Università degli Studi di Napoli Federico II

Corso di Laurea Magistrale di *Informatica*

RELAZIONE DEL PROGETTO DI MACHINE LEARNING (Mod. B)

NEURAL NETWORK & DEEP LEARNING

Prof. Roberto Prevete

Dott. Francesco Calcopietro (Matr. N97000432)

A.A. 2023/2024

Sommario

Traccia

Il progetto proposto è stato concepito per la risoluzione della seguente traccia divisa in 2 parti.

Parte A.

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

Parte B.

Dato il dataset "minist" di immagini di cifre scritte a mano (http://yann.lecun.com/exdb/mnist/), risolvere tale problema: 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 e test set (considerare almeno 10000 elementi per il training set e 2500 per il test set). Si fissi la classica discesa del gradiente come algoritmo di aggiornamento dei pesi. 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) con uno solo strato di nodi interni al variare della modalità di apprendimento: online, batch e mini-batch. Si faccia tale studio per almeno 3 dimensioni diverse (numeri di nodi) per lo strato interno. Scegliere e mantenere invariati le funzioni di attivazione. Se è necessario, per questioni di tempi computazionali e spazio in memoria, si possono ridurre le dimensioni delle immagini raw del dataset mnist (ad esempio utilizzando in matlab la funzione imresize).

Introduzione

Dall'analisi della traccia proposta si intuisce che, per la risoluzione di quest'ultima, si necessita della realizzazione e dell'uso di una Rete Neurale Artificiale per effettuare un'operazione di classificazione di una serie di dati. Più nello specifico, si richiede la presenza di una Vanilla Network o Shallow Network o, ancora, Rete Neurale mono-strato, per fare ciò.
I dati su cui si intende lavorare saranno immagini di uno dei dataset più famosi nell'ambito della classificazione: il dataset MNIST.
La traccia dà libera scelta allo sviluppatore sia sulla tipologia di funzione d'errore da usufruire sia sulle funzioni d'attivazione da applicare ai vari strati della rete.
Si richiede lo studio dell'apprendimento della rete su tutt'e tre le modalità studiate durante il corso: modalità batch, mini-batch e online.
Si richiede, inoltre, come metodo di valutazione della capacità di generalizzazione della rete, l'uso di un validation-set e, come metodo di valutazione della bontà della rete, il metodo Hold-Out, cioè l'uso di un Test-set e, di conseguenza, il calcolo dell'accuratezza della rete su quest'ultimo.
La seguente relazione ha come scopo sia di considerare, seppur non in larga misura, concetti teorici da menzionare per comprendere al meglio la soluzione proposta, sia i vari step, nel dettaglio, della soluzione stessa, descrivendone le routine che compongono la libreria.
Il seguente progetto è disponibile sotto licenza Apache 2.0 al seguente link GitHub: https://github.com/kekkokalko/Vanilla-Network-per-la-classificatione-di-immagini.

Reti Neurali Artificiali Feed-Forward

Una rete neurale artificiale (in inglese **Artificial Neural Network**, abbreviato in **ANN** o anche come **NN**) è un modello di calcolo basato fortemente su un grade numero di unità elementari chiamati **neuroni**. Tali neuroni sono interconnessi fra loro. Per tal motivo, tali modelli sono anche chiamati **modelli connessionisti**. Le caratteristiche di tale modello computazionale sono molto correlate a quelle del sistema nervoso centrale presente negli esseri umani.

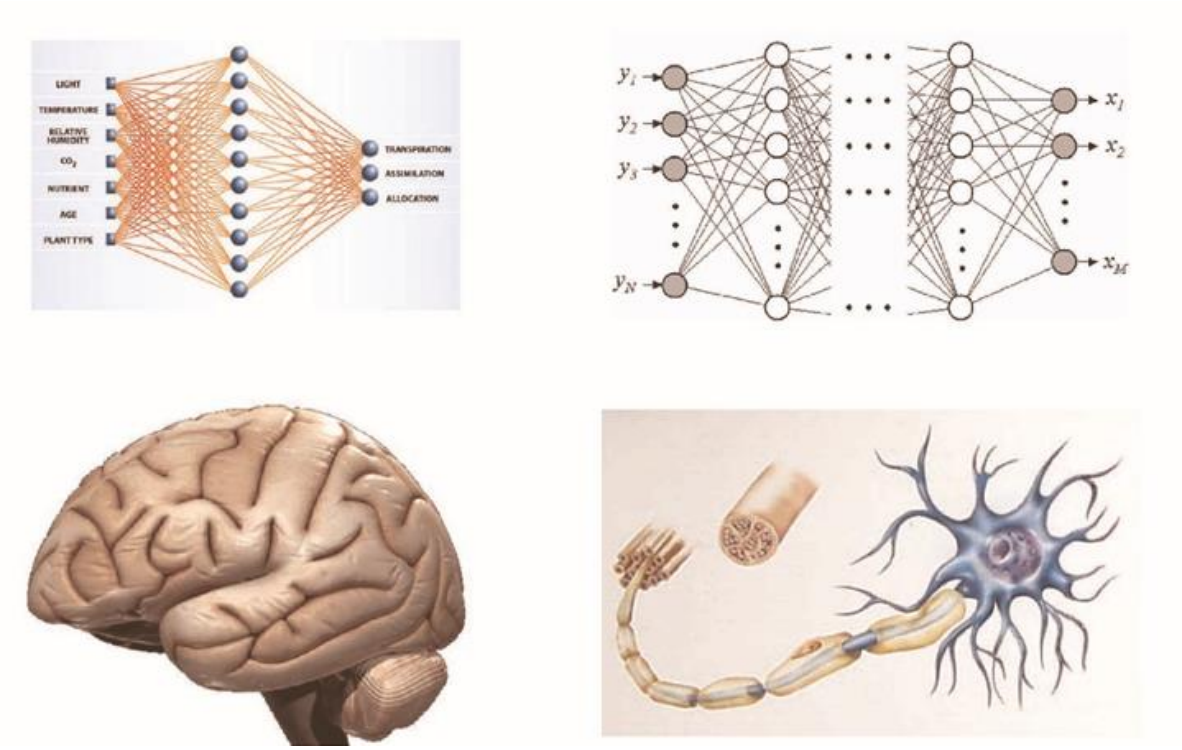


Figura 1 Una ANN e il Sistema Nervoso Centrale

Tali modelli, pertanto, vengono usufruiti per la risoluzione di vaste gamme di problemi, più o meno complessi, e i neuroni che ne fanno parte contribuiscono alla definizione delle soluzioni.
Nell'ambito del Machine Learning esistono varie tipologie di ANN. Una fra queste è la così detta **Rete Neurale Artificiale Feed-Forward.** Di seguito si analizza la struttura di un neurone e, conseguentemente, quella di una rete di tal genere. Tali strutture costituiscono la base della soluzione al problema di classificazione proposto.

Struttura

Essendo una NN, chiaramente una NN Feed-Forward basa la propria esistenza sui neuroni. Questi ultimi possiedono una loro struttura e svolgono compiti precisi.
Per quanto riguarda la struttura, un neurone è caratterizzato dalla presenza di:

1. d **variabili di input**, che compongono un vettore **x** $\in \boldsymbol{R^d}$, generalmente o il così detto **vettore di features** o l'insieme degli output dei neuroni connessi con il neurone corrente.
2. d **linee di input** o **connessioni di input** che permettono il trasferimento delle variabili di input al corpo centrale del neurone. Ad esse è associato un valore numerico chiamato **peso**.
3. **Corpo del neurone**. Ad esso è associato un valore numerico chiamato **bias** e ha una doppia funzionalità: calcolare il suo input sulla base dei pesi delle connessioni che gli giungono, i valori delle variabili di input che gli vengono recapitati tramite esse e il bias, e il suo output, tramite l'applicazione di una così detta **funzione d'attivazione** (associata a tale neurone) applicata al valore d'input.
4. 1 **linea di output** sulla quale viaggerà il valore d'output calcolato.

Come si è appena detto, il corpo del neurone ha come obiettivo quello di calcolare il suo input e il suo output.
L'input viene calcolato con la seguente formula:

$$a = \sum_{1=1}^{d} \omega_i \cdot x_i + b$$

cioè il prodotto scalare tra i pesi delle connessioni che giungono in quel neurone e gli input del neurone stesso. A questo prodotto scalare si aggiunge il bias associato al neurone. Tale calcolo, così come descritto, usufruisce del bias in forma esplicita. Esso può essere anche implicito, eseguendo solo l'operazione di prodotto scalare, includendo il bias all'interno della prima posizione del vettore dei pesi e aggiungendo un '1' nella prima posizione del vettore x, cioè di input.
Dato l'input del neurone si può calcolare il suo output attraverso l'applicazione della funzione d'attivazione (la si chiama *f*) sull'input calcolato:

$$z = f(a)$$

Si può visualizzare la struttura proposta di un generico neurone attraverso la seguente figura:

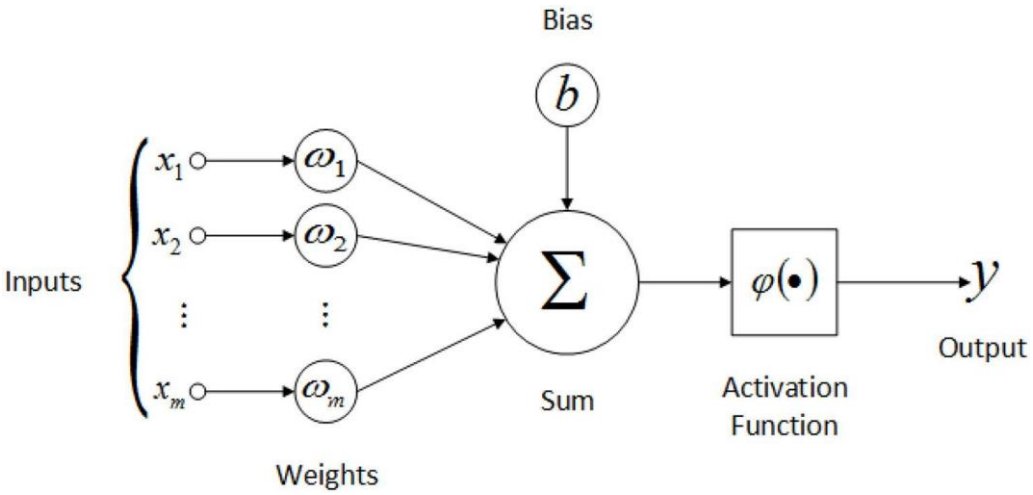


Figura 2 La struttura di un neurone artificiale

È facile convincersi che una NN è tale perché è l'insieme di interconnessioni tra più neuroni. Di fondamentale importanza è, quindi, il modo in cui i neuroni interagiranno tra loro per la risoluzione del problema a monte. Avendo, quindi, più neuroni, si avrà il seguente scenario:

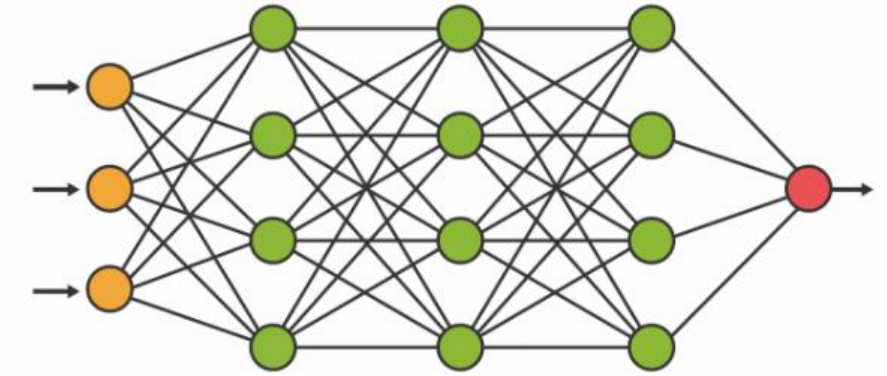


Figura 3 Generica Rete Neurale Artificiale

Tale figura fa comprendere il concetto che, l'output di un neurone contribuirà all'input di un neurone successivo; ogni connessione sarà identificata in base a quali neuroni connetterà e sarà pesata; l'output di quel neurone che non contribuirà all'input di nessun altro neurone sarà l'output dell'intera rete.
La figura appena proposta non è casuale in quanto raffigura un perfetto esempio di **Rete Neurale Artificiale Feed-Forward**. Questa tipologia di rete si differenzia dalle **Reti Ricorrenti** in quanto c'è una corrispondenza 'diretta' tra il vettore di input e l'output della rete. Più nello specifico, in questo tipo di rete l'ordine di computazione corrisponde esattamente all'ordine topologico della stessa: ciò vuol dire che tale rete

non possiederà dei cicli interni o delle ricorrenze tra i neuroni che la costituiscono (feed-forward descrive il concetto, quindi, di "propagazione in avanti").
Inoltre, in una Rete Feed-Forward si possono distinguere 3 macro-elementi:

1. **Strato di input**, nel quale è presente il vettore **x** di input;
2. **Strato hidden** o **nascosto** nel quale sono presenti **m** neuroni, su di essi viene applicata una funzione d'attivazione *g*;
3. **Strato d'output** nel quale sono presenti **c** neuroni**;** su di essi viene applicata un'ulteriore funzione d'attivazione *f.*

Una rete di tal genere che è caratterizzata da un solo livello hidden prende il nome di **Vanilla Network** o **Shallow Network** o **monostrato**. Una rete con più livelli hidden prende il nome di **Deep Network** o **multistrato.**
Si può visualizzare il tutto con la seguente figura:

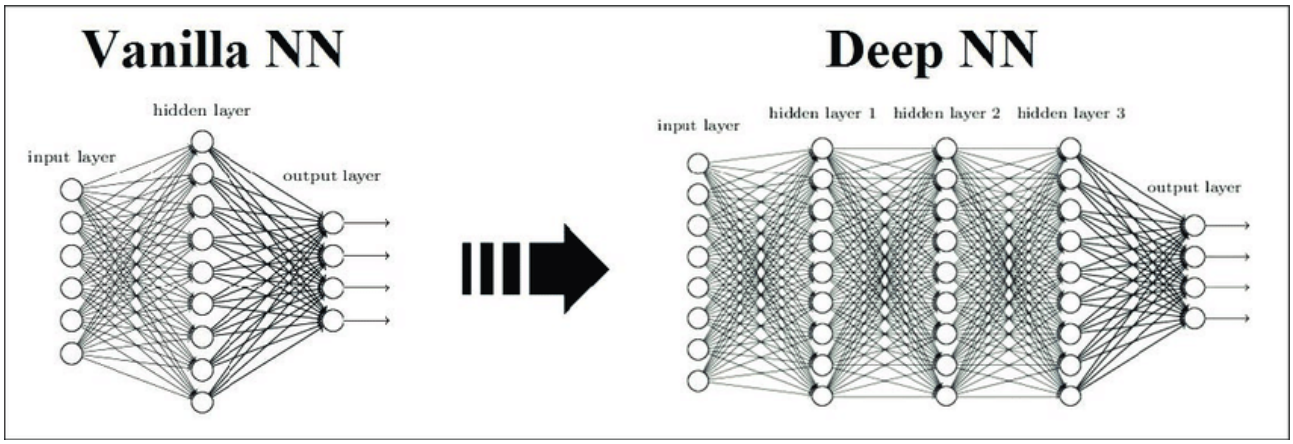


Figura 4 Vanilla Network e Deep Network

Data una rete di uno di tali generi si distinguono:

1. I **parametri** della rete, cioè quegli elementi della rete che dovranno essere aggiornati durante la **fase di learning** in maniera tale da avere una rete sempre più adeguata nel risolvere il problema di partenza; tipici esempi sono i pesi e i bias della rete.
2. Gli **iper-parametri** della rete, cioè quegli elementi il cui valore deve essere stabilito, secondo euristiche apposite, prima della fase di learning; tipici esempi sono: *m* e *g.*

Per la risoluzione del problema proposto si è fatto uso di una **Vanilla Network**.
Tale rete, così come qualsiasi altra tipologia di NN, è stata addestrata e sottoposta, quindi, all'operazione di **training.**

Fase di Training

Una volta che la rete è stata costruita, sono stati definiti i suoi iper-parametri e sono stati dati dei valori iniziali ai parametri, essa ha bisogno di essere addestrata per poter migliorare le sue capacità di risolvere il problema a monte per la quale è stata progettata. La fase che permette di raggiungere quest'obiettivo prende il nome di **fase di learning** o **fase di training.**
In tale fase si cerca di:

1. Minimizzare il valore della **loss function** o **funzione d'errore.** Essa è una metrica necessaria e fondamentale per poter calcolare quanto la rete sta svolgendo bene il suo compito, in questo caso di classificare. È una funzione che dipende strettamente dai parametri della rete (che vengono indicati attraverso l'insieme θ).
2. Generalizzare. Ciò significa che la rete dovrà essere in grado di, in questo caso, classificare non solo dati che sono stati già usufruiti durante l'addestramento, ma anche dati "sconosciuti".

La fase di learning si suddivide in varie sottofasi.

Fase di Forward-Propagation

La fase di **forward-propagation** consiste nell'effettuare l'operazione di "propagazione in avanti" della rete effettuando il calcolo delle variabili principali della rete: gli input dei vari nodi, i loro output, l'output dell'intera rete, derivate delle funzioni d'attivazione e d'errore necessari per la fase di **Back-Propagation**…
All'atto pratico, per poter calcolare l'input e l'output dei neuroni, si eseguono le formule descritte nella sezione dedicata alla struttura della rete.
Pertanto, avendo una generica rete neurale Feed-Forward con le seguenti caratteristiche:

- *d* - dimensione del vettore di input,
- *(x1, x2, ..., xd)* - vettore di input,
- 1 - numero di strati hidden,
- *m* - numero di nodi per ogni strato,
- *Wij* - peso associato alla connessione che va dal j-esimo neurone dello strato precedente al corrente al neurone i-esimo dello strato corrente,
- *f* la funzione di attivazione scelta;

iterando le formule descritte in precedenza, nella fase di Forward-Propagation si calcolano i valori di uscita per ogni nodo di ogni strato della rete. Per l'unico

strato hidden (denominato con (1) nella formula) viene eseguita la seguente operazione che calcola gli output di tutti i nodi che ne fanno parte:

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

Fatto ciò, si può procedere con l'esecuzione della stessa formula per calcolare l'output dei neuroni dello strato d'output, che definiranno l'output dell'intera rete. Vengono calcolate tutte le uscite dei nodi della rete neurale applicando la stessa formula "in avanti", seguendo l'ordine topologico stabilito dalle connessioni.
Si è spesso menzionato il concetto di **funzione d'attivazione.** Di seguito una descrizione delle principali che sono state definite nella libreria.

Funzioni d'attivazione

Sono state definite funzioni d'attivazione sia non lineari che lineari.

Sigmoide: una funzione non-lineare che possiede, dal punto di vista grafico, un andamento "a S". Possiede un dominio a supporto infinito, cioè si estende da -∞ a +∞ e un codominio che si definisce in un intervallo chiuso e limitato [0,1]. Di seguito la sua formula, la sua derivata e la sua rappresentazione grafica:

$$s(x) = \frac{1}{1 + e^{-x}} \qquad s'(x) = s(x) * (1 - s(x))$$

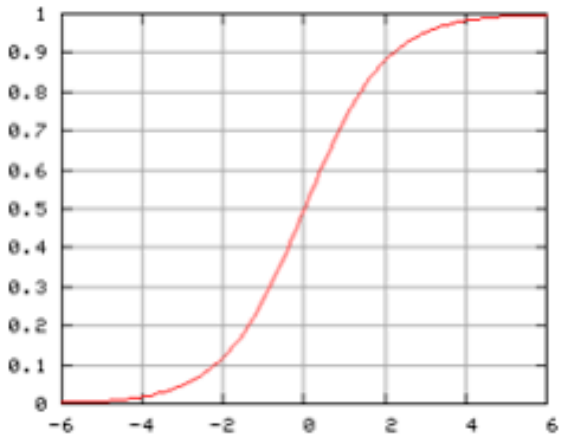


Figura 5 Funzione sigmoide

Identità: funzione lineare, sfruttata molto spesso sullo strato d'output di una NN. È a supporto infinito e anche il codominio si estende da -∞ a +∞. La sua derivata è estremamente semplice da calcolare, per tal motivo è una delle più diffuse nel Machine Learning.

$$f(x) = x \qquad\qquad f'(x) = 1$$

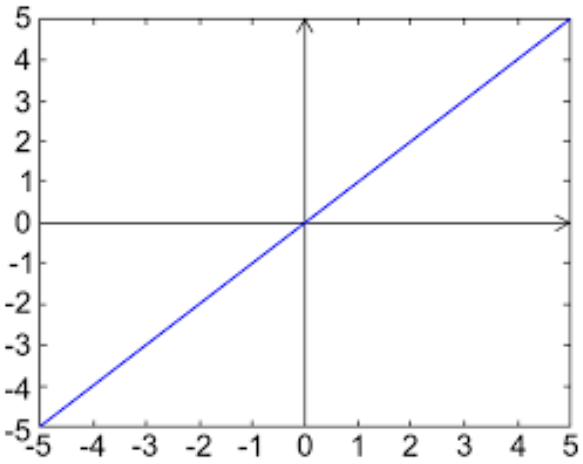


Figura 6 Funzione Identità

Tangente Iperbolica: funzione non lineare, simile alla sigmoide ma con un codominio differente, definito in un intervallo chiuso e limitato [-1,1].

$$\tanh(x) = \frac{1-e^{-2x}}{1+e^{2x}} \qquad\qquad tanh'(x) = 1 - \tanh(x) * \tanh(x)$$

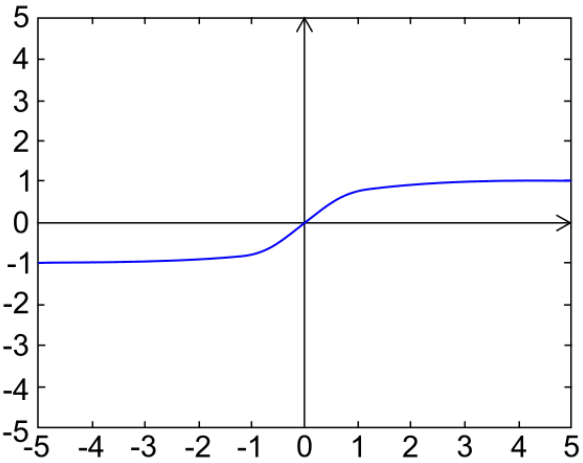


Figura 7 Funzione Tangente Iperbolica

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

$$relu(x) = max(0,x)$$

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

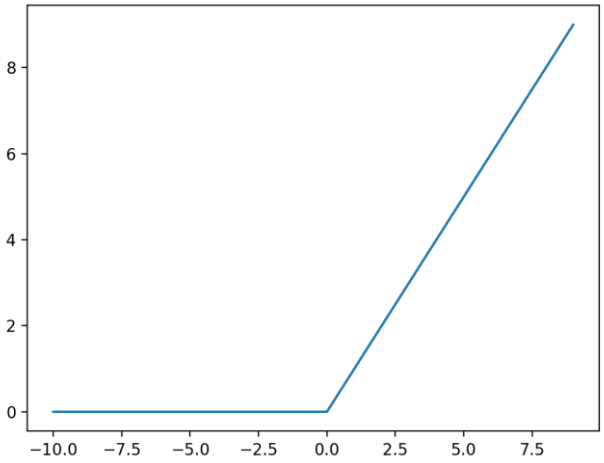


Figura 8 RELU

Essa possiede una derivata, anche in questo caso, semplice da calcolare, rappresentando una funzione d'attivazione vantaggiosa da sfruttare. Un altro grande vantaggio che possiede è che mitiga il problema del **Vanishing Gradient,** aumentando così la probabilità di avere delta valori e derivate diverse e lontane dallo zero, impedendo l'accumulare valori nulli nel considerare pesi di connessioni sempre più lontani dallo strato d'output e impedendo, di conseguenza, aggiornamenti non efficaci dei parametri della rete stessa.

Fase di Back-Propagation

A seguito della fase di forward-propagation c'è la fase di back-propagation che consiste nel calcolare nel miglior modo possibile la derivata della funzione d'errore per poterla minimizzare rispetto ai parametri della rete. Questo perché si cerca di minimizzare in più passi l'errore commesso nel classificare alcuni elementi del dataset; si cerca, quindi, di migliorare la rete affinché quest'ultima "sbagli" il meno possibile nel classificare.

Viene quindi, calcolata la funzione d'errore *E* dipendente dai parametri della rete θ, ove θ = {*W,b}.* Tale funzione la si cerca di minimizzare rispetto a tali parametri, con lo

scopo di ottenere parametri "ottimali" θ* che aumentano la capacità della rete di classificare. Quindi:

$$\theta^* = argmin_\theta\, E\,(\theta)$$

L'algoritmo di Back-Propagation è una procedura che ha lo scopo di calcolare la derivata della loss-function in maniera iterativa "all'indietro". Agisce, quindi, in senso opposto rispetto alla fase di forward-propagation. Infatti, in questo caso, si parte dall'ultimo strato della rete (strato d'output), fino ad arrivare al primo strato. Si può visualizzare il tutto attraverso la figura seguente.

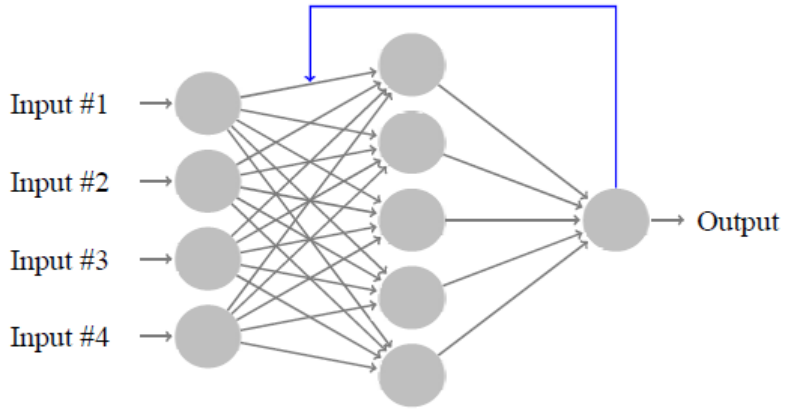


Figura 9 Back-Propagation

Per poter applicare tale algoritmo, però, bisogna rispettare delle ipotesi iniziali:

- la loss-function deve essere derivabile rispetto all'output della rete y_k;
- le funzioni d'attivazione devono essere funzioni derivabili;

La funzione di errore E è data da tutti gli errori $E^{(n)}$, cioè gli errori commessi sulle singole coppie del dataset iniziale (x^n, y^n); pertanto $E = \sum_{n=1}^{N} E^{(n)}$. Di conseguenza, la derivata della somma è uguale alla somma delle derivate, cioè:

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

Siccome ci si trova in una Rete Feed-Forward, c'è sempre una propagazione in avanti tra i neuroni che la compongono. Pertanto, considerando due nodi successivi *J* e *I* connessi attraverso una connessione con peso W_{ij}, tale peso sarà presente solo 1 volta all'interno della rete considerata. Pertanto, W_{ji} contribuisce al calcolo del solo

input di *I* (a_i) e di nessun altro neurone. Per tal motivo, grazie alla *Regola delle Funzioni Composte* si può dire che:

$$\frac{\partial E^{(n)}}{\partial w_{ij}} = \frac{\partial E^{(n)}}{\partial a_i} \frac{\partial a_i}{\partial w_{ij}}$$

Siccome a_i, così come l'input di un qualsiasi altro neurone, è un prodotto scalare tra pesi e output dei nodi del livello precedente connessi con *i*, quando si effettua la $\frac{\partial a_i}{\partial w_{ij}}$, essa darà come risultato quell'unico output z_j che è moltiplicato per il peso di interesse w_{ij}.

Quindi:

$$\frac{\partial E^{(n)}}{\partial w_{ij}} = \frac{\partial E^{(n)}}{\partial a_i} \; z_j$$

Per definizione si avrà che:

$$\delta_i = \frac{\partial E^{(n)}}{\partial a_i}$$

ed essi prenderanno il nome di **Delta-valori.** Tramite essi sarà possibile calcolare, attraverso la così detta **Legge locale,** la $\frac{\partial E^{(n)}}{\partial w_{ij}}$ facendo:

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

In base a ciò, l'algoritmo di Back-Propagtion cerca di calcolare i delta-valori, perché sulla base di essi, com'è stato ben dimostrato, si avrà tutto il necessario per poter calcolare le derivate, sulle singole coppie di dati, della loss function rispetto ai

parametri della rete. All'atto pratico, si effettua ciò "all'indietro", cioè a partire dall'ultimo strato della rete si ritorna allo strato più lontano da quest'ultimo. Pertanto, a seguito della fase di forward-propagation, avendo ottenuto il generico output di uno dei nodi dello strato di output della rete, y_k, si calcolano i delta-valori dei neuroni di tale strato applicando la seguente formula:

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

Tutto ciò supponendo che *f* sia la funzione d'attivazione applicata sullo strato d'output.
Una volta ottenuti i δ_k sarà possibile, "back-propagando", calcolare i delta-valori dei neuroni appartenenti agli strati hidden attraverso la seguente formula:

$$\delta_h = g'(a_h)\sum_k \omega_{kh}\delta_k$$

Una volta ottenuti tutti i delta-valori sarà possibile calcolare tutte le derivate attraverso la **Legge Locale.**
Una volta, poi, ottenute le derivate, sarà possibile calcolare l'errore totale attraverso una semplice sommatoria.
È facile convincersi che, a seconda della tipologia di funzione d'errore usufruita, si avranno risultati differenti. Di seguito una breve descrizione di quelle che è possibile richiamare attraverso la libreria proposta.

Funzioni d'errore

Con tale libreria è possibile richiamare l'uso di 2 principali loss-function:

Sum of squares: solitamente utilizzata per problemi di regressione ed è definita attraverso la seguente formula:

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

Cross Entropy: è solitamente utilizzata per problemi di classificazione ed è definita dalla seguente formula:

$$-\sum_j t_j log(s(y_j))$$

Softmax: non è una funzione d'errore ma viene sfruttata dalla funzione cross-entropy. Essa è una funzione che trasforma le componenti di un vettore in probabilità, cioè valori compresi in [0,1], e facendo sì che la somma di tali probabilità sia 1. Questo è fondamentale in quanto da un classificatore ci si aspetta sempre in uscita delle probabilità, cioè la $P\,(C_k/x)$.
Tale funzione rappresenta un'operazione di post-processing: dopo aver calcolato gli output della rete y_k, si calcolano delle variabili z_k che saranno pari a:

$$\frac{e^{y_i}}{\sum_{k=1}^{N} e^{y_k}}$$

L'uso di tale passo permette di rispettare le 2 proprietà delle probabilità menzionate.

Fase di aggiornamento dei parametri

Una volta ottenute le derivate della funzione di errore rispetto ai pesi $\frac{dE^{(n)}}{dw_{ij}}$, è possibile minimizzare tale funzione E (θ) facendo variare i suoi ingressi, dati dai pesi delle connessioni. Esistono vari tipi di criteri di aggiornamento sfruttati in Machine Learning. Il criterio utilizzato in questo progetto è chiamato **Standard Gradient Descent**.
Il seguente criterio 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 della funzione d'errore (**funzione decrescente**), altrimenti diminuirlo se si sta alla destra del minimo (**funzione crescente**); il comportamento desiderato è riassunto di seguito:

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

A questo punto, sia η un iper-parametro chiamato **learning rate**, tale che $0 < \eta < 1$; si definisce l'aggiornamento di un generico peso w_{ij} attraverso lo Standard Gradient Descent, mediante la seguente assegnazione:

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

In generale si ha che:

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

ove ∇E è il gradiente della funzione d'errore, ovvero il vettore avente come componenti le derivate parziali della funzione di errore E rispetto ai pesi w_{ij}.
La fase di learning, e quindi anche la fase di aggiornamento dei parametri, di un classificatore, può essere effettuata attraverso 3 tipologie di modalità:

1. **modalità batch**: i pesi sono aggiornati dopo che tutte le coppie del dataset sono date come input alla rete e quindi dopo che sono state calcolate tutte le derivate della funzione E rispetto ai pesi;
2. **modalità online**: nella stessa epoca si effettua il calcolo delle derivate e l'aggiornamento dei pesi. È sicuramente più lenta, come modalità, rispetto alla modalità batch e, quindi, più pesante dal punto di vista computazionale, ma riesce a raggiungere un buon risultato con meno epoche.
3. **Modalità minibacth**: si raggruppano dati in batch dell'ordine delle centinaia di dati. Così si processeranno tali dati e si aggiorneranno i pesi in base a tali processi. Tutto ciò fino a quando non saranno terminati tutti i dati. È una modalità che sfrutta i vantaggi delle 2 modalità precedenti: si raggiungerà più velocemente il minimo errore e dal punto di vista computazionale è meno pesante della modalità online.

Fase di Testing

Una volta addestrata la rete e ottenuta la migliore rete possibile, cioè quella che minimizza la loss function e massimizza la capacità di generalizzazione, si effettua la fase di 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, è corretta se la classe assegnata dalla rete all'immagine corrisponde

a quella a cui effettivamente l'immagine appartiene. Per verificare ciò si calcola **l'accuracy** sul test set come il rapporto tra le immagini classificate correttamente e il numero di immagini totali. Di seguito la formula matematica:

$$Accuracy = \frac{numero\ di\ casi\ corretti}{numero\ di\ casi\ totali}$$

Dataset

La traccia proposta richiede la classificazione di un insieme di dati specifico: il dataset **MNIST**. Tale è una vasta base di dati di cifre scritte a mano che è comunemente impiegata come insieme di addestramento in vari sistemi per l'elaborazione delle immagini. La base di dati è anche impiegata come insieme di addestramento e di test nel campo del Machine Learning.
La base di dati MNIST contiene 60 000 immagini di addestramento e 10 000 immagini di test.
Queste quantità sono state mantenute anche per la realizzazione di questo progetto. Più nello specifico si sono mantenute le seguenti percentuali:

- 80% di 60000 per la definizione del Training-set;
- 20% di 60000 per la definizione del Validation-set;
- 10% dell'intero dataset per la definizione del Test-set.

Le immagini si presentano in bianco e nero avendo 28x28 pixel per un totale di 784. Essendo state scritte a mano delle cifre il numero di possibili classi sulla quale è possibile lavorare sarà sempre di un massimo di 10.
Esempi grafici di tali immagini sono riportati di seguito:

Figura 10 Esempi di immagini del dataset MNIST

Subroutine

A questo punto della relazione sono riportate le descrizioni delle varie routine presenti nella libreria che compone il progetto proposto.

Nel file *dataset.py* sono presenti le seguenti routine:

- **loadDataset:** funzione che ha l'obiettivo di caricare il dataset MNIST da un apposito pathname e, conseguentemente, costruire delle matrici che definiranno una prima versione del training-set e il test-set.
  Input:
  - *data_path:* pathname del dataset MNIST

  Output:
  - *xtrain:* insieme delle immagini che compongono il training-set
  - *ytrain:* label delle immagini che compongono il training-set
  - *xtest:* insieme delle immagini che compongono il test-set
  - *ytest:* label delle immagini che compongono il test-set
- **showImage:** funzione che effettua l'operazione di visualizzazione di un'immagine specifica del dataset.
  Input:
  - *immagine:* l'immagine che si intende visualizzare

Nel file *activation_function.py* sono presenti le seguenti routine:

- **sigmoid**: routine che definisce la funzione sigmoide come funzione d'attivazione.
  Input:
  - *x*: variabile di input della funzione d'attivazione specifica
  - *flag*: variabile che avrà valore '0' o '1' a seconda se si intende ritornare solo il valore della funzione avendo in input *x* oppure restituire anche la derivata di tale funzione

  Output:
  - *y*: valore della funzione avendo in input *x*
  - *y*(1-y)*: derivata della funzione sigmoide
- **identity**: routine che definisce la funzione identità come funzione d'attivazione.
  Input:

  - *x*: variabile di input della funzione d'attivazione specifica
  - *flag*: variabile che avrà valore '0' o '1' a seconda se si intende ritornare solo il valore della funzione avendo in input *x* oppure restituire anche la derivata di tale funzione

  Output:
  - *y*: valore della funzione avendo in input *x*
  - *1:* derivata della funzione identità

- **Tanh**: routine che definisce la funzione tangente iperbolica come funzione d'attivazione.
  Input:
  - *x*: variabile di input della funzione d'attivazione specifica
  - *flag*: variabile che avrà valore '0' o '1' a seconda se si intende ritornare solo il valore della funzione avendo in input *x* oppure restituire anche la derivata di tale funzione

  Output:
  - *y*: valore della funzione avendo in input *x*
  - *1-y*y:* derivata della funzione tangente iperbolica
- **Relu:** routine che definisce la funzione RELU come funzione d'attivazione.
  Input:
  - *x*: variabile di input della funzione d'attivazione specifica
  - *flag*: variabile che avrà valore '0' o '1' a seconda se si intende ritornare solo il valore della funzione avendo in input *x* oppure restituire anche la derivata di tale funzione

  Output:
  - *y*: valore della funzione avendo in input *x*
  - *y*1:* derivata della funzione RELU

Nel file *error_function.py* sono presenti le seguenti routine:

- **sumOfSquares**: routine che permette la definizione e l'uso della loss function Sum of Squares.
  Input:
  - *y*: label che è stata rilasciata dal classificatore considerando l'n-esimo elemento del dataset e θ come parametri della rete;
  - *t*: target o classe vera dell'n-esimo dato che si sta considerando
  - *flag*: variabile che avrà valore '0' o '1' a seconda se si intende ritornare solo il valore della funzione avendo in input *y* e *t* oppure restituire anche la derivata di tale funzione.

Output:

- *(1/2)*np.sum(np.power((y-t),2))*: valore della Sum of Squares avendo *y* e *t* come input
- *y-t*: derivata di tale funzione d'errore

- **crossEntropyWithSoftmax**: routine che permette la definizione e l'uso della loss function Cross Entropy usufruendo anche del passo di post processing del Softmax.
  Input:
  - *y*: label che è stata rilasciata dal classificatore considerando l'n-esimo elemento del dataset e θ come parametri della rete;
  - *t*: target o classe vera dell'n-esimo dato che si sta considerando
  - *flag*: variabile che avrà valore '0' o '1' a seconda se si intende ritornare solo il valore della funzione avendo in input *y* e *t* oppure restituire anche la derivata di tale funzione.

  Output:
  - *-(t*np.log(z)).sum()*: valore della Cross Entropy, sfruttando anche il passo di Softmax, avendo *y* e *t* come input
  - *z-t*: derivata di tale funzione d'errore
- **softMax**: routine che definisce il passo di Softmax.
  Input:
  - *y*: label ritornata dal classificatore

  Output:
  - *z*: variabile uscente dal Softmax derivante da *y*

Nel file *network_library.py* sono presenti le seguenti routine:

- **build_netowrk**: routine che ha l'obiettivo di costruire a mo' di dizionario una Neural Network.
  Input:
  - *input_size*: il numero di variabili di ingresso della rete
  - *num_hidden_neurons*: numero di nodi nell'unico strato hidden
  - *output_size*: il numero di neuroni nello strato d'output

  Output:
  - *network:* rete neurale costruita a mo' di dizionario sulla base dell'input di tale routine e avendo pesi e bias inizializzati secondo una distribuzione gaussiana avente media nulla e deviazione standard unitaria.

- **Get_network_structure**: routine che ha l'obiettivo di stampare a video le caratteristiche di una rete datale in input.
  Input:
  - *Network*: rete che dovrà essere sottoposta a 'scansione' per permette la visualizzazione nel terminale delle sue caratteristiche.
- **splitTrainingDataset**: routine che ha l'obiettivo di effettuare l'operazione di split della prima versione del Training-set. Tutto ciò per avere la definizione di Training-set e Validation-set.
  Input:
  - *Y*: set delle label del training-set da 'splittare'.
  - *Percentuale*: variabile che definisce la grandezza che dovrà assumere il Validation-set rispetto al Training-set.

  Output:
  - *indiceTraining*: variabile che stabilisce l'ultimo indice all'interno della quale ci sarà un elemento appartenente al Training-set effettivo.
  - *indiceValidation*: variabile che stabilisce l'ultimo indice all'interno della quale ci sarà un elemento appartenente al Validation-set effettivo.
- **Copy_network**: routine che ha come obiettivo effettuare un'operazione di copia delle caratteristiche di una rete in un'altra.
  Input:
  - *Network*: rete che dovrà essere sottoposta all'operazione di copia

  Output:
  - *Network_1*: rete risultante dalla copia.
- **Train**: routine che permette l'esecuzione della fase di Training di una NN con la definizione di tutte le sue sottofasi.
  Input:
  - *Network*: rete da addestrare;
  - Err_fun*:* funzione d'errore scelta
  - *Xtrain:* Training-set effettivo
  - *YTrain:* label del Training-set effettivo
  - *XValid:* Validation-set effettivo
  - *YValid:* label del Validation-Set effettivo
  - *maxEpoche:* numero massimo di epoche da eseguire per completare il training
  - *eta:* iper parametro della rete necessario per eseguire l'aggiornamento dei pesi mediante discesa del Gradiente
  - *flag:* variabile che identifica la modalità di learning scelta

- *numero_mini_batch:* variabile che indica il quantitativo di batch da creare nel caso in cui si sia scelta come modalità di learning la modalità minibatch. Negli altri casi assumerà valore pari a '0'.

Output:

- *errore_training*: array che definisce l'errore commesso, durante il training e sul training set, ad ogni epoca.
- *Errore_validation:* array che definisce l'errore commesso, durante il training e sul validation set, ad ogni epoca.
- *Accuratezza_training:* array che definisce l'accuratezza ottenuta, durante il training e sul training set, ad ogni epoca.
- *Accuratezza_validation*: array che definisce l'accuratezza ottenuta, durante il training e sul validation set, ad ogni epoca.

- **Split**: routine che effettua l'operazione di partizionamento del training set in base alla modalità di learning che è stata scelta dall'utente.

  Input:

  - *YTrain*: insieme delle label del Training-set che dovrà essere sottoposto all'operazione di split;
  - *K*: variabile che indica in quante porzioni/batch splittare il Training-set

  Output:

  - *IndiciYTrain:* array di indici che definiscono il Training-set effettivo da sottoporre all'operazione di learning da parte della rete.

- **Forward_propagation**: routine che definisce tutte le operazioni necessarie per la fase di propagazione in avanti della rete.

  Input:

  - *Network*: rete da sottoporre a tale operazione;
  - *X*: dati organizzati come matrice dxN ove d è il numero di features e N è il numero di samples. È una struttura utilizzata quando l'unico obiettivo della routine è quello di calcolare l'output dell'intera rete.
  - *X1*: dati organizzati come matrice dxN ove d è il numero di features e N è il numero di samples. È una struttura utilizzata quando l'obiettivo della routine è quello di ritornare gli output di tutti gli strati della rete e le derivate delle funzioni d'attivazione di ogni strato. Ciò sarà necessario per la fase di Back-Propagation.

  Output:

  - *Z:* assumerà il valore dell'output dell'intera rete o la struttura che possiederà tutti gli output dell'intera rete considerata.

  - *Derivate_correnti:* struttura che possiede tutte le derivate delle funzioni d'attivazione dei vari strati della rete.

- **Back_propagation:** routine che definisce la fase di back_propagation per la minimizzazione della loss function scelta.
  Input:
  - *Network:* rete da sottoporre a tale fase;
  - *T:* insieme delle label del Training-set organizzate a mo' di matrice cxN ove c è il numero dei neuroni d'output e N è il numero dei samples.
  - *Error_function:* funzione d'errore da minimizzare.
  - *Output:* struttura che possiederà tutti gli output dell'intera rete considerata.
  - *Derivate:* struttura che possiede tutte le derivate delle funzioni d'attivazione dei vari strati della rete.

  Output:
  - *Derivate:* lista delle derivate della funzione d'errore rispetto ai pesi della rete;
  - *Bias_derivate:* lista delle derivate della funzione d'errore rispetto ai bias della rete;
- **Accuratezza:** routine che implementa l'operazione necessaria per calcolare l'accuracy del classificatore realizzato.
  Input:
  - *Y:* label rilasciata dal calcolatore.
  - *T:* label vera

  Output:
  - Risultato del rapporto che definisce l'accuracy.
- **Accuratezza_rete**: routine che implementa l'operazione di accuracy di una rete su un particolare set di dati.
  Input:
  - *Network*: rete che deve essere sottoposta ad una fase di forward-propagation per ottenere l'output corrispondete ad un dato input.
  - *X*: dataset d'input da considerare.
  - *T*: label vere di quel dataset *X*.

  Output:
  - Il risultato dell'accuratezza della rete su quel particolare dataset specifico.
- **Get_pesi_rete**: routine 'getter' di tutti i pesi delle connessioni che compongono una certa rete datale in input.

Input:

- *Network:* rete su cui applicare tale metodo.

Output:

- *W*: lista dei pesi di tale rete.

- **Get_bias_rete**: routine 'getter' di tutti i bias dei neuroni che compongono una certa rete datale in input.

  Input:

  - *Network:* rete su cui applicare tale metodo.

  Output:

  - *Bias*: lista dei bias di tale rete.

- **Get_funzione_attivazione_rete**: routine 'getter' di tutte le funzioni d'attivazione che compongono una certa rete datale in input.

  Input:

  - *Network:* rete su cui applicare tale metodo.

  Output:

  - *Funzione_Attivazione*: lista delle funzioni d'attivazione di tale rete.

Risultati e considerazioni finali

A questo punto della relazione si vuole discutere i risultati ottenuti da una serie di esperimenti.
Seguendo la traccia, sono stati considerati 4 dimensioni differenti dell'unico strato hidden della rete (50,100,200,500), al variare delle 3 modalità d'apprendimento e rimanendo invariate le funzioni d'attivazione (sono state scelte la sigmoide per lo strato hidden e la funzione identità per lo strato di output).
Di seguito verranno mostrati grafici che illustrano l'andamento dell'apprendimento della rete avendo definito 500 epoche massimo e senza nessun criterio di arresto anticipato; in particolare verrà mostrato l'errore e accuracy sul training-set, validation-set e sul Test-set e tempo totale di esecuzione della fase di training.
Come già detto anche nella sezione riguardante la descrizione del dataset utilizzato, sono stati settati i seguenti iper-parametri:

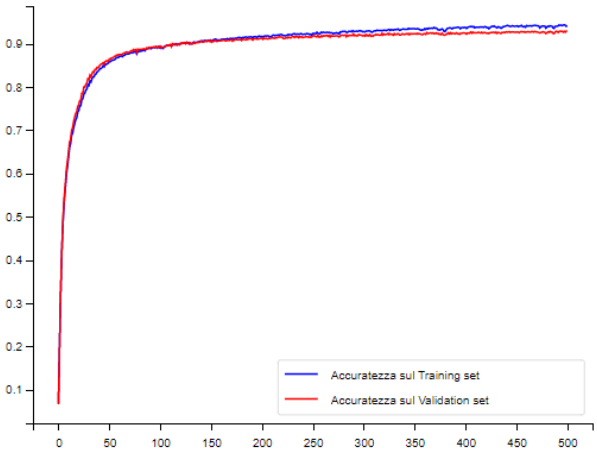
- Lunghezza training-set = 48000
- Lunghezza validation-set = 12000
- Lunghezza test-set = 10000
- η per la discesa del gradiente: 0,00001
- numero di batch = 4000 nel caso si sfrutti la modalità di learning minibatch

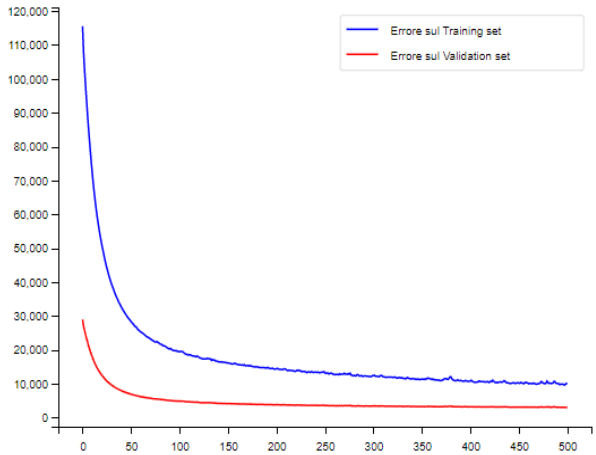
Tra i parametri troviamo:

- pesi iniziali estratti da una distribuzione gaussiana avente media nulla e deviazione standard unitaria
- bias iniziali estratti da una distribuzione gaussiana avente media nulla e deviazione standard unitaria

Di seguito i grafici proposti:

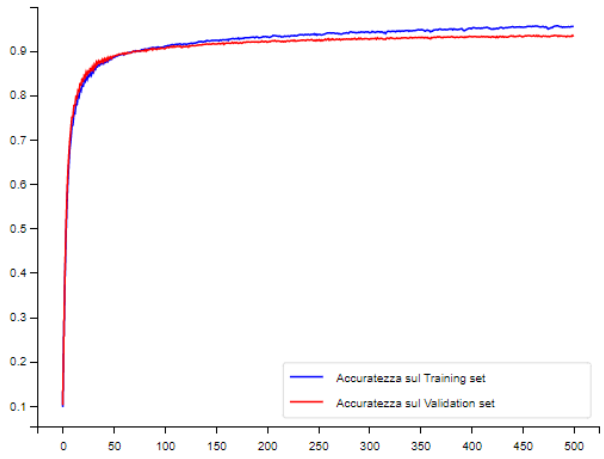
1) 50 hidden neurons, modalità batch

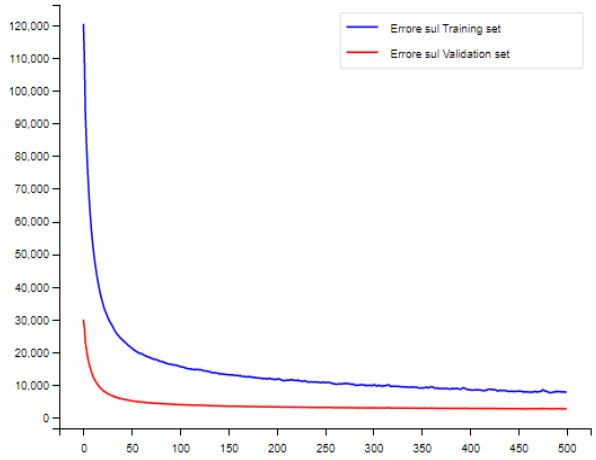




Tempo di esecuzione della fase di training = 188.03817224502563 secondi
Accuratezza sul test set: 0.9279
Accuratezza sul training set: 0.9390333333333334

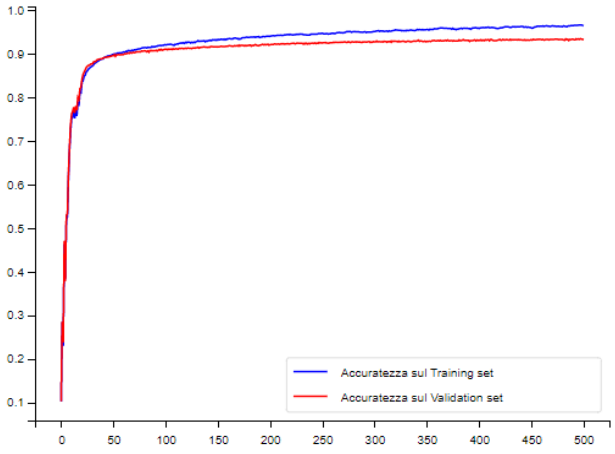
2) 100 hidden neurons, modalità batch

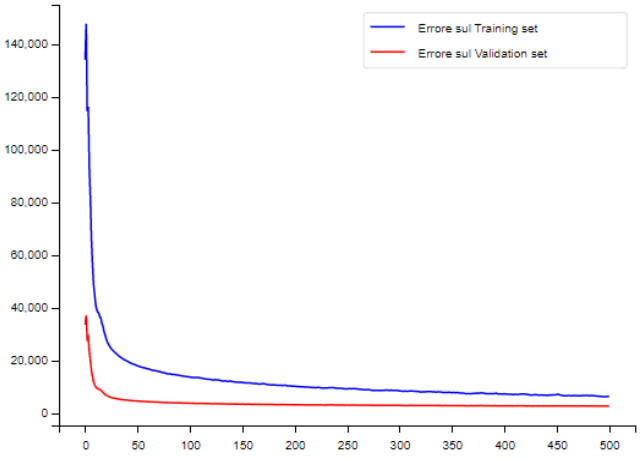




Tempo di esecuzione della fase di training = 269.6377353668213 secondi
Accuratezza sul test set: 0.9303
Accuratezza sul training set: 0.9502333333333334

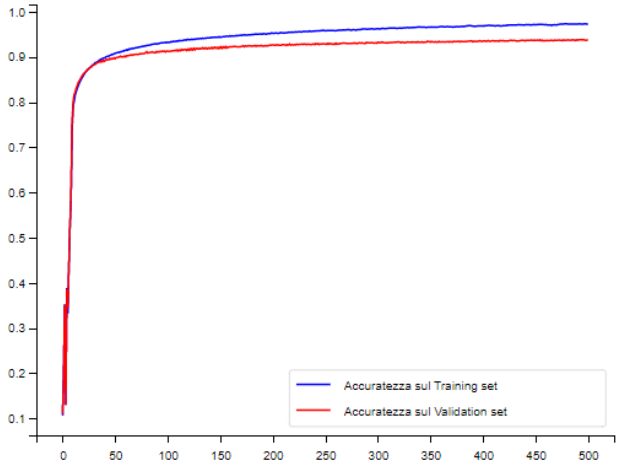
3) 200 hidden neurons, modalità batch

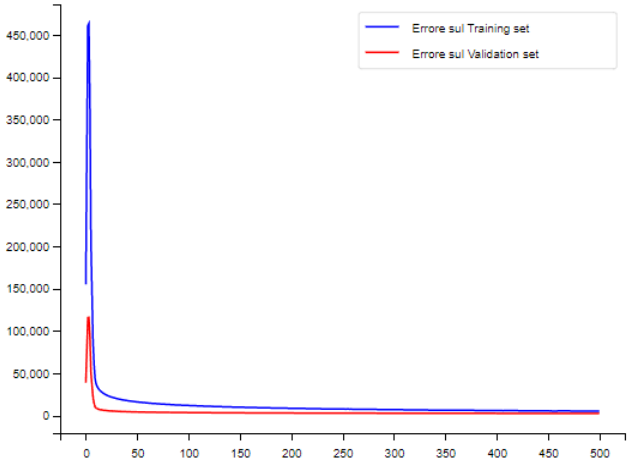




Tempo totale di esecuzione della fase di training: 432.4411289691925 secondi
Accuratezza sul test set: 0.932
Accuratezza sul training set: 0.95805

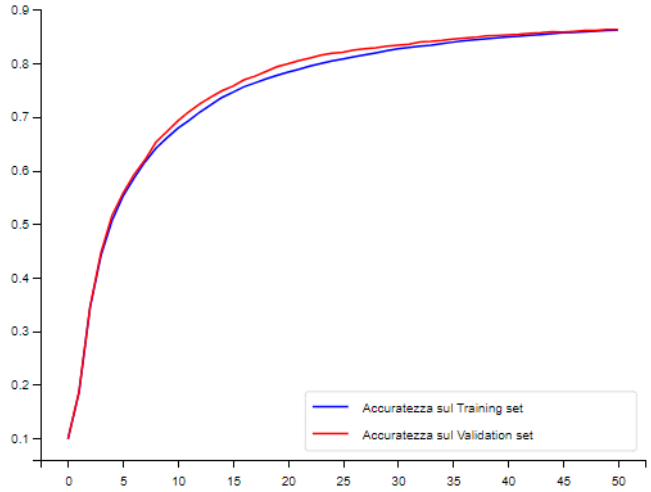
4) 500 hidden neurons, modalità batch

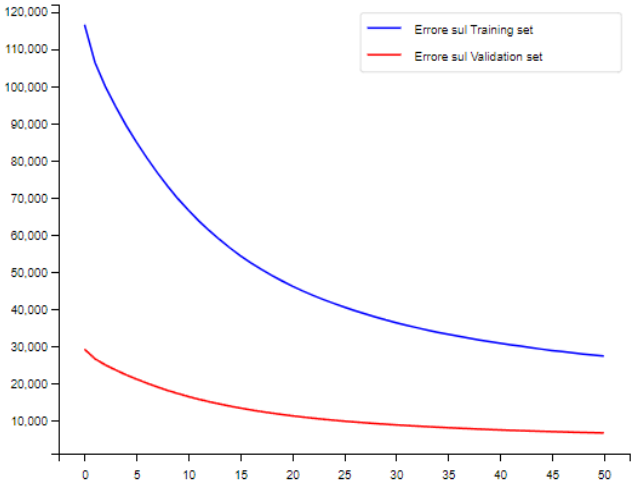




Tempo di esecuzione della fase di training = 996.23584563289645 secondi
Accuratezza sul test set = 0.9353
Accuratezza sul training set = 0.96010

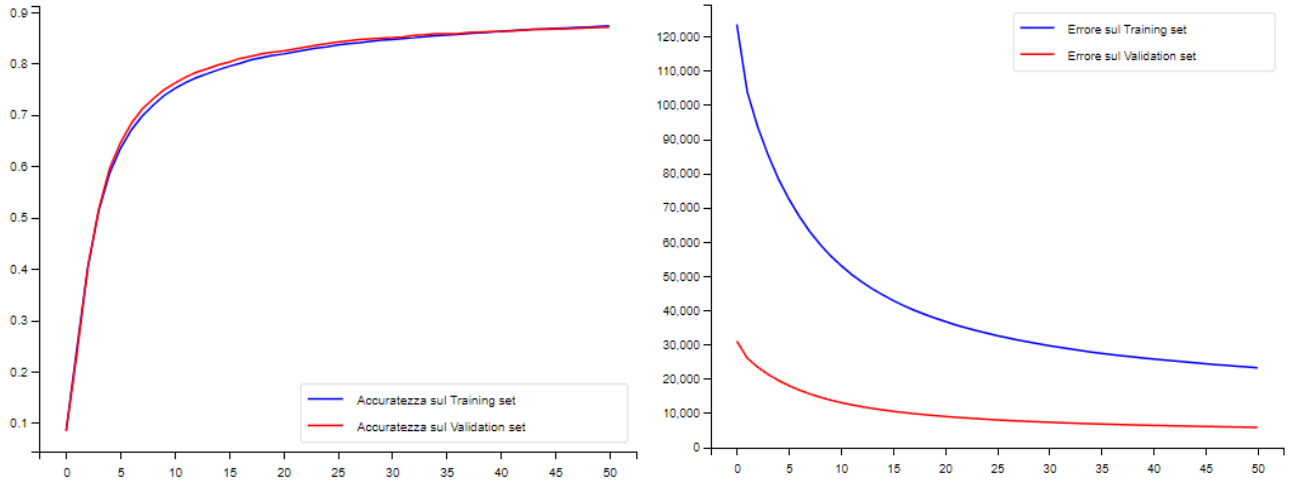
5) 50 hidden neurons, modalità online





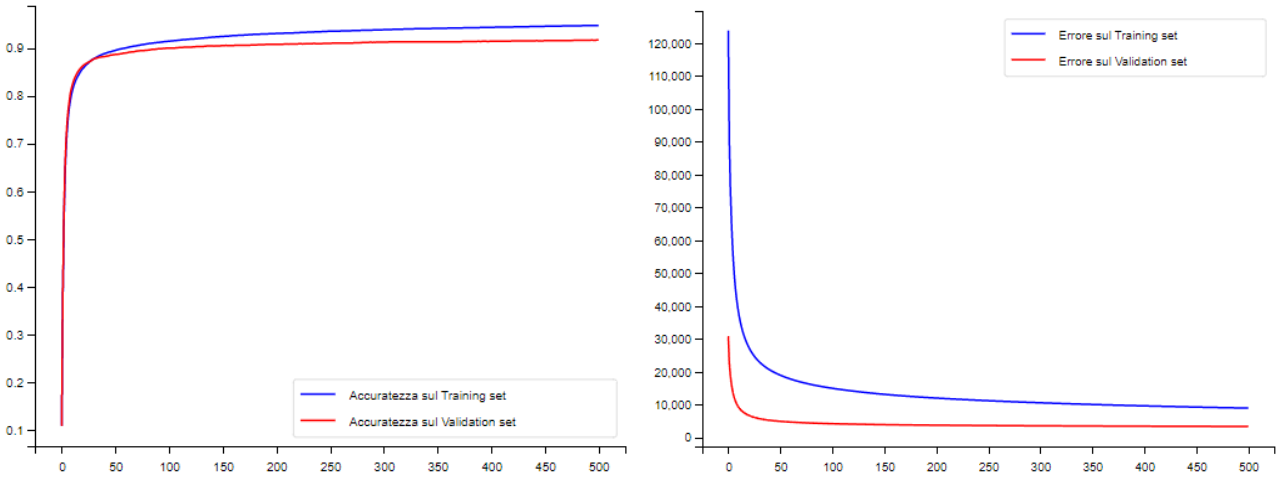
Tempo totale di esecuzione della fase di training: 651.1697552204132 secondi
Accuratezza sul test set: 0.8607
Accuratezza sul training set: 0.8620666666666666

6) 100 hidden neurons, modalità online



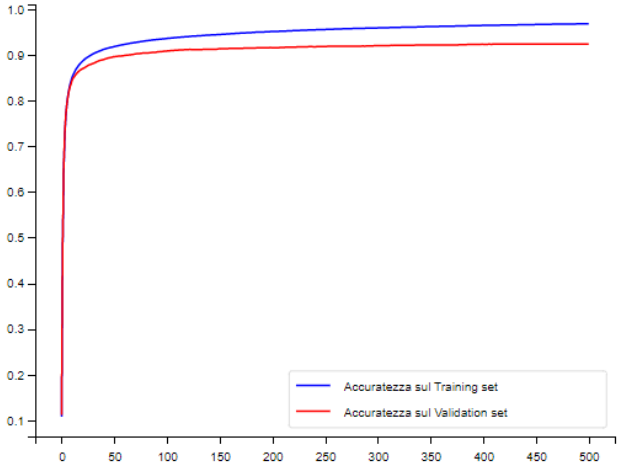
Tempo totale di esecuzione della fase di training: 1052.3237128257751 secondi
Accuratezza sul test set: 0.8739
Accuratezza sul training set: 0.8727333333333334

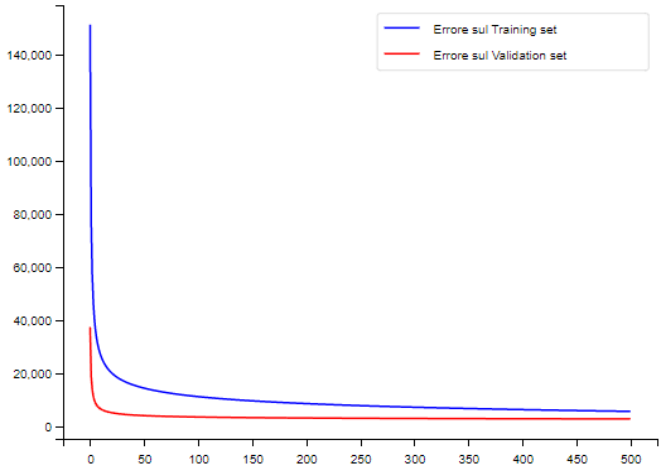
7) 200 hidden neurons, modalità online



Tempo totale di esecuzione della fase di training: 37656.439376831055 secondi
Accuratezza sul test set: 0.9177
Accuratezza sul training set: 0.9410833333333334

8) 500 hidden neurons, modalità online



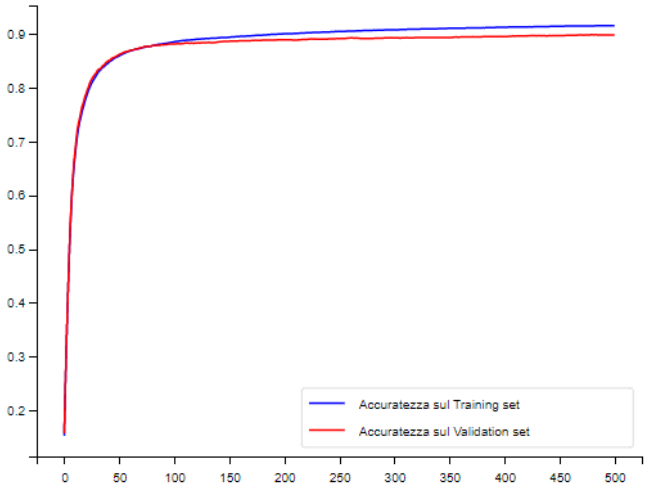


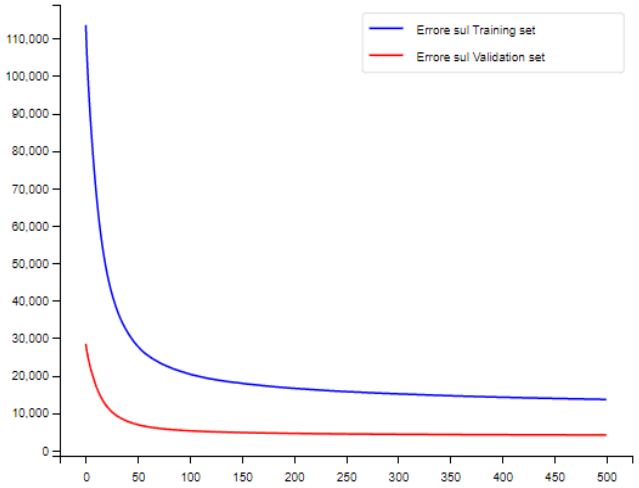
Tempo totale di esecuzione della fase di training: 84318.9764688015 secondi

Accuratezza sul test set: 0.9222

Accuratezza sul training set: 0.95925

9) 50 hidden nueruons, modalità minibatch, 4000 batch



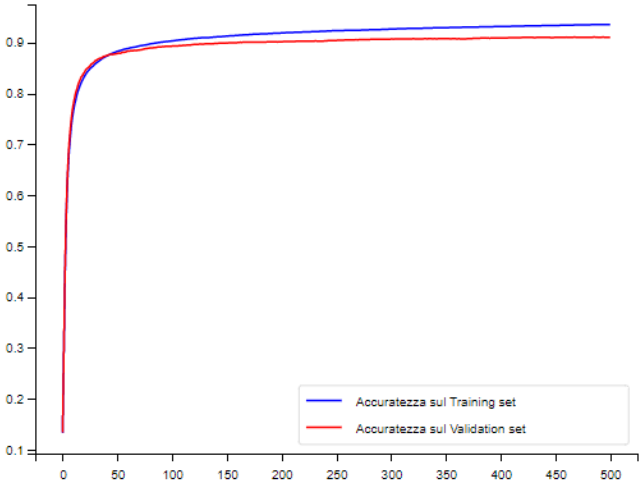


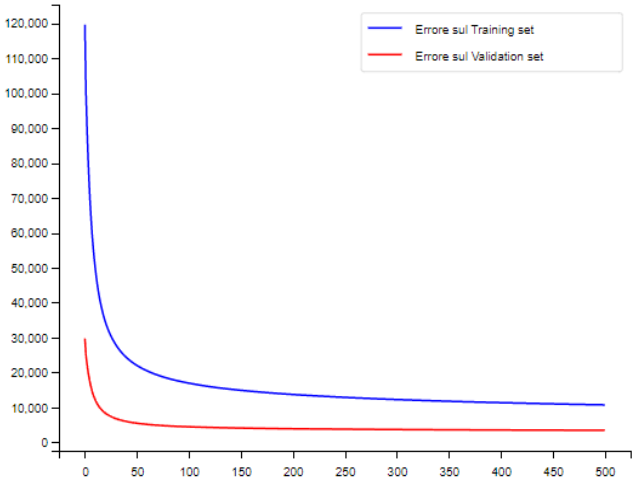
Tempo totale di esecuzione della fase di training: 645.5743408203125 secondi

Accuratezza sul test set: 0.8955

Accuratezza sul training set: 0.9121

10) 100 hidden neurons, modalità minibatch, 4000 batch



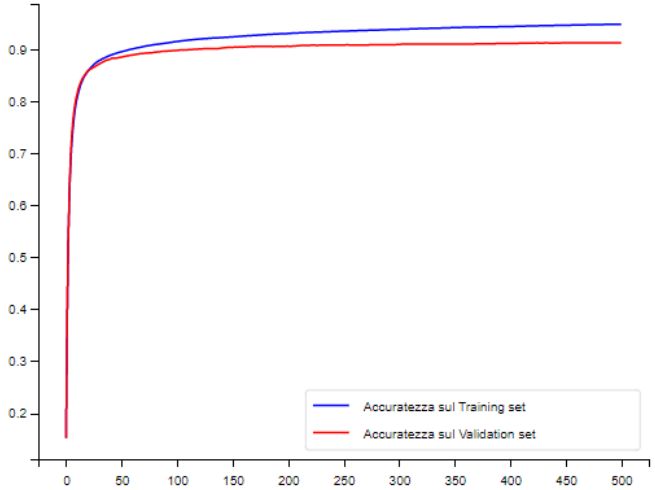


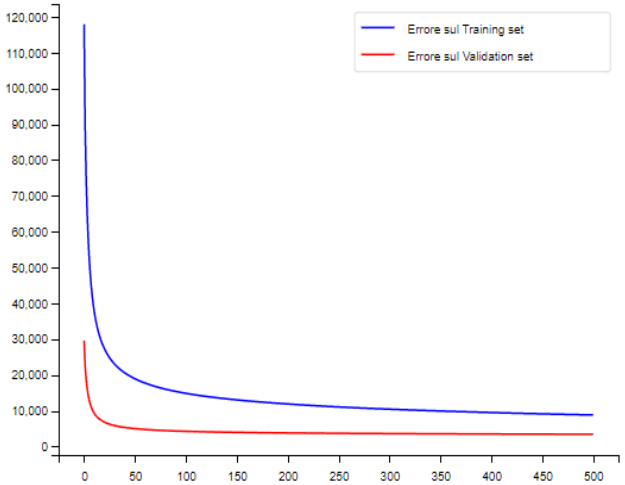
Tempo totale di esecuzione della fase di training: 952.531546831131 secondi

Accuratezza sul test set: 0.9119

Accuratezza sul training set: 0.93045

11) 200 hidden neurons, modalità minibatch, 4000 batch



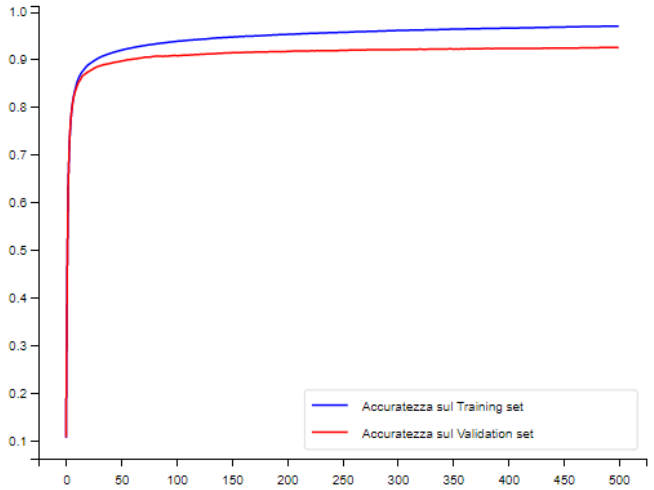


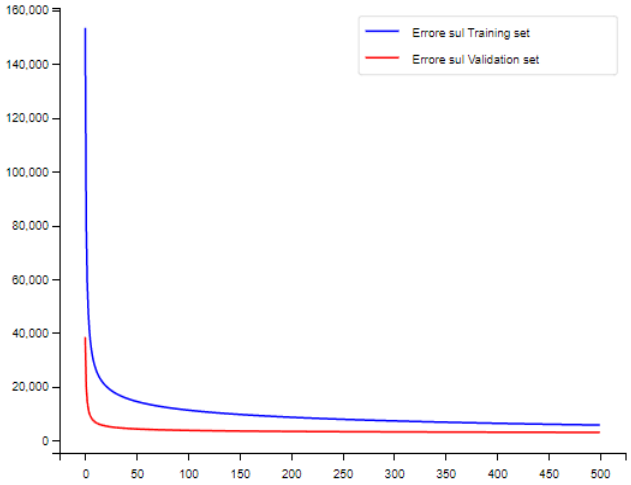
Tempo totale di esecuzione della fase di training: 3133.04838681221 secondi

Accuratezza sul test set: 0.9142

Accuratezza sul training set: 0.9406

12) 500 hidden neurons, modalità minibatch, 4000 batch





Tempo totale di esecuzione della fase di training: 6878.15851187706 secondi
Accuratezza sul test set: 0.9241
Accuratezza sul training set: 0.9607166666666667

Dagli esperimenti effettuati si può notare che, anche variando la modalità di apprendimento della rete, l'accuracy si aggira attorno al 90-95% e l'errore, sia sul training set che validation set, risulta essere grossomodo basso. Un altro aspetto da notare è che tali risultati vengono raggiunti in un numero di epoche anch'esso grossomodo basso (nonostante si sia definito un numero massimo di epoche alto). Da notare anche i tempi necessari per l'esecuzione dei vari esperimenti: si noti come nella modalità online, al crescere del quantitativo di nodi hidden, aumenti a dismisura il tempo di esecuzione della fase di learning (l'ultima esecuzione è durata 2 giorni), mentre le altre esecuzioni, sfruttando le altre modalità di learning, definiscano tempi di esecuzioni ragionevoli raggiungendo, tra l'altro, grossomodo gli stessi risultati in termini di accuracy.
Per migliorare ancora di più tali risultati si potrebbe pensare ai seguenti sviluppi futuri:

- Definire un criterio d'arresto intelligente che permetta di arrestare la fase di learning ad un'epoca in cui si è ottenuto una rete accettabile con tempi computazioni molto più bassi rispetto a quelli ottenuti con la versione corrente di tale progetto. Un esempio è il criterio della Pazienza;
- Implementare metodi di aggiornamento dei pesi differenti dallo Standard Gradient Descent come la Resilient Back-Propagation o sfruttando il Momento; tutto ciò per avere la possibilità di gestire anche casi particolari di funzioni d'errore (come il caso di funzioni piatte) e/o cercando di mitigare il

problema dell'oscillazione eventuale nella ricerca del punto di minimo della loss function.

Codice Python

```
import time
import dataset as ds

#Caricamento del dataset MNIST
Xtrain, Ytrain, Xtest, Ytest = ds.loadDataset()
print("Le caratteristiche del dataset sono:\r")
print("Dimensione Xtrain:", Xtrain.shape)
print("Dimensione Ytrain:", Ytrain.shape)
print("Dimensione Xtest shape:", Xtest.shape)
print("Dimensione Ytest shape:", Ytest.shape, "\r\r")


print("##########################################################Costruzione della
rete################################################\r\r\r")
#Definizione del numero di neuroni hidden
NUM_HIDDEN_NEURONS=500
#Costruzione della rete. Si passa alla funzione: il numero di input, neuroni interni e il numero di classi
network = net_lib.build_network(Xtrain.shape[0],NUM_HIDDEN_NEURONS,Ytrain.shape[0])
#Stampa delle caratteristiche della rete
net_lib.get_network_structure(network)

#Creare una copia della struttura della rete per fare gli esperimenti
network_1=net_lib.copy_network(network)
net_lib.get_network_structure(network_1)

#Split del training set in 2 parti
indiceTraining, indiceValidation = net_lib.splitTrainingDataset(Ytrain.shape[1],20)
print(indiceTraining, indiceValidation)
XT = Xtrain[:,:indiceTraining]
YT = Ytrain[:,:indiceTraining]
XV = Xtrain[:, indiceTraining:indiceTraining+indiceValidation:1]
YV = Ytrain[:, indiceTraining:indiceTraining+indiceValidation:1]
print("XT shape:", XT.shape)
print("YT shape:", YT.shape)
print("XV shape:", XV.shape)
print("YV shape:", YV.shape)

#Training
funzione_errore=err_fun.crossEntropyWithSoftMax
start_time = time.time()
error_training, error_validation,accuracy_training,accuracy_validation=
net_lib.train(network_1,funzione_errore,XT,YT,XV,YV, 500, 0.00001,1,4000)
print("Tempo totale di esecuzione della fase di training: %s secondi" % (time.time() - start_time))

plt.figure()
plt.plot(error_training,'b',label='Errore sul Training set')
plt.plot(error_validation,'r',label='Errore sul Validation set')
plt.legend()
plt.show()
```

```
plt.figure()
plt.plot(accuracy_training,'b',label='Accuratezza sul Training set')
plt.plot(accuracy_validation,'r',label='Accuratezza sul Validation set')
plt.legend()
plt.show()

accuracy_testSet=net_lib.accuratezza_rete(network_1,Xtest,Ytest)
print('Accuratezza sul test set:', accuracy_testSet)
accuracy_trainingSet=net_lib.accuratezza_rete(network_1,Xtrain,Ytrain)
print('Accuratezza sul training set:', accuracy_trainingSet)
```

```
import numpy as np

def sigmoid(x,flag):
    y = 1 / (1 + np.exp(-x))
    if flag==0:
        return y
    else:
        return y, y*(1-y)

def identity(x,flag):
    y=x
    if flag==0:
        return y
    else:
        return y,1

def tanh(x,flag):
    y=(1-np.exp(-2*x))/(1+np.exp(+2*x))
    if(flag==0):
        return y
    else:
        return y, 1-y*y

def relu(x,flag):
    y=(x>0)
    if flag==0:
        return y*x
    else:
        return y*x, y*1
```

```
import numpy as np
import matplotlib.pyplot as plt

def loadDataset(data_path = './MNIST/'):
    image_size=28
    classes=10

    train_data=np.loadtxt(data_path + "mnist_train.csv", delimiter=",")
    test_data=np.loadtxt(data_path + "mnist_test.csv", delimiter=",")
```

```
    #creazione delle 4 matrici xtrain, ytrain, xtest, ytest
    xtrain=np.asarray(train_data[:,1:])
    ytrain=np.asarray(train_data[:,:1])
    xtest=np.asarray(test_data[:,1:])
    ytest=np.asarray(test_data[:,:1])

    #trasformazione delle etichette in rappresentazione one-hot
    range=np.arange(classes) #[0 1 2 3 4 5 6 7 8 9]
    #Se la classe di appartenenza è stata incontrata, poni 1 nella lista,altrimenti 0
    ytrain = (range==ytrain).astype(int)
    ytest = (range==ytest).astype(int)

    return xtrain.transpose(), ytrain.transpose(), xtest.transpose(), ytest.transpose()

#Funzione per la visualizzazione di un'immagine selezionata
def showImage(immagine):
    Immagine=immagine.reshape[(28,28)]
    plt.imshow(Immagine,'gray')
    plt.show()
```

```
#In questo file sono definite le due funzioni d'errore da sfruttare secondo la traccia
import numpy as np

def sumOfSquares(y,t,flag):
    if(flag==0):
        return (1/2)*np.sum(np.power((y-t),2))
    else:
        return y-t

def crossEntropyWithSoftMax(y,t,flag):
    z=softMax(y)
    if flag==0:
        return -(t*np.log(z)).sum()
    else:
        return z-t

def softMax(y):
    esponenziale=np.exp(y)
    z=esponenziale/sum(esponenziale)
    return z
```

```
import numpy as np
import activation_function as fun
import error_function


#Funzione per la costruzione della rete neurale
def build_network(input_size,num_hidden_neurons,output_size):
    pesi=[]
    bias=[]
    funzione_attivazione=[]
    j=input_size
```

```
    hidden_neurons=[num_hidden_neurons]
    #Costruzione degli strati interni della rete
    for i in hidden_neurons:
        #Definizione dei pesi iniziali della rete attraverso una gaussiana con dev.std=0.1 e media nulla
        #Vengono aggiunti alle 2 liste degli array di valori numerici estratti secondo la gaussiana, di i righe e j colonne
        bias.append(np.random.normal(loc=0.0,scale=0.1,size=[i,1]))
        pesi.append(np.random.normal(loc=0.0,scale=0.1,size=[i,j]))
        j=i
        #Definizione della funzione d'attivazione sullo strato hidden: sigmoide
        funzione_attivazione.append(fun.sigmoid)

    #Definizione dello strato d'output con la stessa logica
    pesi.append(np.random.normal(loc=0.0,scale=0.1,size=[output_size, j]))
    bias.append(np.random.normal(loc=0.0,scale=0.1,size=[output_size, 1]))
    funzione_attivazione.append(fun.identity)
    network={'Pesi':pesi,'Bias':bias,'Funzione d attivazione': funzione_attivazione, 'Profondità': len(pesi)}
    return network

#Funzione che restituisce le caratteristiche di una rete datale in input
def get_network_structure(network):
    lunghezza=len(network['Pesi'])
    numero_strati_hidden=lunghezza-1
    input_size= network['Pesi'][0].shape[1]
    output_size= network['Pesi'][numero_strati_hidden].shape[0]
    activation_function=[]
    numero_neuroni_strati_hidden=[]

    for i in range(numero_strati_hidden):
        numero_neuroni_strati_hidden.append(network['Pesi'][i].shape[0])
        activation_function.append(network['Funzione d attivazione'][i].__name__)
    activation_function.append(network['Funzione d attivazione'][numero_strati_hidden].__name__)
    print('Numero di strati hidden: ', numero_strati_hidden)
    print('Input size: ', input_size)
    print('Output size: ', output_size)
    print('Numero di neuroni per ogni strato hidden: ')
    for i in range(len(numero_neuroni_strati_hidden)):
        print(numero_neuroni_strati_hidden[i])
    print('Funzione d ativazione: ')
    for i in range(len(numero_neuroni_strati_hidden)+1):
        print(activation_function[i])
    return

#Funzione che permette di splittare il training set in: 80% training set effettivo e 20% validation set
#Vengono ritornati gli indici, secondo la logica descritta, per ricopiare i valori nei vari insiemi
def splitTrainingDataset(Y, percentuale):
    #Proporzioone => 20:100=x:YTrain.shape
    indiceValidation = int((percentuale*Y)/100)
    #Proporzione => 80:100=x:YTrain.shape
    indiceTraining = int(((100-percentuale)*Y)/100)
    return indiceTraining, indiceValidation

#Funzione che effettua la copia di una rete in un'altra
def copy_network(network):
    Pesi=[]
    Bias=[]
    numero_livelli=len(network['Pesi'])
    for i in range(numero_livelli):
        Pesi.append(network['Pesi'][i].copy())
```

```
        Bias.append(network['Bias'][i].copy())
    network_1={'Pesi':Pesi,'Bias':Bias,'Funzione d attivazione': network['Funzione d attivazione'], 'Profondità':
numero_livelli}
    return network_1

#Funzione che definisce la fase di learning della rete
#network: la rete da addestrare (Parametro di input e output)
#err_fun: funzione d'errore da usufruire
#XTrain,YTrain,XValid,YValid: training set e validation set (coppie di valori, insieme ai target)
#max_epoche: numero massimo di epoche da usufuire
#eta: iperparametro della rete necessario per definire la discesa del gradiente in fase di aggiornamento dei parametri
#flag: variabile che definisce la modalità di learning da seguire (batch, minibatch o online)
#numero_mini_batch: variabile che definisce il quantitativo di batch da creare nel caso si scelga la modalità minibatch
def train(network, err_fun, XTrain, YTrain, XValid, YValid, maxEpoche,eta,flag,numero_mini_batch):
    profondita=network['Profondità']
    errore_training=[]
    errore_validation=[]
    accuratezza_training=[]
    accuratezza_validation=[]
    if flag==0:
        print('Modalità batch')
        numero_mini_batch=1
        #Training set sarà passato tutto in una volta alla rete
        training_effettivo=split(YTrain,numero_mini_batch)
    elif numero_mini_batch>0:
        print('Modalità mini batch')
        #La rete verrà addestrata su un batch alla volta
        training_effettivo=split(YTrain,numero_mini_batch)
    else:
        print('Modalità online')
        #La rete verrà addestrata su pezzi pari alla quantità dei dati di ogni classe
        training_effettivo=[[i] for i in np.arange(YTrain.shape[1])]
    #Addestramento della rete sulla 1° epoca per ricavare la prima rete migliore
    y_training = forward_propagation(network, XTrain, XTrain,0)
    errore = err_fun(y_training, YTrain, 0)
    errore_training.append(errore)
    y_validation = forward_propagation(network, XValid, XValid,0)
    errore_val = err_fun(y_validation, YValid, 0)
    errore_validation.append(errore_val)
    minimo_errore_validation = errore_val
    rete_migliore = copy_network(network)
    accuratezza_training.append(accuratezza(y_training, YTrain))
    accuratezza_validation.append(accuratezza(y_validation, YValid))
    print('Epoca:', 0,
          'Errore sul training set:', errore_training,
          'Errore sul validation set:', errore_validation,
          'Accuracy sul training set:', accuratezza(y_training, YTrain),
          'Accuracy sul validation set:', accuratezza(y_validation, YValid))

    #Fase di learning effettiva
    for epoca in range(maxEpoche):     #Ciclo con il suo criterio d'arresto
        for i in training_effettivo:   #Si lavora sul quantitativo di dati effettivo, in base alla modalità di apprendimento
scelta
            output_strati,derivate_strati=forward_propagation(network,XTrain,XTrain[:,i],1)       #Fase forward che ritorna,
a mo' di liste, gli output di ciascun layer e le loro derivate
            derivate_pesi,derivate_bias=back_propagation(network,YTrain[:,i],err_fun,output_strati,derivate_strati)
#Fase di backpropagation per il calcolo di: delta valori e derivate
            for j in range(profondita):          #Fase di aggiornamento dei pesi secondo lo standard gradient descent
```

```
            network['Pesi'][j]=network['Pesi'][j]-eta*derivate_pesi[j]
            network['Bias'][j]=network['Bias'][j]-eta*derivate_bias[j]
        #Calcolo dell'errore, tramite error function e conseguete ricerca della rete migliore
        y_training = forward_propagation(network, XTrain, XTrain,0)
        errore = err_fun(y_training, YTrain, 0)
        errore_training.append(errore)
        y_validation = forward_propagation(network, XValid, XValid,0)
        errore_val = err_fun(y_validation, YValid, 0)
        errore_validation.append(errore_val)
        accuratezza_training.append(accuratezza(y_training, YTrain))
        accuratezza_validation.append(accuratezza(y_validation, YValid))
        if errore_val < minimo_errore_validation:
            minimo_errore_validation = errore_val
            rete_migliore = copy_network(network)
        print('Epoca:', epoca+1,
              'Errore sul training set:', errore,
              'Errore sul validation set:', errore_val,
              'Accuracy sul training set:', accuratezza(y_training, YTrain),
              'Accuracy sul validation set:', accuratezza(y_validation, YValid))
        print('\r', end='')

    network=copy_network(rete_migliore)
    return errore_training,errore_validation,accuratezza_training,accuratezza_validation

def split(YTrain,k):
    dimensione_dataset=YTrain.shape[0]
    #Creazione dei k batch (al momento di dimensione pari a 0)
    indiciYTrain = [np.ndarray(0, int) for i in range(k)]
    for i in range(dimensione_dataset): #Scorrimento da 0 a 9 (classi)
        v=(YTrain.argmax(0)==i)        #Se l'elemento, di YTrain, più grande (cioè 1) è in posizione i (cioè la classe corrente),
metti true, altrimenti false
        indici=np.argwhere(v==True).flatten()    #Costruisci un array 2D contenente gli indici degli elementi di v posti a 1.
Con flatten si definisce l'array in 1 dimensione
        #Split dell'array appena costruito in k pezzi
        new_index=np.array_split(indici,k)
        for j in range(len(new_index)):
            indiciYTrain[j]=np.append(indiciYTrain[j],new_index[j])
    for i in range(k):     #shuffle dei dati
        indiciYTrain[i]=np.random.permutation(indiciYTrain[i])
    return indiciYTrain

#Funzione che definisce la fase di forward_propagation, cioè fase di calcolo dei valori principali della rete:
#restituisce o l'insieme degli output di tutti gli strati e le derivate delle funzioni d'attivazione di tutti gli strati
#o l'unico output della rete
def forward_propagation(network,X,X1,flag):
    Pesi=get_pesi_rete(network)
    Bias=get_bias_rete(network)
    Funzioni_Attivazione=get_funzione_attivazione_rete(network)
    profondita=network['Profondità']
    if flag==0:    #forward_propagation per il semplice calcolo di y della rete (l'output)
        z=X
        for i in range(profondita):
            #per ogni livello calocla l'input tramite prodotto scalare tra W e Z + il baias
            a=np.matmul(Pesi[i],z)+Bias[i]
            #Calcolo dell'output tramite applicazione della funzione d'attivazione sull'input appena calcolato
            z=Funzioni_Attivazione[i](a,0)
        return z
    else:
```

```
    a=[]
    z=[]
    derivate_correnti=[]
    z.append(X1)
    #Ciclo che calcola l'input (tramite prodotto scalare e bias esplicito) e l'output tramite l'applicazione della funzione d'attivazione
    for i in range(profondita):
      #Calcolo dell'input di uno strato e salvataggio una struttura dedicata
      a.append(np.matmul(Pesi[i],z[i])+Bias[i])
      #Calcolo dell'output: salvatggio in due strutture dedicate, una per gli output e un'altra per il caloclo delle derivate
      #Questo perchè ci occorrerà in fase di back_propagation la derivata della funzione d'attivazione
      output_corrente, derivata_output_corrente = Funzioni_Attivazione[i](a[i],1)
      derivate_correnti.append(derivata_output_corrente)
      z.append(output_corrente)
  return z,derivate_correnti

#Funzione che definisce la fase di back_propagation: calcolo dei delta e derivate della funzione d'errore
def back_propagation(network, t, error_function,output,derivate):
  Pesi=network['Pesi']
  Bias= network['Bias']
  profondita=network['Profondità']
  #Calcolo della derivata della funzione d'errore sull'ultimo valore, cioè l'output
  derivata_k=error_function(output[-1],t,1)
  #Calcolo del delta valore riferito allo strato d'output
  delta_valori=[]
  delta_valori.insert(0,derivate[-1]*derivata_k)
  #Partendo dall'ultimo strato e salendo, calcolo gli altri delta value
  for i in range(profondita-1,0,-1):
    #Si segue la seguente formula (per il calcolo di DeltaH): sommatoria(Wkh*Deltak)
    delta_corrente=derivate[i-1]*np.matmul(Pesi[i].transpose(),delta_valori[0])
    delta_valori.insert(0,delta_corrente)
  #Calcolo delle derivate tramite legge locale: Derivata di E(sull'n-esima coppia rispetto ai parametri)=Delta(i)*Z(j)
  derivate=[]
  bias_derivate=[]
  for i in range(0,profondita):
    derivata_corrente=np.matmul(delta_valori[i],output[i].transpose())
    derivate.append(derivata_corrente)
    bias_derivate.append(np.sum(delta_valori[i],1,keepdims=True))
  return derivate,bias_derivate

#Calcolo dell'accuratezza: numero casi giusti/numero casi totali
def accuratezza(y,t):
  numero_casi_totali=t.shape[1]
  z=error_function.softMax(y)
  #Il caso è detto giusto se il target rilasciato dalla rete = al target vero
  return np.sum(z.argmax(0)==t.argmax(0))/numero_casi_totali

def accuratezza_rete(network,X,t):
  y=forward_propagation(network,X,X,0)
  return accuratezza(y,t)

#Funzione che restituisce i pesi della rete
def get_pesi_rete(network):
  W=network['Pesi']
  return W

#Funzione che restituisce i bias della rete
```

```
def get_bias_rete(network):
    Bias=network['Bias']
    return Bias

#Funzione che restituisce le funzioni d'attivazione applicati ai vari strati della rete
def get_funzione_attivazione_rete(network):
    Funzione_Attivazione=network['Funzione d attivazione']
    return Funzione_Attivazione
```