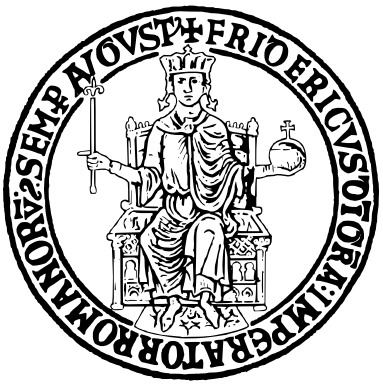
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e
Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Neural Networks and Deep Learning

Progettazione e implementazione di una libreria per il supporto alla costruzione e all'uso di reti neurali dense

Docente
Prof. Roberto Prevete

Candidati
Gennaro Sorrentino N97/347
Gianluca L'Arco N97/393

Anno Accademico 2021 - 2022

Indice

Elenco delle Figure

Elenco delle Tabelle

Capitolo 1

Introduzione

Una rete neurale artificiale (ANN, "Artificial Neural Network"), normalmente indicata solo rete neurale (NN, "Neural Network"), è un modello matematico-informatico di calcolo basato sulle reti neurali biologiche, costituito da un gruppo di interconnessioni di unità elementari dette neuroni artificiali.
La nascita delle reti neurali è datata nei primi anni del '40 con l'idea di neurone artificiale (percettrone) proposta da *W.S. McCulloch* e *Walter Pitts* nell'articolo "*A Logical Calculus of the Ideas Immanent in Nervous Activity*". Nonostante l'euforia iniziale, nel 1969 i ricercatori *Marvin Minsky* e *Seymour Papert*, mostrarono i limiti operativi delle semplici reti a due strati basate sul percettrone, dimostrandone l'impossibilità di risolvere molte classi di problemi, ossia tutti quelli non caratterizzati da separabilità lineare delle soluzioni. Il loro rinnovato interesse e impiego iniziò grazie alla proposta, nel 1986 da parte di *David E. Rumelhart*, dell'algoritmo di retropropagazione dell'errore. Ad oggi le reti neurali sono sempre più utilizzate, per gli scopi e le esigenze più disparati.
In questo lavoro ci si pone come obiettivo lo sviluppo di una libreria che consenta la costruzione e l'impiego di reti neurali dense. Nello specifico si fa riferimento ai seguenti punti:

- Parte A:
  - Progettazione ed implementazione di funzioni per simulare la propagazione in avanti di una rete neurale multi-strato. Dare la possibilità di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione per ciascun strato.

  - Progettazione ed implementazione di funzioni per la realizzazione della back-propagation per reti neurali multi-strato, per qualunque scelta della funzione di attivazione dei nodi della rete e la possibilità di usare almeno la somma dei quadrati o la cross-entropy con e senza soft-max come funzione di errore.

- Parte B:

  - Classificazione delle immagini del dataset MNIST[1], con l'ausilio dell'algoritmo di resilient back-propagation (RPROP) per l'aggiornamento dei pesi. Identificazione di una rete neurale densa a singolo strato interno (shallow) sub-ottimale mediante un approccio di k-fold cross validation per la valutazione degli iper-parametri della rete quali parametri RPROP (eta-positivo ed eta-negativo) e numero di neuroni interni.

Struttura della documentazione

Nel capitolo 2, *Cenni Teorici*, verranno esposti brevi cenni teorici sui principali argomenti affrontati nel corso della progettazione e relativa implementazione.
Nel capitolo 3, *Parte A*, verranno descritte le funzionalità implementate e le soluzioni adottate a tale scopo.
Nel capitolo 4, *Parte B*, verrà descritta la funzionalità aggiuntiva RPROP per l'aggiornamento dei pesi, oltre alla discussione di un caso d'uso di model selection di una rete neurale densa a singolo strato interno per la classificazione delle immagini del dataset MNIST.
Nel capitolo 5, *Conclusioni e Sviluppi Futuri*, verranno esposte le conclusioni e le possibili migliorie applicabili alla libreria implementata.

[1]La base di dati MNIST è una vasta base di dati di cifre scritte a mano che è comunemente impiegata come insieme di addestramento in vari sistemi per l'elaborazione delle immagini: http://yann.lecun.com/exdb/mnist/

Capitolo 2

Cenni Teorici

In questo capitolo, vengono riassunti alcuni degli aspetti teorici che ricoprono gli argomenti principali affrontati nella progettazione e implementazione della libreria.

2.1 Reti neurali dense

L'unità elementare di una rete neurale artificiale (ANN, "Artificial Neural Network"), è il percettrone 2.1. Un tale modello è costituito da:

- d connessioni in ingresso a cui ad ognuna è associata una variabile di ingresso $x_i \in \mathbb{R}$ e un peso $w_i \in \mathbb{R}, \forall i = 1, ..., d$;
- una funzione di attivazione definita come $f : a \in \mathbb{R} \rightarrow f(a) \in \mathbb{R}$, ossia un mapping funzionale da $\mathbb{R}$ ad $\mathbb{R}$;
- una variabile di uscita $y \in \mathbb{R}$ (in generale);
- un bias $b \in \mathbb{R}$, il quale può essere definito come il peso w_0 associato alla connessione in ingresso con costante di ingresso $x_0 = 1$ (notazione omogenea).

La computazione di un neurone segue due fasi distinte:

1. Calcolo dell'input del neurone:

$$a = \sum_{j=1}^{d} w_j \cdot x_j + b \tag{2.1}$$

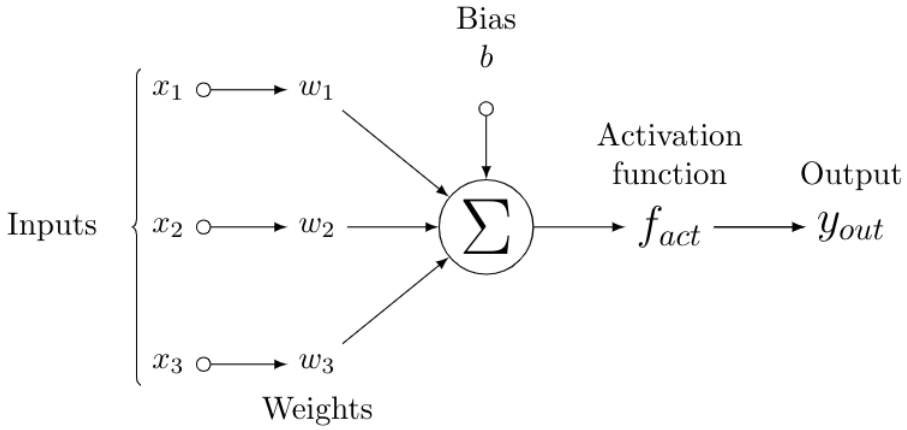


Figura 2.1: Struttura generale di un percettrone

2. Calcolo dell'output del neurone:

$$y = f(a) \tag{2.2}$$

L'interazione tra neuroni varia a seconda dell'architettura presa in considerazione. Nel nostro caso consideriamo reti con architettura feed-forward, ossia reti acicliche, in cui quindi esiste un ordine topologico stabilito dalle connessioni che in generale è parziale.

Un particolare tipo di rete feed-forward sono le reti multistrato, dove i neuroni sono organizzati in strati (o layer), i quali presentano un ordine totale tra loro. In una siffatta rete neurale, i neuroni di un certo strato h possono ricevere connessioni esclusivamente dai neuroni dello strato precedente $h-1$.

Una rete neurale densa altro non è che una rete multistrato dove ogni nodo i di uno strato h riceve connessioni da tutti i nodi dello strato precedente $h-1$.

La nomenclatura generalmente utilizzata è la seguente:

- a_i^h rappresenta l'input dell'i-esimo neurone dello strato h;
- z_i^h rappresenta l'output dell'i-esimo neurone dello strato h;
- f^h rappresenta la funzione di attivazione relativi a tutti i nodi dello strato h (ciò rappresenta una semplificazione in quanto ogni neurone può disporre di una differente funzione di attivazione);

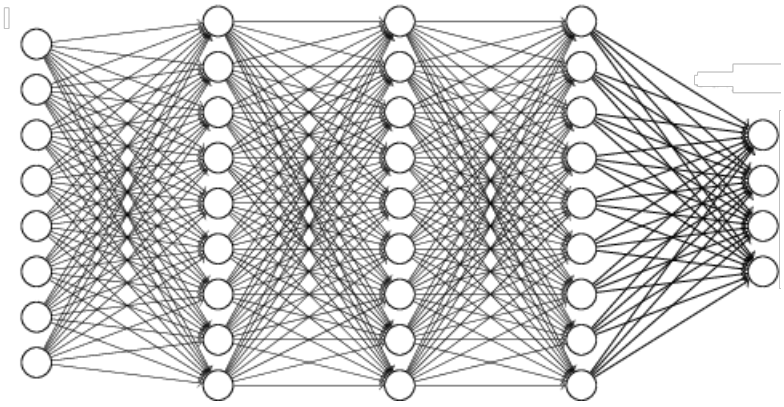


Figura 2.2: Rete neurale densa

- w_{ij}^h rappresenta il peso della connessione che arriva al nodo i-esimo dello strato h a partire dal nodo j dello strato $h-1$;

- b_i^h rappresenta il bias dell'i-esimo neurone dello strato h.

da ciò ne deriva che il comportamento della rete può essere descritto come segue:

$$\forall h \in [1, H], \forall i \in [1, m_h] : \qquad a_i^h = \sum_{j=1}^{m_h - 1} w_{ij}^h \cdot z_j^{h-1} + b_i^h, \quad z_i^h = f_h(a_i^h) \tag{2.3}$$

dove H rappresenta il numero di strati, m_h il numero di neuroni nello strato h e $h = 0$ allora $z_i^h = x_i$.

2.2 Processo di learning

Il processo di learning ha come obiettivo quello di individuare i parametri, pesi e bias, sub-ottimali della rete (ossia che approssimano al meglio il fenomeno), supponendo fissati gli iperparametri della stessa (e.g. numero di nodi, funzioni di attivazione, numero di layer). Ci poniamo in un contesto di apprendimento supervisionato, ovvero guidato da dati etichettati. In tal caso, suddividiamo i problemi in due classi:

- **Classificazione**: problemi le cui etichette sono valori discreti non ordinati che possono essere considerati appartenenti a un gruppo di una classe;

- **Regressione**: problemi in cui si dispone di un numero di variabile predittive (descrittive) e una variabile target continua (output). In questo tipo di problema si cerca di trovare una relazione tra queste variabili al fine di prevedere un risultato.

Il processo di learning si suddivide in due fasi:

1. Generazione di un dataset, ossia un insieme di coppie $DS = \{(\underline{x}^n, \underline{t}^n)\}$ dove $\underline{x}^n$ è un vettore che rappresenta la singola istanza dell'aggetto, e $\underline{t}^n$ indica la classe dell'istanza $\forall n \in [N]$;

2. definizione di un modello di rete il cui compito sarà quello di trovare i parametri (pesi e bias) adatti per un problema di apprendimento dato.

2.2.1 Funzione di errore

La funzione di errore viene introdotta per valutare le prestazioni del modello ottenuto. Esso confronta sostanzialmente i valori previsti con i valori effettivi.
La scelta di una determinata funzione di errore dipende dal tipo di problema che si vuole risolvere, ad esempio:

- **Cross Entropy**: utilizzata per problemi di classificazione;

$$E^{(n)} = -\sum_{k=1}^{c} t_k^n \ln y_k^n \tag{2.4}$$

- **Sum of Squares**: utilizzata per problemi di regressione.

$$E^{(n)} = -\frac{1}{2}\sum_{k=1}^{c}(y_k^n - t_k^n)^2 \tag{2.5}$$

2.2.2 Retro-propagazione dell'errore

Il calcolo del gradiente della funzione di errore E, ∇E, avviene mediante l'algoritmo di retro-propagazione dell'errore. Le derivate parziali ottenute possono poi essere impiegate in tecniche quali la discesa del gradiente 2.2.3.1.
L'algoritmo consta dei seguenti passi:

1. Calcolo degli input a_i^n e output z_i^n di tutti i nodi (i scorre su tutti i nodi);

2. Calcolo dei δ_i^n:

   - Per i nodi interni:

$$\delta_i^n = \sum_k w_{ki} \cdot \delta_k^n \cdot f'(a_i^n) \tag{2.6}$$

   - Per i nodi di output:

$$\delta_k^n = \frac{\partial E^{(n)}}{\partial y_k^n} \cdot g'(a_k^n) \tag{2.7}$$

3. Calcolo della derivata parziale:

$$\frac{\partial E^{(n)}}{\partial w_{ij}} = \delta_i^n \cdot z_j^n \tag{2.8}$$

Con la 2.8 si intende il prodotto locale tra due nodi (i e j) dove z_j^n è l'output del nodo j e δ_i^n è il delta del nodo successivo i.

2.2.3 Modalità e regole di aggiornamento

La modalità di aggiornamento definisce in quale momento e per quanto tempo applicare la regola di aggiornamento.

- **Online learning**: per ogni coppia $(\underline{x}^n, \underline{t}^n)$ del dataset di training si calcolano le derivate di $E^{(n)}$ rispetto a ogni singolo parametro e infine si provvede al loro aggiornamento;

- **Batch learning**: calcolo della derivata di E rispetto a ogni singolo parametro e conseguente aggiornamento;

- **Mini-Batch learning**: suddivisione del dataset in un determinato numero di partizioni su cui si esegue l'aggiornamento in modalità batch.

2.2.3.1 Discesa del gradiente

La discesa del gradiente è una regola di aggiornamento che tramite il gradiente della funzione di errore e il learning rate (tasso di apprendimento) cerca il minimo della funzione di errore. In particolare, una volta ottenuto il gradiente, si aggiornano i pesi mediante la seguente regola di aggiornamento:

$$w_{ij} = w_{ij} - \eta \cdot \frac{\partial E}{\partial w_{ij}} \tag{2.9}$$

L'errore E indicato nella regola di aggiornamento dipende dalla modalità di learning scelta (si parla difatti di stochastic gradient descent, batch gradient descent e mini-batch gradient descent).

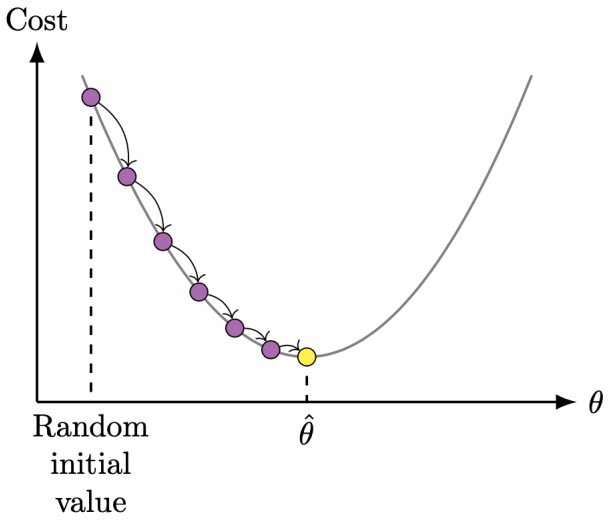


Figura 2.3: Discesa del gradiente

2.2.3.2 RPROP

L'RPROP è una regola di aggiornamento introdotta per rendere la classica discesa del gradiente indipendente dagli iperparametri, ossia η. L'idea alla base della regola è la seguente:

1. Associare a ciascun peso w_{ij} uno step di aggiornamento Δ_{ij};

2. Modificare opportunamente gli step di aggiornamento Δ_{ij} durante la fase di learning.

In questo modo la regola di aggiornamento sarà la seguente:

$$w_{ij}^{(t)} = w_{ij}^{(t-1)} - sign\left(\frac{dE^{(t)}}{dw_{ij}}\right) \cdot \Delta_{ij} \tag{2.10}$$

In questo modo, se la derivata risulta essere positiva, $\frac{dE^{(t)}}{dw_{ij}} > 0$, si decrementa il peso di una quantità Δ_{ij} , mentre se risulta essere negativa, $\frac{dE^{(t)}}{dw_{ij}} < 0$, lo si incrementa di una quantità Δ_{ij}. Tale aggiornamento avviene dunque considerando le derivate $\frac{dE^{(t)}}{dw_{ij}}$ e $\frac{dE^{(t-1)}}{dw_{ij}}$. Difatti, immaginando di voler aggiornare $w_{ij}^{(t)}$ si tiene conto di $w_{i,j}^{(t-1)}$. Dunque guardando le due derivate delle due epoche è possibile capire intuitivamente il salto effettuato. In generale vale che:

- se $\frac{dE^{(t)}}{dw_{ij}} \cdot \frac{dE^{(t-1)}}{dw_{ij}} < 0$ il salto effettuato è troppo grande, è quindi necessario ridurre Δ_{ij};
- se $\frac{dE^{(t)}}{dw_{ij}} \cdot \frac{dE^{(t-1)}}{dw_{ij}} > 0$ il salto effettuato è "corretto", è quindi necessario incrementare Δ_{ij}.

Dunque si utilizzano dei nuovi iperparametri che permettono di incrementare e decrementare i Δ_{ij}:

- η^+ per l'incremento di Δ_{ij}, con $\eta^+ > 1$;
- η^- per il decremento di Δ_{ij}, con $0 < \eta^- < 1$.

Ogni step di aggiornamento Δ_{ij} del singolo peso w_{ij} viene aggiornato con la seguente regola:

$$\Delta_{ij} = \begin{cases} min(\eta^+ \cdot \Delta_{ij}, \Delta_{max}) & \text{se } \frac{dE^{(t)}}{dw_{ij}} \cdot \frac{dE^{(t-1)}}{dw_{ij}} > 0 \\ max(\eta^- \cdot \Delta_{ij}, \Delta_{min}) & \text{se } \frac{dE^{(t)}}{dw_{ij}} \cdot \frac{dE^{(t-1)}}{dw_{ij}} < 0 \\ \Delta_{ij} & \text{altrimenti} \end{cases} \tag{2.11}$$

dove Δ_{max} e Δ_{min} sono ipeparametri aggiuntivi. Dunque in definitiva:

$$w_{ij}^{(t)} = w_{ij}^{(t-1)} + \Delta w_{ij}^{(t)} \qquad \Delta w_{ij}^{(t)} = -sign\left(\frac{dE^{(t-1)}}{dw_{ij}}\right) \cdot \Delta_{ij} \tag{2.12}$$

2.3 Model selection

Le problematiche che si affrontano nella definizione di un sistema di machine learning si dividono nella sua costruzione, valutazione e infine selezione. Gli ultimi due sono processi che servono per la scelta degli iperparametri, ovvero per selezionare un modello di rete neurale, i quali saranno successivamente utilizzati per costruire l'effettivo sistema di machine learning.

2.3.1 Valutazione

Il processo di valutazione racchiude l'insieme di tecniche che consentono di valutare i diversi modelli di rete ottenuti.

2.3.1.1 Hold-out

Questo metodo consiste nella suddivisione del dataset in training set e test set. Il training set viene utilizzato per l'addestramento del modello M, ottenendo come risultato il modello e i parametri associati. Il test set viene poi utilizzato per valutare il modello ottenuto, $M(\theta^*)$. Le metriche che posso essere utilizzate sono diverse, tipicamente si adotta una funzione di errore. È bene notare che sia la suddivisione del dataset che la numerosità dei dati possono influenzare particolarmente l'esito della valutazione, in quanto incidono sulla rappresentatività degli stessi.

2.3.1.2 K-fold cross validation

Questo metodo consiste nell'applicazione dell'approccio hold-out k volte. Fissato un dataset X questo viene suddiviso in k partizioni disgiunte X_i. Ad ogni iterazione k si esclude la partizione X_k e si addestra il modello sull'insieme delle restanti partizioni $X' = \bigcup_{i \neq k} X_i$, infine si utilizza la partizione esclusa X_k come test set. La misura che esprime la valutazione complessiva del modello la si ottiene tramite la media sperimentale e la deviazione standard della metrica di riferimento:

$$m_v = \frac{\sum_{i=1}^{k} v_i}{k}, \quad dev = \sqrt{\frac{1}{k-1} \sum_{i=1}^{k} (v_i - m_v)^2} \tag{2.13}$$

In assenza di un gran numero di dati è possibile effettuare una valutazione mediante il Leave-One-Out (LOO), in cui ad ogni iterazione si esclude una singola coppia dal dataset, per un totale di $k = |X|$ iterazioni. Il caso generale del LOO è il Leave-P-Out in cui ad ogni iterazione si escludono P coppie dal dataset.

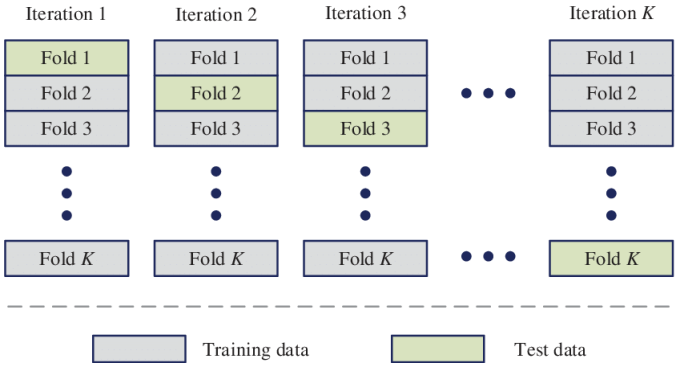


Figura 2.4: Iterazioni del k-fold cross-validation

2.3.2 Selezione

Il processo di selezione è un insieme di tecniche che attraverso le medie sperimentali e le deviazioni standard ottenute nel processo di valutazione dei diversi modelli, ne seleziona il migliore, ossia la configurazione sub-ottimale degli iperparametri. Dunque date le configurazioni di iperparametri $\{HP_1, .., HP_S\}$, i punti su cui bisogna ragionare sono:

1. Quali iperparametri considerare (e.g. numero di nodi, funzione di attivazione, numero di strati, etc.);

2. Quale intervallo di valori utilizzare.

Essendo il primo una scelta dipendente dal problema, poniamo l'attenzione sul secondo aspetto. Per definire l'intervallo di valori degli iperparametri si seguono tipicamente due tecniche:

- **Griglia**: Si fissano gli estremi di un intervallo e un passo di avanzamento;
- **Casuale**: Si fissano gli estremi di un intervallo e si selezionano casualmente un dato numero di valori.

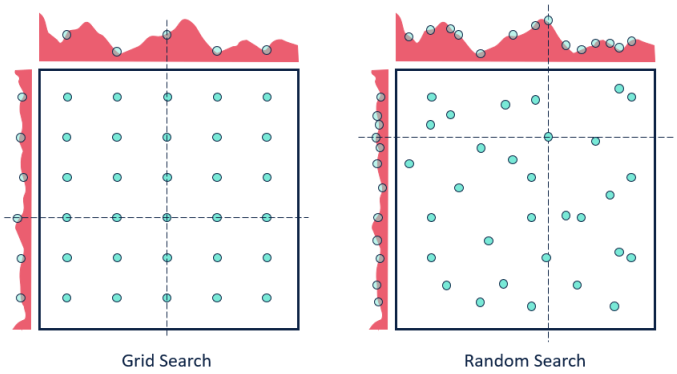


Figura 2.5: (Da sinistra) Strategia di selezione del modello a griglia e casuale

È importante notare che l'approccio a griglia aumenta l'intervallo di valori al crescere del numero di iperparametri da considerare (crescita esponenziale), mentre l'approccio casuale sceglie sempre un numero fissato e casuale-uniforme di valori per costruire l'intervallo. Per tale motivo, l'approccio casuale risulta essere meno esaustivo ma computazionalmente migliore.

2.3.3 Early Stop

L'early stop è un approccio che consiste nel definire una condizione di stop dato un numero abbastanza alto di epoche (e.g. minimizzazione della funzione di costo fino a un certo valore prefissato c). Questo consente di avere un buon compromesso tra complessità computazionale e generalizzazione. Una condizione di stop molto utilizzata è detta pazienza, che consiste nel terminare l'apprendimento se l'errore non diminuisce per un dato numero di epoche.

Algorithm 1: Generico algoritmo di apprendimento

Data: M
Result: θ^*
while *condition* **do**
    learning;
    $epoch \leftarrow epoch + 1$;
end

Capitolo 3

Parte A

In questo capitolo, vengono discussi gli aspetti relativi alla progettazione e implementazione della libreria. Nello specifico viene presentata la struttura del progetto e descritte le principali funzionalità dello stesso con annessi approfondimenti per i principali algoritmi utilizzati.

3.1 Struttura della libreria

La struttura della libreria presenta tre differenti pacchetti:

- **core**: è il nucleo della libreria e contiene il codice per la costruzione e l'impiego dei layer e della rete neurale. Il pacchetto consta di due moduli:
  - *models*: contiene la classe MLP che rappresenta una generica rete neurale con architettura feed-forward;
  - *layers*: contiene la classe astratta Layer e tutte le sue specializzazioni (e.g. Dense, ResilientDense).
- **utils**: contiene le funzioni di attivazione e di errore, le metriche di valutazione, collezioni aggiuntive e decoratori. Il pacchetto consta di quattro moduli:
  - *functions*: contiene le funzioni di attivazione e di errore;
  - *metrics*: contiene le metriche di valutazione e la classe EarlyStop la quale rappresenta una condizione per l'early-stop;
  - *collections*: contiene strutture dati aggiuntive (e.g. Dataset);

  - *decorators*: contiene decoratori utili al controllo di pre-condizioni obbligatorie all'invocazione di determinati metodi.

- **data**: contiene il codice per effettuare il caricamento di un qualunque dataset (in questa versione è implementato il codice per il caricamento del dataset MNIST). Il pacchetto consta del solo modulo *loader*.

Di seguito è riportato il package diagram in UML 3.1:

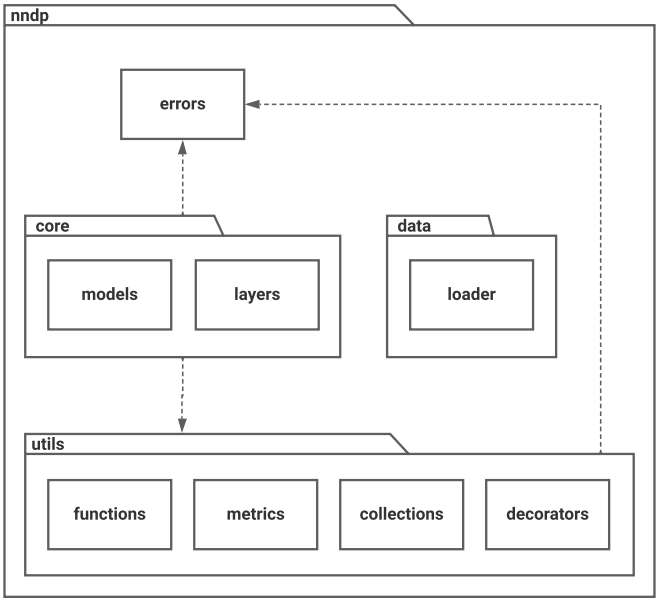


Figura 3.1: Struttura della libreria

3.2 Descrizione delle funzionalità

3.2.1 Dataset

La classe Dataset esprime un insieme di coppie $(\underline{x}, \underline{t})$ dove $\underline{x}$ è un vettore che rappresenta un generico elemento e $\underline{t}$ l'etichetta associata alla classe di appartenenza. L'obiettivo di tale classe è quello di semplificare le operazioni di vista sul dataset quali ad esempio la suddivisione e il mescolamento. Nello specifico gli attributi della classe sono:

- **data**: vettore che contiene gli elementi $\underline{x}$.

- **labels**: vettore che contiene le etichette $\underline{t}$ di ogni elemento $\underline{x}$.

Mentre i metodi che offre sono i seguenti:

```
def split(self, portion: float = 0.25) -> tuple[Dataset, Dataset]
```

Descrizione: Effettua la suddivisione del dataset in due porzioni (il dataset originale non viene mutato).

Input:

- *portion*: rappresenta la dimensione del dataset di destra, quando minore di 1 è intesa come percentuale, altrimenti come dimensione assoluta;

Output: Coppia di dataset, dove quello di destra ha come dimensione la porzione richiesta e quello di sinistra la restante.

```
def random(self, instances: int = 10000) -> Dataset
```

Descrizione: Restituisce un dataset estratto casualmente da quello originale.
Input:

- *instances*: dimensione del dataset da estrarre.

Ouput: Un nuovo dataset estratto casualmente.

```
def k_fold(
    self, n_splits: int = 2, shuffle: bool = False
) -> list[tuple[Dataset, Dataset]]
```

Descrizione: Effettua la suddivisone del dataset in k partizioni e restituisce k coppie di dataset, dove, detta p_i l'i-esima partizione con $i \in [k]$, la generica coppia c_i è $(\bigcup_{k \neq i} p_k, p_i)$ (il dataset originale non viene mutato).
Input:

- *n_splits*: numero di partizioni in cui suddividere il dataset.

- *shuffle*: booleano che indica se effettuare un mescolamento del dataset originale.

Output: Lista di k coppie di dataset $c_i = (\bigcup_{k \neq i} p_k, p_i)$.

3.2.2 Loader

La classe astratta Loader offre un'interfaccia per il caricamento di dati in un dataset 3.2.1 (solitamente da file ma è anche possibile implementare un Loader che recuperi i dati dalla rete) . Nello specifico gli attributi della classe sono:

- **dataset**: istanza della classe Dataset contenente i dati caricati.

Mentre i metodi di cui si richiede l'implementazione sono i seguenti:

```
# Abstract
@staticmethod
def encode(labels: np.ndarray) -> np.ndarray
    raise NotImplementedError
```

Descrizione: Effettua la codifica delle etichette in un formato variabile a seconda delle esigenze.
Input:

- *labels*: insieme delle etichette da codificare.

Output: insieme delle etichette codificate.

```
# Abstract
@staticmethod
def decode(labels: np.ndarray) -> np.ndarray
    raise NotImplementedError
```

Descrizione: Effettua la decodifica delle etichette al formato originale.
Input:

- *labels*: insieme delle etichette da decodificare.

Output: insieme delle etichette decodificate.

```
# Abstract
def scaled_dataset(self) -> Dataset
    raise NotImplementedError
```

Descrizione: Effettua una normalizzazione degli elementi $\underline{x}$ contenuti nel dataset.
Output: Un nuovo dataset normalizzato.

3.2.2.1 MNIST

La classe MNIST, la quale estende la classe Loader 3.2.2, rappresenta il dataset MNIST che è una vasta base di dati di cifre scritte a mano che è comunemente impiegata come set di addestramento in vari sistemi per l'elaborazione delle immagini. I metodi implementati rispettano l'interfaccia definita dalla classe Loader.

3.2.3 EarlyStop

La classe EarlyStop rappresenta una condizione di early-stop per la fase di apprendimento di una rete neurale. Nello specifico gli attributi della classe sono:

- **metric**: criterio di valutazione da utilizzare.
- **trigger**: valore che soddisfa la condizione di early-stop;
- **greedy**: booleano che indica se interpretare il valore di trigger come un valore target (*greedy* = false), oppure, come una percentuale di tolleranza della discrepanza tra il valore migliore raggiunto dalla metrica di riferimento durante l'apprendimento e il valore attuale (*greedy* = true).

Metric: Nel dettaglio, l'attributo *metric*, rappresenta un valore dell'enumerazione *Metric*, la quale offre i seguenti criteri di valutazione:

- **Accuratezza**: percentuale di previsioni corrette sul totale delle previsioni effettuate;

- **Precisione**: percentuale delle previsioni corrette rispetto a tutte le previsioni etichettate come corrette;
- **Completezza**: percentuale delle previsioni corrette rispetto a tutte le previsioni che sarebbero dovute essere etichettate corrette;
- **F1-score**: media armonica tra precisione e completezza;
- **Errore**.

Mentre i metodi che offre sono i seguenti:

```
def is_satisfied(self, value: float) -> bool:
```

Descrizione: Verifica se il valore di riferimento per la condizione di early-stop è stato raggiunto. Si tenga presente che tale metodo per tutti i criteri di valutazione, ad eccezione dell'errore, verifica la massimizzazione.
Input:

- **value**: valore attualmente raggiunto dalla metrica di riferimento.

Output: Booleano che indica se la condizione di early-stop è verificata.

3.2.4 Layer

La classe astratta Layer rappresenta un generico layer di una rete con architettura feed-forward. Non esiste una particolare preferenza nelle connessioni tra i neuroni di due differenti strati, dunque tale aspetto va definito in tutte le classi che la estendono. Data la sua natura, gli attributi presenti in questa classe sono tutti quelli in comune tra generici strati di una rete feed-forward, ossia:

- **width**: numero di neuroni del layer;
- **activation**: funzione di attivazione in comune tra tutti i neuroni del layer;
- **name**: nome attribuito al layer;
- **in_size**: numero di connessioni in ingresso ai neuroni;

- **in_data**: input dei neuroni;
- **in_weighted**: input pesato dei neuroni;
- **out_data**: output dei neuroni;
- **weights**: pesi delle connessioni dei neuroni;
- **biases**: bias dei neuroni;
- **n_trainable**: numero complessivo di parametri addestrabili.

Activation: Nel dettaglio, l'attributo *activation* rappresenta un valore dell'enumerazione *Activation*, la quale offre le seguenti funzioni di attivazione (con annesse derivate):

- **Identità**: $i(x) = x$;
- **Sigmoide** $s(x) = \frac{1}{1+e^{-x}}$;
- **ReLU**: $f(x) = x^+ = max(0, x)$;
- **Tangente iperbolica**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}$.

Anche i metodi, così come gli attributi, sono tutti quelli in comune (fatta qualche eccezione di costruzione) tra generici strati di una rete feed-forward, ossia:

```
def is_built(self) -> bool:
    ...
```

Descrizione: Verifica se il metodo build è stato invocato e dunque se il layer dispone delle informazioni necessarie al corretto funzionamento. Nello specifico con costruzione di un layer si intende la definizione del numero di connessioni in ingresso *in_size* e di conseguenza la costruzione dei parametri del layer *weights* e *biases*.
Ouput: Booleano che indica se il layer è costruito, ovvero se il metodo *build* è stato invocato.

```
# Abstract
@require_not_built
def build(
    self,
    in_size: int,
    weights: Optional[np.ndarray] = None,
    biases: Optional[np.ndarray] = None
) -> None:
     raise NotImplementedError
```

Descrizione: Inizializza i parametri addestrabili *weights* e *biases* del layer.

Input:

- *in_size*: numero di connessioni in ingresso ai neuroni;
- *weights*: eventuali pesi prestabiliti, in assenza la generazione dovrebbe avvenire mediante distribuzione normale in $(0, 10^{-1})$;
- *biases*: eventuali bias prestabiliti, in assenza la generazione dovrebbe avvenire mediante distribuzione normale in $(0, 10^{-1})$.

Nota: Il decoratore *@require_not_built* verifica che tale metodo non sia già stato invocato (e quindi che il layer non sia già stato costruito), in caso contrario solleva l'eccezione *AlreadyBuiltError*.

```
# Abstract
@require_built
def forward_propagation(self, in_data: np.ndarray) -> np.ndarray:
     raise NotImplementedError
```

Descrizione: Effettua la computazione del dato in ingresso salvandone lo stato (dato in ingresso, dato in ingresso pesato, dato in uscita).

Input:

- *in_data*: dato da elaborare.

Output: Risultato della computazione.

Nota: Il decoratore *@require_built* verifica che tale metodo sia invocato su un layer costruito, in caso contrario solleva l'eccezione *NotBuiltError*.

```
# Abstract
@require_built
def backward_propagation(self, expected: np.ndarray) -> np.ndarray:
    raise NotImplementedError
```

Descrizione: Effettua il calcolo delle derivate parziali di ogni parametro rispetto alla funzione di errore adottata nella rete di appartenenza.

Input:

- *expected*: Valore dell'uscita atteso dalla computazione.

Output: Delta dei nodi del layer.

```
# Abstract
@require_built
def predict(self, in_data: np.ndarray) -> np.ndarray:
    raise NotImplementedError
```

Descrizione: Effettua la medesima operazione della feed-forward senza salvarne lo stato (dato in ingresso, dato in ingresso pesato, dato in uscita). Tale metodo dovrebbe essere utilizzato in contesti che non prevedono l'apprendimento (e.g. la rete è stata già precedentemente addestrata).

Input:

- *in_data*: dato da elaborare.

Output: Risultato della computazione.

```
# Abstract
@require_built
def update(self, **kwargs) -> None:
    raise NotImplementedError
```

Descrizione: Effettua l'aggiornamento dei parametri del layer *weights* e *biases*.

Input: Dipende dalla particolare implementazione.

Il costruttore della classe Layer, il quale dovrebbe essere richiamato esclusivamente dalle sue sottoclassi, è il seguente:

```
def __init__(
    self,
    width: int,
    activation: Activation = Activation.IDENTITY,
    name: Optional[str] = None,
)
```

3.2.4.1 Dense

La classe Dense, la quale estende la classe Layer 3.2.4, rappresenta un layer di una rete neurale con architettura feed-forward in cui ogni neurone riceve una connessione da tutti i neuroni del layer precedente.
I metodi implementati rispettano l'interfaccia definita dalla classe Layer. Precisiamo esclusivamente i parametri in input del metodo *update* dal momento che dipendono dalla particolare implementazione:

- **eta**: tasso di apprendimento, ossia la grandezza dei salti da effettuare verso il minimo globale della funzione di errore.

3.2.5 MLP

La classe MLP rappresenta una generica rete neurale con architettura feed-forward non necessariamente multistrato ne tanto meno full-connected, in quanto quest'ultima proprietà dipende direttamente dal layer. Nello specifico gli attributi della classe sono:

- **layers**: lista contenente i layer (interni e l'unico di output) della rete. L'ordine della lista corrisponde all'ordine totale;

- **loss**: funzione di errore da applicare nella fase di apprendimento;
- **name**: nome attribuito alla rete;
- **depth**: profondità della rete, ossia, la dimensione di *layers*;
- **size**: numero di neuroni complessivi della rete;
- **width**: larghezza della rete, ossia, il massimo numero di neuroni nei layer;
- **in_size**: numero di connessioni in ingresso alla rete;
- **in_data**: input della rete;
- **out_size**: dimensione del vettore di output;
- **out_data**: output della rete;
- **n_trainable**: numero complessivo di parametri addestrabili.

Loss: Nel dettaglio, l'attributo *loss* rappresenta un valore dell'enumerazione *Loss*, la quale offre le seguenti funzioni di errore (con annesse derivate):

- **Sum of Squares (SSE)**: $E(\theta) = \sum_{n=1}^{N} \frac{1}{2} \sum_{k=1}^{c} (y_k(\underline{x}^n; \theta) - t_k^n)^2$;
- **Cross Entropy** $E(\theta) = -\sum_{k=1}^{c} t_k^n \ln y_k(\underline{x}^n; \theta)$;
- **Softmax Cross Entropy**: $E(\theta) = -\sum_{k=1}^{c} t_k^n \ln \left(\frac{e^{y_k(\underline{x}^n;\theta)}}{\sum_{h=1}^{c} e^{y_h(\underline{x}^n;\theta)}} \right)$.

Mentre i metodi che offre sono i seguenti:

```
def is_built(self) -> bool:
    ...
```

Descrizione: Verifica se il metodo build è stato invocato e dunque se la rete dispone delle informazioni necessarie al corretto funzionamento. Nello specifico con costruzione di una rete si intende la definizione del numero di connessioni in ingresso *in_size* e la costruzione dei singoli layer.

Output: Booleano che indica se la rete è costruita, ovvero se il metodo *build* è stato invocato.

```
@require_not_built
def build(
    self,
    in_size: int,
    weights: Optional[list[np.ndarray]] = None,
    biases: Optional[list[np.ndarray]] = None
) -> None:
```

Descrizione: Per ogni layer della rete invoca il metodo *build* 3.2.4.

Input:

- *in_size*: numero di connessioni in ingresso alla rete;
- *weights*: lista degli eventuali pesi prestabiliti per ogni layer;
- *biases*: lista degli eventuali bias prestabiliti per ogni layer.

Nota: Quando questo metodo viene invocato è necessario che tutti i layer non siano già stati costruiti, in caso contrario solleva l'eccezione *AlreadyBuiltError*.

```
@require_not_built
def push(self, layer: Layer) -> None:
```

Descrizione: Aggiunge il layer fornito in ingresso in coda alla lista dei layer della rete, di conseguenza tale layer coincide con il layer di output (fino a un nuovo inserimento).

Input:

- *layer*: layer da aggiungere alla rete.

```
@require_not_built
def pop(self):
```

Descrizione: Rimuove l'ultimo layer aggiunto alla rete.

```
@require_built
def predict(self, in_data: np.ndarray) -> np.ndarray:
```

Descrizione: Invoca il metodo *predict* di ogni layer 3.2.4, propagando in avanti la computazione di ogni singola operazione.

Input:

- *in_data*: dato da elaborare.

Output: Risultato della computazione.

```
@require_built
def validate(
    self,
    validation_set: Dataset,
    metrics: list[Metric] = (Metric.LOSS,)
) -> dict:
```

Descrizione: Effettua la valutazione della rete secondo le metriche stabilite.

Input:

- *validation_set*: dataset su cui effettuare la valutazione;
- *metrics*: lista delle metriche da valutare.

Output: valori ottenuti dalle metriche indicate.

```
@require_built
def cross_validate(
    self,
    dataset: Dataset,
    n_splits: int = 5,
    metrics: list[Metric] = (Metric.LOSS,),
    epochs: int = 30,
    n_batches: int = 1
```

```
    **kwargs
) -> dict:
```

Descrizione: Effettua la valutazione con l'approccio di k-fold cross validation.

Input:

- *dataset*: dataset su cui effettuare la valutazione;
- *n_splits*: numero di suddivisioni da effettuare sul dataset;
- *metrics*: lista delle metriche da valutare;
- *epochs*: numero di epoche di addestramento per ogni suddivisione del dataset;
- *n_batches*: indica il numero di partizioni in cui suddividere il dataset di training durante la fase di apprendimento;
- ***kwargs*: parametri da utilizzare nell'invocazione del metodo *update*.

Output: media sperimentale e deviazione standard di ogni metrica indicata.

Nota: l'addestramento ai fini della valutazione non viene effettuato sulla rete originale bensì su delle copie.

```
@require_built
def fit(
    self,
    training_set: Dataset,
    validation_set: Optional[Dataset] = None,
    n_batches: int = 1,
    epochs: int = 500,
    early_stops: Optional[list[EarlyStop]] = None,
    weak_stop: bool = True,
    stats: Optional[list[Metric]] = (Metric.LOSS,),
    **kwargs
) -> list:
```

Descrizione: Effettua l'addestramento della rete, il quale può avvenire in tre differenti modalità: online, full-batch e mini-batch, a seconda del valore del parametro

n_batches.

Input:

- *training_set*: dataset su cui effettuare l'addestramento;
- *validation_set*: dataset su cui effettuare la validazione;
- *n_batches*: indica il numero di partizioni in cui suddividere il *training_set*. Tale valore determina la modalità di learning, infatti, quando è uguale a 0 allora verrà adottata la modalità online, quando è uguale 1 full-batch e infine in tutti gli altri casi mini-batch;
- *epochs*: numero di epoche di addestramento;
- *early_stops*: condizioni di terminazione anticipata dell'apprendimento (early-stop);
- *weak_stop*: booleano che indica se è necessario che tutte le condizioni di early stop debbono verificarsi. Quando è vero allora l'addestramento viene terminato al verificarsi di anche una sola delle condizioni indicate, al contrario, è necessario che tutte le condizioni siano verificate;
- *stats*: lista delle metriche da valutare a scopo statistico;
- ***kwargs*: parametri da utilizzare nell'invocazione del metodo *update*.

Output: statistiche ottenute durante l'apprendimento.

```
def save(self, path: str = None):
```

Descrizione: Effettua la serializzazione di una rete MLP in un file binario.

Input:

- *path*: percorso in cui memorizzare la rete, in assenza la rete viene memorizzata nella directory *saved* del pacchetto *data*.

```
@staticmethod
def load(path: str):
```

Descrizione: Effettua la de-serializzazione di un file binario in una rete MLP.

Input:

- *path*: indica il percorso del file binario.

Output: rete MLP deserializzata.

Infine, il costruttore della classe MLP è il seguente:

```
def __init__(
    self,
    layers: list[Layer] = None,
    loss: Loss = Loss.SSE,
    name: Optional[str] = None
):
```

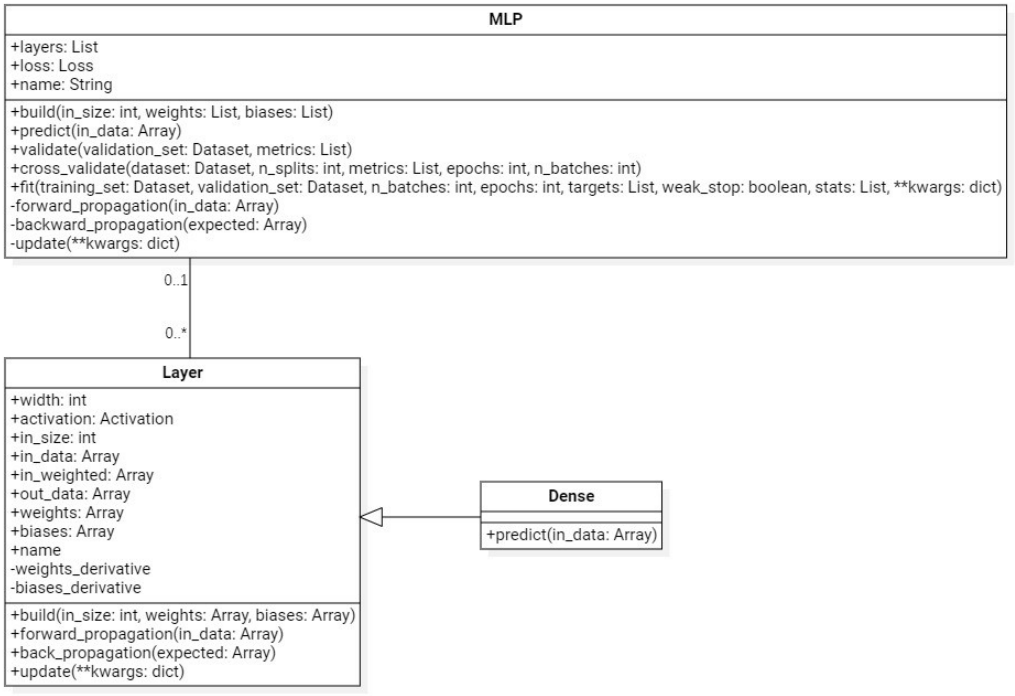


Figura 3.2: Class diagram di una rete neurale

3.3 Approfondimento degli algoritmi

3.3.1 Feed-forward Propagation

La fase di propagazione in avanti è implementata nella classe MLP dal seguente metodo:

```
@require_built
def _forward_propagation(self, in_data: np.ndarray) -> None:
    out_data = in_data
    for layer in self._layers:
        out_data = layer.forward_propagation(out_data)
```

Si osservi che tale metodo è "protetto" (presenta il carattere "_" come prefisso) dal momento che dovrebbe essere utilizzato esclusivamente all'interno della classe MLP, utilizzando in sua sostituzione il metodo *predict*.
Banalmente, il metodo mostrato scorre su tutti gli strati della rete, seguendo il loro ordine totale definito dall'ordinamento stesso della lista, e ne invoca il metodo *forward_propagation*. Consideriamo dunque la propagazione in avanti di un layer Dense:

```
@require_built
def forward_propagation(self, in_data: np.ndarray) -> np.ndarray:
    self._in_data = in_data
    self._in_weighted = self._weights @ in_data + self._biases
    self._out_data = self._activation.function()(self._in_weighted)
    return self._out_data
```

Tale metodo calcola l'input pesato mediante la computazione di $a = W \cdot \underline{x}^T + \underline{b}$ e successivamente restituisce il valore di uscita applicando la funzione di attivazione del layer $z = f(a)$. Si osservi inoltre come il metodo salvi in apposite variabili lo stato dell'intera computazione, ossia, dato in ingresso, dato pesato e dato in uscita, al fine di utilizzarle per l'algoritmo di propagazione all'indietro dell'errore.

3.3.2 Backward Propagation

La fase di propagazione all'indietro è implementata nella classe MLP dal seguente metodo:

```
@require_built
def _backward_propagation(self, expected: np.ndarray) -> None:
    delta = self._loss.prime()(self.out_data, expected)
    for layer in reversed(self._layers):
        delta = layer.backward_propagation(delta)
```

Tale metodo risulta protetto in quanto dovrebbe essere invocato esclusivamente nella fase di apprendimento, ossia dal metodo *fit*.
Inizialmente il metodo effettua il calcolo della derivata della funzione di errore rispetto all'uscita della rete. Il valore calcolato viene poi propagato all'indietro scorrendo sugli strati della rete a partire da quello di output e per ciascuno di essi invoca il loro rispettivo metodo di propagazione dell'errore, ossia:

```
@require_built
def backward_propagation(self, delta: np.ndarray) -> np.ndarray:
    delta = self._activation.prime()(self._in_weighted) * delta
    self._weights_derivative += (delta @ self._in_data.T)
    self._biases_derivative += delta
    return self._weights.T @ delta
```

Osserviamo che inizialmente il metodo effettua il calcolo del delta di ogni neurone del layer per poi calcolare la derivata parziale dell'errore rispetto ai parametri dello strato moltiplicando i delta ottenuti per l'output del layer precedente, il quale coincide con l'input del layer. Infine restituisce il prodotto tra i pesi e i delta locali necessari per il calcolo dei delta dei nodi del layer precedente.
Data l'implementazione non è necessario distinguere tra strato di output e strato nascosto in quanto tale informazione è intrinseca nel parametro delta (il quale in realtà corrisponde a un calcolo parziale del delta effettivo) ricevuto in ingresso, infatti, alla prima iterazione della propagazione dell'errore della rete MLP, solo il layer

di output riceve un "delta" calcolato diversamente, che, tuttavia, non modifica la computazione generale.

3.3.3 Update

La fase di aggiornamento dei parametri della rete avviene mediante il seguente metodo della classe MLP:

```
@require_built
def _update(self, **kwargs) -> None:
    for layer in self._layers:
        layer.update(**kwargs)
```

Il metodo risulta protetto in quanto dovrebbe essere invocato esclusivamente nella fase di apprendimento, ovvero nel metodo *fit*.
Anche in questo caso il metodo scorre su tutti gli strati della rete e ne invoca il metodo *update*. Consideriamo dunque il metodo di aggiornamento per un layer Dense:

```
@require_built
def update(self, **kwargs) -> None:
    eta = kwargs.get("eta", 0.001)
    if not 0 < eta <= 1:
        raise ValueError("eta must be in (0, 1].")
    self._weights -= eta * self._weights_derivative
    self._biases -= eta * self._biases_derivative
    self._weights_derivative = np.zeros(self._weights.shape)
    self._biases_derivative = np.zeros(self._biases.shape)
```

Osserviamo come il metodo effettui l'aggiornamento dei pesi e dei bias del layer facendo uso delle derivate precedentemente calcolate nella *backward_propagation*. Nel dettaglio la regola di aggiornamento impiegata è:

$$\theta_i = \theta_i - \eta \cdot \frac{\partial E}{\partial \theta_i} \tag{3.1}$$

Si tenga infine presente che il metodo *update* dovrebbe essere invocato esclusivamente dopo aver precedentemente effettuato una propagazione all'indietro dell'errore, in caso contrario le derivate potrebbero essere nulle.
Il metodo di aggiornamento è ovviamente differente per un layer ResilientDense, tuttavia tale aspetto verrà trattato nel capitolo Parte B.

3.3.4 Fit

La fase di addestramento della rete avviene mediante il seguente metodo della classe MLP (si tenga presente che, da data la dimensione del metodo, sono stati riportati solo i frammenti di codice principali):

```
@require_built
def fit(
    self,
    training_set: Dataset,
    validation_set: Optional[Dataset] = None,
    n_batches: int = 1,
    epochs: int = 500,
    early_stops: Optional[list[EarlyStop]] = None,
    weak_stop: bool = True,
    stats: Optional[list[Metric]] = (Metric.LOSS,),
    **kwargs
) -> dict:
    # Controllo input ...
    batches = [
        Dataset(data, labels)
        for data, labels in
        zip(
            np.array_split(
                training_set.data, n_batches, axis=1
            ),
            np.array_split(
                training_set.labels, n_batches, axis=1
            )
        )
```

```
    ] if n_batches not in [0, 1] else [training_set]

    for epoch in range(epochs):

        for batch in batches:
            for instance in range(batch.size):
                data = batch.data[:, instance].reshape(-1, 1)
                label = batch.labels[:, instance].reshape(-1, 1)
                self._forward_propagation(data)
                self._backward_propagation(label)
                if n_batches == 0 or instance == batch.size - 1:
                    self._update(**kwargs)

        training_predictions = self.predict(training_set.data)
        validation_predictions = (
            self.predict(validation_set.data)
            if validation_set is not None else None
        )

        early_stop_satisfied = []
        for early_stop in early_stops:
            if early_stop.metric != Metric.LOSS:
                metric_function = early_stop.metric.score()
            else:
                metric_function = self._loss.function()
            current_value = metric_function(
                validation_predictions,
                validation_set.labels
            )
            early_stop_satisfied.append(
                early_stop.is_satisfied(current_value)
            )
        # Calcolo statistiche ...
        if (
            len(early_stop_satisfied) != 0 and (
                weak_stop and any(early_stop_satisfied) or
                not weak_stop and all(early_stop_satisfied)
```

```
            )
        ):
            break
    # Finalizzazione ...
```

A seconda della modalità di aggiornamento scelta, il metodo suddivide il *training_set* fornito in ingresso. In particolare la suddivisione avviene esclusivamente nel caso in cui la modalità di aggiornamento adottata è quella mini-batch, ossia *n_batches* assume un valore diverso da 0 e 1, poiché, nel caso in cui *n_batches* fosse 0, la modalità adottata sarebbe quella online, altrimenti full-batch 2.2.3.

```
batches = [
    Dataset(data, labels)
    for data, labels in
    zip(
        np.array_split(training_set.data, n_batches, axis=1),
        np.array_split(training_set.labels, n_batches, axis=1)
    )
] if n_batches not in [0, 1] else [training_set]
```

A questo punto inizia la fase di apprendimento: per ogni epoca il metodo scorre sulle partizioni di training set precedentemente suddivise e, per ogni istanza, effettua l'operazione di *forward_propagation* 3.3.1 e *backward_propagation* 3.3.2 infine, a seconda della modalità di aggiornamento adottata, effettua l'aggiornamento dei pesi mediante il metodo *update* 3.3.3.

```
for epoch in range(epochs):
    for batch in batches:
        for instance in range(batch.size):
            data = batch.data[:, instance].reshape(-1, 1)
            label = batch.labels[:, instance].reshape(-1, 1)
            self._forward_propagation(data)
            self._backward_propagation(label)
            if n_batches == 0 or instance == batch.size - 1:
                self._update(**kwargs)
```

Terminata l'iterazione delle partizioni del training set, il metodo verifica le condizioni di early-stop, purché sia stato fornito un dataset di validazione in ingresso. A questo punto, a seconda del valore assegnato al parametro *weak_stop* si distinguono due casi:

- *weak_stop* = *false*: in tal caso è richiesto che tutte le condizioni di early-stop siano verificate, ossia che per ogni condizione il metodo *is_satisfied* 3.2.3 restituisca *true* (condizione forte);

- *weak_stop* = *true*: in tal caso è richiesto che almeno una condizione di early-stop sia stata raggiunto (condizione debole).

Il verificarsi dell'early-stop termina immediatamente il metodo non proseguendo ulteriormente l'addestramento:

```
early_stop_satisfied = []
for early_stop in early_stops:
    if early_stop.metric != Metric.LOSS:
        metric_function = early_stop.metric.score()
    else:
        metric_function = self._loss.function()
    current_value = metric_function(
        validation_predictions,
        validation_set.labels
    )
    early_stop_satisfied.append(
        early_stop.is_satisfied(current_value)
    )
# ...
if (
    len(early_stop_satisfied) != 0 and (
        weak_stop and any(early_stop_satisfied) or
        not weak_stop and all(early_stop_satisfied)
    )
):
```

3.3.5 Cross-Validate

La fase di cross-validation della rete avviene mediante il seguente metodo della classe MLP:

```
@require_built
def cross_validate(
    self,
    dataset: Dataset,
    n_splits: int = 5,
    metrics: list[Metric] = (Metric.LOSS,),
    epochs: int = 30,
    n_batches: int = 1,
    **kwargs
) -> dict:
    # Controllo input...
    scores = []
    k_fold = dataset.k_fold(n_splits)
    for k, (training_set, validation_set) in enumerate(k_fold):
        model = deepcopy(self)
        model.fit(
            training_set,
            n_batches=n_batches,
            epochs=epochs,
            stats=None,
            **kwargs
        )
        scores.append(model.validate(validation_set, metrics))
    scores = {
        metric: [score.get(metric) for score in scores]
        for metric in set().union(*scores)
    }
    result = {}
    for key in scores.keys():
        values = np.array(scores[key])
        result[key] = (values.mean(), values.std())
    return result
```

Al fine di effettuare la valutazione del modello, il metodo effettua inizialmente una suddivisione del dataset con approccio *k-fold* 2.3.1.2. Ottenute le k coppie di *training set* e *validation set*, per ognuna di essa effettua l'addestramento di una copia della rete originale, con lo scopo di non utilizzare una rete già precedentemente addestrata andando quindi a invalidare la valutazione stessa. Terminato l'addestramento viene effettuata la validazione della rete mediante il metodo *validate* 3.2.5 con le metriche indicate. Infine, quando tutte le coppie sono state processate, viene calcolata la media sperimentale e la deviazione standard di ogni metrica.

3.4 Esempio d'uso

Di seguito viene mostrato un esempio d'uso che racchiudi la creazione di una rete MLP con conseguente addestramento, plotting delle statistiche di apprendimento e validazione:

```
#!/usr/bin/env python3
import logging
import matplotlib.pyplot as plt
from nndp.data.loader import MNIST
from nndp.utils.metrics import Metric, EarlyStop
from nndp.utils.functions import Activation, Loss
from layers import Dense
from nndp.core.models import MLP


if __name__ == "__main__":

    # Abilitazione logging
    logging.basicConfig(
        format='\n\n\%(message)s\n', level=logging.DEBUG
    )

    # Creazione del caricatore dataset MNIST
    loader = MNIST()
```

```
# Suddivisione dataset in training set,
# validation test e test set
dataset = loader.scaled_dataset.random(instances=10000)
dataset, validation_test = dataset.split(0.25)
training_set, test_set = dataset.split(0.25)

# Creazione di una rete MLP deep con 2 strati interni Densi
#   - Funzione di errore: SOFTMAX CROSS ENTROPY
#   - Funzione di attivazione strati interni: SIGMOIDE
#   - Funzione di output: IDENTITA'
mlp = MLP(
    [
        Dense(10, Activation.SIGMOID),
        Dense(10, Activation.SIGMOID),
        Dense(10, Activation.IDENTITY),
    ],
    Loss.SOFTMAX_CROSS_ENTROPY
)

# Costruzione della rete
# (784 e' la dimensione di un'istanza del dataset MNIST)
mlp.build(784)

print(mlp)
input("\n > Press enter to start... ")

# Addestramento della rete con 300 epoche
# Condizioni di early-stop (debole):
#   - Loss: discrepanza massima del 25 percento
#           dal valore migliore (errore minore raggiunto)
#   - Accuracy >= 0.9
#   - F1-Score >= 0.9
# Statistiche desiderate [LOSS, ACCURACY, F1]
stats = mlp.fit(
    training_set,
    validation_test,
```

```
        epochs=300,
        early_stops=[
            EarlyStop(Metric.LOSS, 25, greedy=True),
            EarlyStop(Metric.ACCURACY, 0.9),
            EarlyStop(Metric.F1, 0.9)
        ],
        weak_stop=True,
        stats=[Metric.LOSS, Metric.ACCURACY, Metric.F1]
    )
    # Esempio output (supponendo early-stop):
    # {
    #   "epochs": 257
    #   "training": {
    #       "loss": [...],
    #       "accuracy": [...],
    #       "f1": [...]
    #    }
    #   "validation": {
    #       "loss": [...],
    #       "accuracy": [...],
    #       "f1": [...]
    #    }
    # }

    # Plotting statistiche
    figure, axis = plt.subplots(3)

    # Plot loss
    training_loss = [
        stats["training"]["loss"][epoch]
        for epoch in range(stats["epochs"])
    ]
    validation_loss = [
        stats["validation"]["loss"][epoch]
        for epoch in range(stats["epochs"])
    ]
    axis[0].set_title("Loss")
```

```
    axis[0].plot(
        range(stats["epochs"]),
        training_loss,
        label="Training"
    )
    axis[0].plot(
        range(stats["epochs"]),
        validation_loss,
        label="Validation"
    )

    # Plot accuracy
    training_accuracy = [
        stats["training"]["accuracy"][epoch]
        for epoch in range(stats["epochs"])
    ]
    validation_accuracy = [
        stats["validation"]["accuracy"][epoch]
        for epoch in range(stats["epochs"])
    ]
    axis[1].set_title("Accuracy")
    axis[1].plot(
        range(stats["epochs"]),
        training_accuracy,
        label="Training"
    )
    axis[1].plot(
        range(stats["epochs"]),
        validation_accuracy,
        label="Validation"
    )

    # Plot F1-score
    training_f1 = [
        stats["training"]["f1"][epoch]
        for epoch in range(stats["epochs"])
    ]
```

```
    validation_f1 = [
        stats["validation"]["f1"][epoch]
        for epoch in range(stats["epochs"])
    ]
    axis[2].set_title("F1-Score")
    axis[2].plot(
        range(stats["epochs"]),
        training_f1,
        label="Training"
    )
    axis[2].plot(
        range(stats["epochs"]),
        validation_f1,
        label="Validation"
    )

    plt.legend()
    plt.show()

    # Validazione
    print(mlp.validate(
        test_set,
        metrics=[Metric.LOSS, Metric.ACCURACY, Metric.F1]
    ))
    # Esempio output:
    # {
    #   "loss": 210.4290285619329,
    #   "accuracy": 0.8733333333333333,
    #   "f1": 0.866651212663067
    # }

    # Salvataggio della rete
    mlp.save()

    # Esempio di logging

    # Epoch 285 of 500 - [full-batch]
```

```
#
#    - Training loss: 976.860
#    - Training accuracy: 0.794
#    - Training f1: 0.780
#
#    - Validation loss: 494.210
#    - Validation accuracy: 0.712
#    - Validation f1: 0.694
#
#    - Early-Stop loss: 25.00 - greedy
#    - Early-Stop accuracy: 0.9
#    - Early-Stop f1: 0.9
```

Capitolo 4

Parte B

In questo capitolo, vengono discusse ulteriori funzionalità della libreria e gli esperimenti condotti con essa. Nello specifico viene presentata una specializzazione della classe Dense 3.2.4.1, ResilientDense, la quale introduce la regola di aggiornamento RPROP 2.2.3.2. Viene inoltre discussa la selezione di un modello di rete neurale a singolo strato, sulla base degli iper-parametri dell'RPROP e del numero di nodi interni, con un approccio di validazione k-fold.

4.1 Funzionalità aggiuntive

4.1.1 ResilientDense

La classe ResilientDense, la quale estende la classe Dense 3.2.4.1, rappresenta un layer di una rete neurale con architettura feed-forward in cui ogni neurone riceve una connessione da tutti i neuroni del layer precedente.
Il comportamento di un layer ResilientDense è il medesimo di un layer Dense, fatta eccezione per l'aggiornamento dei parametri, ossia il metodo *update*, che è dunque ridefinito mediante un override al fine di implementare la regola di aggiornamento RPROP.
I parametri che il metodo riceve in ingresso differiscono da quelli del metodo *update* di Dense dal momento che dipendono dalla particolare implementazione (regola di aggiornamento) e nello specifico sono:

- *eta_positive*: tasso di incremento per Δ_{ij};

- *eta_negative*: tasso di decremento per Δ_{ij};
- *delta_max*: limite superiore di Δ_{ij};
- *delta_min*: limite inferiore di Δ_{ij};
- *delta_zero*: valore iniziale di Δ_{ij}.

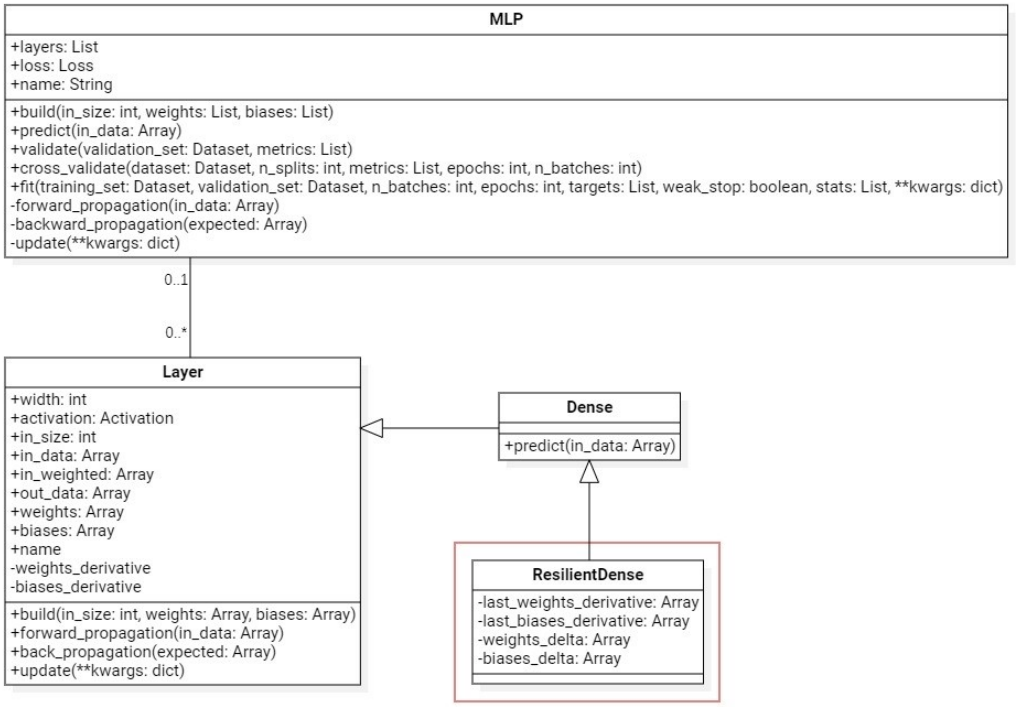


Figura 4.1: Class diagram di una rete neurale (completo)

4.1.1.1 Update

Il metodo *update* di un layer ResilientDense è il seguente:

```
@require_built
def update(self, **kwargs) -> None:

    eta_positive = kwargs.get("eta_positive", 1.2)
    eta_negative = kwargs.get("eta_negative", 0.5)
    delta_max = kwargs.get("delta_max", 50)
    delta_min = kwargs.get("delta_min", 1e-6)
    delta_zero = kwargs.get("delta_zero", 0.0125)
```

```
if not eta_positive > 1:
    raise ValueError("eta_positive must be in (1, inf).")
if not 0 < eta_negative < 1:
    raise ValueError("eta_negative must be in (0, 1).")

if (
    self._last_weights_derivative is None
    or self._last_biases_derivative is None
):
    self._last_weights_derivative = np.zeros(
        self._weights_derivative.shape
    )
    self._last_biases_derivative = np.zeros(
        self._biases_derivative.shape
    )
    self._weights_delta = (
        np.ones(self._weights_derivative.shape) * delta_zero
    )
    self._biases_delta = (
        np.ones(self._biases_derivative.shape) * delta_zero
    )

same_sign = (
    self._weights_derivative *
    self._last_weights_derivative > 0
)
self._weights_delta[same_sign] = np.minimum(
    self._weights_delta[same_sign] * eta_positive, delta_max
)
self._weights[same_sign] -= (
    np.sign(self._weights_derivative[same_sign]) *
    self._weights_delta[same_sign]
)
self._last_weights_derivative[same_sign] = (
    self._weights_derivative[same_sign]
)
```

```
diff_sign = (
    self._weights_derivative *
    self._last_weights_derivative < 0
)
self._weights_delta[diff_sign] = np.maximum(
    self._weights_delta[diff_sign] * eta_negative, delta_min
)
self._last_weights_derivative[diff_sign] = 0

no_sign = (
    self._weights_derivative *
    self._last_weights_derivative == 0
)
self._weights[no_sign] -= (
    np.sign(self._weights_derivative[no_sign]) *
    self._weights_delta[no_sign]
)
self._last_weights_derivative[no_sign] = (
    self._weights_derivative[no_sign]
)

same_sign = (
    self._last_biases_derivative *
    self._biases_derivative > 0
)
self._biases[same_sign] -= (
    np.sign(self._biases_derivative[same_sign]) *
    self._biases_delta[same_sign]
)
self._biases_delta[same_sign] = np.minimum(
    self._biases_delta[same_sign] * eta_positive, delta_max
)
self._last_biases_derivative[same_sign] = (
    self._biases_derivative[same_sign]
)
```

```
diff_sign = (
    self._biases_derivative *
    self._last_biases_derivative < 0
)
self._biases_delta[diff_sign] = np.maximum(
    self._biases_delta[diff_sign] * eta_negative, delta_min
)
self._last_biases_derivative[diff_sign] = 0

no_sign = (
    self._biases_derivative *
    self._last_biases_derivative == 0
)
self._biases[no_sign] -= (
    np.sign(self._biases_derivative[no_sign]) *
    self._biases_delta[no_sign]
)
self._last_biases_derivative[no_sign] = (
    self._biases_derivative[no_sign]
)

self._weights_derivative = np.zeros(self._weights.shape)
self._biases_derivative = np.zeros(self._biases.shape)
```

Osserviamo come tale metodo effettui l'aggiornamento dei pesi e dei bias del layer facendo uso delle derivate precedentemente calcolate nella *backward_propagation* proprio come il metodo *update* della classe Dense, tuttavia, in questo caso la regola di aggiornamento impiegata è quella dell' RPROP 2.2.3.2, ossia:

$$w_{ij}^{(t)} = w_{i,j}^{(t-1)} - sign\left(\frac{dE^{(t)}}{dw_{ij}}\right) \cdot \Delta_{ij} \tag{4.1}$$

Inizialmente il metodo verifica l'esistenza delle derivate dei parametri calcolate nell'epoca precedente, in caso contrario inizializza i Δ_{ij} con il parametro *delta_zero* e le derivate dell'epoca precedente a zero (si tenga presente che sarebbe stato possibile inizializzare la derivata all'epoca $t = 1$ applicando una classica discesa del gradiente, tuttavia, per leggerezza del codice, si è scelto di adottare l'inizializzazione mediante

parametro *delta_zero*).
Successivamente viene effettuato l'aggiornamento dei Δ_{ij}, confrontando il segno del prodotto tra la derivata dell'epoca corrente t e quella dell'epoca precedente $t-1$, per poi aggiornare i pesi e i bias mediante:

$$w_{ij} = w_{ij} - sign\left(\frac{\partial E^{(t)}}{\partial w_{ij}}\right) \cdot \Delta_{ij} \tag{4.2}$$

4.2 Set-up sperimentale

In questo capitolo, viene illustrato il processo di selezione di una rete neurale densa a singolo strato interno, con resilient back-propagation (RPROP) come algoritmo di aggiornamento dei parametri. Nello specifico, il processo di selezione viene affrontato secondo un approccio di k-fold cross-validation con ricerca a griglia e ha come obiettivo l'identificazione degli iper-parametri sub-ottimali per la rete, quali, numero di nodi interni e parametri dell'RPROP, ovvero, *eta_negative* e *eta_positive*. I vari esperimenti sono stati effettuati sul dataset MNIST, dunque, il compito di una generica rete neruale è quello di classificare l'input ricevuto in ingresso, il quale rappresenta un'immagine di una cifra scritta a mano, in una tra le 10 classi possibili.

4.2.1 Selezione del modello

Al fine di raffinare al meglio la ricerca degli iper-parametri desiderati, ossia, numero di nodi interni e parametri dell'RPROP (η^-, η^+), la valutazione di ogni modello considerato si è basata su un approccio di tipo k-fold cross validation, inoltre, il processo di selezione è stato iterato due volte, al fine di effettuare una ricerca più granulare e dunque esaustiva.

Caratteristiche del modello di rete: Le caratteristiche comuni ad ogni modello di rete neurale considerato sono le seguenti:

- **Numero di strati nascosti**: 1 (Shallow);
- **Numero di nodi in input**: 784;

- **Numero di nodi di output**: 10;
- **Funzione di attivazione interna**: Sigmoide;
- **Funzione di attivazione di output**: Identità;
- **Funzione di errore**: Softmax Cross Entropy;
- **Regola di aggiornamento**: RPROP.

4.2.1.1 Primo processo di selezione

Nel primo processo di selezione è stata considerata la seguente configurazione:

- **Dimensione del dataset**: 5000;
- **Approccio di valutazione**: k-fold Cross Validation con $k = 5$;
- **Epoche di addestramento per ogni fold**: 50;
- **Metrica considerata**: Accuratezza;

Mentre, i valori considerati per gli iper-parametri, sono i seguenti:

- **Numero di nodi interni**: $C = \{20, 40, 60, 80, 100\}$;
- $\boldsymbol{\eta^-}$: $N^- = \{0.10, 0.30, 0.50, 0.70, 0.90\}$;
- $\boldsymbol{\eta^+}$: $N^+ = \{1.10, 1.30, 1.50, 1.70, 1.90\}$.

Il totale dei test effettuati è quindi $|\, C \times N^- \times N^+| = 125$, per una durata di circa 11 ore. Si tenga presente che, al fine di rendere le valutazioni il più attendibili possibile, il dataset impiegato è il medesimo per ogni modello. Si riportano di seguito i risultati ottenuti:

Neuroni	$\boldsymbol{\eta^-}$	$\boldsymbol{\eta^+}$	**Accuratezza**
20	0.10	1.10	0.87960 ± 0.01068
20	0.10	1.30	0.87520 ± 0.01319
20	0.10	1.50	0.86999 ± 0.01088

Neuroni	η^-	η^+	**Accuratezza**
20	0.10	1.70	0.86959 ± 0.00557
20	0.10	1.90	0.86380 ± 0.00950
20	0.30	1.10	0.88720 ± 0.00574
20	0.30	1.30	0.86700 ± 0.00715
20	0.30	1.50	0.86359 ± 0.00909
20	0.30	1.70	0.86199 ± 0.00769
20	0.30	1.90	0.86159 ± 0.00796
20	0.50	1.10	0.88559 ± 0.00257
20	0.50	1.30	0.86979 ± 0.00604
20	0.50	1.50	0.86840 ± 0.01125
20	0.50	1.70	0.86140 ± 0.00677
20	0.50	1.90	0.86399 ± 0.01258
20	0.70	1.10	0.88780 ± 0.00470
20	0.70	1.30	0.87120 ± 0.00416
20	0.70	1.50	0.86640 ± 0.00682
20	0.70	1.70	0.86520 ± 0.00762
20	0.70	1.90	0.83719 ± 0.00640
20	0.90	1.10	0.88540 ± 0.00674
20	0.90	1.30	0.86460 ± 0.01409
20	0.90	1.50	0.85300 ± 0.01519
20	0.90	1.70	0.40820 ± 0.07335
20	0.90	1.90	0.43480 ± 0.03517
40	0.10	1.10	0.89420 ± 0.00980
40	0.10	1.30	0.88340 ± 0.00821
40	0.10	1.50	0.88460 ± 0.00615
40	0.10	1.70	0.87459 ± 0.00581
40	0.10	1.90	0.87300 ± 0.00777
40	0.30	1.10	0.90320 ± 0.00430
40	0.30	1.30	0.88579 ± 0.00890

Neuroni	η^-	η^+	**Accuratezza**
40	0.30	1.50	0.87759 ± 0.00674
40	0.30	1.70	0.87220 ± 0.00879
40	0.30	1.90	0.87420 ± 0.00818
40	0.50	1.10	0.90020 ± 0.00778
40	0.50	1.30	0.87720 ± 0.00708
40	0.50	1.50	0.87620 ± 0.00591
40	0.50	1.70	0.86959 ± 0.01142
40	0.50	1.90	0.86880 ± 0.00800
40	0.70	1.10	0.89819 ± 0.00591
40	0.70	1.30	0.88260 ± 0.00915
40	0.70	1.50	0.87740 ± 0.00601
40	0.70	1.70	0.86920 ± 0.00890
40	0.70	1.90	0.85820 ± 0.01663
40	0.90	1.10	0.89580 ± 0.00943
40	0.90	1.30	0.87880 ± 0.00865
40	0.90	1.50	0.86919 ± 0.00901
40	0.90	1.70	0.46740 ± 0.08485
40	0.90	1.90	0.41079 ± 0.08081
60	0.10	1.10	0.90100 ± 0.00869
60	0.10	1.30	0.89320 ± 0.01121
60	0.10	1.50	0.88500 ± 0.01131
60	0.10	1.70	0.88160 ± 0.00553
60	0.10	1.90	0.88279 ± 0.00556
60	0.30	1.10	0.90040 ± 0.00803
60	0.30	1.30	0.88720 ± 0.00567
60	0.30	1.50	0.88260 ± 0.00257
60	0.30	1.70	0.88040 ± 0.00830
60	0.30	1.90	0.87820 ± 0.00744
60	0.50	1.10	0.90000 ± 0.00513

Neuroni	η^-	η^+	**Accuratezza**
60	0.50	1.30	0.88400 ± 0.00389
60	0.50	1.50	0.88639 ± 0.00287
60	0.50	1.70	0.87260 ± 0.00402
60	0.50	1.90	0.87419 ± 0.00908
60	0.70	1.10	0.89879 ± 0.00604
60	0.70	1.30	0.88599 ± 0.00517
60	0.70	1.50	0.87560 ± 0.00656
60	0.70	1.70	0.87799 ± 0.00887
60	0.70	1.90	0.80820 ± 0.08798
60	0.90	1.10	0.89819 ± 0.01008
60	0.90	1.30	0.89100 ± 0.00635
60	0.90	1.50	0.87880 ± 0.01151
60	0.90	1.70	0.54140 ± 0.13577
60	0.90	1.90	0.45640 ± 0.06249
80	0.10	1.10	0.90579 ± 0.00746
80	0.10	1.30	0.89680 ± 0.00711
80	0.10	1.50	0.88120 ± 0.00386
80	0.10	1.70	0.87880 ± 0.00495
80	0.10	1.90	0.87520 ± 0.01085
80	0.30	1.10	0.90360 ± 0.00538
80	0.30	1.30	0.88240 ± 0.00449
80	0.30	1.50	0.88920 ± 0.00370
80	0.30	1.70	0.87579 ± 0.00652
80	0.30	1.90	0.87600 ± 0.00916
80	0.50	1.10	0.89740 ± 0.00349
80	0.50	1.30	0.88680 ± 0.00503
80	0.50	1.50	0.88179 ± 0.00785
80	0.50	1.70	0.87940 ± 0.00300
80	0.50	1.90	0.87799 ± 0.00340

Neuroni	η^-	η^+	**Accuratezza**
80	0.70	1.10	0.90240 ± 0.00422
80	0.70	1.30	0.88540 ± 0.01267
80	0.70	1.50	0.87820 ± 0.00614
80	0.70	1.70	0.87199 ± 0.00969
80	0.70	1.90	0.84160 ± 0.04511
80	0.90	1.10	0.90280 ± 0.00773
80	0.90	1.30	0.88279 ± 0.00746
80	0.90	1.50	0.87859 ± 0.00997
80	0.90	1.70	0.80120 ± 0.06783
80	0.90	1.90	0.38739 ± 0.08326
100	0.10	1.10	0.90779 ± 0.00719
100	0.10	1.30	0.89540 ± 0.00615
100	0.10	1.50	0.88900 ± 0.00855
100	0.10	1.70	0.88100 ± 0.00485
100	0.10	1.90	0.89000 ± 0.00481
100	0.30	1.10	0.90640 ± 0.00520
100	0.30	1.30	0.89300 ± 0.00860
100	0.30	1.50	0.88460 ± 0.00813
100	0.30	1.70	0.87600 ± 0.00721
100	0.30	1.90	0.87840 ± 0.00806
100	0.50	1.10	0.90560 ± 0.00574
100	0.50	1.30	0.88179 ± 0.00416
100	0.50	1.50	0.88640 ± 0.00861
100	0.50	1.70	0.87379 ± 0.00705
100	0.50	1.90	0.88180 ± 0.00515
100	0.70	1.10	0.90339 ± 0.00640
100	0.70	1.30	0.88160 ± 0.00900
100	0.70	1.50	0.88200 ± 0.00989
100	0.70	1.70	0.87920 ± 0.01016

Neuroni	η^-	η^+	Accuratezza
100	0.70	1.90	0.87900 ± 0.00952
100	0.90	1.10	0.90380 ± 0.00945
100	0.90	1.30	0.89260 ± 0.00516
100	0.90	1.50	0.88520 ± 0.00788
100	0.90	1.70	0.82040 ± 0.04975
100	0.90	1.90	0.42460 ± 0.08977

Tabella 4.1: Primo processo di selezione

Le righe della tabella evidenziate in verde indicano i modelli che hanno ottenuto le prestazioni migliori (molto superiori rispetto la media di 0.85160), mentre, quelle evidenziate in rosso, indicano i modelli che hanno ottenuto le prestazioni peggiori. Il modello evidenziato in verde scuro è quello che ha conseguito la prestazione (accuratezza di 0.90779 ± 0.00719) più alta tra tutti i test, ossia, il modello avente i seguenti iper-parametri:

- **Numero di nodi interni**: 100;
- η^-: 0.10;
- η^+: 1.10.

D'ora in avanti, per semplicità di notazione, indicheremo il modello ottenuto con Q. Il prossimo passo è la raffinazione degli iper-parametri di Q mediante una ricerca a griglia più fine, con lo scopo di individuare un modello $Q^{'}$, tale per cui, detta A l'accuratezza, $A(Q^{'}) > A(Q)$.

4.2.1.2 Secondo processo di selezione

La configurazione del secondo processo di selezione è la medesima del primo:

- **Dimensione del dataset**: 5000;
- **Approccio di valutazione**: k-fold Cross Validation con $k = 5$;
- **Epoche di addestramento per ogni fold**: 50;

- **Metrica considerata**: Accuratezza;

Mentre, i valori considerati per gli iper-parametri, sono i seguenti:

- **Numero di nodi interni**: $C = \{85, 90, 95, 100\}$;
- $\boldsymbol{\eta^-}$: $N^- = \{0.05, 0.10, 0.15, 0.20, 0.25\}$;
- $\boldsymbol{\eta^+}$: $N^+ = \{1.05, 1.10, 1.15, 1.20, 1.25\}$.

Il totale dei test effettuati è quindi $|C \times N^- \times N^+| = 100$, per una durata di circa 8 ore. Si tenga presente che, al fine di rendere le valutazioni il più attendibili possibile, il dataset impiegato è il medesimo per ogni modello. Si riportano di seguito i risultati ottenuti:

Neuroni	$\boldsymbol{\eta^-}$	$\boldsymbol{\eta^+}$	**Accuratezza**
85	0.05	1.05	0.89500 ± 0.01012
85	0.05	1.10	0.90620 ± 0.00475
85	0.05	1.15	0.90900 ± 0.00715
85	0.05	1.20	0.91160 ± 0.00471
85	0.05	1.25	0.89720 ± 0.00924
85	0.10	1.05	0.90680 ± 0.00741
85	0.10	1.10	0.90940 ± 0.00417
85	0.10	1.15	0.90780 ± 0.00231
85	0.10	1.20	0.90380 ± 0.00842
85	0.10	1.25	0.90160 ± 0.00313
85	0.15	1.05	0.91560 ± 0.00320
85	0.15	1.10	0.91700 ± 0.00641
85	0.15	1.15	0.90640 ± 0.00776
85	0.15	1.20	0.90660 ± 0.00646
85	0.15	1.25	0.90880 ± 0.00825
85	0.20	1.05	0.92000 ± 0.00850
85	0.20	1.10	0.91200 ± 0.00480

Neuroni	η^-	η^+	**Accuratezza**
85	0.20	1.15	0.91060 ± 0.00475
85	0.20	1.20	0.91200 ± 0.00477
85	0.20	1.25	0.90480 ± 0.01119
85	0.25	1.05	0.91820 ± 0.00720
85	0.25	1.10	0.91900 ± 0.00600
85	0.25	1.15	0.90460 ± 0.00662
85	0.25	1.20	0.90440 ± 0.00873
85	0.25	1.25	0.90040 ± 0.00646
90	0.05	1.05	0.89260 ± 0.008662
90	0.05	1.10	0.90240 ± 0.00890
90	0.05	1.15	0.90540 ± 0.00742
90	0.05	1.20	0.91180 ± 0.00611
90	0.05	1.25	0.90640 ± 0.00911
90	0.10	1.05	0.91340 ± 0.01032
90	0.10	1.10	0.90980 ± 0.00702
90	0.10	1.15	0.90680 ± 0.00519
90	0.10	1.20	0.90480 ± 0.00515
90	0.10	1.25	0.90360 ± 0.00523
90	0.15	1.05	0.90720 ± 0.00591
90	0.15	1.10	0.91180 ± 0.00440
90	0.15	1.15	0.90660 ± 0.00740
90	0.15	1.20	0.90560 ± 0.00947
90	0.15	1.25	0.90240 ± 0.00863
90	0.20	1.05	0.91720 ± 0.00390
90	0.20	1.10	0.91400 ± 0.00782
90	0.20	1.15	0.90500 ± 0.00770
90	0.20	1.20	0.90400 ± 0.00460
90	0.20	1.25	0.90380 ± 0.00808
90	0.25	1.05	0.91840 ± 0.00508

Neuroni	η^-	η^+	**Accuratezza**
90	0.25	1.10	0.91660 ± 0.00845
90	0.25	1.15	0.90700 ± 0.00460
90	0.25	1.20	0.90480 ± 0.00349
90	0.25	1.25	0.89740 ± 0.00621
95	0.05	1.05	0.90060 ± 0.00752
95	0.05	1.10	0.90840 ± 0.00801
95	0.05	1.15	0.90740 ± 0.00608
95	0.05	1.20	0.90700 ± 0.00374
95	0.05	1.25	0.90200 ± 0.00346
95	0.10	1.05	0.90240 ± 0.00387
95	0.10	1.10	0.91360 ± 0.00668
95	0.10	1.15	0.90580 ± 0.00810
95	0.10	1.20	0.90840 ± 0.00895
95	0.10	1.25	0.90340 ± 0.00950
95	0.15	1.05	0.91080 ± 0.00305
95	0.15	1.10	0.91060 ± 0.00546
95	0.15	1.15	0.91020 ± 0.01057
95	0.15	1.20	0.90820 ± 0.00235
95	0.15	1.25	0.90060 ± 0.00372
95	0.20	1.05	0.91480 ± 0.00915
95	0.20	1.10	0.91140 ± 0.00771
95	0.20	1.15	0.90620 ± 0.00503
95	0.20	1.20	0.90500 ± 0.00562
95	0.20	1.25	0.90340 ± 0.00789
95	0.25	1.05	0.91440 ± 0.00708
95	0.25	1.10	0.91180 ± 0.01133
95	0.25	1.15	0.91060 ± 0.00691
95	0.25	1.20	0.90200 ± 0.00701
95	0.25	1.25	0.90240 ± 0.00711

Neuroni	η^-	η^+	**Accuratezza**
100	0.05	1.05	0.90360 ± 0.00744
100	0.05	1.10	0.90600 ± 0.01145
100	0.05	1.15	0.90700 ± 0.00635
100	0.05	1.20	0.90680 ± 0.00349
100	0.05	1.25	0.90800 ± 0.00424
100	0.10	1.05	0.90160 ± 0.00808
100	0.10	1.10	0.91480 ± 0.00897
100	0.10	1.15	0.90678 ± 0.00872
100	0.10	1.20	0.90320 ± 0.00560
100	0.10	1.25	0.90460 ± 0.00628
100	0.15	1.05	0.91700 ± 0.00666
100	0.15	1.10	0.90740 ± 0.00705
100	0.15	1.15	0.91560 ± 0.00538
100	0.15	1.20	0.90400 ± 0.00723
100	0.15	1.25	0.90220 ± 0.00685
100	0.20	1.05	0.92040 ± 0.00886
100	0.20	1.10	0.91880 ± 0.00324
100	0.20	1.15	0.90980 ± 0.00386
100	0.20	1.20	0.90920 ± 0.00491
100	0.20	1.25	0.90740 ± 0.00531
100	0.25	1.05	0.92240 ± 0.00770
100	0.25	1.10	0.91560 ± 0.00794
100	0.25	1.15	0.91260 ± 0.00492
100	0.25	1.20	0.90180 ± 0.00600
100	0.25	1.25	0.90320 ± 0.00523

Tabella 4.2: Secondo processo di selezione

In questo caso, le righe della tabella evidenziate in verde indicano i modelli che hanno ottenuto le prestazioni migliori rispetto al modello Q ottenuto nella prima selezione

(evidenziato in blu), mentre, quelle evidenziate in rosso, indicano i modelli che hanno ottenuto delle prestazioni peggiori rispetto a Q. Il modello $Q^{'}$ evidenziato in verde scuro è quello che ha conseguito la prestazione (accuratezza di 0.92240 ± 0.00770) più alta tra tutti i test, ossia, il modello avente i seguenti iper-parametri:

- **Numero di nodi interni**: 100;
- $\boldsymbol{\eta}^{-}$: 0.25;
- $\boldsymbol{\eta}^{+}$: 1.05.

Avendo individuato il modello definito $Q^{'}$ è ora possibile procedere all'addestramento di quest'ultimo avvalendosi dei criteri di early-stop disponibili. Al fine di valutare il vantaggio ottenuto dalla selezione, i risultati ottenuti dall'addestramento di $Q^{'}$ verranno confrontati con i risultati ottenuti da altri modelli, tra i quali Q.

4.2.2 Processo di apprendimento

Di seguito vengono elencati i modelli considerati nella fase di addestramento. I modelli *Q1-100* e *Q2-100* indicano rispettivamente quelli ottenuti dalla prima e dalla seconda fase di selezione, mentre, i restanti indicano quelli ottenuti generando gli iper-parametri in modo casuale: $n \in [20, 100]$ numero di neuroni, $\eta^{-} \in [0.1, 0.9]$ e $\eta^{+} \in [1.1, 1.9]$.

Modello	**Neuroni**	$\boldsymbol{\eta}^{-}$	$\boldsymbol{\eta}^{+}$	**Ottenuto da**
Q1-100	100	0.10	1.10	I selezione
Q2-100	100	0.25	1.05	II selezione
R1-36	36	0.69	1.50	casuale
R2-66	66	0.24	1.70	casuale
R3-70	70	0.41	1.64	casuale

Tabella 4.3: Modelli addestrati

Addestramento: La fase di addestramento è stata eseguita, per ogni modello, con le seguenti condizioni:

- **Dimensione del dataset**: 60000;
- **Dimensione del training set**: 45000 (75% del dataset);
- **Dimensione del validation set**: 15000 (25% del dataset);
- **Numero di epoche massimo**: 5000;
- **Early-Stop**: discrepanza del valore dell'errore di validazione del 5% rispetto al miglior valore (minimo assoluto) raggiunto;
- **Modalità di aggiornamento**: full-batch.

4.2.2.1 Apprendimento del modello Q1-100

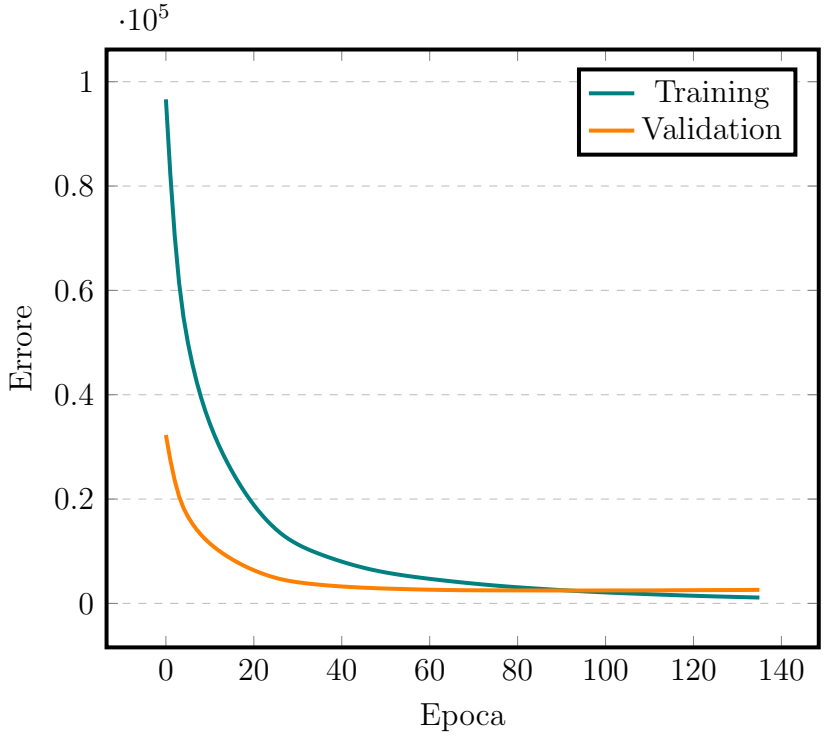


Figura 4.2: Errore del modello Q1-100

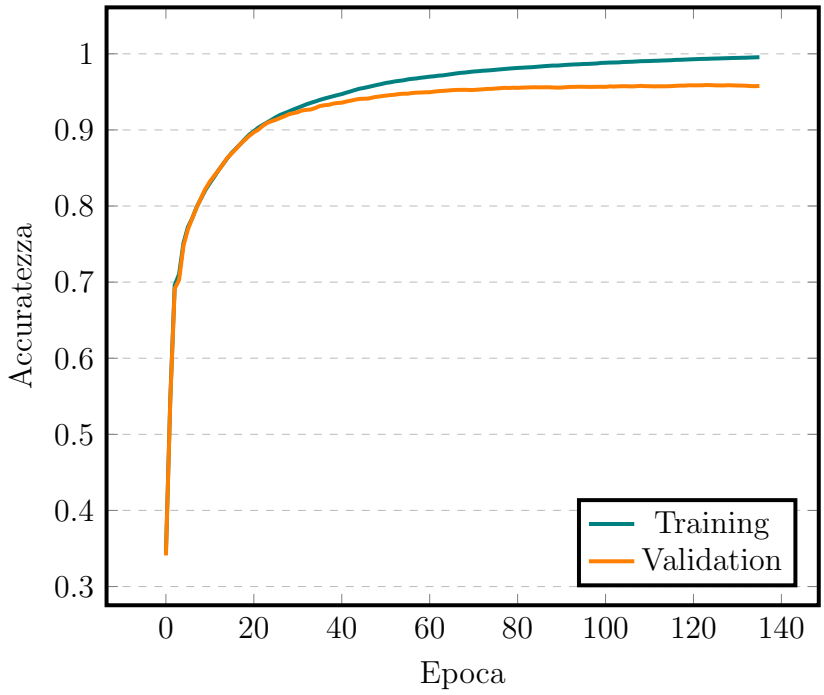


Figura 4.3: Accuratezza del modello Q1-100

4.2.2.2 Apprendimento del modello Q2-100

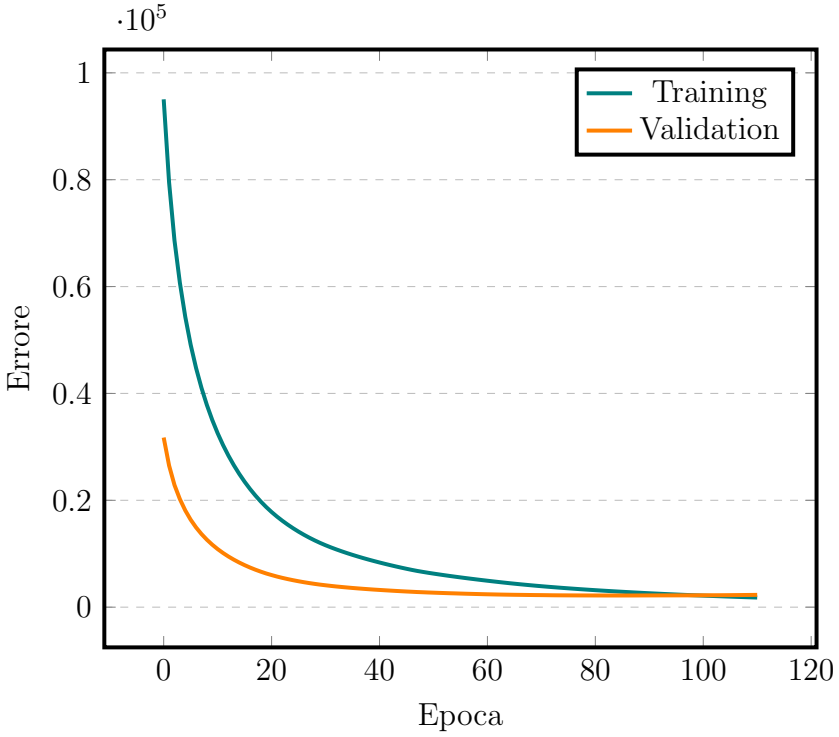


Figura 4.4: Errore del modello Q2-100

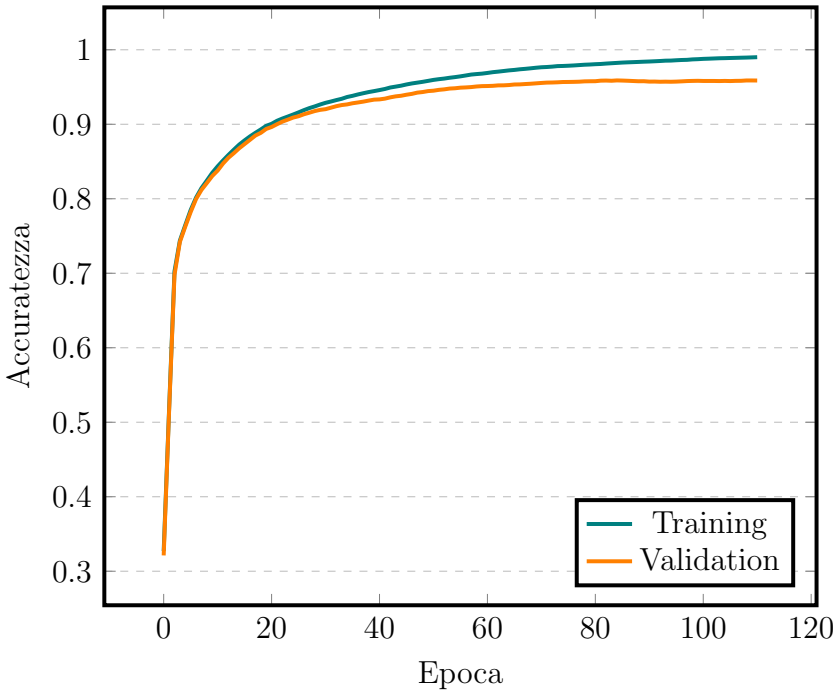


Figura 4.5: Accuratezza del modello Q2-100

4.2.2.3 Apprendimento del modello R1-36

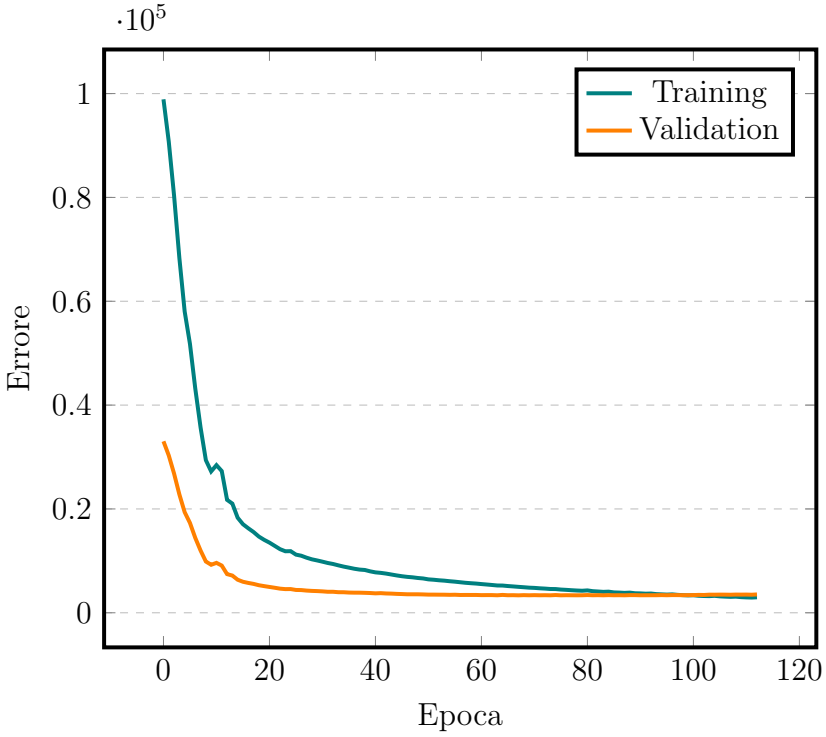


Figura 4.6: Errore del modello R1-36

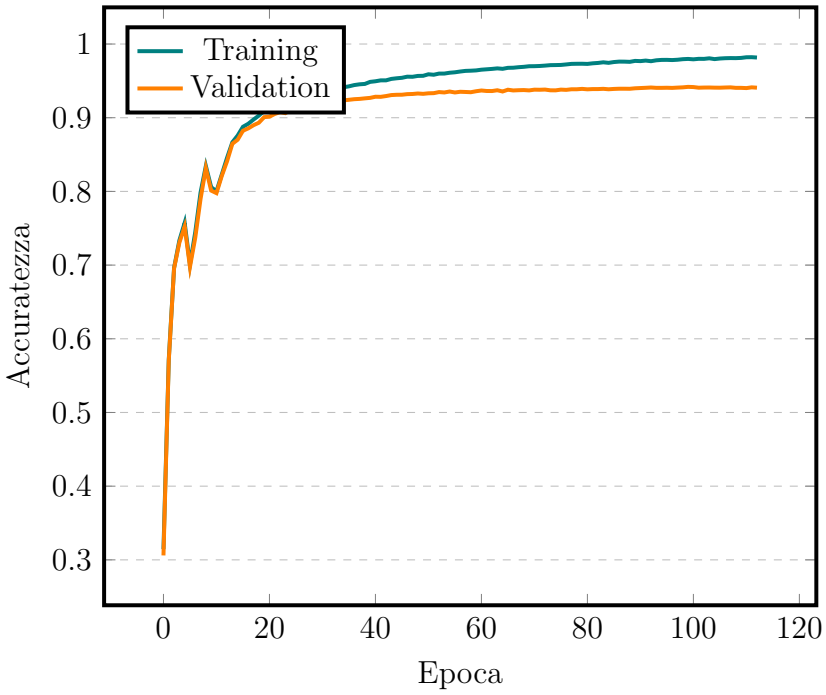


Figura 4.7: Accuratezza del modello R1-36

4.2.2.4 Apprendimento del modello R2-66

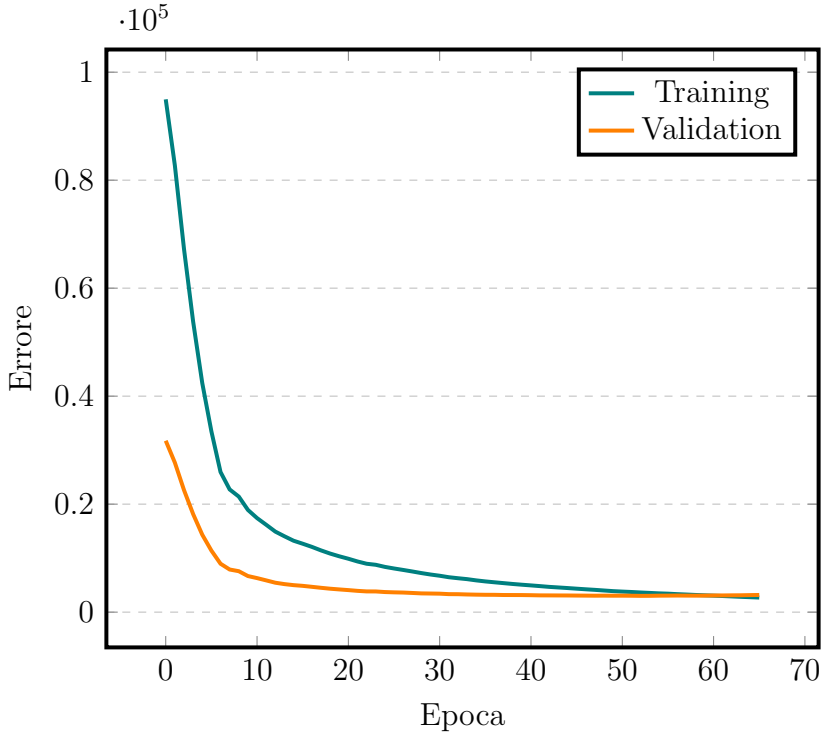


Figura 4.8: Errore del modello R2-66

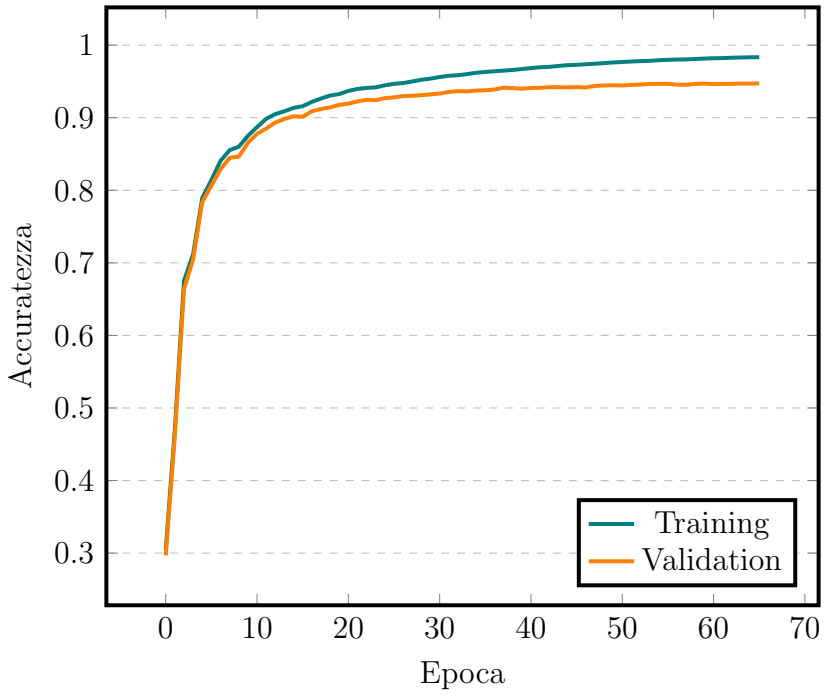


Figura 4.9: Accuratezza del modello R2-66

4.2.2.5 Apprendimento del modello R3-70

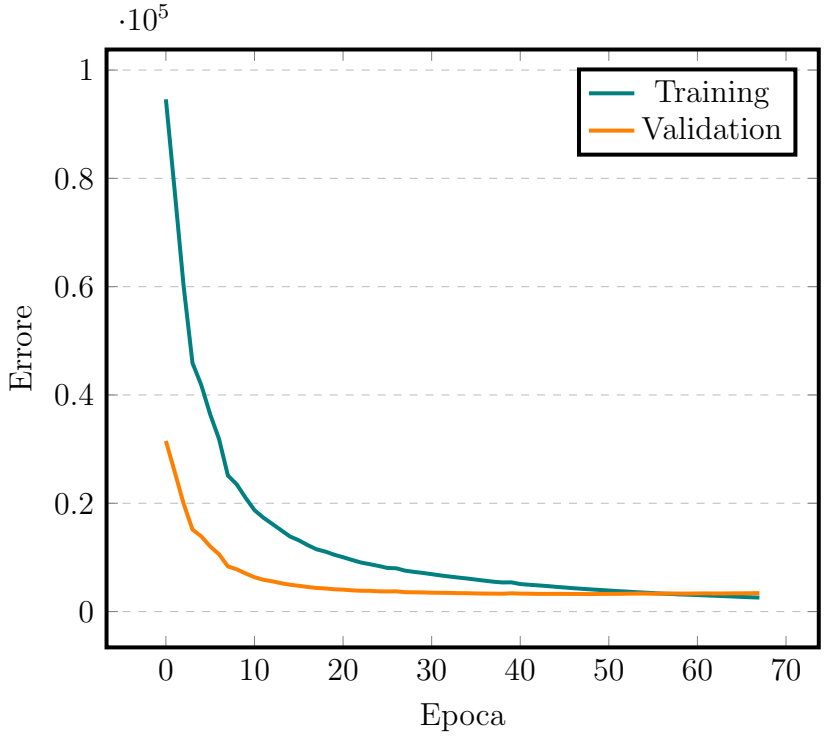


Figura 4.10: Errore del modello R3-70

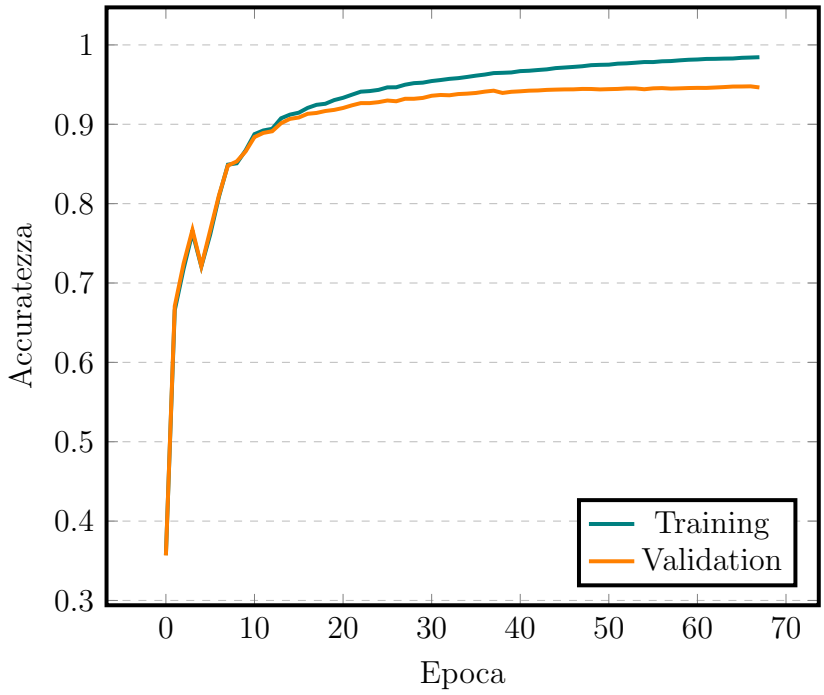


Figura 4.11: Accuratezza del modello R3-70

4.3 Discussione dei risultati ottenuti

Di seguito si elencano i risultati ottenuti durante la fase di addestramento dei modelli indicati in 4.3:

Modello	**Epoca**	**Accuratezza**		**F1-Score**	
		T	**V**	**T**	**V**
Q1-100	135	0.99555	0.95780	0.99549	0.95742
Q2-100	110	0.99004	0.95873	0.99000	0.95837
R1-36	112	0.98166	0.94073	0.98152	0.94014
R2-66	65	0.98337	0.94726	0.98324	0.94674
R3-70	67	0.98451	0.94640	0.98440	0.94573

Tabella 4.4: Risultati ottenuti dall'addestramento (DS - 60000)

I due modelli *Q1-100* e *Q2-100* ottenuti, rispettivamente, nel primo e nel secondo processo di selezione, hanno conseguito delle prestazioni migliori di circa un punto percentuale rispetto ai modelli "generati" in modo casuale, tuttavia, l'ottimizzazione desiderata dal secondo processo di selezione su *Q1-100* non ha introdotto differenze significative per giustificarne l'elevato costo computazionale. Da tali considerazioni si deduce che la griglia inizialmente impiegata per la ricerca degli iper-parametri era sufficientemente "densa".

Inoltre, è interessante osservare che i modelli *R2-66* e *R3-70* hanno raggiunto l'overfitting (ricordiamo che il criterio di early-stop adottato era una discrepanza del valore dell'errore di validazione del 5% rispetto al minimo raggiunto) prima degli altri modelli, rendendo di fatto l'apprendimento più rapido seppur meno accurato. Tuttavia, ripetendo le sperimentazioni con un dataset di 5000 elementi, il vantaggio della model selection diviene più chiaro, infatti, le discrepanze tra i modelli di tipo Q e R sono nette e, inoltre, il vantaggio ottenuto dalla seconda fase di selezione è più evidente:

Modello	Epoca	Accuratezza		F1-Score	
		T	V	T	V
Q1-100	50	0.99413	0.90960	0.99420	0.90942
Q2-100	45	0.99173	0.92240	0.99164	0.92189
R1-36	23	0.96560	0.88960	0.96539	0.88565
R2-66	17	0.96480	0.88800	0.96371	0.88563
R3-70	7	0.81733	0.79040	0.81594	0.78577

Tabella 4.5: Risultati ottenuti dall'addestramento (DS - 5000)

Ripetendo un'ultima volta le sperimentazioni con un dataset di 20000 elementi, notiamo come i modelli di tipo R ottengano nuovamente accuratezze alte, tuttavia, la distinzione tra Q e R rimane netta, assieme al vantaggio introdotto dalla seconda model selection.

Modello	Epoca	Accuratezza		F1-Score	
		T	V	T	V
Q1-100	64	0.98413	0.93666	0.98409	0.93499
Q2-100	84	0.99513	0.94320	0.99540	0.94222
R1-36	45	0.97113	0.91920	0.97091	0.91869
R2-66	33	0.97360	0.91900	0.97324	0.91822
R3-70	29	0.96453	0.92120	0.96400	0.92035

Tabella 4.6: Risultati ottenuti dall'addestramento (DS - 20000)

In conclusione possiamo affermare che, nonostante potrebbe non apparire utile in presenza di una grande mole di dati, la model selection è comunque necessaria per valutare l'esistenza di modelli con performance migliori, inoltre, è opportuno considerare che, a seconda del campo di applicazione, la differenza percentuale tra i modelli Q e R potrebbe essere ritenuta considerevole.

Capitolo 5

Conclusioni e Sviluppi Futuri

Questo progetto ha avuto come obiettivo quello di inoltrasi nelle dinamiche di progettazione, implementazione e sperimentazione di una rete neurale con architettura feed-forward dense (full-connected). In particolare, è stata evidenziata l'utilità di una buona model selection, al fine di ottenere dei modelli con prestazioni superiori, anche in presenza di dataset con dimensioni ridotte.
Infine, grazie alla modularità e alla semplicità della libreria implementata, è possibile introdurre ulteriori tipologie di layer e modelli (e.g. CNN), oltre a nuove condizioni di early-stop, metriche e funzioni.