

Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Corso di Machine Learning

(mod. Neural Networks and Deep Learning)

Prof. Roberto Prevete

Progettazione ed implementazione di una libreria di funzioni per la realizzazione di una rete neurale multi-strato full-connected

Traccia n.7

CANDIDATI:

CAROFANO Mario Gabriele - N97000437
TRINCONE Alessandro - N97000461

Anno Accademico 2023-2024

Indice

1	Introduzione	1
1.1	Contesto e motivazioni	1
1.2	Sinossi	2
2	La libreria Neural-Network	3
2.1	Struttura delle directory	5
2.2	File aggiuntivi	6
2.2.1	Classi di errore	6
2.2.2	Costanti	7
2.2.3	Alias di tipo	11
2.2.4	Funzioni di attivazione	11
2.2.5	Funzioni di errore	13
2.2.6	Funzioni per la gestione del MNIST dataset	13
2.2.7	Funzioni per il plotting della fase di addestramento	14
2.2.8	Funzioni per il plotting della fase di testing	15
2.2.9	Altre funzioni	16
2.3	La classe Layer	17
2.3.1	Attributi di classe	17
2.3.2	Costruttore	18
2.3.3	Metodi pubblici	18
2.4	La classe TrainingReport	20
2.4.1	Attributi di classe	20
2.4.2	Metodi pubblici	21
2.5	La classe TrainingParams	23
2.5.1	Attributi di classe	24
2.6	La classe NeuralNetwork	26
2.6.1	Attributi di classe	26
2.6.2	Costruttore	28
2.6.3	Metodi privati	28
2.6.4	Metodi pubblici	33
3	Addestramento e sperimentazione	37
3.1	Setup sperimentale	37
3.1.1	Architettura della macchina	40
3.1.2	Architettura della rete	40
3.1.3	Iper-parametri di addestramento: Grid Search e Random Search	41
3.1.4	Utilizzo del dataset	41

3.2	Risultati	42
3.2.1	Grid Search (n.10): compromesso tra errore e accuratezza	43
3.2.2	Grid Search (n.17): massima accuratezza	44
3.2.3	Grid Search (n.1): compromesso tra accuratezza e tempo	45
3.2.4	Random Search (n.1): compromesso tra errore e accuratezza	46
3.2.5	Random Search (n.3): compromesso tra accuratezza e tempo	47
3.2.6	Random Search (n.0): la miglior combinazione di iper-parametri	48
4	Conclusioni	51
4.1	Considerazioni	51
4.2	Futuri sviluppi e miglioramenti	52
	Bibliografia	53

Elenco delle figure

1.1	Cifre scritte a mano	1
2.1	Class diagram della libreria Neural-Network	4
2.2	Struttura delle directory	5
2.3	Struttura di un neurone della rete neurale artificiale	19
2.4	Visualizzazione del contenuto di un oggetto della classe Layer	19
2.5	Visualizzazione del contenuto di un oggetto della classe TrainingReport	22
2.6	Visualizzazione del contenuto di un oggetto della classe TrainingParams	23
2.7	Pseudo-codice RPROP+	32
2.8	Visualizzazione del contenuto di un oggetto della classe NeuralNetwork	35
3.1	Esempio di applicazione della K-fold Cross Validation	38
3.2	Risultati degli esperimenti tramite Grid Search	42
3.3	Risultati degli esperimenti tramite Random Search	42
3.4	Report della Grid Search sulla combinazione n.10	43
3.5	Report della Grid Search sulla combinazione n.17	44
3.6	Report della Grid Search sulla combinazione n.1	45
3.7	Report della Random Search sulla combinazione n.1	46
3.8	Report della Random Search sulla combinazione n.3	47
3.9	Report della Random Search sulla combinazione n.0	48
3.10	Report di classificazione sulla combinazione n.0 (Random Search)	49
3.11	Predizione corretta della rete neurale per la cifra n.1 dal MNIST test set	50
3.12	Predizione errata della rete neurale per la cifra n.43 dal MNIST test set	50

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

Il riconoscimento di cifre scritte a mano è un problema classico nel campo del machine learning e delle reti neurali. Uno dei dataset più importanti per questo tipo di applicazione è il MNIST dataset, composto da migliaia di immagini in scala di grigi di cifre scritte a mano.

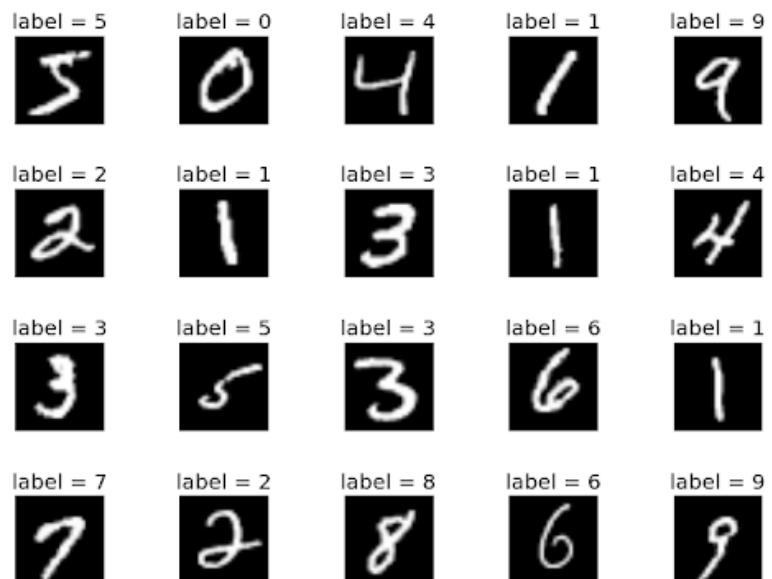


Figura 1.1: Alcuni esempi di cifre scritte a mano con etichetta dal MNIST dataset

In questo contesto, il presente studio si propone di esplorare e analizzare la progettazione, l'implementazione e la sperimentazione di una libreria di funzioni per reti neurali, con l'obiettivo di risolvere il problema del riconoscimento e della classificazione di cifre scritte a mano.

Si pone particolare attenzione alla fase di sperimentazione, nella quale si valutano le prestazioni di più modelli, caratterizzati da diverse combinazioni di iper-parametri ottenute da tecniche di ricerca come la "Grid Search" e la "Random Search", tramite l'utilizzo della nota tecnica di validazione "K-fold cross validation". In particolare, per la valutazione delle prestazioni dei modelli addestrati, sono stati analizzati e confrontati grafici e tabelle che illustrano i risultati di media e deviazione standard dell'errore di validazione, media e deviazione standard dell'accuracy di validazione, nonché l'andamento delle curve di apprendimento dell'errore e dell'accuracy di validazione.

1.2 Sinossi

Questa documentazione si divide nei seguenti capitoli:

- Il capitolo 2 - ***La libreria Neural-Network*** - a pagina 3 propone una relazione dettagliata riguardo la progettazione e l'implementazione di tutte le funzionalità, costanti, classi, attributi e metodi per la realizzazione della libreria **Neural-Network** [1].
- Il capitolo 3 - ***Addestramento e sperimentazione*** - a pagina 37 si occupa di descrivere il setup sperimentale, l'architettura dei modelli utilizzati e gli iper-parametri scelti per la fase di addestramento. Questa sezione si conclude con l'analisi dei risultati degli esperimenti effettuati.
- La documentazione si chiude con il capitolo 4 - ***Conclusioni*** - a pagina 51 che riassume e discute di quanto sviluppato nel lavoro di progetto e, inoltre, propone anche alcuni spunti di riflessione per dei possibili miglioramenti futuri.

Capitolo 2

La libreria Neural-Network

La libreria che viene presentata in questo capitolo implementa tutte le funzioni necessarie alla realizzazione di una rete neurale artificiale feed-forward fully-connected (e.g. Multilayer Perceptron) tramite paradigma di programmazione a oggetti.

- La libreria è interamente implementata in **linguaggio Python**. È stato scelto questo linguaggio principalmente per la vasta disponibilità di librerie e framework utili per il machine learning e l'addestramento di reti neurali artificiali.
- Una libreria fondamentale per lo sviluppo di questa libreria è **Numpy**, utile per un'elaborazione numerica efficiente. Essa fornisce supporto nell'utilizzo di array multidimensionali e nel calcolo delle operazioni matriciali, indispensabili per lo sviluppo di una rete neurale da zero (e.g. gestione pesi / bias, forward propagation, ...).
- Nonostante l'introduzione di overhead in termini di prestazioni, l'implementazione tramite **paradigma di programmazione a oggetti** offre, sicuramente, molti vantaggi in termini di organizzazione, manutenibilità e riutilizzabilità del codice.

Inizialmente, si era pensato di introdurre tre componenti chiave per la libreria: la classe **Neuron**, la classe **Layer** e la classe **Neural Network**. Successivamente, è stato pensato di rimuovere la classe **Neuron** per migliorare i tempi di esecuzione e sfruttare al massimo il parallelismo delle moltiplicazioni matriciali.

La figura 2.1 a pagina 4 mostra il class diagram.

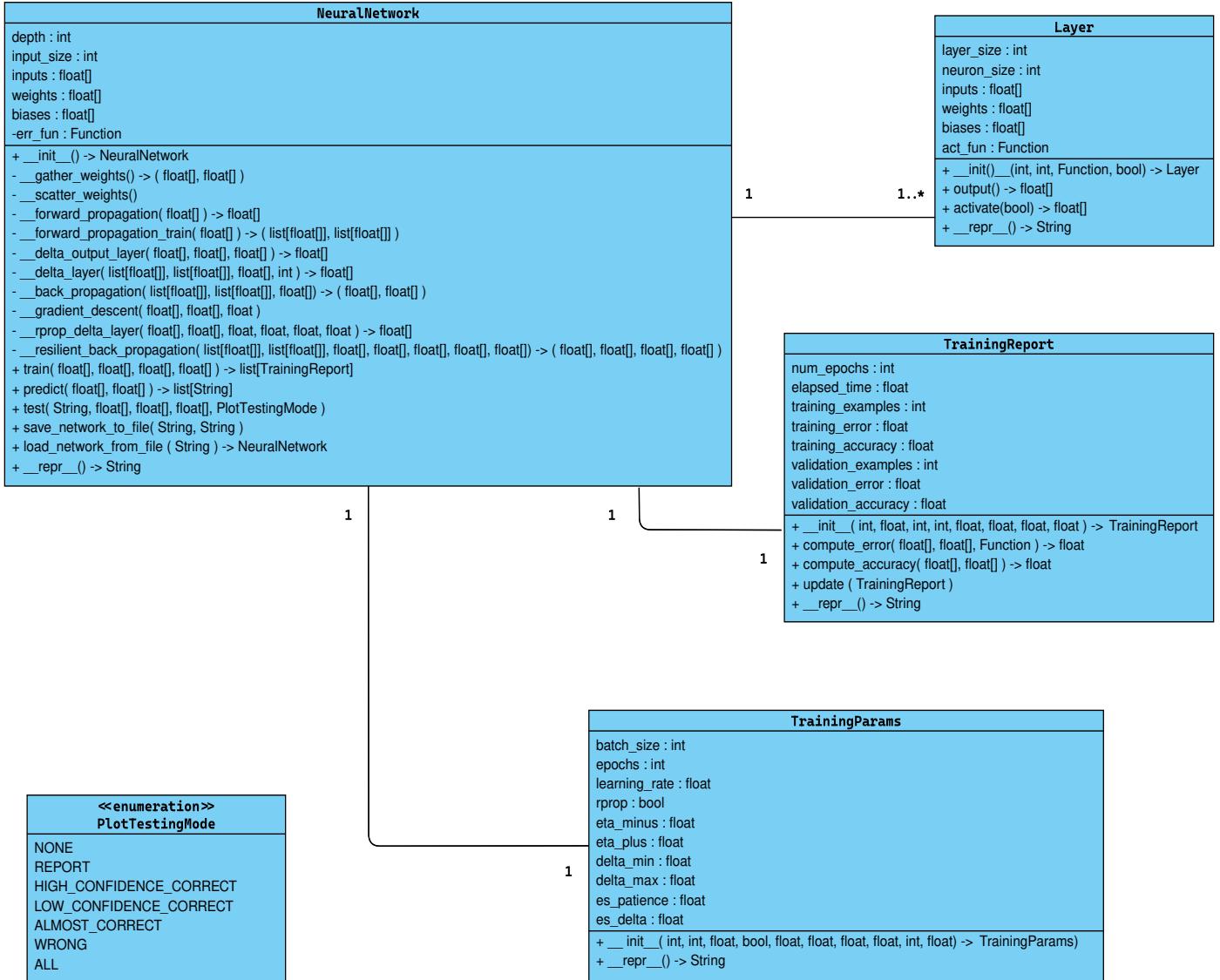


Figura 2.1: Class diagram della libreria Neural-Network

2.1 Struttura delle directory

Nome	Data di modifica	Dimensioni	Tipo
code	oggi, 19:31	--	Cartella
artificial_layer.py	l'altro ieri, 01:29	11 KB	Python Source
artificial_neural_network.py	oggi, 15:46	56 KB	Python Source
auxfunc.py	11 lug 2024, 19:10	16 KB	Python Source
constants.py	oggi, 15:18	13 KB	Python Source
dataset_functions.py	11 lug 2024, 15:55	7 KB	Python Source
k_fold.ipynb	oggi, 16:17	22 KB	Documento
plot_functions.py	11 lug 2024, 23:54	17 KB	Python Source
training_params.py	l'altro ieri, 14:52	9 KB	Python Source
training_report.py	l'altro ieri, 11:37	10 KB	Python Source
dataset	18 mag 2024, 11:23	--	Cartella
mnist_test.csv	13 mag 2024, 11:51	18.3 MB	valori...a virgola
mnist_train.csv	18 mag 2024, 11:23	109.6 MB	valori...a virgola
output	oggi, 19:32	--	Cartella
2024-07-12_15-20	ieri, 21:30	--	Cartella
grid_search	oggi, 19:32	--	Cartella
comb_1	l'altro ieri, 15:38	--	Cartella
comb_10	ieri, 11:15	--	Cartella
comb_17	ieri, 14:48	--	Cartella
stats.csv	ieri, 21:27	1 KB	valori...a virgola
stats.xlsx	oggi, 18:18	18 KB	Micros...(xlsx)
2024-07-14_10-03	oggi, 19:31	--	Cartella
random_search	oggi, 19:32	--	Cartella
comb_0	oggi, 10:50	--	Cartella
comb_1	oggi, 11:13	--	Cartella
comb_3	oggi, 11:59	--	Cartella
stats.csv	oggi, 14:09	397 byte	valori...a virgola
stats.xlsx	oggi, 14:12	13 KB	Micros...(xlsx)
2024-07-14_15-25	oggi, 16:12	--	Cartella
high_confidence_corrects	oggi, 16:16	--	Cartella
wrongs	oggi, 16:12	--	Cartella
net.pkl	oggi, 16:09	720 KB	Documento
testing-report.pdf	oggi, 16:12	16 KB	PDF
README.md	25 apr 2024, 12:29	3 KB	Markdown Text

Figura 2.2: Struttura delle directory

La libreria implementata dal team di sviluppo include tutti i file del codice sorgente, scritti in linguaggio Python, nella directory *code*. Ognuno di essi fornisce un’ampia documentazione interna tramite *docstring*, con descrizione delle funzioni, parametri in input e valori di ritorno.

Nella directory *dataset* sono presenti i file *.csv* del MNIST dataset, file di testo che contengono una rappresentazione in formato tabellare di immagini di cifre scritte a mano. Ogni riga corrisponde a un’immagine, mentre ogni colonna rappresenta il valore dei pixel, un valore intero compreso tra 0 (nero) e 255 (bianco). L’unica eccezione è la prima colonna, che è dedicata all’etichetta associata alla cifra (un valore intero compreso tra 0 e 9).

Infine, nella directory *output* sono memorizzati i grafici generati dall’esecuzione del notebook *k_fold.ipynb* nella directory *code*. Sono presenti sia grafici relativi alla fase di addestramento e validazione della rete neurale, sia grafici relativi alla fase di testing della rete neurale. Si vedano le sezioni 2.2.7 a pagina 14 e 2.2.8 a pagina 15 per una descrizione più accurata delle funzioni di plotting.

2.2 File aggiuntivi

Si inizia la trattazione della libreria implementata dal team di sviluppo introducendo il file *constants.py*, contenente le classi di errore, le costanti e gli alias di tipo.

La sezione prosegue con un'accurata descrizione delle funzioni accessorie necessarie per lo sviluppo della libreria. Tali funzioni sono collocate nei file *auxfunc.py*, *dataset_functions.py* e *plot_functions.py*. È stato scelto di separare il codice in molte funzioni principalmente per favorire la modularità / riusabilità, la manutenibilità e la semplicità, ma anche una migliore leggibilità dell'intera libreria, evitando sia il monolitismo eccessivo che la frammentazione estrema.

2.2.1 Classi di errore

Nel file *constants.py* sono state implementate le seguenti classi di errore, utili a rappresentare le cause di terminazione del programma.

Per la loro definizione si utilizza una classe che estende la classe *Exception*. Ogni classe di errore contiene al suo interno la definizione dei metodi *__init__()* e *__str__()*: il primo è il metodo che istanzia un oggetto della classe richiesta; il secondo è un metodo che fornisce una rappresentazione in stringa dell'oggetto (in questo caso, il messaggio di errore).

- *LayerError(Exception)*

È una classe di eccezione lanciata da errori relativi ai layer della rete neurale.

- *InputLayerError(Exception)*

È una classe di eccezione lanciata da errori relativi all'input layer della rete neurale.

-
- *TrainError(Exception)*

È una classe di eccezione lanciata da errori relativi alla fase di addestramento.

- *TestError(Exception)*

È una classe di eccezione lanciata da errori relativi alla fase di test.

-
- *ActivationFunctionError(Exception)*

È una classe di eccezione lanciata da errori relativi all'esecuzione della funzione di attivazione di un layer della rete neurale.

- *ErrorFunctionError(Exception)*

È una classe di eccezione lanciata da errori relativi all'esecuzione della funzione di errore della rete neurale.

2.2.2 Costanti

Nel file *constants.py* sono definite anche le seguenti costanti, utili per la definizione di alcuni parametri nel codice. Nel linguaggio di programmazione Python, le costanti si definiscono utilizzando nomi di variabili i cui caratteri sono tutti maiuscoli.

- **ETICHETTE_CLASSI = ["Cifra 0", ..., "Cifra 9"]**

La lista delle etichette rappresentative delle classi di output del training set e del test set del MNIST dataset.

- **NUMERO_CLASSI = len(ETICHETTE_CLASSI)**

Numero di classi di output del training set e del test set del MNIST dataset. Il suo valore è calcolato al tempo di esecuzione in base al numero di etichette disponibili.

- **COPPIE_TRAINING = 12500**

Numero di elementi da estrarre dal training set su cui eseguire la fase di addestramento della rete neurale.

- **COPPIE_TEST = 2500**

Numero di elementi da estrarre del test set su cui eseguire la fase di test della rete neurale.

- **DEFAULT_RANDOM_SEED = 0**

Valore di default del seed per la generazione random di pesi e bias, utile per la riproducibilità delle fasi di addestramento e test della rete neurale.

- **DEFAULT_DISTRIBUTION_MEAN = 0.0**

Valore di default per la media della distribuzione gaussiana utilizzata per l'inizializzazione dei pesi e bias della rete neurale. Questo valore è stato scelto per bilanciare le attivazioni positive e negative e, di conseguenza, supportare una convergenza più rapida e stabile durante la fase di addestramento.

- **DEFAULT_STANDARD_DEVIATION = 1.0**

Valore di default per la deviazione standard utilizzata per l'inizializzazione dei pesi e bias della rete neurale. Questo valore indica che tutti i pesi. Questo valore è stato scelto per evitare che i pesi iniziali siano troppo grandi o troppo piccoli, riducendo il rischio di saturazione dei neuroni.

- **DEFAULT_INPUT_LAYER_NEURONS = 784**

Valore di default per il numero di neuroni dell'input layer della rete neurale. Questo valore è dato dalla dimensione di una singola immagine del MNIST dataset in input alla rete neurale (e.g. 28x28).

- ***DEFAULT_LAYER_NEURONS*** = [64, *NUMERO_CLASSI*]

Valori di default per il numero di neuroni dell'unico layer interno e dell'output layer della rete neurale. Il numero di neuroni del layer interno è stato scelto per l'alta dimensionalità delle immagini nel MNIST dataset, mentre il numero di neuroni per l'output layer è stato scelto in base al numero di classi in cui sono suddivise le immagini del MNIST dataset.

- ***DEFAULT_EPOCHS*** = 500

Valore di default del numero di epoche per la fase di addestramento della rete neurale. Un'epoca è un'esecuzione completa dell'addestramento (e validazione, se presente) sul training set (e validation set).

- ***DEFAULT_MINI_BATCH_SIZE*** = 1125

Dimensione di default del mini-batch utilizzata durante la fase di addestramento della rete neurale. Questo valore è stato scelto per avere esattamente 10 mini-batch di addestramento.

- ***DEFAULT_EARLY_STOPPING_PATIENCE*** = 15

Valore di default per il numero di epoche dopo il quale fermare l'addestramento se l'errore di validazione non diminuisce di una certa soglia (e.g. se si è raggiunta la convergenza oppure se l'errore di validazione ricomincia a salire).

- ***DEFAULT_EARLY_STOPPING_DELTA*** = 0.1

Valore di default per la soglia che l'errore di validazione deve superare affinché si possa dire che la configurazione attuale di pesi e bias porta un miglioramento significativo nell'addestramento. È stato scelto il valore 0.1 perchè, essendo il MNIST dataset grande e relativamente pulito, il modello dovrebbe migliorare in modo più consistente.

- ***DEFAULT_LEARNING_RATE*** = 0.2

Valore di default per il tasso di apprendimento utilizzato nella fase di addestramento della rete neurale. Indica quanto i pesi debbano essere modificati in risposta all'errore calcolato.

- ***DEFAULT_LEAKY_RELU_ALPHA*** = 0.01

Valore di default per l'iper-parametro *alpha*, usato nella funzione di attivazione *leaky_relu*. Determina la pendenza per i valori negativi dell'input (invece di annullarli come nella ReLU standard). Serve a migliorare la stabilità del modello in presenza di input che possono produrre attivazioni negative.

- ***DEFAULT_BACK_PROPAGATION_MODE*** = True

Valore di default per la scelta dell'algoritmo di retropropagazione da utilizzare durante la fase di addestramento della rete neurale. Il valore **True** indica l'utilizzo dell'algoritmo di *__resilient_back_propagation()*, mentre il valore **False** indica l'utilizzo degli algoritmi di *__back_propagation()* e *__gradient_descent()*.

- **DEFAULT_RPROP_ETA_MINUS = 0.5**

Valore di default per l'iper-parametro `eta_minus`, usato nell'algoritmo di `--resilient_back_propagation()`. È il fattore di riduzione utilizzato per ridurre il passo di aggiornamento dei pesi (step size) quando il gradiente cambia segno. Serve per stabilizzare il processo di ottimizzazione.

- **DEFAULT_RPROP_ETA_PLUS = 1.2**

Valore di default per l'iper-parametro `eta_plus`, usato nell'algoritmo di `--resilient_back_propagation()`. È il fattore di incremento utilizzato per aumentare il passo di aggiornamento dei pesi (step size) quando il gradiente mantiene lo stesso segno. Serve per accelerare la convergenza verso il minimo della funzione di costo.

- **DEFAULT_RPROP_DELTA_MIN = 1e-6**

Valore di default per l'iper-parametro `delta_min`, usato nell'algoritmo di `--resilient_back_propagation()`. Definisce il limite inferiore per il passo di aggiornamento dei pesi (step size). Serve a garantire che l'ottimizzazione con *RPROP* rimanga efficiente, evitando che i passi di aggiornamento diventino troppo piccoli per contribuire significativamente al processo di apprendimento.

- **DEFAULT_RPROP_DELTA_MAX = 50.0**

Valore di default per l'iper-parametro `eta_max`, usato nell'algoritmo di `--resilient_back_propagation()`. Definisce il limite superiore per il passo di aggiornamento dei pesi (step size). Serve a controllare la crescita del passo di aggiornamento nell'algoritmo *RPROP*, bilanciando l'accelerazione della convergenza al fine di mantenere un processo di addestramento del modello efficiente e stabile.

- **DEFAULT_RANDOM_COMBINATIONS = 5**

Valore di default del numero di combinazioni di iper-parametri da testare nell'utilizzare la tecnica del random search.

- **DEFAULT_K_FOLD_VALUE = 10**

Valore di default del numero di fold in cui dividere il training set per il tuning degli iper-parametri tramite utilizzo della tecnica della k-fold cross validation.

- **PIXEL_INTENSITY_LEVELS = 255**

Il numero di livelli di intensità della scala di grigio del singolo pixel nelle immagini grayscale a 8-bit del MNIST dataset.

- **DIMENSIONE_IMMAGINE = 28**

Il numero di pixel su una singola dimensione delle immagini del MNIST dataset. Siccome le immagini di questo dataset sono quadrate, il numero di pixel sulle due dimensioni è lo stesso.

- ***DEBUG_MODE = False***

Consente di attivare (con il valore *True*) la modalità di debug, per stampare in console i valori attuali delle strutture dati coinvolte nella fase di addestramento della rete neurale.

- ***PRINT_DATE_TIME_FORMAT = "%d-%m-%Y, %H:%M:%S"***

Formato di default per la visualizzazione di data e ora nelle stampe in console.

- ***OUTPUT_DATE_TIME_FORMAT = "%Y-%m-%d_%H-%M"***

Formato di default per la visualizzazione di data e ora nelle directory di output.

- ***OUTPUT_DIRECTORY = "../output/"***

Percorso relativo della directory dove salvare tutti i file di output, tra cui le configurazioni di parametri delle reti addestrate e i report (su immagine e su file *.csv*) del tuning degli iper-parametri tramite **Grid Search** e **Random Search**.

- ***PlotTestingMode = Enum('PlotTestingMode', ['NONE', 'REPORT', 'HIGH_CONFIDENCE_CORRECT', 'LOW_CONFIDENCE_CORRECT', 'ALMOST_CORRECT', 'WRONG', 'ALL'])***

Un'enumerazione che raccoglie le modalità di visualizzazione dei risultati di testing.

- **'NONE'** : per stampare i risultati del testing direttamente in console, senza creare e/o salvare alcun grafico.
- **'REPORT'** : per stampare un report generale, cioè un barchart che mostra a quale categoria appartengono le predizioni restituite in output dalla rete neurale.
- **'HIGH_CONFIDENCE_CORRECT'** : oltre a stampare il report generale, fornisce anche i grafici delle sole predizioni corrette ad alta confidenza.
- **'LOW_CONFIDENCE_CORRECT'** : oltre a stampare il report generale, fornisce anche i grafici delle sole predizioni corrette a bassa confidenza.
- **'ALMOST_CORRECT'** : oltre a stampare il report generale, fornisce anche i grafici delle sole predizioni errate che superano la soglia di confidenza sull'etichetta esatta.
- **'WRONG'** : oltre a stampare il report generale, fornisce anche i grafici di tutte le altre predizioni errate.
- **'ALL'** : oltre a stampare il report generale, fornisce anche i grafici di tutte le singole predizioni divisi nelle quattro categorie di cui sopra (in sottocartelle).

- ***PLOT_SEARCH_FIGSIZE = (22, 12)***

Le dimensioni di altezza e larghezza del report del tuning degli iper-parametri tramite Grid Search e Random Search.

- ***PLOT_TESTING_FIGSIZE = (12, 4)***

Le dimensioni di altezza e larghezza del report della singola predizione.

- *PLOT_TESTING_IMAGE_PLOT_INDEX = 0*

L'indice di colonna nel report della singola predizione in cui disegnare l'immagine in scala di grigi della cifra scritta a mano contenuta nel test set.

- *PLOT_TESTING_BAR_CHART_INDEX = 1*

L'indice di colonna nel report della singola predizione in cui disegnare il barchart della distribuzione di probabilità della predizione in output.

- *PLOT_TESTING_CONFIDENCE_THRESHOLD = 0.55*

Indica la soglia di confidenza che la predizione in output deve superare affinché possa essere categorizzata come "risultato corretto ad alta confidenza". Il suo complemento a 1 (e.g. 0.45) è, invece, utilizzato per categorizzare le predizioni in output come "risultati quasi corretti".

2.2.3 Alias di tipo

Infine, nel file *constants.py* vi è la definizione degli alias di tipo, utilizzati per la tecnica dell'hint typing sulle funzioni di attivazione e le funzioni di errore implementate nella libreria.

- *ActivationFunctionType = Callable[[float | np.ndarray, Optional[bool]], float | np.ndarray]*

Semplifica in un alias di tipo la struttura della firma di una funzione di attivazione.

Accetta in input: un valore **float** oppure un **numpy.ndarray**; opzionalmente, un valore **bool**. Restituisce in output: un valore **float** oppure un **numpy.ndarray**.

- *ErrorFunctionType = Callable[[np.ndarray, np.ndarray, Optional[bool]], float | np.ndarray]*

Semplifica in un alias di tipo la struttura della firma di una funzione di errore.

Accetta in input: due **numpy.ndarray**; opzionalmente, un valore **bool**. Restituisce in output: un valore **float** oppure un **numpy.ndarray**.

2.2.4 Funzioni di attivazione

Il file *auxfunc.py* fornisce l'implementazione delle principali funzioni di attivazione utilizzate all'interno delle reti neurali artificiali.

Le funzioni di attivazione sono quelle funzioni che, applicate al risultato di un neurone in una rete neurale, determinano se e in quale misura quel neurone dovrebbe essere attivato, influenzando così l'output finale della rete. Queste funzioni sono fondamentali per introdurre non linearità nei dati e consentono alle reti neurali di apprendere relazioni complesse tra input e output.

Come specificato dall'alias di tipo nella sezione 2.2.3 a pagina 11, queste funzioni ricevono in input il valore su cui applicare la funzione di attivazione, un parametro ***der*** che indica se si vuole calcolare la derivata prima o meno (per la fase di back-propagation) e, infine, restituiscono l'attivazione della funzione richiesta.

- `def leaky_relu(input : float / np.ndarray, der : bool = False) -> float / np.ndarray`

Calcola il valore di attivazione di uno o più neuroni utilizzando un miglioramento della classica ReLU (Rectified Linear Unit), in quanto aggiunge una pendenza costante (specificata dall'iper-parametro **alpha**) per gli input negativi, garantendo una maggiore stabilità nell'apprendimento dei modelli.

- `def sigmoid(input : float / np.ndarray, der : bool = False) -> float / np.ndarray`

Calcola il valore di attivazione di uno o più neuroni utilizzando la funzione logistica per normalizzare l'input nell'intervallo tra 0 e 1. In particolare, per evitare problemi di overflow causati dalla funzione esponenziale, si esegue il clipping dell'input utilizzando la funzione `clip()` della libreria **Numpy**.

- `def tanh(input : float / np.ndarray, der : bool = False) -> float / np.ndarray`

Calcola il valore di attivazione di uno o più neuroni utilizzando la tangente iperbolica per normalizzare l'input nell'intervallo tra **-1** e **1**. A causa dell'utilizzo della funzione esponenziale, anche qui si esegue il clipping dell'input.

- `def identity(input : float / np.ndarray, der : bool = False) -> float / np.ndarray`

Calcola il valore di attivazione di uno o più neuroni utilizzando la funzione identità, mantenendo invariato l'input.

- `def softmax(input : float / np.ndarray, der : bool = False) -> float / np.ndarray`

Converte i valori di attivazione di uno o più neuroni in un vettore di probabilità, dove ogni valore rappresenta la probabilità che l'input appartenga a una determinata classe. Per questo motivo, è utilizzata principalmente per i problemi di classificazione multi-classe.

Rispetto alle altre funzioni di attivazione descritte in questa sezione, la funzione `softmax` è una funzione **globale**, perché dipende dall'input di tutti i neuroni.

In particolare, in questa libreria, non è fornita l'implementazione della derivata prima della `softmax` che, se utilizzata, lancia l'eccezione `NotImplementedError`.

2.2.5 Funzioni di errore

Il file *auxfunc.py* fornisce l'implementazione delle principali funzioni di errore utilizzate all'interno delle reti neurali artificiali.

Le funzioni di errore misurano la differenza (o perdita) tra le previsioni della rete neurale e i valori reali dei dati di training (ground truth), guidando l'ottimizzazione del modello durante il processo di apprendimento.

Come specificato dall'alias di tipo nella sezione 2.2.3 a pagina 11, queste funzioni ricevono in input il parametro *prediction*, cioè il vettore di valori di attivazione di output fornito dalla rete neurale su una determinata coppia del dataset, il parametro *target* nel quale è memorizzata l'etichetta di classificazione di una determinata coppia del dataset e, infine, il parametro *der* che indica se si vuole calcolare la derivata prima o meno (per la fase di back-propagation) rispetto al target.

- `def sum_of_squares(predictions : np.ndarray, target : np.ndarray, der : bool = False) -> float | np.ndarray`

È una funzione di errore tipicamente utilizzata per i problemi di regressione che restituisce la somma dei quadrati degli errori componenti per componente.

- `def cross_entropy(predictions : np.ndarray, target : np.ndarray, der : bool = False) -> float | np.ndarray`

È una funzione di errore tipicamente utilizzata per i problemi di classificazione che restituisce l'entropia incrociata delle due variabili aleatorie discrete relative al target e alla predizione.

- `def cross_entropy_softmax(predictions : np.ndarray, targets : np.ndarray, der : bool = False) -> float | np.ndarray`

È una funzione di errore tipicamente utilizzata per i problemi di classificazione che restituisce l'entropia incrociata delle due variabili aleatorie discrete relative al target e alla predizione, applicando la softmax su quest'ultima.

2.2.6 Funzioni per la gestione del MNIST dataset

Il file *dataset_functions.py* fornisce l'implementazione di alcune funzioni accessorie necessarie per l'estrazione del MNIST training set e del MNIST test set dai file presenti nella directory *dataset*, nonché la preparazione e la gestione dei dati per l'addestramento dei modelli di machine learning.

- `def loadDataset(train_length : int, test_length : int) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]`

Carica il MNIST dataset e ne restituisce gli identificativi, i dati delle immagini delle cifre scritte a mano e le etichette corrispondenti, sia dal training set che dal test set. In particolare, il parametro *train_length* indica il numero di esempi da estrarre dal training set, mentre il parametro *test_length* indica il numero di esempi da estrarre dal test set.

- `def split_dataset(dataset : np.ndarray, labels : np.ndarray, k : int)-> tuple[list[np.ndarray], list[np.ndarray]]`

Divide il dataset e le corrispondenti etichette in k sottoinsiemi. Si utilizza questa funzione per la creazione delle fold da utilizzare nella K-fold cross validation.

- `def convert_to_one_hot(vet : np.ndarray)-> np.ndarray`

Converte un vettore di interi (un'etichetta per ogni esempio del dataset) nella rappresentazione one-hot in forma di matrice (un vettore one-hot per ogni esempio di training).

- `def convert_to_label(vet : np.ndarray)-> str`

Converte la distribuzione di probabilità dell'output della rete neurale nell'etichetta corrispondente alla predizione.

2.2.7 Funzioni per il plotting della fase di addestramento

Il file `plot_functions.py` offre l'implementazione di alcune funzioni accessorie utili per disegnare grafici relativi alla fase di addestramento di una rete neurale. Questi grafici consentono di visualizzare l'andamento delle metriche di performance, come l'errore di validazione e l'accuracy di validazione, durante il processo di addestramento, aiutando a monitorare e diagnosticare il comportamento del modello.

- `def plot_learning_curves(lc_plot : Axes, y_label : str, history_training : list[float], history_validation : list[float])-> None`

Disegna un grafico che confronta le misure calcolate sui dati di addestramento e sui dati di validazione in fase di addestramento.

L'asse delle ascisse mostra il numero di epoch, mentre l'asse delle ordinate mostra l'errore di validazione oppure l'accuracy di validazione. In particolare, l'asse delle ordinate è in comune a tutti i grafici: in questo modo è possibile apprezzare meglio i risultati visualizzati, ma anche riuscire a confrontarli con più facilità.

- `def plot_search_report(out_directory : str, title : str, k_fold_report : list[dict], search_report : pd.Series)-> None`

Disegna una singola immagine contenente: un primo barchart per mostrare tutti i valori di errore medio ottenuti dall'addestramento dei modelli sulle diverse fold; un secondo barchart per mostrare tutti i valori di accuracy media ottenuti dall'addestramento dei modelli sulle diverse fold.

Inoltre, visualizza anche tutti i valori contenuti nel `search_report`, un dizionario contenente i valori degli iper-parametri della combinazione valutata, la media e la deviazione standard dei punteggi di errore e accuracy ottenuti dall'esecuzione della k-fold cross validation.

2.2.8 Funzioni per il plotting della fase di testing

Il file `plot_functions.py` offre anche l'implementazione di altre funzioni per il disegno di grafici relativi alla fase di testing di una rete neurale. I grafici ottenuti da queste funzioni vengono salvati nella directory `output`, e sono utili per visualizzare i risultati del modello e l'accuratezza delle predizioni sui dati di test.

- `def plot_testing(idTest : list[int], Xtest : np.ndarray, Ytest : np.ndarray, probabilities : np.ndarray, out_directory : str) -> None`

Disegna un'immagine divisa in 2 colonne: la prima colonna contiene la rappresentazione in scala di grigi dell'immagine 28x28 delle cifre del MNIST test set e, come titolo, la relativa etichetta; la seconda colonna contiene la rappresentazione della predizione della rete neurale sull'esempio di testing corrispondente, tramite un bar chart.

- `def plot_predictions(idTest : np.ndarray, Xtest : np.ndarray, Ytest : np.ndarray, probabilities : np.ndarray, out_directory : str, plot_mode : constants.PlotTestingMode = constants.PlotTestingMode.REPORT) -> None`

Disegna un barchart che mostra a quale categoria appartengono le predizioni restituite in output dalla rete neurale. In particolare, i parametri `Xtest` e `Ytest` sono, rispettivamente, gli esempi e le etichette di testing, mentre `probabilities` è la matrice contenente la distribuzione di probabilità delle predizioni della rete neurale. Invece, il parametro `plot_mode` serve a distinguere per quali esempi in input e quali predizioni in output si devono disegnare i grafici completi (si veda la documentazione della costante `PlotTestingMode` a pagina 10).

2.2.9 Altre funzioni

Durante lo sviluppo di questa libreria, il team di sviluppo ha avuto la necessità di implementare anche altre funzioni, descritte di seguito.

- `def print_progress_bar(iteration : int, total : int, prefix : str = '', suffix : str = '', length : int = 50, fill : str = '#')-> None`

Stampa in console una barra di caricamento che si aggiorna ad ogni chiamata di un loop, mostrando il numero dell'iterazione corrente rispetto al totale delle iterazioni.

- `def compute_batches(length : int, batch_size : int = constants.DEFAULT_MINI_BATCH_SIZE)-> list[tuple[int, int]]`

Calcola gli indici di inizio e fine dei mini-batch di un dataset sulle quali eseguire la fase di addestramento della rete neurale.

Il parametro `length` indica la lunghezza della porzione di dataset da esaminare.

Il parametro `batch_size`, invece, indica la lunghezza desiderata del singolo mini-batch. Si veda la descrizione dell'attributo `batch_size` nella sezione 2.5.1 a pagina 24 per un approfondimento sui valori che può assumere questo parametro.

- `def get_random_color()-> str`

Genera un numero intero di 24-bit corrispondenti alle 3 componenti di colore RGB, rispettivamente di 8 bit, e ne restituisce la rappresentazione in stringa del codice esadecimale.

2.3 La classe Layer

Il file *artificial_layer.py* nella directory *code* contiene l'implementazione di un layer di una rete neurale artificiale feed-forward fully-connected (e.g. Multilayer Perceptron) tramite paradigma di programmazione a oggetti.

La classe **Layer** è il cuore della rete neurale. Rappresenta un singolo livello di neuroni. Gestisce in modo indipendente rispetto al resto della rete neurale i valori dei pesi / bias e le operazioni di attivazione per il livello specifico, facilitando il monitoraggio della propagazione dei dati in avanti (e.g. forward-propagation) e all'indietro (e.g. back-propagation) attraverso la rete neurale.

2.3.1 Attributi di classe

Gli attributi della classe **Layer** rappresentano le proprietà essenziali di uno strato di neuroni in una rete neurale artificiale. Questi includono:

- ***layer_size*** : *int*

È la dimensione del layer, cioè il numero di neuroni di cui è composto. Per quest'attributo di classe è stato implementato solo il metodo getter, in modo tale da lasciare invariato il suo valore dopo l'inizializzazione da parte del costruttore.

- ***neuron_size*** : *int*

È la dimensione di un neurone del layer, cioè il numero di connessioni provenienti dal livello precedente. Anche per quest'attributo di classe si è scelto di non implementare il metodo setter, per le stesse ragioni di cui sopra.

- ***inputs*** : *numpy.ndarray*

È la matrice di valori in input al layer. Ha un numero di colonne pari al numero di neuroni del layer precedente ed un numero di righe pari al numero di esempi in input alla rete neurale. Il metodo setter relativo a questo attributo di classe, prima dell'assegnamento vero e proprio, controlla che il nuovo valore sia di tipo *numpy.ndarray* e che la sua forma sia compatibile con la struttura del layer in esame.

- ***weights*** : *numpy.ndarray*

È la matrice dei pesi del layer. In essa sono contenuti tutti i pesi di tutti i neuroni del layer. La prima dimensione indica il numero di neuroni del layer, mentre la seconda dimensione indica il numero di pesi all'interno del singolo neurone.

Proprio come per l'attributo *inputs*, anche nel setter di quest'attributo, prima di assegnare il nuovo valore, si controlla che questo sia compatibile con il layer in esame.

- **biases** : `numpy.ndarray`

È il vettore colonna dei bias del layer. In esso sono contenuti tutti i bias di tutti i neuroni del layer. La prima dimensione indica il numero di neuroni del layer, mentre la seconda dimensione è **1** perché il bias è uno scalare.

Anche in questo caso, nel setter di quest'attributo, prima di assegnare il nuovo valore, si controlla che questo sia compatibile con il layer in esame.

- **act_fun** : `constants.ActivationFunctionType`

È la funzione di attivazione di tutti i neuroni del layer, con dominio e codominio a valori reali. Per una descrizione più accurata del ruolo che riveste la funzione di attivazione nella rete neurale, si veda la sezione 2.2.4 a pagina 11.

Per una descrizione più accurata dell'alias di tipo utilizzato, si veda la sezione 2.2.3 a pagina 11.

2.3.2 Costruttore

Il costruttore della classe **Layer**:

1. Inizializza gli attributi `layer_size` e `neuron_size`, controllando che i valori passati per parametro siano strettamente maggiori di **0**.
2. Inizializza il puntatore a funzione `act_fun`.
3. Inizializza la matrice degli input con una matrice nulla della giusta dimensione, compatibile con la struttura del layer.
4. In base al valore del parametro `random_init`, inizializza la matrice dei pesi ed il vettore dei bias in modo casuale con o senza seed fissato, rispettivamente. Il motivo di questa scelta implementativa è quella di favorire la riproducibilità delle fasi di addestramento e test della rete neurale.

2.3.3 Metodi pubblici

I metodi pubblici della classe **Layer** sono quelle funzioni della classe che consentono agli utenti della libreria di interfacciarsi con i singoli layer di una rete neurale artificiale.

Per una miglior comprensione delle seguenti funzionalità, essenziali per il funzionamento della rete neurale, la figura 2.3 a pagina 19 mostra la struttura di un neurone della rete neurale artificiale e tutti i valori coinvolti nel calcolo degli output e delle attivazioni.

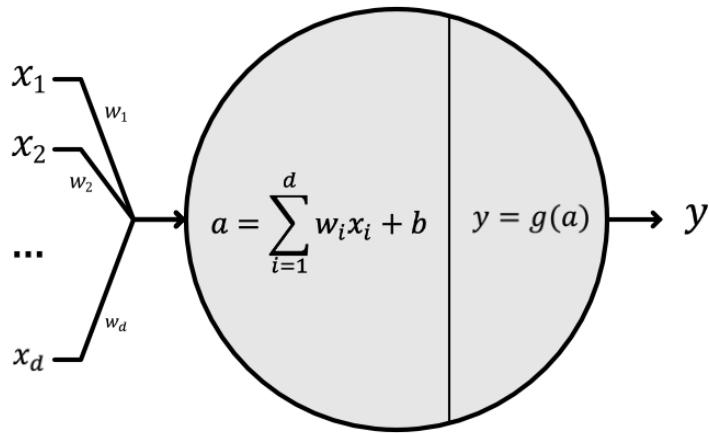


Figura 2.3: Struttura di un neurone della rete neurale artificiale

- **`def output(self)-> np.ndarray`**

Calcola gli output (o input pesati) dei neuroni del layer corrente come una combinazione lineare dei valori in input al neurone e i pesi lungo le connessioni aggiungendo, infine, il bias.

- **`def activate(self, train : bool = False)-> np.ndarray | tuple[np.ndarray, np.ndarray]`**

Calcola i valori di attivazione dei neuroni del layer corrente, applicando la funzione di attivazione del layer corrente agli input pesati ottenuti dall'esecuzione della funzione `output()`.

In particolare, questa funzione, se `train=False`, restituisce un `numpy.ndarray` contenente tutti e soli i valori di attivazione dei neuroni. Al contrario, se `train=True`, restituisce sia un `numpy.ndarray` per gli output intermedi prima dell'applicazione della funzione di attivazione sia un altro per i valori di attivazione.

- **`def __repr__(self)-> str`**

Restituisce una rappresentazione dettagliata del contenuto di un oggetto della classe Layer. Viene principalmente utilizzata per stampare in console i valori delle proprietà del layer con una formattazione più precisa. La seguente figura ne mostra un esempio:

```
Layer(
    size = 32,
    act_fun = <function leaky_relu at 0x1071183a0>,
    inputs_size = (1, 784)
    weights_shape = (32, 784),
    biases_shape = (32, 1)
)
```

Figura 2.4: Visualizzazione del contenuto di un oggetto della classe `Layer`

2.4 La classe TrainingReport

Il file *training_report.py* nella directory *code* fornisce l'implementazione della classe *TrainingReport*, il cui scopo è quello di memorizzare tutte le misure di valutazione riguardanti l'addestramento di una rete neurale feed-forward fully-connected.

Il metodo *train()*, la cui descrizione è disponibile nella sezione 2.6.4 a pagina 33, genera un oggetto della classe *TrainingReport* per ogni epoca della fase di addestramento, per valutare in modo efficace l'evoluzione del modello.

Durante la fase di addestramento, l'oggetto della classe *TrainingReport* presente tra gli attributi della classe *NeuralNetwork* viene aggiornato ad ogni epoca per memorizzare le prestazioni ottenute dalla configurazione di parametri che riporta il minor errore di validazione. Per questo motivo, al termine del training, tale oggetto fornirà un report preciso della miglior epoca. Per semplicità, si omette la descrizione del costruttore di classe.

2.4.1 Attributi di classe

Gli attributi della classe *TrainingReport* rappresentano le misure di valutazione e i dati raccolti durante la fase di addestramento e di validazione di una rete neurale feed-forward fully-connected, fornendo un riepilogo dettagliato delle prestazioni del modello.

- *num_epochs* : *int*

Il numero di epoche impiegate dalla fase di addestramento. Questo valore è utile per tracciare l'evoluzione del training, soprattutto nel plotting dei grafici relativi alla fase di addestramento.

- *elapsed_time* : *float*

Il tempo totale impiegato per completare il numero di epoche indicato dall'attributo di classe *num_epochs*, utile per valutare l'efficienza del processo di addestramento.

-
- *training_examples* : *int*

Il numero di esempi del training set utilizzati per la fase di addestramento. Questo valore risulta utile per confrontare le prestazioni ottenute da modelli allenati su una quantità di esempi di training diversa.

- *training_error* : *float*

L'errore di addestramento ottenuto al termine della fase di addestramento dopo un numero di epoche indicate dall'attributo di classe *num_epochs*. Questo valore viene rappresentato graficamente nei plot delle curve di apprendimento.

- ***training_accuracy*** : *float*

L'accuracy di addestramento ottenuta al termine della fase di addestramento dopo un numero di epoche indicate dall'attributo di classe *num_epochs*. Questo valore viene rappresentato graficamente nei plot delle curve di apprendimento.

- ***validation_examples*** : *int*

Il numero di esempi del validation set utilizzati per la validazione del modello. Proprio come per l'attributo *training_examples*, anche questo valore è utile per confrontare le prestazioni ottenute da modelli diversi.

- ***validation_error*** : *float*

L'errore di validazione ottenuto al termine della validazione del modello dopo un numero di epoche indicate dall'attributo di classe *num_epochs*. Questo valore viene rappresentato graficamente nei plot delle curve di apprendimento.

- ***validation_accuracy*** : *float*

L'accuracy di validazione ottenuta al termine della validazione del modello dopo un numero di epoche indicate dall'attributo di classe *num_epochs*. Questo valore viene rappresentato graficamente nei plot delle curve di apprendimento.

2.4.2 Metodi pubblici

I metodi pubblici della classe *TrainingReport* sono quelle funzioni che consentono agli utenti della libreria di accedere e interagire con le misure di valutazione raccolte durante l'addestramento di una rete neurale feed-forward fully-connected.

- **`def compute_error(self, predictions : np.ndarray, targets : np.ndarray, err_fun : constants.ErrorFunctionType) -> float`**

Calcola l'errore della rete neurale come media delle distanze tra la matrice *predictions*, contenente tutte le previsioni della rete, e la matrice *targets*, contenente le etichette vere corrispondenti alle previsioni (ground truth).

- **`def compute_accuracy(self, predictions : np.ndarray, targets : np.ndarray) -> float`**

Calcola l'accuratezza della rete neurale come il rapporto la matrice *predictions*, contenente tutte le previsioni della rete, e la matrice *targets*, contenente le etichette vere corrispondenti alle previsioni (ground truth).

- `def update(self, value)-> None`

Aggiorna i valori del report corrente con i valori provenienti da un altro oggetto della classe `TrainingReport`. Il metodo controlla che il parametro `value` passato in input sia di tipo `TrainingReport`, quindi procede con gli assegnamenti ai valori degli attributi.

- `def __repr__(self)-> str`

Restituisce una rappresentazione dettagliata del contenuto di un oggetto della classe `TrainingReport`. La seguente figura mostra un esempio di stampa in console:

```
TrainingReport(  
    num_epochs = 1,  
    elapsed_time = 0.106 secondi,  
    training_examples = 90,  
    training_error = 69.91385,  
    training_accuracy = 13.33%,  
    validation_examples = 10,  
    validation_error = 57.74627,  
    validation_accuracy = 20.00%  
)
```

Figura 2.5: Visualizzazione del contenuto di un oggetto della classe `TrainingReport`

2.5 La classe TrainingParams

Il file `training_params.py` nella directory `code` comprende l'implementazione della classe **TrainingParams**, il cui scopo è quello di memorizzare una parte dei valori degli iper-parametri riguardanti l'addestramento di una rete neurale feed-forward fully-connected (perchè altri sono intrinsecamente memorizzati nell'implementazione della classe **NeuralNetwork**). Memorizzando esplicitamente questi iper-parametri, la classe **TrainingParams** facilita la riproducibilità degli esperimenti e la documentazione delle configurazioni di addestramento.

La classe non fornisce alcun metodo pubblico, in quanto gli attributi di classe possono essere manipolati solo attraverso il costruttore di classe che, per semplicità, non viene descritto in questa documentazione. L'unico metodo a cui possono accedere gli utenti della libreria è `def __repr__(self) -> str`, il quale restituisce una rappresentazione dettagliata del contenuto di un oggetto della classe `TrainingParams`. La seguente figura mostra un esempio di stampa in console:

```
TrainingParams(  
    batch_size = 200,  
    epochs = 2,  
    learning_rate = 0.20,  
    rprop = True,  
    eta_minus = 0.50000,  
    eta_plus = 1.30000,  
    delta_min = 0.00000,  
    delta_max = 50.00000,  
    es_patience = 15,  
    es_delta = 0.10000  
)
```

Figura 2.6: Visualizzazione del contenuto di un oggetto della classe **TrainingParams**

2.5.1 Attributi di classe

Gli attributi della classe ***TrainingParams*** raccolgono alcuni valori degli iper-parametri specifici per il processo di addestramento di una rete neurale. La scelta di separare i seguenti iper-parametri in una classe indipendente da quella che implementa la rete neurale, è data dal voler fornire agli utenti della libreria maggior flessibilità nella scelta degli iper-parametri, consentendo loro di sperimentare più configurazioni e trovare e/o salvare quelle ottimali.

- ***epoch*** : *int*

Il massimo numero di iterazioni per cui il modello deve essere addestrato.

Siccome in questa libreria è stata implementata la tecnica di early stopping, che consente di terminare il processo di addestramento dopo un determinato numero di epoche in cui l'errore di validazione si è stabilizzato o inizia a peggiorare, può succedere che il valore di questo attributo non corrisponda effettivamente al numero di epoche per cui il modello è stato addestrato.

- ***learning_rate*** : *float*

Il tasso di apprendimento utilizzato nella fase di addestramento per l'aggiornamento dei pesi / bias.

-
- ***batch_size*** : *int*

Il numero di esempi di addestramento contenuti in un singolo mini-batch.

Se il valore di questo attributo è esattamente uguale al valore dell'attributo ***training_examples*** in **TrainingReport**, allora si ottiene un unico batch (e.g. batch learning); se è minore di ***training_examples***, allora si ottiene un maggior numero di batch (e.g. mini-batch learning); infine, se è 1, allora si ottiene il massimo numero di batch (e.g. online learning).

- ***rprop*** : *bool*

Un flag che indica quale algoritmo di retro-propagazione utilizzare nella fase di addestramento. Si rimanda alla sezione 2.2.2 a pagina 8 per una descrizione precisa dei valori che può assumere questo attributo.

- ***eta_minus*** : *float*

Il fattore di riduzione utilizzato per ridurre il passo di aggiornamento dei pesi (step size) quando il gradiente cambia segno. Serve per stabilizzare il processo di ottimizzazione.

Tipicamente, il valore per ***eta_minus*** è compreso tra 0.5 e 0.9. Valori piccoli riducono considerevolmente il passo di aggiornamento migliorando la stabilità, ma potenzialmente rallentando la convergenza. Al contrario, valori più grandi consentono una convergenza più rapida ma con il rischio di instabilità.

- ***eta_plus*** : *float*

Il fattore di incremento utilizzato per aumentare il passo di aggiornamento dei pesi (step size) quando il gradiente conserva lo stesso segno. Serve per accelerare la convergenza verso il minimo della funzione di costo.

Nella maggior parte delle applicazioni, il valore per ***eta_plus*** è compreso tra 1.2 e 1.5. Si possono utilizzare valori più o meno piccoli in base alle necessità della propria applicazione, e i vantaggi / svantaggi di questa scelta sono gli stessi di cui sopra.

- ***delta_min*** : *float*

Il limite inferiore che il passo di aggiornamento dei pesi (step size) può raggiungere, assicurando che gli aggiornamenti non diventino poco significativi.

- ***delta_max*** : *float*

Il limite superiore che il passo di aggiornamento dei pesi (step size) può raggiungere durante l'addestramento, utilizzato per limitare l'entità le oscillazioni.

- ***es_patience*** : *int*

Il numero di epoche dopo il quale fermare il processo di addestramento se l'errore di validazione non è diminuito di una certa soglia (e.g. si è raggiunta la convergenza oppure l'errore di validazione ricomincia a salire).

- ***es_delta*** : *float*

La soglia che l'errore di validazione deve superare rispetto alla miglior epoca durante la fase di addestramento per capire se ci sono stati miglioramenti significativi. Il valore di default è esplicitato alla sezione 2.2.2 a pagina 8

2.6 La classe NeuralNetwork

Il file *artificial_neural_network.py* nella directory *code* contiene l'implementazione di una rete neurale artificiale feed-forward fully-connected (e.g. Multilayer Perceptron) tramite paradigma di programmazione a oggetti.

In particolare, come si evince dalla figura 2.1 a pagina 4, la classe che implementa la rete neurale (**NeuralNetwork**) può essere composta di uno o più strati (**Layer**).

È altamente configurabile, in quanto offre la possibilità di scegliere i valori di diversi iper-parametri specifici della rete (e.g. funzioni di attivazione, funzione di errore, numero di layer, ...), ma anche altri iper-parametri specifici della fase di addestramento (e.g. learning rate, numero di epoche, batch size, ...) tramite l'oggetto della classe **TrainingParams**. I valori di default sono specificati nella sezione 2.2.2 a pagina 7.

2.6.1 Attributi di classe

Gli attributi della classe **NeuralNetwork** includono i parametri necessari per l'implementazione e il funzionamento di una rete neurale. Permettono di strutturare in modo organizzato e accessibile le componenti fondamentali di una rete neurale, facilitando lo sviluppo e la gestione del modello.

Tra questi, troviamo:

- **depth** : *int*

È la profondità della rete, cioè il numero totale di **Layer**. A questo attributo non è associato il metodo setter in quanto il suo valore viene calcolato dinamicamente dal costruttore, in base al numero di layer interni richiesti.

- **layers** : *list[Layer]*

Una lista di oggetti **Layer** che costituiscono i vari strati della rete neurale, determinando la struttura e la profondità della rete.

A questo attributo non è associato il metodo setter, siccome è possibile modificare la struttura della rete solo tramite l'inizializzazione del costruttore.

-
- **input_size** : *int*

È la dimensione del vettore di caratteristiche di un singolo esempio di input per la rete neurale. Ad esempio, nel MNIST dataset, le immagini sono composte di 784 pixel in totale: per questo motivo, una possibile applicazione richiede che l'input layer abbia 784 neuroni di input.

- *inputs* : *numpy.ndarray*

È la matrice di esempi in input alla rete neurale sulla quale, ad esempio, si vuole eseguire l’addestramento della rete o il testing andando ad elaborarne le predizioni in output.

Il metodo setter relativo a questo attributo di classe, prima dell’assegnamento vero e proprio, controlla che il nuovo valore sia di tipo *numpy.ndarray* e che la sua forma sia compatibile con la struttura dell’input layer della rete.

- *weights* : *numpy.ndarray*

È il vettore serializzato di tutti i pesi di tutti i neuroni della rete neurale. La sua dimensione è pari al numero totale di connessioni tra neuroni della rete.

Il metodo setter si occupa di aggiornare solo localmente il contenuto di questo vettore. Per riportare la modifica dei pesi in tutta la struttura della rete neurale, bisogna utilizzare il metodo *__scatter_weights()*.

- *biases* : *numpy.ndarray*

È il vettore serializzato di tutti i bias di tutti i neuroni della rete neurale. La sua dimensione è pari al numero totale di neuroni in tutta la rete.

Il metodo setter si occupa di aggiornare solo localmente il contenuto di questo vettore. Per riportare la modifica dei pesi in tutta la struttura della rete neurale, bisogna utilizzare il metodo *__scatter_weights()*.

- *err_fun* : *constants.ErrorFunctionType*

È la funzione di errore utilizzata per verificare la qualità della rete neurale, calcolando l’errore tra le previsioni del modello e i valori reali degli esempi forniti in input.

- *training_report* : *TrainingReport*

È un’istanza della classe **TrainingReport**, contenente le metriche di valutazione della fase di addestramento. Si veda la sezione 2.4 a pagina 20 per maggiori informazioni riguardo questa classe.

- *training_params* : *TrainingParams*

È un’istanza della classe **TrainingParams**, contenente i valori degli iper-parametri specifici della fase di addestramento. Si veda la sezione 2.5 a pagina 23 per maggiori informazioni riguardo questa classe.

2.6.2 Costruttore

Il costruttore della classe Layer:

1. Inizializza la dimensione dell'input layer della rete neurale, controllando che quest'ultima sia maggiore di **0**.
2. Inizializza le dimensioni dei layer interni e ne assegna le rispettive funzioni di attivazione, controllando che il numero di layer interni ed il numero di funzioni di attivazione passati per parametro sia compatibile e anche maggiore di **2**.
3. Inizializza i vettori serializzati dei pesi e dei bias recuperando dai layer interni della rete neurale tali valori mediante l'utilizzo del metodo privato `__gather_weights()`.
4. Calcola la profondità della rete neurale in base al numero totale di layer, sia quelli interni che quello di output.
5. Inizializza gli altri attributi di classe, tra cui: `err_fun`, per la funzione di errore; `training_report`, per le metriche di valutazione della fase di addestramento; `training_params`, per i valori degli iper-parametri specifici del processo di training.

2.6.3 Metodi privati

I metodi privati della classe **NeuralNetwork** sono funzioni interne alla classe che non sono accessibili direttamente dagli utenti esterni, ma sono utilizzate per eseguire operazioni ausiliarie, gestire la logica interna e / o supportare i metodi pubblici.

Isolare la logica interna dalle funzionalità accessibili dagli utenti della libreria non solo permette di nascondere la maggior parte dei dettagli implementativi, ma è anche un'ottima scelta per facilitare la manutenzione ed il debugging della classe, dato che questi metodi possono essere testati e migliorati separatamente.

- **def __gather_weights(self) -> tuple[np.ndarray, np.ndarray]**

Serializza le matrici dei pesi e dei bias di tutti i layer della rete neurale in due vettori separati della giusta dimensione, cioè un vettore di dimensione pari al numero di connessioni della rete (per i pesi) ed un vettore di dimensione pari alla somma delle dimensioni dei layer della rete (per i bias). In altre parole, aggiorna le property `weights` e `biases` della classe **NeuralNetwork**.

- **def __scatter_weights(self) -> None**

Distribuisce i vettori serializzati dei pesi e dei bias memorizzati nella classe **NeuralNetwork** in tutti i layer della rete neurale. Secondo l'implementazione dei vettori dei pesi e dei bias, il solo assegnamento agli attributi di classe `weights` e `biases` non basta, ma bisogna utilizzare questa funzione per completare correttamente l'aggiornamento dei pesi / bias.

- `def __forward_propagation(self, x : np.ndarray, train : bool = False) -> np.ndarray | tuple[list[np.ndarray], list[np.ndarray]]`

Calcola l'output complessivo della rete neurale, propagando il vettore x di dati in input attraverso le connessioni di input dell'input layer, attraverso i calcoli intermedi degli hidden layers e, infine, attraverso l'ultimo strato dell'output layer.

In particolare: se `train=False`, allora l'esecuzione del metodo non è per la fase di training, quindi viene restituito un `numpy.ndarray` contenente i soli valori di attivazione dell'output layer; invece, se `train = True`, allora l'esecuzione del metodo riguarda il processo di addestramento, quindi viene restituita una prima lista contenente i `numpy.ndarray` degli input pesati di ogni layer della rete ed una seconda lista contenente i `numpy.ndarray` dei valori di attivazione di ogni layer della rete.

- `def __delta_output_layer(self, output_layer_outputs : np.ndarray, output_layer_activations : np.ndarray, targets : np.ndarray) -> np.ndarray`

Calcola la matrice le cui componenti sono le derivate prime parziali, ovvero il gradiente, della funzione di costo della rete neurale rispetto agli input pesati dell'output layer su tutti gli esempi di training. Il numero di righe di questa matrice corrisponde al numero di esempi di training, mentre il numero di colonne corrisponde alla dimensione dell'output layer.

È l'implementazione della seguente equazione proposta nel libro "Neural Networks and Deep Learning" [2] di Michael Nielsen.

$$\delta^L = \Delta_a C \odot \sigma'(z^L) \quad (2.1)$$

- `def __delta_layer(self, network_outputs : list[np.ndarray], network_activations : list[np.ndarray], delta_output_layer : np.ndarray, layer_index: int) -> np.ndarray`

Calcola il vettore le cui componenti sono le derivate prime parziali della funzione di costo della rete neurale rispetto agli input pesati di un layer qualsiasi della rete. Il numero di righe corrisponde al numero di esempi di training, mentre il numero di colonne corrisponde alla dimensione del layer scelto.

Internamente, se il layer in esame è proprio l'output layer, allora questa funzione effettua una chiamata alla funzione `__delta_output_layer()`.

È l'implementazione della seguente equazione di Nielsen [2]:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (2.2)$$

- `def __back_propagation(self, network_outputs : list[np.ndarray], network_activations : list[np.ndarray], training_labels : np.ndarray)-> tuple[np.ndarray, np.ndarray]`

Aggiusta i valori dei pesi / bias della rete neurale al fine di diminuire il valore della funzione di costo rispetto agli esempi di training e le corrispondenti etichette in input.

Calcola il gradiente della funzione di costo rispetto a tutti i pesi della rete utilizzando le quattro equazioni fondamentali di Nielsen [2] (mostrate in questa sezione).

Tali equazioni sono una diretta conseguenza della regola della catena del calcolo multivariabile (essendo la rete fully-connected, l'aggiustamento dei pesi di un layer provoca una catena di effetti in tutti i layer successivi).

In particolare, la sua implementazione prevede i seguenti step:

1. *Si calcola il delta dell'errore sull'output layer.*
2. *Per ogni layer della rete neurale*, si applica la retro-propagazione del delta dell'errore del layer corrente ai layer precedenti.
3. *Per ogni layer della rete neurale*, si calcola il gradiente della funzione di costo rispetto ai bias della rete neurale, cioè come cambia la funzione di costo rispetto ai bias di tutti i neuroni della rete neurale.

È l'implementazione della seguente equazione di Nielsen [2]:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (2.3)$$

4. *Per ogni layer della rete neurale*, si calcola il gradiente della funzione di costo rispetto ai pesi della rete neurale, cioè come cambia la funzione di costo rispetto al peso di tutte le connessioni tra due neuroni di due layer adiacenti della rete neurale.

Si applica la regola della catena come segue:

$$\frac{\partial C}{\partial w_{ij}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial C}{\partial a_j^{(l)}} \quad (2.4)$$

In questa equazione: C è il valore della funzione di costo; $a_j^{(l)}$ è il valore di attivazione del neurone j nel layer l ; $z_j^{(l)}$ è l'input pesato del neurone j nel layer l ; $w_{ij}^{(l)}$ è il valore del peso sulla connessione tra il neurone i nel layer $l - 1$ con il neurone j nel layer l .

In termini di *delta*, Nielsen [2] propone la seguente equazione:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (2.5)$$

- `def __gradient_descent(self, gradient_weights : np.ndarray, gradient_biases : np.ndarray, learning_rate : float) -> None`

Calcola l'aggiornamento dei pesi e bias della rete neurale utilizzando l'algoritmo di discesa del gradiente, il quale sottrae ad ogni componente del vettore serializzato dei pesi e ad ogni componente del vettore serializzato dei bias un valore dato dal prodotto del tasso di apprendimento per il gradiente del peso o del bias, rispettivamente.

Per riportare l'aggiornamento dei pesi / bias in tutti i layer della rete, si utilizza il metodo `__scatter_weights()`.

- `def __rprop_delta_layer(self, prod_gradients : np.ndarray, prev_delta_layer : np.ndarray) -> np.ndarray`

Calcola e restituisce un array contenente gli *step size* relativi ad ogni peso e bias della rete neurale. Con il termine *step size* si indica la quantità con cui aggiornare ogni peso / bias.

In particolare, con la Rprop, gli *step size* sono indipendenti dai valori assoluti delle derivate parziali della funzione di costo, ma dipendono solo dal loro segno.

Internamente, questo metodo utilizza alcuni valori degli iper-parametri specificati nell'oggetto della classe **TrainingParams** tra cui, ad esempio, gli iper-parametri *eta minus* ed *eta plus*.

- `def __resilient_back_propagation(self, network_outputs : list[np.ndarray], network_activations : list[np.ndarray], training_labels, prev_gw, prev_gb, prev_dlw, prev_dlb) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]`

Per il corretto funzionamento di questo metodo, i parametri *training_labels*, *prev_gw*, *prev_gb*, *prev_dlw*, *prev_dlb* dovrebbero essere tutti di tipo **numpy.ndarray**.

Questo metodo implementa un algoritmo di apprendimento Rprop (e.g. resilient back-propagation). Sono algoritmi iterativi che aggiornano i parametri di una rete neurale (e.g. pesi e bias) ottimizzando la funzione di costo tramite calcolo del suo gradiente rispetto ai pesi della rete neurale stessa, considerando solo il segno di questi valori e non il loro valore assoluto. Cioè, ad ogni iterazione, ciascun peso della rete viene aumentato o diminuito se il segno della derivata parziale è positivo o negativo, rispettivamente.

Inoltre, sono algoritmi in grado di aggiustare la *step size* dinamicamente in base a come cambia il segno della derivata parziale della funzione di costo su iterazioni consecutive: se il segno cambia, si diminuisce il learning rate (perché è stato superato il punto di minimo); altrimenti, lo si incrementa (per scendere più velocemente verso il punto di minimo).

Da questa descrizione si evince che esistono molti algoritmi per la resilient back-propagation. Il team di sviluppo ha implementato in questa libreria l'algoritmo **RPROP+ con weight-backtracking** proposta da Christian Igel nel paper "Empirical evaluation of the improved Rprop learning algorithms" [3]. La figura a pagina 32 mostra un estratto del pseudo-codice proposto da Igel.

```

for each  $w_{ij}$  do {
    if  $\frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} > 0$  then {
         $\Delta_{ij}^{(t)} := \min \left( \Delta_{ij}^{(t-1)} \cdot \eta^+, \Delta_{\max} \right)$ 
         $\Delta w_{ij}^{(t)} := -\text{sign} \left( \frac{\partial E}{\partial w_{ij}}^{(t)} \right) \cdot \Delta_{ij}^{(t)}$ 
         $w_{ij}^{(t+1)} := w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$ 
    }
    elseif  $\frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} < 0$  then {
         $\Delta_{ij}^{(t)} := \max \left( \Delta_{ij}^{(t-1)} \cdot \eta^-, \Delta_{\min} \right)$ 
         $w_{ij}^{(t+1)} := w_{ij}^{(t)} - \Delta w_{ij}^{(t-1)}$ 
         $\frac{\partial E}{\partial w_{ij}}^{(t)} := 0$ 
    }
    elseif  $\frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} = 0$  then {
         $\Delta w_{ij}^{(t)} := -\text{sign} \left( \frac{\partial E}{\partial w_{ij}}^{(t)} \right) \cdot \Delta_{ij}^{(t)}$ 
         $w_{ij}^{(t+1)} := w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$ 
    }
}

```

Figura 2.7: Pseudo-codice di una singola iterazione dell'algoritmo **RPROP+** con weight-backtracking

2.6.4 Metodi pubblici

I metodi pubblici della classe **NeuralNetwork** sono quelle funzioni accessibili agli utenti della classe che consentono loro di interagire con la rete neurale, gestire l'addestramento, effettuare previsioni e valutare le prestazioni del modello attraverso la visualizzazione del contenuto dell'attributo **training_report** o attraverso il disegno di specifici grafici relativi al processo di addestramento o alla classificazione delle previsioni.

- `def train(self, training_data : np.ndarray, training_labels : np.ndarray, validation_data : np.ndarray = None, validation_labels : np.ndarray = None) -> list[TrainingReport]`

Avvia il processo di addestramento della rete neurale utilizzando i dati di addestramento e di validazione in input secondo gli iper-parametri specificati.

Il processo di addestramento ripete le fasi di forward propagation, backpropagation (con calcolo della funzione di costo) e il conseguente aggiornamento dei pesi per un numero limitato di iterazioni (o epoche).

Internamente, questo metodo verifica se la matrice **training_data** contenente una riga per ogni esempio di addestramento è compatibile con la matrice **training_labels** contenente una riga per ogni etichetta di un esempio di addestramento. Lo stesso vale anche per i dati di validazione, se presenti.

Non appena termina un'epoca di addestramento, dopo aver calcolato le misure di errore e di accuracy sia per la fase di addestramento vera e propria che per la fase di validazione, genera un oggetto della classe **TrainingReport** per raccogliere tutte queste metriche di valutazione.

Quindi, confrontando l'errore di validazione dell'epoca corrente con l'errore di validazione della miglior epoca rispetto al parametro **es_delta**, verifica se ci sono stati miglioramenti significativi nell'aggiornamento dei pesi / bias. Di conseguenza, se il contatore **es_counter** raggiunge il numero di epoche indicato da **es_patience** si interrompe forzatamente il processo di addestramento (early stopping).

Infine, al completamento dell'intera fase di training, ritorna alla configurazione di parametri indicata dalla migliore epoca (e.g. quella che ha ottenuto il minor errore di validazione).

- `def predict(self, idTest : np.ndarray, Xtest : np.ndarray) -> list[str]`

Calcola le predizioni per la matrice di esempi di testing da elaborare, in base alla configurazione attuale di pesi e bias della rete neurale, utilizzando la funzione **softmax**. Restituisce in output la lista di etichette calcolate dalla rete neurale per ogni esempio di testing in input.

- `def test(self, out_directory : str, idTest : np.ndarray, Xtest : np.ndarray, Ytest : np.ndarray, plot_mode : constants.PlotTestingMode = constants.PlotTestingMode.REPORT)-> None`

Proprio come per il metodo `predict()`, calcola le predizioni per la matrice di esempi di testing da elaborare, in base alla configurazione attuale di pesi e bias della rete neurale, utilizzando la funzione `softmax`.

Quindi, confronta le etichette della ground truth contenute nella matrice `Ytest` con le etichette delle predizioni, mostrando i risultati in un grafico dedicato. Per maggiori informazioni, si veda la documentazione di `plot_predictions()` nella sezione 2.2.8 a pagina 15.

- `def save_network_to_file(self, out_directory : str, out_name : str = "net.pkl")-> None`

Salva tutti gli iper-parametri e parametri della rete neurale in un file binario alla directory specificata, consentendo di conservare e caricare il modello in futuro.

- `def load_network_from_file(filename : str)`

Carica la configurazione completa di iper-parametri e parametri della rete neurale direttamente da un file alla directory specificata, consentendo di ripristinare un modello precedentemente allenato e/o testato.

- `def __repr__(self) -> str`

Restituisce una rappresentazione dettagliata del contenuto di un oggetto della classe **NeuralNetwork**. La seguente figura mostra un esempio di stampa in console:

```

1
2 NeuralNetwork(
3     depth = 2,
4     input_size = 784,
5     network_layers = [
6         Layer(
7             size = 121,
8             act_fun = <function leaky_relu at 0x1071183a0>,
9             inputs_size = (1, 784)
10            weights_shape = (121, 784),
11            biases_shape = (121, 1)
12        ),
13        Layer(
14            size = 10,
15            act_fun = <function identity at 0x1074f6680>,
16            inputs_size = (1, 121)
17            weights_shape = (10, 121),
18            biases_shape = (10, 1)
19        )
20    ],
21    err_fun = <function cross_entropy_softmax at 0x1074f68c0>,
22    training_report = TrainingReport(
23        num_epochs = 1,
24        elapsed_time = 0.197 secondi,
25        training_examples = 90,
26        training_error = 66.96509,
27        training_accuracy = 14.44%,
28        validation_examples = 10,
29        validation_error = 363.26759,
30        validation_accuracy = 0.00%
31    ),
32    training_params = TrainingParams(
33        batch_size = 200,
34        epochs = 2,
35        learning_rate = 0.20,
36        rprop = True,
37        eta_minus = 0.68314,
38        eta_plus = 1.31068,
39        delta_min = 0.00000,
40        delta_max = 50.00000,
41        es_patience = 15,
42        es_delta = 0.10000
43    )
44)
45

```

Figura 2.8: Visualizzazione del contenuto di un oggetto della classe **NeuralNetwork**

Capitolo 3

Addestramento e sperimentazione

In questo capitolo si esplora in modo dettagliato come il team di sviluppo ha affrontato il processo di addestramento e sperimentazione delle reti neurali tramite l'utilizzo della propria libreria [1]. Si inizia con una descrizione accurata del setup sperimentale, in modo tale da fornire al lettore tutte le informazioni necessarie affinché possa riprodurre gli esperimenti proposti utilizzando altre librerie.

Il capitolo, quindi, si conclude con l'esposizione di tutti i risultati ottenuti tramite grafici e tabelle, e commentati scrupolosamente per facilitare il lettore nella comprensione delle informazioni visualizzate.

3.1 Setup sperimentale

Il file *k_fold.ipynb* nella directory *code* è un notebook che fornisce un ambiente interattivo per l'esplorazione e l'addestramento di reti neurali per il riconoscimento delle cifre del MNIST dataset, utilizzando la libreria messa a disposizione dagli autori.

L'obiettivo principale di questo studio è confrontare le prestazioni ottenute in fase di training di un numero considerevole di modelli che condividono la stessa architettura di rete ma che differiscono solo per la scelta di alcuni iper-parametri, tra cui **eta minus**, **eta plus** e il **numero di neuroni** dell'unico layer interno.

In particolare, la scelta dei valori degli iper-parametri in queste differenti combinazioni è data dall'applicazione di tecniche di ricerca come la **Grid Search** e la **Random Search**. Invece, la validazione di tutti questi modelli è stata eseguita grazie alla tecnica **K-fold Cross Validation**.

Infatti, con questo quaderno, gli utenti possono:

- 1. Eseguire la *K-fold cross validation*.**

È una tecnica di validazione che suddivide il dataset in **k** sottoinsiemi (e.g. folds) di uguale dimensione e utilizza, per ognuna delle **k** iterazioni, una parte indipendente del training set per la fase di validazione e tutte le altre per l'addestramento.

Alla fine, i risultati delle **k** iterazioni vengono aggregati (calcolandone media e deviazione standard) per fornire una stima complessiva delle prestazioni del modello e scegliere, infine, la combinazione di iper-parametri che restituiscono il minor errore medio di validazione.

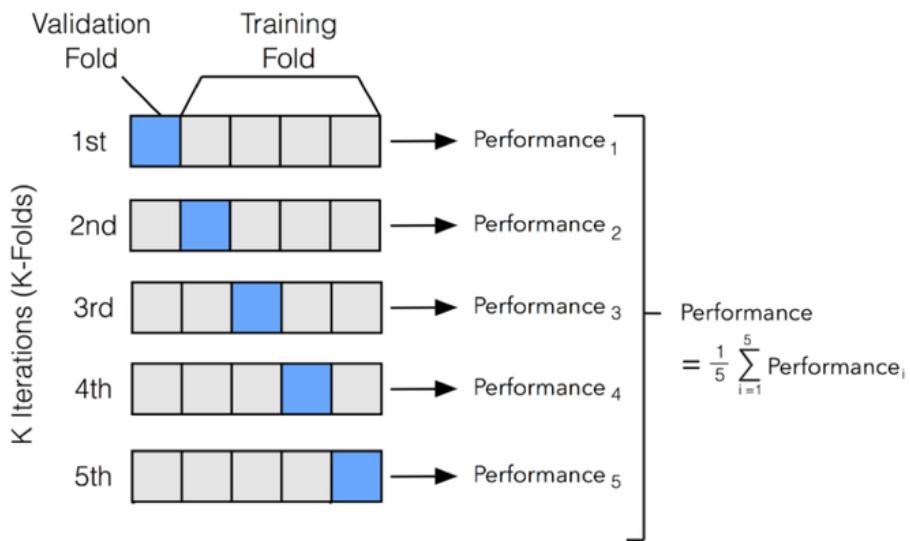


Figura 3.1: Esempio di applicazione della K-fold Cross Validation

In particolare, si salvano anche altre metriche di valutazione, tra cui: l'errore di validazione minimo e massimo; l'accuracy di validazione minima e massima; lo storico di tutti i valori di errore e accuracy, sia della fase di training sia della fase validation (utile per il plotting delle curve di apprendimento).

Si utilizza la funzione `def k_fold_cross_validation(out_directory : str, Xtrain : list [np.ndarray], Ytrain : list [np.ndarray], k : int = constants.DEFAULT_K_FOLD_VALUE, l_sizes : list[int] = constants.DEFAULT_LAYER_NEURONS, params : TrainingParams = None) -> list[dict].`

- 2. Eseguire la *grid search* per il tuning degli iper-parametri.**

Esegue una ricerca sistematica degli iper-parametri del modello esplorando tutte le possibili combinazioni di uno spazio di valori ben delimitato da una griglia (da qui, il nome **Grid Search**) per ogni iper-parametro.

Questa funzione addestra e valuta il modello per ogni combinazione, utilizzando come tecnica di validazione la K-fold cross validation. Infine, mostra la combinazione di iper-parametri che ottimizza l'errore di validazione, così da scegliere il modello con le migliori prestazioni finali.

Per questo studio, sono state valutate le seguenti combinazioni di iper-parametri:

Tabella 3.1: Combinazioni di iper-parametri per la **Grid Search**

<i>Id</i>	<i>Eta minus</i>	<i>Eta plus</i>	<i>Neuroni hidden layer</i>
0	0.5	1.2	32
1	0.5	1.2	64
2	0.5	1.2	128
3	0.5	1.3	32
4	0.5	1.3	64
5	0.5	1.3	128
6	0.5	1.5	32
7	0.5	1.5	64
8	0.5	1.5	128
9	0.7	1.2	32
10	0.7	1.2	64
11	0.7	1.2	128
12	0.7	1.3	32
13	0.7	1.3	64
14	0.7	1.3	128
15	0.7	1.5	32
16	0.7	1.5	64
17	0.7	1.5	128
18	0.7	1.2	32
19	0.7	1.2	64

Si utilizza la funzione `def grid_search_cv() -> list[tuple[float, float, int]]`. Si consiglia di approfondire la lettura alla sezione 2.2.2 a pagina 7 per capire come sono stati i scelti i valori di ricerca proposti nella tabella 3.1.

3. Eseguire la random search per il tuning degli iper-parametri.

Ricerca la miglior combinazione di iper-parametri campionando in modo casuale (da qui, il nome **Random Search**) un numero limitato di combinazioni dell'intero spazio dei parametri. In altre parole, invece di esaminare tutte le possibili combinazioni come nella Grid Search, la Random Search seleziona un numero specificato di combinazioni casuali e valuta il modello con ciascuna di esse.

Per questo studio, sono state valutate le seguenti combinazioni di iper-parametri:

Tabella 3.2: Combinazioni di iper-parametri per la **Random Search**

<i>Id</i>	<i>Eta minus</i>	<i>Eta plus</i>	<i>Neuroni hidden layer</i>
0	0.88980	1.35678	113
1	0.58882	1.28495	125
2	0.57416	1.41315	54
3	0.79120	1.49696	36
4	0.81470	1.30539	103

Questo approccio può essere più efficiente in termini di risorse computazionali, permettendo di trovare buone combinazioni di iper-parametri con meno iterazioni e in un minor tempo.

Si utilizza la funzione `def random_search_cv(r : int = constants.DEFAULT_RANDOM_COMBINATIONS) -> list[tuple[float, float, int]]`, dove `r` è il numero di

combinazioni di iper-parametri da campionare. Si consiglia di approfondire la lettura alla sezione 2.2.2 a pagina 7 per capire come sono stati i scelti i valori di ricerca proposti nella tabella 3.2.

3.1.1 Architettura della macchina

La macchina scelta per l'esecuzione del processo di addestramento è un MacBook Pro 2022, un laptop di fascia alta progettato da Apple.

Il modello specifico utilizzato è equipaggiato con il processore Apple M2 (4 core ad alte prestazioni da 3.20GHz e 4 core ad alta efficienza da 2GHz), 16GB di memoria RAM unificata e un'unità di memoria SSD da 512GB. In particolare, il processore Apple M2 è un System-on-a-Chip che integra una GPU da 10 core, un Neural Engine a 16 core (per accelerare le operazioni di machine learning e AI) e la memoria RAM in un unico circuito integrato, offrendo miglioramenti significativi in termini di velocità ed efficienza energetica.

3.1.2 Architettura della rete

Negli esperimenti condotti dal team di sviluppo, tutti i modelli di rete neurale addestrati condividono la seguente architettura:

- L'input layer è costituito di 784 neuroni, in modo tale da avere una connessione in input per ogni pixel delle immagini 28×28 del MNIST dataset.
- L'output layer è composto di 10 neuroni, rappresentativi delle dieci classi del dataset. Si è deciso di utilizzare la funzione **identità** come funzione di attivazione per questo strato, in modo tale da poter sfruttare la funzione **cross-entropy softmax** come funzione di errore.
- Data la relativa semplicità del MNIST dataset, un singolo strato interno è sufficiente per catturare le principali caratteristiche degli esempi di addestramento proposti. Per questo motivo, ma anche per ridurre la complessità del modello, si è deciso di utilizzare un singolo layer nascosto.

Questa scelta progettuale porta con sé due vantaggi: è stato possibile valutare meglio l'impatto delle combinazioni di iper-parametri sull'addestramento dei modelli, senza introdurre ulteriori variabili che potrebbero complicare l'interpretazione dei risultati; in secondo luogo, è stato possibile eseguire un maggior numero di esperimenti in tempi ragionevoli.

- Per lo strato nascosto, è stata impiegata la funzione di attivazione **Leaky ReLU**. Si rimanda alla sezione 2.2.4 a pagina 12 per una descrizione più precisa di questa funzione.

- La funzione di errore scelta è la funzione **cross-entropy softmax**, comunemente utilizzata per problemi di classificazione multiclass, in grado di calcolare la distribuzione di probabilità delle predizioni rispetto all'input dato e alle classi del problema utilizzando la funzione **softmax**. Si rimanda alla sezione 2.2.5 a pagina 13 per una descrizione più precisa di questa funzione.

3.1.3 Iper-parametri di addestramento: Grid Search e Random Search

Per la fase di addestramento del modello, la scelta degli iper-parametri gioca un ruolo cruciale nel determinare le prestazioni finali. I due approcci utilizzati per il tuning degli iper-parametri, quali **Eta minus**, **Eta plus** ed il **numero di neuroni** dell'hidden layer, sono stati la Grid Search e la Random Search. Si rimanda alla sezione 1 a pagina 38 per una descrizione esaustiva di queste tecniche di ricerca.

La tecnica del **Random Search** è particolarmente efficace in contesti dove l'ottimizzazione dei tempi di calcolo è cruciale e dove non è ben chiaro lo spazio degli iper-parametri a priori. Questo metodo è stato proposto da James Bergstra e Yoshua Bengio [4], e si basa sull'idea che esplorare una vasta gamma di combinazioni casuali possa condurre a migliori risultati di ottimizzazione rispetto a strategie più deterministiche come la Grid Search.

Il lavoro di Bergstra e Bengio [4] ha, inoltre, dimostrato che la tecnica del **Random Search** può essere una strategia molto efficiente per trovare la combinazione di iper-parametri ottimale in diversi contesti di machine learning.

Gli altri iper-parametri specifici della fase di addestramento, esplicitati nella sezione 2.5 a pagina 23, sono inizializzati tramite i valori di default esposti nella sezione 2.2.2 a pagina 7.

3.1.4 Utilizzo del dataset

L'estrazione ed il caricamento del MNIST training set e del MNIST test set sono affidati alla funzione **loadDataset()**, descritta nella sezione 2.2.6 a pagina 13.

Questa funzione, che viene eseguita prima di avviare la ricerca degli iper-parametri e la loro valutazione, segue questi passaggi:

- Mescola i dati in modo casuale applicando la funzione ***np.random.shuffle()***, per evitare che la sequenzialità delle immagini possa in qualche modo alterare i risultati ottenuti.
- Normalizza i dati in un range tra 0 e 1, in modo tale da puntare una convergenza più rapida riducendo il rischio di incorrere in problemi come il vanishing o exploding gradient.
- Separa le etichette degli esempi dai dati relativi alle immagini, per poi convertirle in formato one-hot utilizzando la funzione ***convert_to_one_hot()***.

3.2 Risultati

A partire dal setup sperimentale descritto nella sezione 3.1 a pagina 37, sono stati condotti diversi esperimenti per ottimizzare la selezione degli iper-parametri tramite le due tecniche di ricerca **Grid Search** e **Random Search**, entrambe valutate grazie all’addestramento con la tecnica della **K-fold Cross Validation**.

In questa sezione vengono mostrati i grafici di addestramento, realizzati tramite l’utilizzo delle funzioni esplicite nella sezione 2.2.7 a pagina 14, e si esplorano le conclusioni tratte dal team di sviluppo per: 3 esperimenti realizzati tramite la tecnica **Grid Search**; 2 esperimenti realizzati tramite la tecnica **Random Search**.

Infine, si mostrano i risultati di testing, realizzati grazie alle funzioni spiegate nella sezione 2.2.8 a pagina 15, per una rete neurale addestrata sull’intero dataset con la combinazione di iper-parametri che ha ottenuto la massima accuratezza tra le due tipologie di ricerca.

Di seguito sono riportati i risultati di tutti gli esperimenti effettuati per le combinazioni di iper-parametri valutate, anche di quelli che non sono commentati in dettaglio in questa documentazione.

stats.xlsx

	Eta minus	Eta plus	Hidden layer	Mean error	Std error	Mean accuracy	Std accuracy	Elapsed time
0	0.50000	1.20000	32	3.50795	1.38298	86.93%	1.17%	0.15914
1	0.50000	1.20000	64	6.60420	2.88122	88.45%	0.94%	0.17868
9	0.70000	1.20000	32	8.04148	1.91988	89.66%	0.93%	0.18079
3	0.50000	1.30000	32	9.34357	6.78640	87.19%	1.09%	0.17314
2	0.50000	1.20000	128	11.24878	1.72766	90.34%	0.98%	0.35908
10	0.70000	1.20000	64	14.70169	3.49692	91.58%	0.64%	0.23096
4	0.50000	1.30000	64	16.70726	4.07364	89.97%	0.82%	0.26087
11	0.70000	1.20000	128	20.78389	2.07093	92.50%	0.61%	0.38584
5	0.50000	1.30000	128	25.31499	7.79747	90.59%	1.05%	0.39481
17	0.70000	1.50000	128	45.88648	6.58758	93.53%	0.93%	1.02955
16	0.70000	1.50000	64	54.70817	9.72367	92.28%	1.37%	0.65812
19	0.90000	1.20000	64	85.84737	66.22457	83.58%	18.44%	0.48415
18	0.90000	1.20000	32	93.83661	36.74053	63.48%	27.68%	0.22758
8	0.50000	1.50000	128	96.42400	25.64346	86.40%	3.62%	0.91724
15	0.70000	1.50000	32	96.88507	50.19836	79.62%	20.58%	0.32075
12	0.70000	1.30000	32	123.68096	37.16580	65.10%	22.59%	0.17883
7	0.50000	1.50000	64	127.95712	36.96545	81.94%	5.22%	0.40758
14	0.70000	1.30000	128	133.46216	43.92417	81.18%	6.20%	0.72081
13	0.70000	1.30000	64	148.23136	43.90035	75.51%	15.34%	0.40700
6	0.50000	1.50000	32	151.61578	35.12149	71.18%	18.23%	0.22772

Figura 3.2: Risultati degli esperimenti tramite **Grid Search**

stats.xlsx

	Eta minus	Eta plus	Hidden layer	Mean error	Std error	Mean accuracy	Std accuracy	Elapsed time
1	0.58882	1.28495	125	32.90657	7.11731	91.68%	1.04%	0.39079
0	0.88980	1.35678	113	45.09240	5.79128	93.64%	0.82%	0.77434
4	0.81470	1.30539	103	68.30844	53.32236	88.57%	12.87%	0.83246
3	0.79120	1.49696	36	77.13150	31.57367	84.99%	16.68%	0.42700
2	0.57416	1.41315	54	135.10816	44.71996	76.65%	14.77%	0.34131

Figura 3.3: Risultati degli esperimenti tramite **Random Search**

3.2.1 Grid Search (n.10): compromesso tra errore e accuratezza

Per questo esperimento sono stati usati i seguenti iper-parametri:

- **Eta minus** = 0.70000
- **Eta plus** = 1.20000
- **Hidden layer** = 64

L'immagine 3.4 mostra i risultati dell'esecuzione della K-fold cross validation sulla combinazione n.10 della lista di iper-parametri restituita dalla Grid Search.

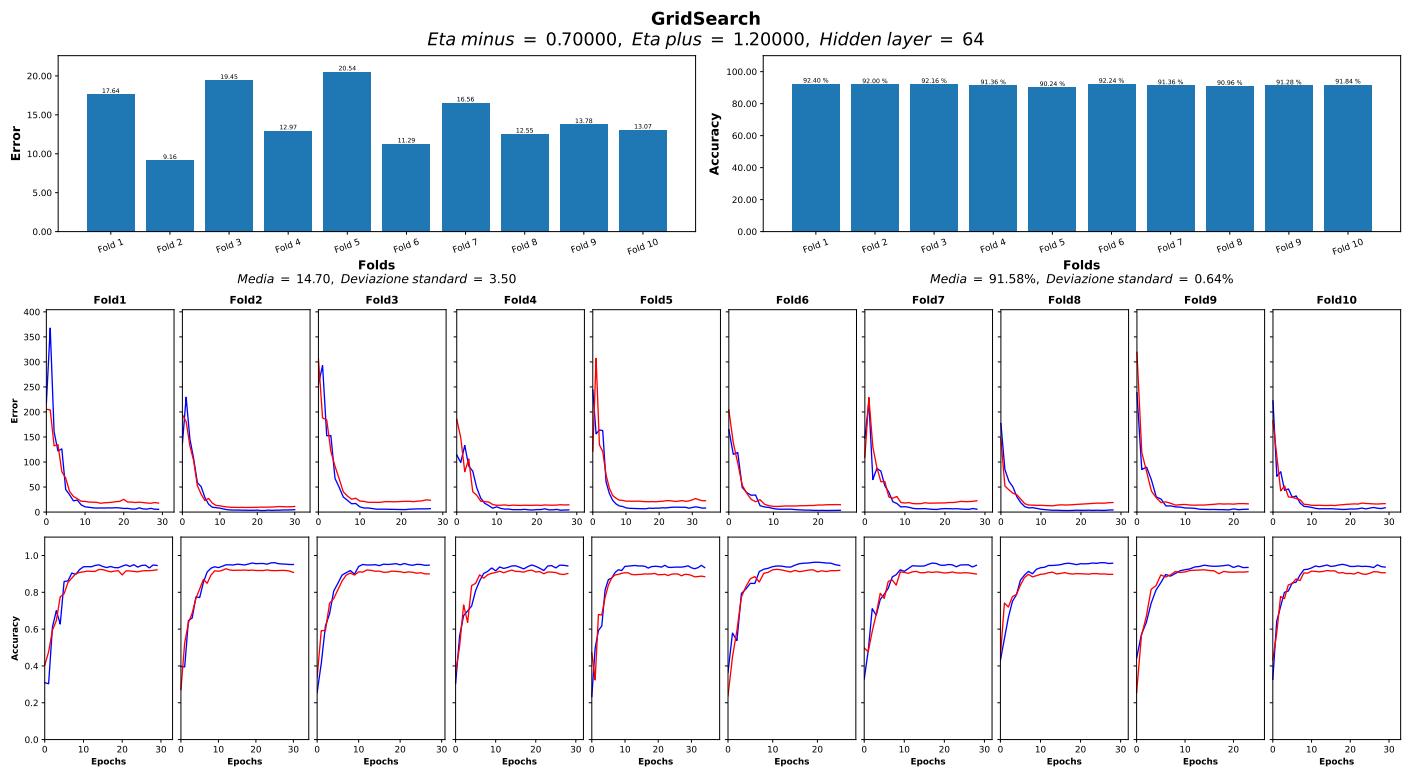


Figura 3.4: Report della Grid Search sulla combinazione n.10

In quest'esperimento (come anche in tutti quelli successivi) è possibile notare **una forte variabilità delle misure di errore di validazione rispetto alla media** (e.g. deviazione standard), ma **una variabilità molto più stabile per i valori di accuratezza**.

Questo può essere dovuto alla diversa natura di queste due metriche di valutazione: gli errori tendono naturalmente a una maggiore variazione, siccome sono calcolati su una scala molto più ampia rispetto a quella dell'accuracy che è, invece, normalizzata tra 0 e 1 (e.g. tra 0% e 100%).

Le curve di apprendimento mostrano **un'evoluzione del processo di addestramento molto stabile**, che raggiunge la miglior configurazione di parametri tra l'epoca n.7 e l'epoca n.9 (per quasi tutte le fold). Ciò potrebbe risultare dal valore dell'iper-parametro **Eta plus**: infatti, valori piccoli per questo iper-parametro portano una maggiore stabilità nel training.

3.2.2 Grid Search (n.17): massima accuratezza

Per questo esperimento sono stati usati i seguenti iper-parametri:

- **Eta minus** = 0.70000
- **Eta plus** = 1.50000
- **Hidden layer** = 128

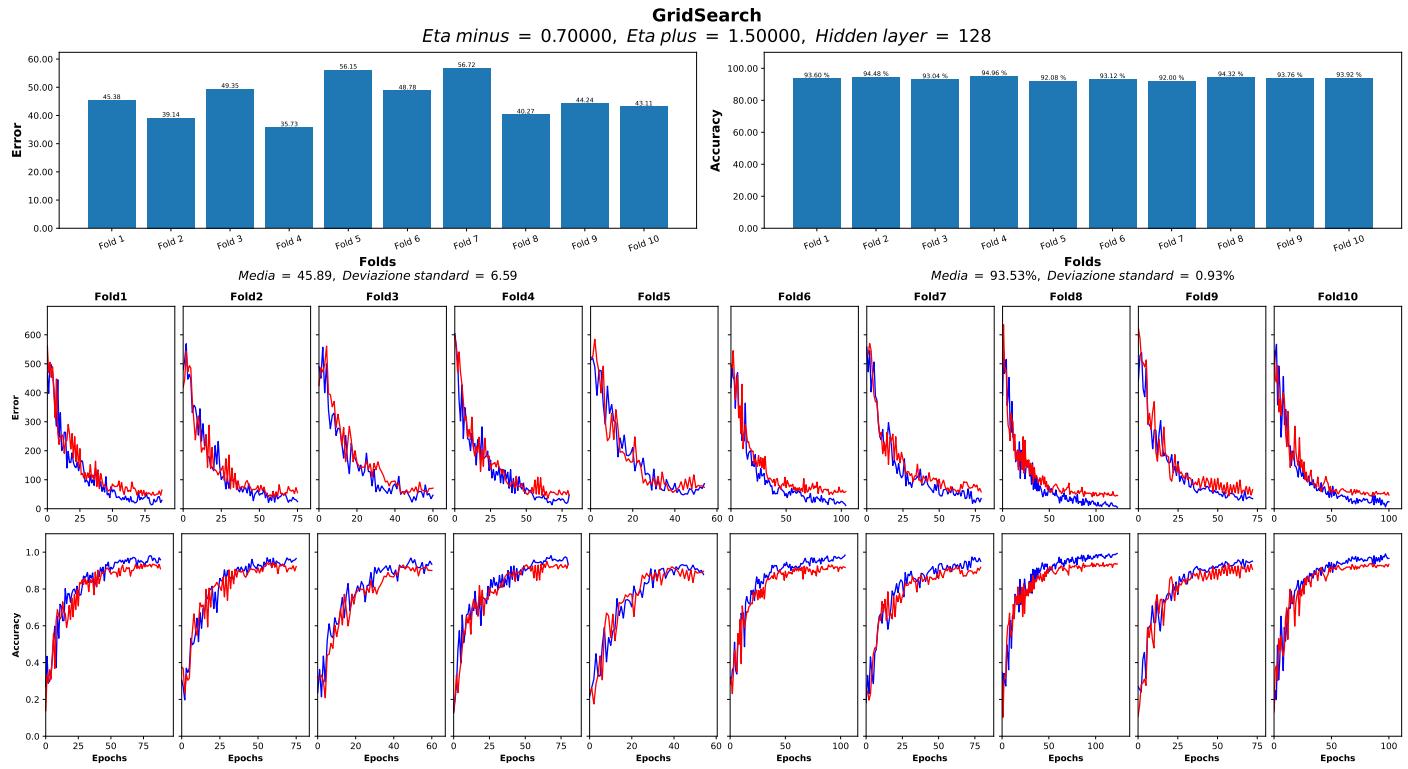


Figura 3.5: Report della Grid Search sulla combinazione n.17

Rispetto alla combinazione di iper-parametri vista nella sezione precedente (e.g. la combinazione n.10 della Grid Search), questa combinazione ha generato delle **curve di apprendimento molto più instabili**.

Nonostante questo, però, l'addestramento si è concluso con la massima accuratezza tra tutte le combinazioni valutate per la Grid Search, e un errore di validazione compreso tra 35.73 e 56.72: per questo motivo, tale combinazione di iper-parametri **potrebbe mostrare un livello di generalizzazione non adatto all'applicazione in un contesto reale**.

3.2.3 Grid Search (n.1): compromesso tra accuratezza e tempo

Per questo esperimento sono stati usati i seguenti iper-parametri:

- **Eta minus** = 0.50000
- **Eta plus** = 1.20000
- **Hidden layer** = 64

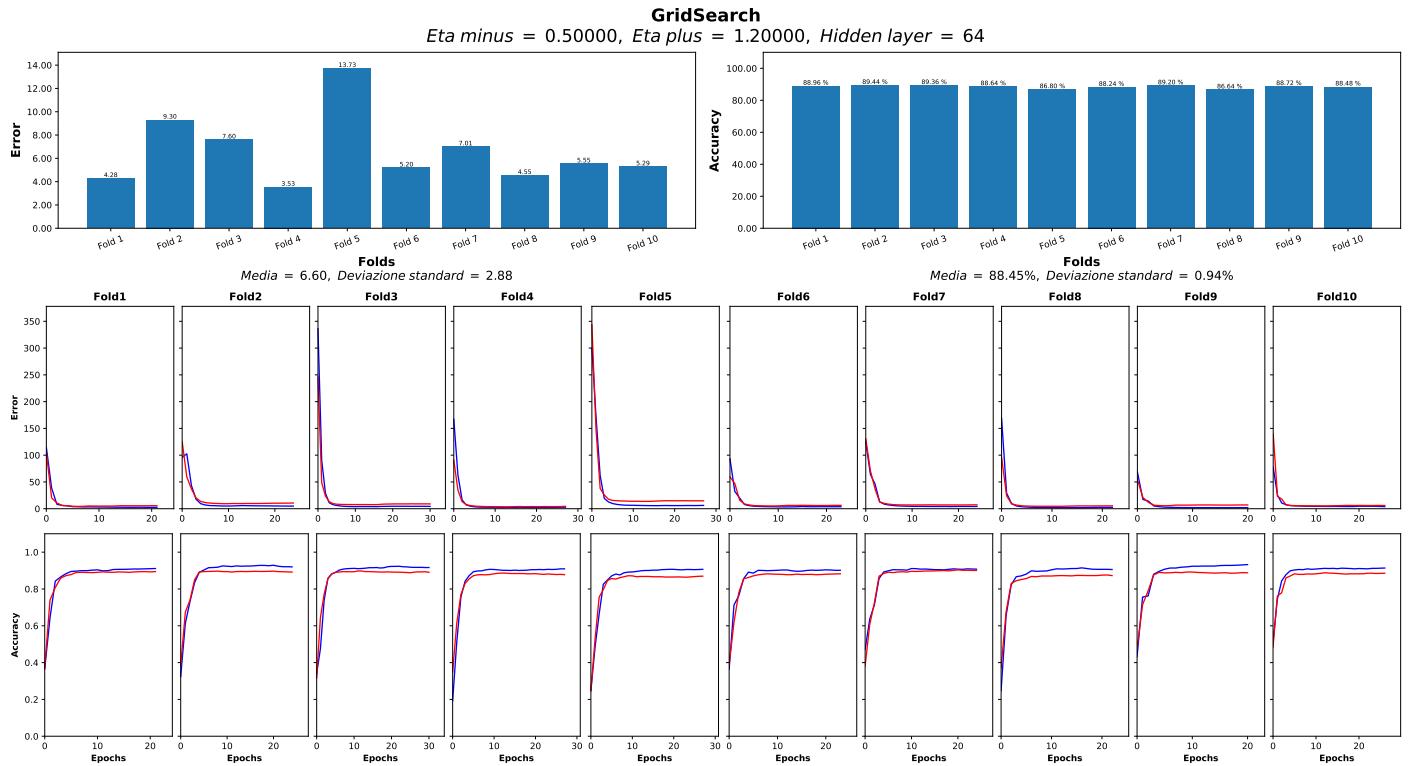


Figura 3.6: Report della Grid Search sulla combinazione n.1

In questo terzo esperimento, si osserva **una più rapida riduzione dell'errore** che, già a partire dalle prime epoche di addestramento, raggiunge il minimo. Quindi, dopo 15 epoche (e.g. il limite imposto dalla pazienza dall'algoritmo di early stopping) il training si interrompe prima di raggiungere le 30 epoche (nella maggior parte dei casi).

Pertanto, il tempo impiegato dalla K-fold cross validation per completare la valutazione di questa combinazione di iper-parametri è più o meno minore di tutti gli altri (e.g. circa 11 minuti), ma ottiene comunque **un'accuracy di validazione alta**, probabilmente **grazie al numero di neuroni dell'hidden layer**.

3.2.4 Random Search (n.1): compromesso tra errore e accuratezza

Per questo esperimento sono stati usati i seguenti iper-parametri:

- **Eta minus** = 0.58882
- **Eta plus** = 1.28495
- **Hidden layer** = 125

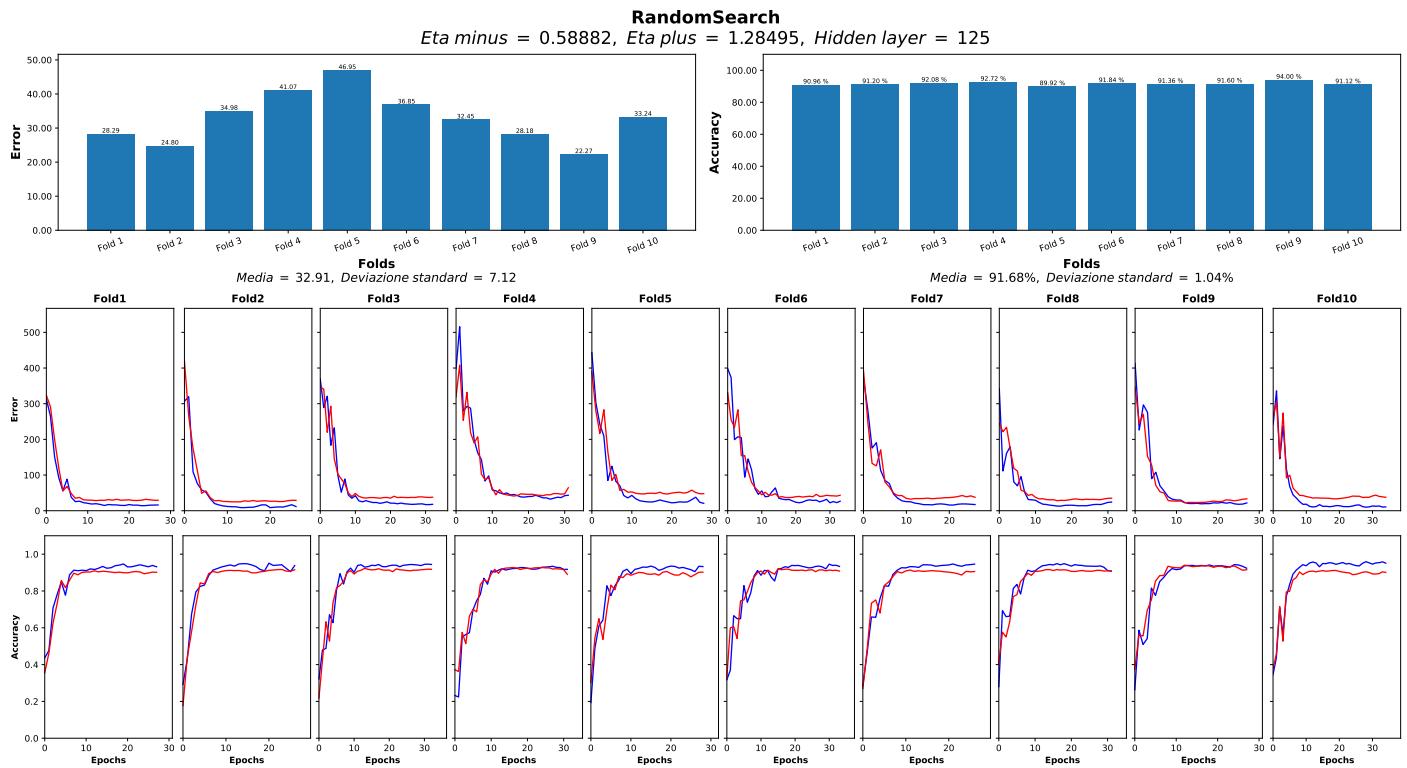


Figura 3.7: Report della Random Search sulla combinazione n.1

L'**accuratezza media** del modello è notevole, registrando un valore dell'84.99%. Tuttavia, la **deviazione standard** elevata del 16.68% suggerisce una significativa variabilità nelle performance tra le diverse fold.

Anche l'**errore medio**, piuttosto alto a 77.13 con una deviazione standard del 31.57, evidenzia una notevole variabilità nelle prestazioni del modello. Alcune fold mostrano errori considerevolmente superiori ad altri.

Questi risultati indicano che il modello può essere estremamente preciso in certe circostanze, ma manifesta gravi errori in altre: in altre parole, la **combinazione valutata non generalizza bene il problema** della classificazione delle cifre.

In conclusione, sebbene l'accuratezza media sia promettente, la notevole variabilità nell'errore indica la presenza di **margini significativi per miglioramenti**, che potrebbero rendere le performance del modello più coerenti e affidabili in diverse condizioni.

3.2.5 Random Search (n.3): compromesso tra accuratezza e tempo

Per questo esperimento sono stati usati i seguenti iper-parametri:

- **Eta minus** = 0.79120
- **Eta plus** = 1.49696
- **Hidden layer** = 36

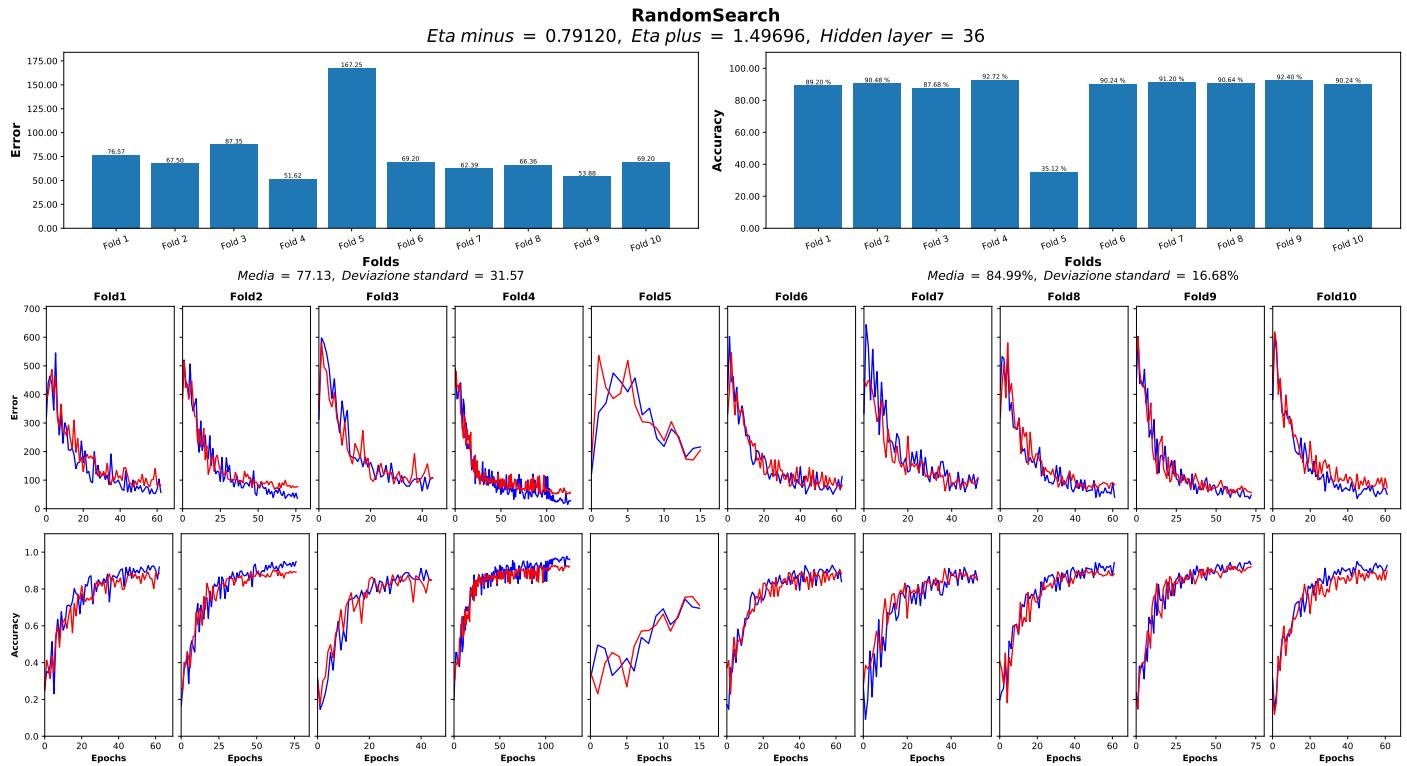


Figura 3.8: Report della Random Search sulla combinazione n.3

Per questo esperimento, **si noti il numero di epoche** che è stato necessario per i modelli in questione per raggiungere la convergenza: alcuni a poco più di 40 epoche, altri **addirittura sopra le 100 epoche**. I risultati di accuratezza e di errore sono abbastanza simili, tuttavia l'instabilità delle curve di apprendimento, probabilmente introdotta dai valori elevati di **Eta minus** e **Eta plus**, è considerevole.

Molto particolare **il caso della fold n.5**, che ha riportato valori anomali sia per l'errore di validazione che per l'accuracy di validazione. Questo comportamento è quasi certamente **causato dagli esempi di addestramento forniti in input**, nonché dalla configurazione casuale di parametri iniziali: la rete neurale in questione ha ottenuto già alla prima epoca le migliori metriche di valutazione che, quindi, hanno causato l'interruzione della fase di addestramento esattamente alla 15° epoca (e.g. per l'early stopping).

3.2.6 Random Search (n.0): la miglior combinazione di iper-parametri

Per questo esperimento sono stati usati i seguenti iper-parametri:

- **Eta minus** = 0.58882
- **Eta plus** = 1.28495
- **Hidden layer** = 125

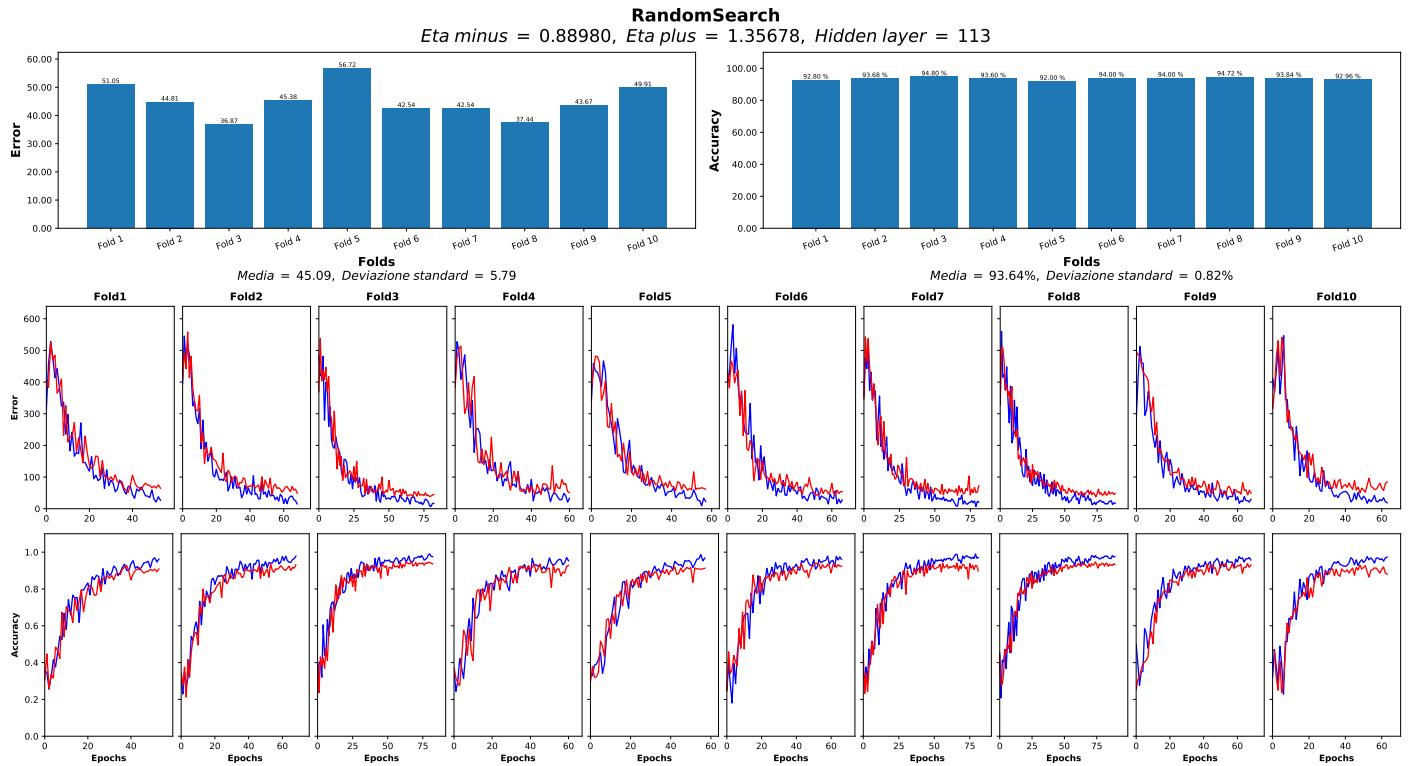


Figura 3.9: Report della Random Search sulla combinazione n.0

Da un'analisi dettagliata di questo report, si può evidenziare che il modello con questa combinazione di iper-parametri restituisce un'ottima **accuratezza complessiva** (e.g. la migliore tra tutti gli esperimenti condotti), con una media del 93.64% e una deviazione standard molto bassa dello 0.82%. Questo suggerisce la rete neurale è generalmente coerente nelle sue previsioni per tutte le fold valutate.

Tuttavia, l'analisi degli errori rivela una maggiore variabilità: si parte **da un minimo** di 36.87, fino ad arrivare **ad un massimo** di 56.72, con una **media** di 45.09 ed una **deviazione standard** di 5.79. Questo indica che la rete neurale può comportarsi in modo significativamente diverso su diversi subset di dati: in altre parole, sono certamente necessarie ulteriori ottimizzazioni oppure uno studio più approfondito delle cause di questa variabilità.

In particolare, dato che questa combinazione di iper-parametri ha restituito le migliori performance in termini di accuratezza media di validazione tra tutti gli esperimenti condotti, è stato deciso di testare un ulteriore modello di rete neurale. Tale modello, addestrato e validato sull'intero MNIST dataset (in proporzione, 80% **training set** e 20% **validation set**), per verificare l'attendibilità dei risultati ottenuti dalla K-fold cross validation.

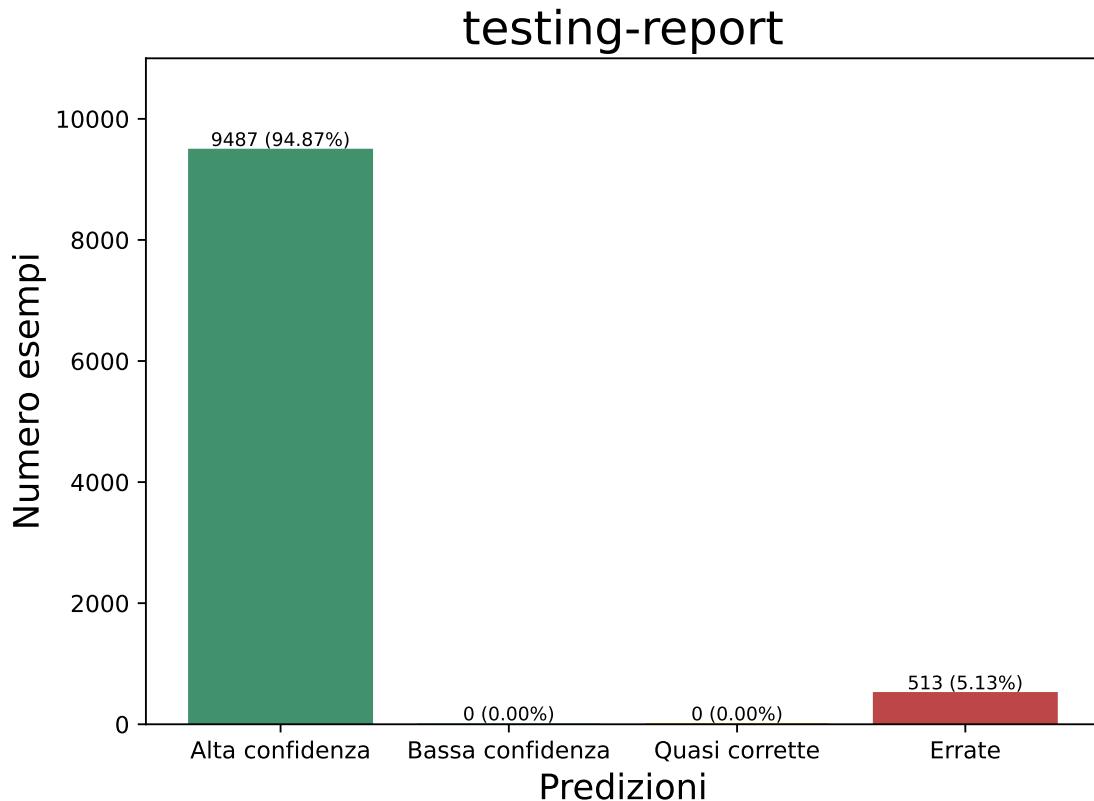


Figura 3.10: Report di classificazione sulla combinazione n.0 (Random Search)

Per concludere, si mostrano i risultati di due previsioni generate dalla rete neurale, la prima corretta e l'altra errata, in un'immagine che:

- Presenta sulla sinistra la rappresentazione in scala di grigi dell'esempio di testing e la corrispondente etichetta.
- Propone sulla destra un istogramma che descrive la distribuzione di probabilità della previsione generata dalla rete neurale e la relativa etichetta.

testing-report_1

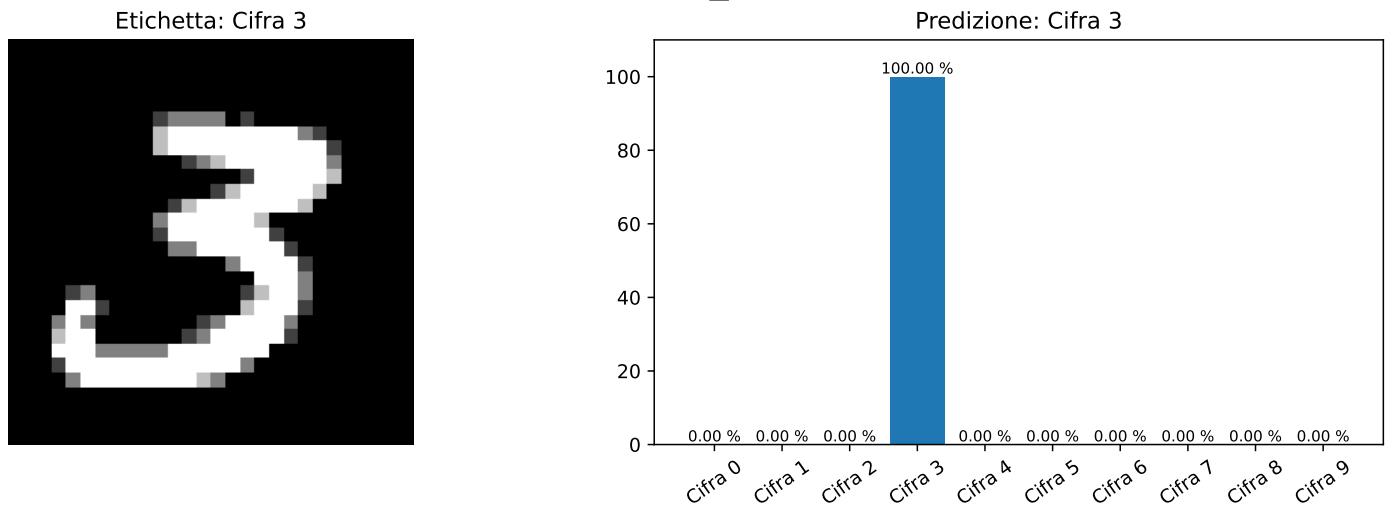


Figura 3.11: Predizione corretta della rete neurale per la cifra n.1 dal MNIST test set

testing-report_43

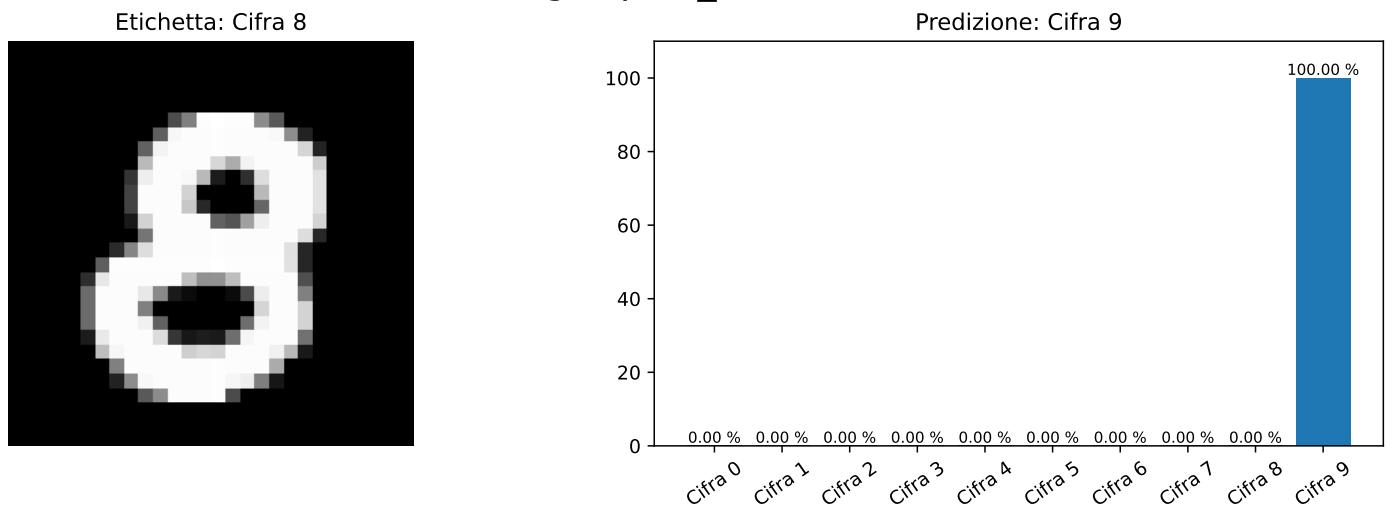


Figura 3.12: Predizione errata della rete neurale per la cifra n.43 dal MNIST test set

Capitolo 4

Conclusioni

4.1 Considerazioni

In questo capitolo conclusivo sono esposte alcune considerazioni personali del team di sviluppo riguardo i risultati ottenuti durante il lavoro al progetto. È fondamentale riflettere sui risultati ottenuti e sulle implicazioni che emergono dagli esperimenti condotti. L'analisi dei dati e delle performance dei modelli addestrati, ottimizzati attraverso la selezione degli iper-parametri con Grid Search e Random Search, ha fornito importanti indicazioni sull'efficacia di tali approcci nello studio effettuato.

L'adozione della **Grid Search** ha consentito un'esplorazione sistematica delle combinazioni di iper-parametri, rivelando configurazioni che massimizzano l'accuratezza dei modelli di rete neurale. D'altro canto, l'implementazione della **Random Search** ha dimostrato di essere una strategia altrettanto valida, specialmente in contesti in cui una ricerca esaustiva non è praticabile. Infatti, nei risultati degli esperimenti nella sezione 3.2 a pagina 42, si può notare che la miglior combinazione di iper-parametri (per l'accuracy di validazione) è stata ottenuta proprio grazie alla Random Search.

Per quanto riguarda l'utilizzo della **K-fold Cross Validation**, entrambi i metodi hanno mostrato robustezza nella generalizzazione dei modelli, garantendo che le performance osservate siano rappresentative e affidabili.

4.2 Futuri sviluppi e miglioramenti

Guardando al futuro, ci sono diverse direzioni promettenti per espandere e migliorare ulteriormente il lavoro svolto in questo studio. Alcuni possibili sviluppi includono:

- ***Ottimizzazione dell'implementazione interna dei layer***

Rivedere e ottimizzare l'implementazione interna dei layer neurali potrebbe portare a miglioramenti significativi delle prestazioni. Questo potrebbe includere l'adozione di tecniche avanzate di ottimizzazione numerica o la riprogettazione della classe **Layer** in un'ottica di programmazione a oggetti ancora più efficiente.

- ***Incorporazione di nuovi dati e domini applicativi***

Esplorare l'applicazione dei modelli ad altri dataset e domini applicativi potrebbe rivelare nuove sfide e opportunità per adattare e ottimizzare ulteriormente i modelli neurali.

- ***Valutazione delle prestazioni in contesti realistici***

Testare le prestazioni dei modelli su dati reali e in contesti operativi potrebbe fornire una valutazione più completa delle loro capacità e dei loro limiti.

- ***Espansione delle metriche di valutazione***

Considerare l'utilizzo di metriche aggiuntive oltre all'errore e all'accuratezza, per una valutazione più completa delle prestazioni.

Bibliografia

- [1] Alessandro Trincone e Mario Gabriele Carofano, *La libreria Neural-Network*,
<https://github.com/Trincalex/Neural-Network>.
- [2] M. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. indirizzo:
<https://books.google.it/books?id=STDBswEACAAJ>.
- [3] C. Igel e M. Hüskens, «Empirical evaluation of the improved Rprop learning algorithms»,
Neurocomputing, vol. 50, pp. 105–123, 2003, ISSN: 0925-2312. DOI:
[https://doi.org/10.1016/S0925-2312\(01\)00700-7](https://doi.org/10.1016/S0925-2312(01)00700-7). indirizzo:
<https://www.sciencedirect.com/science/article/pii/S0925231201007007>.
- [4] J. Bergstra e Y. Bengio, «Random Search for Hyper-Parameter Optimization», *Journal of Machine Learning Research*, vol. 13, n. 10, pp. 281–305, 2012. indirizzo:
<http://jmlr.org/papers/v13/bergstra12a.html>.