UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

Neural Networks and Deep Learning

Studio dell'apprendimento di una rete neurale confrontando le varianti della RProp

Alessandro Mauro **N97000371**

Progetto di Neural Networks and Deep Learning

Anno Accademico 2020/21

PREFAZIONE

Nel seguente documento saranno illustrati i vari passaggi, ragionamenti e implementazioni svolte per risolvere il problema di classificazione per il *dataset* **mnist**. Saranno nell'ordine analizzati:

- Specifiche del problema
- Analisi di risoluzione del problema
- Tecniche implementative, con varie implementazioni in *Matlab*
- Analisi dei risultati ottenuti
- Considerazioni

La parte I è l'introduzione. Nell'introduzione saranno spiegate le specifiche del problema. Inoltre saranno date delle breve nozioni su cosa si intende risolvere un problema di classificazione.

Nella parte II sarà mostrata l'implementazione scelta per risolvere il problema. La parte si divide in 3 sezioni: "Parte A", "Parte B" e "Il processo di learning". Nella sezione *Parte A* sarà mostrato come costruire e simulare il comportamento di una rete multistrato e come implementare la back propagation. Nella sezione *Parte B* sarà approfondito l'articolo *Empirical evaluation of the improved Rprop learning algorithms*: saranno mostrate le idee alla base delle varianti dell'RProp, il loro significato grafico e come tali varianti sono state implementate. Nella sezione *Il processo di learning* sarà mostrato come tutte le implementazioni mostrate nelle parti A e B sono state combinate per risolvere effettivamente il problema di classificazione. Per ogni implementazione saranno analizzati gli elementi necessari, il flusso della implementazione e sarà mostrato un frammento di codice in Matlab relativo alle operazioni salienti.

Nella parte III saranno analizzate e mostrate le performance delle varianti dell'RProp. Saranno dapprima fatte alcune considerazioni in merito agli iperparametri standard. Dopodiché sarà fatta una scelta degli iperparametri più formale e sarà rivalutato il tutto.

Nella parte IV saranno effettuate delle considerazioni conclusive sulla regola di aggiornamento RProp e sulle valutazioni ottenute

È stato utilizzato ***Matlab*** come linguaggio di programmazione in quanto è ottimizzato per i calcoli matriciali.

Il progetto è stato svolto dallo studente Alessandro Mauro (N97000371).

Contents

Part I
INTRODUZIONE

1 Specifiche del progetto

L'esercitazione prevede la realizzazione di una rete neurale multistrato feed forward che permette di fare **classificazione** sul *dataset* di immagini ***mnist***.

Nello specifico, la traccia del progetto consta nella realizzazione di due macro-punti:

- **Parte A**
  - Implementazione di funzioni per **simulare la propagazione in avanti di una rete** multi-strato, dando la possibilità di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione per ciascuno strato.
  - Implementazione di funzioni per **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 funzone di errore.
- **Parte B**
  - Considerando come input le immagini del *dataset* ***mnist***, si ha 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.
  - Si confronti la classica resilient backpropagation (RProp) con almeno 2 varianti proposte nell'articolo "*Empirical evaluation of the improved Rprop learning algorithms, Christian Igel, Michael Husken, neurocomputing, 2003*". Si fissi la funzione di attivazione ed il numero di nodi interni e si confrontino i risultati ottenuti con i diversi algoritmi di apprendimento.

In altre parole, la parte A prevede la realizzazione di una rete neurale multi-strato e la realizzazione della back-propagation, mentre la parte B prevede di risolvere un problema di classificazione utilizzando la RProp e le sue varianti come algoritmo di aggiornamento dei parametri, confrontando i risultati ottenuti.

2 Le reti neurali e problemi di classificazione

Prima di procedere alla risoluzione dei punti A e B, nella seguente sezione sarà spiegato brevemente che cos'è una rete neurale e cosa significa risolvere un problema di classificazione.

Una rete neurale è un insieme di elementi di base (detti **neuroni**) collegati tra loro mediante delle **connessioni pesate** W_{ij}. Una rete neurale è formata dalle seguenti componenti: uno **strato di input** di dimensione d, ***H-1* strati hidden** di dimensioni $m_1, .., m_{h-1}$ e **uno strato di output** di dimensioni c. In una rete **multi-strato full-connected tutti i neuroni** dello strato h ricevono connesioni solo da **tutti i neuroni** dallo strato $h-1$.

In un problema di classificazione, l'obiettivo è quello di identificare a quale classe appartiene un determinato input. In un problema supervisionato (così come il problema studiato in questo documento), le classi sono predefinite, ovvero si conoscono a priori.

Un approccio di Machine Learning utilizzando le reti neurali, permette di adottare un approccio standard che sia lo stesso per qualsiasi problema di classificazione. Il procedimento è il seguente:

1. **Raccogliere un *dataset***
   Un *dataset* è un insieme di coppie $\{(x^n, t^n)\}_{n=1}^N$ in cui $x^n \in \mathbb{R}^d$, $t^n \in \mathbb{R}^c$ e N =numero di valori di input del problema. t^i indica x^i a quale classe appartiene.

2. **Fissare un modello di rete**
   Viene fissato un modello di rete, ovvero viene fissato lo "*scheletro*" della rete. Sono dunque fissati **gli iperparametri** m (numero nodi interni), f (funzione di attivazione strato interno), g (funzione di attivazione strato output), c (numero di classi, solitamente date in input) e altri iperparametri legati all'algoritmo utilizzato per l'aggiornamento dei parametri. A questo punto, rimangono da fissare i **parametri** W, b.

3. **Risolvere il problema**
   Avendo fissato un modello di rete, restano da trovare i **parametri pesi ($\boldsymbol{W}$ e bias $\boldsymbol{b}$)** della rete. Si può dimostrare che $y_k(x)$ (la risposta del k-esimo neurone di output quando la rete riceve in input x) corrisponde alla **probabilità che, dato x, x appartenga alla classe C_k**. Se si riesce ad ottenere una rete interpretabile in questo modo, allora è possibile costruire un classificatore applicando una **regola di decisione** (ad esempio, x apparterrà alla classe che ha probabità maggiore).

Quindi, dato un *dataset* e una regola di decisione, **il goal è approssimare al meglio possibile** $P(C_k|x)$ tramite la rete, **trovando i parametri migliori**.

In modo da stabilire la bontà dei parametri, si introduce una **funzione di errore $\boldsymbol{E}$**. Il processo di learning è un processo iterativo che consiste nel modificare volta per volta i parametri al fine di **minimizzare la funzione di errore**.

Dunque, si vuole trovare $\Theta^* = argmin(E(\Theta))$. Per calcolare il minimo si uti-

lizza qualche forma di **discesa del gradiente**[1]. Il gradiente sarà calcolato tramite l'algoritmo di **Back-Propagation**. Una volta calcolato il gradiente, saranno aggiornati i parametri mediante una **regola di aggiornamento**.

[1]Il gradiente è un vettore di derivate parziali che da informazioni sulla direzione e verso in cui muoversi (nello spazio dei parametri) per raggiungere il minimo)

Part II
IMPLEMENTAZIONE

In questa parte saranno mostrati i ragionamenti effettuati per risolvere il problema. Saranno inoltre mostrate le varie specifiche degli algoritmi risolutivi attraverso la loro formalizzazione in diagrammi di flusso e/o implementazioni in *Matlab*.

Nella Parte A sarà mostrato come implementare una rete neurale multistrato feed forward, il suo funzionamento e come implementare la back propagation per calcolare il gradiente della funzione di errore.

Nella Parte B saranno mostrate le varianti dell'algoritmo di aggiornamento dei parametri RProp, i loro algoritmi e le rispettive implementazioni in Matlab.

Nella sezione "Il processo di learning", sarà mostrato come le varie implementazioni sono state allacciate per permettere il processo di apprendimento dei parametri e risolvere il problema.

3 Parte A

Si ricordano le specifiche della Parte A. Essa prevede la realizzazione dei seguenti punti:

1. Implementazione di funzioni per **simulare la propagazione in avanti di una rete** multi-strato, dando la possibilità di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione per ciascuno strato.

2. Implementazione di funzioni per **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 funzone di errore.

Si pone l'attenzione al punto 1: l'obiettivo è quello di implementare funzioni per simulare la propagazione in avanti. A tal proposito, è necessaria l'implementazione di due funzioni principali:

- Una funzione che permetta di **costruire la rete**, ovvero fissare gli iperparametri e inizializzare i parametri.

- Una funzione che permette di **simulare il comportamento della rete**. Ovvero, dato un input $x \in \mathbb{R}^d$ restituire un output $y \in \mathbb{R}^c$ che corrisponde all'output dei neuroni dello strato di output.

L'idea implementativa riguardante il punto 1 è discussa nella sezione 3.1

Invece, per quanto riguarda il punto 2, l'obiettivo è quello di implementare funzioni per realizzare la back-propagation. Ciò sarà discusso nella sezione 3.2

3.1 Implementazione propagazione in avanti

Come già accennato, per implementare la propagazione in avanti di una rete multi-strato è necessario utilizzare due funzioni principali. Bisogna innanzitutto implementare una funzione che permette di **costruire la rete**. Dopodiché, si costruisce una funzione che simula il **comportamento della rete**. Tali funzioni saranno implementate in Matlab tramite le funzioni ***newNet*** (per la costruzione della rete) e ***simNet*** (per la simulazione del comportamento della rete).

3.1.1 Costruzione della rete multistrato

L'obiettivo della funzione è quello di costruire una struttura che contenga le informazioni della rete. Per poter costruire la rete neurale sono necessarie le seguenti informazioni in input:

- **d** $\rightarrow$ Un intero che rappresenta il numero dei nodi di input
- ***m*** $\rightarrow$ Un array in cui sono contenuti **il numero dei nodi** degli strati interni. La lunghezza di m determinerà **il numero di strati interni** ($m(1)$ numero nodi dello strato 1,..., $m(h)$ numero nodi dello strato h)
- **c** $\rightarrow$ Numero dei nodi dello strato di output
- **f** $\rightarrow$ Funzioni di attivazione degli strati interni[2].
- **g** $\rightarrow$ Funzione di attivazione dello strato di output

Attraverso queste informazioni, viene creata una rete multistrato full-connected, **composta da H layer** totali in cui ci sono:

- d variabili di ingresso $[x_1, ..., x_d]$
- $H - 1$ hidden layer in cui ogni layer h contiene m_h neuroni
- Un layer di output composto da $m_H \equiv c$ neuroni

Sarà generata una rete come mostrato in Figura 1.

Pesi e i bias A questo punto, una volta definito lo scheletro della rete, saranno inizializzati i valori dei ***pesi*** W ***e bias*** b:

- **I pesi** W possono essere considerati come delle matrici corrispondenti ai **pesi delle connessioni entranti in uno strato**. Per quanto riguarda lo strato i, la matrice W^i corrisponde al peso delle connessioni entranti nello strato i e che partono dallo strato $i - 1$. Quindi, la matrice dei pesi avrà:
  - come **numero di righe**, il numero di nodi dello strato h.

[2] È possibile assegnare qualsiasi funzione di attivazione per qualsiasi strato

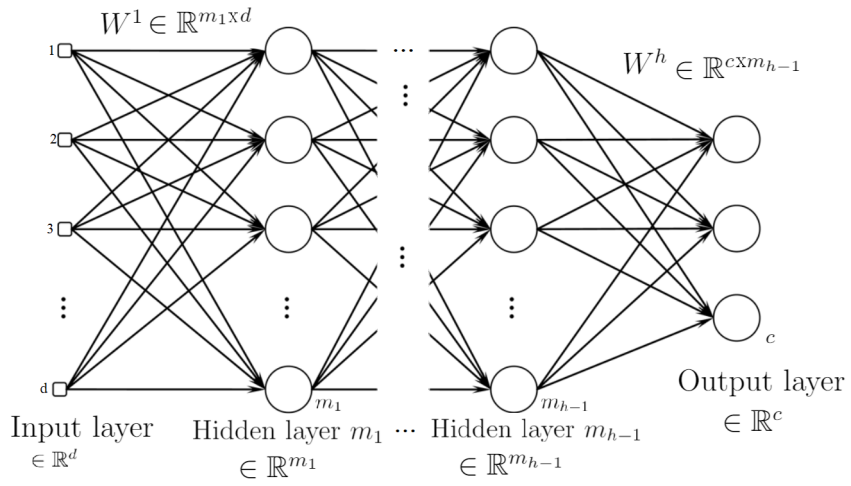


Figure 1: Rete full-connected con d neuroni di ingresso, $m_1, ..m_{h-1}$ nodi interni e c nodi di output

  - come **numero di colonne**, il numero di nodi dello strato $h-1$. Nel caso in cui $h-1=0$, il numero di colonne sarà uguale a d (numero variabili d'ingresso).

  Dunque, la matrice avrà dimensione $W^i \in \mathbb{R}^{m_i \times m_{i-1}}$.
  Sulla prima riga ci sono i pesi del primo neurone, sulla seconda riga i pesi del secondo neurone, e così via.

- Il bias è considerato un vettore colonna. Ogni bias fa riferimento ad ogni nodo di uno strato. Per quanto riguarda lo strato i, il bias avrà lunghezza uguale al numero di nodi dello strato i $[b^i \in \mathbb{R}^{m_i \times 1}]$

Lo scopo della rete è quello di trovare i valori migliori di pesi e bias per risolvere il problema. Una prima inizializzazione dei pesi viene fatta in modo **random**.

In Matlab Per poter realizzare in Matlab la costruzione di una rete multistrato full-connected, si è implementata la funzione ***newNet***.

chiamata a funzione newNet

```
net=newNet(d,m,c,f,g)
```

La funzione necessita dei parametri *d,m,c,f,g*, in cui d è un intero, m è un intero oppure un array, c è un intero, f è un *cell_array* in cui ogni $f\{i\}$ è un $function\_handle$ e corrisponde alla funzione di attivazione dello strato i e g è un $function\_handle$.

La funzione restituisce una struttura *net* dove sono specificati i valori di *d,m,c,f,g*. Nel caso in cui la lunghezza di m sia superiore ad 1, gli strati interni saranno più di uno. Il numero di strati interni corrisponde alla lunghezza di m.

Il parametro f è un *cell_array* contenente *function_handle* relativi alle funzioni di attivazione degli strati interni. Per comodità, nel caso di più strati interni, è possibile

passare un *cell_array* contentente un solo function_handle: in questo caso, sarà assegnata la stessa funzione di attivazione a tutti gli strati interni. Se invece si vuole implementare strati interni con funzioni di attivazioni diverse, è necessario specificare le funzioni di ogni strato, specificando in f tutti i $function\_handle$.

Di seguito viene rappresentato il *flow-chart* della funzione *newNet*. A destra del diagramma sono chiariti i passaggi:

1. **Controllo input**, in cui si controlla che:
   - il numero di argomenti sia corretto
   - d, m, c siano numeri decimali (se m è un vettore, m contiene tutti numeri decimali)
   - d, m, c siano maggiori di 0 (se m vettore, m contiene tutti numeri > 0)
   - Il parametro f sia un *cell_array* contenente solo *function_handle*.
   - Il parametro g sia un *function_handle*
2. **Creazione strati interni**, in cui vengono inizializzati (in modo random) pesi e bias degli strati interni.
3. **Creazione strato output**, in cui vengono inizializzati (in modo random) pesi e bias per lo strato di output
4. **Definizione struttura net**, in cui vengono assegnati i valori dei campi (d,m,c,f,g,numLayers) della struttura net.

Di seguito si riporta un frammento della formalizzazione dell'algoritmo risolutivo in Matlab, che comprende la creazione degli strati interni e dello strato di output.

newNet.m

```
...
%% CREAZIONE STRATI INTERNI
SIGMA=0.2; prec_layer=d;
for i = 1:length(m)
    net.W{i} = SIGMA*randn(m(i),prec_layer);
    net.B{i} = SIGMA*randn(m(i),1);
    prec_layer=m(i);
end
%% CREAZIONE STRATO OUTPUT
net.W{i+1} = SIGMA*randn(c,m(length(m)));
net.B{i+1} = SIGMA*randn(c,1);
```

In modo tale da implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione, i valori dei pesi W e bias sono memorizzati utilizzando le strutture *cell array*. Tale struttura permette di memorizzare, in ogni cella, qualsiasi tipo di dato. Dunque, ogni cella $W\{i\}$ contiene **una matrice di dimensioni presumibilmente diverse**. La matrice $W\{i\}$ corrisponde ai pesi delle connessioni entranti ai neuroni dello *strato* i.

Le dimensioni delle matrici W^i sono rispettate in quanto con $m(i)$ si recupera il numero di nodi dello strato $i-esimo$ e con *prec_layer* si recupera il numero di nodi dello strato $i-1$. Dunque, la matrice $net.W\{i\}$ avrà dimensione $\mathbb{R}^{m_i \times m_{i-1}}$

Esempio *newNet* La funzione restituisce una struttura *net* contenente le informazioni sulla rete creata. Per crearla basterà richiamare la funzione *newNet* con i giusti parametri, ad esempio:

```
>> net=newNet(3, [4 3 4], 5, {@sigmoide}, @identity);
```

La rete neurale *net* risultante avrà la seguente struttura:

Esempio output newNet

```
W: {[4x3 double]  [3x4 double]  [4x3 double]  [5x4 double]}
B: {[4x1 double]  [3x1 double]  [4x1 double]  [5x1 double]}
d: 3
m: [4 3 4]
c: 5
f: {@sigmoide  @sigmoide  @sigmoide}
g: @identity
numLayers: 4
```

3.1.2 Comportamento della rete

Avendo definito come costruire una rete neurale, il comportamento della rete ha come obiettivo quello di restituire l'output della rete quando essa riceve in ingresso un input x. Il comportamento della rete può essere eseguito in forma matriciale. Ogni neurone effettua il **calcolo dell'input** e il **calcolo dell'output**. Utilizzando un calcolo matriciale è come se l'unità di base diventasse lo strato e non più il singolo neurone. Quindi si possono considerare le seguenti operazioni:

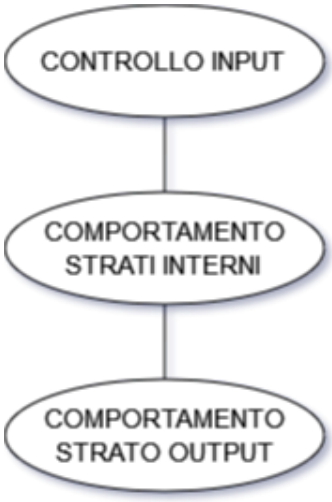
- **Calcolo input** $a \rightarrow$ Lo strato $i-esimo$ della rete calcola a_i effettuando il prodotto della matrice W^i * z_{i-1}. Nel caso in cui $i = 1$ (caso primo strato), $z_{i-1} \equiv x$. A tale prodotto verrà poi aggiunto il bias b^i.
- **Calcolo output** $z \rightarrow$ L'output dello strato $i-esimo$ della rete viene calcolato applicando la funzione di attivazione f^i all'input a_i.

In Matlab In Matlab, la funzione adibita a simulare il comportamento della rete è ***simNet***.

chiamata a funzione simNet

```
y=simNet(net,x)
```

Dove *net* è una struttura ottenuta dalla funzione *newNet* mentre x è l'input della rete. L'output restituito è y. La funzione *simNet* effettua i seguenti passaggi per calcolare l'output y:



1. **Controllo dimensioni input** → Si controlla che l'input sia adatto alla rete (Ovvero, che il numero di righe dell'input sia uguale a d).
2. **Comportamento strati interni** → Si calcolano $a_1, .., a_{H-1}$ e $z_1, ..., z_{H-1}$, ovvero gli input e gli output degli strati interni.
3. **Comportamento strato output** → Si calcolano input e output dello strato di output. L'output dello strato corrisponderà all'output della rete.

Un punto saliente della funzione simNet **è il calcolo dell'input e dell'output degli strati interni,** in modo particolare quando essi sono più di uno. Di seguito si riporta l'implementazione scelta.

simNet.m

```
...
%% COMPORTAMENTO STRATI INTERNI (a1...a_h-1 e z1..z_h-1)
z=x;
for i=1:(net.numLayers)-1
    a = net.W{i}*z + net.B{i};
    z = net.f{i}(a);
end
%% COMPORTAMENTO STRATO OUTPUT (calcolo a_c e y)
a = net.W{i+1}*z + net.B{i+1};
y = net.g(a);
```

Per quanto riguarda il calcolo del comportamento gli strati interni, si conserva di volta in volta il valore *dell'output dello strato precedente* nella variabile z. Nella prima iterazione (ovvero per il primo strato) $z = x$.

3.2 Implementazione back-propagation

È stato definito e implementato come costruire una rete neurale e calcolare il suo output. Il goal della rete è quello di *imparare* i giusti parametri che risolvano il

problema. L'apprendimento consiste nel trovare i parametri che **minimizzano una funzione di errore**, ovvero trovare $\theta^* = argmin_\theta E(\theta)$. Per trovare tale minimo si utilizza qualche forma di **discesa del gradiente**[3].

La back-propagation è la tecnica che permette di **calcolare il gradiente della funzione di errore rispetto ai parametri**. Una volta calcolato il gradiente, è possibile costruire una *regola di aggiornamento dei parametri* e definire così il processo di learning.

La back propagation prevede 3 step principali:

1. **Forward-Step** → In cui vengono calcolati (e conservati opportunamente) tutti gli input a_i e gli output z_i per tutti i nodi della rete.
2. **Calcolo delta**, che si dividono in
   - **Delta Output** $[\delta_k^n = g'(a_k^n) * \frac{dE^n}{dY_k}]$
   - **Delta Input** $[\delta_h^n = f'(a_h^n) * \sum_k W_{kh} * \delta_k^h]$
3. **Calcolo derivate** → Per ogni W_{ij} si calcola $\frac{dE^n}{dW_ij} = \delta_i^n * z_j^n$

In Matlab Per implementare la back-propagation in Matlab, è stata implementata la funzione ***backPropagation.m***.

La back-propagation deve garantire il funzionamento 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, per questo motivo viene passato ***funzErr*** come parametro alla funzione. Il parametro ***funzErr*** deve essere un *function_handle* di una funzione di errore implementata.

chiamata a funzione backPropagation

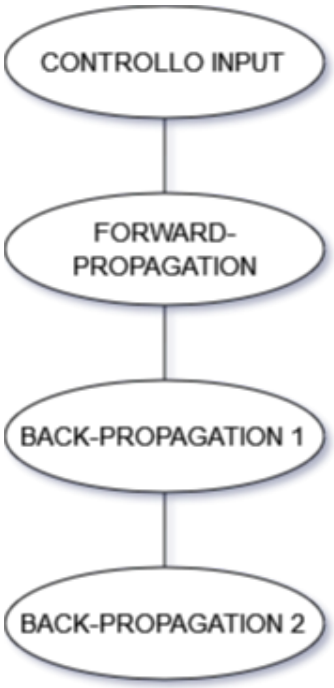
```
1 function gradiente=backPropagation(net,x,t,funzErr)
```

La funzione:

- **Prende in input** la rete *net*, l'input della rete x, i target degli input t e la funzione di errore *funzErr* che si vuole utilizzare.
- **Restituisce in output** la struttura ***gradiente*** contenente i campi:
  - ***gradiente.W***, che corrisponde ad un *cell_array* contenente le derivate dei pesi W (gradiente.W{1}, gradiente.W{2},....);
  - ***gradiente.B***, un *cell_array* contenente le derivate dei bias dei nodi.

[3]Il gradiente è un vettore di derivate parziali che dà informazioni sulla direzione e verso in cui muoversi nello spazio dei parametri per raggiungere il minimo

Di seguito viene rappresentato il diagramma di flusso della funzione *backPropagation* per calcolare le derivate. La funzione effettua i seguenti passaggi:



1. **Controllo input** → Si controlla che il numero di righe del target t sia uguale a *net.c*.
2. **Forward-propagation** → In cui vengono **calcolati e memorizzati tutti gli input** a_i **e gli output** z_i per tutti i nodi della rete. Ciò sarà fatto utilizzando la funzione *forwardStep*

```
function [A,Z,y]=forwardStep
    (net,x)
```

   La funzione si comporta come *simNet* ma restituisce in output anche i vari input e output dei nodi.
3. **Fase back-propagation 1** → In cui si effettua il **calcolo dei delta di output** ($\delta_k^n = g'(a_k^n) \cdot \frac{dE^n}{dy_k^n}$)
4. **Fase back-propagation 2** → In cui si effettua il **calcolo dei delta hidden** ($\delta_h^n = f'(a_h^n) \cdot \sum_k W_{kh}\delta_k^h$) (Con k che corre sui nodi che ricevono connessioni da h (quindi si back-propagano i delta all'indietro))
5. **Calcolo derivate** in cui si calcolano le derivate $gradiente.W^1, gradiente.W^2, ...$ $\frac{dE^n}{dW_{ij}} = \delta_i^n \cdot z_j^n$

Per poter calcolare i vari delta, è necessario **conoscere la derivata delle funzioni di attivazione e la derivata delle funzioni di errore**. Per fare ciò, sono state implementate due funzioni, rispettivamente:

```
function y=derivFunzAct(funzAct,x) %derivata funz attivazione
function z=derivFunzErr(funzErr,y,t) %derivata funz errore
```

Entrambe le funzioni effettuano uno switch in base alla funzione passata per parametro e ne restituiscono le derivate.[4]

[4]Le funzioni di attivazione implementate di default sono: sigmoide ed identità. Le funzioni di errore implementate di default sono: sumOfSquares e Cross Entropy con e senza softmax. Per poter garantire l'utilizzo di qualsiasi funzione di errore e di attivazione è necessario implementare un file per la funzione, e un file che ne calcoli la derivata.

Nella funzione *backPropagation* è possibile recuperare le informazioni relative a funzioni di attivazione e funzione di errore in quanto: **la funzione di attivazione** *funzAct* è conservata nella struttura *net* nei campi *net.g* e *net.f*; **la funzione di errore** è passata come parametro della funzione *backPropagation*. A questo punto, è possibile passare i giusti parametri alle funzioni per calcolarne le derivate.

Un punto saliente della funzione *backPropagation* **è il calcolo dei delta e delle derivate**. Di seguito si riporta un frammento per l'implementazione scelta.

backPropagation.m

```
...
%% FASE FORWARD-PROPAGATION
[A,Z,y] = forwardStep(net,x);

%% FASE BACK-PROPAGATION 1 (CALCOLO DELTA DI OUTPUT)
delta_out = derivFunzAct(net.g,A{H});
delta_out= delta_out .* derivFunzErr(funzErr,y,t);

%% FASE BACK-PROPAGATION 2 (CALCOLO DELTA HIDDEN)
delta_stratoSucc=delta_out;
for i=m:-1:1 %dall'ultimo strato interno al primo
    delta_hidden{i} = (net.W{i+1})' * delta_stratoSucc;
    delta_hidden{i} = delta_hidden{i} .*
        derivFunzAct(net.f{i}, A{i});
    delta_stratoSucc=delta_hidden{i};
end

%% CALCOLO DERIVATE [deru(Wi)=delta(i)*z(i-1)')
z_prev=x;
for i=1:m %calcolo derivate per gli m strati interni
    gradiente.W{i}=delta_hidden{i}*z_prev';
    z_prev=Z{i};
    gradiente.B{i} = sum(delta_hidden{i},2);
end

%calcolo derivata strato output (strato H)
gradiente.W{H} = delta_out*z_prev';
gradiente.B{H}= sum(delta_out,2);
```

4 Parte B

Nelle sezioni precedenti è stato mostrato come realizzare una rete neurale e come calcolare il gradiente della funzione di errore. Ciò che rimane da fare è definire una regola di aggiornamento dei parametri per *imparare* i giusti parametri che risolvono il problema. A tal proposito, Si pone l'attenzione allo sviluppo della *Parte B* del progetto. Se ne ricordano le specifiche:

1. Considerando come input le immagini del *dataset* ***mnist***, si ha 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.

2. Si confronti la classica resilient backpropagation (RProp) con almeno 2 varianti proposte nell'articolo "*Empirical evaluation of the improved Rprop learning algorithms, Christian Igel, Michael Husken, neurocomputing, 2003*". Si fissi la funzione di attivazione ed il numero di nodi interni e si confrontino i risultati ottenuti con i diversi algoritmi di apprendimento.

In altre parole, la parte B prevede lo sviluppo di due sotto-punti: nel punto 1 bisogna estrarre il *dataset* ***mnist*** e suddividerlo opportunamente in *training set, validation set e test set*. Invece, nel punto 2 si richiede di confrontare i risultati della classificazione della classica RProp con le varianti proposte nell'articolo.

Nella sezione 4.1 si discuterà di come avviene la suddivisione del *dataset*, mentre nella sezione 4.2 si discuterà delle varianti della RPROP e di come esse sono state implementate.

4.1 Suddivisione Dataset

Il *dataset* ***mnist*** è un dataset contenete cifre scritte a mano. L'obiettivo è quello di estrarre il *dataset* ***mnist*** e di suddividerlo opportunamente in *training set, validation set e test set*. È importante effettuare tale suddivisione poiché lo scopo principale della rete non è quello di rispondere bene solamente sui dati di *training*, bensì lo scopo è quello di **generalizzare** e, quindi, rispondere bene anche su dati "nuovi" su cui non si è fatto train.

Per quantificare la capacità della rete di generalizzare si suddivide il *dataset* in *training set* e *validation set*. Il training set è utilizzato per aggiornare i parametri della rete, mentre il validation set è usato per verificare la capacità di generalizzare. In altre parole, ad ogni epoca sarà generata una rete diversa con un errore diverso, si sceglie la rete con minimo errore sul validation set, ovvero quella che generalizza meglio.

Avendo ottenuto ottenuto la rete che generalizza meglio, è necessaria una terza porzione di dati chiamata *test set*. Il *test set* è utilizzato per **valutare** la rete

ottenuta.

È importante che i tre insiemi (*training, validation* e *test* siano rappresentativi della popolazione. Una scelta possibile di suddivisione è quella di estrarre i tre set in maniera **randomica**.

Per quanto riguarda la **cardinalità dei set**, solitamente il *training set* rappresenta il 50% dei dati, il *validation set* il 25% dei dati e il *test set* il restante 25%

L'implementazione segue i seguenti passaggi:

1. Estrazione delle immagini X e delle *label* $Label$
2. Estrazione dei $Target$ dalle $Label$
3. Shuffle degli indici per suddivisione randomica
4. Suddivisione del *dataset*

Di seguito viene riportata l'implementazione in Matlab per estrarre e suddividere opportunamente il dataset *mnist*:

Estrazione dataset

```
%% ESTRAZIONE IMMAGINI E LABELS
X=loadMNISTImages('mnist/t10k-images-idx3-ubyte');
Labels=loadMNISTLabels('mnist/t10k-labels-idx1-ubyte');

%% ESTRAZIONE TARGET
T=getTargetsFromLabels(Labels);

%% SHUFFLE
ind=randperm(size(X,2));
X=X(:,ind);
T=T(:,ind);

%% SUDDIVISIONE DATASET IN TRAINING, VALIDATION E TEST
half=size(X,2)/2;
three_quarter=half+size(X,2)/4;

XTrain= X(:,1:half);
TTrain= T(:,1:half);

XVal=X(:,half+1:three_quarter);
TVal= T(:,half+1:three_quarter);

XTest=X(:,three_quarter+1:end);
TTest= T(:,three_quarter+1:end);
```

4.2 La RPROP

RPROP (*Resilient back-propagation)* è un **algoritmo di aggiornamento dei parametri** in modalità batch per l'apprendimento di reti neurali. L'idea alla base è quella di attenuare la dipendenza dei risultati dagli iperparametri.

A tal proposito, si associa ad ogni parametro W_{ij} uno **step di aggiornamento** Δ_{ij}. Tali Δ_{ij} saranno opportunamente modificati durante il learning.

Osservazione Δ_{ij} Ad ogni iterazione (epoca) t, i vari Δ_{ij} saranno incrementati o decrementati a seconda del prodotto della derivata parziale all'iterazione t per la derivata parziale all'iterazione $t-1$. Il motivo è abbastanza intuibile: considerando l'esempio in figura 2, all'iterazione *t-1*, la derivata risulta essere minore di zero (dunque la funzione in quel punto è decrescente). All'iterazione t i casi possibili sono due: ci si può trovare al punto blu (in cui la derivata diventa >0, e dunque funzione crescente) oppure al punto rosso (in cui la derivata è ancora minore di zero).

- Spostarsi al punto blu significa **superare il minimo locale**. Ciò avviene perché sono stati fatti incrementi troppo grandi. **Dunque, bisogna fare incrementi più piccoli**.
- Spostarsi al punto rosso, invece, significa **spostarsi verso la direzione giusta per raggiungere il minimo locale**. In questo caso ci si può permettere di **fare incrementi più grandi** per velocizzare il processo.

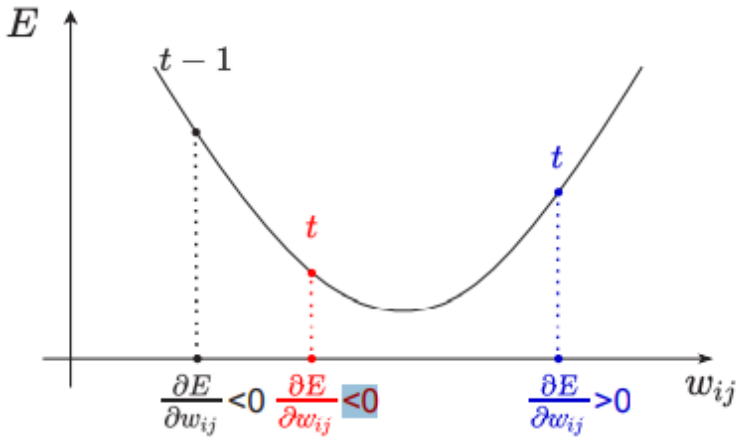


Figure 2:

Indicando con $g_{ij}^t \equiv \frac{\delta E^t}{\delta W_{ij}}$ la derivata della funzione di errore allo step (epoca) t, si può concludere che:

- se $(g_{ij}^t \cdot g_{ij}^{t-1} < 0)$ è stato effettuato un incremento troppo grande (caso punto blu). Dunque, bisogna **decrementare** Δ_{ij}
- se $(g_{ij}^t \cdot g_{ij}^{t-1} > 0)$ è stato effettuato un incremento piccolo (caso punto rosso). Dunque, ci si può permettere di **incrementare** Δ_{ij}

A tal proposito, si introducono due nuovi iperparametri η^+ ed η^- con $0 < \eta^- < 1 < \eta^+$ Utilizzando η^+ si incrementa Δ_{ij}. Invece, utilizzando η^- si decrementa.

Il valore dei Δ_{ij} viene regolato mediante la seguente regola

$$\Delta_{ij}^t := \left\{ \begin{array}{ll} min(\eta^+ \cdot \Delta^{t-1}, \Delta_{max}) & if(g_{ij}^t \cdot g_{ij}^{t-1}) > 0 \\ max(\eta^- \cdot \Delta^{t-1}, \Delta_{min}) & if(g_{ij}^t \cdot g_{ij}^{t-1}) < 0 \\ \Delta_{ij} & altrimenti \end{array} \right\} \quad (1)$$

I vari Δ_{ij} sono delimitati da valori soglia Δ_{max} e Δ_{min}.

Come accennato, la RPROP mira ad attenuare la dipendenza dagli iperparametri. Di fatto, si possono tranquillamente settare i valori degli iperparametri ai loro valori di default: $\eta^+ = 1.2$, $\eta^- = 0.5, \Delta_{min} = 0, \Delta_{max} = 50$ ed ottenere risultati accettabili.

Osservazioni Sorgono due osservazioni: la prima riguarda **la prima epoca**, dove non ci sono derivate precedenti (*t-1*) da considerare. Ciò si può risolvere effettuando una classica discesa del gradiente al primo step. La seconda osservazione riguarda **l'inizializzazione** dei Δ_{ij}. Essi possono essere inizializzati tranqullamente al valore 0.125, oppure attraversi valori randomici.

A questo punto, la regola di aggiornamento dell'RProp è la seguente:

$$W_{ij}^{t+1} = W_{ij}^t + \Delta W_{ij}^t \quad (2)$$

Dove ΔW_{ij} è il **modificatore dei pesi** e, a **seconda della variante RProp,** ΔW_{ij} **assume un valore diverso**.

Di seguito è mostrata l'implementazione in Matlab per poter Calcolare i vari Delta_ij e aggiornare i parametri a seconda della variante scelta

RPROP.m

```
if epoch==1
    net=discesaDelGradienteStandard(net,eta, gradiente);
else
for i=1:net.numLayers
    %% prodotto gradiente attuale e gradiente precedente
    gg = gradiente.W{i}.*oldGrad.W{i};
    ggB = gradiente.B{i}.*oldGrad.B{i};

    %% Calcolo Delta_ij
    Delta.W{i} =...
        min(Delta.W{i}*eta_p,Delta.max.W{i}).*(gg>0) +...
        max(Delta.W{i}*eta_n,Delta.min.W{i}).*(gg<0) +...
        Delta.W{i}.*(gg==0);

    Delta.B{i} =...
```

```
16          min(Delta.B{i}*eta_p,Delta.max.B{i}).*(ggB>0) +...
17          max(Delta.B{i}*eta_n,Delta.min.B{i}).*(ggB<0) +...
18          Delta.B{i}.*(ggB==0);
19
20      %% Calcolo modificatoriW e B in base alla variante di
           RPROP
21      switch(method)
22          case 'rprop-'
23              ...
24          case 'rprop+'
25              ...
26          case 'irprop+'
27              ...
28          case 'irprop-'
29              ...
30      end
31      %% AGGIORNAMENTO PESI
32      net.W{i} = net.W{i} + modificatoreW;
33      net.B{i} = net.B{i} + modificatoreB;
34  end
35  end
```

L'implementazione rispetta la condizione che ad ogni peso W_{ij} venga associato un parametro Δ_{ij}. Difatti, la variabile Delta è una struttura che contiene i seguenti campo:

- *Delta.W*, ovvero i Δ_{ij} **relativi ai pesi** → Questo campo è un cell array in cui all'i-esima posizione sono contenuti i Delta dei pesi W^i. Dunque, le dimensioni di $Delta.W\{i\}$ coincidono con le dimensioni di $net.W\{i\}$
- *Delta.B*, ovvero i Δ_{ij} **relativi ai bias** → Questo campo è un cell array in cui l'i-esima posizione contiene i Delta dei bias b^i. Dunque, le dimensioni di $Delta.B\{i\}$ coincidono con le dimensioni di $net.B\{i\}$
- *Delta.max*, ovvero i Δ_{max}. Questo campo è una struttura che contiene i campi $Delta.max.W$ e $Delta.max.B$. Questi sottocampi contengono dei cell array relativi ai valori massimi dei Δ_{ij}. Le dimensioni di $Delta.max.W\{i\}$ coincidono con $net.W\{i\}$.
- *Delta.min*, ovvero i Δ_{min}. Per questo campo è stato fatto un ragionamento analogo ai *Delta.max*

Inoltre, l'implementazione rispetta il calcolo dei Delta mostrato nell'equazione 1. Nella variabile *gg* viene memorizzato il prodotto tra il gradiente attuale e il gradiente dell'epoca precedente. L'espressione $(gg > 0)$ ritorna una matrice delle stesse dimensioni di *gg* con tutti uno nelle celle in cui il valore è > 0, zero altrimenti. Quindi, effettuando un prodotto *element wise* tra $(Delta.W\{i\} * eta_p).*(gg > 0)$ si ottiene una matrice in cui nelle celle dove il valore del prodotto dei gradienti è positivo $(gg > 0)$, il valore dei Delta è $(Delta.W\{i\} * eta\_p)$. Lo stesso ragionamento

è implementato nei casi di $gg < 0$ e $gg == 0$. Sommando queste 3 matrici, si ottengono i Delta desiderati.

4.2.1 RPROP-

Nell'implementazione standard (***Rprop without weight-backtracking*** o anche $RPROP^-$), ΔW_{ij} assume valore:

$$\Delta W_{ij}^t = -sign(\frac{\delta E^t}{\delta W_{ij}}) \cdot \Delta_{ij}. \tag{3}$$

Ad ogni iterazione (epoca) t, ogni componente W_{ij}^t è incrementato o decrementato a seconda del segno della derivata parziale della funzione di errore all'iterazione t ($\frac{\delta E^t}{\delta W_{ij}}$).

In questo modo, se si supera un minimo locale (segno derivata positivo), viene decrementato il peso (perché si considera il segno opposto della derivata), altrimenti, se si procede verso un minimo locale, viene aumentato il peso (segno derivata negativo).

La RProp- considera solamente il **segno delle derivate parziali della funzione di errore rispetto ai parametri**, e non i loro valori effettivi. Il segno della derivata parziale permette di sapere in quale direzione spostarsi per raggiungere il minimo locale.

È necessario memorizzare anche la derivata all'iterazione precedente per aggiornare i Δ_{ij}.

Sostituendo i ΔW_{ij}^t all'equazione 2 si ottiene la nuova regola di aggiornamento. Di seguito sono riportati l'algoritmo in pseudocodice e l'implementazione in Matlab che permette di calcolare i modificatori ΔW_{ij}.

Algorithm 1: RPROP-

forall W_{ij} **do**
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) > 0$ **then**
    $\Delta_{ij}^t := min(\eta^+ \cdot \Delta_{ij}^t, \Delta_{max})$;
  end
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) < 0$ **then**
    $\Delta_{ij}^t := max(\eta^- \cdot \Delta_{ij}^t, \Delta_{min})$;
  end
  $W_{ij}^{(t+1)} := W_{ij}^t - sign(g_{ij}^t) \cdot \Delta_{ij}$;
end

RProp–

```
case 'rprop-'
    modificatoreW=-sign(gradiente.W{i}).*Delta.W{i};
    modificatoreB=-sign(gradiente.B{i}).*Delta.B{i};
```

4.2.2 RPROP+

RPROP+ (oppure *Rprop with weight-backtracking*) è una variante dell'Rprop. Consiste nel *back propagare* un update precedente (modificatore al tempo *t-1* (ΔW_{ij}^{t-1})) per alcuni (o tutti) dei pesi. La propagazione all'indietro dei modificatori ΔW_{ij} è deciso mediante un'euristica.

- Quando ci si sposta seguendo la direzione del gradiente (caso rosso della figura 2), si può incrementare lo step Δ_{ij} per un fattore η^+, così come accadeva per la RProp standard (RProp-).
- Nel caso in cui il gradiente cambia direzione, ovvero quando si supera un minimo locale (caso blu figura 2), si riduce lo step Δ_{ij} per un fattore η^- e si annulla il movimento verso quella direzione **ritornando al punto precedente**. Inoltre, si **considera la derivata parziale come se fosse 0**.

Le operazioni di annullare il movimento ritornando al punto precedente e considerare il gradiente a zero, producono degli effetti all'iterazione successiva. Ricordando che all'iterazione t lo *step-size* Δ_{ij} è stato dimezzato (perché moltiplicato per $\eta^- = 0.5$), all'iterazione $t+1$ si avrà che:

- Considerando il gradiente a zero, all'iterazione $t+1$ il prodotto tra i gradienti sarà 0 e dunque Δ_{ij} rimarrà invariato (e quindi dimezzato rispetto a quando c'è stato il superamento del minimo locale).
- Inoltre, siccome è stato *back-propagato* il peso dell'iterazione precedente, la derivata parziale rispetto al peso in quel punto sarà sicuramente < 0 e quindi i pesi saranno incrementati di uno *step-size*. Questo significa che il movimento per raggiungere il minimo locale avviene solamente lungo la "discesa della curva" (dove la derivata è negativa) (in un certo senso, i pesi saranno sempre incrementati, invece di decrementarli si ritorna allo step precedente).

L'effetto è che ci si continuerà a muovere verso il minimo locale ma questa volta con uno *step size* dimezzato (cercando di raggiungere il minimo lungo la curva decrescente).

Quindi, dopo aver calcolato i vari Δ_{ij}, i modificatori dei pesi ΔW_{ij} sono così determinati:

$$\Delta W_{ij}^t := \left\{ \begin{array}{ll} -sign(g^t) \cdot \Delta_{ij} & if(g_{ij}^t \cdot g_{ij}^{t-1} \geq 0) \\ -\Delta W_{ij}^{t-1};\ \textbf{and}\ g_{ij}^t = 0 & if(g_{ij}^t \cdot g_{ij}^{t-1} < 0) \end{array} \right\} \tag{4}$$

Tali aggiornamenti di ΔW_{ij}^t comportano la storicizzazione non solo del gradiente dello step precedente (g^{t-1}), ma si ha necessità di memorizzare anche il modificatore dei pesi allo step precedente (ΔW_{ij}^{t-1}). Sostituendo i ΔW_{ij}^t all'equazione 2 si ottiene la nuova regola di aggiornamento.

Di seguito sono riportati l'algoritmo in pseudocodice e l'implementazione in Matlab che permette di calcolare i modificatori ΔW_{ij}.

Algorithm 2: RPROP+

forall W_{ij} **do**
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) > 0$ **then**
    $\Delta_{ij}^t := min(\eta^+ \cdot \Delta_{ij}^t, \Delta_{max})$;
    $\Delta W_{ij}^t := -sign(g_{ij}^t) \cdot \Delta_{ij}$;
  end
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) < 0$ **then**
    $\Delta_{ij}^t := max(\eta^- \cdot \Delta_{ij}^t, \Delta_{min})$;
    $\Delta W_{ij}^t := -\Delta W_{ij}^{(t-1)}$;
    $g_{ij}^t := 0$;
  end
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) = 0$ **then**
    $\Delta W_{ij}^t := -sign(g_{ij}^t) \cdot \Delta_{ij}$;
  end
  $W_{ij}^{(t+1)} := W_{ij}^t + \Delta W_{ij}^t$;
end

RProp+

```
case 'rprop+'
    modificatoreW=...
    (-sign(gradiente.W{i}).*Delta.W{i}).*(gg>=0)...
    -oldMod.W{i}.*(gg<0);

    modificatoreB=...
    (-sign(gradiente.B{i}).*Delta.B{i}).*(ggB>=0)...
    -oldMod.B{i}.*(ggB<0);

    oldMod.W{i}=modificatoreW;
    oldMod.B{i}=modificatoreB;

    %Derivate a 0 per influenzare la prossima iterazione
    gradiente.W{i}=gradiente.W{i}.*(gg>=0);
    gradiente.B{i}=gradiente.B{i}.*(ggB>=0);
```

4.2.3 iRPROP+

Il cambio di segno tra il prodotto delle derivate parziali significa che l'algoritmo ha superato un minimo locale. Quando viene superato un minimo, non viene specificato se le modifiche dei pesi W_{ij} hanno causato un incremento o decremento dell'errore. Quindi, la versione *iRProp+* rivisita l'*RPROP+* aggiungendo che i pesi saranno effettivamente propagati all'indietro solo se hanno causato un **aumento dell'errore**.

$$\Delta W_{ij}^t := \left\{ \begin{array}{ll} -sign(g^t) \cdot \Delta_{ij} & if(g_{ij}^t \cdot g_{ij}^{t-1} \geq 0) \\ -\Delta W_{ij}^{t-1}; \textbf{ and } g_{ij}^t = 0 & if(g_{ij}^t \cdot g_{ij}^{t-1} < 0) \wedge (E > E\_old) \\ 0 & altrimenti \end{array} \right\} \quad (5)$$

Dunque, viene combinata un informazione "locale" (il segno del prodotto delle derivate parziali) con un informazione "globale" (l'errore della rete). In questo modo, si decide per ogni peso se ritornare al punto precedente solo se effettivamente c'è stato un peggioramento della condizione.
In questo modo, quando si effettua uno step troppo grande (superando il minimo locale) si considera l'errore e lo si confronta con l'errore allo step precedente.

- Se l'errore nella nuova posizione è effettivamente peggiore di quello precedente, allora ciò che accade è la situazione analoga all'RPROP+, ovvero si *backpropaga* il peso all'indietro e si considera la derivata parziale uguale a 0.

- Invece, se superando il minimo locale l'errore è effettivamente minore, non si vuole nè tornare al punto precedente (perché ci si trova in una situazione migliorativa), nè aumentare ancora lo *step-size* (che è stato già dimezzato). Quello che si vuole fare è solamente "fermarsi". Ovvero, si considera la derivata parziale uguale a 0 in modo tale che lo *step size* all'iterazione successiva non venga modificato ulteriormente (è già stato dimezzato perché è stato superato il minimo locale). A questo punto, ci si sposterà verso il minimo locale raggiungendolo dal lato "crescente" in cui la derivata è maggiore di zero (a differenza dell'RPROP+ che lo raggiungeva solo dal lato "decrescente").

Algorithm 3: iRPROP+

forall W_{ij} **do**
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) > 0$ **then**
    $\Delta_{ij}^t := min(\eta^+ \cdot \Delta_{ij}^t, \Delta_{max})$;
    $\Delta W_{ij}^t := -sign(g_{ij}^t) \cdot \Delta_{ij}$;
  end
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) < 0$ **then**
    $\Delta_{ij}^t := max(\eta^- \cdot \Delta_{ij}^t, \Delta_{min})$;
    if $E^{(t)} > E^{(t-1)}$ **then**
      $\Delta W_{ij}^t := -\Delta W_{ij}^{(t-1)}$;
    end
    $g_{ij}^t := 0$;
  end
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) = 0$ **then**
    $\Delta W_{ij}^t := -sign(g_{ij}^t) \cdot \Delta_{ij}$;
  end
  $W_{ij}^{(t+1)} := W_{ij}^t + \Delta W_{ij}^t$;
end

iRProp+

```
case 'irprop+'
    E=err(epoch);
    oldErr=err(epoch-1);

    modificatoreW=...
```

```
    (-sign(gradiente.W{i}).*Delta.W{i}).*(gg>=0)...
    -oldMod.W{i}.*(gg<0)*(E>oldErr);

    modificatoreB=...
    (-sign(gradiente.B{i}).*Delta.B{i}).*(ggB>=0)...
    -oldMod.B{i}.*(ggB<0)*(E>oldErr);

    oldMod.W{i}=modificatoreW;
    oldMod.B{i}=modificatoreB;

    gradiente.W{i}=gradiente.W{i}.*(gg>=0);
    gradiente.B{i}=gradiente.B{i}.*(ggB>=0);
```

4.2.4 iRProp-

Nel momento in cui si supera un minimo locale, l'algoritmo *iRProp-* dimezza effettivamente lo *step-size* (perché moltiplicato per $\eta^- = 0.5$) ma non si modifica il peso corrispondente. Ciò implica che allo step successivo il peso sarà modificato usando lo *step-size* dimezzato.

Per fare ciò, si *setta* la derivata parziale uguale a 0.

Algorithm 4: iRPROP-

forall W_{ij} **do**
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) > 0$ **then**
    $\Delta_{ij}^t := min(\eta^+ \cdot \Delta_{ij}^t, \Delta_{max})$;
  end
  if $(g_{ij}^t \cdot g_{ij}^{t-1}) < 0$ **then**
    $\Delta_{ij}^t := max(\eta^- \cdot \Delta_{ij}^t, \Delta_{min})$;
    $g_{ij}^t := 0$;
  end
  $W_{ij}^{(t+1)} := W_{ij}^t - sign(g_{ij}^t) \cdot \Delta_{ij}$;
end

iRProp–

```
case 'irprop-'
    gradiente.W{i}=gradiente.W{i}.*(gg>=0);
    gradiente.B{i}=gradiente.B{i}.*(ggB>=0);

    modificatoreW=-sign(gradiente.W{i}).*Delta.W{i};
    modificatoreB=-sign(gradiente.B{i}).*Delta.B{i};
```

Impostando la derivata parziale uguale a 0, $sign(g_{ij}^t)$ sarà uguale a 0 e quindi $W_{ij}^{(t+1)} := W_{ij}^t$. Inoltre, allo step successivo, il peso prodotto tra le derivate sarà 0 e quindi W_{ij} sarà aggiornato usando lo *step-size* dimezzato.

5 Il processo di learning

Arrivati a questo punto, sono state definite tutte le funzioni che permettono di implementare il processo di learning. In particolare è stato definito:

- Come estrarre il *dataset* ricavando *training set, validation set* e *test set*
- Come costruire una rete (***newNet.m***)
- Come simulare il comportamento di una rete (***simNet.m/forwardStep.m***)
- Come calcolare il gradiente con *back-propagation* (***backPropagation.m***)
- Come aggiornare i parametri tramite la RProp e le sue varianti (***RPROP.m***)

Inoltre, sono state definite le funzioni apposite per calcolare l'errore, le funzioni di attivazione e le rispettive derivate.

Il processo di learning è un processo iterativo che consiste nel modificare volta per volta i parametri al fine di minimizzare la funzione di errore. Ogni iterazione corrisponde ad una **epoca**.

Una singola epoca del processo di learning consta nei seguenti punti.

1. Si calcola il gradiente tramite la *back propagation*
2. Si aggiornano i parametri tramite la variante di RProp scelta
3. Si valuta la rete e si calcola l'errore sul *training set* e sul *validation set*
4. Si conserva la rete che ha errore minimo sul validation set.
5. Ripetere i punti 1...4 finché non si verifica una condizione di uscita (banalmente, effettuare 50 epoche).

Ad ogni epoca sarà generata una rete diversa con un errore diverso. Alla fine di tutte le epoche (o al verificarsi di una condizione di uscita), sarà restituita in output la rete con minimo errore sul validation set, ovvero quella che generalizza meglio.

Il processo di learning è stato implementato in Matlab nel modo seguente:

learningPhase.m

```
1 ...
2 for epoch=1:NumEpocs
3     %% AGGIORNAMENTO PARAMETRI
4     gradiente=backPropagation(net,x,t,funzErr);
5
6     [net,Delta,oldMod,gradiente]=RPROP(net,method,eta,eta_p,
          eta_n,gradiente,oldGrad,Delta,oldMod,errTrain,epoch);
7
8     oldGrad=gradiente;
```

```
9       %% VALUTAZIONE RETE (errore e accuracy)
10      y=simNet(net,x);
11      y_val=simNet(net,x_val);
12
13      errTrain(epoch)=funzErr(y,t)/10000;
14      errVal(epoch)=funzErr(y_val,t_val)/10000;
15
16      accTrain(epoch)=accuracy(y,t);
17      accVal(epoch)=accuracy(y_val,t_val);
18
19      %% CALCOLO RETE CON ERRORE MINIMO
20      if errVal(epoch)< min_err
21          min_err=errVal(epoch);
22          netScelta=net;
23      end
24  end
```

Alcune varianti dell'RProp (come RProp+) effettuano una modifica del gradiente. Per questo motivo, la funzione RProp ritorna anche il valore del gradiente. Inoltre, per calcolare i valori dei Δ_{ij} si ha bisogno anche del valore del gradiente dell'epoca precedente, che sarà memorizzato in *oldGrad*.

La funzione *accuracy* applica la seguente **regola di decisione**:

$$P(C_k|x) > P(C_j|x) \forall j \neq k \Rightarrow x \in C_k$$

Ovvero, x apparterrà alla classe che ha probabilità maggiore.

Al termine del processo di learning, sarà restituita la rete neurale che **generalizza meglio**, vale a dire la rete che ha minor errore sul *validation set*.

Part III
RISULTATI

In questa parte saranno comparate le quattro varianti dell'RProp (RProp-,RProp+,iRProp+,iRProp-) in un problema di classificazione sul *dataset* ***mnist***.

Nella sezione 6 saranno effettuate alcune considerazioni sugli iperparametri di default della RProp, ovvero $\eta^+ = 1.2$ ed $\eta^- = 0.5$ sulla base dei risultati ottenuti nella valutazione.

Tali considerazioni porteranno ad effettuare una scelta più formale degli iperparametri della rete e della RProp. Ciò è descritto nella sezione 7.

Una volta ottenuti gli iperparametri migliori, nella sezione 8 saranno nuovamente comparate le quattro varianti della RProp in un problema di classificazione sul *dataset* ***mnist***.

6 Prime considerazioni

In questa sezione saranno valutate le quattro varianti dell'RProp implementate. Sarà considerato un modello di rete multistrato *feed-forward*. Durante ogni valutazione saranno considerati gli stessi iperparametri (della rete e delle RProp), vale a dire:

- M=150 (un singolo strato interno con 150 nodi)
- $f = sigmoide$ (funzione attivazione strati interni)
- $g = identità$ (funzione attivazione strato output)
- $\eta = 0.0005$ (nella prima epoca sarà effettuata una discesa del gradiente standard)
- $\eta^+ = 1.2$
- $\eta^- = 0.5$
- $\Delta_0 = 0.0125$ (valore di inizializzazione dei vari Δ_{ij})
- $\Delta_{max} = 50$
- $\Delta_{min} = 0$

I valori degli iperparametri dell'RPROP (ovvero $\eta^+ = 1.2$, $\eta^- = 0.5$, $\Delta_0 = 0.0125$,$\Delta_{max} = 50$,$\Delta_{min} = 0$) sono stati ricavati dall'articolo *Empirical evaluation of the improved Rprop learning algorithms*.

Saranno effettuate 100 prove indipendenti per ogni variante. Saranno considerate, dunque, le valutazioni medie delle 100 prove.

I valori utilizzati per la valutazione sono i seguenti: **accuracy sul test set**, **numero di epoche necessarie per individuare la rete migliore**, **errore sul test set**. Per ogni parametro saranno considerati i valori medi delle 100 prove. La percentuale di errore è definita come $CrossEntropySoftMax(y,t)/10000$.
Sarà inoltre considerata la deviazione standard che permette di definire il range sul quale i valori si distribuiscono nel 95% dei casi.

La seguente tabella riporta i risultati ottenuti.

	Accuracy Test	Errore Test	Convergenza
RProp-	0.9308 ± 0.0050	0.0623 ± 0.0050	23.2±1.4
RProp+	0.9277 ± 0.0059	0.0666 ± 0.0048	26 ± 1.6
iRProp+	0.9355 ± 0.0043	0.0580 ± 0.0056	23.2 ± 1.3
iRProp-	0.9352 ± 0.0053	0.0581 ± 0.0046	23 ± 1

Tabella 1: Valutazioni con iperparametri standard

Si nota come le quattro varianti abbiano risultati pressoché simili. Inoltre esse hanno un andamento simile come si può notare dai grafici in fiugra 3.

Attraverso la configurazione di default, tutte le quattro varianti hanno rilevato delle ottime prestazioni. In particolare, le varianti con le prestazioni migliori sul problema di classificazione mnist sono risultate essere le varianti *iRProp+* ed *iRProp-*. Tali varianti hanno prestazioni simili tra loro e migliori rispetto alle altre due; però, c'è da considerare che la variante *iRProp-*, rispetto alla *iRProp+* occupa meno spazio di memoria in quanto non ha necessità di memorizzare il modificatore dei pesi dell'epoca precedente.
La variante *RProp-* performa meglio della variante *RProp+*, che risulta essere la peggiore tra le quattro[5]

La figura 3 mostra l'andamento delle quattro varianti rispetto all'errore sul validation set. Si nota come esse abbiano un andamento pressoché simile. Per intuire il comportamento delle quattro varianti è significante fare uno zoom sulle aree di interesse. In particolare, è interessante considerare il loro andamento all'avvicinarsi all'epoca di convergenza, ovvero dove raggiungono l'errore minimo.

La figura 4 mostra che anche graficamente viene rispettato ciò che riportato nella tabella 1, ovvero che le varianti *improved* (iRProp+ ed iRProp-) hanno prestazioni migliori rispetto alle altre due; inoltre, la peggiore risulta essere la variante RProp+.

[5]La peggiore con questi determinati iperparametri e per questo specifico problema.

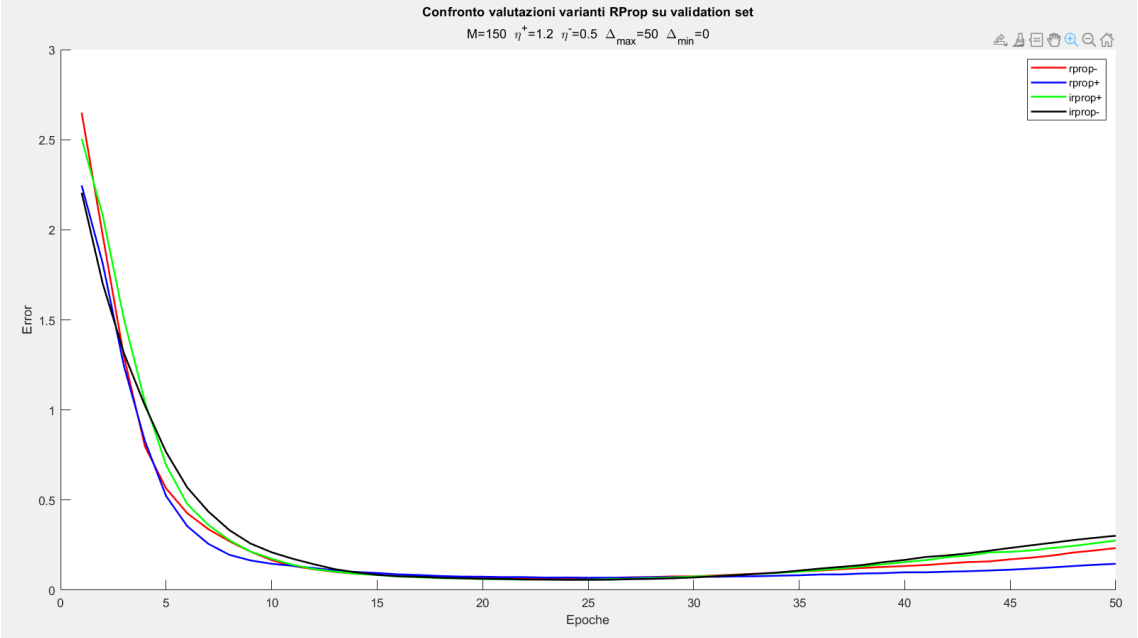


Figure 3: Confronto grafico tra le quattro varianti sull'errore sul validation set

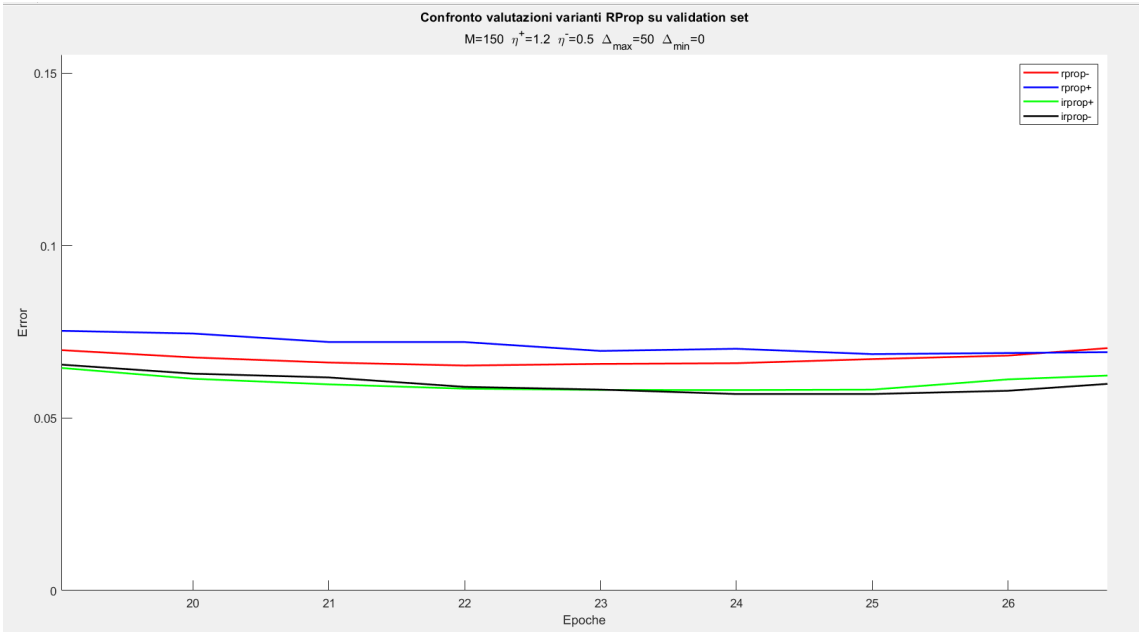


Figure 4: Zoom sulla zona di interesse, le varianti RProp-, RProp+, iRProp+, iRProp- convergono rispettivamente alle epoche 22 25 24 24

6.1 Nota sugli iperparametri

L'algoritmo RProp è noto per essere poco dipendente rispetto ai suoi iperparametri η^+ ed η^-. Infatti, pur mantenendo la configurazione standard degli iperparametri, si

ottengono prestazioni discrete. Tuttavia, le prestazioni sono "limitate" in quanto avviene un cambiamento troppo brusco dei parametri (aumento del 20% e diminuzione del 50%) che non rendono possibile la convergenza ad un minimo locale.

A questo punto, quello che viene da chiedersi è: ***Qual è la configurazione di iperparametri che consente prestazioni migliori?*** E soprattutto ***quanto conviene cambiare iperparametri?***

A tal proposito, sarà effettuata una scelta degli iperparametri più formale, diversa dall'inizializzazione standard. Tale scelta includerà i seguenti iperparametri: M, η^+ ed η^-, ovvero il numero di nodi dello strato interno della rete e gli iperparametri dell'RProp.

7 Scelta del modello

La scelta del modello consiste nello scegliere i giusti iperparametri per risolvere al meglio il problema. Considerate le valutazioni effettuate nella sezione 6, la scelta del modello è stata considerata sulla variante RProp+ (variante sulla quale si aveva una valutazione peggiore).

Gli iperparametri da considerare sono: M (numero nodi interni), η^+, η^-. A tal proposito, l'idea implementativa ha visto una **strategia a griglia**. Gli intervalli degli iperparametri sono i seguenti:

- $M = [150, 170, 200, 220, 250]$
- $\eta^+ = [1.2, 1.15, 1.1, 1.07, 1.05]$
- $\eta^- = [0.5, 0.4, 0.3, 0.25, 0.2]$

Sono stati scelti 5 possibili valori per ogni iperparametro. Ciò significa che si avranno $5 \times 5 \times 5 = 125$ configurazioni diverse.

Le possibili configurazioni sono memorizzate in una matrice del seguente tipo:

M	η^+	η^-	Accuracy test	Errore test	convergenza
150	1.2	0.5	...	...	...
...	...	...	...	...	...
250	1.05	0.2	...	...	...

Tabella 2: Esempio di matrice per memorizzare i risultati delle valutazioni

Dunque, ogni singola valutazione sarà memorizzata in una matrice di dimensione 125×6. Ottenute le possibili configurazioni, è necessario effettuare una valutazione **significativa** per ogni configurazione. Si è scelto di implementare la strategia **Hold-Out**. Vale a dire che sono state considerate diverse suddivisioni per *training, validation* e *test* per ogni possibile configurazione. In particolare, sono stati effettuate

10 diverse suddivisioni.

Ciò vuol dire che per ogni configurazione (M, η^+, η^-) si hanno a disposizione 10 valutazioni $v_1, ..., v_{10}$ per un totale di $125 \times 10 = 1250$ computazioni.

Ogni valutazione ha valori relativi ad: accuracy del test set, errore sul test set, convergenza (numero epoche necessarie per trovare i parametri migliori).

Ottenute le 10 valutazioni per ogni configurazione, è possibile calcolare **media** e **deviazione standard** per ogni configurazione dei parametri della valutazione. Per motivi logistici, saranno mostrate in tabella solo le 10 più performanti sulla base di errore medio minimo, considerata anche la deviazione standard.

M	η^+	η^-	Accuracy test	Errore test	convergenza
250	1.05	0.4	0.9434 ± 0.0043	0.0497 ± 0.0037	44 ± 3.6
250	1.05	0.5	0.9420 ± 0.0047	0.0505 ± 0.0051	41 ± 2
220	1.05	0.5	0.9407 ± 0.0051	0.0505 ± 0.0045	40 ± 3.3
200	1.05	0.5	0.9399 ± 0.0059	0.0508 ± 0.0052	41 ± 2.4
220	1.05	0.3	0.9412 ± 0.0056	0.0510 ± 0.0048	47 ± 3.2
220	1.05	0.4	0.9416 ± 0.0055	0.0515 ± 0.0050	44 ± 2.7
220	1.07	0.4	0.9420 ± 0.0043	0.0516 ± 0.0043	38 ± 2.2

Tabella 3: Migliori iperparametri

È facile notare che modificando gli iperparametri si possono ottenere prestazioni migliori, migliorando circa del 2% l'accuracy. Le prestazioni migliori si ottengono con gli iperparametri $M = 250, \eta^+ = 1.05, \eta^- = 0.4$.

Tali iperparametri mostrano che è effettivamente più performante effettuare modifiche leggere dei parametri riducendo i vari step size. Infatti, rispetto alla configurazione standard, l'iperparametro η^+ è diminuito drasticamente (da 1.2 a 1.05), mentre l'iperparametro η^- è diminuito "solamente" da 0.5 a 0.4.

Considerando lo spazio dei parametri, quello che avviene è un movimento più lento verso un minimo locale. Tale andamento rappresenta una caratteristica desiderata se si suppone una buona inizializzazione randomica dei pesi W_{ij}

In Matlab Per poter effettuare una scelta degli iperparametri tramite l'approccio a griglia si è implementato il seguente flusso:

script sceltaIperparametri

```
1
2  ...
3
4  %% INTERVALLI CONSIDERATI DEGLI IPERPARAMETRI
5  VM=[150,170,200,220,250];
6  Veta_p=[1.2, 1.15, 1.1, 1.07, 1.05];
```

```
Veta_n=[0.5, 0.4, 0.3, 0.25, 0.2];

for k=1:10
    %% SHUFFLE
        ...
    %% SUDDIVISIONE DATASET IN TRAINING, VALIDATION E TEST
        ...
    for ind_M=1:length(VM)
        M=VM(ind_M);
        for ind_etap=1:length(Veta_p)
            eta_p=Veta_p(ind_etap);
            for ind_etan=1:length(Veta_n)
                net=newNet(...,M,...);
                netScelta=learningPhase(...,eta_p,eta_n);
                 %% VALUTAZIONE RETE
                    ...
                 matriceRisultatiTotale{k}(riga,:)=[M,eta_p,
                    eta_n, accTest, erroreTest,convergenza];
            end
        end
    end
end
```

La variabile *matriceRisultatiTotale* rappresenta le 10 valutazioni procurate per ognuna delle 125 configurazioni. Le valutazioni sono state memorizzate in un *cell array* di dimensione 10. Ogni cella è una matrice 125×6 (come mostrato in tabella 2) in cui ogni riga rappresenta una configurazione dei parametri (M, η^+, η^-). Per ogni configurazione è memorizzata una valutazione.

Sulla base di questa variabile, sarà creata una matrice 125×9 che conterrà i valori medi delle valutazioni per ogni configurazione, comprensivi di deviazione standard. L'ordinamento di tale matrice rispetto all'errore medio ha permesso la ricerca degli iperparametri migliori.

8 Valutazione modello

Nella sezione 7 si è mostrato come è stata effettuata la scelta del modello migliore. Si ricorda che tale scelta è stata effettuata sulle valutazioni relative alla sola variante *RProp+*.

A questo punto, scelto il modello con $M = 250, \eta^+ = 1.05, \eta^- = 0.4$, saranno valutate le prestazioni sulle diverse varianti. Così come nella sezione 6, saranno effettuate 100 prove indipendenti per ogni variante. Saranno considerate, dunque, le valutazioni medie delle 100 prove.
I valori utilizzati per la valutazione sono i seguenti: **accuracy sul test set**, **numero di epoche necessarie per individuare la rete migliore**, **errore sul test set**. Per ogni parametro saranno considerati i valori medi delle 100 prove. La percentuale

di errore è definita come $CrossEntropySoftMax(y,t)/10000$.
Sarà inoltre considerata la deviazione standard che permette di definire il range sul quale i valori si distribuiscono nel 95% dei casi.

La seguente tabella riporta i risultati ottenuti.

	Accuracy Test	Errore Test	Convergenza
RProp-	0.9430 ± 0.0059	0.0517 ± 0.0059	43 ± 2.5
RProp+	0.9433 ± 0.0053	0.0519 ± 0.0053	44 ± 3
iRProp+	0.9444 ± 0.0047	0.0505 ± 0.0039	42 ± 2
iRProp-	0.9447 ± 0.0045	0.0507 ± 0.0049	43 ± 2

Tabella 4: Valutazione delle quattro varianti utilizzando gli iperparametri migliori

Utilizzando quelli che sono stati definiti gli iperparametri migliori, tutte le quattro varianti hanno avuto dei miglioramenti di prestazione, sia per quanto riguarda l'accuracy, sia per quanto riguarda l'errore. Ovviamente, il maggior aumento di performance l'ha avuto la variante *RProp+*, sulla quale si è basata la scelta degli iperparametri. Tali miglioramenti sono ottenuti a discapito della velocità di convergenza, che è quasi raddoppiata rispetto alla casistica tramite iperparametri standard.

Con questa configurazione, tutte le quattro varianti hanno un andamento molto simile, quasi interscambiabile. Tuttavia, la variante con le performance migliori rimane una variante del tipo *improved*, ovvero la *iRProp+*, così come accadeva per la configurazione standard degli iperparametri.

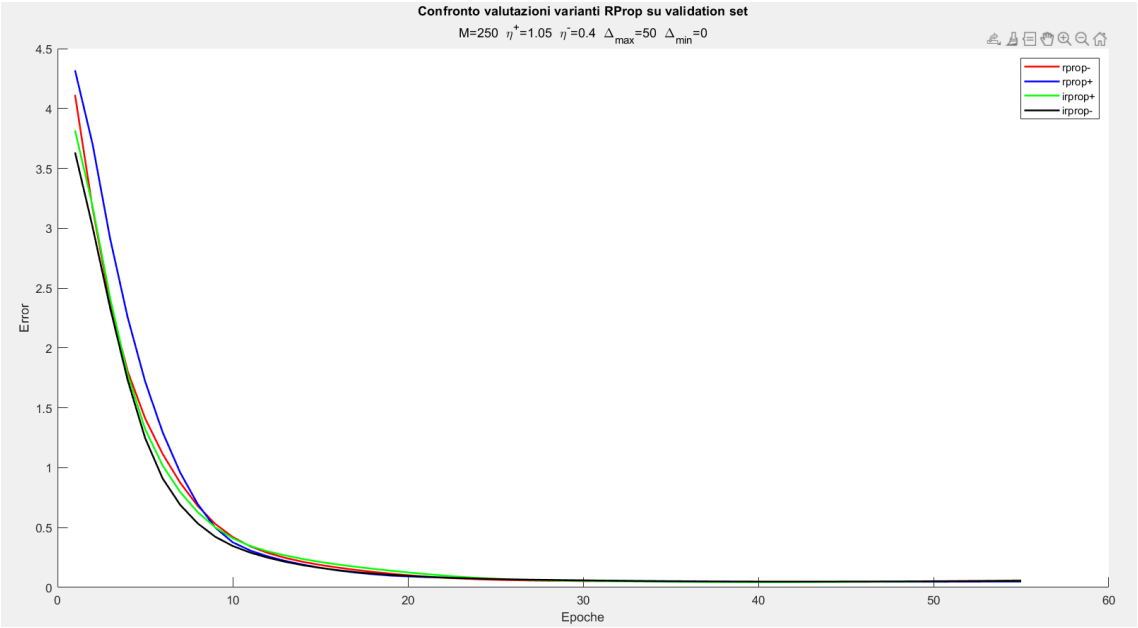


Figure 5: Confronto grafico tra le quattro varianti sull'errore sul validation set

8.1 Nota su tempi computazionali

Tutte le valutazioni e considerazioni effettuate, sono state analizzate considerando il dataset *mnist/t10k-images-idx3-ubyte*, contenente 10'000 esempi di input. L'obiettivo di questa sezione è **confrontare i tempi computazionali e le prestazioni delle due configurazioni sul dataset *mnist* completo**, contenente 60'000 esempi di input. In questo modo, si ottengono valutazioni sui tempi computazionali più legate alla realtà.

I tempi saranno valutati in base ai valori medi di 10 computazioni per ogni configurazione. La variante RProp utilizzata per acquisire i valori è una variante *improved*, in particolare la variante *iRProp-*.

Per avere un quadro generale completo, sarà considerato come criterio di stop l'effettuare 100 epoche.

$M = 150; \eta^+ = 1.2; \eta^- = 0.5$		$M = 250; \eta^+ = 1.05; \eta^- = 0.4$	
Tempo	**Convergenza**	**Tempo**	**Convergenza**
$30.0s$	34	$47.4s$	75

Tabella 5: Tempi computazionali sul dataset mnist completo con variante iRprop-

Il tempo computazionale è stato preso considerando il verificarsi di 100 epoche. Si è scelto di riportare anche l'epoca di convergenza per avere un immagine più completa di quello che succede utilizzando configurazioni di iperparametri diversi.

Utilizzando un adeguato criterio di stop, i tempi computazionali avranno un distacco ancora più ampio.

N.B. Ovviamente i tempi computazionali sul dataset ridotto a 10'000 immagini sono minori. In particolare, per effettuare 100 epoche:

- Per la configurazione $M = 150; \eta^+ = 1.2; \eta^- = 0.5$ il tempo medio è circa $5.5s$
- Mentre per la configurazione $M = 250; \eta^+ = 1.05; \eta^- = 0.4$ il tempo medio è circa $8.5s$

Part IV
CONCLUSIONI

L'RProp è un ottimo algoritmo di aggiornamento che raggiunge ottime performance limitando la dipendenza dagli iperparametri. Nel seguente documento sono state valutate le quattro varianti utilizzando due configurazioni degli iperparametri diverse.

- **Considerando una configurazione standard degli iperparametri** (ovvero $M = 150, \eta^+ = 1.2, \eta^- = 0.5$), tutte le varianti hanno avuto ottimi risultati. Inoltre, quello che salta all'occhio utilizzando questa configurazione è la velocità con la quale si trova la rete che generalizza meglio, con una non eccessiva deviazione standard (circa 24 epoche per tutte le varianti).

- **Utilizzando una configurazione tramite gli iperparametri migliori**, si ha avuto un aumento delle prestazioni per tutte le varianti, in particolare per la variante RProp+.

  Tuttavia, si può notare come la convergenza avviene dopo un numero molto alto di epoche (circa 43 per tutte le varianti, quasi il doppio rispetto alla configurazione standard).

Esaminando attentamente tali considerazioni, la domanda da porsi è la seguente: "*in un problema di classificazione sul dataset mnist, vale la pena effettuare una scelta degli iperparametri più formale?*" La risposta ovviamente è **dipende**:

- Utilizzando la configurazione di default si beneficia della velocità di convergenza a discapito di una piccola diminuzione delle performance.
- Utilizzando una configurazione più ricercata, si ha la libera scelta di utilizzare una qualsiasi delle varianti ottenendo risultati pressoché simili. Inoltre, si ottiene un aumento delle performance a discapito, però, di una lentezza della convergenza.

Quindi, se l'obiettivo principale del problema è avere la percentuale di accuratezza maggiore, è una buona idea utilizzare la configurazione degli iperparametri migliori, sacrificando del tempo computazionale a vantaggio delle prestazioni.

Se, invece, l'obiettivo principale del problema è avere delle prestazioni alte ed una velocità di convergenza, è una buona idea utilizzare la configurazione standard degli iperparametri.

Tuttavia, qualsiasi sia l'obiettivo del problema, si è mostrato come le varianti *improved* (iRProp+ ed iRProp-) risultino le più performanti sia in termini di accuratezza, che in termini di errore minimo in entrambe le configurazioni.