UNIVERSITÀ DEGLI STUDI DI NAPOLI

FEDERICO II

Machine Learning Mod. B

Progetto FF-ML

Prof. Roberto Prevete

De Lucia Gianluca	N97000311
Mennella Carlo	N97000319
Martino Ciro Michele	N97000308

A.A. 2019/2020

Indice

1. Introduzione 3
2. Fase di training 4
   2.1 Forward Propagation 4
      2.1.1 Funzioni di attivazione 6
   2.2 Backward Propagation 7
   2.3 Aggiornamento dei pesi 9
      2.3.1 Discesa del gradiente 9
      2.3.2 RProp 10
   2.4 Calcolo dell'errore 11
      2.4.1 Sum of squares 11
      2.4.2 Cross Entropy 11
      1.4.3 Criterio di fermata: Generalization Loss 12
3. Fase di Testing 13
4. Implementazione parte facoltativa 14
5. Subroutine 15
6. Sperimentazioni con RProp 23
7. Conclusione sulle sperimentazioni 33
8. Codice Matlab 37
9. Riferimenti Bibliografici 43

1. Introduzione

Parte A

Progettazione ed implementazione di funzioni per simulare la propagazione in avanti di una rete neurale multi-strato con almeno: due strati di pesi, con la sigmoide come funzione di output dei nodi interni e l'identità come funzione di output dei nodi di output.

(FACOLTATIVO: permettere all'utente di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di output per ciascun strato)

Progettazione ed implementazione di funzioni per la realizzazione della backpropagation per reti neurali multi-strato con almeno: due strati di pesi, con la sigmoide come funzione di output dei nodi interni e l'identità come funzione di output dei nodi di output, con la somma dei quadrati o la cross-entropy con soft-max come funzione di errore.

(FACOLTATIVO: permettere all'utente di realizzare la back-propagation con più di uno strato di nodi interni, con qualsiasi funzione di output per ciascun strato e con qualsiasi funzione di errore derivabile rispetto all'output).

Parte B

(Difficoltà media) Si consideri come input le immagini raw del dataset mnist. Si ha, allora, un problema di classificazione a C classi, con C=10. Si estragga opportunamente un dataset globale di N coppie, e lo si divida opportunamente in training, validation e test set (ad esempio, 2000 per il training set, 500 per il validation set, 500 per il test set). Si fissi la resilient backpropagation (RProp) come algoritmo di aggiornamento dei pesi(aggiornamento batch). Si studi l'apprendimento di una rete neurale (ad esempio epoche necessarie per l'apprendimento, andamento dell'errore su training e validation set, accuratezza sul test) facendo variare il numero di strati interni da 1 a 5 confrontando il caso in cui si utilizza come funzione di output dei nodi la sigmoide con quello in cui si usa come funzione di output dei nodi la ReLu (max(0, a)). Provare diverse scelte del numero dei nodi per gli strati interni. Se è necessario, per questioni di tempi computazionali e spazio in memoria, si possono ridurre (ad esempio dimezzarle) le dimensioni delle immagini raw del dataset mnist (ad esempio utilizzando in matlab la funzione imresize).

2. Fase di training

Di seguito sono riportati, in maniera sintetica, tutti i passi dal punto di vista teorico che definiscono la fase di training della rete neurale svolta sia nella Parte A del progetto che nella Parte B.

2.1 Forward Propagation

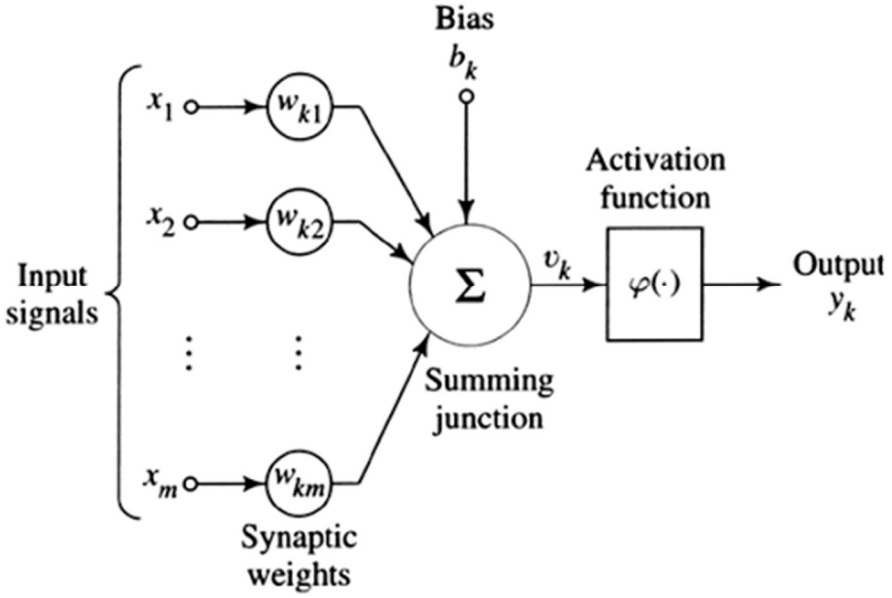


Figura 1: Neurone artificiale

La fase di forward propagation consiste nel "propagare in avanti" il calcolo delle uscite dei nodi di una rete neurale feed forward multistrato (dove per strati della rete si esclude quello di input, quindi solo quelli intermedi e quello di output). In particolare, al primo passo vengono calcolati i nodi al primo strato, grazie a questi ultimi valori è possibile calcolare quelli al secondo strato, fino a calcolare quelli all'ultimo strato (ovvero i nodi di output), ottenendo un valore per ogni nodo della rete neurale.

Di seguito è riportata la procedura per il calcolo di un'uscita di un solo nodo. Il valore di uscita *a* del neurone artificiale di McCullach e Pitts (ad

esempio il neurone rappresentato in Figura 1) viene calcolato nel seguente modo:

$$a = f\left(\sum_{i=0}^{d} w_i x_i + b\right)$$

In tale funzione, w_i sono i pesi delle connessioni presenti tra gli ingressi e il neurone, x_i sono i valori degli ingressi, d è il numero di ingressi e b è il bias del neurone, ovvero un valore aggiuntivo usato nel calcolo del neurone. In particolare, se si pone $b = x_0$ e $x_0 = 1$, aggiungendo una connessione w_0 tra b e il neurone, allora l'uscita del neurone può essere scritta equivalentemente nel seguente modo:

$$a = f\left(\sum_{i=0}^{d} w_i x_i\right) \qquad (1)$$

è possibile osservare come tale meccanismo può essere utilizzato nel calcolo di tutte le uscite di tutti i nodi di un'intera rete neurale.

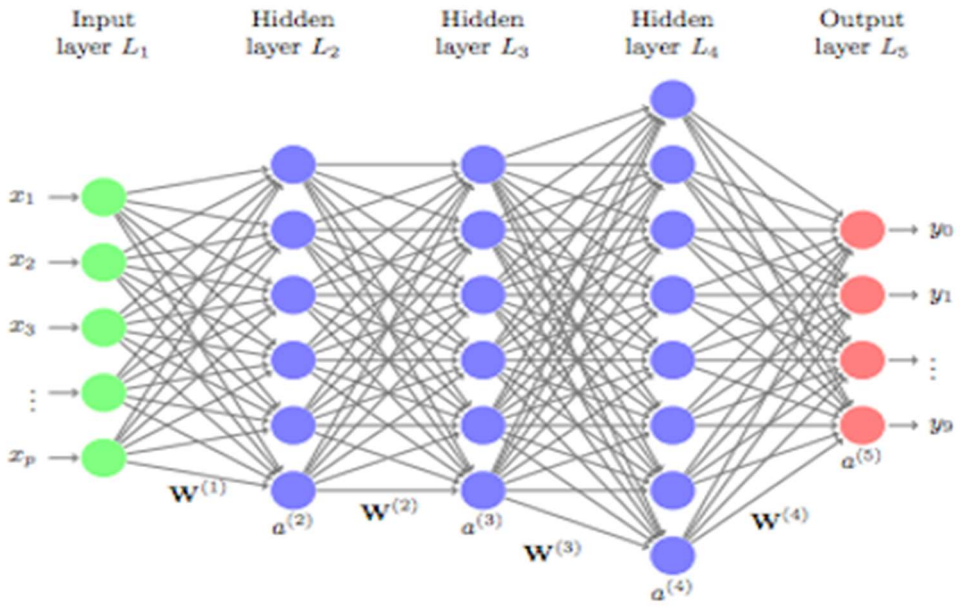


Figura 2: Rete neurale FF-ML

Iterando opportunamente la procedura, vengono così calcolati tutti i nodi di una rete neurale feed forward multistrato (o più brevemente rete neurale

FF-ML) con l strati. Data una rete neurale FF-ML (ad esempio come quella mostrata in Figura 2) con le seguenti caratteristiche:

- d - dimensione del vettore di input,
- $(x_1,x_2,...,x_d)$ - vettore di input,
- l - numero di strati,
- $(m_1,m_2,...,m_L)$ - numero di nodi per ogni strato,
- w_{ij} - peso associato alla connessione che va dal neurone j dello, strato l-1 al neurone i dello strato l,
- f la funzione di attivazione; iterando la formula del neurone artificiale, la feed forward calcola i valori di uscita per ogni nodo della rete. Per il primo strato (denominato con (1) nella formula) viene eseguita la seguente operazione che calcola tutti i nodi $z_i^{(1)}$ di tale strato:

$$z_i^{(1)} = f\left(\sum_{j=1}^{d} w_{ij}^{(1)} x_j + b_i^{(1)}\right)$$

Per il secondo strato (denominato con (2) nella formula) viene eseguita la seguente operazione che calcola tutti i nodi $z_i^{(2)}$ di tale strato:

$$z_i^{(2)} = f\left(\sum_{j=1}^{m_1} w_{ij}^{(2)} z_j^{(1)} + b_i^{(2)}\right)$$

In generale, per un generico strato l (denominato con (l) nella formula) si ha che la seguente operazione che calcola tutti i nodi $z_i^{(l)}$ dello strato:

$$z_i^{(l)} = f\left(\sum_{j=1}^{m_{l-1}} w_{ij}^{(l)} z_j^{(l-1)} + b_i^{(l)}\right) \quad (2)$$

Vengono calcolate tutte le uscite dei nodi della rete neurale applicando la stessa formula "in avanti", seguendo l'ordine topologico stabilito dalle connessioni (da qui il nome propagazione in avanti o forward propagation).

2.1.1 Funzioni di attivazione

Sono state utilizzate due diverse funzioni di attivazione, ovvero la sigmoide e la ReLU.

Sigmoide: La funzione della sigmoide è la seguente:

$$s(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

Possiamo notare come la derivata di tale funzione sia estremamente semplice, questo è uno dei motivi per cui rende la sigmoide una funzione molto utilizzata nell'ambito delle reti neurali:

$$s'(x) = s(x) * (1 - s(x)) \tag{4}$$

ReLU: La funzione della Rectified Linear Unit (ReLU) è la seguente:

$$relu(x) = max(0,x) \tag{5}$$

Anche in questo caso ci troviamo di fronte ad una funzione la cui derivata è abbastanza semplice:

$$relu'(x) = \begin{cases} 0, & \text{se } x \leq 0 \\ 1, & \text{altrimenti} \end{cases} \tag{6}$$

Tale funzione è molto utilizzata nell'ambito del deep learning, poichè risolve il problema del vanishing gradient, ed inoltre i costi computazionali si riducono notevolmente. Più avanti (in Sezione 6) viene effettuato un confronto tra la ReLU e la sigmoide, in cui vengono discussi tali punti, in relazione ad una serie di esperimenti effettuati.

2.2 Backward Propagation

Nel machine learning molti classificatori si basano su una procedura iterativa che si pone l'obiettivo di minimizzare una data funzione di errore. In questo modo, cercando di minimizzare in più passi l'errore commesso nel classificare alcuni elementi del dataset, si cerca di migliore l'algoritmo di apprendimento, affinchè quest'ultimo "sbagli" il meno possibile nel classificare. Un approccio simile viene utilizzato nel modello delle reti neurali. Dopo la fase di forward propagation, viene calcolata una funzione di errore $\boldsymbol{E}(\underline{\boldsymbol{w}})$, dipendente da tutti i Back propagation

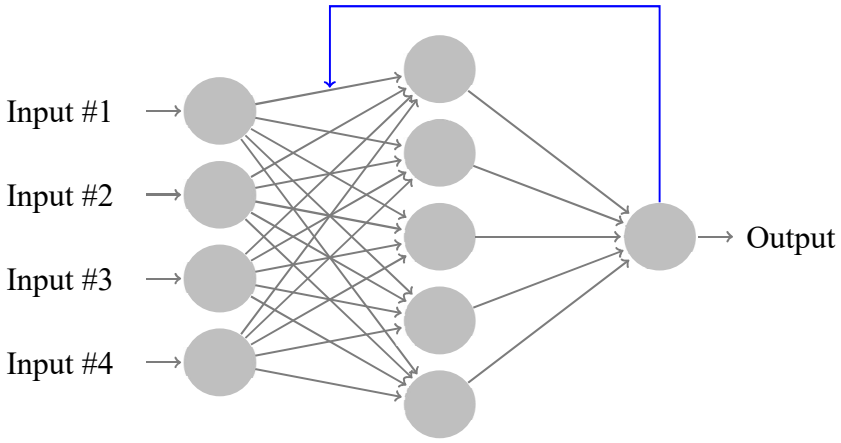


Figura 3: Rete neurale FF-ML e back propagation

pesi delle connessioni $\underline{w}$, la si cerca di minimizzare rispetto ai pesi, allo scopo di ottenere quei pesi "ottimali" **w∗** indispensabili per l'apprendimento, ovvero calcolando:

$$w^* = \underset{\underline{w}}{\operatorname{argmin}}\, E(\underline{w})$$

Di seguito è presentata la backward propagation (o più semplicemente back propagation), una procedura che ha lo scopo di calcolare la derivata dell'errore rispetto ai pesi in maniera iterativa "all'indietro", a partire dall'ultimo strato (quello di output), fino ad arrivare al primo strato (come rappresentato concettualmente in Figura 3). Così facendo in una fase successiva è possibile, grazie al calcolo delle derivate, calcolare i pesi w^*. La back propagation è un metodo molto efficiente per il calcolo delle derivate della funzione di errore rispetto ai pesi di una rete. Tale procedura vale per:

- Reti neurali FF-ML
- Funzioni di attivazione derivabili
- Funzioni di errore derivabili rispetto all'output della rete

La funzione di errore E è data da tutti gli errori $E^{(n)}$, per ogni elemento n del dataset, ovvero $E = \sum_{n=1}^{N} E^{(n)}$. Di conseguenza la sua derivata è data da:

$$\frac{dE}{dw_{ij}} = \sum_{n=1}^{N} \frac{dE^{(n)}}{dw_{ij}} \qquad (7)$$

In particolare, per il calcolo di tale derivata allora si vuole calcolare $\frac{dE^{(n)}}{dw_{ij}}$. Si può dimostrare che:

$$\frac{dE^{(n)}}{dw_{ij}} = \delta_j z_i \tag{8}$$

Dove $\delta_j = \frac{dE^{(n)}}{da_j}$. Il calcolo di tali delta cambia in base al tipo di nodi.

- Per i nodi di output

$$\delta_k = f'(a_k)\frac{dE^{(n)}}{dy_k} \tag{9}$$

- Per i nodi interni

$$\delta_j = f'(a_j)\sum_k w_{kj}\delta_k \tag{10}$$

Si ha quindi una formula definita induttivamente il cui caso base è dato dal calcolo dei delta per l'ultimo strato (strato di output). Risalendo "a ritroso", ovvero per il passo induttivo, vengono calcolati i delta per il primo strato, da cui il nome di back propagation. Una volta calcolati i delta è possibile moltiplicarli per le uscite dei nodi, come in formula 7, ottenendo le derivate rispetto ai pesi.

2.3 Aggiornamento dei pesi

Una volta ottenute le derivate della funzione di errore rispetto ai pesi $\frac{dE^{(n)}}{dw_{ij}}$, è possibile minimizzare tale funzione $E(\underline{w})$ (a più variabili) facendo variare i suoi ingressi, dati dai pesi delle connessioni.

2.3.1 Discesa del gradiente

Il seguente metodo, ovvero la discesa del gradiente, è sensibile ai minimi locali della funzione e permette di effettuare l'aggiornamento dei pesi desiderato. L'idea è quella di partire da un certo peso w_{ij}, aumentarlo se ci si trova alla sinistra del minimo (**funzione decrescente**), altrimenti diminuirlo viceversa, se si sta alla destra del minimo (**funzione crescente**), il comportamento desiderato è riassunto di seguito:

$$\begin{cases} \text{Incrementare il peso } w_{ij}, & \text{se } \frac{dE^{(n)}}{dw_{ij}} < 0 \\ \text{Decrementare il peso } w_{ij}, & \text{se } \frac{dE^{(n)}}{dw_{ij}} > 0 \end{cases}$$

Sia η un parametro chiamato learning rate, tale che $0 < \eta < 1$, allora si definisce l'aggiornamento di un generico peso w_{ij} attraverso la discesa del gradiente viene effettuato attraverso la seguente assegnazione:

$$w_{ij} \leftarrow w_{ij} - \eta \left(\frac{dE^{(n)}}{dw_{ij}} \right)$$

In generale per tutti i pesi $\underline{w}$ si ha che:

$$\underline{w} \leftarrow \underline{w} - \eta \left(\nabla E \right)$$

Dove ∇E è il gradiente della funzione, ovvero l'insieme di tutte le derivate parziali della funzione di errore E rispetto ai pesi w_{ij}. Inoltre, l'aggiornamento dei pesi può essere eseguito nei seguenti modi:

- **Online learning**
  I pesi w_{ij} vengono aggiornati per ogni elemento del training set
- **Batch learning**
  I pesi w_{ij} vengono aggiornati sommando i gradienti calcolati per ogni elemento del training set al termine di un'epoca

2.3.2 RProp

La RProp, che sta per Resilient Backpropagation, è un metodo utilizzato per l'apprendimento di una rete neurale, nello specifico usato per l'aggiornamento dei pesi. Si tratta di una versione euristica del metodo della discesa del gradiente. Il peso all'epoca t cambia esclusivamente in base ad un valore $\Delta w_{ij}^{(t)}$, quest'ultimo viene calcolato nel seguente modo:

$$\Delta w_{ij}{}^{(t)} = \begin{cases} -\Delta_{ij}{}^{(t)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} > 0 \\ +\Delta_{ij}{}^{(t)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} < 0 \\ 0, & \text{altrimenti} \end{cases} \tag{11}$$

Dove $\frac{dE}{dw_{ij}}{}^{(t)}$ denota la derivata dell'errore rispetto al peso w_{ij} all'epoca t. Quindi non resta che determinare i valori di $\Delta_{ij}^{(t)}$. Questi valori dipendono dalla derivata dell'errore all'epoca precedente e da quella all'epoca attuale, in particolare si ottengono nel seguente modo:

$$\Delta_{ij}{}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}{}^{(t-1)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} * \frac{dE}{dw_{ij}}{}^{(t-1)} > 0 \\ \eta^- * \Delta_{ij}{}^{(t-1)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} * \frac{dE}{dw_{ij}}{}^{(t-1)} < 0 \\ \Delta_{ij}{}^{(t-1)}, & \text{altrimenti} \end{cases} \tag{12}$$

dove η^+ ed η^- sono degli "amplificatori" della variazione del peso, per questi inoltre vale $0 < \eta^- < 1 < \eta^+$ (nel paper [1] è stato verificato che euristicamente fissando $\eta^+ = 1.2$ ed $\eta^- = 0.5$ si ottengono buoni amplificatori). Inoltre devono essere fissati i valori di Δ_0 (nel paper [1] sono stati fissati a 0.1) e i valori di **upper bound** (Δ_{max}) e **lower bound** (Δ_{min}) di tali variazioni dei pesi, in particolare si ha che $\Delta_{max} = 50$ e $\Delta_{min} = 1e^{-6}$ [1].

2.4 Calcolo dell'errore

Nel progetto, nonostante sia possibile inserire qualunque funzione di errore, sono stati effettuati gli esperimenti su due funzioni di errore, dipendenti da y, gli output della rete neurale, e $\underline{t}$, i target effettivi. Durante il processo di training, è necessario capire quale configurazione di rete risulta essere quella migliore. Per fare ciò viene utilizzato un insieme, chiamato **validation set**, sul quale viene calcolato l'errore rispetto ai target. La rete migliore risulta essere quella sul cui viene ottenuto l'errore minore utilizzando, appunto, il validation set. Sono mostrate di seguito le funzioni di errore implementate per un singolo elemento del dataset.

2.4.1 Sum of squares

Una di queste è la sum of squares, che è solitamente utilizzata per problemi di regressione ed è data da:

$$\frac{\sum_j (y_j - t_j)^2}{2} \tag{13}$$

2.4.2 Cross Entropy

Un'altra funzione di errore è la cross entropy, la quale è solitamente utilizzata per problemi di classificazione ed è data da:

$$-\sum_j t_j log(s(y_j)) \tag{14}$$

Softmax. Quest'ultima fa uso della funzione di **softmax** $s(\underline{y})$, la quale è una funzione che trasforma le componenti di un vettore in probabilità, in particolare applicando la softmax la somma delle componenti è pari ad 1. Essa, rispetto alle uscite, è calcolata nel seguente modo:

$$\frac{e^{y_i}}{\sum_{k=1}^{N} e^{y_k}} \qquad (15)$$

In particolare, è possibile osservare che tale funzione non è stabile numericamente e potrebbe presentare dei casi in cui la macchina restituisce risultati indesiderati una volta applicata. Così facendo è stata adottata per gli esperimenti una variante della softmax, denominata la **softmax stabile**, che è equivalente alla prima ma stabile numericamente. La softmax stabile è ricavabile semplicemente a partire dalla softmax moltiplicando e dividendo il numeratore e il denominatore per una costante ***C***, quindi si hanno i seguenti passi:

$$\begin{aligned}
s(y_i) &= \frac{e^{y_i}}{\sum_{k=1}^{N} e^{y_k}} && \text{(Per la definizione 15)} \\
&= \frac{Ce^{y_i}}{C\sum_{k=1}^{N} e^{y_k}} && \text{(Moltiplicando e dividendo per C)} \\
&= \frac{e^{log(C)}e^{y_i}}{e^{log(C)}\sum_{k=1}^{N} e^{y_k}} && \text{(Ponendo } C = e^{log(C)}\text{)} \\
&= \frac{e^{y_i+log(C)}}{\sum_{k=1}^{N} e^{y_k+log(C)}} && \text{(Per più proprietà degli esponenziali)}
\end{aligned}$$

Ponendo $log(C) = -max(y)$, tale versione, essendo stabile numericamente a differenza della prima, è stata implementata nel progetto. Viene effettuata tale assegnazione poiché in questo modo gli esponenti avranno valori compresi nell'intervallo $(-\infty,0)$, evitando in questo modo di raggiungere un overflow.

1.4.3 Criterio di fermata: Generalization Loss

Nell'addestramento della rete neurale, si è deciso di utilizzare un criterio di stop. In particolare è stato utilizzato il criterio della **generalization loss**. L'addestramento viene terminato quando vale:

$$\left|\left(100 * \left(\frac{emin}{eval} - 1\right)\right)\right| > threshold \qquad (16)$$

3. Fase di Testing

In tale fase viene eseguita la forward propagation su tutti gli elementi del test set e si controlla quale tra i nodi di output presenta il valore maggiore. La classificazione di un'immagine, da parte della rete neurale, è corretta se la classe assegnata dalla rete all'immagine corrisponde a quella a cui effettivamente l'immagine appartiene. In tale fase viene calcolata l'accuracy sul test set come il rapporto tra le immagini classificate correttamente e il numero di immagini.

4. Implementazione parte facoltativa

Oltre ad aver sviluppato la parte A, sono stati sviluppati anche i relativi punti facoltativi. In particolare, la funzione che crea la rete accetta due vettori:

- **VettoreStrati**
  Ovvero il numero di nodi per ogni strato (compreso quello di input).
- **VettoreFunzioni**
  Ovvero le funzioni di attivazione per ogni strato.

In questo modo il programma può creare una rete con qualsiasi tipo di configurazione di nodi, e con qualsiasi funzione di attivazione.
Inoltre, il programma può eseguire l'addestramento utilizzando qualsiasi funzione di errore. In particolare, per quanto riguarda le funzioni di attivazione e le funzioni di errore, è stata creata una apposita struttura per ognuna di esse. Tale struttura contiene un puntatore alla funzione stessa, ed un puntatore alla sua derivata.
Per dettagli implementati andare alla sezione 5.

5. Subroutine

Sono riportate di seguito tutte le subroutine presenti nel progetto.

loadMNIST

Input:
- *Filename1*: Path del file che contiene le immagini MNIST.

- *Filename2*: Path del file che contiene le label MNIST.

Output:
- *Out*: Una struttura contenente due matrici, la prima di dimensione Nx784 (dove N è il numero delle immagini contenute nel file), la seconda di dimensione Nx1 (contenente le etichette).

Descrizione:
Il file delle immagini viene trasformato in una matrice in cui ogni immagine 28x28 viene rappresentata da una riga (l'immagine viene quindi vista come un array monodimensionale).

relu

Input:
- Nessun parametro.

Output:
- Struttura ReLU: Una struttura contenente due puntatori a funzione, che sono la ReLU e la derivata della ReLU.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

identità

Input:
- Nessun parametro.

Output:
- Struttura identità: Una struttura contenente due puntatori a funzione, che sono l'identità e la derivata dell'identità.

Descrizione:

Tale funzione ritorna la struttura sopra descritta.

sigmoide

Input:
- Nessun parametro.

Output: - Struttura sigmoide: Una struttura contenente due puntatori a funzione, che sono la sigmoide e la derivata della sigmoide.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

softmax

Input:
- X: parametro della funzione.

Output:
- Softmax(x): l'input a cui è stata applicata la funzione di softmax stabile.

Descrizione:
La softmax stabile viene calcolata come

createNetwork

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato della rete.
- *vettoreFunzioni*: vettore contenente le funzioni di attivazioni per ogni strato di neuroni.
- *Pesi*: un numero reale tramite il quale si definisce l'intervallo di appartenenza dei pesi della rete.

Output:
- *Net*: Una struttura contenente tutte le proprietà relative alla rete: i pesi per ogni collegamento tra nodi, i bias per ogni nodo, il numero di strati interni della rete, le funzioni di attivazione per ogni strato di nodi, l'output della rete.

Descrizione:
Questa funzione crea una rete feed-forward full connected multi-layers in base ai parametri passati in input (numero di strati, funzioni di attivazione, inizializzazione dei pesi).

CrossEntropy

Input:
- Nessun parametro

Output:
- Struttura crossEntropy: una struttura contenente due puntatori a funzione, i quali puntano alla funzione e alla derivata della crossEntropy. Tale struttura, inoltre, contiene un altro campo, ovvero un puntatore alla funzione che deve essere applicata all'ultimo strato durante la forward propagation, cioè la funzione identità.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

SumOfSquares

Input: - Nessun parametro

Output:
- Struttura crossEntropy: una struttura contenente due puntatori a funzione, i quali puntano alla funzione e alla derivata della SumOfSquares. Tale struttura, inoltre, contiene un altro campo, ovvero un puntatore alla funzione che deve essere applicata all'ultimo strato durante la forward propagation, cioè la softmax.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

forwardPropagation

Input:
- *net*: rete neurale creata con la funzione create Network(...).
- *input*: una matrice contenente le immagini del training set (viste come vettore).
- *FunErr*: è la funzione di errore utilizzata per l'addestramento.

Output:
- net: la stessa rete neurale data in ingresso in cui sono stati aggiornati gli output.
- *Z*: matrice contenente gli output della rete in base all'input.

Descrizione:
Questa funzione, partendo dagli input iniziali (immagini viste come vettori), calcola gli output di ogni strato utilizzando come funzione di attivazione quella che viene specificata all'interno della rete neurale. All'ultimo strato viene applicata una funzione, oltre all'identità, che dipende dal tipo di funzione di errore. Nel caso della cross entropy viene applicata la softmax.

backPropagation

Input:
- *net*: rete neurale.
- *output*: output della rete.
- *targets*: valori attesi degli output.
- *funErr*: struttura della funzione di errore che si vuole usare (ricordiamo che la struttura contiene sia la funzione di errore che la sua derivata).

Output:
- *delta*: un cell array nel quale nella posizione i-esima vi sono i delta calcolati per lo strato i-esimo.

Descrizione:
La funzione parte con il calcolare i delta dell'ultimo strato di nodi, per poi utilizzarli nel calcolo dei delta degli strati superiori.

calcolaDerivate

Input:
- *net*: rete neurale.
- *delta*: delta dei nodi della rete.
- *Input*: immagini di input (viste come vettori).

Output:
- *derW*: un cell array in cui nella posizione i-esima vi sono le derivate dei pesi dello strato i-esimo.
- *derB*: un cell array in cui nella posizione i-esima vi sono le derivate dei bias dello strato i-esimo.

Descrizione:
Vengono calcolate e derivate dei pesi e dei bias tramite i valori dei delta (dati in ingresso alla funzione) e dei valori dei nodi della rete.

aggiornaPesi

Input:
- *net*: rete neurale.
- *derW*: derivate dei pesi.
- *derB*: derivate dei bias.
- *Eta*: rappresenta il learning rate.

Output:
- *net*: rete in cui vi sono i pesi aggiornati.

Descrizione:
La funzione aggiorna tutti i pesi della rete, utilizzando il metodo della discesa del gradiente, tenendo conto del learning rate dato in ingresso (parametro eta).

testing

Input:
- *net*: la rete neurale addestrata.
- *tset*: il test set sul quale calcolare l'accuracy.

Output:
- *accuracy*: Valore dell'accuratezza.

Descrizione:
Effettua il testing sulla rete neurale addestrata valutando l'accuracy sul test set.

initVariazioni

Input:
- *net*: rete neurale.
- *var*: variazione.

Output:
- *varW*: un cell array in cui nella posizione vi sono le variazioni dei pesi per ogni strato.
- *varB*: un cell array in cui nella posizione vi sono le variazioni dei bias per ogni strato.

Descrizione:
Le variazioni vengono inizializzate tramite il parametro var dato in ingresso. Sia le variazioni dei pesi che dei bias vengono inizializzate allo stesso modo.

rprop

Input:
- *net*: rete neurale.
- *derW*: derivate dei pesi.
- *derB*: derivate dei bias.
- *oldDerW*: derivate dei pesi all'epoca precedente.
- *oldDerB*: derivate dei bias all'epoca precedente.
- *varW*: variazione per i pesi.
- *varB*: variazione per i bias.
- *etaP*: valore di eta+.
- *etaN*: valore di eta-.

Output:
- *net*: rete con pesi e bias aggiornati.
- *varW*: nuove variazioni calcolate per i pesi in base al segno del prodotto delle derivate oldDerW e derW.
- *varB*: nuove variazioni calcolate per i bias in base al segno del prodotto delle derivate oldDerB e derB.
- *oldDerW*, *OldDerB*: contengono le derivate dei pesi e dei bias, così da utilizzarle per il calcolo dell'eta alla prossima epoca.

Descrizione:
La funzione calcola il segno del prodotto tra oldDerW e derW, per capire l'andamento di tale funzione. In base al segno ottenuto e ai valori etaP e

etaN, aggiorna le variazioni che saranno effettuate sui pesi. Si noti che le variazioni aggiornate sono sempre comprese nell'intervallo [1e-6, 50]. Viene eseguito lo stesso procedimento per i bias, e vengono aggiornate oldDerW e oldDerB in modo da usarle correttamente durante l'epoca successiva.

TrainingBatch

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato.
- *vettoreFunzioni*: vettore contenente le funzioni da utilizzare per ogni strato.
- *Pesi*: range di valori nel quale sono contenuti i pesi.
- *funErr*: struttura della funzione di errore che si vuole utilizzare (che contiene la funzione d'errore e la sua derivata).
- *training Set*: struttura contenente sia le immagini che le labels che si vogliono usare come training set.
- *validationSet*: struttura contenente sia le immagini che le label che si vogliono usare come validation set.
- *nEpoche*: numero di epoche che si vuole utilizzare.
- *Eta*: rappresenta il learning rate.
- *etaP*: valore di eta+.
- *etaN*: valore di eta-.
- *Variation*: un reale per inizializzare le variazioni per la rprop.

Output:
- *bestNet*: restituisce la rete migliore trovata (in termini di funzione di errore).

Descrizione:
La funzione crea una rete feed-forward full connected multi-layer, che viene addestrata utilizzando i parametri dati in input. Il tipo di aggiornamento usato per i pesi è il batch. Lo scopo di questa funzione è di restituire la migliore rete trovata durante il learning.

ParteA_ TrainingOnline

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato.
- *vettoreFunzioni*: vettore contenente le funzioni da utilizzare per ogni strato.
- *Pesi*: range di valori nel quale sono contenuti i pesi.
- *funErr*: struttura della funzione di errore che si vuole utilizzare.

- *training Set*: struttura contenente sia le immagini che le labels che si vogliono usare come training set.
- *validationSet*: struttura contenente sia le immagini che le label che si vogliono usare come validation set.
- *nEpoche*: numero di epoche che si vuole utilizzare.

- *Eta*: rappresenta il learning rate.

Output:
- *bestNet*: restituisce la rete migliore trovata (in termini di funzione di errore).
- soglia: soglia per il criterio di generalization loss.

Descrizione:
La funzione crea una rete feed-forward full connected multi-layer, che viene addestrata utilizzando i parametri dati in input. Viene effettuato un online learning. Lo scopo di questa funzione è di restituire la migliore rete trovata durante l'addestramento.

ParteA_ TrainingRProp

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato.
- *vettoreFunzioni*: vettore contenente le funzioni da utilizzare per ogni strato.
- *Pesi*: range di valori nel quale sono contenuti i pesi.
- *funErr*: struttura della funzione di errore che si vuole utilizzare.
- training Set: struttura contenente sia le immagini che le labels che si vogliono usare come training set.
- *validationSet*: struttura contenente sia le immagini che le label che si vogliono usare come validation set.
- *nEpoche*: numero di epoche che si vuole utilizzare.

- *Eta*: rappresenta il learning rate.

Output:
- *bestNet*: restituisce la rete migliore trovata (in termini di funzione di errore).

- *soglia*: soglia per il criterio di generalization loss.

Descrizione:
La funzione crea una rete feed-forward full connected multi-layer, che viene addestrata utilizzando i parametri dati in input. Il tipo di aggiornamento dei pesi è il batch. Lo scopo di questa funzione è di restituire la migliore rete trovata durante l'addestramento

6. Sperimentazioni con RProp

In questa sezione si fa riferimento alla parte B del progetto.In particolare vengono effettuati degli addestramenti utilizzando l'RProp come metodo di aggiornamento dei pesi. Alla prima epoca però, si è deciso di utilizzare il metodo della discesa del gradiente, in modo tale da non dover inizializzare le derivate dei pesi e dei biases.
Per quanto riguarda le sperimentazioni effettuate, è stato preso in considerazione il dataset MNIST. Sono state quindi caricate le immagini e mappate su un vettore da 784 elementi. A questo punto possiamo ben capire come le varie configurazioni della rete neurale che viene addestrata, siano tutte del tipo: 784 - X1 - ... – Xn -10.
Sono poi stati settati diversi parametri, in particolare:
Lunghezza training set: 50000
Lunghezza validation set: 10000
Lunghezza test set: 10000
η per la discesa del gradiente: 0,00001
η^+: 1,2
η^-: 0,5
Delta iniziali per l'RProp: 0,001
Inizializzazione dei pesi: compresi nell'intervallo $(-0.01, 0.01)$.

Si è deciso di inserire i risultati ottenuti dagli esperimenti in apposite tabelle, in cui sono presenti valori di accuracy e di errore minimo sul validation set. In particolare nella **Tabella 1** e nel **Grafico 1** vengono mostrati i valori di accuracy ottenuti utilizzando il test set e i valori dell'errore minimo registrato sul validation set e sul training set.

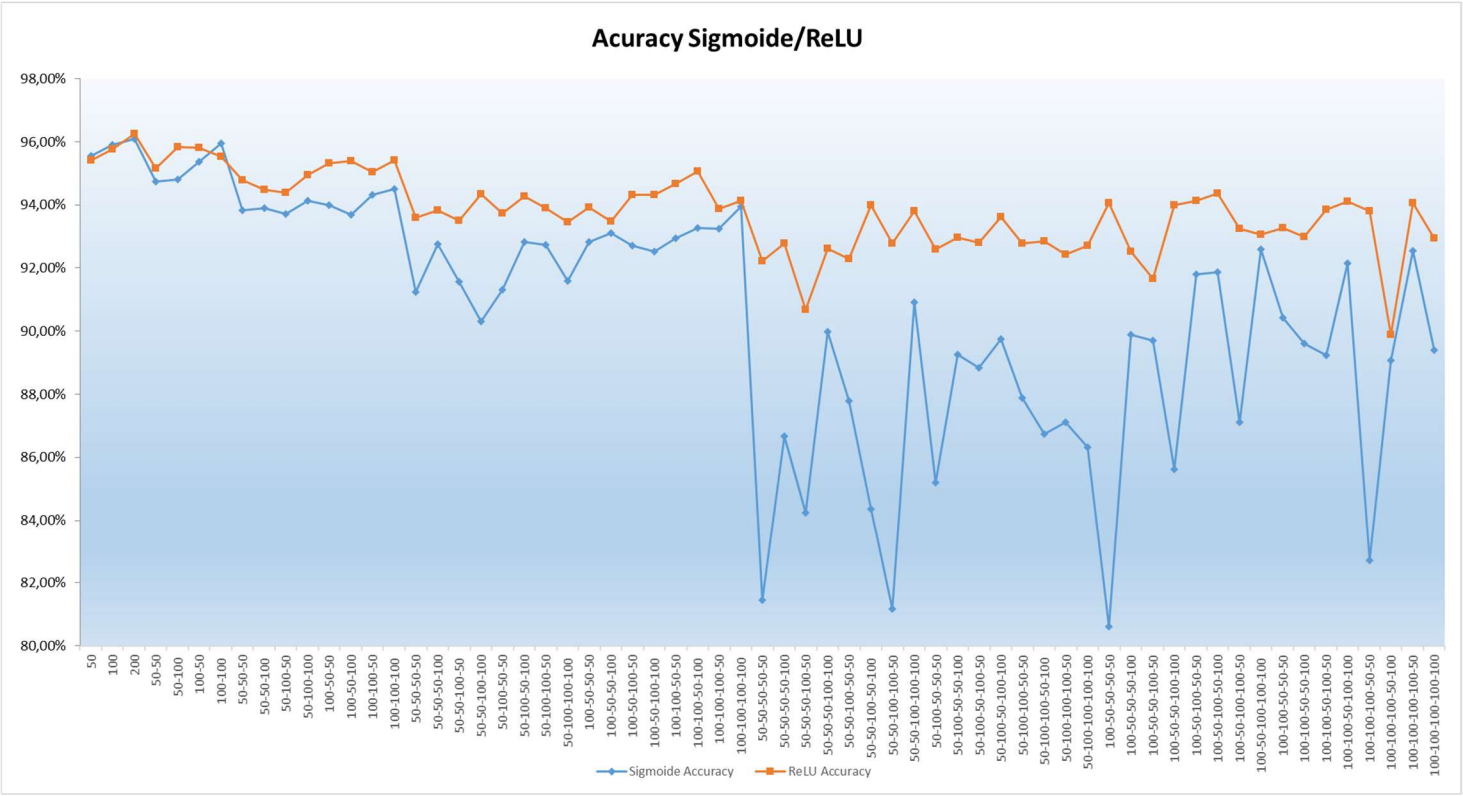
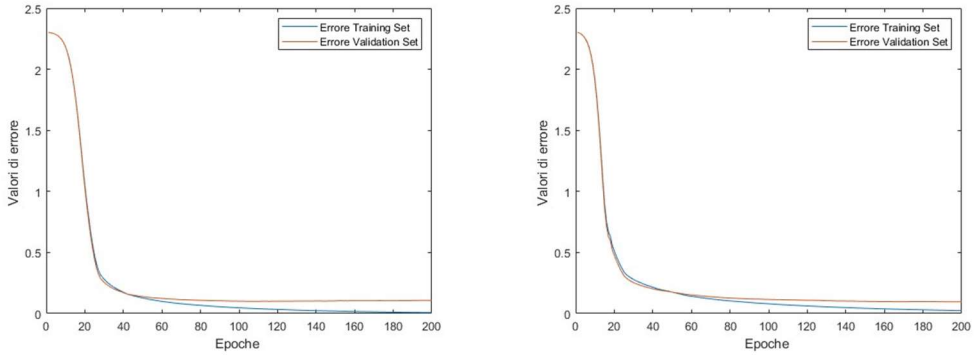


Grafico 1: Andamento accuratezza Sigmoide e ReLU al crescere del numero di strati interni

Nodi interni	Sigmoide			ReLU		
Strati interni	Accuracy	Err min Train	Err min Val	Accuracy	Err min Train	Err min Val
50	95,56%	0,049734	0,940241	95,42%	0,344339	0,088076
100	95,91%	0,001955	0,917676	95,77%	0,053603	0,94923
200	96,08%	0,0000112	0,080431	96,26%	0,000516	0,0879386
50-50	94,74%	0,071796	0,204026	95,15%	0,519876	0,106005
50-100	94,81%	0,027401	0,062282	95,84%	0,336135	0,004014
100-50	95,36%	0,009808	0,962706	95,82%	0,024136	0,098557
100-100	95,95%	0,000031	0,906343	95,53%	0,189769	0,98151
50-50-50	93,82%	0,198225	0,259816	94,79%	0,796943	0,238835
50-50-100	93,91%	0,188423	0,361631	94,49%	0,903892	0,230085
50-100-50	93,72%	0,31338	0,399657	94,39%	0,904491	0,294039
50-100-100	94,14%	0,147392	0,244497	94,94%	0,724989	0,167777
100-50-50	94,00%	0,051569	0,254691	95,32%	0,450641	0,072568
100-50-100	93,69%	0,057245	0,32585	95,39%	0,433605	0,09378
100-100-50	94,33%	0,009226	0,23197	95,04%	0,44484	0,101463
100-100-100	94,50%	0,012889	0,221271	95,41%	0,365972	0,991523
50-50-50-50	91,24%	0,1597584	0,255685	93,60%	0,094925	0,512389
50-50-50-100	92,76%	0,709699	0,67972	93,83%	0,093159	0,536867
50-50-100-50	91,56%	0,983135	0,946967	93,50%	0,215614	0,537625
50-50-100-100	90,30%	0,188918	0,246718	94,34%	0,93322	0,339962
50-100-50-50	91,30%	0,979574	0,950032	93,74%	0,11868	0,481747
50-100-50-100	92,83%	0,59265	0,569475	94,27%	0,868076	0,305485
50-100-100-50	92,74%	0,699016	0,714239	93,90%	0,046656	0,390776
50-100-100-100	91,60%	0,815726	0,869107	93,45%	0,071509	0,48948
100-50-50-50	92,82%	0,625081	0,617261	93,92%	0,864241	0,399182
100-50-50-100	93,11%	0,353956	0,522477	93,48%	0,154678	0,585088
100-50-100-50	92,71%	0,39275	0,56876	94,32%	0,863269	0,386827
100-50-100-100	92,53%	0,311886	0,597434	94,33%	0,719634	0,283184
100-100-50-50	92,95%	0,322884	0,667853	94,67%	0,577738	0,202496
100-100-50-100	93,28%	0,21098	0,458455	95,07%	0,582033	0,171988
100-100-100-50	93,25%	0,31236	0,465806	93,87%	0,819608	0,345394
100-100-100-100	93,95%	0,162886	0,338885	94,14%	0,77798	0,311172
50-50-50-50-50	81,45%	0,496809	0,710694	92,22%	0,573542	0,783753
50-50-50-50-100	86,67%	0,640169	0,323665	92,78%	0,498586	0,747147
50-50-50-100-50	84,24%	0,1824	0,854293	90,68%	0,034682	0,274907
50-50-50-100-100	89,98%	0,1790468	0,2314351	92,61%	0,411033	0,741837
50-50-100-50-50	87,78%	0,67935	0,081241	92,29%	0,63349	0,857959
50-50-100-50-100	84,36%	0,387131	0,457578	94,00%	0,989595	0,415189
50-50-100-100-50	81,17%	0,996018	0,594717	92,77%	0,407174	0,675245
50-50-100-100-100	90,92%	0,152804	0,2159647	93,81%	0,163537	0,453277
50-100-50-50-50	85,20%	0,081891	0,648234	92,59%	0,478721	0,803525
50-100-50-50-100	89,26%	0,941763	0,601192	92,97%	0,330177	0,666049
50-100-50-100-50	88,84%	0,186636	0,72343	92,81%	0,386765	0,706071
50-100-50-100-100	89,75%	0,922633	0,494313	93,63%	0,205744	0,519653
50-100-100-50-50	87,88%	0,440737	0,06809	92,77%	0,433694	0,73384
50-100-100-50-100	86,75%	0,348014	0,596142	92,85%	0,3827	0,606022
50-100-100-100-50	87,12%	0,84464	0,368428	92,43%	0,41491	0,751245
50-100-100-100-100	86,33%	0,149253	0,403356	92,70%	0,139443	0,168757
100-50-50-50-50	80,61%	0,576543	0,146581	94,07%	0,842084	0,371681
100-50-50-50-100	89,90%	0,914301	0,439956	92,53%	0,447362	0,759636
100-50-50-100-50	89,70%	0,917195	0,590861	91,65%	0,55629	0,947081
100-50-50-100-100	85,62%	0,07731	0,562026	93,99%	0,88539	0,356661
100-50-100-50-50	91,81%	0,15291	0,912208	94,14%	0,020253	0,446388
100-50-100-50-100	91,88%	0,004932	0,841516	94,36%	0,908539	0,365099
100-50-100-100-50	87,12%	0,418119	0,181445	93,25%	0,128057	0,1578
100-50-100-100-100	92,60%	0,992772	0,763987	93,07%	0,260705	0,596824
100-100-50-50-50	90,42%	0,112091	0,235917	93,28%	0,129664	0,533815
100-100-50-50-100	89,61%	0,731723	0,470653	92,99%	0,318638	0,698099
100-100-50-100-50	89,23%	0,166045	0,263538	93,86%	0,924099	0,330653
100-100-50-100-100	92,16%	0,867519	0,679742	94,10%	0,084797	0,139594
100-100-100-50-50	82,70%	0,374661	0,4270035	93,80%	0,047075	0,491967
100-100-100-50-100	89,08%	0,027291	0,621012	89,89%	0,441865	0,503463
100-100-100-100-50	92,55%	0,095157	0,175887	94,06%	0,081602	0,133504
100-100-100-100-100	89,39%	0,249245	0,573147	92,95%	0,264237	0,550043

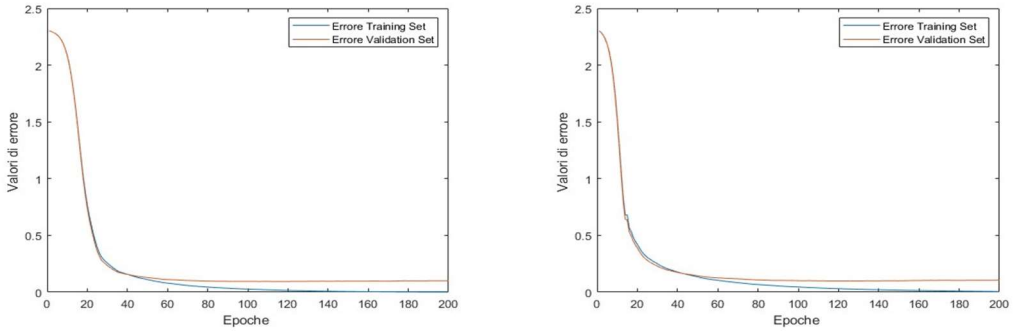
Tabella 1: Accuratezza Errore Training e Errore Validation Sigmoide e ReLU

Come configurazioni della rete neurale sono state prese tutte le possibili combinazioni utilizzando 50 e 100 come numero di nodi per gli strati interni, considerando 1, 2,3, 4 e 5 strati interni. Nella prossima pagina sono riportati i grafici delle funzioni di errore (sia dell'errore sul **training** che sul **validation**) per le configurazioni discusse.



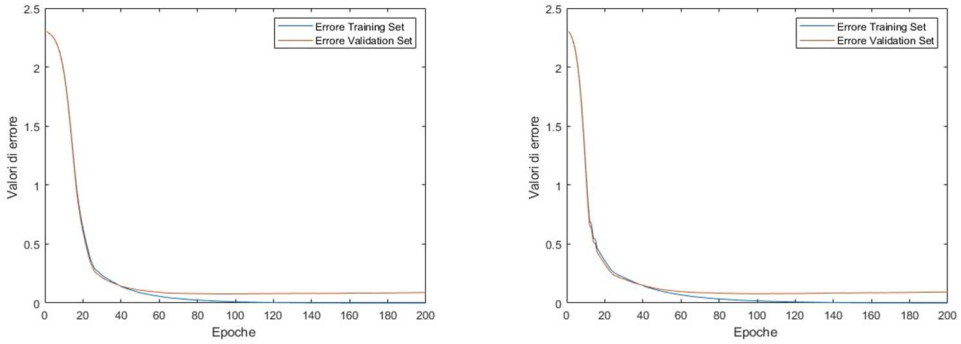
(a) 50 nodi Sigmoide (b) 50 nodi ReLU

Figura 4: 50 nodi confronto Sigmoide e ReLU



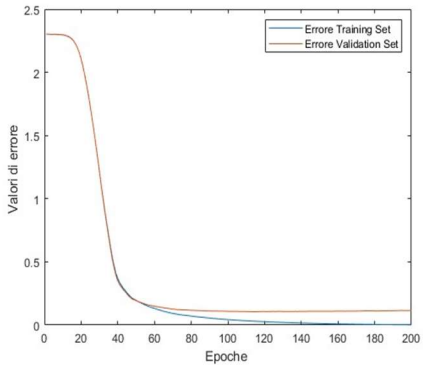
(a) 100 nodi Sigmoide (b) 100 nodi ReLU

Figura 5: 100 nodi confronto Sigmoide e ReLU

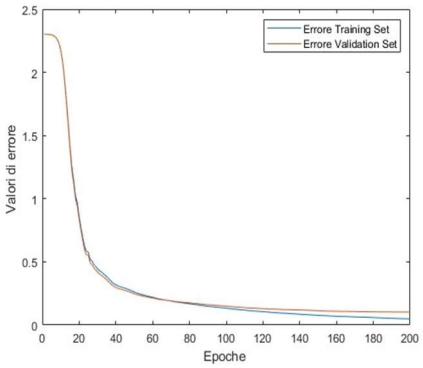


(a) 200 nodi Sigmoide (b) 200 nodi ReLU

Figura 6: 200 nodi confronto Sigmoide e ReLU

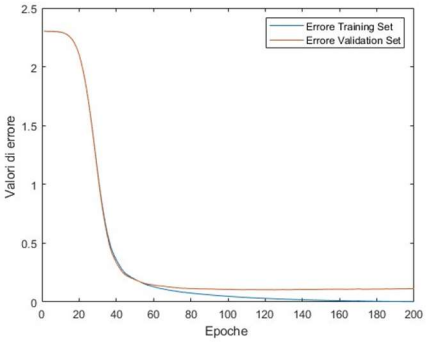


(a) 50-50 nodi Sigmoide

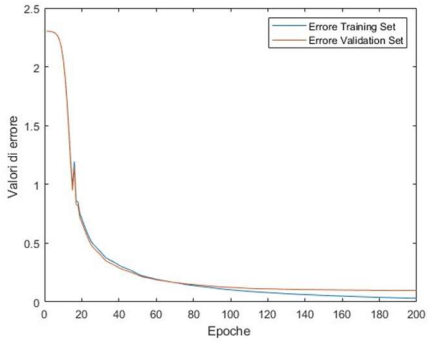


(b) 50-50 nodi ReLU

Figura 7: 50-50 nodi confronto Sigmoide e ReLU

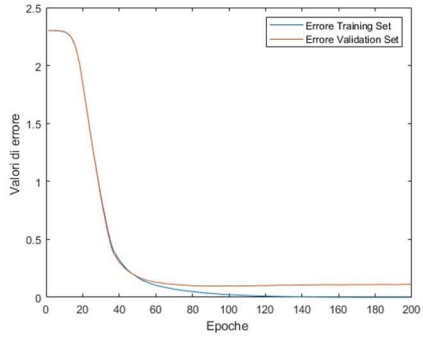


b(a) 50-100 nodi Sigmoide

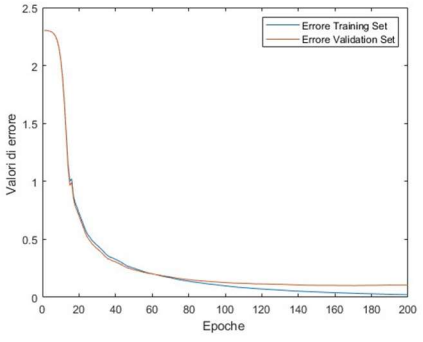


(b) 50-100 nodi ReLU

Figura 8: 50-100 nodi confronto Sigmoide e ReLU

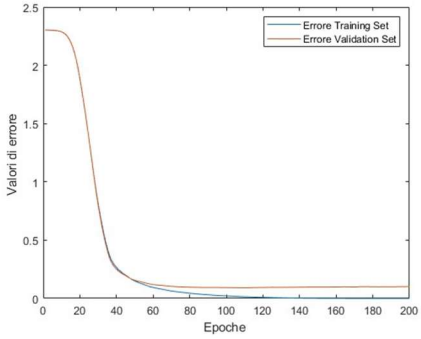


(a) 100-50 nodi Sigmoide

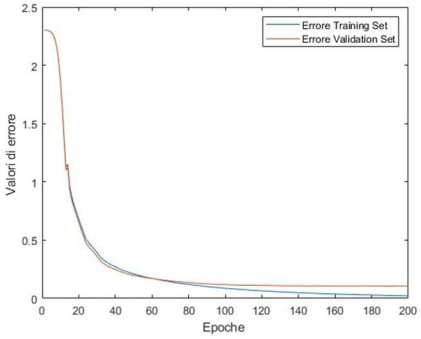


(b) 100-50 nodi ReLU

Figura 9: 100-50 nodi confronto Sigmoide e ReLU

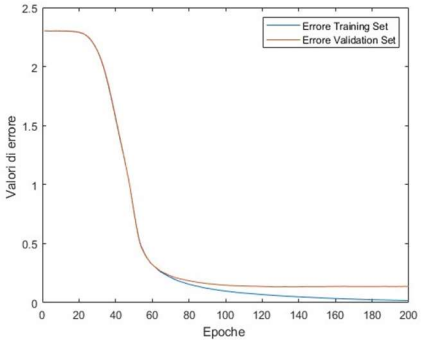


(a) 100-100 nodi Sigmoide

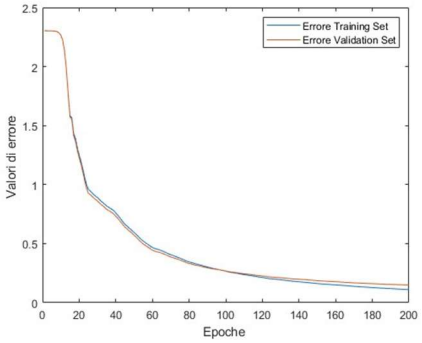


(b) 100-100 nodi ReLU

Figura 10: 100-100 nodi confronto Sigmoide e ReLU

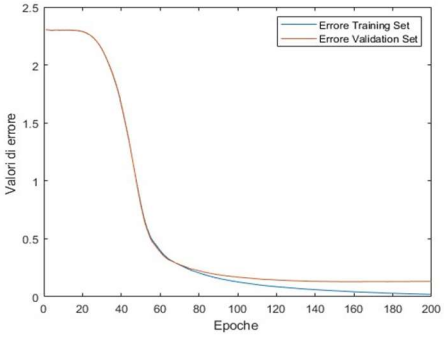


(a) 50-50-50 nodi Sigmoide

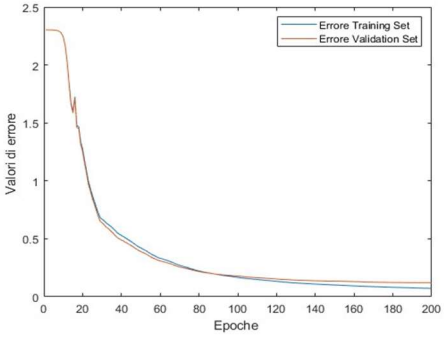


(b) 50-50-50 nodi ReLU

Figura 11: 50-50-50 nodi confronto Sigmoide e ReLU

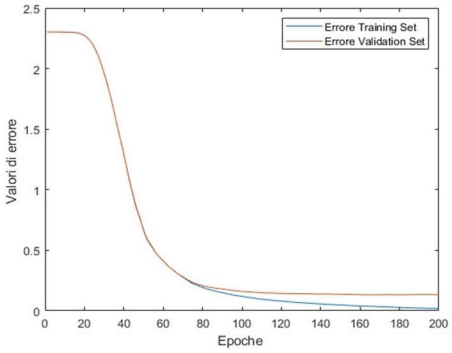


(a) 50-50-100 nodi Sigmoide

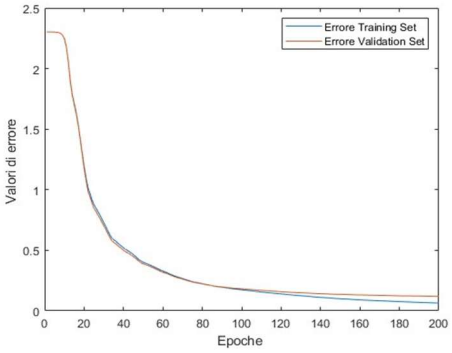


(b) 50-50-50 nodi ReLU

Figura 12: 50-50-100 nodi confronto Sigmoide e ReLU

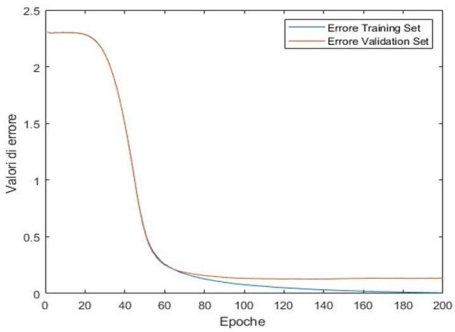


(a) 50-100-50 nodi Sigmoide

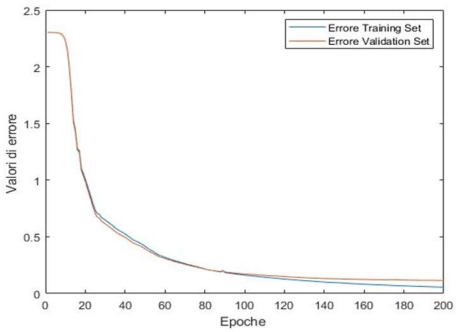


(b) 50-100-50 nodi ReLU

Figura 13: 50-100-50 nodi confronto Sigmoide e ReLU

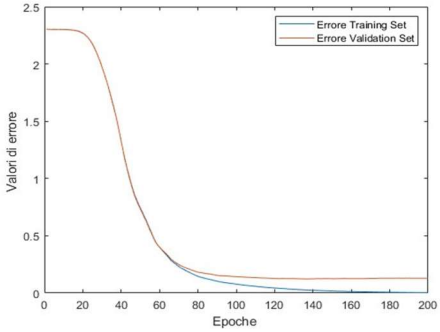


(a) 50-100-100 nodi Sigmoide

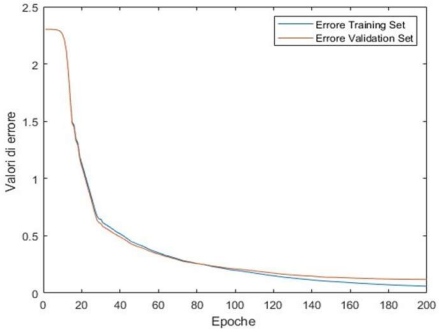


(b) 50-100-100 nodi ReLU

Figura 14: 50-100-100 nodi confronto Sigmoide e ReLU

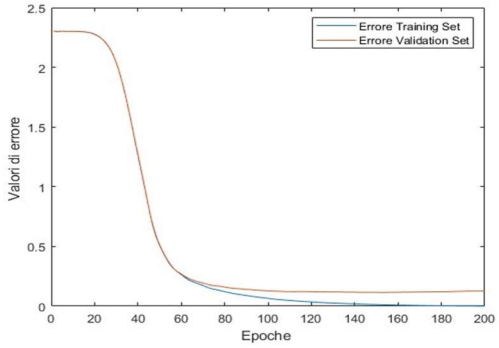


(a) 100-50-50 nodi Sigmoide

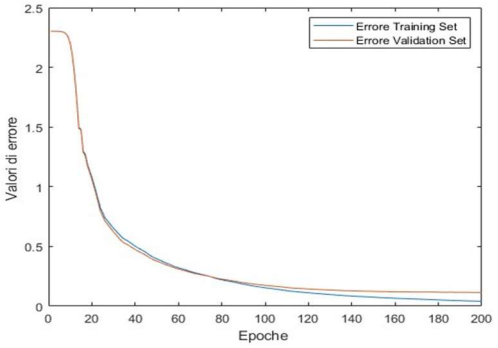


(b) 100-50-50 nodi ReLU

Figura 15: 100-50-50 nodi confronto Sigmoide e ReLU

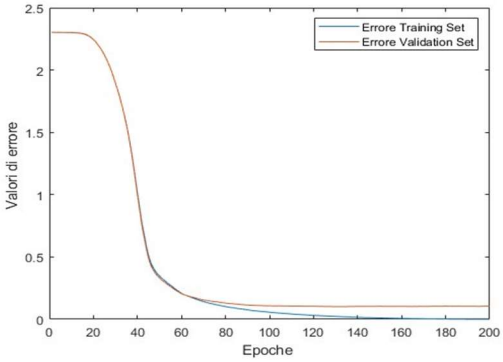


(a) 100-50-100 nodi Sigmoide

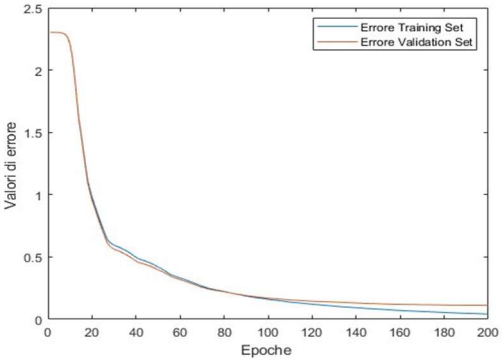


(b) 100-50-100 nodi ReLU

Figura 16: 100-50-100 nodi confronto Sigmoide e ReLU

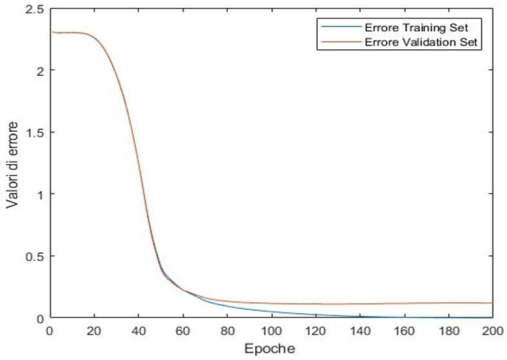


(a) 100-100-50 nodi Sigmoide

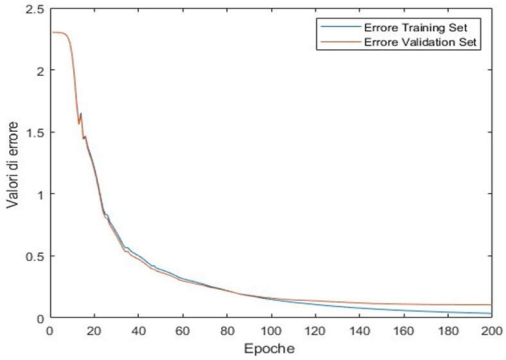


(b) 100-100-50 nodi ReLU

Figura 17: 100-100-50 nodi confronto Sigmoide e ReLU



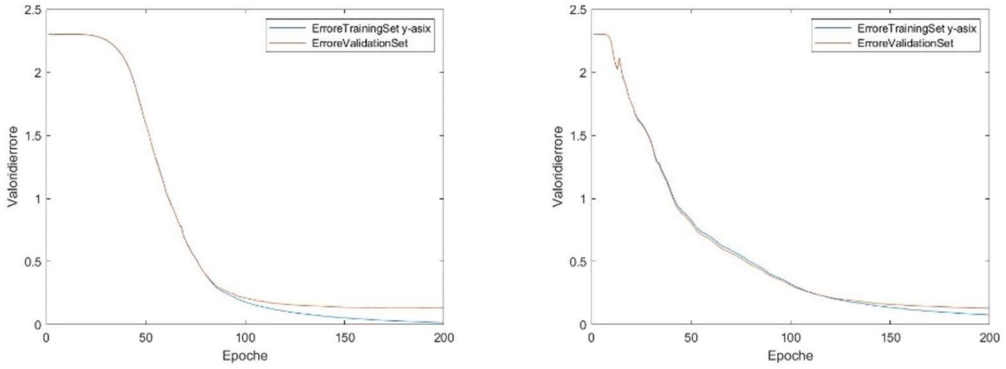
(a) 100-100-100 nodi Sigmoide



(b) 100-100-100 nodi ReLU

Figura 18: 100-100-100 nodi confronto Sigmoide e ReLU

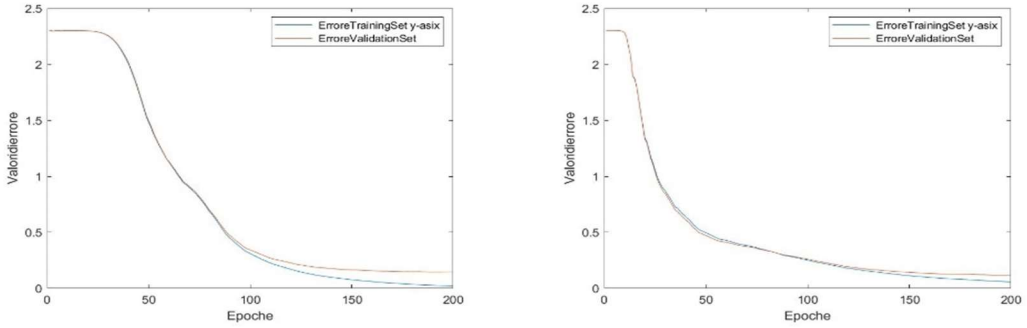
Massimizza Sigmoide in 4 strati:



(a) 100-100-100-100 nodi Sigmoide (b) 100-100-100-100 nodi ReLU

Figura 19: 100-100-100-100 nodi confronto Sigmoide e ReLU

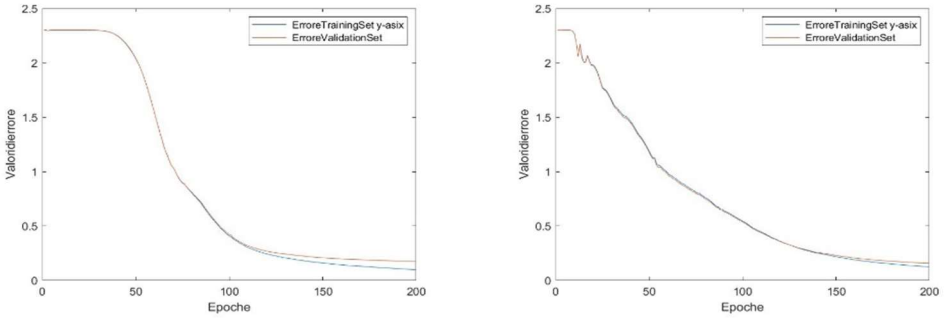
Massimizza ReLU in 4 strati:



(a) 100-100-50-100 nodi Sigmoide (b) 100-100-50-100 nodi ReLU

Figura 20: 100-100-50-100 nodi confronto Sigmoide e ReLU

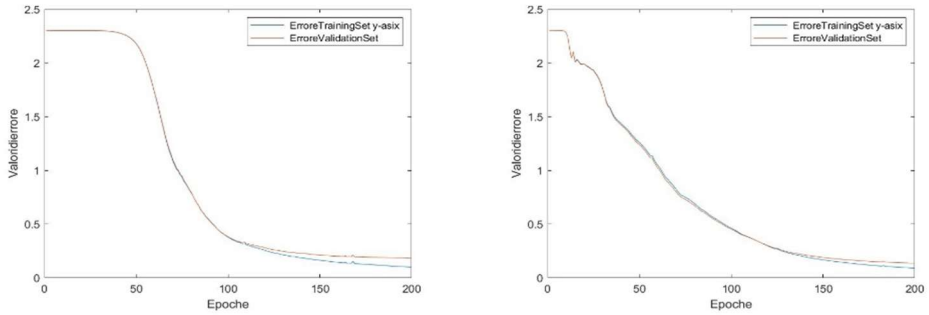
Massimizza Sigmoide in 5 strati:



(a) 100-50-100-100-100 nodi Sigmoide (b) 100-50-100-100-100 nodi ReLU

Figura 21: 100-50-100-100-100 nodi confronto Sigmoide e ReLU

Massimizza ReLU in 5 strati:



(a) 100-50-100-50-100 nodi Sigmoide (b) 100-50-100-50-100 nodi ReLU

Figura 22: 100-50-100-50-100 nodi confronto Sigmoide e ReLU

7. Conclusione sulle sperimentazioni

E' possibile innanzitutto notare che i risultati ottenuti, su tutte le diverse configurazioni di una rete neurale, sono molto positivi. Infatti notiamo che in quasi tutti i casi l'accuracy si aggira intorno ai valori del 94% e 95%. Dalle prove eseguite quindi viene confermato che l'aggiornamento dei pesi tramite RProp funziona in maniera estremamente positiva. Infatti, non solo vengono raggiunti valori di errore molto bassi, ma ciò avviene in un numero di epoche relativamente piccolo. Appurata dunque la bontà della RProp, è stato effettuato un confronto tra i risultati ottenuti utilizzando la **sigmoide** e quelli ottenuti utilizzando la **ReLU**. Tali risultati sono positivi sia utilizzando l'una che l'altra funzione di attivazione, anche se la ReLU per molte configurazioni si è comportata generalmente meglio, soprattutto all'aumentare del numero di strati. Tale risultato è dovuto al fenomeno del vanishing gradient. Infatti con più strati si ottiene una funzione di errore che presenta molti minimi relativi, cioè punti in cui il gradiente è pari a 0. Questo ovviamente può creare problemi per il raggiungimento del minimo assoluto. Infatti, la backward propagation calcola il gradiente utilizzando la **chain rule**, ed essendo la derivata della sigmoide compresa tra i valori (0 e 0.25), vengono effettuati molti prodotti tra numeri molto piccoli. Se si tiene conto che i valori ottenuti su uno strato, influiscono sui valori che si otterranno allo strato precedente, possiamo ben capire come si arrivi alla presenza di valori molto piccoli del gradiente, in particolare via via che si risale durante la backward propagation. Di conseguenza, se i gradienti sono prossimi a 0, vuol dire che l'aggiornamento dei pesi sarà pressoché impercettibile. Se guardiamo il grafico della sigmoide (rappresentato in Figura 23) possiamo notare che, se

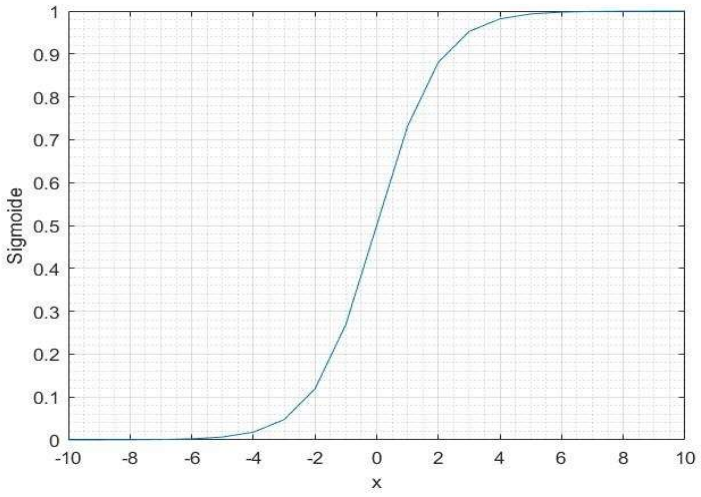


Figura 23: Sigmoide

l'input di un nodo non è vicino allo 0, la sigmoide tende ad assomigliare ad una funzione costante. Di conseguenza la derivata della sigmoide in quel punto è prossima allo 0, ma quindi anche il delta corrispondente a quel nodo è prossimo allo 0. Visto che i delta influiscono per il calcolo delle derivate, allora anche le

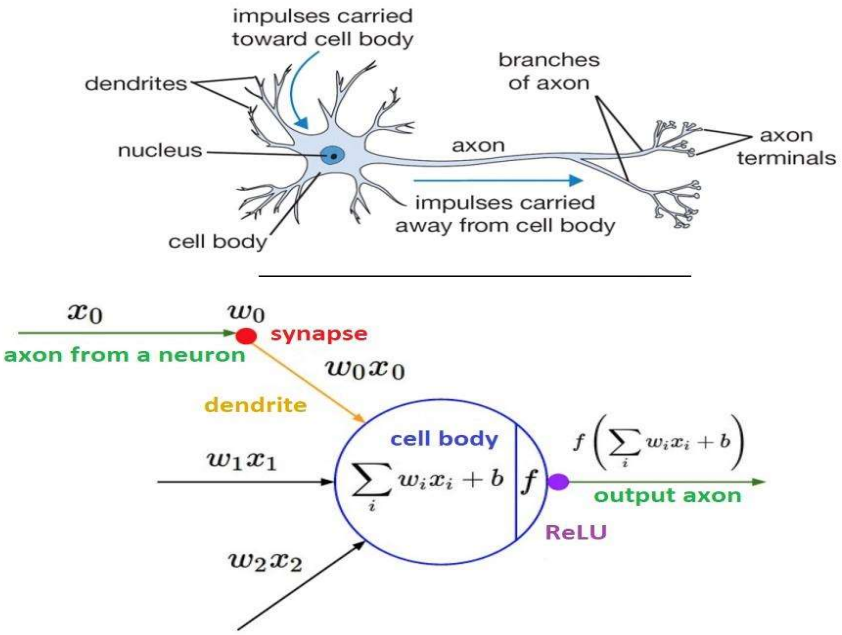


Figura 24: Neurone biologico ed implementazione attraverso ReLU

derivate dei pesi degli archi che vanno in tale nodo sono prossime allo 0. I delta inoltre, come detto precedentemente, verranno utilizzati anche per il calcolo dei delta dello strato precedente. Ma ci vuol dire che anche i delta dello strato precedente saranno prossimi allo 0, e di conseguenza lo saranno le derivate calcolate utilizzando questi delta. Dunque possiamo notare come ci sia un aumento dei minimi locali. Proprio per evitare il proliferare di minimi locali, nelle reti neurali deep viene utilizzata la Rectified Linear Unit (ReLU). Tale funzione è definita come $f(x) = max(0\ ,\ x)$ (rappresentata in Figura 25).
La ReLU non soffre del problema del vanishing gradient, infatti per i nodi i cui input sono positivi, la derivata della ReLU è 1, e ci garantisce che il gradiente non arrivi a valori molto vicini allo 0 durante la fase di back propagation. Notiamo infatti dai grafici ottenuti dalle sperimentazioni che la ReLU converge verso l'errore minimo più velocemente di quanto lo fa la sigmoide. Con la ReLU inoltre non tutti i nodi partecipano al calcolo dell'output, poiché i nodi il cui valore di attivazione risulta minore di 0 vengono "disattivati", cioè valgono 0 e quindi non in uniscono in alcun modo sui valori di input dei nodi dello strato successivo. In questo modo viene

rappresentata quella che è la vera relazione che intercorre tra i nodi di input e quelli di output. Ci fa pensare alla rete neurale biologica, infatti in essa è presente un'infinità di neuroni, ma solo alcuni di questi vengono attivati in base agli impulsi esterni (si vedi Figura 24). La sparsità conferisce maggiore potere predittivo alla rete e garantisce maggiore robustezza sui piccoli cambiamenti dell'input, poiché l'insieme di nodi attivi resterà pressoché lo stesso. La sparsità inoltre, come possiamo notare dai grafici ottenuti, permette

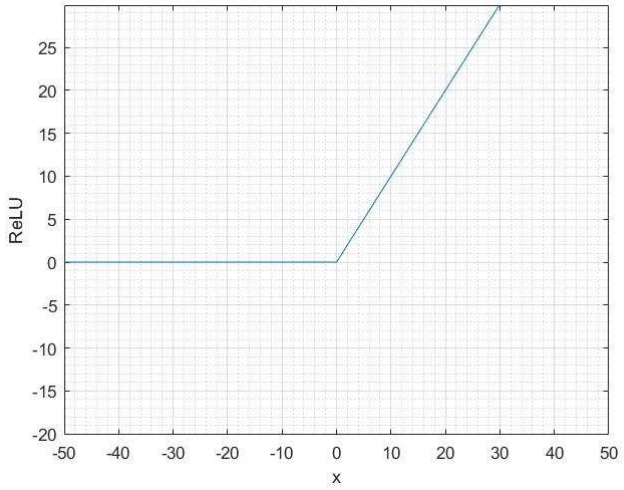


Figura 25: ReLU

anche di avere un overfitting minore. Un ulteriore vantaggio della ReLU è che i costi computazionali vengono notevolmente ridotti, poiché i calcoli da eseguire sono molto semplici. Sono stati infatti presi i tempi computazionali su un addestramento di una rete neurale 784-100-100-100-10 utilizzando 50000 immagini di training e 10000 di validation e sono stati ottenuti i seguenti valori: con la sigmoide il training ha avuto durata pari a 559,469 secondi mentre con la ReLU una durata pari a 424,295 secondi. La ReLU per può anche presentare degli svantaggi, in particolare quando ci sono troppi nodi disattivi. Ciò ovviamente comporta una perdita di potenza predittiva da parte della rete, che si manifesta in maniera totale quando tutti i neuroni sono disattivati (Dying ReLU). A tal proposito sono state proposte diverse soluzioni per risolvere questo problema, una tra le più appetibili risulta essere quella di scegliere come funzione di attivazione la Leaky ReLU (LReLU). La LReLU (rappresentata in Figura 26) è definita nel seguente modo:

$$\begin{cases} x, & \text{se } x \geq 0 \\ \alpha x, & \text{altrimenti} \end{cases} \tag{17}$$

con α solitamente piccolo. Tale funzione permette di dare una certa importanza anche a quei neuroni che con la ReLU sarebbero stati disattivi. Tuttavia, è stato osservato empiricamente che la LReLU non è generalmente migliore della ReLU, bensì converge verso l'errore minimo più velocemente.
È stato infatti effettuato un esperimento con la LReLU, ponendo $\alpha = 0.01$ su una rete 784-100-100-100-10, ed è stata ottenuta la curva rappresentata in Figura 27. L'accuracy della rete è del 95,9% e l'errore minimo registrato sul validation set è: 0.0979305. Notiamo dal grafico dunque, come effettivamente tale funzione di errore converga più velocemente al minimo, rispetto a quella ottenuta dall'addestramento utilizzando la ReLU su una rete con la stessa configurazione di nodi e strati interni.

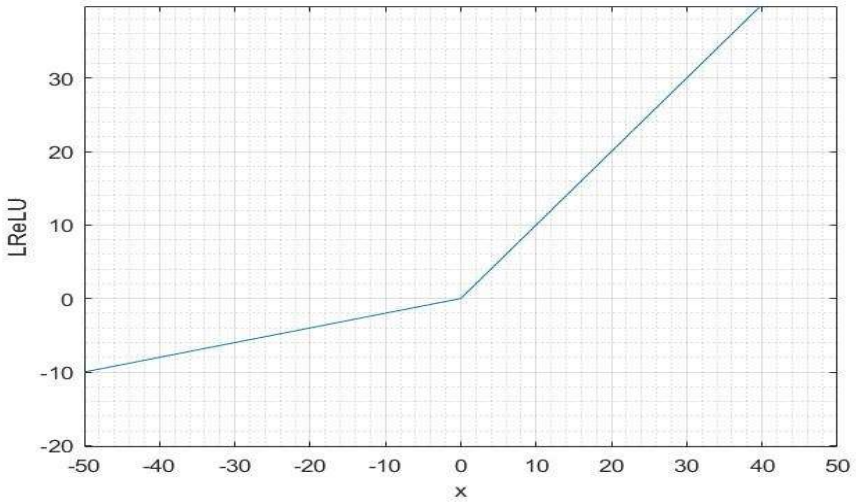


Figura 26: LReLU

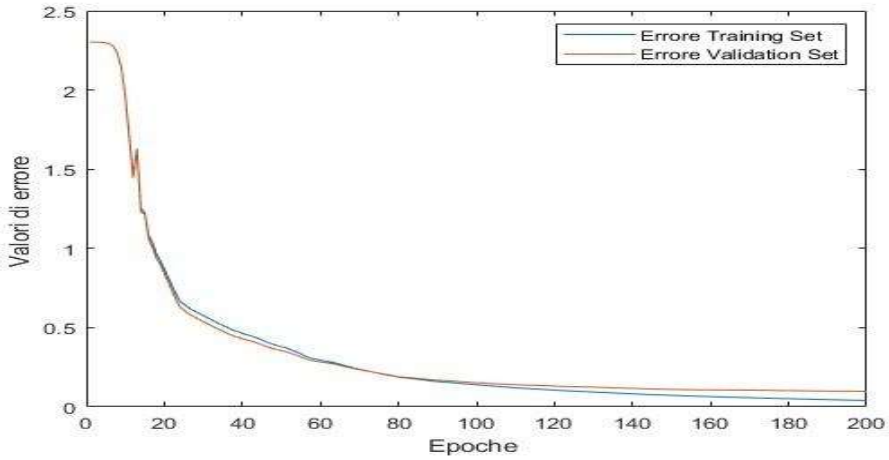


Figura 27: Esempio di utilizzo della LReLU

8. Codice Matlab

```
function net = aggiornaPesi(net,derW,derB,eta)
    %Discesa del gradiente
    for strato=1 : net.nStrati
       net.w{strato} = net.w{strato} - eta*derW{strato};
       net.b{strato} = net.b{strato} - eta*derB{strato};
    end
end

function [net,delta] = backPropagation(net,output,targets,funErr)
    %Inizializzazione di un cell array per i delta.
    delta = cell(1,net.nStrati);
    %Calcolo dei delta all'ultimo strato
    delta{net.nStrati} = funErr.der(output,targets);
    %Calcolo dei delta per i strati interni
    for strato = (net.nStrati-1) : -1 : 1
       delta{strato} = delta{strato+1} * net.w{strato+1};
       delta{strato} = net.funzioni{strato}.der(net.actValue{strato}) .* delta{strato};
    end
end

function [derW,derB] = calcolaDerivate(net,delta,input)
    %Inizializzazione di cell array per le derivate dei pesi e dei bias
    derW = cell(1,net.nStrati);
    derB = cell(1,net.nStrati);
    %Calcolo delle derivate per tutti i pesi
    z = input;
    for strato=1 : net.nStrati
        derW{strato} = delta{strato}' *z;
        derB{strato} = sum(delta{strato},1);
        z = net.output{strato};
    end
end

function net = createNetwork(vettoreStrati,vettoreFunzioni,pesi)
    nStrati = length(vettoreStrati) - 1;
    w = cell(1,nStrati);
    b = cell(1,nStrati);
    dim1 = vettoreStrati(1);
    for strato = 1:nStrati
       dim2 = vettoreStrati(strato+1);
       w{strato} = pesi - (2*pesi)*rand(dim2,dim1);
       b{strato} = pesi - (2*pesi)*rand(1,dim2);
       dim1=dim2;
    end
    net.w=w;
    net.b=b;
    net.nStrati = nStrati;
    net.funzioni = vettoreFunzioni;
    net.actValue = cell(1,nStrati);
    net.output = cell(1,nStrati);
end

%Funzione che ritorna la struttura della crossEntropy
function e = crossEntropy()
    e.fun = @f;
    e.der = @d;
    e.lastLayerFun = @lastLayerFun;
end
%Funzione della cross entropy
function y = f(out,target)
    y = -sum(sum(target .* log(max(out,0.09)),2));
end
%Derivata della cross entropy
function y = d(out,target)
    y = (out-target);
end

%Funzione da applicare all'ultimo strato della rete durante la forward
%propagation
function y = lastLayerFun(x)
    y = softmax(x);
end
```

```
function [net,z] = forwardPropagation(net,input,funErr)
    z = input;
    %Cardinalità degli elementi di input
    c = size(z,1);
    %Propagazione in avanti
    for strato= 1:net.nStrati
        a = (z * net.w{strato}') + repmat(net.b{strato},c,1);
        z = net.funzioni{strato}.fun(a);
        net.actValue{strato}=a;
        net.output{strato}=z;
    end
    %Applicazione di una ulteriore funzione sull'ultimo strato. Nel caso
    %della cross-entropy si tratta della softmax.
    net.output{net.nStrati} = funErr.lastLayerFun(net.output{net.nStrati});
    z = net.output{net.nStrati};
end

%Funzione che ritorna la struttura dell'identità
function s = identita()
    s.fun=@f;
    s.der=@d;
end
%Funzione identità
function y = f(x)
    y=x;
end
%Derivata dell'identità
function y = d(x)
    y=ones(size(x));
end

function [varW,varB] = initVariazioni(net,var)
    %Inizializzazione dei cell array
    varW = cell(1,net.nStrati);
    varB = cell(1,net.nStrati);
    %Per ogni strato si crea una matrice con tutti valori uguali a var.
    for strato=1:net.nStrati
       varW{strato} = var*ones(size(net.w{strato}));
       varB{strato} = var*ones(size(net.b{strato}));
    end

end

function out = loadMNIST(filename1,filename2)
    %Caricamento del dataset MNIST
    fp = fopen(filename1, 'rb');
    assert(fp ~= -1, ['Could not open ', filename1, '']);
    magic = fread(fp, 1, 'int32', 0, 'ieee-be');
    assert(magic == 2051, ['Bad magic number in ', filename1, '']);
    numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
    numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
    numCols = fread(fp, 1, 'int32', 0, 'ieee-be');
    images = fread(fp, inf, 'unsigned char');
    images = reshape(images, numCols, numRows, numImages);
    images = permute(images,[2 1 3]);
    fclose(fp);
    %Reshape to #pixels x #examples
    images = reshape(images, size(images, 1) * size(images, 2), size(images, 3));
    %Convert to double and rescale to [0,1]
    images = double(images) / 255;
    fp = fopen(filename2, 'rb');
    assert(fp ~= -1, ['Could not open ', filename2, '']);
    magic = fread(fp, 1, 'int32', 0, 'ieee-be');
    assert(magic == 2049, ['Bad magic number in ', filename2, '']);
    numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');
    labels = fread(fp, inf, 'unsigned char');
    assert(size(labels,1) == numLabels, 'Mismatch in label count');
    fclose(fp);
    out.Images = images';
    out.Labels = labels;
end
```

```
function s = lrelu()
    s.fun = @f;
    s.der = @d;
end
function y = f(x)
    y = (x>=0).*x + (x<0).*(0.2*x);
end
function y=d(x)
    y = (x>0).*1 + (x==0).*0 + (x<0).*(0.2);
end

function bestNet = ParteA_trainingBatch( vettoreStrati,
vettoreFunzioni,pesi,funErr,trainingSet,validationSet,nEpoche,eta,soglia)
    %Creazione di una rete neurale
    net = createNetwork(vettoreStrati, vettoreFunzioni,pesi);
    %Array degli errori per training set e validation set
    errT = zeros(1,nEpoche);
    errV = zeros(1,nEpoche);
    errMin = realmax;
    %Processo di addestramento
    e=1;
    stop=0;
    while(e<=nEpoche && stop==0)
        [net,output] = forwardPropagation(net,trainingSet.images,funErr);
        [net,delta] = backPropagation(net,output,trainingSet.targets,funErr);
        [derW,derB] = calcolaDerivate(net,delta,trainingSet.images);
        [net] = aggiornaPesi(net,derW,derB,eta);
        %Calcolo dell'errore su training set
        [net,out]=forwardPropagation(net,trainingSet.images,funErr);
        errT(e) = funErr.fun(out,trainingSet.targets)/size(trainingSet.targets,1);
        %Calcolo dell'errore su validation set
        [net,out]=forwardPropagation(net,validationSet.images,funErr);
        errV(e)=funErr.fun(out,validationSet.targets)/size(validationSet.targets,1);
        %Se l'errore all'epoca corrente è minore del minimo ottenuto, la
        %rete corrente diventa la migliore.
        if errV(e) < errMin
            errMin = errV(e);
            bestNet = net;
        end
        %Stampa degli errori
        fprintf("e: %d errT: %.7f errV: %.7f\n",e,errT(e),errV(e));
        %Criterio di fermata
        t = abs(100*((errMin/errV(e))-1));
        if(t > soglia)
            stop = 1;
        end
        e=e+1;

    end
    %Stampa errore minimo
    fprintf("Errore minimo validation: %.7f\n",errMin);
    figure
    x=1:nEpoche;
    plot(x,errT(x),x,errV(x));
    xlabel('Epoche ') % x-axis label
    ylabel('Valori di errore ') % y-axis label
    legend('Errore Training Set','Errore Validation Set')
end
```

```
function bestNet = ParteA_trainingOnline( vettoreStrati,
vettoreFunzioni,pesi,funErr,trainingSet,validationSet,nEpoche,eta,soglia)
    %Creazione di una rete neurale
    net = createNetwork(vettoreStrati, vettoreFunzioni,pesi);
    %Array degli errori per training set e validation set
    errT = zeros(1,nEpoche);
    errV = zeros(1,nEpoche);
    errMin = realmax;
    %Processo di addestramento
    e=1;
    stop=0;
    while(e<=nEpoche && stop==0)
        for i=1:size(trainingSet.images,1)
            [net,output] = forwardPropagation(net,trainingSet.images(i,:),funErr);
            [net,delta]= backPropagation(net,output,trainingSet.targets(i,:),funErr);
             [derW,derB] = calcolaDerivate(net,delta,trainingSet.images(i,:));
            [net] = aggiornaPesi(net,derW,derB,eta);
        end
    %Calcolo dell'errore su training set
    [net,out]=forwardPropagation(net,trainingSet.images,funErr);
    errT(e) = funErr.fun(out,trainingSet.targets)/size(trainingSet.targets,1);
    %Calcolo dell'errore su validation set
    [net,out]=forwardPropagation(net,validationSet.images,funErr);
    errV(e) = funErr.fun(out,validationSet.targets)/size(validationSet.targets,1);
    %Se l'errore all'epoca corrente è minore del minimo ottenuto, la
    %rete corrente diventa la migliore.
    if errV(e) < errMin
        errMin = errV(e);
        bestNet = net;
    end
        %Stampa degli errori
        fprintf("e: %d errT: %.7f errV: %.7f\n",e,errT(e),errV(e));
        %Criterio di fermata
        t = abs(100*((errMin/errV(e))-1));
        if(t > soglia)
            stop = 1;
        end
       e=e+1;
    end
    %Stampa errore minimo
    fprintf("Errore minimo validation: %.7f\n",errMin);
    figure
    x=1:nEpoche;
    plot(x,errT(x),x,errV(x));
    xlabel('Epoche ') % x-axis label
    ylabel('Valori di errore ') % y-axis label
    legend('Errore Training Set','Errore Validation Set')
end

%Funzione che ritorna la struttura della relu
function s = relu()
    s.fun = @f;
    s.der = @d;
end
%Funzione della relu
function y = f(x)
    y = max(0,x);
end
%Derivata della relu
function y=d(x)
    y = (x>0);
end

function [net,oldDerW,oldDerB,varW,varB]
=rprop(net,derW,derB,oldDerW,oldDerB,varW,varB,etaP,etaN)
for strato=1:net.nStrati
    %Segno del prodotto tra le nuove e le vecchie derivate.
    signProd = sign(oldDerW{strato} .* derW{strato});
    %Individuazione degli eta da utilizzare nell'aggiornamento dei pesi
    etaM = ((signProd>0) * etaP) + ((signProd<0) * etaN) + ((signProd==0) * 1);
    %Calcolo del delta che rappresenta la variazione del peso
    varW{strato} = etaM .* varW{strato};
    %I delta sono compresi tra 1e-6 e 50
    varW{strato} = min(max(varW{strato},0.000001),50);
    %Aggiornamento dei pesi
    net.w{strato} = net.w{strato} - (sign(derW{strato}) .* varW{strato});
```

```
    %Segno del prodotto tra le nuove e le vecchie derivate.
    signProd = sign(oldDerB{strato} .* derB{strato});
    %Individuazione degli eta da utilizzare nell'aggiornamento dei bias
    etaM = ((signProd>0) * etaP) + ((signProd<0) * etaN) + ((signProd==0) * 1);
    %Calcolo del delta che rappresenta la variazione del bias
    varB{strato} = etaM .* varB{strato};
    %I delta sono compresi tra 1e-6 e 50
    varB{strato} = min(max(varB{strato},0.000001),50);
    %Aggiornamento dei bias
    net.b{strato} = net.b{strato} - (sign(derB{strato}) .* varB{strato});

end
oldDerW = derW;
oldDerB = derB;
end

%Funzione che ritorna la struttura della sigmoide
function s = sigmoide()
    s.fun=@f;
    s.der=@d;
end

%Funzione della sigmoide
function y = f(x)
    y = 1 ./ (1+exp(-x));
end

%Derivata della sigmoide
function y = d(x)
    y = f(x) .* (1-f(x));
end

function s = softmax(x)
    %s = exp(x-max(x'))./sum(exp(x-max(x')),2);
    m = max(x,[],2);
    s = exp(x-m)./sum(exp(x-m),2);
end

%Funzione che ritorna la struttura della 'somma dei quadrati'
function e = sumOfSquares()
    e.fun = @f;
    e.der = @d;
    e.lastLayerFun = @lastLayerFun;
end

%Funzione della 'somma dei quadrati'
function y= f(out,t)
    y = sum(sum((out-t).^2,2) /2);
end

%Derivata della 'somma dei quadrati'
function z = d(y,t)
    z = y-t;
end

%Funzione da applicare all'ultimo strato della rete durante la forward
%propagation
function y = lastLayerFun(x)
    y = x;
end

function acc = testing(net,testSet,funErr)
    acc=0;
    for i=1:size(testSet.Images,1)
        net = forwardPropagation(net,testSet.Images(i,:),funErr);
        [val,k]=max(net.output{net.nStrati});
        %fprintf("BEST: immagine: %d, MAX: %.7f, res: %d, label:
%d\n",i,max(net.output{net.nStrati}),k,test.Labels(i,1));
        if(k==10)
            k=0;
        end
        if( k == testSet.Labels(i,1))
            acc=acc+1;
        end
    end
    acc = acc/size(testSet.Images,1);
end
```

```
function bestNet = trainingRProp ( vettoreStrati , vettoreFunzioni , pesi , funErr ,
trainingSet , validationSet , nEpoche , eta , etaP , etaN , variation , soglia )
%Creazione  di  una rete neurale
net = createNetwork ( vettoreStrati ,   vettoreFunzioni ,pesi );
%Inizializzazione dei delta per l'aggiornamento dei pesi della rProp
[varW, varB ] = initVariazioni ( net , variation ) ;
%Array  degli   errori  per training set e validation set
errT = zeros (1,nEpoche) ;
errV = zeros(1,nEpoche) ;
errMin = realmax;
errMin2= realmax;
%Processo di addestramento
e=1;
stop=0;
while (e<=nEpoche && stop==0)
    [ net,output ] = forwardPropagation ( net ,trainingSet.images ,funErr ) ;
    [ net , delta ] = backPropagation ( net , output ,trainingSet.targets,funErr ) ;
    [derW, derB ] = calcolaDerivate ( net , delta ,trainingSet.images ) ;
    %Alla prima epoca viene eseguito un passo di aggiornamento dei pesi
    %tramite la discesa del gradiente .
    if ( e==1)
        [ net ] = aggiornaPesi ( net ,derW, derB , eta ) ;
        oldDerW=derW;
        oldDerB=derB ;
    else
        %Aggiornamento  dei pesi    tramite rprop.
        [ net , oldDerW , oldDerB ,varW, varB ] = rprop ( net ,derW, derB , oldDerW , oldDerB
,varW, varB , etaP ,etaN) ;
    end
    %Calcolo    dell ' errore   su  training    set
    [ net , out]=forwardPropagation ( net , trainingSet . images , funErr ) ;
    errT ( e ) = funErr . fun (out , trainingSet . targets )/ size ( trainingSet.targets ,1) ;
    %Calcolo dell' errore   su  validation  set
    [ net , out]=forwardPropagation ( net , validationSet . images , funErr ) ;
    errV (e) = funErr . fun (out , validationSet . targets )/ size (validationSet.targets,1) ;
    %Se l' errore all' epoca corrente e' minore del minimo ottenuto , la
    %rete   corrente    diventa la  migliore .
    %calcolo l'errore minimo validation
    if  errV(e) < errMin
        errMin = errV(e);
        bestNet = net;
    end
    %calcolo l'errore minimo training
    if  errT(e) < errMin2
        errMin2 = errT(e);
    end
    %Stampa degli   errori
    fprintf("e: %d errT: %.7f errV: %.7f\n",e,errT(e),errV(e));
    %Criterio   di  fermata
    t = abs(100*(( errMin/errV(e))-1)) ;
    if ( t > soglia )
        stop = 1;
    end
    e=e+1;
end
%Stampa errore minimo
fprintf ("Errore minimo validation : %.7f\n" ,errMin);
fprintf ("Errore minimo training : %.7f\n" ,errMin2);
figure
x=1:e-1;
plot (x , errT (x) ,x , errV (x) );
xlabel ( "Epoche" ) % x-axis    label
ylabel ( 'Valori di errore  ' ) %y-axis label
legend ( 'Errore Training Set y-asix' , ' Errore    Validation  Set ' )
end
```

9. Riferimenti Bibliografici

[1] Martin Riedmiller, Rprop - Description and Implementation Details, Citeseer, 1994.

[2] Christopher Bishop, Neural networks for pattern recognition, Clarendon Press, 1996.

[3] Roberto Prevete, Appunti del corso di Machine Learning Mod. B.