

UNIVERSITÀ DEGLI  
STUDI DI NAPOLI  
FEDERICO II

Scuola Politecnica e delle Scienze di Base  
Corso di Laurea Magistrale in Informatica

---

# Documentazione

## Progettazione di una rete neurale

Insegnamento “Neural Networks and Deep Learning”

Anno Accademico 2021/2022

---

### AUTORI

Borzacchiello Francesco N97000380

Grimaldi Carmine N97000394

### DOCENTE

Prof. Prevete Roberto

Traccia: 4

## Sommario

1.	Traccia .....	4
1.1.	Parte A .....	4
1.2.	Parte B .....	4
2.	Cenni teorici.....	5
2.1.	Rete neurale fully connected.....	5
3.	Parte A .....	6
3.1.	Funzionalità principali .....	6
3.1.1	Inizializzazione della rete neurale .....	6
3.1.2	Propagazione in avanti.....	7
3.1.3	Retropropagazione dell'errore.....	9
3.2.	Strutture dati principali .....	11
3.2.1	Dataset .....	11
3.2.2	Rete neurale .....	12
3.3.	Principali algoritmi realizzati.....	13
3.3.1	Forward propagation .....	13
3.3.2	Backpropagation .....	14
4.	Parte B .....	17
4.1.	Ulteriori funzionalità principali .....	17
4.1.1	Caricamento e divisione dataset MNIST .....	17
4.1.2	Stochastic gradient descent .....	18
4.1.3	Resilient backpropagation .....	19
4.2.	Principali algoritmi realizzati.....	21
4.2.1	Stochastic gradient descent .....	21
4.2.2	Resilient backpropagation.....	22
4.3.	Parte sperimentale .....	25
4.3.1	Inizializzazione del modello.....	25
4.3.2	Divisione del dataset .....	26

4.3.3	Fase di apprendimento .....	27
4.3.4	Risultati ottenuti.....	29
4.3.5	Discussione dei risultati.....	39
4.3.6	Conclusioni ed eventuali sviluppi futuri .....	40

## 1. Traccia

### 1.1. Parte A

- Progettazione ed implementazione di funzioni per simulare la propagazione in avanti di una rete neurale multi-strato. Dare la possibilità di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione per ciascun strato.
- Progettazione ed implementazione di funzioni per la realizzazione della back-propagation per reti neurali multi-strato, per qualunque scelta della funzione di attivazione dei nodi della rete e la possibilità di usare almeno la somma dei quadrati o la cross-entropy con e senza soft-max come funzione di errore.

### 1.2. Parte B

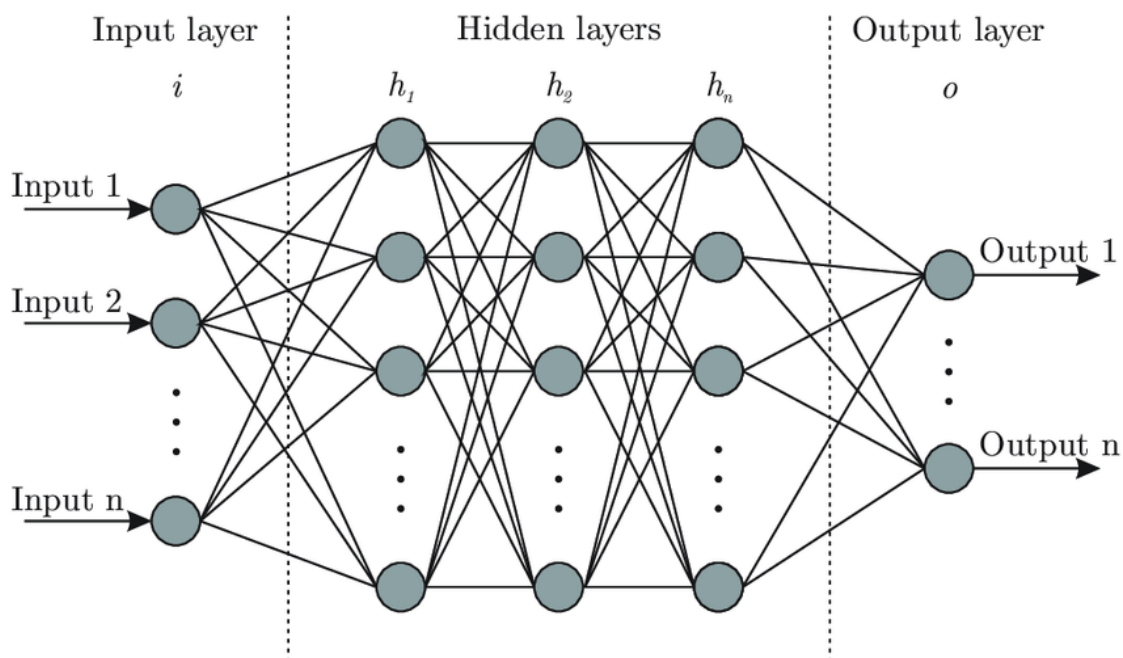
Traccia 4:

- 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, 5000 per il training set, 2500 per il validation set, 2500 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 attivazione dei nodi la sigmoide con quello in cui si usa come funzione di attivazione 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. Cenni teorici

### 2.1. Rete neurale fully connected

La rete neurale implementata in questo progetto è una rete neurale **full-connected**, in cui ogni neurone di un layer riceve connessioni da tutti i neuroni del layer precedente; questa tipologia di reti neurali appartiene alla classe delle reti **feed-forward**.



L'obiettivo di una rete feed-forward è di approssimare una funzione  $f^*$ . Questo tipo di rete definisce un mapping  $y = f(x; \theta)$  e apprende il valore dei parametri  $\theta$  che risultano nella miglior approssimazione della funzione. La funzione  $f$  è composta da una catena di funzioni:  $f = f^{(k)}(f^{(k-1)}(\dots f^{(1)}))$ , dove  $f^{(1)}$  rappresenta il primo layer, e così via. La profondità della rete è  $k$ , dove in particolare l'ultimo layer è chiamato output layer. Piuttosto che pensare a un layer come a una singola funzione da vettore a vettore, possiamo anche pensare a un layer come a un insieme di unità che agiscono in parallelo, ognuna delle quali rappresenta una funzione da un vettore a uno scalare.

## 3. Parte A

### 3.1. Funzionalità principali

Di seguito sono illustrati i dettagli delle funzionalità principali implementate, i quali fanno riferimento alla parte A della traccia assegnata.

#### 3.1.1 Inizializzazione della rete neurale

Come richiesto dalla traccia, l'inizializzazione della rete neurale è tale da risultare flessibile verso un numero arbitrario di strati interni, di neuroni per ciascun strato, scelta delle funzioni di attivazione e di errore. In particolare, le funzioni di errore richieste sono:

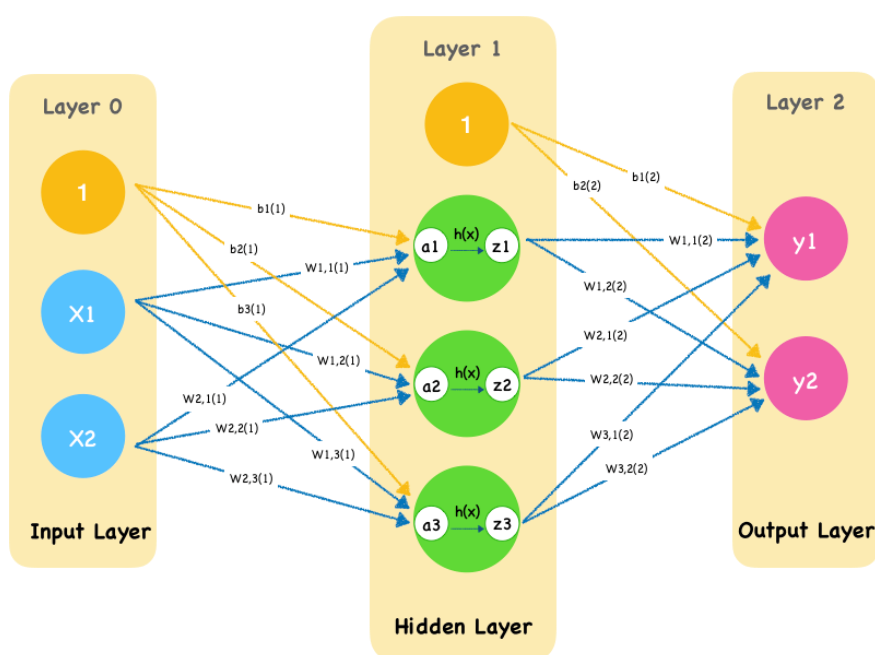
- Cross-Entropy:  $\sum_{n=1}^N - \sum_{k=1}^c (t_k^n \cdot \log y_k^n)$
- Cross-Entropy con SoftMax:  $\sum_{n=1}^N - \sum_{k=1}^c \left( t_k^n \cdot \log \left( \frac{e^k}{\sum_{h=1}^c e^h} \right) \right)$
- Sum of Squares:  $\sum_{n=1}^N \frac{1}{2} \sum_{k=1}^c (y_k^n - t_k^n)^2$

Dove in particolare:

- $N$  è la dimensione del dataset;
- $c$  è il numero di neuroni di output, dunque il numero di classi predette;
- $t_k^n$  è l'etichetta associata al  $k$ -esimo neurone dello strato di output con input  $n$ ;
- $y_k^n$  è l'output prodotto dal  $k$ -esimo neurone dello strato di output con input  $n$ .

### 3.1.2 Propagazione in avanti

Una delle prime richieste della parte A della traccia è stata l'implementazione della propagazione in avanti di una rete neurale multi-strato; tale fase rappresenta uno dei processi centrali della fase di apprendimento. Un modello di rete neurale individua i parametri (pesi e bias) con l'aiuto della propagazione in avanti e della backpropagation. In questa fase i dati di input vengono forniti alla rete seguendo una direzione in “avanti” (dallo strato di input verso quello di output).



*Rappresentazione grafica della propagazione in avanti per una rete neurale a singolo strato nascosto.*

Ogni livello nascosto riceve i dati in ingresso, li elabora secondo la funzione di attivazione e passa al livello successivo. Per ogni neurone in uno strato nascosto o di output, l'elaborazione avviene in due fasi:

1. **Pre-attivazione:** è una somma pesata calcolata tra i dati in input e i relativi pesi, con l'aggiunta del bias, ottenendo così un singolo valore; in formule:

$$\forall h \in [1, H], a_i^h = \sum_j w_{ij}^h \cdot z_j^{h-1} + b_i^h$$

Dove:

- $a_i^h$  è l'input del neurone i-esimo appartenente allo strato h;
- $z_j^h$  è l'output del neurone j-esimo appartenente allo strato h;
- $w_{ij}^h$  è il peso della connessione che arriva al nodo i-esimo dello strato h, a partire dal neurone j-esimo dello strato h-1;
- $b_i^h$  è il bias del neurone i-esimo appartenente allo strato h.

2. **Attivazione:** la somma così calcolata viene passata alla funzione di attivazione associata al neurone. Tale funzione consente di aggiungere una non-linearità alla rete. Esistono diverse funzioni di attivazione comunemente usate, come ad esempio sigmoide e ReLU. L'output computato dalla funzione di attivazione possiamo esprimerlo in formule:

$$z_i^h = f_h(a_i^h)$$

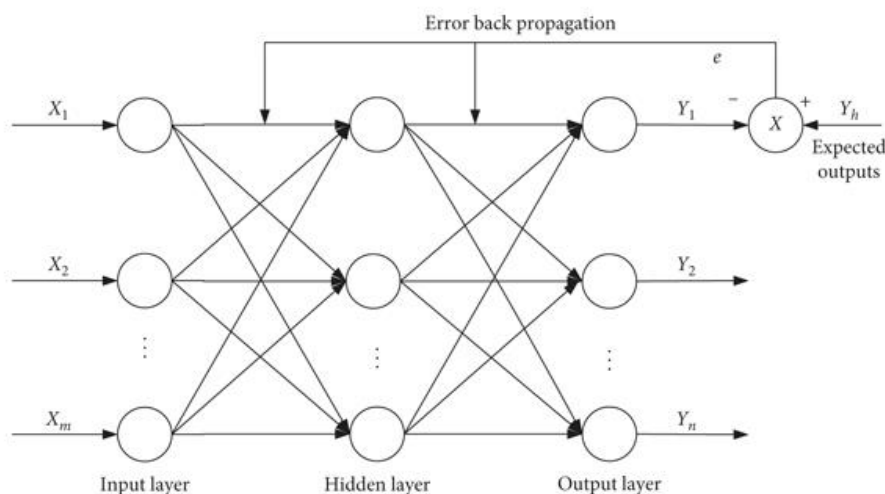
Con  $f_h$  la funzione di attivazione relativa a tutti i neuroni dello strato h (semplificazione/vincolo: in generale ogni neurone ha una sua funzione di attivazione).

Questo processo si ripete per ogni strato successivo della rete, dove infine si ottiene una sequenza di valori, associati allo strato di output, interpretabili come predizione di appartenenza dell'input alle possibili classi.



### 3.1.3 Retropropagazione dell'errore

A seguire con le richieste della parte A troviamo la realizzazione della back-propagation; tale fase sussegue a quella di propagazione in avanti descritta nel paragrafo precedente. Tale algoritmo non aggiorna i pesi della rete, perché questo è svolto da una regola di aggiornamento (es. learning rate); esso si limita al calcolo del gradiente, ovvero il calcolo delle derivate parziali della funzione di errore rispetto ai parametri della rete, quali pesi e bias, così da poter successivamente modificare e migliorare tali parametri grazie ad un algoritmo di aggiornamento, come ad esempio la discesa del gradiente, al fine di minimizzare la funzione di errore stessa.



*Rappresentazione grafica della retropropagazione dell'errore per una rete neurale a singolo strato nascosto.*

Per calcolare le derivate parziali dell'errore rispetto ai parametri, è necessario, per ogni neurone  $i$ , calcolare il cosiddetto  $\delta_i$ , con una formula diversa a seconda del tipo di strato  $i$  contenente il neurone (nascosto o di output).

Formalmente:

- se il nodo  $i$  appartiene allo strato di output:

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

- se il nodo  $i$  appartiene ad uno strato nascosto:

$$\delta_i^n = f'_n(a_i^n) \cdot \sum_j (\delta_j^n \cdot w_{ji})$$

Dove:

- $\sum_j (\delta_j^n \cdot w_{ji})$  è la somma pesata tra i pesi che partono dal neurone  $i$  ai neuroni  $j$  dello strato precedente;
- $g'(a_k^n)$  è la derivata prima della funzione di attivazione dello strato di output applicata al neurone  $k$  appartenente allo strato  $n$ ;
- $f'_n(a_i^n)$  è la derivata prima della funzione di attivazione dello strato nascosto  $n$  applicata al neurone  $i$  appartenente allo strato  $n$ ;
- $\frac{\partial E^{(n)}}{\partial y_k^n}$  è la derivata parziale della funzione dell'errore commesso sull'input  $n$ , rispetto all'output del neurone  $i$ .

I delta dei nodi si calcolano a partire dallo strato di output e tramite una retropropagazione allo strato precedente si calcolano i restanti delta.

## 3.2. Strutture dati principali

In questo paragrafo sono descritte le principali strutture dati utilizzate ai fini implementativi. In particolare, è stato utilizzato il linguaggio di programmazione Matlab.

### 3.2.1 Dataset

La struttura dati rappresentante il dataset, che in questo caso è MNIST, è rappresentata da una struttura contenente due campi, i quali sono:

- **Images:** matrice di dimensioni 784x60000, dove 784 sono il numero di pixel associati a ciascun'immagine, e 60000 rappresenta il numero di immagini contenute nel dataset.
- **Labels:** matrice di dimensioni 60000x1, dove sono contenute le etichette associate a ciascuna delle 60000 immagini del dataset. In questa matrice sono contenute le etichette espresse in formato decimale, rappresentanti della classe di appartenenza dell'immagine considerata.

Successivamente vengono eseguite operazioni di post-processing che fanno sì che le etichette vengano convertite nel formato one-hot.

	1						
1	1						
2	6						
3	6						
4	2						
5	2						
6	5						

Conversione in  
one-hot →







	1	2	3	4	5	6	
1	1	0	0	0	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	1	1	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0

A seguito della conversione in one-hot le etichette sono state trasposte per avere una dimensione coerente alla matrice delle immagini, al fine di poter eseguire le operazioni matriciali necessarie.

### 3.2.2 Rete neurale

La rete neurale è rappresentata mediante una classe che è costituita dai seguenti campi privati, a cui si accederà mediante opportuni getter e setter:

- **weight:** array di celle, in cui ogni cella contiene una matrice che contiene i pesi di ciascun layer della rete. Il numero di celle dipende dal numero di layer. La dimensione delle matrici dipende dal numero di neuroni, in cui ci sono tante righe quante il numero di neuroni del layer successivo e tante colonne quante il numero di neuroni del layer corrente;
- **bias:** array di celle, in cui ogni cella contiene un vettore colonna che contiene i bias di ciascun layer della rete. Il numero di celle dipende dal numero di layer; il numero di righe dei vettori colonna dipende dal numero di neuroni nel layer successivo;
- **layersInfo:** array di celle, dove il numero di celle è uguale al numero di layer della rete neurale che si sta costruendo, in cui ogni cella contiene una struttura rappresentante un layer della rete neurale, avente i seguenti campi:
  - **neuronsNum:** numero di neuroni;
  - **activationFunction:** funzione di attivazione;
- **layersNum:** numero di layer della rete neurale;
- **activationValue:** array di celle, in cui ogni cella contiene una matrice che contiene i valori di attivazione di ciascun layer. Il numero di celle dipende dal numero di layer. La dimensione delle matrici dipende dal numero di neuroni e dal numero di istanze prese in input, in cui ci sono tante righe quante il numero di neuroni del layer successivo e tante colonne quante il numero di istanze prese in input;
- **output:** array di celle, in cui ogni cella contiene una matrice che contiene i valori di uscita di ciascun layer. Per quanto riguarda le dimensioni vale lo stesso discorso fatto per i valori di attivazione.

Property ^	Value
 weight	2x1 cell
 bias	2x1 cell
 layersInfo	3x1 cell
 layersNum	3
 activationValue	2x1 cell
 output	2x1 cell

### 3.3. Principali algoritmi realizzati

In questo paragrafo sono illustrate più nel dettaglio le funzionalità richieste dalla parte A dal punto di vista implementativo.

#### 3.3.1 Forward propagation

Di seguito è illustrato l'algoritmo per la propagazione in avanti di una rete neurale multi-strato.

```
1 function net = forwardPropagation(input, net, errorFunction)
2     z = input;
3     for layer = 1 : net.getLayersNum()-1
4         a = (net.getNetWeight(layer) * z) + net.getNetBias(layer);
5         f = net.getActivationFunction(layer);
6         z = f(a);
7
8         net = net.setActivationValue(layer, a);
9         net = net.setOutput(layer, z);
10    end
11    g = getLastLayerFunction(errorFunction);
12    net = net.setOutput(layer, g(z));
13 end
```

La funzione prende i seguenti parametri in input:

- **input:** matrice contenente le istanze del training set, avente numero di righe pari al numero di neuroni dello strato di input della rete e numero di colonne pari al numero di istanze del training set;
- **net:** [struttura di una rete neurale](#)
- **errorFunction:** funzione di errore che consente di capire quale funzione di attivazione applicare all'output della rete.

#### Descrizione dell'algoritmo:

Per prima cosa si copia l'input in una variabile z, dopodiché si itera lungo gli strati della rete eccetto quello di output; ad ogni iterazione, si calcola l'input di ciascun neurone mediante una somma pesata, al quale contribuisce anche un bias. A questo punto, si recupera la funzione di

attivazione associata a quel layer e la si applica all'input calcolato precedentemente per ottenere l'output dei neuroni dello strato corrente; a quel punto si chiamano due metodi della rete neurale per memorizzare i nuovi input e output calcolati per quello strato. Terminata l'iterazione, si recupera la funzione di attivazione da applicare all'ultimo strato della rete, la quale dipende dalla funzione di errore che si è scelta; a quel punto la si applica all'output del penultimo strato e si aggiorna così l'output della rete. In output viene restituita la rete neurale con i parametri aggiornati.

### 3.3.2 Backpropagation

Di seguito è illustrato l'algoritmo per la retropropagazione dell'errore di una rete neurale multi-strato.

```
1 function delta = backPropagation(net, labels, errorFunction)
2     layersNum = net.getLayersNum();
3     delta = cell(1, layersNum-1);
4
5     % Calcolo i delta dell'ultimo strato
6     derivFun = getDerivatives(net.getActivationFunction(layersNum-1));
7     a = net.getActivationValue(layersNum-1);
8     delta{layersNum-1} = derivFun(a);
9     derivFun = getDerivatives(errorFunction);
10    output = net.getOutput(net.getLayersNum()-1);
11    delta{layersNum-1} = delta{layersNum-1} .* derivFun(output, labels);
12
13    % Calcoli i delta degli strati interni
14    for layer = layersNum-2 : -1 : 1
15        w = net.getNetWeight(layer+1);
16        delta{layer} = w' * delta{layer+1};
17
18        derivFun = getDerivatives(net.getActivationFunction(layer));
19        a = net.getActivationValue(layer);
20        delta{layer} = delta{layer} .* derivFun(a);
21    end
22 end
```

La funzione prende i seguenti parametri in input:

- **net:** [struttura di una rete neurale](#)
- **labels:** matrice contenente le etichette associate alle immagini del training set, avente stesse dimensioni della matrice di output;
- **errorFunction:** funzione di errore che consente di calcolare la derivata della funzione di errore.

## Descrizione dell'algoritmo:

Per prima cosa si recupera il numero di strati della rete e si inizializza un array di celle per il calcolo dei delta. Si comincia poi con il calcolo dei delta associati all'ultimo strato della rete, per cui si recupera la derivata della funzione di attivazione e i valori di attivazione dei neuroni, a questo punto si inizia a calcolare la derivata della funzione di attivazione dell'ultimo strato e la si moltiplica per la derivata della funzione di errore rispetto all'output predetto, secondo la seguente formula:

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

A questo punto si continua con il calcolo dei delta degli strati interni, per cui si itera a ritroso a partire dallo strato precedente il penultimo fino al primo; ad ogni iterazione si recuperano i pesi dello strato successivo, si calcola la sommatoria dei prodotti tra i delta e i pesi dello strato successivo e a quel punto si moltiplica tale somma per la derivata della funzione di attivazione dello strato corrente, ottenendo così i delta. Il tutto seguendo la formula:

$$\delta_i^n = f'_n(a_i^n) \cdot \sum_j (\delta_j^n \cdot w_{ji})$$

Al termine delle iterazioni si restituiscono i delta così calcolati per gli strati interni e per lo strato di output. A questo punto resta da calcolare le derivate; tale operazione viene svolta da una funzione a parte, che è la seguente:

```
1  function [derW,derB] = calcolaDerivate(net, delta, input)
2      layersNum = net.getLayersNum();
3      derW = cell(layersNum-1, 1);
4      derB = cell(layersNum-1, 1);
5
6      z = input;
7      for layer = 1 : layersNum-1
8          derW{layer} = delta{layer} * z';
9          derB{layer} = sum(delta{layer}, 2);
10         z = net.getOutput(layer);
11     end
12 end
```

Tale funzione prende in input i seguenti parametri:

- **net:** [struttura di una rete neurale](#)
- **delta:** array di celle, in cui sono contenute le matrici contenenti i delta calcolati nella fase di back-propagation, avente numero di righe che dipende dal numero di neuroni degli strati della rete (escluso quello di input) e numero di colonne pari al numero di istanze del training set;
- **input:** matrice contenente le istanze del training set, avente numero di righe pari al numero di neuroni dello strato di input della rete e numero di colonne pari al numero di istanze del training set;

### **Descrizione dell'algoritmo:**

Per prima cosa si calcola il numero di strati della rete per poter inizializzare gli array di celle che conterranno le derivate da calcolare; a quel punto si copia l'input in una variabile  $z$  e si itera lungo gli strati della rete. Ad ogni iterazione si calcolano le derivate dei pesi moltiplicando i delta dello strato corrente per l'input di tale strato; mentre per i bias occorre una semplice somma dei delta dello strato corrente.



## 4. Parte B

### 4.1. Ulteriori funzionalità principali

In questo paragrafo sono illustrate le restanti funzionalità principali che sono state implementate, le quali fanno riferimento alla parte B della traccia assegnata.

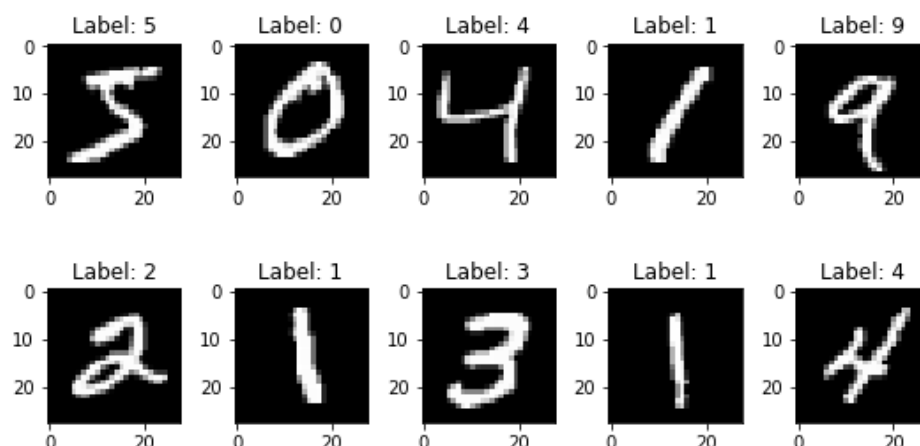
#### 4.1.1 Caricamento e divisione dataset MNIST

Il dataset MNIST è una raccolta di 70000 immagini etichettate, di cui 60000 sono dedicate al training set e le restanti 10000 sono dedicate al test set. Il criterio di divisione del dataset adottato è il seguente:

- **Training set:** dedicato alla fase di addestramento della rete.
- **Test set:** dedicato alla fase di test della rete, mediante la quale è possibile calcolare l'accuratezza raggiunta dal modello.
- **Validation set:** estrapolato dal training set per valutare la rete ottenuta dal processo di learning.

Il criterio di divisione del dataset è ~70% per il training set, ~15% per il test set e ~15% per il validation set.

La dimensione di questi insiemi verrà fatta variare a fini sperimentali.



*Esempi di immagini contenute nel dataset MNIST*

#### 4.1.2 Stochastic gradient descent

Si vuole minimizzare l'errore commesso, il quale, fissati gli iperparametri come dataset e modello della rete, dipende solo dai parametri  $\theta$ . Dunque, si dice che la funzione di errore  $E$  è in funzione di  $\theta$  e ciò che si vuole trovare è:

$$\theta^* = \arg \min_{\theta \in \Theta} E(\theta)$$

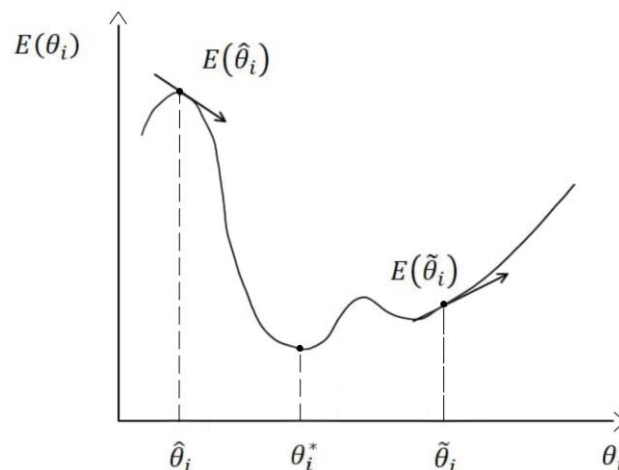
Per calcolare il minimo di questa funzione adoperiamo un metodo, il quale non è l'unico possibile, noto come discesa del gradiente.

Il gradiente di una funzione di errore  $E$  è il vettore delle derivate parziali:

$$\nabla E(\theta) = \left[ \frac{\partial E}{\partial \theta_1}, \frac{\partial E}{\partial \theta_2}, \dots, \frac{\partial E}{\partial \theta_p} \right]$$

dove  $\theta_i$  è il singolo peso (o bias)  $\forall i = 1, \dots, p$ .

Il gradiente ci fornisce informazioni sulla direzione e verso in cui spostarsi nello spazio dei parametri con lo scopo di raggiungere il minimo. Consideriamo il seguente grafico:



Sia  $\hat{\theta}_i$  il valore dei parametri attuale, allora per poter passare a  $\theta_i^*$  ottengo informazioni da  $\left. \frac{\partial E}{\partial \theta_i} \right|_{\theta_i = \hat{\theta}_i} < 0$  (vettore decrescente), di conseguenza si incrementa  $\hat{\theta}_i$  per avvicinarci a  $\theta_i^*$ .

Se invece il valore dei parametri attuale è  $\tilde{\theta}_i$ , allora per poter passare a  $\theta_i^*$  ottengo informazioni da  $\left. \frac{\partial E}{\partial \theta_i} \right|_{\theta_i=\tilde{\theta}_i} > 0$  (vettore crescente), di conseguenza si decrementa  $\tilde{\theta}_i$  per avvicinarci a  $\theta_i^*$ .

In entrambi i casi si esegue l'aggiornamento secondo la seguente regola:

$$\theta_i \leftarrow \theta_i - \eta \left. \frac{\partial E}{\partial \theta_i} \right|_{\theta_i=\hat{\theta}_i}$$

Dove  $\eta$  è detto learning rate e ci consente di regolare la dimensione del passo di aggiornamento; in particolare  $0 < \eta < 1$ . Il segno meno ci permette di spostarci in direzione opposta rispetto al segno della derivata.

#### 4.1.3 Resilient backpropagation

La Resilient backpropagation è una regola di aggiornamento dei pesi, il cui obiettivo principale, oltre che un miglioramento dell'efficacia, è la mancata dipendenza dagli iperparametri, in particolare dal learning rate  $\eta$ . L'idea di base è:

1. Associare a ciascun parametro  $w_{ij}$  un parametro che rappresenta il passo di aggiornamento  $\Delta_{ij}$ ;
2. Modificare opportunamente i  $\Delta_{ij}$  durante l'apprendimento.

Dunque, sia  $g_{ij}^{(t)} = \frac{\partial E^{(t)}}{\partial w_{ij}}$ , allora la regola di aggiornamento diventa:

$$w_{ij} \leftarrow w_{ij} - \text{sign}(g_{ij}^{(t)}) \cdot \Delta_{ij}$$

Quindi a seconda del segno della derivata si incrementa o decrementa il peso di una quantità  $\Delta_{ij}$ . Tale aggiornamento avviene considerando le derivate all'epoca corrente  $g_{ij}^{(t)}$  e quella all'epoca precedente  $g_{ij}^{(t-1)}$ . Guardando le due derivate delle due epoche è possibile capire il salto effettuato.

In generale vale che:

- Se  $g_{ij}^{(t)} \cdot g_{ij}^{(t-1)} > 0$  allora le derivate hanno senso concorde e il salto effettuato è “corretto”, quindi possiamo incrementare il parametro  $\Delta_{ij}$ ;
- se  $g_{ij}^{(t)} \cdot g_{ij}^{(t-1)} < 0$  allora le derivate hanno senso discorde e il salto effettuato è stato troppo grande, quindi occorre decrementare il parametro  $\Delta_{ij}$ .

Per poter incrementare e decrementare  $\Delta_{ij}$  si introducono due nuovi iperparametri, che sono:

- $\eta^+$  che permette di incrementare  $\Delta_{ij}$ , per cui  $\eta^+ > 1$ ;
- $\eta^-$  che permette di decrementare  $\Delta_{ij}$ , per cui  $0 < \eta^- < 1$ ;

A questo punto la regola di aggiornamento del parametro  $\Delta_{ij}$  diventa la seguente:

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

dove la dimensione del passo di aggiornamento è limitata inferiormente e superiormente da  $\Delta_{min}$  e  $\Delta_{max}$ .

Anche se all'apparenza il numero di iperparametri è aumentato ( $\Delta_{max}$ ,  $\Delta_{min}$ ,  $\eta^-$ ,  $\eta^+$ ), in letteratura si conoscono valori che funzionano tipicamente bene.

## 4.2. Principali algoritmi realizzati

In questo paragrafo sono illustrate più nel dettaglio le funzionalità richieste dalla parte B dal punto di vista implementativo.

### 4.2.1 Stochastic gradient descent

Questo algoritmo implementa la discesa stocastica del gradiente, la quale verrà utilizzata come algoritmo di aggiornamento dei pesi e dei bias solamente la prima volta, dopodiché l'algoritmo di aggiornamento utilizzato è RProp, come richiesto dalla traccia.

```
1 function net = gradientDescent(net, derW, derB, eta)
2     for layer = 1: net.getLayersNum()-1
3         net = net.setNetWeight(layer, net.getNetWeight(layer) - (eta * derW{layer}));
4         net = net.setNetBias(layer, net.getNetBias(layer) - (eta * derB{layer}));
5     end
6 end
```

La funzione prende i seguenti parametri in input:

- **net:** [struttura di una rete neurale](#)
- **derW:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene una matrice che contiene le derivate dei pesi calcolate nella fase di backpropagation precedente.
- **derB:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene un vettore colonna che contiene le derivate dei bias calcolate nella fase di backpropagation precedente.
- **eta:** iperparametro fissato dall'utente che consiste nella grandezza del passo di aggiornamento, in particolare  $0 < eta < 1$ .

### Descrizione dell'algoritmo:

Si itera lungo gli strati della rete neurale e ad ogni iterazione si aggiornano i pesi e i bias seguendo la regola della discesa del gradiente:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

$$b_{ij} \leftarrow b_{ij} - \eta \frac{\partial E}{\partial b_{ij}}$$

## 4.2.2 Resilient backpropagation

Di seguito è illustrato il codice che implementa la regola di aggiornamento dei pesi tramite RProp.

```
1 function [net, deltaW, deltaB] = rprop(net, derW, derB, precDerW, precDerB, deltaW, deltaB, etaP, etaN)
2     for layer = 1 : net.getLayersNum()-1
3         % Calcolo il segno del prodotto tra le nuove g(t) e le vecchie g(t-1) derivate (t = epoca)
4         % prodSign = sign(g(t-1) * g(t))
5         prodSign = sign(derW{layer} .* precDerW{layer});
6
7         % Calcolo i delta dei pesi
8         deltaW = getDelta(deltaW, layer, prodSign, etaP, etaN);
9
10        % Aggiornamento dei pesi
11        % Δw(t) = -sign(g(t)) * Δ(t)
12        % w(t) = w(t) + Δw(t)
13        deltaW_t = (-sign(derW{layer})) .* deltaW{layer};
14        net = net.setNetWeight(layer, net.getNetWeight(layer) + deltaW_t);
15
16        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
17
18        % Eseguo le operazioni precedenti ma questa volta per i bias
19        % prodSign = sign(g(t-1) * g(t))
20        prodSign = sign(derB{layer} .* precDerB{layer});
21
22        % Calcolo i delta dei bias
23        deltaB = getDelta(deltaB, layer, prodSign, etaP, etaN);
24
25        % Aggiornamento dei bias
26        % Δb(t) = -sign(g(t)) * Δ(t)
27        % b(t) = b(t) + Δb(t)
28        deltaB_t = (-sign(derB{layer})) .* deltaB{layer};
29        net = net.setNetBias(layer, net.getNetBias(layer) + deltaB_t);
30    end
31 end
32
33 function delta = getDelta(delta, layer, prodSign, etaP, etaN)
34     % Parametri di default della rProp
35     deltaMin = 1e-6;
36     deltaMax = 50;
37
38     % Calcolo i delta
39     for i = 1 : size(prodSign, 1)
40         for j = 1 : size(prodSign, 2)
41             if prodSign(i, j) > 0
42                 % Δij(t) = min(η+ * Δij(t), Δmax)
43                 delta{layer}(i, j) = min(etaP * delta{layer}(i, j), deltaMax);
44             elseif prodSign(i, j) < 0
45                 % Δij(t) = max(η- * Δij(t), Δmin)
46                 delta{layer}(i, j) = max(etaN * delta{layer}(i, j), deltaMin);
47             %}
48             else prodSign(prodSign_i, prodSign_j) == 0
49                 Δij(t) = Δij(t)
50             %}
51         end
52     end
53 end
54 end
```

La funzione prende i seguenti parametri in input:

- **net:** [struttura di una rete neurale](#)
- **derW:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene una matrice che contiene le derivate dei pesi calcolate nella fase di backpropagation nell'epoca corrente.
- **derB:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene un vettore colonna che contiene le derivate dei bias calcolate nella fase di backpropagation nell'epoca corrente.
- **precDerW:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene una matrice che contiene le derivate dei pesi calcolate nella fase di backpropagation nell'epoca precedente.
- **precDerB:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene un vettore colonna che contiene le derivate dei bias calcolate nella fase di backpropagation nell'epoca precedente.
- **deltaW:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene una matrice che inizialmente è inizializzata con tutti i valori uguali a un iperparametro fissato dall'utente; dopodiché la matrice conterrà i valori dei delta dei pesi calcolati nelle epoche precedenti da questo stesso algoritmo.
- **deltaB:** array di celle, in cui il numero di celle dipende dal numero di strati della rete neurale; ogni cella contiene una matrice che inizialmente è inizializzata con tutti i valori uguali a un iperparametro fissato dall'utente; dopodiché la matrice conterrà i valori dei delta dei bias calcolati nelle epoche precedenti da questo stesso algoritmo.
- **etaP:** iperparametro fissato dall'utente che consente di incrementare i delta; in particolare  $etaP > 1$ .
- **etaN:** iperparametro fissato dall'utente che consente di decrementare i delta; in particolare  $0 < etaN < 1$ .

### **Descrizione dell'algoritmo:**

Si itera lungo gli strati della rete; ad ogni iterazione, calcolo il segno del prodotto tra le derivate dell'epoca corrente e quelle dell'epoca precedente. A questo punto procedo con il calcolo dei delta dei pesi richiamando la funzione getDelta(), la quale itera lungo la matrice dei segni calcolata precedentemente e a seconda del segno calcola la seguente:

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

Fatto ciò, si procede con l'aggiornamento dei pesi, ricordando in particolare che l'aggiornamento è basato sulla seguente regola:

$$\begin{aligned} \Delta_{w(t)} &= -\text{sign}(g^{(t)}) \cdot \Delta_t \\ w^{(t)} &= w^{(t-1)} + \Delta_{w(t)} \end{aligned}$$

A questo punto, si ripetono i passaggi precedenti per il calcolo dei delta dei bias.



## 4.3. Parte sperimentale

### 4.3.1 Inizializzazione del modello

In questa fase di sperimentazione per l'aggiornamento dei pesi è stata fissata la resilient backpropagation (RProp) come richiesto dalla traccia, eccetto alla prima epoca in cui si utilizza la discesa stocastica del gradiente. Di seguito sono illustrati gli iperparametri che si sono fissati per gli esperimenti:

- Dimensione training set: 50000
- Dimensione validation set: 10000
- Dimensione test set: 10000
- Numero di epoche: 200
- $\eta$ : 0,00001
- $\eta^+$ : 1,2
- $\eta^-$ : 0,5
- Valore iniziale dei  $\Delta_{ij}$  per l'algoritmo di RProp: 0,001
- Soglia per il criterio di early-stopping: 10
- Inizializzazione dei pesi: [-0.1,+0.1]
- Funzione di errore: Cross-Entropy + Softmax

Sempre facendo riferimento alla traccia, il numero di strati interni varia da 1 a 5, così come il numero di nodi che li compongono. Infine, si confronta il caso in cui si utilizza come funzione di attivazione la sigmoide con quello in cui si usa come funzione di attivazione la ReLu.

### 4.3.2 Divisione del dataset

Per le sperimentazioni effettuate è stato utilizzato il dataset MNIST; il dataset estratto viene poi sottoposto a una procedura di shuffle, in cui i dati vengono rimescolati allo scopo di migliorare la generalizzazione del modello e ridurre così il rischio di overfitting. Ci si vuole assicurare quindi che i set di addestramento, test e validazione siano rappresentativi della distribuzione complessiva dei dati.

In particolare, la funzione per lo shuffle dei dati è la seguente:

```
61  function set = shuffleSet(dataSet)
62      index = randperm(size(dataSet.images, 2));
63      set.images = dataSet.images(:,index);
64      set.labels = dataSet.labels(index,:);
65  end
```

Dove dato il dataset, si genera un indice casuale ad ogni esecuzione che consente di rimescolare i dati. In particolare, questa operazione viene eseguita sia per il training set che per il test set; per quanto riguarda il validation set, questo viene estrapolato dal training set una volta che l'operazione di shuffle viene eseguita su di esso.

### 4.3.3 Fase di apprendimento

La fase di learning è stata implementata con una procedura di tipo batch, come richiesto dalla traccia. Con questo tipo di apprendimento, l'aggiornamento dei parametri della rete avviene passandole in input tutti i punti del training set, aggiornando così, ad ogni epoca, i pesi mediante la derivata dell'errore totale commesso. Il batch learning causa un'importante occupazione di memoria dovendo mantenere le derivate delle singole funzioni di errore  $E^{(n)}$  per poi calcolare quella complessiva:

$$E = \sum_{n=1}^N E^{(n)}$$

L'algoritmo di aggiornamento utilizzato è RProp, il quale può essere utilizzato solo con questo tipo di apprendimento.

Di seguito è illustrato lo pseudocodice del batch learning:

```
while (e <= MAX_EPOCH and not stopCriterion)
    [W, b, output] = forwardPropagation(trainingInput, W, b)
    delta = backPropagation(output, trainingTargets, W, b)
    derivatives = getDerivates(W, b, delta, trainingInput)

    if(e == 1)
        [W, b] = gradientDescent(W, b, derivatives, eta)
    else
        [W, b, deltaW, deltaB] = rprop(W, b, derivatives, oldDerivatives, deltaW, deltaB, etaP, etaN)
    end

    oldDerivatives = derivatives

    [W, b, output] = forwardPropagation(trainingInput, W, b)
    trainingLoss(e) = getLoss(output, trainingTargets)

    [W, b, output] = forwardPropagation(validationInput, W, b)
    validationLoss(e) = getLoss(output, validationTargets)

    if validationLoss(e) < minValidLoss
        minValidLoss = validationLoss(e)
        bestNet = [W, b]
    end

    if trainingLoss(e) < minTrainLoss
        minTrainLoss = trainingLoss(e)
    end

    stopCriterion = checkGeneralizationLoss(minValidLoss, validationLoss(e));
    e=e+1
end
```

Tramite un criterio di early stop, che consente di terminare l'apprendimento prima del raggiungimento del numero massimo di epoche, si raggiunge un compromesso tra complessità computazionale e generalizzazione. In particolare, il criterio di stop adottato prende il nome di *Generalization Loss*:

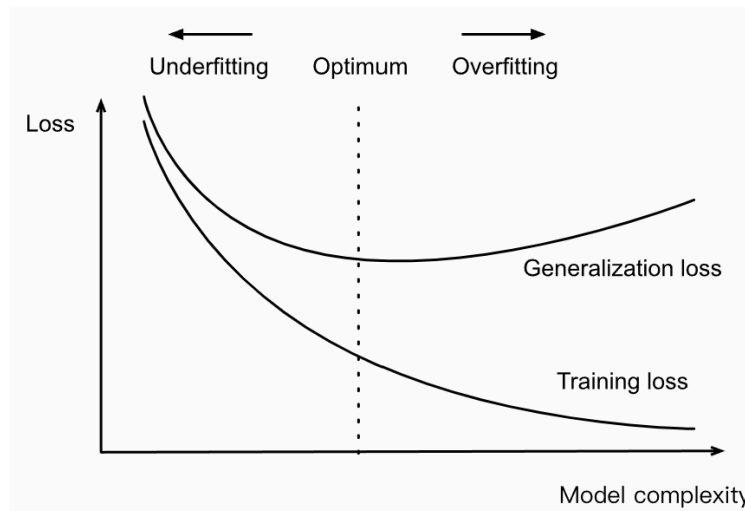


Figura 1: Influenza della complessità del modello su underfitting e overfitting

Come si evince dal grafico soprastante, tramite questo criterio possiamo interrompere la fase di apprendimento nel momento più opportuno.

#### 4.3.4 Risultati ottenuti

Al fine di automatizzare le sperimentazioni, le quali presentano un numero elevato di test case diversi, è stato creato uno script che leggeva da un file Excel i test da fare, creando di conseguenza la relativa architettura della rete per l'esecuzione del test, facendo quindi variare il numero di strati interni, numero di neuroni per ciascuno strato e le funzioni di attivazione. Dopodiché, riportava in questo stesso file i risultati ottenuti, ossia l'accuratezza e i valori minimi di errore sul training set e sul validation set; questo script esporta anche i grafici delle curve di errore in una cartella dedicata, unendo orizzontalmente il test eseguito con lo stesso modello di rete neurale ma utilizzando funzioni di attivazione differenti (Sigmoid e ReLU).

Come configurazioni della rete neurale sono state prese tutte le possibili combinazioni utilizzando 100 e 200 come numero di nodi per gli strati interni, considerando 1, 2, 3, 4, 5 strati interni.

Di seguito sono riportati i grafici delle funzioni di errore sul training set e sul validation set per le configurazioni appena citate. Al fine di poter eseguire un confronto equo tra le varie misurazioni, il criterio di early stop è stato disabilitato.

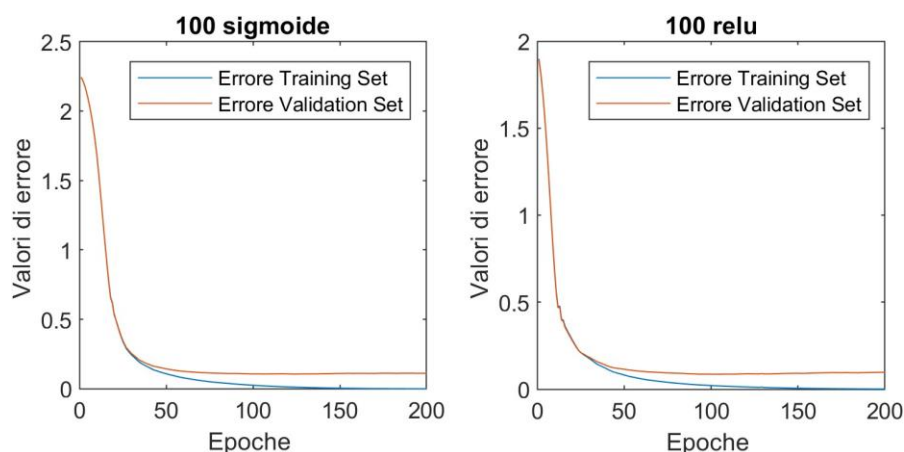


Figura 2: 100 nodi confronto Sigmoid e ReLU

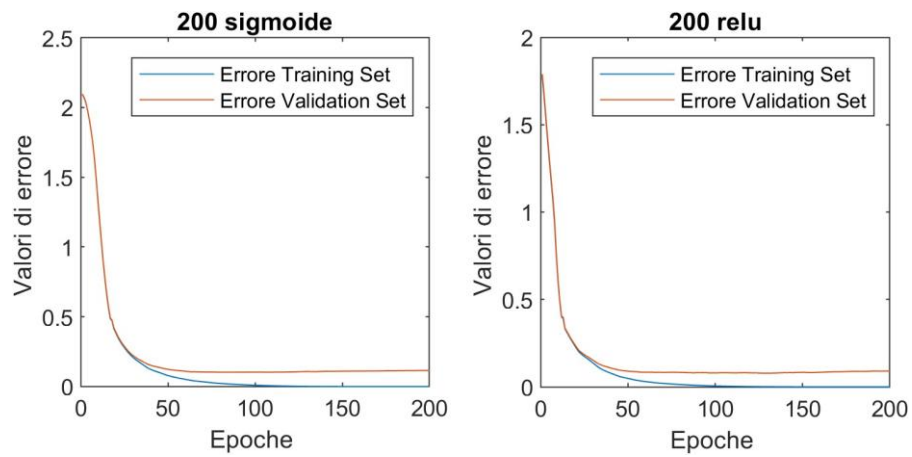


Figura 3: 200 nodi confronto Sigmoid e ReLU

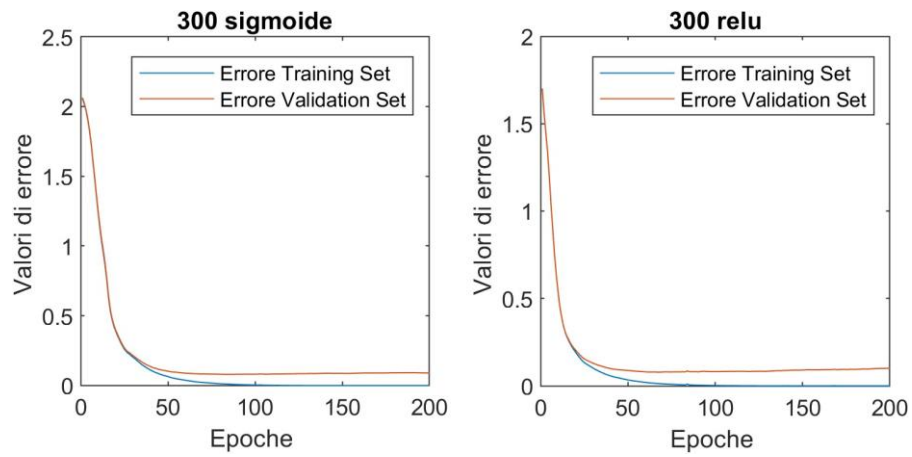


Figura 4: 300 nodi confronto Sigmoid e ReLU

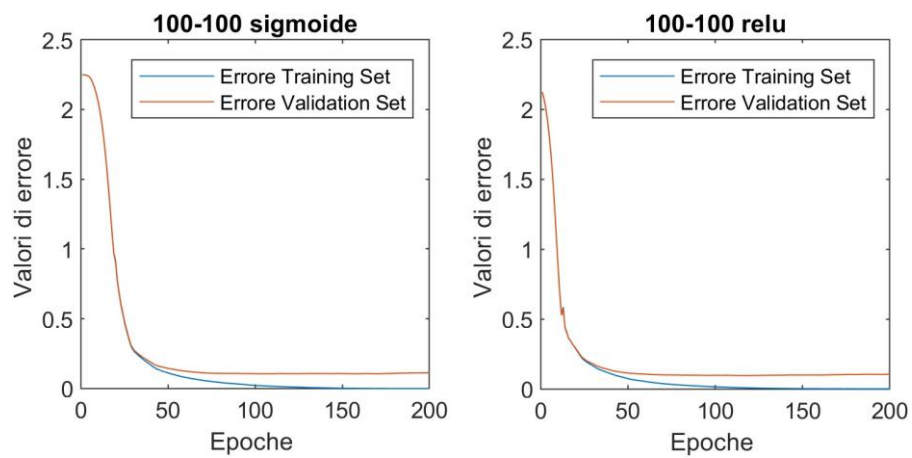


Figura 5: 100-100 nodi confronto Sigmoid e ReLU

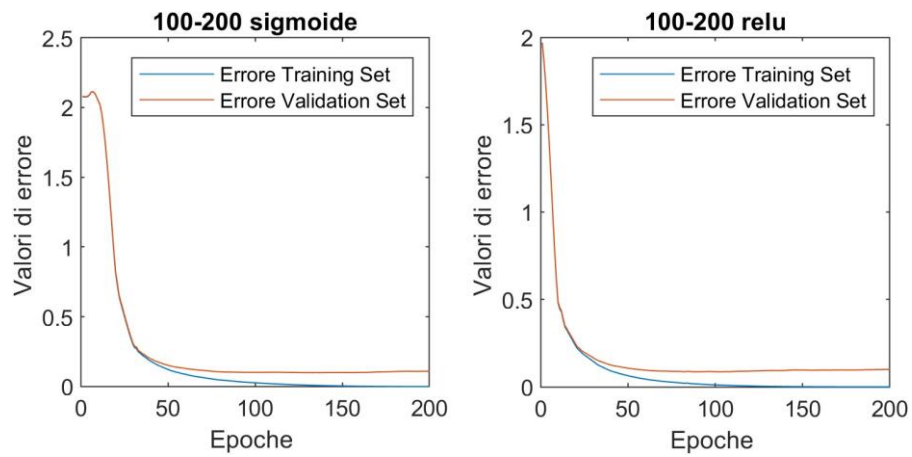


Figura 6: 100-200 nodi confronto Sigmoide e ReLU

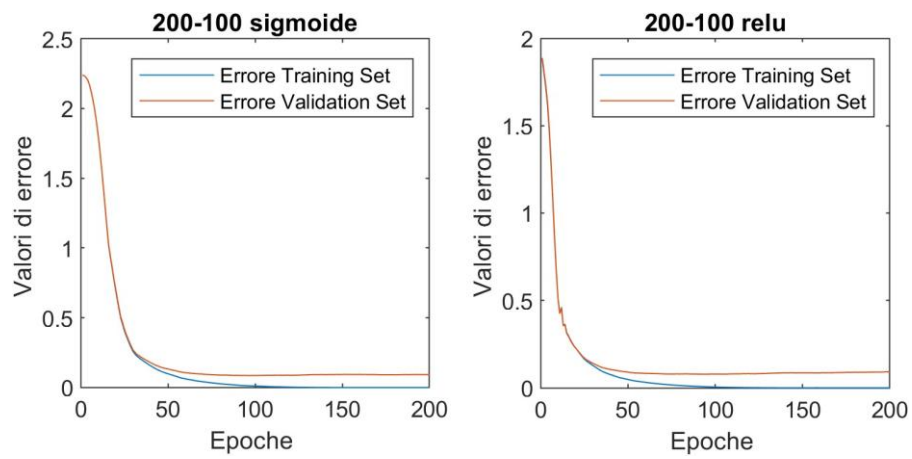


Figura 7: 200-100 nodi confronto Sigmoide e ReLU

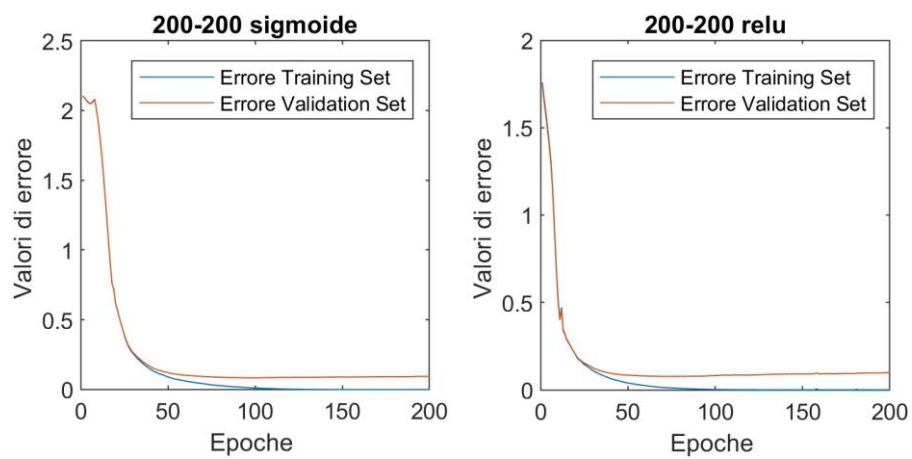


Figura 8: 200-200 nodi confronto Sigmoide e ReLU

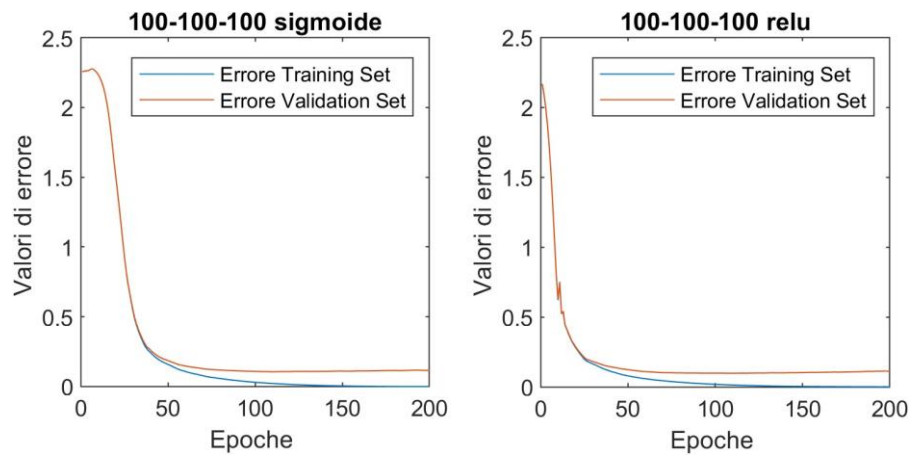


Figura 9: 100-100-100 nodi confronto Sigmoide e ReLU

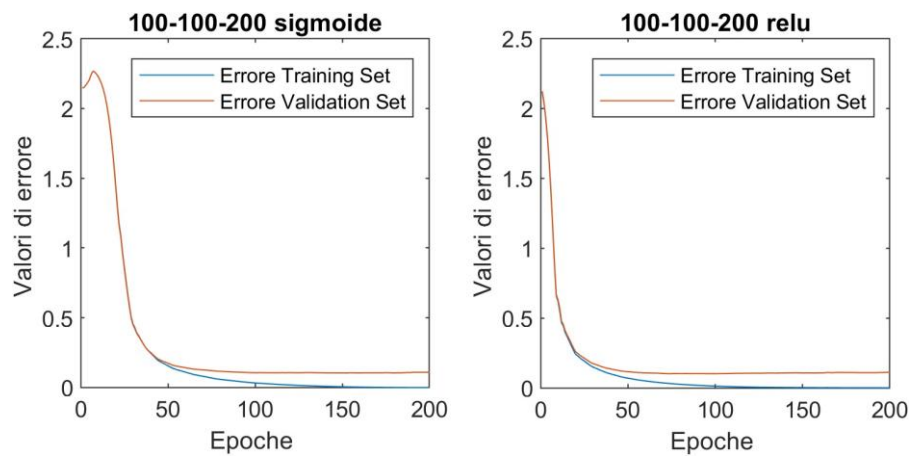


Figura 10: 100-100-200 nodi confronto Sigmoide e ReLU

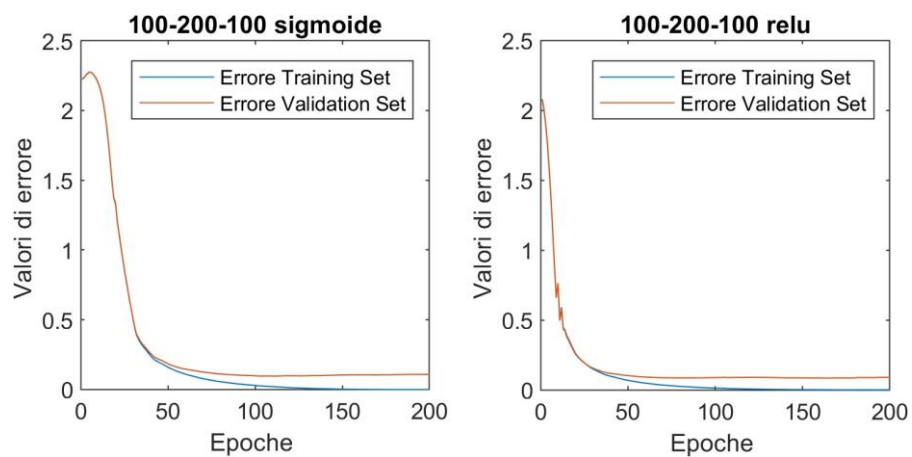


Figura 11: 100-200-100 nodi confronto Sigmoide e ReLU



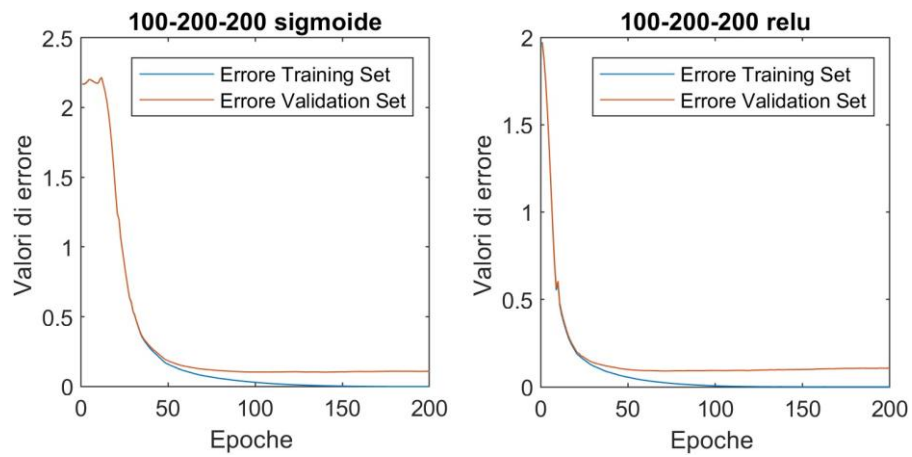


Figura 12: 100-200-200 nodi confronto Sigmoide e ReLU

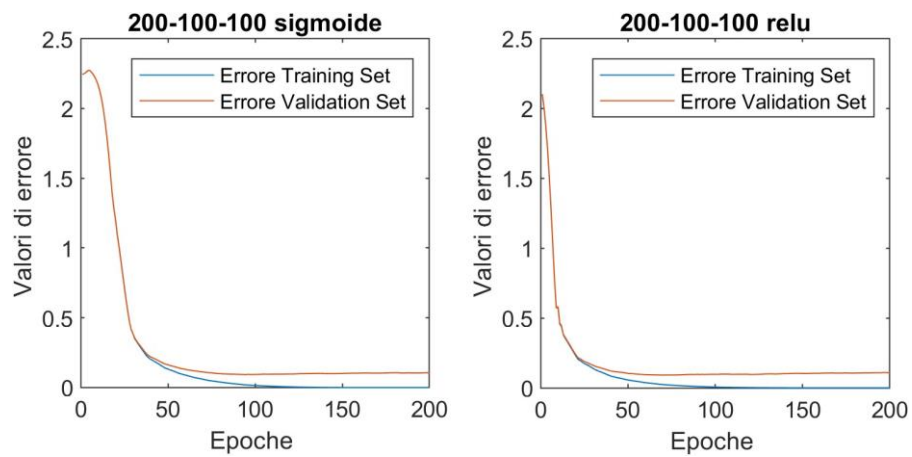


Figura 13: 200-100-100 nodi confronto Sigmoide e ReLU

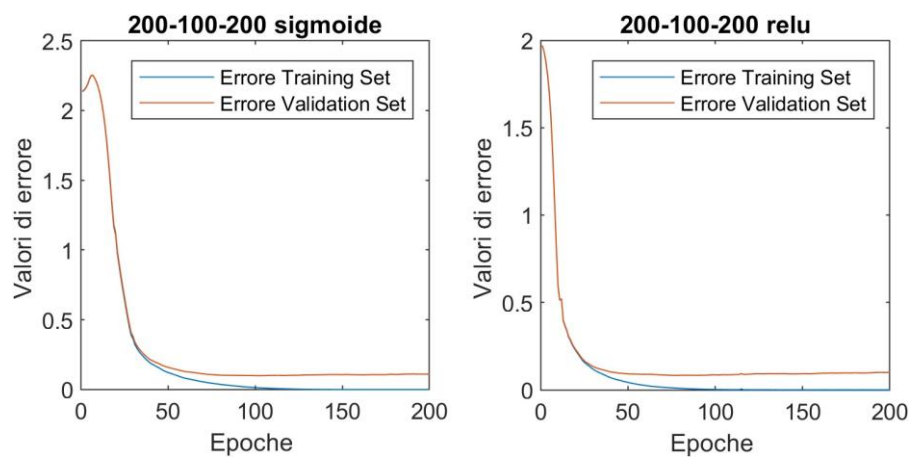


Figura 14: 200-100-200 nodi confronto Sigmoide e ReLU

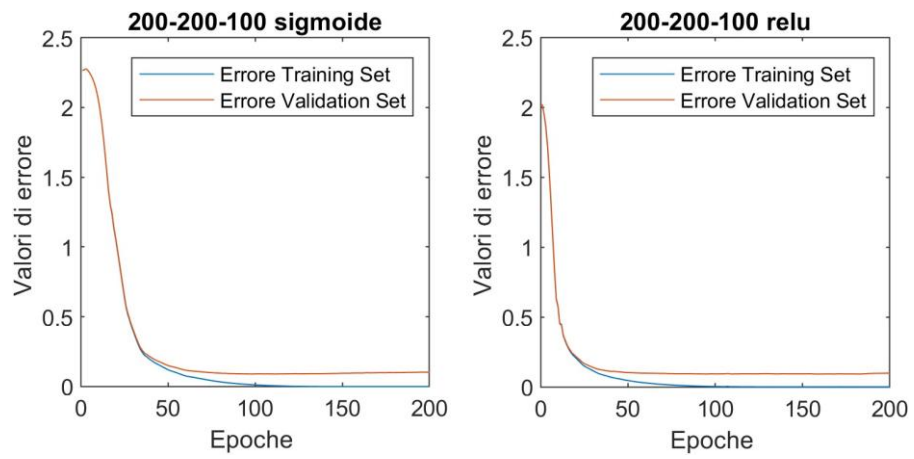


Figura 15: 200-200-100 nodi confronto Sigmoide e ReLU

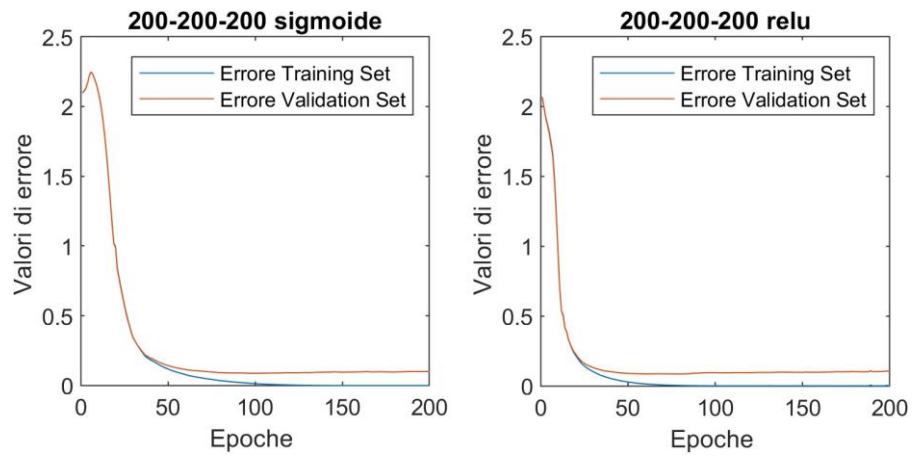


Figura 16: 200-200-200 nodi confronto Sigmoide e ReLU

Massima accuratezza (95,88%) con Sigmoide e 4 strati nascosti:

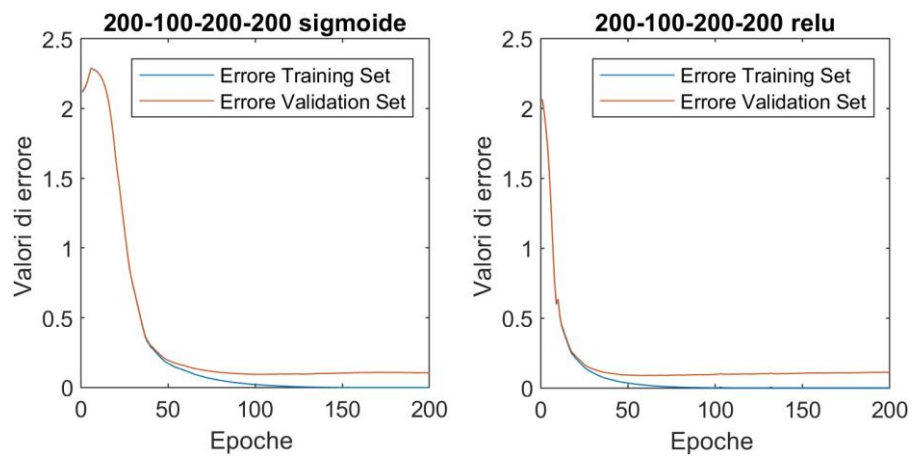


Figura 17: 200-100-200-200 nodi confronto Sigmoide e ReLU

Massima accuratezza (96,50%) con ReLU e 4 strati nascosti:

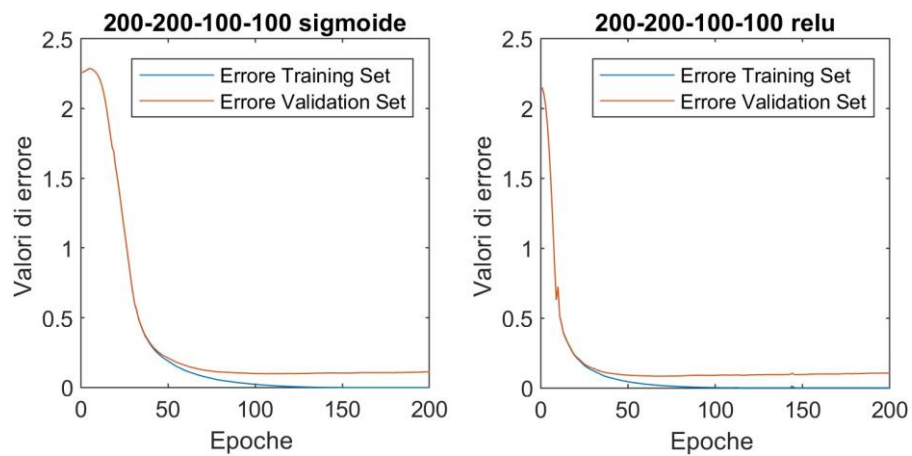


Figura 18: 200-200-100-100 nodi confronto Sigmoide e ReLU

Massima accuratezza (95,34%) con Sigmoid e 5 strati nascosti:

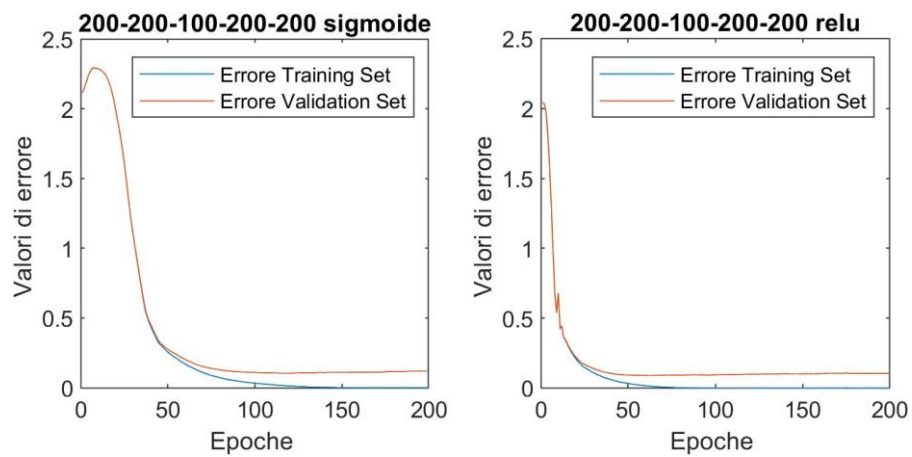


Figura 19: 200-200-100-200-200 nodi confronto Sigmoid e ReLU

Massima accuratezza (96,33%) con ReLU e 5 strati nascosti:

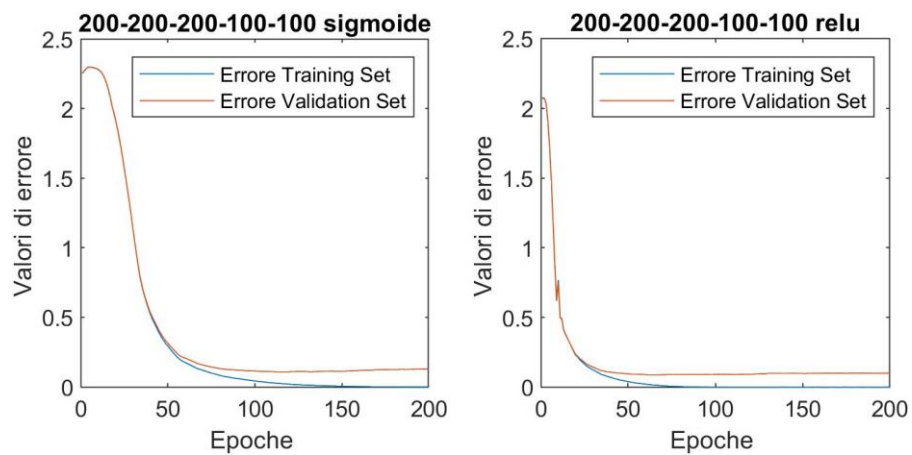


Figura 20: 200-200-200-100-100 nodi confronto Sigmoid e ReLU

Strati e nodi interni	Sigmoides			ReLU		
	Accuracy	Err min Train	Err min Val	Accuracy	Err min Train	Err min Val
100	95,84%	0,000589	0,106784	96,43%	0,000877	0,085793
200	96,49%	0,000006	0,102566	96,59%	0,000011	0,077924
300	96,63%	0,000000	0,080302	96,81%	0,000002	0,078312
100-100	95,64%	0,000073	0,105892	96,26%	0,000155	0,095098
100-200	95,93%	0,000169	0,099228	96,09%	0,000075	0,085927
200-100	96,32%	0,000000	0,085802	96,57%	0,000006	0,078042
200-200	96,49%	0,000000	0,083807	96,67%	0,000006	0,076702
100-100-100	95,56%	0,000078	0,106137	96,14%	0,000359	0,097270
100-100-200	95,46%	0,000167	0,104515	96,32%	0,000425	0,101084
100-200-100	95,82%	0,000028	0,096444	96,45%	0,000062	0,084115
100-200-200	95,58%	0,000052	0,102584	96,25%	0,000024	0,090489
200-100-100	96,19%	0,000000	0,093120	96,54%	0,000015	0,091577
200-100-200	96,10%	0,000000	0,099070	96,28%	0,000036	0,082623
200-200-100	96,60%	0,000000	0,090069	96,08%	0,000003	0,091727
200-200-200	96,26%	0,000000	0,087955	96,55%	0,000000	0,085001
100-100-100-100	95,03%	0,000897	0,120691	96,17%	0,000879	0,104440
100-100-100-200	95,21%	0,000389	0,113971	95,69%	0,000474	0,101852
100-100-200-100	95,54%	0,000411	0,104200	95,88%	0,000155	0,092601
100-100-200-200	95,55%	0,000013	0,107627	96,03%	0,000007	0,097525
100-200-100-100	95,14%	0,000188	0,111220	96,07%	0,000038	0,091390
100-200-100-200	95,31%	0,000028	0,111501	96,41%	0,000013	0,094125
100-200-200-100	95,36%	0,000289	0,109413	96,24%	0,000060	0,094886
100-200-200-200	95,46%	0,000084	0,116125	95,99%	0,000002	0,097764
200-100-100-100	95,82%	0,000000	0,087830	96,05%	0,000098	0,092078
200-100-100-200	95,70%	0,000000	0,101136	96,31%	0,000004	0,095258
200-100-200-100	95,76%	0,000000	0,101181	96,09%	0,000000	0,085281
200-100-200-200	95,88%	0,000000	0,094640	96,28%	0,000000	0,088515
200-200-100-100	95,76%	0,000000	0,098939	96,50%	0,000002	0,083608
200-200-100-200	95,79%	0,000000	0,098587	96,29%	0,000000	0,094293
200-200-200-100	95,83%	0,000002	0,108437	96,24%	0,000001	0,084646
200-200-200-200	95,62%	0,000000	0,096847	96,28%	0,000000	0,083323
100-100-100-100-100	94,06%	0,009258	0,137199	96,08%	0,000169	0,100205
100-100-100-100-200	94,91%	0,001638	0,119473	95,54%	0,000715	0,102036
100-100-100-200-100	94,59%	0,002796	0,126319	95,74%	0,000097	0,100101
100-100-100-200-200	94,52%	0,002455	0,142127	95,78%	0,000028	0,097643
100-100-200-100-100	94,60%	0,004431	0,130822	95,90%	0,000081	0,097168
100-100-200-100-200	94,94%	0,002677	0,132356	95,78%	0,000130	0,107682
100-100-200-200-100	94,85%	0,004034	0,126897	95,56%	0,000004	0,099594
100-100-200-200-200	94,78%	0,000687	0,125887	95,81%	0,000001	0,089400

100-200-100-100-100	94,62%	0,001398	0,115116	96,10%	0,000055	0,095636
100-200-100-100-200	95,16%	0,000575	0,117889	95,88%	0,000006	0,104867
100-200-100-200-100	94,65%	0,005522	0,116138	95,83%	0,000019	0,100844
100-200-100-200-200	95,14%	0,000823	0,115279	96,00%	0,000001	0,102108
100-200-200-100-100	94,67%	0,002346	0,117759	96,06%	0,000003	0,091446
100-200-200-100-200	94,60%	0,002694	0,118511	95,90%	0,000006	0,095211
100-200-200-200-100	94,97%	0,001200	0,107493	96,19%	0,000001	0,098364
100-200-200-200-200	95,03%	0,000137	0,119747	95,93%	0,000001	0,099799
200-100-100-100-100	94,84%	0,000493	0,121487	96,15%	0,000036	0,094915
200-100-100-100-200	94,80%	0,000305	0,119179	96,15%	0,000033	0,102617
200-100-100-200-100	95,04%	0,000121	0,120125	96,32%	0,000004	0,086529
200-100-100-200-200	95,22%	0,000007	0,113800	95,73%	0,000004	0,090268
200-100-200-100-100	94,83%	0,000042	0,115030	95,93%	0,000004	0,098144
200-100-200-100-200	94,97%	0,000043	0,118576	96,08%	0,000002	0,094882
200-100-200-200-100	94,94%	0,000085	0,116112	96,26%	0,000027	0,092602
200-100-200-200-200	95,31%	0,000014	0,110846	96,06%	0,000000	0,089429
200-200-100-100-100	95,15%	0,000008	0,119266	96,21%	0,000000	0,089636
200-200-100-100-200	95,26%	0,000013	0,113010	96,14%	0,000002	0,092071
200-200-100-200-100	94,80%	0,000362	0,118937	96,06%	0,000000	0,082660
200-200-100-200-200	95,34%	0,000000	0,104127	96,25%	0,000000	0,089879
200-200-200-100-100	95,26%	0,000101	0,107987	96,33%	0,000000	0,089177
200-200-200-100-200	95,15%	0,000017	0,114669	96,20%	0,000000	0,082243
200-200-200-200-100	95,33%	0,000003	0,108006	96,10%	0,000008	0,090694
200-200-200-200-200	95,26%	0,000001	0,108521	96,27%	0,000000	0,087762

*Tabella 1: Accuratezza errore training ed errore validation con Sigmoid e ReLU*

#### 4.3.5 Discussione dei risultati

Innanzitutto, i risultati ottenuti su tutte le diverse configurazioni sono molto positivi; in tutti i casi l'accuratezza si aggira intorno ai valori tra il 94% e il 97%. L'aggiornamento dei pesi tramite RProp ha consentito di raggiungere valori di errore molto bassi e in un numero di epoche relativamente piccolo. È possibile notare inoltre un andamento comune, da un certo punto in poi, l'errore di validazione inizia ad aumentare, mentre quello di addestramento continua a diminuire. Questo andamento è un sinonimo di overfitting, ossia quando il modello di rete neurale performa molto bene sui dati del training set a discapito della generalizzazione; per cui, la qualità delle classificazioni effettuate su altri dati è destinata a diminuire progressivamente. Per quanto riguarda il confronto tra i risultati ottenuti utilizzando la Sigmoidale e quelli ottenuti utilizzando la ReLU, essi si traducono in un risultato finale molto simile, ossia un modello avente un'accuratezza pressoché identica. Tuttavia, se si guarda il processo di apprendimento, la ReLU si comporta generalmente meglio. Infatti, all'aumentare del numero di strati, la curva di errore di ReLU decresce in maniera più regolare e in un numero di epoche minore rispetto alla Sigmoidale. Tale risultato è dovuto al fenomeno del vanishing gradient, poiché con più strati si ottiene una funzione di errore che presenta molti minimi locali, cioè punti in cui il gradiente è pari a 0. Questo può ostacolare il raggiungimento del minimo globale. Infatti, essendo la derivata della Sigmoidale compresa tra i valori  $[0, 0.25]$ , durante la fase di backpropagation 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, è evidente come si arrivi alla presenza di valori molto piccoli del gradiente. Di conseguenza, se i gradienti sono prossimi a 0, vuol dire che l'aggiornamento dei pesi sarà pressoché impercettibile. La ReLU, invece, non soffre del problema del vanishing gradient, in quanto per i nodi i cui input sono positivi, la derivata della ReLU è 1, e ciò garantisce che il gradiente non arrivi a valori molto vicini allo 0 durante la fase di backpropagation. Ecco perché dalle sperimentazioni effettuate è possibile notare che la ReLU converge verso l'errore minimo più velocemente di quanto lo fa la Sigmoidale.

#### 4.3.6 Conclusioni ed eventuali sviluppi futuri

In conclusione, possiamo affermare che per un training efficace non è necessario utilizzare un numero elevato di strati nascosti; in particolare, è sufficiente utilizzare un solo strato nascosto, in cui al suo interno è presente un numero di neuroni almeno pari a 100 per ottenere degli ottimi risultati di accuratezza e non eccedere drasticamente con i tempi computazionali.

Per quanto riguarda ulteriori sviluppi futuri che potrebbero rendere l'esperimento più completo, delle idee interessanti potrebbero essere le seguenti:

- Analizzare il comportamento della rete utilizzando funzioni di attivazione diverse al variare degli strati interni;
- utilizzare un altro algoritmo di aggiornamento dei pesi, come ad esempio la discesa del gradiente con momento, confrontando i risultati con quelli ottenuti con RProp;
- provare altre combinazioni degli iperparametri fissati per la sperimentazione precedentemente mostrata, ad esempio cambiando il learning rate e l'inizializzazione dei pesi con un range di valori diverso.