} + \overline{\mathbf{B}} u_t \end{aligned} \tag{2.7}$$

Dove:

- $\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_t$ sono rispettivamente il vettore di input, l'hidden state e memoria al time step t
- $\mathbf{W}_x$, $\mathbf{W}_h$, $\mathbf{W}_m$ sono le matrici dei pesi che collegano rispettivamente l'input all'hidden layer, l'hidden layer a se stesso e la memoria all'hidden layer
- $\mathbf{e}_x$, $\mathbf{e}_h$, $\mathbf{e}_m$ sono i vettori di pesi che combinano rispettivamente l'input, lo stato nascosto precedente e lo stato di memoria precedente nel calcolo di u_t, un numero intermedio al time step t
- $\overline{\mathbf{A}}$ e $\overline{\mathbf{B}}$ sono matrici di pesi che definiscono la dinamica del vettore di memoria $\mathbf{m}_t$
- $\sigma(\cdot)$ è una *funzione di attivazione* non lineare.

Avere diverse matrici di pesi permette di disaccoppiare la dimensione dello stato nascosto n dalla dimensione della memoria d, e richiede di mantenere $n + d$ variabili in memoria tra time step.

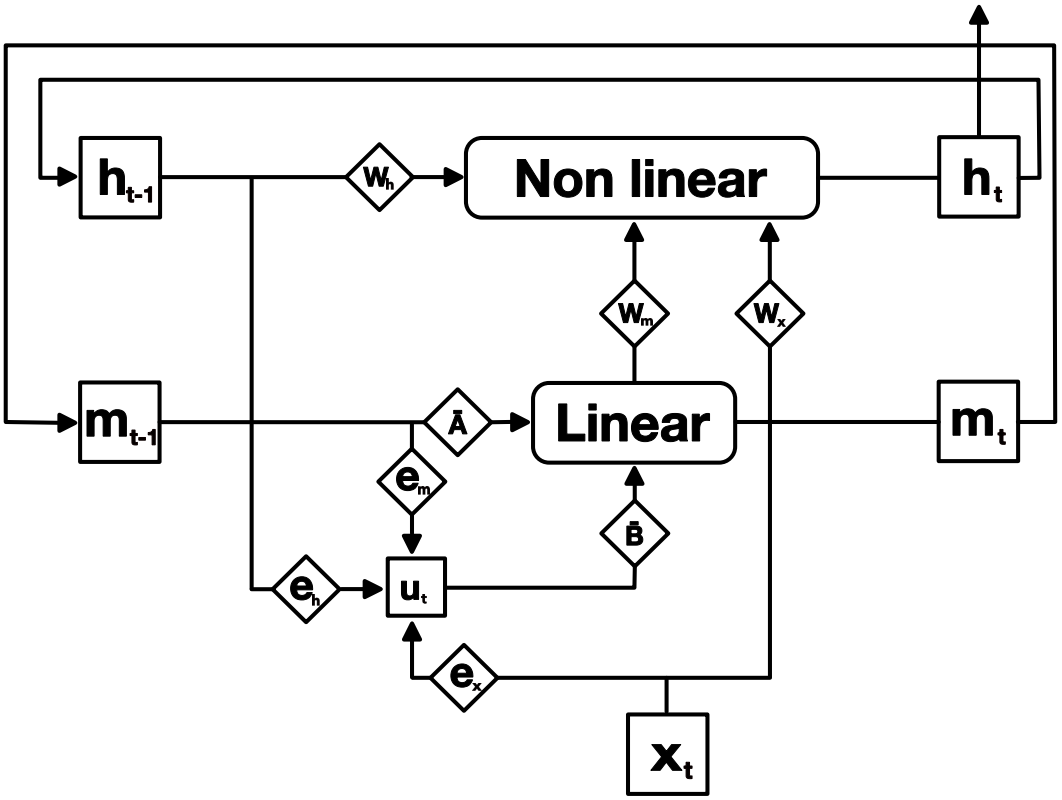

Figure 2.6: Schema compatto di una layer LMU.

2.5 Metodologia

In questa sezione verranno illustrati i concetti metodologici per l' implementazione dei modelli di machine learning con reti neurali. La prima sotto sezione presenta dei concetti generali mentre nella successiva si descrivono i framework TensorFlow e Keras, usati per l'implementazione dei modelli discussi in questa tesi.

2.5.1 Concetti generali

Per comprendere l'allenamento dei modelli di *machine learning* basati su *reti neurali* è importante conoscere alcuni concetti generali applicabili in tutti i framework.

Tensori

I ***tensori*** sono la struttura dati principalmente utilizzata per l'elaborazione e l'archiviazione dei dati, come i parametri del modello, gli input e gli output dei vari layer.[2]

I *tensori* sono caratterizzati da due proprietà principali: la dimensione (o rango) e la forma (o shape). La dimensione di un tensore si riferisce al numero di indici anche detti assi, necessari per individuare un singolo elemento all'interno del tensore, mentre, la forma di un tensore è una ennupla di numeri interi che specifica l'indice massimo lungo ciascun asse del tensore.

I *tensori* sono una generalizzazione di vettori e matrici, infatti un vettore è un *tensore* a 1 dimensione e una matrice è un *tensore* a 2 dimensioni e in questo caso i due indici sono di riga e di colonna.

Tra le possibili operazioni tra tensori, è importante conoscere la moltiplicazione matriciale, calcolata tra tensori e usata principalmente per calcolare i valori di *attivazione dei neuroni*. Per essere applicata ha bisogno di rispettare determinati criteri sulla forma dei tensori coinvolti nell'operazione, motivo per cui esiste anche l'operazione di *reshape*, che riorganizza in modo diverso gli elementi all'interno del tensore mantenendo invariati sia i valori che il numero di elementi e permette di soddisfare i requisiti necessari per fare l'operazione di moltiplicazione matriciale.

Overview dal training all'inferenza

Quando si allena un modello di macchine learning esiste una procedura generale. Prima di tutto bisogna dividere i dati a disposizione in tre sotto insiemi disgiunti, chiamati *training set*, *validation set* e *test set*. Ogni set viene usato per uno scopo diverso: il primo per il training, il secondo per la *model selection* e il terzo per valutare la performance finale del modello selezionato [2].

La *model selection* è il processo di selezione del modello più appropriato per un dato problema di machine learning. Questo processo include la scelta dell'architettura della rete e l'ottimizzazione degli iperparametri, ovvero quei parametri che definiscono il modello, ad esempio, nelle feed-forward, il numero di hidden layer e il numero di neuroni per ogni layer.

La *model selection* avviene in diverse fasi: dopo aver scelto l'architettura del modello vengono cambiati gli iperparametri, viene allenato il modello sul training set e ne vengono valutate le performance sul validation set. Se queste non sono soddisfacenti, si riesegue il processo ricambiando gli iperparametri, altrimenti è terminato. Una volta completata la procedura, avremo un modello con iperparametri ottimizzati sul validation set. Per valutare la performance reale del modello è pratica comune riallenarlo da zero utilizzando sia i dati del training set che quelli del validation set. Infine si testano le performance sul test set.

Questa procedura è necessaria in quanto utilizzare un validation set ci permette di selezionare un modello che non vada in over-fitting o under fititng, ovvero si seleziona un modello capace di generalizzare.

Dopo aver misurato le performance del modello selezionato, questo si può utilizzare per fare inferenza sui nuovi dati.

2.5.2 TensorFlow e Keras

Per l'implementazione pratica dei modelli usati in questa tesi sono stati usati i framework ***TensorFlow*** e ***Keras***. Il primo è un framework di computazione tra tensori che è stato usato per eseguire efficientemente i calcoli tra tensori su GPU necessari per l'addestramento e le inferenze mentre, il secondo, come interfaccia ad alto livello per l'implementazione dei modelli e dei nuovi layer [2].

Keras è inizialmente nato come framework indipendente open source ed è poi stato totalmente adottato da *TensorFlow*. Il codice del Framework può essere trovato al link `https://github.com/keras-team/keras`.

Api Funzionali

Grazie a *Keras* la composizione dei modelli può essere effettuata utilizzando le API funzionali, che consentono di creare architetture complesse e modulari. Un esempio di modello che classifica le immagini del MNIST, costruito con API funzionali, è fornito nel listato 1.

```python
#Definizione dell'input layer
inputs = tf.keras.Input(shape=(784,))
#Definizione dell'hidden layer
feature = tf.keras.layers.Dense(64, activation='relu')(inputs)
#Definizione dell'output layer per la classificazione
outputs = tf.keras.layers.Dense(10, activation='softmax')(feature)
#Creazione del modello
model = tf.keras.Model(inputs=inputs, outputs=outputs)
#Spicifica dell'ottimizzatore e della funzione di loss da utilizzare e le metriche di cui tenere traccia
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

Listing 1: Codice python per costruire un modello con api funzionali. Nello specifico questo modello è composto da layer di input da 784 unità una per ogni pixel di un'immagine del dataset, un hidden layer da 64 unità con *funzione di attivazione* ReLu e layer di output da 10 unità con *funzione di attivazione* Softmax, utile per fare la classificazione multi classe.

Per istanziare il modello si crea un oggetto "modello" di Keras specificando il layer di input e di output. Successivamente questo viene finalizzato richiamando il metodo compile, utile per specificare *l'ottimizzatore* e la *funzione di loss* da usare, e anche le metriche di cui il framework deve tenere traccia duratene l'addestramento e il test. Queste metriche, a differenza della loss, possono servire all'utente e non sono usate dall' ottimizzatore.

Allenamento e Testing

Per l'allenamento del modello, il testing e l'inferenze si utilizzano funzioni standard, rispettivamente *fit*, *evaluate* e *predict*. La funzione fit presenta anche delle funzionalità per fare la *model selection* usando il validation set.

```python
# Divisione del dataset in training set e validation set
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
↪  test_size=0.2)
# Allenamento del modello con validation set
model.fit(x_train, y_train, epochs=5, validation_data=(x_val, y_val))
# Valutazione del modello sui dati di test
model.evaluate(x_test, y_test)
# Previsione sui nuovi dati
predictions = model.predict(x)
```

Listing 2: Codice python per allenare un modello testarlo e farci inferenza. Continuazione del listato 2.

Custom Layer

In *Keras*, per implementare un nuovo layer, si utilizza il subclassing della classe Layer. A questo scopo è necessario fare l'override dei metodi *build* e *call*. Il metodo *build* serve a inizializzare i pesi del layer e viene richiamato dal framework, accettando come parametro la "*shape*" del tensore di input. Il metodo *call*, invece, viene utilizzato per eseguire i calcoli del layer.

```python
import tensorflow as tf

# Definizione di un layer custom trammite subclassig di Layer
class MyCustomLayer(tf.keras.layers.Layer):
    def __init__(self, units=32):
        super(MyCustomLayer, self).__init__()
        self.units = units

    def build(self, input_shape):
        # Aggiungiunta di un peso al modello
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True<)

    def call(self, inputs):
        # Operazione del layer
        return tf.matmul(inputs, self.w)
```

Listing 3: Codice python per costruire un custom layer.

Ricerca degli iperparametri

In *Keras* la ricerca degli iperparametri migliori può essere automatizzata grazie a un modulo chiamato *Keras-Tuner*, che ne implementa le funzionalità

necessarie.

Per condurre questa ricerca, è necessario definire un *hypermodel* utilizzando il subclassing della classe *HyperModel* di Keras-Tuner. L'hypermodel stabilisce quali iperparametri devono essere ottimizzati, specificati attraverso l'oggetto *HyperParameter*. Questo oggetto viene passato all'hypermodel tramite la funzione *build*, di cui è necessario fare l'override nell'implementazione del hypermodel [12].

```python
from keras_tuner import HyperModel, RandomSearch
import tensorflow as tf

# Definizione della classe HyperModel per la ricerca degli iperparametri
class MyHyperModel(HyperModel):
    def build(self, hp):

        inputs = tf.keras.Input(shape=(784))
        # Definizione di un iperparametro per il numero di unita' nel
        ↪ layer denso
        units = hp.Int('units', min_value=32, max_value=512, step=32)
        feature = tf.keras.layers.Dense(64, activation='relu')(inputs)

        # Aggiunta di un altro layer denso con 10 unita' per la
        ↪ classificazione
        outputs = tf.keras.layers.Dense(10,
        ↪ activation='softmax')(feature)

        # Creazione e compilazione del modello
        model = tf.keras.Model(inputs=inputs, outputs=outputs)
        model.compile(
            optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )

        return model
```

Listing 4: Codice python per costruire HyperModel.

Questo hypermodel, grazie al metodo *Int* dell'oggetto *hyperParameter*, *hp* nel codice, definisce un modello con un solo iperparametro chiamato "unit", che rappresenta il numero di unità dell' hidden layer. L'oggetto hp espone anche altri metodi per gestire altri modi di selezionale gli altri iperparamentri. Alcuni esempi sono: *float*, *Boolean* e *choice*. Quest'ultimo permette di scegliere un valore tra una lista di possibili valori.

Una volta definito un hypermodel, si avvia una delle diverse tipologie di ricerca nello spazio dei parametri, cioè l'insieme di tutti i valori possibili che

gli iperparametri possono assumere. La più semplice di queste è la *RandomSearch*, che esplora configurazioni casuali fino a quando non esaurisce il numero di tentativi disponibili o lo spazio degli iperparametri, con l'obiettivo di trovare la combinazione ottimale rispetto a una metrica obiettivo. Nel listato 5 si ha un esempio con Keras-Tuner.

```
# Creazione di un'istanza di RandomSearch per la ricerca degli iperparametri
tuner = RandomSearch(
    MyHyperModel(),
    objective='val_accuracy',   # Obiettivo da ottimizzare
    max_trials=10,              # Numero massimo di configurazioni da provare
    executions_per_trial=2,     # Numero di esecuzioni per ciascuna configurazione
    directory='my_dir',         # Directory per salvare i risultati
    project_name='hyperparam_search'
)

# Avvio della ricerca degli iperparametri
tuner.search(
    x_train, y_train, # Dati per il training
    epochs=5,
    validation_data=(x_val, y_val) # Dati sui cui fare la validazione
)

# Recupero del miglior modello trovato dalla ricerca
best_model = tuner.get_best_models(num_models=1)[0]
```

Listing 5: Codice python per avviare una ricerca randomica degli iperparametri sul hypermodel.

In questo modo possiamo salvare solo il modello migliore e utilizzarlo ai nostri scopi.

Chapter 3

Modelli

In questo capitolo introduciamo i modelli che sono stati considerati nella fase di sperimentazione di questa tesi. In ogni sezione parleremo del relativo modello approfondendone l'aspetto concettuale e l'implementazione pratica utilizzando le interfacce del framework *Keras*. I modelli sono presentati in ordine decrescente sul numero di pesi che allenano spostandoci sempre di più verso un modello che rispetta i paradigmi del *Reservoir Computing*. Si parte infatti da un primo modello che, a parità di pesi presenti, ne allena un numero maggiore rispetto al successivo e cosi via. Nella sezione 3.1 introduciamo la *Legendre Reservoir Memory Unit* (*LMRU*), una variante del LMU dove non si allenano gli encoder, nella sezione 3.2 parliamo del *LMU-ESN* che utilizza un layer LMU che ha come cella non lineare una *ESN*, nella sezione 3.3 parliamo del *LRMU-ESN*, un modello che utilizza il layer LRMU e che ha come cella non lineare una *ESN*. Infine nella sezione 3.4 viene presentato il modello LRMU-ESN-R ovvero un modello simile al precedente ma che utilizza una ridge regression come readout. Il codice mostrato può essere trovato al link GitHub https://github.com/VitoBarra/Keras-LRMU.

3.1 Legendre Reservoir Memory Unit

Il modello ***Legendre Reservoir Memory Unit*** (***LRMU***) è un modello che utilizza il layer *LRMU*, il quale deriva dal layer *LMU* introdotto nella sezione 2.4. Di questo se ne mantiene l'architettura ma se ne modificano alcune caratteristiche. In particolare, nel *LRMU*, gli encoder vengono inizializzati seguendo la metodologia usata per l'inizializzazione della matrice di input nell' ***ESN*** vista nella sezione 2.3.1. Ovvero, viene scelto un valore di scaling e si inizializza l'encoder randomicamente con una distribuzione uniforme con valori compresi tra $[-scaler, scaler]$ e nella successiva fase di training questi non vengono alleanti e lasciati quindi fissi. Questa stessa procedura viene anche eseguita per il bias. Il valore di scaling è un iperparametro del modello ed essendo 3 gli encoder, di cui 2 opzionali, più il bias, anche esso opzionale,

il numero di iperparametri aumenta a 4, in quanto ogni valore di scaling è scelto indipendentemente dagli altri.

Implementazione

Per implementare il modello *LRMU* va prima implementato l'omonimo layer *LRMU*. Ciò è stato fatto tramite il subclassig della classe *Layer* di *Keras*, come visto nella sezione 2.5.2. Il codice di questo layer, mostrato nel listato 6, è derivato dall' implementazione del Layer LMU con l'aggiunta di funzionalità per l'inizializzazione degli encoder con i relativi scaler.

Partiamo dall'inizializzazione che è identica all' inizializzazione del LMU con l'aggiunta dei 4 valori di scaling. Il costruttore assegna i valori alle variabili *state_size* e *output_size* per permettere a Keras di conoscere la shape degli input e degli output del layer. Ogni parametro che viene passato al costruttore viene salvato nei campi della classe per essere utilizzato negli altri metodi; ciò non è mostrato per brevità.

```
1
2  def __init__(self, memoryDimension, order, theta, hiddenCell=None,
3               hiddenToMemory=False, memoryToMemory=False,
                 ↪  inputToHiddenCell=False, useBias=False,
4               memoryEncoderScaler=1.0, hiddenEncoderScaler=1.0,
                 ↪  inputEncoderScaler=1.0, biasScaler=1.0,
5               seed=0, **kwargs):
6      super().__init__(**kwargs)
7      ... # Assegnazione dei valori ai campi della classe
8      if self.HiddenCell is None:
9          if self.HiddenToMemory:
10             raise ValueError(
11                 "hiddenToMemory must be False if hiddenCell is None"
12             )
13         self.HiddenOutputSize = self.MemoryDim * self.Order
14         self.HiddenStateSize = []
15     elif hasattr(self.HiddenCell, "state_size"):
16         self.HiddenOutputSize = self.HiddenCell.output_size
17         self.HiddenStateSize = self.HiddenCell.state_size
18     else:
19         self.HiddenOutputSize = self.HiddenCell.units
20         self.HiddenStateSize = [self.HiddenCell.units]
21
22     self.state_size = [self.MemoryDim * self.Order] +
       ↪  tf.nest.flatten(self.HiddenStateSize)
23     self.output_size = self.HiddenOutputSize
24
25
```

Listing 6: Codice python dell'inizializzazione del Custom Layer LRMU.

Procediamo poi con il metodo *build*, mostrato nel listato 7, implementato in modo da gestire tutte le varie combinazioni delle possibili connessioni. Il metodo utilizza le funzioni *createWeight*, per gestire l'inizializzazione dei vari pesi utilizzando il relativo valore di scaling, e *GenerateABMatrix*, per generare le matrici **A** e **B** discretizzando con il metodo *Zero-order hold* (zoh). Gli encoder per l'input e per l'output della cella non lineare vengono concatenati per costruire il *MemoryKernel*, ovvero un unico tensore utilizzato in questa forma per motivi di efficienza di calcolo. Il metodo imposta anche un *seed* in modo da assicurare la riproducibilità dei test.

```

def GenerateABMatrix(self):
    # Calcola le matrici A e B
    Q = np.arange(self.Order, dtype=np.float64)
    R = (2 * Q + 1)[:, None]
    j, i = np.meshgrid(Q, Q)
    A = np.where(i < j, -1, (-1.0) ** (i - j + 1)) * R
    B = (-1.0) ** Q[:, None] * R

    # Discretizza le matrici A e B usando il metodo Zero Order Hold (ZOH)
    self._base_A = tf.constant(A.T, dtype=self.dtype)
    self._base_B = tf.constant(B.T, dtype=self.dtype)
    self.A, self.B = M._cont2discrete_zoh(self._base_A / self.Theta,
     ↪  self._base_B / self.Theta)

def createWeight(self, shape, scaler=1.0):
    # Inizializza i pesi utilizzando una distribuzione uniforme con
     ↪  limiti determinati da 'scaler'
    initializer = RandomUniform(minval=-scaler, maxval=scaler, self.Seed)
    # Crea un tensore non addestrabile (trainable=False)
    return self.add_weight(shape, initializer, trainable=False)

def build(self, input_shape):
    # Imposta il seed per il generatore di numeri casuali di TensorFlow
    tf.random.set_seed(self.Seed)
    super().build(input_shape)

    inputDim = input_shape[-1]
    outDim = self.HiddenOutputSize

    # Costruzione dell' input encoder
    input_encoder = self.createWeight((inputDim,
     ↪  self.MemoryDim),self.InputEncoderScaler)


    # Costruzione dei kernel

    # Se la connessione HiddenToMemory è abilitata
```

```
    if self.HiddenToMemory:
        # Il kernel e composto dal encoder per l'input e per l'output del hidden cell
        hidden_encoder = self.createWeight((outDim, self.MemoryDim), self.HiddenEncoderScaler)
        self.MemoryKernel = tf.concat([input_encoder, hidden_encoder], axis=0)
    else:
        # Altrimenti il kernel è composto solo dal encoder per l'input
        self.MemoryKernel = input_encoder

    # Se la connessione MemoryToMemory è abilitato, crea il kernel della memoria composto da dall'encoder per la memoria
    if self.MemoryToMemory:
        self.RecurrentMemoryKernel = self.createWeight((self.Order * self.MemoryDim, self.MemoryDim), self.MemoryEncoderScaler)

    # Se è abilitato l'uso del bias, crea il bias per la memoria
    if self.UseBias:
        self.Bias = self.createWeight((self.MemoryDim,), self.BiasScaler)

    # Se è presente una cella nascosta e non è ancora stata costruita, la costruisce qui
    if self.HiddenCell is not None and not self.HiddenCell.built:
        # Calcola la dimensione dell'input per la cella nascosta
        hidden_input_d = self.MemoryDim * self.Order

        # Se la connessione InputToHiddenCell è abilitata, aggiunge la dimensione dell'input alla dimensione totale
        if self.InputToHiddenCell:
            hidden_input_d += input_shape[-1]

        # Costruisce la cella nascosta con l'input calcolato
        with tf.name_scope(self.HiddenCell.name):
            self.HiddenCell.build((input_shape[0], hidden_input_d))

    # Chiama la funzione per generare le matrici A e B
    self.GenerateABMatrix()

```

Listing 7: Codice python del metodo build del Custom Layer LRMU.

Nel metodo *Call* del Layer LRMU, mostrato nel listato 8, viene implementato l'algoritmo di computazione descritto dalle equazioni (2.7) e visualizzate dallo schema dell'architettura in figura 2.6.

```
def call(self, inputs, states, training=False):
    # Estrae lo stato corrente (memory state e hidden state) dagli stati forniti
    states_fat = tf.nest.flatten(states)
```

```
    memory_state = states_fat[0]
    hidden_state = states_fat[1:]

    # Concatenazione degli input. Se la connessione HiddenToMemory è abilitata, aggiunge lo stato nascosto all'input
    concat_input = inputs
    if self.HiddenToMemory:
        concat_input = tf.concat((concat_input, hidden_state[0]), axis=1)

    # Calcolo del nuovo stato di memoria
    u = tf.matmul(concat_input, self.MemoryKernel)

    # Se la connessione MemoryToMemory è abilitata, aggiunge il contributo dello stato di memoria precedente
    if self.MemoryToMemory:
        u += tf.matmul(memory_state, self.RecurrentMemoryKernel)

    # Se è abilitato l'uso del bias, lo aggiunge al risultato
    if self.UseBias:
        u += self.Bias

    # Espande la dimensione di 'u' per adattarlo alla moltiplicazione con la matrice B
    u = tf.expand_dims(u, -1)

    # Reshape dello stato di memoria in una matrice 3D per la moltiplicazione matriciale
    memory_state = tf.reshape(memory_state, (-1, self.MemoryDim, self.Order))

    # Calcola il nuovo stato di memoria combinando matrici A e B
    new_memory_state = tf.matmul(memory_state, self.A) + tf.matmul(u, self.B)

    # Reshape dello stato di memoria alla forma originale (dimensione piatta)
    new_memory_state = tf.reshape(new_memory_state, (-1, self.MemoryDim * self.Order))

    # Calcolo del nuovo stato nascosto
    # Se la connessione InputToHiddenCell è False, usa solo new_memory_state; altrimenti concatena gli input
    hidden_input = (
        new_memory_state if not self.InputToHiddenCell else tf.concat((new_memory_state, inputs), axis=1)
    )

    # Se HiddenCell non e' impostata, allora trata come se fosse l'identita
    if self.HiddenCell is None:
        output = hidden_input
```

```
        new_hidden_state = []
    # Se la HiddenCell è una cella RNN (controllando se ha 'state_size'), calcola il nuovo stato nascosto
    elif hasattr(self.HiddenCell, "state_size"):
        output, new_hidden_state = self.HiddenCell(hidden_input, hidden_state, training=training)
    else:
        # Se HiddenCell non è una RNN, considera solo l'output senza stato
        output = self.HiddenCell(hidden_input, training=training)
        new_hidden_state = [output]

    # Restituisce l'output e il nuovo stato composto da new_memory_state e il nuovo hidden_state
    return output, [new_memory_state] + new_hidden_state
```

Listing 8: Codice python del metodo call del Custom Layer LRMU.

E in fine nel listato 9 viene mostrato come costruire il modello LRMU utilizzando le api funzionali descritte nella sezione 2.5.2. Il modello viene presentato in modo generale e quindi tutte le variabili a cui non è assegnato un valore specifico sono iperparametri selezionabili.

```

def LRMU_Model():
    # Definizione della cella non lineare
    hiddenCell = SimpleRNN(unit, init)

    # Definizione del modello
    inputs = ks.Input(shape=(sequenceLenght, 1), name=f"LRMU_Input")
    feature = LRMU(memoryDim, order, theta, hiddenCell, hiddenToMemory, memoryToMemory, inputToHiddenCell, useBias, hiddenEncoderScaler, memoryEncoderScaler, inputEncoderScaler, biasScaler, seed)(inputs)
    output = Dense(classNuber, activation, kernel_initializer=GlorotUniform(seed))(feature)
    model = ks.Model(inputs, outputs, name=f"ProblemName_LRMU_Model")
    model.compile(optimizer, loss, metrics)
    return model
```

Listing 9: Codice python per costruire un modello utilizzando il Custom Layer LRMU.

3.2 Legendre Memory Unit con ESN

Il modello ***LMU-ESN*** è il modello originale LMU che utilizza come cella non lineare una *ESN*. In questo modo gli encoder vengono allenati ma il Kernel di

input e quello ricorrente dell'*ESN* restano fissi. Anche in questo caso vanno selezionati i valori di scaling, ma solo per l'*ESN*.

Implementazione

Il modello LMU-ESN è costruito su un layer LMU e su una ESN pre-esistenti, perciò non bisogna definire un nuovo layer. Il codice per costruire il modello è mostrato nel listato 10.

```
1
2  def LMU_ESN_Model():
3      # Definizione dell cella non lineare
4      hiddenCell = ESN(unit, spectralRadius, leaky inputScaler)
5
6      # Definizione del modello
7      inputs = ks.Input(shape=(sequenceLenght, 1), name=f"LMU-ESN_Input")
8      feature = LMU(memoryDim, order, theta, hiddenCell, hiddenToMemory,
       ↪  memoryToMemory, inputToHiddenCell, useBias, seed)(inputs)
9      output = Dense(classNuber, activation,
       ↪  kernel_initializer=GlorotUniform(seed))(feature)
10     model = ks.Model(inputs, outputs, name=f"ProblemName_LMU-ESN_Model")
11     model.compile(optimizer, loss, metrics)
12     return model
```

Listing 10: Codice python per costruire un modello utilizzando il layer LMU e una ESN.

3.3 Legendre Reservoir Memory Unit con ESN

Il modello ***LRMU-ESN*** è il modello LRMU che utilizza una ESN come cella non lineare. In questo modello non si allenano né gli encoder né i kernel della cella non lineare. Quello che viene effettivamente allenato è solo il Readout, che in questo caso è formato da un layer denso di neuroni.

Implementazione

Il modello ***LRMU-ESN*** è costruito utilizzando il layer LRMU descritto nella sezione 3.1 e una *ESN*. Il codice per la costruzione del modello è mostrato nel listato 11.

```python

def LRMU_ESN_Model():
    # Definizione della cella non lineare
    hiddenCell = ESN(unit, spectralRadius, leaky, inputScaler)

    # Definizione del modello
    inputs = ks.Input(shape=(sequenceLenght, 1), name=f"LRMU-ESN_Input")
    feature = LRMU(memoryDim, order, theta, hiddenCell, hiddenToMemory,
    ↪  memoryToMemory, inputToHiddenCell, useBias, hiddenEncoderScaler,
    ↪  memoryEncoderScaler, inputEncoderScaler, biasScaler,
    ↪  seed)(inputs)
    output = Dense(classNuber, activation,
    ↪  kernel_initializer=GlorotUniform(seed))(feature)
    model = ks.Model(inputs, outputs, name=f"ProblemName_LRMU-ESN_Model")
    model.compile(optimizer, loss, metrics)
    return model
```

Listing 11: Codice python per costruire un modello utilizzando il Custom Layer LRMU.

3.4 Legendre Reservoir Memory Unit con ESN e Ridge come Readout

Il modello ***LRMU-ESN-R*** è un modello LRMU che utilizza ESN come cella non lineare e la Ridge regression o la Ridge classifier come readout. Anche in questo modello non vengono allenati né gli encoder né i kernel della cella non lineare e il Readout è l'unica componente che viene effettivamente allenata. Per definizione, il readout, utilizza una mean square come funzione di loss che viene minimizzata utilizzando la formula diretta. Questo modello rispetta pienamente il paradigma del *resorvoir computing* introdotto nella sezione 2.3.

Implementazione

Il modello ***LRMU-ESN-R*** è stato definito tramite subclassing dell'oggetto *Model* di Keras. Questo internamente utilizza un layer LRMU, descritto nella sezione 3.1, una *ESN* come reservoir e, come readout, un Ridge regression o Ridge classifier, definiti nella libreria *sklearn*. Il codice di definizione del modello è mostrato nel listato 12.

```
def __init__(self, ModelType, sequenceLenght, memoryDimension, order,
↪  theta, hiddenToMemory=False, memoryToMemory=False,
↪  inputToHiddenCell=False, useBias=False, hiddenEncoderScaler=None,
↪  memoryEncoderScaler=None, InputEncoderScaler=None, biasScaler=None,
↪  units=1, activation=tf.nn.tanh, spectral_radius=0.99, leaky=1,
↪  input_scaling=1.0, bias_scaling=1.0, readout_regularizer=1.0,
↪  features_dim=1, seed=0, **kwargs):

    super().__init__(**kwargs)
    # Creazione della rete reservoir con parametri personalizzati
    self.reservoir = keras.Sequential([
        keras.Input(shape=(sequenceLenght, 1)),
        LRMU(memoryDimension, order, theta,
             ReservoirCell(units, spectral_radius, leaky, input_scaling,
             ↪  bias_scaling, activation),
             hiddenToMemory, memoryToMemory, inputToHiddenCell, useBias,
             hiddenEncoderScaler, memoryEncoderScaler,
             ↪  InputEncoderScaler, biasScaler, seed)])

    # Definizione del livello di readout in base al tipo di modello
    if ModelType == ModelType.Classification:
        self.readout = RidgeClassifier(alpha=readout_regularizer,
        ↪  solver='svd')
    elif ModelType == ModelType.Prediction:
        self.readout = Ridge(alpha=readout_regularizer, solver='svd')

    self.units = units
    self.features_dim = features_dim



def call(self, inputs):
    # Forward pass attraverso il reservoir
    reservoir_states = self.reservoir(inputs)

    # Predizione tramite il livello di readout
    output = self.readout.predict(reservoir_states.numpy())
    return output

def fit(self, x, y, validation_data, **kwargs):

    x_train_states = self.reservoir(x)
    self.readout.fit(x_train_states, y)
    prediction = self.readout.predict(x_train_states)

    # Calcolo delle metriche sul set di allenamento
    metric_results = {}
    for metric in self.metrics:
        metric_results[metric.name] = metric(y, prediction)

```

```
    # Se fornita la valutazione sul set di validazione
    if validation_data is not None:
        val_x_state = self.reservoir(validation_data[0])
        val_pred = self.readout.predict(val_x_state)
        metric_results[f"val_{metric.name}"] =
        ↪  self.metric(validation_data[1], val_pred)


    # Memorizza i risultati delle metriche
    history = keras.callbacks.History()
    history.history = metric_results
    return history

def evaluate(self, x, y, **kwargs):
    x_states = self.reservoir(x)
    y_prediction = self.readout.predict(x_states)

    # Restituisce le metriche calcolate dove [0] = loss, [1] = altra
    ↪  metrica
    return [self.metrics[0](y, y_prediction),
            self.metrics[1](y, y_prediction)]
```

Listing 12: Codice python per la definizione del modello LRMU-ESN-R tramite subclassing dell'oggetto Model di Keras.

Chapter 4

Sperimentazione e Risultati

In questo capitolo vengono presentati e discussi i risultati ottenuti dagli esperimenti eseguiti sui modelli esposti nel capitolo 3. La sperimentazione è stata fatta su problemi comunemente utilizzati come benchmark nella letteratura scientifica nei casi in cui si vuole valutare che il modello abbia una certa capacità di memoria. Questi problemi sono il psMNIST, una variante del MNIST, e il Mackey-Glass, un problema di predizione di serie temporali. Nella sezione 4.1 vengono introdotti gli obiettivi della sperimentazione e le procedure con cui sono stati selezionati i modelli. Nella sezione 4.2 viene spiegato il dataset psMNIST e vengono esposti i risultati ottenuti dai modelli. Infine, nella sezione 4.3 si espone il dataset Mackey-Glass e i relativi risultati ottenuti.

4.1 Caratterizzazione generale degli esperimenti

In questa sezione si riportano alcune informazione generali da conoscere sulla sperimentazione eseguita, come gli obiettivi e le procedure per la selezione dei modelli mostrati. Queste informazioni sono comuni per tutti i problemi sotto elencati.

4.1.1 Obiettivi delle sperimentazioni

Le sperimentazioni esposte in questo capitolo sono fatte in modo da fornire un confronto equo tra i modelli presentati nel capitolo 3 e il modello con un solo layer *LMU*. Viene mostrato quanto non allenare alcune parti dell'architettura influisca sulle performance del modello e quanto ne modifica i tempi di training. Le performance sono espresse nei termini della metrica selezionata per ogni problema specifico.

4.1.2 Selezioni dei modelli

Partendo da un modello base *LMU*, composto da un layer di input, un layer *LMU* e un layer di output, i modelli utilizzati per la sperimentazione sono stati scelti mantenendo fissi gli iperparametri che controllano la dimensione del modello e ricercando i rimanenti, come mostrato nella sezione 2.5.2, utilizzando una ricerca Bayesiana. In questo modo viene garantita la confrontabilità tra i modelli in quanto per ogni modello viene mantenuto lo stesso numero di parametri totali.

4.2 psMNIST

Il dataset ***Permuted Sequential MNIST*** (***psMNIST***) [6] è una variante del dataset *MNIST*. Il *MNIST* contiene dati per il problema del riconoscimento delle cifre scritte a mano dove ogni dato è un immagine 28×28 pixel e la label associata è la cifra rappresentata in quell'immagine. Applicando una permutazione fissa a ogni immagine del MNIST si ottiene il dataset *pMNIST*, di cui si può visualizzare un elemento nella figura 4.1.

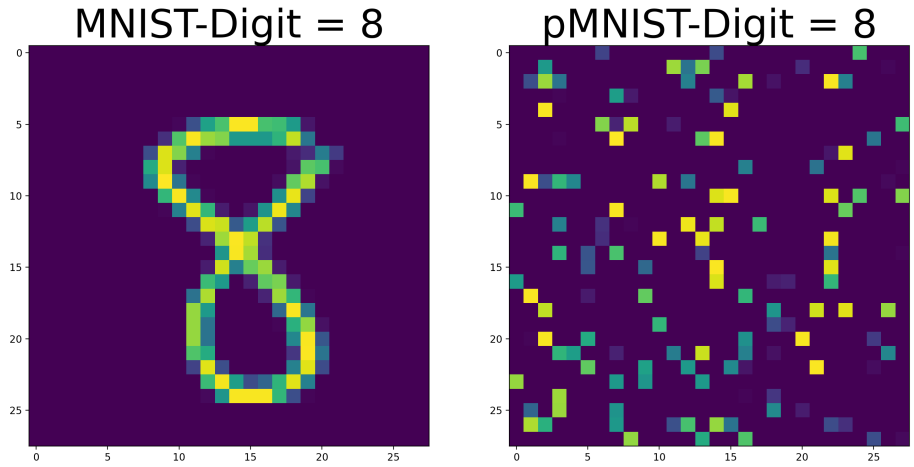

Figure 4.1: Il grafico a sinistra mostra un elemento del MNIST con label 8 mentre il grafico a destra mostra lo stesso elemento trasformato in pMNIST.

Dal *pMNIST* si ottiene il ***psMINST*** trasformando ogni immagine in serie temporale, ovvero sotto forma di lista di pixel ordinata secondo la posizione dei pixel nell'immagine, partendo da in alto a sinistra e procedendo per righe. Un elemento di questo datatset è raffigurato nella figura 4.2.

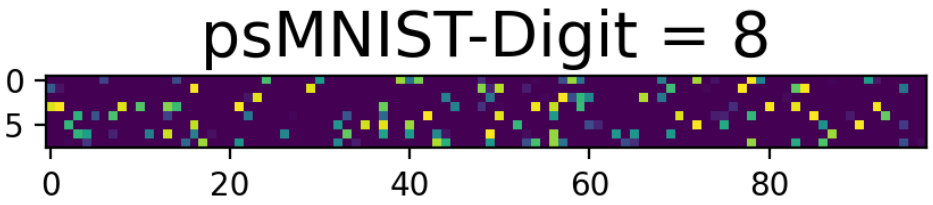

Figure 4.2: Illustrazione dello stesso elemento della figura 4.1, nel dataset ***psMNIST*** visualizzato come un rettangolo 8×98 pixel.

Il ***psMINST*** è adatto per fare benchmark delle reti neurali ricorrenti. Infatti, per una corretta classificazione, la rete deve essere in grado di imparare a costruire una memoria lungo termine di tutta la sequenza dei pixel in input. È importante notare che la difficoltà del problema potrebbe variare in quanto dipende dalla permutazione scelta.

Modelli

Le sperimentazioni su questo problema sono state fatte partendo dal modello proposto nel paper del LMU [15], modello che ha riportato un test accuracy del $97,15\%$. Per ogni modello presentato nel capitolo 3 è stata seguita la procedura di selezione descritta nella sezione 4.1.2. La ricerca è stata eseguita per 50 trial da 5 epoche ciascuno. Gli iperparametri mantenuti fissi sono i seguenti: memoryDimension= 1, order = 256, theta = 784, unit = 212. L'econder dell'input è l'unico attivo, il bias per la memoria non è stato incluso e l'input viene passato anche alla cella non lineare. Gli altri iperparametri sono impostati come mostrato nella tabella 4.1 mentre, nella tabella 4.2, è riportato un confronto tra i modelli in termini di pesi allenabili e non allenabili.

	LRMU	**LMU-ESN**	**LRMU-ESN**	**LRMU-ESN-R**
Raggio spettrale	-	0.99	0.87	0.87
Leaky	-	0.8	0.9	0.9
Input scaling	-	1.75	1	1
Bias scaling	-	1	1	1
Input enc. scaling	1.75	-	2	2
Ridge regularizer	-	-	-	1

Table 4.1: Iperparametri che non controllano la dimensione per i modelli per il problema psMNIST.

Modello	Totale	Allenabili	Non allenabili
LMU	101,771	101,771	0
LRMU	101,771	101,770	1
LMU-ESN	101,771	2,131	99,640
LRMU-ESN	101,771	2,130	99,641
LRMU-ESN-R	101,771	2,130	99,641

Table 4.2: Confronto tra modelli usati per il psMNIST in termini di pesi allenabili e non allenabili.

Risultati

I risultati presentati in questa sezione sono stati ottenuti utilizzando il dataset ***psMINST*** che in totale conta 70.000 elementi partizionati in training, validation e test set, rispettivamente in: 54.000, 6.000 e 10.000 elementi. Ogni modello è stato allenato per 25 epoche o fino a convergenza, con dimensione del batch da 64 elementi e utilizzando adam come ottimizzatore, lasciando i parametri di default di Keras. La funzione di *loss* scelta è la *sparse categorical crossentropy* mentre come metrica è stata scelta l'*accuracy*.
I risultati sono presentati nella tabella 4.3, mostrando per ogni modello l'accuracy, sul validation set e sul test set, e il tempo impiegato per ogni step.

Modello	Validation	Test	Step(ms)
LMU	**96.97**%	**97.15**%	$94_{\pm 2}$
LRMU	96.85%	96.22%	$80_{\pm 1}$
LMU-ESN	91.66%	91.02%	$80_{\pm 1}$
LRMU-ESN	90.36%	90.12%	$33_{\pm 3}$
LRMU-ESN-R	86.61%	86.66%	$--$
FF-baseline	92.37%	92.65%	$2_{\pm 1}$

Table 4.3: Validation accuracy, Test accuracy e tempo per ogni step calcolati su una GPU Tesla V100-PCIE-16GB per il dataset psMNIST.

Discussione

Il modello LRMU mostra una leggera degradazione delle performance rispetto al LMU, con una test accuracy inferiore di circa 1% (96.22% rispetto a 97.15%), ma senza variazioni significative nei tempi di training. Questo era un risultato atteso, poiché l'unico parametro non allenabile del LRMU è l'encoder dell'input, che in questo caso ha un unico parametro.

Il modello LMU-ESN, invece, mostra una test accuracy del 91.02%, un valore estremamente negativo se si considera che i tempi di training sono

simili a quelli del modello LRMU, che ottiene una test accuracy di 96.22%.

Il modello LRMU-ESN è il più veloce in termini di tempo per step, impiegando solo 33 millisecondi. Tuttavia, questa velocità viene al costo di una grande riduzione delle performance, con una test accuracy pari a 90.12%.

Infine, il modello LRMU-ESN-RC non consente di misurare i singoli step, poiché l'intero training avviene in un unico step prolungato. Nonostante ciò, il tempo totale di training è inferiore rispetto a quello degli altri modelli, ma le performance risultano essere le più basse tra i modelli considerati, con una test accuracy di 86.66%.

Dai risultati emerge che, in generale, questo approccio non ha raggiunto gli obiettivi sperati per questo problema. Si osserva una tendenza in cui la riduzione dei parametri allenabili del modello porta un calo delle performance, con una diminuzione dei tempi di training che non è sufficiente a giustificare il trade-off tempo-performance.

4.3 Mackey-Glass

Mackey-Glass è una famiglia di equazioni differenziali con delay non lineare definita come:

$$\frac{df(t)}{dt} = \frac{\beta_0 \theta^n f(t-\tau)}{1 + f(t-\tau)^n} - \gamma f(t) \tag{4.1}$$

dove $\beta_0, \theta, n, , \gamma, \tau$ sono parametri dell'equazione. Questa famiglia è nata in un contesto di studio biologico e, a seconda dei parametri, mima sia dinamiche salutari che patologiche di certi contesti biologici [9]. Nel *Macchine Learning* il problema diventa di predizione dell'evoluzione dell'equazione Mackey-Glass ed è comunemente utilizzato come benchmark, soprattutto nel *Reservoir Computing*, in quanto cambiando il parametro τ, che controlla il delay dell'equazione differenziale, si può controllare la caoticità della soluzione dell'equazione. Questo è un buon benckmarck in quanto per predire correttamente quest'equazione il modello deve avere la capacità di mantenere della memoria su delle serie temporali complesse. Per *caoticità* si intende quanto una variazione, arbitrariamente piccola, nell'input iniziale faccia divergere la serie in direzioni totalmente diverse. I valori di τ comunemente usati sono $\tau = 17$, per una lieve caoticità, e $\tau = 30$, per una grande caoticità, mentre gli altri parametri sono solitamente impostati come segue: w z e come stato iniziale $f(0) = 0.1$. Questi sono gli stessi valori utilizzati nel paper originale [9] dove l'equazione è stata presentata. Plottando i punti $(f(t), f(t-\tau))$ si può visualizzare l'attrattore dell'equazione.

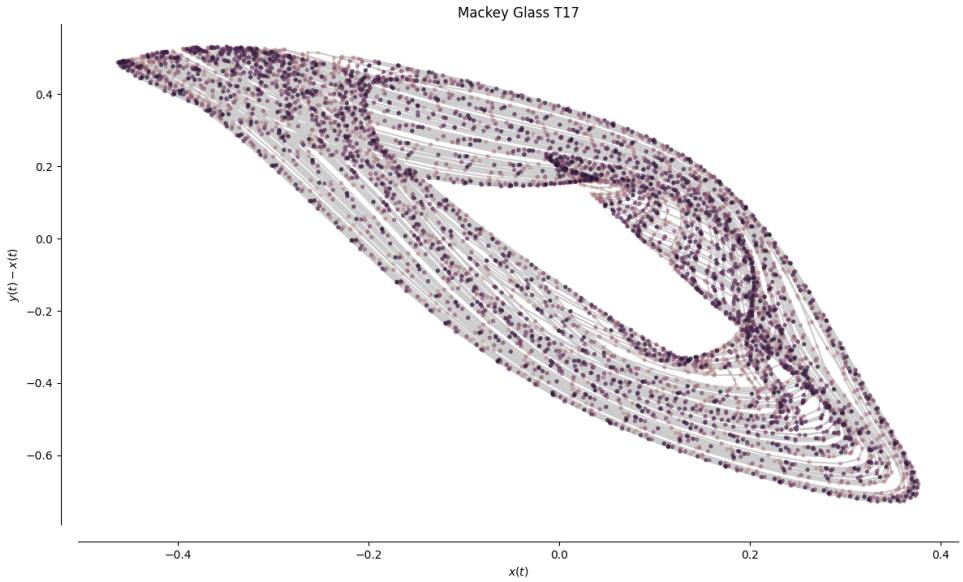

Figure 4.3: Mackey-Glass con $\tau = 17$.

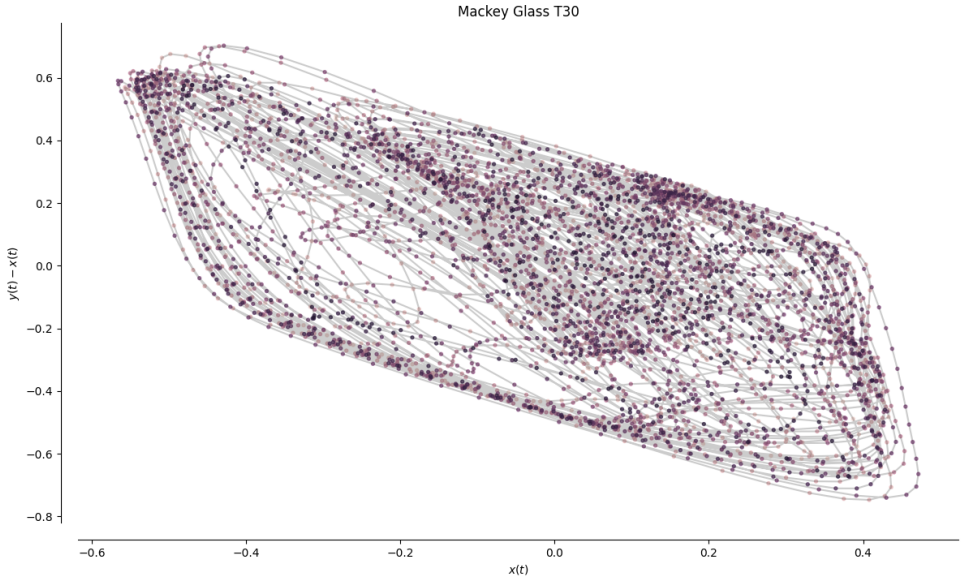

Figure 4.4: Mackey-Glass con $\tau = 30$.

Modelli

Per questo problema non era disponibile un modello LMU, motivo per cui il modello di partenza è stato selezionato tramite ricerca degli iperparametri. La ricerca è stata effettuata sia per la versione con $\tau = 17$ che con $\tau = 30$ ed è stato infine selezionato un unico modello per entrambe le varianti. Gli

altri iperparametri sono stati trovati seguendo la procedura descritta nella sezione 4.1.2. La ricerca è stata eseguita per 50 trial da 50 epoche ciascuno e da questa si evince che sono presenti più configurazioni che convergono alla stessa *validation loss* di 0.046621, nel caso $\tau = 17$, e di 0.0693053, nel caso $\tau = 30$. Il modello LMU scelto come base ha i seguenti iperparametri: memoryDimension = 1, Order = 16, theta = 64, unit= 176. Inoltre sono abilitati gli encoder tra la cella non lineare e la memoria, e tra memoria e la memoria. Anche l'input viene passato alla cella non lineare e il bias per la memoria non è stato incluso. Gli altri iperparametri sono stati ricercati come esposto nella sezione 4.1.2 e sono riportati nella tabella 4.4, mentre nella tabella 4.5 è riportato un confronto tra i modelli in termini di pesi allenabili e non allenabili.

	LRMU	**LMU-ESN**	**LRMU-ESN**	**LRMU-ESN-R**
Raggio spettrale	-	0.87	1.09	1.09
Leaky	-	0.5	0.7	0.7
Input scaling	-	1.75	1.5	1.5
Bias scaling	-	1.75	1.5	1.5
Hidden enc. scaling	0.5	-	0.5	0.5
Memory enc. scaling	0.5	-	0.5	0.5
Input enc. scaling	1.75	-	1.5	1.5
Ridge regularizer	-	-	-	1

Table 4.4: Iperparametri che non controllano la dimensione per i modelli per il problema Mackey-Glass.

Modello	**Totale**	**Allenabili**	**Non allenabili**
LMU	34,514	34,514	0
LRMU	34,514	34,321	193
LMU-ESN	34,514	370	34,144
LRMU-ESN	34,514	177	34,337
LRMU-ESN-R	39,337	5000	34,337

Table 4.5: Confronto tra modelli usati per il Mackey-Glass in termini di pesi allenabili e non allenabili.

Risultati

I risultati presentati in questa sezione sono stati ottenuti utilizzando il dataset ***Mackey-Glass***, il quale è composto da 128 serie da 5000 elementi, le cui label sono anch'esse 128 serie da 5000 elementi, ma spostate nel futuro di 15 step. In questo modo il modello impara a predire i 15 step successivi alla serie di input. Il dataset è diviso in training, validation e test set rispettivamente in:

102, 12 e 14 elementi. Ogni modello è stato allenato per 500 epoche o fino a convergenza, con dimensione del batch da 64 elementi e utilizzando adam come ottimizzatore lasciando i parametri di default di Keras. La funzione di *loss* scelta è la *mean squere error* (*MSE*) mentre come metrica di valutazione è stata scelta la *mean absolute error* (*MAE*) per la sua facile interpretabilità. I risultati per il Mackey-Glass con $\tau = 17$ sono presentati nella tabella 4.6, mentre i risultati per Mackey-Glass con $\tau = 30$ sono mostrati nella tabella 4.7. Per ogni modello vengono mostrati la validation MAE, la test MAE e il tempo per processare un singolo step.

Modello	**Validation**	**Test**	**Step(ms)**
LMU	0.22284	0.19269	$726_{\pm 25}$
LRMU	0.19957	0.18438	$657_{\pm 20}$
LMU-ESN	0.18919	0.19755	$633_{\pm 22}$
LRMU-ESN	0.19161	0.18414	$389_{\pm 21}$
LRMU-ESN-R	**0.16384**	**0.16438**	$--$
FF-BaseLine	0.21037	0.25690	$58_{\pm 3}$

Table 4.6: Validation e test MAE e tempo per ogni step calcolati su una GPU Tesla V100-PCIE-16GB per il problema Mackey-Glass con $\tau = 17$.

Modello	**Validation**	**Test**	**Step(ms)**
LMU	0.28175	0.25518	$726_{\pm 25}$
LRMU	0.24399	0.24229	$657_{\pm 20}$
LMU-ESN	0.25691	0.27440	$633_{\pm 22}$
LRMU-ESN	0.24529	0.22015	$389_{\pm 21}$
LRMU-ESN-R	**0.20486**	**0.20637**	$--$
FF-BaseLine	0.29038	0.28654	$58_{\pm 3}$

Table 4.7: Validation e test MAE e tempo per ogni step calcolati su una GPU Tesla V100-PCIE-16GB per il problema Mackey-Glass con $\tau = 30$.

Discussione

Il modello FF-BaseLine mostra risultati peggiori in entrambi i casi, con un test MAE di 0.25690 per $\tau = 17$ e 0.28654 per $\tau = 30$, dimostrando che sia il modello LMU che i modelli esposti in questa tesi offrono vantaggi rispetto a questa semplice rete feed-forward. Questo è il modello più veloce, con un tempo per step di 58 millisecondi, risultato non sorprendente a causa del funzionamento parallelo delle reti feed-forward, che si contrappongono alle reti ricorrenti, per natura sequenziali. Pertanto, questo dato non deve essere messo a confronto con gli altri ed è riportato per completezza.

Il modello LMU ha bisogno di 726 millisecondi per step, risultando il più lento tra i modelli. Con un test MAE di 0.19269 per $\tau = 17$ e 0.25518 per $\tau = 30$, le sue prestazioni non sono tra le migliori, con risultati inferiori rispetto agli altri modelli eccetto per la baseline.

Il modello LRMU mostra un leggero miglioramento rispetto al LMU per il caso $\tau = 17$, con un test MAE di 0.18438, mentre per $\tau = 30$ si ottiene un test MAE di 0.24229. Il tempo per un singolo step è di 657 millisecondi, leggermente più rapido del LMU.

Il modello LMU-ESN presenta un test MAE di 0.19755 per $\tau = 17$ e 0.27440 per $\tau = 30$, mostrando un peggioramento in entrambi i casi rispetto al LRMU, pur mantenendo un tempo per step molto simile, pari a 633 millisecondi rispetto ai 657 millisecondi del LRMU.

Il modello LRMU-ESN con $\tau = 17$ ha un test MAE di 0.18414, molto vicino al risultato del LRMU, e con $\tau = 30$ ottiene un test MAE di 0.22015, migliorando rispetto agli altri modelli. Il tempo per step è di 389 millisecondi, risultando molto più rapido rispetto agli altri modelli.

Il modello LRMU-ESN-R ottiene i risultati migliori in assoluto, con un test MAE di 0.16438 per $\tau = 17$ e di 0.20637 per $\tau = 30$. Contrariamente a quanto ci si potrebbe aspettare, non è il più veloce in tempo di training totale. Il training viene effettuato in un unico step che, probabilmente, richiede molto tempo per via del fatto che la sequenza per ogni sample è molto lunga.

Dai risultati ottenuti per il problema Mackey-Glass, si evince che la riduzione dei parametri allenabili, oltre a migliorare i tempi di training, migliora anche la performance dei modelli mostrando che, utilizzare una memoria lineare con encoder non allenati e una cella ESN può portare vantaggi nella predizione di serie caotiche. Per via della lentezza in tempo totale di training del modello LRMU-ESN-R, il modello LRMU-ESN risulta essere il miglior compromesso tra tempi di training e performance.

Chapter 5

Conclusioni

In questa tesi sono state presentate e analizzate delle varianti del modello LMU, implementate con l'obiettivo di combinare le performance di questo modello con la velocità di allenamento tipica dei modelli basati sul paradigma del reservoir. L'intento principale è stato ridurre il tempo necessario per l'addestramento, cercando di mantenere, per quanto possibile, performance simili al LMU.
I risultati degli esperimenti hanno dimostrato che il raggiungimento di questo obiettivo non è sempre possibile e dipende dal problema affrontato. Per esempio, nel caso del problema psMNIST, i risultati ottenuti non sono stati soddisfacenti: sebbene sia stato osservato un miglioramento nei tempi di training, la riduzione significativa in termini di accuratezza ha evidenziato che, in questo contesto, il trade-off tra tempo di allenamento e prestazioni non risulta vantaggioso.
Al contrario, nel caso del problema Mackey-Glass, i risultati si sono rivelati migliori. In particolare, oltre alla riduzione dei tempi di training, alcuni dei modelli considerati hanno riportato una MAE inferiore rispetto al modello LMU standard. Ciò suggerisce che, in certi contesti, è possibile ottenere sia un miglioramento della velocità di addestramento che delle buone performance. Tuttavia, va notato che, nell'esperimento condotto, le prestazioni assolute per il problema Mackey-Glass non risultano tra le migliori, poiché con modelli più grandi è possibile ottenere una MAE inferiore. Ciononostante, l'obiettivo della sperimentazione è stato raggiunto, ovvero mantenere buone prestazioni riducendo il numero di parametri da allenare, e quindi il tempo di training, rispetto al modello LMU originale.
Guardando al futuro, ci sono diverse possibilità per espandere e migliorare questo lavoro. Un primo passo potrebbe essere quello di testare i modelli presentati su altri dataset, sia di benchmark che provenienti da applicazioni reali. Questo permetterebbe di verificare la generalizzabilità dei risultati e comprendere meglio in quali contesti queste varianti del modello LMU possano offrire vantaggi concreti.

Un'altra direzione per futuri sviluppi potrebbe consistere nell'implementazione di altre architetture, come la concatenazione di più modelli LRMU-ESN, formando una struttura deep in grado di catturare più efficacemente le dipendenze a lungo termine. Infine, si potrebbe esplorare la possibilità di modificare ulteriormente la struttura del modello, mantenendo l'idea della separazione tra la cella lineare di memoria e la cella non lineare per il calcolo delle relazioni complesse, ma sperimentando nuove configurazioni dei pesi.
In conclusione, i risultati ottenuti in questa tesi mostrano che esistono margini per ulteriori sperimentazioni e perfezionamenti. Il bilanciamento tra accuratezza e velocità di training resta una sfida aperta, ma le prospettive di miglioramento sono tangibili.

Bibliography

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[2] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[3] Niklas Donges. *A Complete Guide to Recurrent Neural Networks (RNNs)*. 2024. URL: https://builtin.com/data-science/recurrent-neural-networks-and-lstm.

[4] Claudio Gallicchio. *Echo State Network Slide*. 2019.

[5] Herbert Jaeger. "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note". In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148.34 (2001), p. 13.

[6] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[7] Kasper Groes Albin Ludvigsen. *the carbon footprint of gpt 4*. 2023. URL: https://towardsdatascience.com/the-carbon-footprint-of-gpt-4-d6c676eb21ae.

[8] Mantas Lukoševičius and Herbert Jaeger. "Reservoir computing approaches to recurrent neural network training". In: *Computer science review* 3.3 (2009), pp. 127–149.

[9] Michael C Mackey and Leon Glass. "Oscillation and chaos in physiological control systems". In: *Science* 197.4300 (1977), pp. 287–289.

[10] Tom M Mitchell and Tom M Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[11] *Nerual network definition*. 2024. URL: https://www.oxfordreference.com/display/10.1093/oi/authority.20110803100229917.

[12] Tom O'Malley et al. *KerasTuner*. 2019. URL: https://github.com/keras-team/keras-tuner.

[13] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. URL: http://aima.cs.berkeley.edu/.

[14] Arthur L Samuel. “Some studies in machine learning using the game of checkers”. In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[15] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. “Legendre memory units: Continuous-time representation in recurrent neural networks”. In: *Advances in neural information processing systems* 32 (2019).