

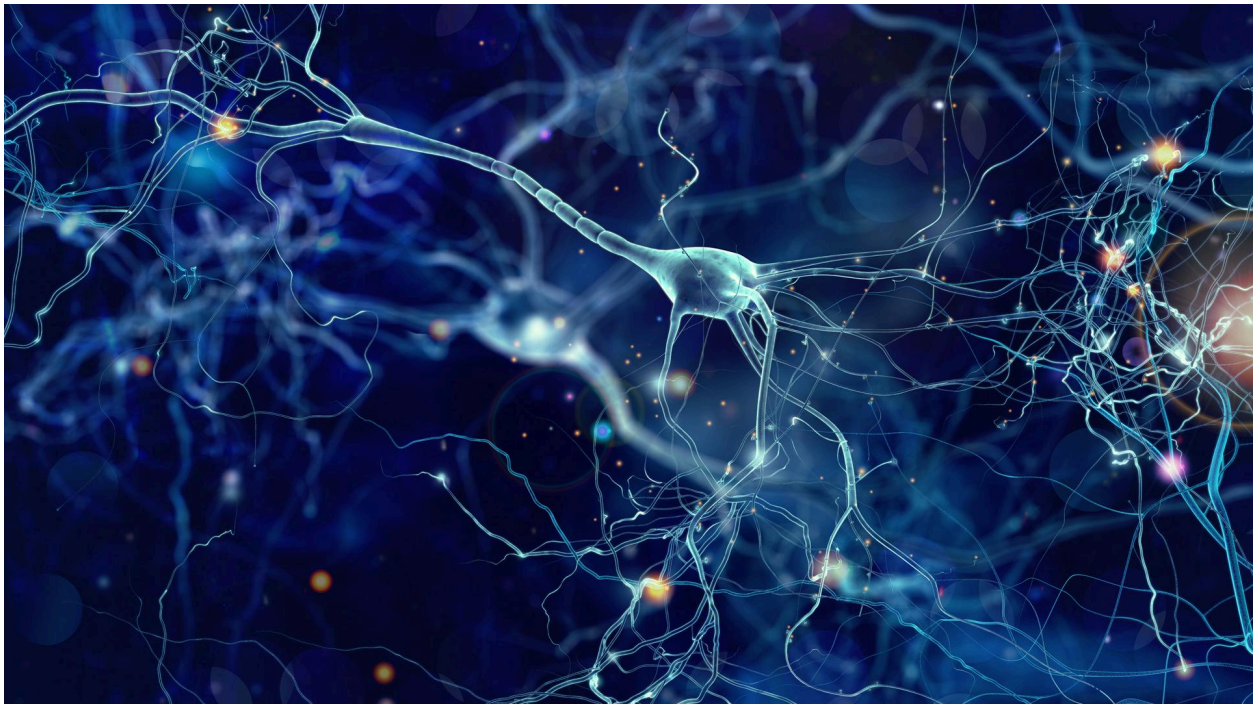


UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

Machine Learning (Mod. Neural Networks and Deep Learning)

Progetto di Neural Network and Deep Learning

Traccia 4



Pietro Malara - N97000462

Antonio Curione - N97000443

CdS Magistrale in Informatica

A.A. 2023/2024

1. Introduzione	4
1.1 Traccia	4
1.2 Cenni teorici	5
1.2.1 Rete neurale	5
1.2.2 Feed forward	6
1.2.3 Back propagation e discesa del gradiente	6
1.2.4 Resilient propagation	7
1.2.5 K-fold cross validation	8
2. Architettura del sistema	10
2.1 Tecnologie utilizzate	10
2.1.1 Python e le sue librerie	10
2.1.2 Github	10
2.1.2 Git LFS	11
2.1.3 Visual Studio Code	11
2.1.4 Discord	11
2.2 Dataset	11
2.3 Funzionalità richieste	12
2.4 Strutture dati utilizzate	12
2.5 Algoritmi utilizzati	13
3. Analisi del codice e delle prestazioni	18
3.1 Specifiche del sistema	18
3.1.1 loadDataset	18
3.1.2 oneHotEnc	19
3.1.3 Activation Functions	19
3.1.4 Error Functions	20
3.1.5 newNetwork	21
3.1.6 setActFunct	22
3.1.7 Funzioni ausiliarie	22
3.1.8 forwardPropagation	23
3.1.9 trainForwardPropagation	23
3.1.10 backPropagation	24
3.1.11 trainBackPropagation	25
3.1.12 trainResilientPropagation	26
3.1.13 networkAccuracy & testAccuracy	27
3.1.14 crossValidationKFold	28
3.1.15 myPlot	30
3.2 Discussione dei risultati	30

3.2.1 Set up sperimentale	31
3.2.2 Risultati	32
4. Conclusioni	36

1. Introduzione

In questo documento, è stato esaminato in dettaglio il comportamento e le prestazioni di varie reti neurali in contesti specifici. Sono state analizzate diverse configurazioni di iperparametri e funzioni di attivazione. L'algoritmo Rprop è stato impiegato per l'aggiornamento dei pesi e l'analisi è stata condotta utilizzando un approccio di cross validation k-fold.

1.1 Traccia

Il progetto è diviso in due sezioni:

- Progettazione e implementazione di una libreria di funzioni per la gestione e la simulazione di reti neurali multi-strato full-connected. La libreria deve essere in grado di simulare la propagazione in avanti di una rete neurale con più strati di nodi interni, consentendo la selezione di qualsiasi funzione di attivazione per ciascun strato. Inoltre, la libreria deve anche implementare l'algoritmo di back-propagation per l'addestramento delle reti neurali multi-strato. Questo deve includere il supporto per diverse funzioni di attivazione dei nodi della rete e la capacità di utilizzare almeno una funzione d'errore tra la somma dei quadrati e la cross-entropy, con l'opzione di applicare o meno la funzione softmax.
- Applicazione della libreria creata tramite l'utilizzo del dataset MNIST per un problema di classificazione delle cifre scritte a mano in 10 categorie. Estrarre un dataset di almeno 10.000 immagini raw del MNIST, creare una rete neurale con un unico strato di nodi interni e addestrarla impiegando l'algoritmo Resilient Backpropagation (RProp). Selezionare gli iper-parametri della RProp e il numero di nodi interni tramite k-fold cross-validation con $k=10$, garantendo un

addestramento efficace e prestazioni valutate accuratamente. Gli altri parametri, come le funzioni di attivazione, devono essere mantenuti costanti.

1.2 Cenni teorici

1.2.1 Rete neurale

Le reti neurali, note anche come Artificial Neural Networks (ANN), sono dei modelli di calcolo basati su un grande numero di unità di calcolo elementari fortemente interconnesse tra di loro, i neuroni. Singolarmente sono in grado di svolgere operazioni elementari e mediante la loro interconnessione riescono a svolgere compiti complessi proprio come a livello biologico. Prendono anche il nome di modelli di connessione. Un neurone è definito da una struttura e da un comportamento specifici.

La struttura comprende diverse connessioni di input, attraverso le quali vengono ricevuti i dati. Ogni connessione porta con sé un valore, generalmente appartenente ai numeri reali (R), indicato come x_i . Il corpo del neurone svolge la computazione ed è composto da un input e un output e gli è associato un valore reale chiamato bias (b). Le connessioni di input sono collegate a pesi specifici, indicati per i -esima connessione come w_i . Inoltre, ogni neurone ha una funzione di output, o funzione di attivazione, che determina il risultato finale.

Il comportamento di un neurone si articola in due passi principali:

- l'input del neurone viene calcolato utilizzando i valori di input, i pesi e il bias secondo la seguente formula

$$z_j = \sum_i w_{ij}x_i + b_j$$

- il valore di output è determinato in base all'input calcolato e alla funzione di attivazione.

$$a_j = \phi(z_j)$$

Questo processo permette al neurone di elaborare i dati e contribuire al funzionamento complessivo della rete neurale.

1.2.2 Feed forward

Una rete neurale multistrato tipica consiste in:

- Strato di input: i neuroni di tale layer ricevono i dati non ancora elaborati in input
- Strati nascosti o hidden layer: dove avviene l'elaborazione dell'informazione
- Strato di output: questi neuroni restituiscono l'output della rete

La propagazione in avanti in una rete neurale multistrato completamente connessa consiste nel passare i dati di input attraverso ogni strato della rete, applicando pesi, bias e funzioni di attivazione, per ottenere l'output finale.

Tale processo si divide in 3 step, di cui uno ricorrente:

1. I valori di input vengono passati ai neuroni del primo strato nascosto
2. Ogni neurone dell'hidden layer calcola il proprio output. Step ricorrente per ogni strato nascosto
3. Gli output dei neuroni dell'ultimo strato nascosto vengono passati ai neuroni nello strato di output, dove viene calcolato l'output finale della rete

1.2.3 Back propagation e discesa del gradiente

Il processo di back propagation o propagazione all'indietro consiste nel calcolare il gradiente della funzione di perdita o loss rispetto ai pesi della rete, utilizzando la discesa del gradiente per aggiornare i pesi e minimizzare l'errore.

La discesa del gradiente è una tecnica di ottimizzazione volta a minimizzare una funzione di perdita o costo aggiornando iterativamente i parametri del modello nella direzione opposta al gradiente della funzione di perdita rispetto a quei parametri

La propagazione indietro si divide in 4 step principali:

1. Viene calcolato l'output della rete tramite un processo di feed forward
2. Scelta una funzione di perdita, viene calcolata la loss, ad esempio tramite la Cross-Entropy
3. Si calcola il gradiente della funzione di perdita rispetto ai pesi della rete utilizzando la regola della catena del calcolo differenziale

a. output layer:

$$\delta^{(o)} = \frac{\partial L}{\partial \hat{y}} \cdot \phi'(z^{(o)})$$

ϕ funzione di attivazione del neurone

$z^{(o)}$ input del neurone nello strato di output

b. hidden layer

$$\delta^{(h)} = (\delta^{(o)} \cdot W^{(o)}) \cdot \phi'(z^{(h)})$$

$W^{(o)}$ pesi tra lo strato nascosto e lo strato di output

$z^{(h)}$ input del neurone nello strato nascosto

4. i pesi vengono aggiornati utilizzando l'algoritmo della discesa del gradiente

$$W_{ij} := W_{ij} - \eta \frac{\partial L}{\partial W_{ij}} \quad \eta \text{ tasso di apprendimento}$$

1.2.4 Resilient propagation

L'algoritmo della Resilient propagation è un algoritmo di ottimizzazione efficace per l'addestramento di reti neurali che migliora la velocità di convergenza rispetto ai metodi tradizionali di discesa del gradiente. Adatta il tasso di apprendimento η in base alla direzione del gradiente della funzione di perdita ed utilizza un fattore di aggiornamento indipendente per ogni peso.

Step principali dell'algoritmo:

1. Viene calcolato il gradiente della funzione di perdita rispetto ai pesi della rete
2. Viene considerato solo il segno del gradiente, non la sua magnitudine

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t), & \text{se } \frac{\partial L}{\partial w_{ij}(t)} > 0 \\ \Delta_{ij}(t), & \text{se } \frac{\partial L}{\partial w_{ij}(t)} < 0 \\ 0, & \text{se } \frac{\partial L}{\partial w_{ij}(t)} = 0 \end{cases}$$

3. I pesi vengono aggiornati
 - a. Se il segno del gradiente attuale è lo stesso del gradiente del passo precedente, il tasso di apprendimento per quel peso viene aumentato.
 - b. Se il segno del gradiente attuale è diverso dal segno del gradiente del passo precedente, il fattore di apprendimento per quel peso viene ridotto.
 - c. Se il gradiente è zero, il tasso di apprendimento rimane invariato.

$$\Delta_{ij}(t) = \begin{cases} \eta^+ \cdot \Delta_{ij}(t-1), & \text{se } \frac{\partial L}{\partial w_{ij}(t-1)} \cdot \frac{\partial L}{\partial w_{ij}(t)} > 0 \\ \eta^- \cdot \Delta_{ij}(t-1), & \text{se } \frac{\partial L}{\partial w_{ij}(t-1)} \cdot \frac{\partial L}{\partial w_{ij}(t)} < 0 \\ \Delta_{ij}(t-1), & \text{se } \frac{\partial L}{\partial w_{ij}(t-1)} \cdot \frac{\partial L}{\partial w_{ij}(t)} = 0 \end{cases}$$

4. Il peso viene aggiornato sottraendo o aggiungendo il tasso di apprendimento, a seconda del segno del gradiente.

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

1.2.5 K-fold cross validation

La k-fold cross validation è una tecnica utilizzata per valutare l'efficacia di un modello di apprendimento e aiuta a garantire che il modello generalizzi bene su dati non visti, riducendo così il rischio di overfitting. Funziona suddividendo l'insieme dei dati

disponibili in k sottoinsiemi, chiamati *fold*, di dimensioni approssimativamente uguali. Il modello viene addestrato sui $k-1$ fold e valutato sul fold rimanente. Questo processo viene ripetuto k volte, in modo che ogni fold venga utilizzato esattamente una volta come set di valutazione. Al termine delle k iterazioni, viene calcolata la media dei risultati ottenuti, come l'accuratezza, il punteggio F1 o qualsiasi altra metrica di performance, per fornire una stima complessiva delle prestazioni del modello.

La k -fold cross validation presenta diversi vantaggi:

- permette un utilizzo completo dei dati, poiché ogni istanza del dataset viene impiegata sia per l'addestramento che per il test, migliorando così l'affidabilità delle stime di performance.
- riduce il bias che può derivare da una suddivisione arbitraria del dataset, poiché ogni istanza viene testata esattamente una volta
- consente di osservare la variabilità delle prestazioni del modello attraverso diverse suddivisioni del dataset, offrendo una visione più completa della sua efficacia.

Un caso estremo di k -fold cross validation è la Leave-One-Out (LOO). In questa variante, il numero di k fold è pari al numero di istanze nel dataset. Ogni iterazione utilizza una singola istanza come test set, mentre tutte le altre istanze vengono utilizzate per l'addestramento. Questo approccio massimizza l'uso dei dati per l'addestramento e fornisce una valutazione molto dettagliata, sebbene possa essere computazionalmente costoso per dataset di grandi dimensioni.

2. Architettura del sistema

L'obiettivo è creare una rete neurale multistrato full-connected, in cui ogni neurone è collegato a tutti i neuroni del livello precedente.

2.1 Tecnologie utilizzate

Nel processo di sviluppo del progetto vengono utilizzate diverse tecnologie.

2.1.1 Python e le sue librerie

Python è un linguaggio di programmazione ad alto livello, interpretato, versatile e molto popolare. La sua sintassi è progettata per essere leggibile e intuitiva, il che lo rende un'ottima scelta sia per i principianti sia per i professionisti. Le librerie in Python sono collezioni di moduli predefiniti e funzioni che offrono funzionalità aggiuntive al linguaggio di base. Le librerie consentono agli sviluppatori di estendere le capacità di Python senza dover scrivere tutto da zero. Uno dei principali motivi per cui Python è così popolare e versatile è l'esistenza di librerie per quasi ogni tipo di applicazione o dominio. Alcune di queste librerie utilizzate nel progetto sono *numpy* e *matplotlib*. NumPy è una libreria open source per il linguaggio di programmazione Python, utilizzata principalmente per il calcolo scientifico e l'analisi dei dati. Fornisce supporto per array multidimensionali e matrici, insieme a una vasta collezione di funzioni matematiche per operare su questi array in modo efficiente. Matplotlib, invece, è una libreria open source per la rappresentazione grafica dei dati.

2.1.2 Github

GitHub è un servizio web che consente agli sviluppatori di lavorare insieme, condividere e collaborare su progetti software in modo efficace e organizzato.

2.1.2 Git LFS

Git LFS è un'estensione per Git che consente agli utenti di gestire file di grandi dimensioni, come immagini, video, audio e dati binari, spostando i file fuori dal normale flusso di Git e memorizzandoli su un server separato. Git LFS memorizza un piccolo riferimento ai file al posto del contenuto effettivo, migliorando le prestazioni e riducendo le dimensioni del repository. In questo progetto, Git LFS è stato utilizzato poiché le dimensioni del dataset superano i 100 MB.

2.1.3 Visual Studio Code

Visual Studio Code (VS Code) è stato utilizzato per la stesura e gestione del codice, sviluppato da Microsoft ed è gratuito oltre a essere altamente popolare. Si tratta di un editor di codice sorgente leggero, flessibile e personalizzabile, progettato principalmente per lo sviluppo di applicazioni software.

2.1.4 Discord

Discord è un applicativo per la gestione di videochiamate ed è stato utilizzato durante tutta la fase del progetto per permettere una programmazione in maniera sincrona.

2.2 Dataset

Il dataset utilizzato per la realizzazione di questo progetto è il MNIST, un ampio archivio di cifre scritte a mano. Questo dataset è ampiamente impiegato nel campo del Machine Learning, in particolare per l'addestramento di sistemi di elaborazione delle immagini. Il dataset MNIST contiene 60.000 immagini per il training e 10.000 immagini per il test, ciascuna rappresentante una cifra scritta a mano appartenente all'insieme {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

L'implementazione e pre-elaborazione di questo dataset è stata realizzata nel file *myDataset.py*.

2.3 Funzionalità richieste

Modificando il codice nel main, è possibile selezionare e personalizzare diversi aspetti della rete neurale. Si può scegliere la funzione di attivazione sia per i nodi nascosti sia per quelli di output, e la funzione di errore, come la Cross-Entropy. Inoltre, è possibile selezionare la funzione di aggiornamento dei pesi, optando per l'algoritmo di Back propagation o quello della Resilient propagation. Oltre a queste scelte, è possibile regolare anche gli iperparametri della rete. Si può impostare il tasso di apprendimento (*eta*), e quelli relativi la Resilient propagation (*eta pos* e *eta neg*), il numero di nodi nello strato interno (*hidden_size*), il numero massimo di epoche da effettuare durante la fase di apprendimento (*n_epoch*).

È possibile effettuare un'operazione di hyperparameter tuning tramite l'algoritmo di *k-fold cross validation* per il quale è possibile scegliere il numero di fold (*k*).

2.4 Strutture dati utilizzate

La rete neurale può essere descritta come un'entità caratterizzata da diversi attributi fondamentali. Questi attributi sono essenziali per comprendere la struttura e il funzionamento della rete stessa.

La struttura dati utilizzata per la rete neurale è il dizionario:

- *'Weights'* → lista dei pesi, sono parametri che determinano l'importanza di ciascun collegamento tra i nodi della rete
- *'Biases'* → lista dei bias, sono valori aggiunti al calcolo dei nodi che consentono di spostare la funzione di attivazione e migliorare l'apprendimento del modello.
- *'ActFunc'* → lista delle funzioni d'attivazione per layer

-
- *'Depth'* → valore di profondità della rete

Alla rete vengono dati in input i seguenti valori:

- *input_size* indica il numero di variabili d'ingresso
- *hidden_size* rappresenta il numero di neuroni nascosti e numero di strati hidden
- *output_size* denota il numero di nodi d'uscita dell'ultimo strato
- *list_act_func* una lista dove vengono elencate le funzioni d'attivazione per layer.

Ogni strato può avere una funzione di attivazione diversa, scelta in base alle necessità del modello e alla natura del problema da risolvere

2.5 Algoritmi utilizzati

In questa sezione, verranno esaminati in dettaglio il funzionamento della Rprop e della k-fold cross validation, analizzando il loro rispettivo pseudocodice. Lo scopo è fornire una comprensione completa e approfondita di come questi due *algoritmi* contribuiscano all'efficienza dell'ottimizzazione nelle reti neurali.

Resilient Propagation

Input: net, X_t, Y_t, X_v, Y_v, err_func, eta_pos, eta_neg, eta, n_epoch, alpha, beta

eta_ij ← eta * net_depth

Y_t_fp ← forwardPropagation(net, X_t)

training_error ← err_func(Y_t_fp, Y_t)

err_train ← inserimento in lista di training_error

if X_v non è None **then**

 Y_v_fp ← forwardPropagation(net, X_v)

 validation_error ← err_func(Y_v_fp, Y_v)

 err_val ← inserimento in lista di validation_error

 stampa a schermo delle informazioni di training per train e valid set

else then

 stampa a schermo delle informazioni di training per train set

end if

while epoch < n_epoch **do**

der_weights, der_biases \leftarrow backPropagation(net, X_t, Y_t, err_funct)

der_w_list \leftarrow inserimento in lista di der_weights

der_b_list \leftarrow inserimento in lista di der_biases

for layer in [0, ..., net_depth - 1] **do**

if epoch > 0 **then**

for n in [0, ..., weight_matrix_raw_dim - 1] **do**

for i in [0, ..., weight_matrix_column_dim - 1] **do**

prod_der_w \leftarrow elemento scorsa epoca der_w_list * elemento
epoca attuale der_w_list

if prod_der_w > 0 **then**

eta_ij \leftarrow min(eta_ij * eta_pos, alpha)

else if prod_der_w < 0 **then**

eta_ij \leftarrow max(eta_ij * eta_neg, beta)

end if

peso della rete \leftarrow peso della rete - (eta_ij *
segno(elemento der_weights))

end for

prod_der_b \leftarrow elemento scorsa epoca der_b_list * elemento epoca
attuale der_b_list

if prod_der_b > 0 **then**

eta_ij \leftarrow min(eta_ij * eta_pos, alpha)

else if prod_der_b < 0 **then**

eta_ij \leftarrow max(eta_ij * eta_neg, beta)

end if

bias della rete \leftarrow bias della rete - (eta_ij * segno(elemento der_biases))

end for

end if

end for

```

Y_t_fp ← forwardPropagation(net, X_t)
training_error ← err_funct(Y_t_fp, Y_t)
err_train ← inserimento in lista di training_error
epoch ← epoch + 1
if X_v non è None then
    Y_v_fp ← forwardPropagation(net, X_v)
    validation_error ← err_funct(Y_v_fp, Y_v)
    err_val ← inserimento in lista di validation_error
    stampa a schermo delle informazioni di training per train e valid set
else then
    stampa a schermo delle informazioni di training per train set
end if
end while
if X_v non è None then
    return err_train, err_val
else then
    return err_train

```

L'algoritmo della Resilient propagation della funzione *trainResilientPropagation* permette il suo utilizzo sia nel caso di addestramento valutato tramite un validation set sia nel caso non lo sia. Lo pseudocodice comincia con la stampa dei valori di perdita e di accuratezza del training e del validation set se presente sulla rete non ancora addestrata dopodichè inizia il ciclo di addestramento di n_epoch epoche. Per ciascun ciclo di training avviene il meccanismo di backpropagation dopodichè per ciascun neurone di ogni livello, a seconda del segno del gradiente, vi è un cambio del valore del tasso di apprendimento η , infine vi è l'aggiornamento dei pesi di ciascun neurone e dei bias per ciascuno strato secondo i propri η . Il ciclo di ciascuna epoca si conclude con la

stampa delle perdite e delle accuratze e la funzione si chiude ritornando le liste contenenti le perdite di train e di validazione per ogni epoca.

Cross validation k fold

Input: X, Y, test_x, test_y, err_funct, net_input_size, net_output_size, list_hidden_size, list_eta_pos, list_eta_neg, eta, k, n_epoch)

combinations \leftarrow combinazione degli elementi di list_hidden_size, list_eta_pos, list_eta_neg

if Y_column_dim è divisibile per k **then**

 partitions \leftarrow lista di array di interi vuoti per per ciascuna partizione

for lab in [0, ..., Y_row_dim -1] **do**

 bool_vector \leftarrow tupla con valore True dove il valore di Y è uguale a lab

 index_vector \leftarrow trova gli indici degli elementi in bool_vector che sono True

 lab_partition \leftarrow divide index_vector in k parti uguali

for sample in [0, lunghezza partitions - 1] **do**

 partition di sample \leftarrow inserimento in lista di lab_partition di sample

end for

end for

for i in [0, ..., k - 1] **do**

 partition di i \leftarrow permutazione casuale partition di i

end for

 count \leftarrow 1

for combination in combinations **do**

 s_acc_train, s_acc_val \leftarrow 0

 net \leftarrow newNetwork(net_input_size, combination[0], net_output_size, list_act_funct)

 stampa informazioni della rete e della partizione

for v in [0, ..., k - 1] **do**

 s_err_train, s_err_val \leftarrow 0

 XV \leftarrow v-esima partizione di X

 YV \leftarrow v-esima partizione di Y

```

train_partition ← copia di partitions
elimina v-esima partizione da train_partition
t_index ← concatena train_partition
XT ← partizione di X con indici in t_index
YT ← partizione di Y con indici in t_index
net ← newNetwork(net_input_size, combination[0], net_output_size,
list_act_funct)
err_train, err_val ← trainResilientPropagation(net, XT, YT, XV, YV, err_funct,
combination[1], combination[2], eta, n_epoch)

stampa di controllo sul numero di partizioni
s_err_train ← s_err_train + ultimo valore di err_train
s_err_val ← s_err_val + ultimo valore di err_val
acc_train ← testAccuracy(net, X, Y)
acc_test ← testAccuracy(net, test_x, test_Y)
s_acc_train ← s_acc_train + acc_train
s_acc_test ← s_acc_test + acc_test
stampa dell'accuracy del train e del test set

end for

list_err_train ← inserimento in lista di s_err_train/k
list_err_val ← inserimento in lista di s_err_val/k
avg_acc_train ← s_acc_train/k
avg_acc_test ← s_acc_test/k
stampa la combinazione degli iperparametri con l'accuracy del train e test set
list_acc_train ← inserimento in lista di avg_acc_train
list_acc_test ← inserimento in lista di avg_acc_test
count ← count + 1

end for

stampa del train e val loss e dell'accuracy di train e test

return list_err_train, list_err_val, list_acc_train, list_acc_test, combinations

```

else then

Solleva un'eccezione con relativo messaggio

end if

Questo pseudocodice riassume le operazioni principali della funzione *crossValidationKFold* suddividendo i dati in k fold, addestrando la rete su diverse combinazioni di iperparametri e calcolando i valori medi per le k partizioni di perdita e accuratezza per ogni combinazione.

3. Analisi del codice e delle prestazioni

Questa sezione si occuperà di spiegare le varie funzioni e algoritmi utilizzati nel codice.

3.1 Specifiche del sistema

3.1.1 loadDataset

```
1 def loadDataset(datapath='C:/Users/Pietro20/Desktop/'):
2     train_set = np.loadtxt(datapath + "mnist_train.csv", delimiter=',', skiprows=1)
3     test_set = np.loadtxt(datapath + "mnist_test.csv", delimiter=',', skiprows=1)
4
5     train_X_norm = (train_set[:, 1:]) / 255
6     test_X_norm = (test_set[:, 1:]) / 255
7
8     train_Y = train_set[:, 0]
9     test_Y = test_set[:, 0]
10
11     train_lab = oneHotEnc(train_Y)
12     test_lab = oneHotEnc(test_Y)
13
14     return train_X_norm.transpose(), test_X_norm.transpose(), train_lab.transpose(), test_lab.transpose()
```

Carica il dataset MNIST dai file CSV presenti nel percorso specificato. I dati vengono normalizzati dividendo i valori dei pixel per 255. La funzione restituisce i dati di addestramento e test normalizzati (senza la colonna delle etichette) e le etichette

codificate in formato one-hot. I dati e le etichette sono trasposti per essere compatibili con la forma di input delle funzioni di rete neurale successive.

3.1.2 oneHotEnc

```
1 def oneHotEnc(labels_set):
2     row = len(labels_set)
3     col = 10
4
5     sparse_matrix = np.zeros((row, col), dtype=int)
6     labels = labels_set.astype(int)
7
8     for i in range(row):
9         sparse_matrix[i, labels[i]] = 1
10
11     return sparse_matrix
```

Converte un array di etichette numeriche in una matrice di codifica one-hot. Crea una matrice di zeri con un numero di righe pari al numero di etichette e 10 colonne (una per ciascuna delle classi da 0 a 9). Imposta a 1 la posizione corrispondente alla classe corretta per ogni etichetta. La funzione restituisce la matrice codificata one-hot.

3.1.3 Activation Functions

```
1 def identity(matrix, der=0):
2     if der == 0:
3         return matrix
4     else:
5         return matrix, 1
6
7 def tanh(matrix, der=0):
8     res = np.tanh(matrix)
9     if der == 0:
10        return res
11    else:
12        return res, 1 - res*res
13
14 def sigm(matrix, der=0):
15     res = 1 / (1 + np.exp(-matrix))
16     if der == 0:
17        return res
18    else:
19        return res, res*(1-res)
```

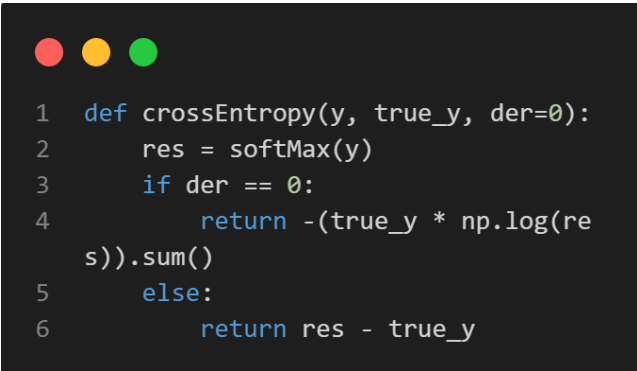
Sono state implementate tre diverse funzioni di attivazione:

- identity
- tahn
- sigmoid

Ogni funzione accetta una matrice come input e una flag '*der*' inizializzata di default a 0.

- *der* = 0, la funzione restituisce semplicemente l'output della funzione di attivazione applicata all'input
- *der* != 0, la funzione restituisce l'output insieme alla derivata della funzione di attivazione rispetto all'input

3.1.4 Error Functions

A code block with a dark background and light blue text. It contains a Python function definition for crossEntropy. The function takes three arguments: y, true_y, and der (with a default value of 0). It first calls softmax(y) and stores the result in res. Then, it checks if der is 0. If so, it returns the negative sum of true_y multiplied by the log of res. Otherwise, it returns res minus true_y.

```
1 def crossEntropy(y, true_y, der=0):  
2     res = softmax(y)  
3     if der == 0:  
4         return -(true_y * np.log(re  
5 s)).sum()  
6     else:  
7         return res - true_y
```

La funzione `crossEntropy` calcola l'errore tra le predizioni e le etichette vere usando la cross-entropy. Se *der* è 0, restituisce il valore della loss. Se *der* è diverso da 0, restituisce la derivata della loss rispetto all'input, utile per la fase di backpropagation.

```
1 def softmax(y):
2     yExp = np.exp(y - y.max(0))
3     res = yExp / sum(yExp)
4     return res
```

La funzione softmax trasforma l'input in una distribuzione di probabilità, normalizzando gli esponenziali degli input.

3.1.5 newNetwork

```
1 def newNetwork(input_size, hidden_size, output_size, list_act_func=[]):
2     sigma = 0.1
3     biases = []
4     weights = []
5     act_func = []
6     prev_layer = input_size
7
8     if np.isscalar(hidden_size):
9         hidden_size = [hidden_size]
10
11     for layer in hidden_size:
12         biases.append(sigma * np.random.normal(size = [layer, 1]))
13         weights.append(sigma * np.random.normal(size = [layer, prev_layer]))
14         prev_layer = layer
15     act_func = setActFunc(len(weights), list_act_func)
16     biases.append(sigma * np.random.normal(size = [output_size, 1]))
17     weights.append(sigma * np.random.normal(size = [output_size, prev_layer]))
18     act_func.append(act.identity)
19     net = {'Weights':weights, 'Biases':biases, 'ActFun':act_func, 'Depth':len(weights)}
20
21     return net
```

Crea una nuova rete neurale con i pesi e i bias inizializzati casualmente, utilizzando la distribuzione normale scalata da un parametro sigma. La rete può avere una singola dimensione per il livello nascosto o un elenco di dimensioni per più livelli nascosti.

La funzione restituisce un dizionario che rappresenta la rete neurale tramite le chiavi *Weights*, *Biases*, *ActFun*, *Depth*.

3.1.6 setActFunc

```
1 def setActFunc(depth, list_act_func, act_def=act.tanh):
2     if not list_act_func:
3         return [act_def for _ in range(0, depth)]
4     elif len(list_act_func) < depth:
5         act_list_1 = [list_act_func[i] for i in range(0, len(list_act_func))]
6         act_list_2 = [act_def for _ in range(len(list_act_func), depth)]
7         return act_list_1 + act_list_2
8     else:
9         raise Exception("Exception: Too many item in the activation function list\n")
```

Imposta le funzioni di attivazione per i vari strati della rete neurale. Se l'utente non fornisce una lista di funzioni di attivazione, viene utilizzata una funzione di attivazione predefinita (tanh). Se la lista fornita ha meno elementi del necessario, la funzione predefinita viene utilizzata per i rimanenti strati. Se la lista ha troppi elementi, viene sollevata un'eccezione.

3.1.7 Funzioni ausiliarie

```
1 def getInfo(net):
2     hidden_layers = net['Depth'] - 1
3     print("Depth network:", net["Depth"])
4     print("Number of input neurons:", net["Weights"][0].shape[1])
5     print("Number of hidden layers:", hidden_layers)
6     print("Number of hidden neurons:", [net["Weights"][layer].shape[0] for layer in
7     range(0, hidden_layers)])
8     print("Number of output neurons:", net["Weights"][(net["Depth"] - 1)].shape[0])
9     print("Weights shape:", [net["Weights"][i].shape for i in range(0, (1 + hidden_
10 layers + 1) - 1)])
11     print("Activation functions:", [(net["ActFun"][i]).__name__ for i in range(0, n
12 et["Depth"])]))
```

Vi sono quattro funzioni getter ausiliarie le quali restituiscono informazioni sulla rete passata in input, quali, la struttura di tale rete, la lista dei bias, la lista dei pesi e la lista delle funzioni ausiliarie.

3.1.8 forwardPropagation

```
1 def forwardPropagation(net, X):
2     B = getBiasesList(net)
3     W = getWeightsList(net)
4     AF = getActFunList(net)
5     d = net['Depth']
6     res = X
7
8     for layer in range(d):
9         ith_layer = np.matmul(W[layer], res) + B[layer]
10        res = AF[layer](ith_layer)
11
12    return res
```

Propagazione in avanti della rete neurale calcolando l'output a partire dagli input forniti e applicando i pesi, i bias e le funzioni di attivazione a ciascun livello.

3.1.9 trainForwardPropagation

```
1 def trainForwardPropagation(net, X):
2     B = getBiasesList(net)
3     W = getWeightsList(net)
4     AF = getActFunList(net)
5     d = net['Depth']
6     ith_layer = []
7     res = []
8     der_act = []
9     res.append(X)
10
11    for layer in range(d):
12        ith_layer.append(np.matmul(W[layer], res[layer]) + B[layer])
13        a, da = AF[layer](ith_layer[layer], 1)
14        der_act.append(da)
15        res.append(a)
16
17    return res, der_act
```

Propagazione in avanti della rete che calcola anche le derivate delle funzioni di attivazione, necessarie per la fase di backpropagation.

Restituisce gli output di ciascun livello e le derivate delle funzioni di attivazione.

3.1.10 backPropagation

```
1 def backPropagation(net, X, Y_true, err_funct):
2     W = getWeightsList(net)
3     d = net['Depth']
4     X_list, X_der_list = trainForwardPropagation(net, X)
5     delta_list = []
6     delta_list.insert(0, err_funct(X_list[-1], Y_true, 1) * X_der_list[-1])
7
8     for layer in range(d-1, 0, -1):
9         delta = X_der_list[layer-1] * np.matmul(W[layer].transpose(), delta_list[0])
10        delta_list.insert(0, delta)
11
12    weight_der = []
13    bias_der = []
14
15    for layer in range(0, d):
16        der_w = np.matmul(delta_list[layer], X_list[layer].transpose())
17        weight_der.append(der_w)
18        bias_der.append(np.sum(delta_list[layer], 1, keepdims=True))
19
20    return weight_der, bias_der
```

Backpropagation per calcolare le derivate dei pesi e dei bias rispetto alla funzione di errore. Restituisce le derivate dei pesi e dei bias per ogni strato della rete.

3.1.11 trainBackPropagation

```
1 def trainBackPropagation(net, X_t, Y_t, X_v, Y_v, err_funct, n_epoch=1, eta=0.1):
2     err_train = []
3     err_val = []
4     Y_t_fp = forwardPropagation(net, X_t)
5     training_error = err_funct(Y_t_fp, Y_t)
6     err_train.append(training_error)
7     Y_v_fp = forwardPropagation(net, X_v)
8     validation_error = err_funct(Y_v_fp, Y_v)
9     err_val.append(validation_error)
10
11     d = net['Depth']
12     epoch = 0
13
14     print("Epoch:", epoch, "Training error:", training_error,
15           "Accuracy Training:", networkAccuracy(Y_t_fp, Y_t),
16           "Validation error:", validation_error,
17           "Accuracy Validation:", networkAccuracy(Y_v_fp, Y_v))
18
19     while epoch < n_epoch:
20         der_weights, der_biases = backPropagation(net, X_t, Y_t, err_funct)
21
22         for layer in range(d):
23             net['Weights'][layer] = net['Weights'][layer] - eta * der_weights[layer]
24             net['Biases'][layer] = net['Biases'][layer] - eta * der_biases[layer]
25
26         Y_t_fp = forwardPropagation(net, X_t)
27         training_error = err_funct(Y_t_fp, Y_t)
28         err_train.append(training_error)
29         Y_v_fp = forwardPropagation(net, X_v)
30         validation_error = err_funct(Y_v_fp, Y_v)
31         err_val.append(validation_error)
32
33         epoch += 1
34
35         print("Epoch:", epoch, "Training error:", training_error,
36               "Accuracy Training:", networkAccuracy(Y_t_fp, Y_t),
37               "Validation error:", validation_error,
38               "Accuracy Validation:", networkAccuracy(Y_v_fp, Y_v), end='')
39         print('\n', end='')
40     print()
41
42     return err_train, err_val
```

Esegue l'addestramento della rete neurale utilizzando la backpropagation per un numero specificato di epoche. Durante l'allenamento, aggiorna i pesi e i bias e calcola gli errori di addestramento e di validazione ad ogni epoca. Stampa anche l'accuratezza della rete sul set di addestramento e di validazione.

3.1.12 trainResilientPropagation

```
1 def trainResilientPropagation(net, X_t, Y_t, X_v=None, Y_v=None, err_func=ef.crossEntropy, eta_pos=1.2,
2   eta_neg=0.5, eta=0.1, n_epoch=1, alpha=0.001, beta=0.0001):
3     err_train = []
4     err_val = []
5     der_w_list = []
6     der_b_list = []
7
8     d = net['Depth']
9     epoch = 0
10    eta_ij = eta * d
11
12    Y_t_fp = forwardPropagation(net, X_t)
13    training_error = err_func(Y_t_fp, Y_t)
14    err_train.append(training_error)
15
16    if X_v is not None:
17        Y_v_fp = forwardPropagation(net, X_v)
18        validation_error = err_func(Y_v_fp, Y_v)
19        err_val.append(validation_error)
20
21        print("Epoch:", epoch, "Training error:", training_error,
22              "Accuracy Training:", networkAccuracy(Y_t_fp, Y_t),
23              "Validation error:", validation_error,
24              "Accuracy Validation:", networkAccuracy(Y_v_fp, Y_v))
25    else:
26        print("Epoch:", epoch, "Training error:", training_error,
27              "Accuracy Training:", networkAccuracy(Y_t_fp, Y_t))
28
29    while epoch < n_epoch:
30        der_weights, der_biases = backPropagation(net, X_t, Y_t, err_func)
31        der_w_list.append(der_weights)
32        der_b_list.append(der_biases)
33
34        for layer in range(d):
35            if epoch > 0:
36                neurons = net['Weights'][layer].shape
37                for n in range(neurons[0]):
38                    for i in range(neurons[1]):
39                        prod_der_w = der_w_list[epoch-1][layer][n][i] * der_w_list[epoch][layer][n][i]
40                        if prod_der_w > 0:
41                            eta_ij = min(eta_ij * eta_pos, alpha)
42                        elif prod_der_w < 0:
43                            eta_ij = max(eta_ij * eta_neg, beta)
44
45                        net['Weights'][layer][n][i] -= eta_ij * np.sign(der_weights[layer][n][i])
46
47                        prod_der_b = der_b_list[epoch-1][layer][n] * der_b_list[epoch][layer][n]
48                        if prod_der_b > 0:
49                            eta_ij = min(eta_ij * eta_pos, alpha)
50                        elif prod_der_b < 0:
51                            eta_ij = max(eta_ij * eta_neg, beta)
52
53                        net['Biases'][layer][n] -= eta_ij * np.sign(der_biases[layer][n])
54
55        Y_t_fp = forwardPropagation(net, X_t)
56        training_error = err_func(Y_t_fp, Y_t)
57        err_train.append(training_error)
58
59    epoch += 1
```

```

1  if X_v is not None:
2      Y_v_fp = forwardPropagation(net, X_v)
3      validation_error = err_func(Y_v_fp, Y_v)
4      err_val.append(validation_error)
5
6      print("Epoch:", epoch, "Training error:", training_error,
7            "Accuracy Training:", networkAccuracy(Y_t_fp, Y_t),
8            "Validation error:", validation_error,
9            "Accuracy Validation:", networkAccuracy(Y_v_fp, Y_v), end='')
10 else:
11     print("Epoch:", epoch, "Training error:", training_error,
12           "Accuracy Training:", networkAccuracy(Y_t_fp, Y_t), end='')
13     print('\r', end='')
14     print()
15     if X_v is not None:
16         return err_train, err_val
17     else:
18         return err_train

```

Esegue l'addestramento della rete neurale utilizzando la Resilient propagation (RProp). Durante l'allenamento, aggiorna i pesi e i bias in base ai gradienti e ai parametri di incremento e decremento, calcolando anche l'errore e l'accuratezza sul set di addestramento e di validazione.

3.1.13 networkAccuracy & testAccuracy

```

1  def networkAccuracy(Y, Y_true):
2      tot = Y_true.shape[1]
3      true_positive = 0
4      for i in range(0, tot):
5          true_label = np.argmax(Y_true[:, i])
6          y_label = np.argmax(Y[:, i])
7          if true_label == y_label:
8              true_positive += 1
9      return true_positive / tot

```

Calcola l'accuratezza della rete neurale confrontando l'output predetto con le etichette vere. Restituisce la percentuale di predizioni corrette.

```

1 def testAccuracy(net, test_X, test_Y):
2     net_Y = forwardPropagation(net, test_X)
3     return networkAccuracy(net_Y, test_Y)

```

Calcola l'accuratezza della rete neurale su un set di test richiamando la funzione network accuracy.

3.1.14 crossValidationKFold

```

1 def crossValidationKFold(X, Y, test_X, test_Y, err_func, net_input_size, net_output_size, list_hidden_size=[], list_eta_pos=[], list_eta_neg=[],
eta=0.1, k=10, n_epoch=50, write_on_file=False):
2     combinations = list(product(list_hidden_size, list_eta_pos, list_eta_neg))
3     samples_dim = Y.shape[1]
4     list_err_train = []
5     list_err_val = []
6     list_acc_train = []
7     list_acc_test = []
8     if (samples_dim % k) == 0:
9         d = Y.shape[0]
10        partitions = [np.ndarray(0,int) for i in range(k)]
11        for lab in range(d):
12            bool_vector=(Y.argmax(0)==lab)
13            index_vector=np.argwhere(bool_vector==True).flatten()
14            lab_partition=np.array_split(index_vector,k)
15            for sample in range(len(partitions)):
16                partitions[sample]=np.append(partitions[sample],lab_partition[sample])
17        for i in range(k):
18            partitions[i]=np.random.permutation(partitions[i])
19        count = 1
20        n_combination = len(combinations)
21        for combination in combinations:
22            s_acc_train = 0
23            s_acc_test = 0
24            net = newNetwork(input_size=net_input_size, hidden_size=combination[0], output_size=net_output_size, list_act_func=[])
25            getInfo(net)
26            print('\nIperparametri selezionati:\n-hidden size ', combination[0], '\n-eta+ ', combination[1], '\n-eta- ', combination[2], '\n')
27            print(count, '/', n_combination, ' Combinazioni iperparametri\n')
28            for v in range(k):
29                s_err_train = 0
30                s_err_val = 0
31                XV = X[:, partitions[v]]
32                YV = Y[:, partitions[v]]
33                train_partition = partitions.copy()
34                del train_partition[v]
35                t_index = np.concatenate(train_partition)
36                XT = X[:, t_index]
37                YT = Y[:, t_index]
38                net = newNetwork(input_size=net_input_size, hidden_size=combination[0], output_size=net_output_size, list_act_func=[])
39                err_train, err_val = trainResilientPropagation(net,
40                                                                XT, YT,
41                                                                XV, YV,
42                                                                err_func,
43                                                                eta_pos=combination[1],
44                                                                eta_neg=combination[2],
45                                                                eta=eta,
46                                                                n_epoch=n_epoch)

```

```

1 print('\n', v+1, '/', k, ' Partizioni analizzate\n')
2
3         s_err_train += err_train[-1]
4         s_err_val += err_val[-1]
5         acc_train = testAccuracy(net, X, Y)
6         acc_test = testAccuracy(net, test_X, test_Y)
7         s_acc_train += acc_train
8         s_acc_test += acc_test
9         print('Accuracy train set partizione:', acc_train)
10        print('Accuracy test set partizione:', acc_test)
11        print()
12
13        avg_err_train = s_err_train/k
14        avg_err_val = s_err_val/k
15        list_err_train.append(avg_err_train)
16        list_err_val.append(avg_err_val)
17        avg_acc_train = s_acc_train/k
18        avg_acc_test = s_acc_test/k
19        print('Combinazione numero', count, 'accuracy train set:', avg_acc_train)
20        print('Combinazione numero', count, 'accuracy test set:', avg_acc_test)
21        print()
22        list_acc_train.append(avg_acc_train)
23        list_acc_test.append(avg_acc_test)
24        if write_on_file:
25            append_to_file(valuation_path_1, count, avg_acc_train, avg_acc_test, avg_err_train, avg_err_val)
26        count += 1
27
28        print('List loss train:', list_err_train)
29        print('List loss val:', list_err_val)
30        print('List avg acc train: ', list_acc_train)#
31        print('List avg acc test: ', list_acc_test)#
32
33        return list_err_train, list_err_val, list_acc_train, list_acc_test, combinations
34
35    else:
36        raise Exception("Exception: each fold must be the same size")

```

Esegue la cross-validation K-Fold per testare le diverse combinazioni di iperparametri per la rete neurale. Divide i dati in k partizioni, addestra la rete su k-1 partizioni e valida sulla partizione rimanente.

Restituisce gli errori di addestramento e di validazione e le accuratèzze medie per ogni combinazione di iperparametri.

3.1.15 myPlot

```
1 def myPlot(list_err_train, list_err_val, list_acc_train, list_acc_test, combinations):
2     x_plot_lab = []
3     for c in combinations:
4         x_plot_lab.append(str(c))
5
6     fig, axs = plt.subplots(1, 2)
7     axs[0].plot(x_plot_lab, list_err_train, color='r', label='Train loss')
8     axs[0].plot(x_plot_lab, list_err_val, color='g', label='Validation loss')
9     axs[0].set_title("Loss")
10
11    axs[1].set_title("Accuracy")
12    axs[1].plot(x_plot_lab, list_acc_train, color='r', label='Train accuracy')
13    axs[1].plot(x_plot_lab, list_acc_test, color='g', label='Test accuracy')
14    fig.tight_layout()
15    plt.show()
```

Traccia i grafici delle loss e delle accuratze di addestramento e di validazione per diverse combinazioni di iperparametri utilizzando matplotlib.

3.2 Discussione dei risultati

Per l'addestramento del dataset MNIST è stata presa in esame una rete neurale artificiale full connected con uno strato di input e un unico hidden layer aventi come funzione di attivazione la tangente iperbolica ed uno strato di output con la funzione di attivazione identità ed è stato utilizzato un algoritmo di Resilient propagation come algoritmo di addestramento.

È stata effettuata una ricerca dei migliori iperparametri con un approccio di cross validation k-fold a ricerca esaustiva tra le combinazioni delle liste di iperparametri forniti all'algoritmo che comprendevano i parametri di aggiornamento dei tassi di apprendimento η_+ e η_- e il numero di neuroni dello strato nascosto.

3.2.1 Set up sperimentale

Durante i test effettuati, molti iperparametri sono stati mantenuti costanti, tra cui:

- *net_input_size* = 784: la dimensione di un elementi di input
- *net_output_size* = 10: numero di nodi d'uscita
- *err_funct* = crossEntropy: funzione d'errore
- *eta* = 0.001: tasso di apprendimento per il gradiente standard
- *k* = 10: numero di fold
- *n_epoch* = 100: numero di epoche
- *act_funct* = tanh, tanh, identity
- *alpha* = 0.001
- *beta* = 0.0001

Per quanto riguarda i dati analizzati, sono stati sfruttati tutti gli elementi del dataset MINST, in particolare:

- 60000 elementi per il training set (di cui, grazie al k-fold con $k=10$, 54000 elementi sono stati utilizzati per il training e 6000 per la validation a ogni iterazione)
- 10000 elementi per il test set

In totale, sono stati utilizzati 70000 elementi.

Sono stati impiegati valori non costanti per i diversi parametri chiave del modello. In particolare, il numero dei neuroni dello strato nascosto, η_+ (η_+) e η_- (η_-) sono stati variati per analizzare il loro impatto sulle prestazioni complessive del sistema. Questo approccio, noto come *grid search*, ci ha permesso di esplorare una gamma più ampia di configurazioni e di identificare le combinazioni di parametri migliori. Il grid search è spesso utilizzato per garantire una buona copertura delle possibili configurazioni.

3.2.2 Risultati

Questi sono i risultati presentati sotto forma tabellare. In particolare, oltre ai vari parametri quali neuroni interni, eta+ ed eta- avremo la train accuracy, la test accuracy, il train loss e il validation loss:

- *train accuracy*: la 'train accuracy', ovvero l'accuratezza sul set di addestramento, è la percentuale di correttezza delle predizioni del modello sui dati di addestramento. In parole semplici, rappresenta quanto bene il modello si comporta sui dati con cui è addestrato.
- *test accuracy*: la 'test accuracy', ovvero l'accuratezza sul test set, è la percentuale di correttezza delle predizioni del modello sui dati di test, che non sono stati usati durante l'addestramento. In parole semplici, indica quanto bene il modello generalizza a nuovi dati non visti prima.
- *train loss*: la 'train loss', ovvero la perdita sul set di addestramento, è una misura che rappresenta quanto il modello si discosta dai risultati desiderati sui dati di addestramento. Un valore più basso di perdita, generalmente, indica una migliore performance del modello sui dati di addestramento.
- *validation loss*: la 'validation loss', ovvero la perdita sul set di validazione, è simile alla train loss, ma calcolata sui dati di validation. Sostanzialmente, serve per valutare la performance del modello su dati non visti durante l'addestramento e aiuta a rilevare problemi come l'overfitting, cioè quando il modello si adatta troppo ai dati di addestramento e non generalizza bene.

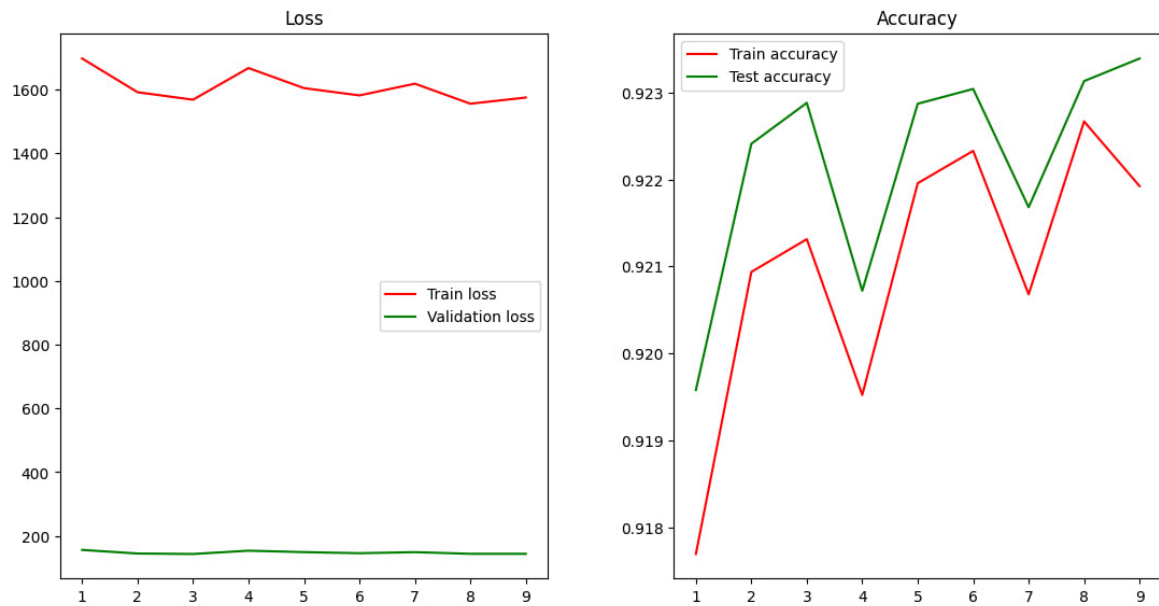
I valori delle metriche sono le medie dei risultati ottenuti in ogni iterazione delle varie partizioni del k-fold cross validation.

#	Hidden Neurons	$\eta+$ (eta+)	$\eta-$ (eta-)	Train Accuracy %	Test Accuracy %	Train Loss	Validation Loss
1	64	1.1	0.6	91.76	91.95	1697.33	156.28
2	64	1.1	0.7	92.09	92.24	1591.53	144.99
3	64	1.1	0.8	92.13	92.28	1568.11	143.55
4	64	1.2	0.6	91.95	92.07	1667.43	153.86
5	64	1.2	0.7	92.19	92.28	1604.48	149.38
6	64	1.2	0.8	92.23	92.30	1581.57	145.96
7	64	1.3	0.6	92.06	92.16	1618.38	149.28
8	64	1.3	0.7	92.26	92.31	1555.53	144.11
9	64	1.3	0.8	92.19	92.33	1574.93	144.09
10	128	1.1	0.6	92.46	92.47	1490.29	137.85
11	128	1.1	0.7	92.63	92.61	1452.22	131.11
12	128	1.1	0.8	92.81	92.85	1393.45	127.79
13	128	1.2	0.6	92.59	92.56	1453.94	131.41
14	128	1.2	0.7	92.84	92.80	1369.17	124.72
15	128	1.2	0.8	93.32	93.17	1287.86	117.69
16	128	1.3	0.6	92.66	92.68	1378.18	125.89
17	128	1.3	0.7	92.99	92.91	1321.20	122.71
18	128	1.3	0.8	93.50	93.33	1266.03	117.75
19	256	1.1	0.6	92.00	92.02	1457.07	135.59

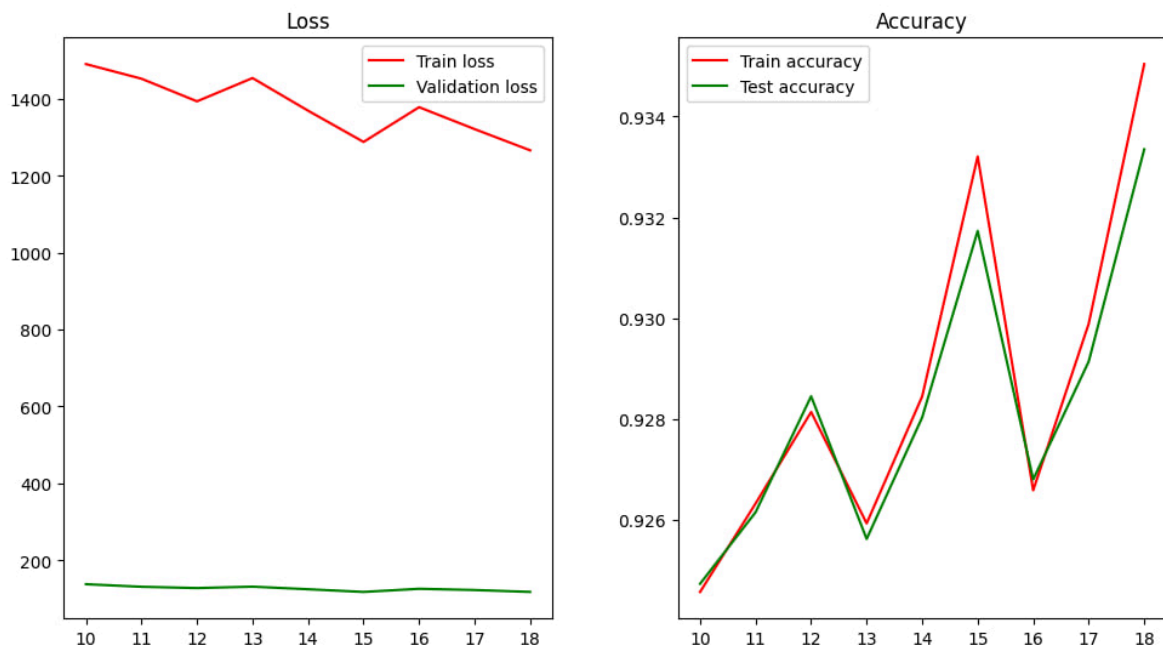
#	Hidden Neurons	$\eta+$ (eta+)	$\eta-$ (eta-)	Train Accuracy %	Test Accuracy %	Train Loss	Validation Loss
20	256	1.1	0.7	92.01	92.01	1474.98	136.26
21	256	1.1	0.8	92.02	92.02	1490.40	137.34
22	256	1.2	0.6	92.16	92.10	1562.12	145.80
23	256	1.2	0.7	92.04	91.91	1524.58	139.29
24	256	1.2	0.8	92.35	92.31	1460.23	137.97
25	256	1.3	0.6	92.31	92.31	1439.26	132.47
26	256	1.3	0.7	92.44	92.34	1385.47	129.65
27	256	1.3	0.8	92.76	92.64	1374.29	128.95

Per le diverse combinazioni, l'accuratezza sul train varia da 91.76% a 93.50%, mentre l'accuratezza sul test varia da 91.95% a 93.33%. Sia l'accuratezza sul train che sul test sono abbastanza stabili e simili, che sta a significare che non ci sono problemi di overfitting o underfitting tra le varie configurazioni dei parametri. Notiamo che sia il train loss che il validation loss diminuiscono con l'aumento di eta- da 0.6 a 0.8 per un eta+ fisso. Quando eta+ aumenta da 1.1 a 1.3, sia l'accuratezza sul train che sul test mostrano lievi miglioramenti. La massima accuratezza sul test viene ottenuta dalla configurazione 18 e il minimo valore del validation loss viene ottenuta dalla configurazione 15 (entrambe in grassetto). Sebbene la combinazione di eta+ = 1.2 e eta- = 0.8 con 128 neuroni nascosti fornisca il minimo validation loss, la configurazione con eta+ = 1.3 e eta- = 0.8 con 128 neuroni nascosti offre il miglior equilibrio tra accuratezza e valore di perdita. Di seguito sono riportati i grafici divisi per numero di neuroni nascosti aventi il numero di configurazione sull'asse delle x e loss e accuracy sull'asse delle y, rispettivamente:

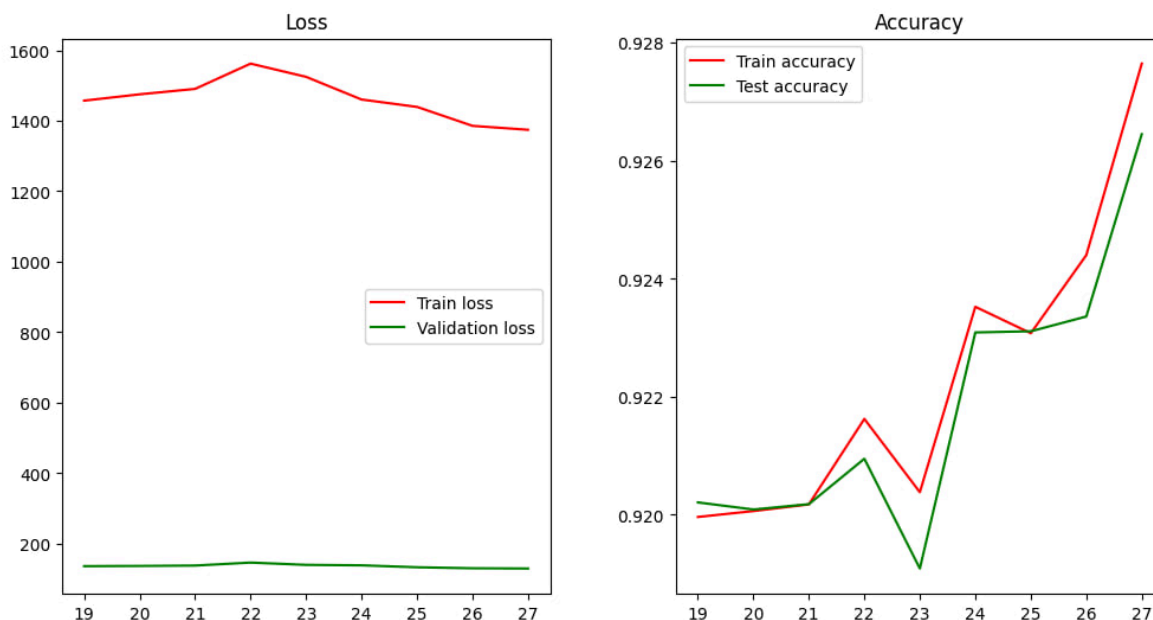
- Combinazione da 1 a 9 con 64 neuroni nascosti:



- Combinazione da 10 a 18 con 128 neuroni nascosti:



- Combinazione da 19 a 27 con 256 neuroni nascosti:



4. Conclusioni

In conclusione notiamo che il miglior valore per il numero di neuroni nascosti sia 128 e che 100 epoche siano sufficienti a non far decrementare la curva di apprendimento. Quindi, sono state prese in considerazione le combinazioni numero 15 (128, 1.2, 0.8) e numero 18 (128, 1.3, 0.8) e sono stati allenati 10 modelli per ciascuna combinazione con l'utilizzo del dataset completo, senza l'utilizzo di un validation set. Le metriche di accuratezza medie risultanti sono:

#	Train Accuracy %	Test Accuracy %
15	93.38	93.27
18	93.56	93.42

Il modello migliore rimane quello della combinazione 18.