PROGETTO DI RETI NEURALI

Sviluppo di una rete neurale Feed Forward - fully connected
per il riconoscimento di cifre digitali del dataset MNIST

Isabella Tieri N97000360
Giuseppina Russo N97000361
Davide Somma N97000367

Sommario

Introduzione

In questo elaborato, la rete neurale feed-forward sarà oggetto di analisi. In particolare saranno trattati gli elementi che la costituiscono e le relative implementazioni in MATLAB al fine di soddisfare la seguente richiesta:

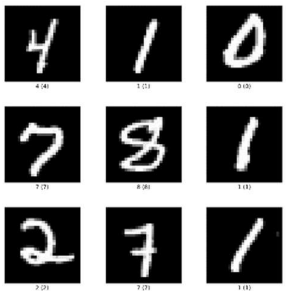
> *Si consideri come input le immagini raw del dataset MNIST. Si ha, allora, un problema di classificazione a C classi, con C=10. Si estragga opportunamente un dataset globale di N coppie, e lo si divida opportunamente in training, validation e test set (ad esempio, 5000 per il training set, 2500 per il validation set, 2500 per il test set). Si fissi la resilient backpropagation (RProp) come algoritmo di aggiornamento dei pesi (aggiornamento batch). Si studi l'apprendimento di una rete neurale (ad esempio epoche necessarie per l'apprendimento, andamento dell'errore su training e validation set, accuratezza sul test) facendo variare il numero di strati interni da 1 a 5 confrontando il caso in cui si utilizza come funzione di attivazione dei nodi la sigmoide con quello in cui si usa come funzione di attivazione dei nodi la ReLu (max(0,a)). Provare diverse scelte del numero dei nodi per gli strati interni. Se è necessario, per questioni di tempi computazionali e spazio in memoria, si possono ridurre (ad esempio dimezzarle) le dimensioni delle immagini raw del dataset mnist.*

Nella prima sezione si esaminerà il dataset menzionato nella richiesta e verrà introdotto in maniera formale il problema della classificazione.
Le parti susseguenti saranno dedicate esclusivamente alla trattazione e all'implementazione della rete neurale e la sua fase di apprendimento.
Il segmento conclusivo dell'elaborato è destinato alla valutazione e allo studio della rete al variare di alcuni dei suoi iperparametri.

I capitolo

Dataset MNIST

La base di dati MNIST (*modified National Institute of Standards and Technology database*) 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 dell'apprendimento automatico. La base di dati è stata creata rimescolando le immagini presenti nell'insieme di dati del NIST. La base di dati MNIST contiene 60 000 immagini di addestramento e 10 000 immagini di test; metà dell'insieme di addestramento e metà dell'insieme di test sono state prelevate dall'insieme di addestramento del NIST, mentre le altre metà sono state ottenute dall'insieme di test del NIST stesso.



1.1 Problema di classificazione

Il problema che la rete dovrà ritrovarsi a risolvere è un problema definito di "classificazione" multi-classe, ovvero date 10 classi, corrispondenti alle cifre che vanno da 0 a 9, sarà possibile determinare a quale classe appartiene una certa immagine prelevata dal dataset precedentemente descritto e che corrisponde appunto ad una cifra digitale scritta a mano.

Formalmente: abbiamo la rete feed forward fully-connected multi-strato che prende in input dati (immagini) etichettati (valore associato all'immagine) e produrrà in output una probabilità di appartenenza di ogni immagine ad una delle 10 classi, in modo tale da poter valutare l'errore di classificazione commesso dalla rete nel predire la classe di appartenenza e la sua accuratezza, ovvero la sua specificità (banalmente vista come il complemento dell'errore di predizione).

Il dataset MNIST quindi altro non è che un insieme $DS = \{(x^n, t^n)\}_{n=1}^{N}$ $tale\ che\ x^n \in R^d\ e\ t^n \in \{0,1\}^C,\ con\ C = 10\ e\ N \equiv numero\ totale\ di\ immagini.$

L'output della rete sarà dunque una probabilità tale che:

- ✓ $\forall k\ P(C_k|x)$, ovvero per ogni classe k valuto la probabilità condizionata che l'immagine x sia appartenente alla classe C_k
- ✓ Definisco una funzione di decisione h: $P(C_h|x) > P(C_k|x)$, $\forall h \neq k$

In questo modo sono in grado di dividere il dominio in più regioni distinte che mi identificano le varie classi e l'obiettivo sarà quindi quello di stabilire se un dato x appartenga ad una regione piuttosto che ad un'altra adoperando una funzione discriminante del tipo:

- ✓ Data la funzione discriminante per ogni i-esima classe $\varphi(i) = P(x|C_i)P(C_i)$, che indica la probabilità che x appartenga a C_i data la probabilità della classe.
- ✓ Definisco la funzione di discriminazione

$$\begin{cases} C_1 & se\ \varphi_1(x) > \varphi_2(x)\ cioè\ x \in R_1 \\ C_2 & altrimenti, cioè\ x \in R_2 \end{cases}$$

che indica la classe di appartenenza del dato x, valutando le funzioni di discriminazione associate ad ogni classe.

1.2 Costruzione del dataset

Per l'implementazione della rete è stato scelto di dividere il dataset in tre insiemi:

- ✓ Training set da 5000 immagini, per addestrare la rete senza però farle "conoscere" troppi dati rischiando di provocare un overfitting.
- ✓ Validation set da 2500 immagini, per effettuare predizioni su immagini non tutte già viste nella fase di training.
- ✓ Test set da 2500 immagini, per testare la rete e valutare accuracy ed errore.

Le immagini all'interno del dataset sono state soggette però ad uno *shuffle* iniziale in modo tale da evitare l'occorrenza di una stessa immagine più di una volta. Di seguito verrà riportato il frammento di codice che si occupa di fare ciò che è stato precedentemente descritto.

```
%estraggo i dati dal DS
function [newNet, err, errVal, score] = main(params)
    files = ["t10k-images-idx3-ubyte" "t10k-labels-idx1-ubyte"];
    [X,T] = prepareDataset(files);

    %prelevo i dati in modo random
%   ind = randperm(size(X,2));
%   X = X(:,ind);
%   T = T(:,ind);

    %divido in train, test e validation set
    percentTrain = size(X,2)*params.percentTrain;
    percentValidation = size(X,2)*params.percentValidation;
    offset = percentTrain+percentValidation;

    XTrain = X(:,1:percentTrain); % 5000 per il train
    TTrain = T(:,1:percentTrain);

    XVal =X(:,percentTrain+1:offset); % 2500 per il validation
    TVal = T(:,percentTrain+1:offset);

    XTest = X(:,offset+1:end); %2500 per il test
    TTest = T(:,offset+1:end);

    %creo la rete
    net = createNet(params,size(XTrain,1),size(TTrain,1));
```

La funzione *prepareDataset* viene utilizzata per estrarre i dati dal dataset e codificare mediante codifica *one-hot* le labels associate ad ogni immagine. I parametri *percentTrain, percentValidation* hanno il solo scopo di trasformare sotto forma di percentuale il quantitativo di dati che si vogliono assegnare ai corrispettivi insiemi di Train e di Validation.

Nel capitolo successivo verrà illustrata la costruzione/creazione della rete (funzione *createNet*) e come avvengono i vari step di addestramento e aggiornamento della stessa, al fine di valutarne il comportamento.

Il capitolo

Costruzione della rete

In questo capitolo verrà descritto come è stata costruita la rete partendo dalle scelte implementative che hanno portato alla sua realizzazione. Si partirà quindi dall'introduzione delle due parti (*A* e *B*) e in ognuna ci si focalizzerà sui punti richiesti.

2.1 Parte A

La seguente è una delle due parti di cui l'intero progetto è composto. Per questa sezione sono state richieste due specifiche, ovvero:

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

2.1.1 Struttura della rete

Per poter implementare quanto suddetto, è stata sviluppata la funzione createLayers che permette di generare gli strati della rete con i parametri che le vengono passati in input.
La struttura utilizzata per memorizzare tutte le informazioni è fatta in questo modo:

```
layers = repmat(struct('W', zeros(0)), 1, nLayers);
```

Grazie alla funzione di matlab `repmat`, verrà creato una struct di array di dimensione 1xnLayers con il campo W inizializzato a 0 (servirà per memorizzare i pesi dei neuroni di ogni strato). Man mano questa struttura si arricchirà e vedremo di seguito come...

Per adesso, lo snippet della funzione che crea gli strati della rete è qui riportato:

```
function layers = createLayers(params,inputSize,outputSize)
%params: parametri dati dall'utente utili alla costruzione della rete
%inputSize: numero di variabili in ingresso (x1,...,xd)
%outputSize: numero nodi dello strato di output pari al numero di classi

    neurons = params.neurons;
    nLayers = params.nLayers;
    layers = repmat(struct('W', zeros(0)), 1, nLayers); %uno strato sarà una
struct con i suoi campi

    %inizializzo il parametro size per ogni strato per determinare le
    %dimensioni delle matrici dei pesi e del gradiente
    if isempty(neurons)
```

```
        layers(1).size = [inputSize,outputSize];
    else

        layers(1).size = [inputSize, neurons(1)]; %primo strato
        for i = 2:nLayers-1  %strati interni

            layers(i).size = [neurons(i-1), neurons(i)];
        end
        layers(nLayers).size = [neurons(nLayers-1), outputSize]; %ultimo strato

    end

    sigma=0.1;
    deltaZero=0.1;
    %imposto i parametri per ogni strato
    for i=1:nLayers

        if i<nLayers
            layers(i).act = params.act(i); %funzione di attivazione per ogni
strato
        end

        layers(i).W = randn(layers(i).size(2),layers(i).size(1))*sigma; %matrice
pesi
        layers(i).B = zeros(layers(i).size(2),1); %vettore di bias

        layers(i).gradient.W = zeros(layers(i).size(2),layers(i).size(1));
%matrice gradiente pesi
        layers(i).gradient.B = zeros(layers(i).size(2),1); %matrice gradiente
bias

        layers(i).D.W = ones(layers(i).size(2),layers(i).size(1))*deltaZero;
%matrice dei delta per la rprop
        layers(i).D.B = ones(size(layers(i).B)) * deltaZero;
    end

    layers(nLayers).act = "identity"; % funzione di attivazione per l'ultimo
strato
end
```

Come riportato dal primo if…end, la prima cosa da fare, dopo aver inizializzato i neuroni, il numero di strati e istanziati i layers, è quella di impostare la dimensione di ogni strato diversificandoli per strato di input, strati interni e strato di output.
Successivamente, si impostano tutti i parametri tramite un ciclo for che scorre su tutti gli strati: per ogni strato, abbiamo la funzione di attivazione ad esso relativa e tale viene estrapolata da params.act(i) con params passato per input alla funzione.
In particolare, un buon modo per poter inizializzare i pesi è prenderli casuali e che possano oscillare intorno allo zero. Motivo per il quale l'inizializzazione dei pesi W segue la sintassi seguente:

```
layers(i).W = randn(layers(i).size(2),layers(i).size(1))*sigma;
```

con sigma=0.1.

Gli step di aggiornamento Δ_{ij} (che serviranno per poter aggiornare i pesi con RPROP) vengono inizializzati in modo random moltiplicando per un deltaZero=0.1 come suggerisce il paper [1].

Viene inoltre espansa la struttura layers con i bias B, i gradienti e i delta (utilizzati per la RPROP) calcolati rispetto a W e B.
Il perché è stato scelto di porre come funzione di attivazione l'identità per l'ultimo strato verrà spiegato nel paragrafo successivo.

La funzione createLayers viene invocata all'interno di createNet come unica sua istruzione restituendo così la rete creata:

```
function net = createNet(params,inputSize,outputSize)
%creo la rete
%params: parametri della rete
%inputSize: numero di linee d'ingresso
%outputSize: numero di classi

    net.layers = createLayers(params,inputSize,outputSize); %inizializzo gli
strati della rete
end
```

Questa è la prima funzione che viene invocata all'interno del main prima di far partire l'addestramento sulla rete generata.

2.1.2 Forward Propagation

In un modello a più neuroni, quest'ultimi interagiscono tra loro nel modo seguente: supponiamo di avere d linee di ingresso $x_1, \ldots, x_d$ e n neuroni enumerati come d+1, d+2, … , d+ n. Se supponiamo che i neuroni sono collegati in modo da non avere cicli, allora esiste un ordine parziale tra loro. Per cui il calcolo dell'output di un neurone è, $\forall i \in [\, d+1, d+n\,]$:

1. $a = \sum_{J}^{d} w_{i,j} y_j + b_i$ — j scorre su tutti gli indici dei neuroni che inviano connessioni ad i; $w_{i,j}$ è il peso della connessione che parte dal neurone j arriva al neurone i
2. $y_i = f_i(a_i)$ — y_i è il valore d'uscita del neurone j quando j>d altrimenti $y_j = x_j$

La sequenza di computazione segue l'ordine parziale tramite una propagazione in avanti dell'input. Questa è proprio la fase di forward propagation.
Nello sviluppo di questo algoritmo all'interno del progetto, la funzione *forwardProp* viene invocata durante l'addestramento della rete come prima istruzione di ogni nuova iterazione che cicla sul numero di epoche: prende in ingresso gli strati della rete su cui la propagazione in avanti dell'input deve avvenire e l'input stesso, restituendo gli strati con i

valori aggiornati. Affinchè ciò avvenga, viene eseguito un for sul numero di strati interni (compreso quello di input) che, dopo aver inizializzato i pesi `W`, i bias `b` e la funzione di attivazione `act` dello strato i-esimo, calcola l'input `a` tramite la formula (1). Le funzioni di attivazione di questi strati non possono essere di qualsiasi natura: basta che siano derivabili e non polinomiali (e non lineari). Si sono scelte, allora, queste funzioni:

1. Sigmoide: funzione sigmoide.
$$P(t) = \frac{1}{1 + e^{-t}}$$
2. Relu: funzione Rectified Linear Units
$$Relu(x) = \max(0, x)$$
3. Hv: funzione Heaviside
$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Dopodichè a seconda del valore che `act` assume in ogni strato (Codice # - riga 18), viene calcolato l'output dello strato i-esimo applicando ad alla funzione di attivazione i-esima (formula (2)). L'ultimo strato fa eccezione allo `switch` poiché la funzione di attivazione dello strato di output deve essere la funzione identità (`identity`) poiché non derivabile.

Di seguito, è riportato lo snippet di quanto appena descritto:

```
function layers = forwardProp(layers,input)
%forward propagation per aggiornare i valori
%layers: struct contenente i vari strati della rete
%input: input del primo strato

    z = input; %l'input del primo strato coincide con l'output

    for i=1:length(layers)

       W = layers(i).W; %inizializzo i pesi
       B = layers(i).B; %inizializzo i bias
       act = layers(i).act; %inizializzo le funzioni di attivazione

       a = W*z+B; % calcolo della a per poi calcolare z = f(a)

       if i < length(layers)

           switch (act)

               case "sigmoide"
                   z = sigmoide(a);

               case "relu"
                   z = relu(a);

               case "hv"
                   z = hv(a);
           end
       else
```

```
            z = identity(a); %per lo strato di output derivabile
        end

        layers(i).a = a;
        layers(i).z = z;
    end
end
```

2.1.3 BACK PROPAGATION

Sia R una rete fully-connected e $DS = \{(x^{(n)}, t^{(n)})\}_{n=1}^{N}$ con

$x^{(n)} \in \mathbb{R}^d$ e $t^{(n)} \in \{0,1\}^C$ (per problemi di classificazione) o con $t^{(n)} \in \mathbb{R}^d$ (per problemi di regressione), il dataset.

Dunque

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

è la funzione di errore con $\theta = \{W^1, b^1, W^2, b^2 \dots, W^p, b^p\}$ l'insieme dei parametri di una rete.

Quindi

$$\nabla E = [\frac{\partial E}{\partial \theta_1}, \frac{\partial E}{\partial \theta_2}, \dots, \frac{\partial E}{\partial \theta_p}]$$

è il gradiente della funzione di errore rispetto a θ .

Il calcolo del gradiente si basa sull'algoritmo di Back Propagation che consente di calcolare le derivate rispetto ai vari parametri in modo molto efficiente.

Dato un punto n qualsiasi del training set, si ha che

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

Fissando $\delta_i = \frac{\partial E^{(n)}}{\partial a_i}$ e ricordando che $\frac{\partial a_i}{\partial w_{ij}} = z_j$ abbiamo:

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

In particolare

$$\delta_i = \begin{cases} f'(a_i) \sum_h w_{hi}\delta_h, & se\ i\ nodo\ interno \\ f'(a_i)\dfrac{\partial E^{(n)}}{\partial y_i}, & se\ i\ nodo\ output \end{cases}$$

Con h indice dei nodi che ricevono connessione dal nodo i e f' derivata della funzione di attivazione del livello i-esimo.

Per determinare δ_i di nodi interni è necessario conoscere i delta dei nodi dello strato successivo. L'algoritmo procede quindi in maniera iterativa partendo dall'ultimo layer:

```
function layers = backProp(layers, gradErr,x)
%layers: array di struct di lunghezza NLayers
%gradErr: derivata della funzione di errore relativa all'output della rete

    for i=length(layers):-1:1
        […]
    end
end
```

Ad ogni passo si verifica se il layer considerato è interno o di output. In base allo scenario si calcola opportunamente δ_i .

```
      if i == length(layers)

           h = derivFunction(layers(i).act,layers(i).a);
           delta = h .* gradErr;
      else

           delta = (layers(i+1).W)' * delta;
           h = derivFunction(layers(i).act,layers(i).a);
           delta = delta .* h;

      end
```

Successivamente possiamo calcolare il gradiente, tenendo conto però anche del caso in cui j sia una variabile di ingresso e quindi $z_j = x_j$.

```
       if i==1

           layers(i).gradient.W = delta*(x)';
       else

           layers(i).gradient.W = delta*(layers(i-1).z)';
       end
       layers(i).gradient.B = sum(delta,2);
```

La backpropagation è stata quindi implementata in questo modo:

```
function layers = backProp(layers,gradErr,x)
%layers: array di struct di lunghezza NLayers
%gradErr: derivata della funzione di errore relativa all'output della rete
%x: input della rete

    for i=length(layers):-1:1

       if i == length(layers)

           h = derivFunction(layers(i).act,layers(i).a);
           delta = h .* gradErr;
       else

           delta = (layers(i+1).W)' * delta;
           h = derivFunction(layers(i).act,layers(i).a);
           delta = delta .* h;

       end

       if i==1
           layers(i).gradient.W = delta*(x)';
       else

           layers(i).gradient.W = delta*(layers(i-1).z)';
       end
       layers(i).gradient.B = sum(delta,2);

    end
end
```

2.1.4 FUNZIONE DI ERRORE

Al fine di calcolare δ_i dei nodi dello strato di output, è stato necessario calcolare la derivata della funzione di errore correntemente utilizzata dal modello.

Le funzioni di errore considerate in questo elaborato sono la Sum of Squares e la Cross Entropy.

La prima, è definita in questo modo:

$$E^{(n)} = \frac{1}{2}\sum_k (y_k^n - t_k^n)^2$$

e implementata come segue;

```
function err = sumOfSquares(y,t)
%calcolo dell'errore tramite sum-of-squares
%y: output della rete
%t: label del DS

    err = 0.5*sum(sum((y-t).^2));
end
```

La Cross Entropy invece è determinata in questo modo

$$E^{(n)} = \sum_{k=1}^{C} t_k^n \log_e y_k^n$$

e realizzata come segue:

```
function err = crossEntropy(y,t)
%calcolo dell'errore tramite cross-entropy
%y: output della rete
%t: label del DS

    err = (-sum(sum(t .* log(y),2)));

end
```

In entrambe le funzioni di errore sono state effettuate due sommatorie dal momento che è necessario sommare infine gli errori commessi su ciascun punto del training set.

Per calcolare la derivata della funzione di errore, è stata implementata una funzione apposita, *computeGradErr*, che la restituisce e la calcola opportunamente sulla base della funzione di errore correntemente utilizzata e sull'utilizzo o meno della softmax come operatore di post-processing.

La funzione agirà come segue:

$$\begin{cases} \frac{\partial E^{(n)}}{\partial y_k} = (y_k - t_k) \;\; se \text{ "ss"} \\ \frac{\partial E^{(n)}}{\partial y_k} = -\frac{t_h^n}{y_h^n} \qquad se \text{ "ce" } senza\ softmax \\ \frac{\partial E^{(n)}}{\partial y_k} = (y_k - t_k) \;\; se \text{ "ce" } con\ softmax \end{cases}$$

```
function gradErr = computeGradErr(y,t,loss,hassoftmax)
%calcolo della derivata della funzione di errore con/senza softmax
%y: output della rete
%t: label del DS
%hassoftmax: booleano

    if ~hassoftmax
```

```
        switch(loss)

            case("ss")
                gradErr = y-t;

            case("ce")
                n = size(y); %quante volte iterare la sommatoria
                gradErr = -sum(t./y, n(2));
        end

    else

        z = softmax(y); %applico softmax
        gradErr = z-t; %cross-entropy con softmax
   end
end
```

2.2 Parte B

Questa seconda parte è focalizzata sull'aggiornamento dei parametri della rete seguendo delle particolari procedure che fanno uso di alcuni iperparametri che hanno lo scopo di calibrare il valore che verrà poi assegnato ai pesi e ai bias con l'obiettivo di rendere quanto più affidabile e corretto l'addestramento della rete. Inoltre verrà anche descritta una particolare funzione di attivazione, la relu, che è stata utilizzata per condurre un'analisi più approfondita relativamente al comportamento della rete al variare di tale funzione.

2.2.1 RPROP: RESILIENT PROPAGATION

Questo algoritmo fu creato come alternativa alla Discesa del Gradiente in quanto meno sensibile alla variazione degli iperparametri. Gli step principali per l'aggiornamento sono i seguenti:

- ✓ Ad ogni peso w_{ij} si associa uno step di aggiornamento $\Delta_{ij}>0$
- ✓ Si modificano opportunamente i Δ_{ij} durante la fase di learning
- ✓ La regola base diventa: $w_{ij}^t = w_{ij}^{t-1} - sign\left(g_{ij}^t\right) \cdot \Delta_{ij}$,
  dove $g_{ij}^t \equiv \frac{\partial E^t}{\partial w_{ij}}$ nel caso batch learning, con E funzione di errore
- ✓ In questo modo otteniamo due situazioni che richiedono azioni differenti:
  - se $g_{ij}^t<0$ allora si procede verso un minimo locale ed è quindi possibile fare dei passi più grandi per avvicinarvisi. Ciò si traduce con un incremento dei Δ_{ij}
  - se $g_{ij}^t>0$ allora il passo che è stato compiuto ha comportato un superamento del minimo locale, per cui si è subentrati in una situazione negativa. Ciò si traduce con un decremento dei Δ_{ij}

Per l'incremento e il decremento dei Δ_{ij} vengono utilizzati due iperparametri che sono stati settati in un certo modo seguendo il paper [1] e sono rispettivamente:

- ✓ $\eta^+ > 1$ per incrementare
- ✓ $0 < \eta^- < 1$ per decrementare

```
etaP = 1.2; %etaP > 1 per incrementare delta(i,j)

etaN = 0.5; %0 < etaN < 1 per decrementare delta(i,j)
```

Ciò che adesso bisogna analizzare, è come i Δ_{ij} vengono aggiornati durante il processo di learning. Per farlo basta semplicemente capire com'è il segno della derivata della funzione g poiché in base a questo si hanno informazioni circa la grandezza del passo effettuato:

- ✓ se $g_{ij}^t \times g_{ij}^{t-1}$<0, allora lo step è stato troppo grande → riduzione dei Δ_{ij}
- ✓ se $g_{ij}^t \times g_{ij}^{t-1}$>0, allora lo step è stato troppo piccolo → incremento dei Δ_{ij}

Lo snippet seguente mostra solo il caso di aggiornamento dei delta associati ai pesi (equivalentemente verrà fatto anche per i bias) e sfrutta una particolare facilitazione di matlab che permette di sostituire i valori aggiornati direttamente nella matrice contenente i segni del prodotto delle derivate, in modo tale che valori > 0 verranno sostituiti con il prodotto tra quello stesso valore e η^+, altrimenti con il prodotto con η^-. Inoltre il paper [1] prevede che i valori dei delta vengano normalizzati all'interno di un range che è specificato da *deltaMin* e *deltaMax*, in modo tale da non fargli assumere valori troppo elevati o piccoli.

```
%range di valori per i delta
deltaMax = 0.00025;
deltaMin = 10^(-6);

if epoch > 1

  for i=1:length(layers)

    %prodotto positivo
    layers(i).D.W(layers(i).matrixSign.W>0) =
layers(i).D.W(layers(i).matrixSign.W>0)*etaP;

    layers(i).D.W(layers(i).matrixSign.W>0) =
min(layers(i).D.W(layers(i).matrixSign.W>0),deltaMax);

     %prodotto negativo
     layers(i).D.W(layers(i).matrixSign.W<0) =
layers(i).D.W(layers(i).matrixSign.W<0)*etaN;

     layers(i).D.W(layers(i).matrixSign.W<0) =
max(layers(i).D.W(layers(i).matrixSign.W<0),deltaMin);
```

La creazione e l'aggiornamento della matrice *matrixSign* vengono effettuati all'interno della Back-Propagation, come mostra lo snippet seguente:

```
%salvo il gradiente del layer i-esimo prima che cambi
layers(i).preGradient.W = layers(i).gradient.W;
layers(i).preGradient.B = layers(i).gradient.B;

[…]

%calcolo del segno
layers(i).matrixSign.W = sign(layers(i).preGradient.W .* layers(i).gradient.W);
```

Viene salvato il gradiente corrente nella struttura preGradient.W prima di fare il calcolo di quello aggiornato, per poi effettuarne il prodotto e calcolare il segno inserendo il valore nella matrice, adoperata nella RPROP.

Ora manca l'ultimo step che è quello che segue l'aggiornamento dei delta, ovvero modificare opportunamente i pesi (equivalentemente anche i bias) adoperando i valori cambiati precedentemente.

Lo snippet seguente altro non è che la formalizzazione di quanto è stato accennato in precedenza, ovvero si effettua il prodotto element-wise tra il segno della derivata associata a quei pesi e i delta aggiornati. Infine si effettua la differenza tra il valore dei pesi e ciò che è stato calcolato prima.

```
% aggiornamento dei pesi
g = sign(layers(i).gradient.W).*layers(i).D.W;
layers(i).W = layers(i).W-g;
```

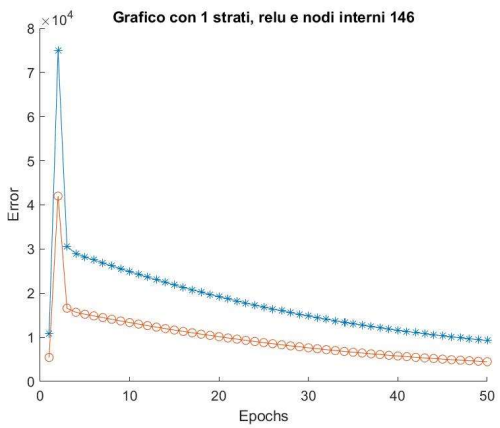
Come è possibile notare, questo procedimento dipende esclusivamente da solo due iperparametri e una loro drastica variazione non si riflette sull'aggiornamento dei parametri della rete, mantenendo sempre un certo equilibrio che serve al corretto addestramento.

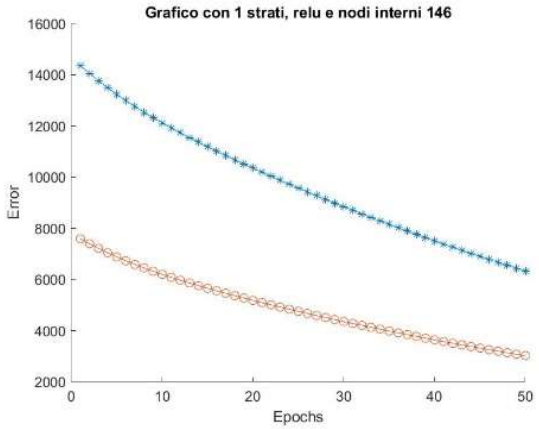
Gli ultimi due aspetti da valutare sono relativi all'inizializzazione dei Δ_{ij} che può avvenire seguendo un approccio randomico e l'aggiornamento dei parametri della rete al primo step, o meglio alla prima 'epoca' di addestramento. E' possibile seguire due strade:

- ✓ O non aggiornare i parametri e lasciare quindi inalterati i valori già presenti
- ✓ O utilizzare la discesa del gradiente che dipende da un ulteriore iperparametro, detto coefficiente di learning che solitamente è $\theta \cong 0.001$

Nello sviluppo di questa strategia di aggiornamento i Δ_{ij} sono stati inizializzati ponendo i loro valori a zero, mentre per quanto riguarda la seconda problematica, è stato scelto di procedere non aggiornando i pesi poiché è stato riscontrato che con l'utilizzo della discesa del gradiente, nella valutazione dell'errore di training e di validazione la rete provocava un picco di entrambi gli errori nella seconda epoca quindi quella successiva a quella in cui vengono aggiornati per la prima volta i parametri della rete usando tale tecnica. Ciò

invece non avviene lasciando inalterati i parametri e provocando quindi un comportamento più lineare e coerente, come si potrà osservare dai seguenti grafici:



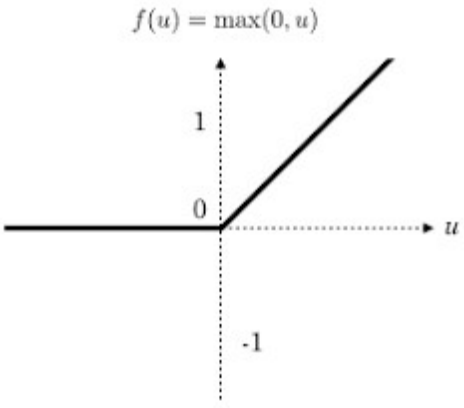


Il primo grafico mostra il comportamento degli errori di training e validation con la discesa del gradiente come metodo di aggiornamento dei parametri per la prima epoca, mentre il secondo non effettua alcuna operazione di aggiornamento. Ciò dimostra come l'algoritmo renda instabile l'andamento dell'errore a causa del fatto che setta i parametri in modo poco coerente rispetto a come viene fatto successivamente dalla RPROP.

2.2.2 RELU: RECTIFIED LINEAR UNITS

La relu è una funzione derivabile e non lineare definita nel seguente modo:

$$Relu(a) = \max(0, a)$$



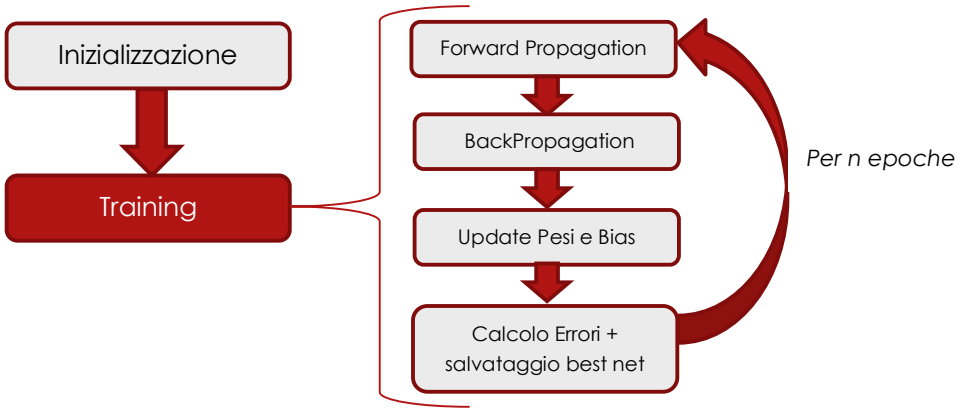
Ovvero è una funzione che ritorna a quando è maggiore di zero, zero altrimenti, come mostrato nella figura 3. Chiaramente ci sono infiniti punti per cui la derivata è 1 e ciò ha reso possibile l'integrazione della fase di estrazione delle caratteristiche nel processo di learning, anche in reti con un numero elevato di strati. La funzione Relu aiuta quindi le reti neurali a formare modelli per un deep-learning ottimizzato. A causa dei problemi del vanishing gradient in diversi strati, non è possibile utilizzare l'attivazione a tangente iperbolica e sigmoide ed è quindi possibile superarli tramite la Relu. Questo permette al modello di eseguire meglio e imparare più velocemente.

III capitolo

Fase di addestramento

3.1 La funzione Train

Una volta realizzate in maniera modulare la rete e le altre componenti necessarie appena descritte, esse sono state impiegate insieme all'interno di una funzione "*train*" dedicata esclusivamente alla fase di addestramento ed è strutturata come segue:



La funzione train prende in ingresso i seguenti parametri:

- La rete precedentemente costruita *net;*
- il training Set costituito dagli input (*XTrain*) e dai corrispondenti labels (*Ttrain*);
- il Validation Set costituito dagli input (*XVal*) e dai corrispondenti labels (*TVal*);
- una stringa *loss* che indica la funzione d'errore da adoperare;
- un intero *epochs* per indicare il numero di epoche
- un booleano *hassoftmax* per utilizzare o meno la softmax come operazione di post-processing.

La fase di inizializzazione include la creazione di strutture da restituire alla fine dell'addestramento ossia *err* ed *errVal* , vettori di lunghezza *epochs* che contengono l'errore calcolato rispettivamente sul training set e sul validation set per ogni epoch.

```
function [err, newNet, errVal] =
train(net,XTrain,TTrain,XVal,TVal,loss,epochs,hassoftmax)
    err = zeros(1,epochs);
    errVal = zeros(1,epochs);
    score = zeros(1,epochs);



    yVal = predict(net,XVal,hassoftmax);
    minErr = crossEntropy(yVal,TVal);
```

```
    newNet = net;
    […]
end
```

Dopodiché, prima ancora di avviare il training della rete, si adopera la rete per risolvere il task di classificazione sul validation set e calcolare quindi l'errore commesso, per poter poi effettuare in maniera consistente, come vedremo a breve, un confronto per valutare se la rete in una certa epoca risulta essere la migliore in quel momento, al fine di salvarla nella variabile *newNet*.

A questo punto inizia il loop all'interno del quale avviene l'addestramento per n epoche.

```
        net.layers = forwardProp(net.layers,XTrain);
        y = net.layers(size(net.layers,2)).z;
        gradErr = computeGradOut(y,TTrain,loss,hassoftmax);
        net.layers = backProp(net.layers,gradErr,XTrain);
        net.layers = resilientProp(net.layers,epoch);
```

Dopo aver effettuato la forward propagation, aver calcolato la derivata della funzione di errore e calcolato il gradiente di pesi e bias di ogni strato si effettua l'aggiornamento dei pesi mediante l'algoritmo RPROP.

A questo punto, la rete, appena aggiornata, eseguirà la classificazione sul training set e il validation set. Quindi, in base al parametro *loss*, si calcolano gli errori con la funzione d'errore indicata.

```
        y = predict(net,XTrain,hassoftmax);
        yVal = predict(net,XVal,hassoftmax);

        if loss == "ce"

            err(epoch) = crossEntropy(y,TTrain);
            errVal(epoch) = crossEntropy(yVal,TVal);
        else

            err(epoch) = sumOfSquares(y,TTrain);
            errVal(epoch) = sumOfSquares(yVal,TVal);
        end
```

Dunque se l'errore sul validation set ottenuto all'epoca corrente, risulta essere minore di quello ritenuto essere il più piccolo, viene aggiornato *minErr* e la rete migliore viene considerata quella dell'epoca appena conclusa.

```
        if errVal(epoch) < minErr

            minErr = errVal(epoch);
            newNet = net; %salvo la rete con errore minimo
        end
```

Alla fine dell'addestramento la funzione restituisce la rete migliore, gli errori sul TS e VS di ogni epoca.

```
function [err, newNet, errVal] =
train(net,XTrain,TTrain,XVal,TVal,loss,epochs,hassoftmax)

    %inizializzo gli errori e lo score
    err = zeros(1,epochs);
    errVal = zeros(1,epochs);
    score = zeros(1,epochs);

    %predico l'output e valuto l'errore di predizione
    yVal = predict(net,XVal,hassoftmax);
    minErr = crossEntropy(yVal,TVal);
    newNet = net;

    for epoch=1:epochs

        net.layers = forwardProp(net.layers,XTrain); %imposto i parametri
        y = net.layers(size(net.layers,2)).z;

        gradErr = computeGradOut(y,TTrain,loss,hassoftmax); %calcolo der.err

        net.layers = backProp(net.layers,gradErr,XTrain); %calcolo gradiente
        net.layers = resilientProp(net.layers,epoch); %aggiorno i parametri

        y = predict(net,XTrain,hassoftmax);
        yVal = predict(net,XVal,hassoftmax);

        %calcolo l'errore di predizione
        if loss == "ce"

            err(epoch) = crossEntropy(y,TTrain);
            errVal(epoch) = crossEntropy(yVal,TVal);
        else

            err(epoch) = sumOfSquares(y,TTrain);
            errVal(epoch) = sumOfSquares(yVal,TVal);
        end

        score(epoch) = accuracy(yVal,TVal);

        if errVal(epoch) < minErr

            minErr = errVal(epoch);
            newNet = net; %salvo la rete con errore minimo
        end
    end
end
```

IV capitolo

Studio della rete

In questa sezione viene analizzata la rete precedentemente costruita al variare di alcuni iperparametri, in particolare:

- ✓ Numeri di strati interni
- ✓ Funzione di attivazione (sigmoide o ReLU)
- ✓ Numero di nodi per ogni strato

Migliori parametri: 2 strati, relu e nodi interni 146 100 61 44 13>>

4.1 Costruzione dello script

Per selezionare il modello più adeguato per soddisfare la richiesta, è stato implementato uno script che usa un approccio a griglia per far variare gli iperparametri precedentemente elencati e la rete selezionata è quella che ottiene l'accuratezza media maggiore sul TestSet.

La funzione che effettua il calcolo dell'accuratezza, è stata implementata nel modo seguente:

```
function score = accuracy(y,t)

    scores = zeros(1,size(y,2)); %inizializzo gli score

    %fisso una soglia di valutazione
    y(y>=0.5)=1;
    y(y<0.5)=0;

    for img=1:size(y,2)
        for class=1:size(y,1)

            if y(class,img) == t(class,img)
                scores(img) = 1;
            else
                scores(img) = 0;
            end
        end
    end

    score = mean(scores); %faccio la media degli score
end
```

Di seguito viene riportato il frammento di codice atto a svolgere le suddette funzionalità:

```
parameters;

neuronsChoices = [randi([104 150],1,3);randi([87 103],1,3);randi([61
86],1,3);randi([31 60],1,3);randi([1 30],1,3)];
act = ["relu" "sigmoide"];
nHiddenLayers = 5;
choiceNeurons= 3;
```

```
oldScore = 0;
currentPath = convertCharsToStrings(pwd);


for totalLayers=1:nHiddenLayers
   for actFunction=1:length(act)
       for numberNeurons=1:choiceNeurons
           neurons = zeros(1,totalLayers);
           disp(["Strati interni: " num2str(totalLayers) "F. attivazione: "
act(actFunction) "Numero neuroni: " num2str(numberNeurons)]);
           for layer=1:totalLayers
                neurons(layer) = neuronsChoices(layer,numberNeurons);
                params.neurons = neurons;
                params.nLayers = totalLayers+1;
                params.act(layer) = act(actFunction);
           end
           [newNet, err, errVal, score] = main(params);
           if oldScore < score
               bestParams.layers = totalLayers;
               bestParams.act = act(actFunction);
               bestParams.choice = neuronsChoices(:,numberNeurons);
           end
          % creazione file dei plot
           figure('visible','off');
           hold on;
           plot((1:1:params.epochs),err,'-*',(1:1:params.epochs),errVal,'-o');
           title(sprintf("Grafico con %d strati, %s e nodi interni
%s",totalLayers,act(actFunction),num2str(neuronsChoices(1:totalLayers,numberNeur
ons)')));
           xlabel('Epochs');
           ylabel('Error');
           set(gca, 'XTick', (0:10:params.epochs));
           set(gca, 'xlim', [0 params.epochs]);
           fileName = totalLayers + "_" + actFunction + "_" + numberNeurons +
".jpg";
           directoryName = currentPath + '\images\';
           saveas(gcf,directoryName + fileName);
           hold off;
       end
   end
end
fprintf("Migliori parametri: %d strati, %s e nodi interni %s",bestParams.layers,
bestParams.act, num2str((bestParams.choice))');
```

Tutti i modelli candidati sono stati addestrati per 50 epoche.

Per implementare la strategia a griglia sono state utilizzate delle iterazioni innestate:

- ✓ La prima iterazione, quella più esterna varia sul numero di strati interni della rete, il cui numero massimo è stato fissato a 5.
- ✓ La seconda iterazione invece opera variando la funzione di attivazione da considerare.
- ✓ La terza iterazione invece varia il numero di nodi degli strati interni. Per realizzare questa variazione è stata adoperata una matrice di dimensione 5x3 che, per ogni strato considera tre soluzioni diverse per il numero di nodi generate randomicamente in un range prefissato e la scelta dei nodi per ogni strato i è stata realizzata in modo tale da ordinare i valori in ordine decrescente.

Nella iterazione più interna, per ogni modello candidato è stato descritto graficamente l'andamento dell'errore sul training set e sul validation set per ogni epoca.

Ogni grafico è stato opportunamente salvato in formato .jpg in un path prefissato e con un nome la cui codifica è strutturata concatenando gli interi associati agli iperparametri che variano.

Alla fine dello script viene poi mostrato a schermo la combinazione di iperparametri migliore che ne formalizzano il modello.

4.2 Valutazione e conclusioni

Strati interni	F. di attivazione	Numero neuroni	Accuratezza (%)
1	ReLU	110	90,68
1	ReLU	119	90,84
1	ReLU	146	92,32
1	Sigmoid	110	90,4
1	Sigmoid	119	90,4
1	Sigmoid	146	90,4
2	ReLU	110 100	90,84
2	ReLU	119 102	91,48
2	**ReLU**	**146 100**	**93,24**
2	Sigmoid	110 100	90,4
2	Sigmoid	119 102	90,4
2	Sigmoid	146 100	90,4
3	ReLU	110 100 81	90,4
3	ReLU	119 102 72	90,44
3	ReLU	146 100 61	90,4
3	Sigmoid	110 100 81	90,4
3	Sigmoid	119 102 72	90,4
3	Sigmoid	146 100 61	90,4
4	ReLU	110 100 81 44	90,44
4	ReLU	119 102 72 59	90,44
4	ReLU	146 100 61 44	90,4
4	Sigmoid	110 100 81 44	90,4
4	Sigmoid	119 102 72 59	90,4
4	Sigmoid	146 100 61 44	90,4
5	ReLU	110 100 81 44 30	90,4
5	ReLU	119 102 72 59 22	90,4
5	ReLU	146 100 61 44 13	90,4
5	Sigmoid	110 100 81 44 30	90,4
5	Sigmoid	119 102 72 59 22	90,4
5	Sigmoid	146 100 61 44 13	90,4

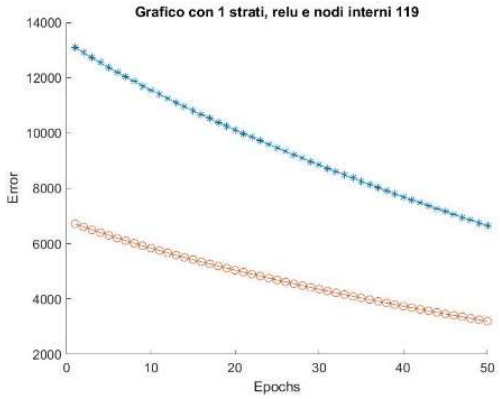
L'avvio dello script ha generato la seguente scelta di possibili nodi interni per ogni strato:

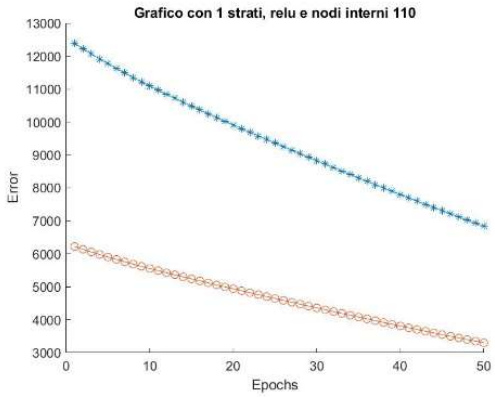
110	119	146
100	102	100
81	72	61
44	59	44
30	22	13

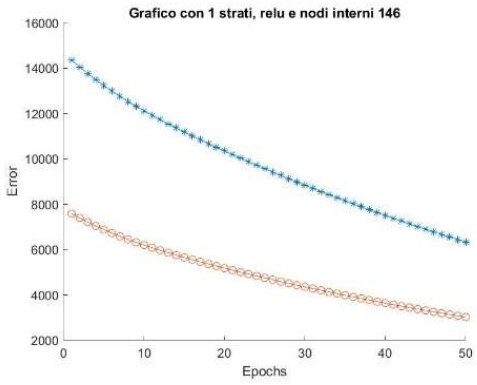
I grafici e i risultati prodotti facendo variare gli iperparametri sono i seguenti:

Legenda:

~~xxx~~ : errore sul Training Set
~~ooo~~ : errore sul Validation Set







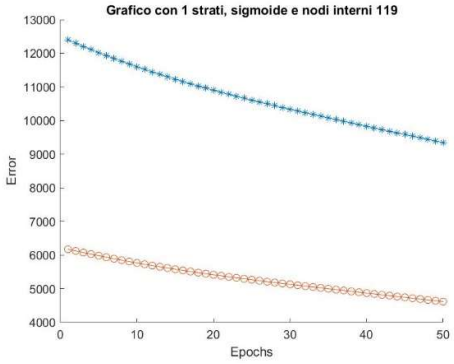


Grafico con 1 strati, sigmoide e nodi interni 119
Error
Epochs

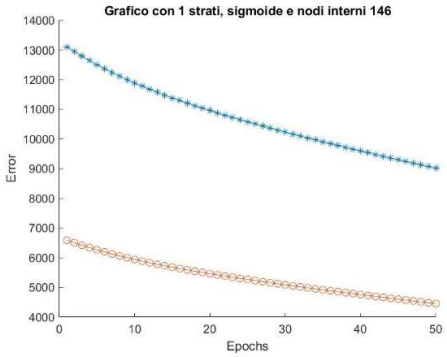


Grafico con 1 strati, sigmoide e nodi interni 146
Error
Epochs

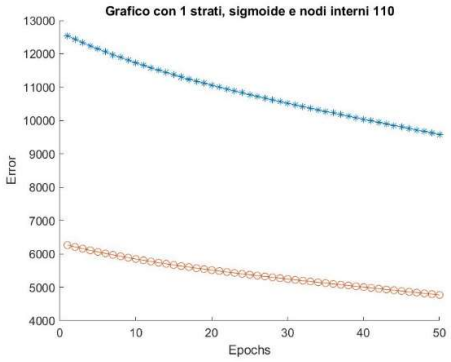


Grafico con 1 strati, sigmoide e nodi interni 110
Error
Epochs

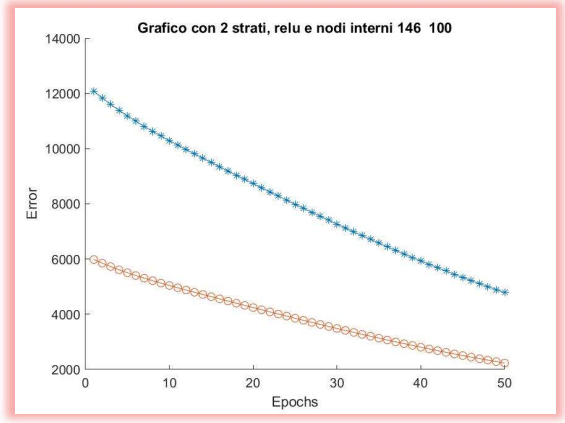


Grafico con 2 strati, relu e nodi interni 146 100
Error
Epochs

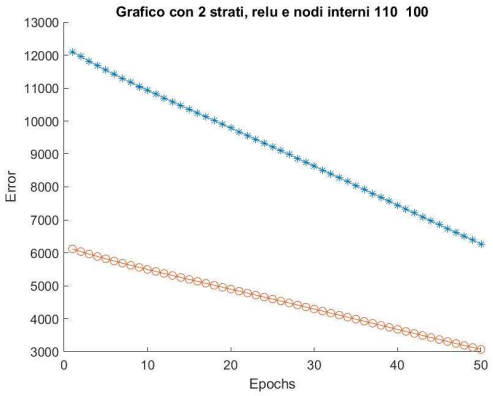


Grafico con 2 strati, relu e nodi interni 110 100
Error
Epochs

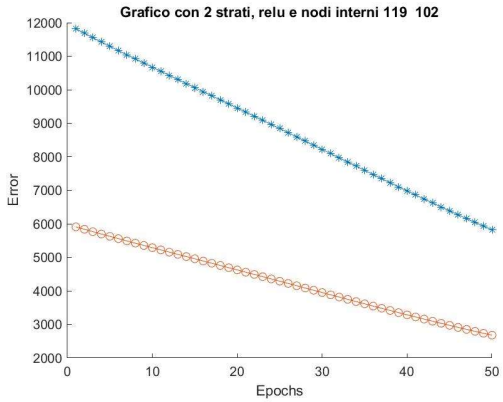


Grafico con 2 strati, relu e nodi interni 119 102
Error
Epochs

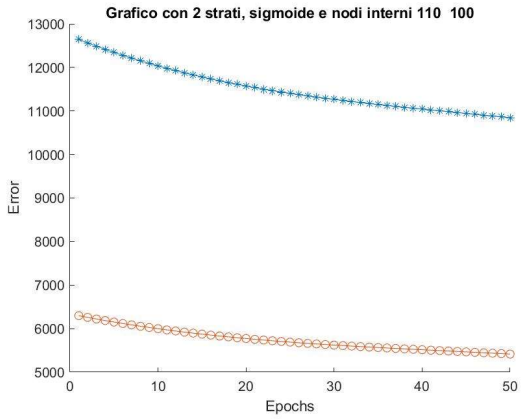


Grafico con 2 strati, sigmoide e nodi interni 110 100
Error
Epochs

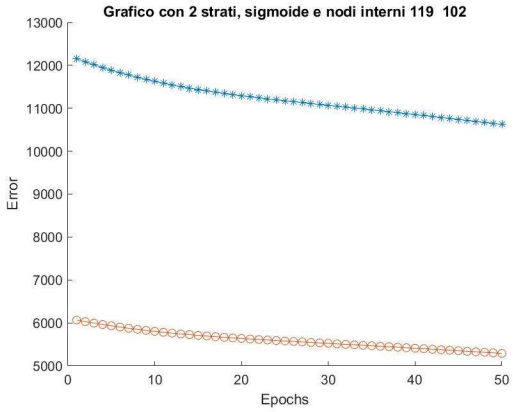


Grafico con 2 strati, sigmoide e nodi interni 119 102
Error
Epochs

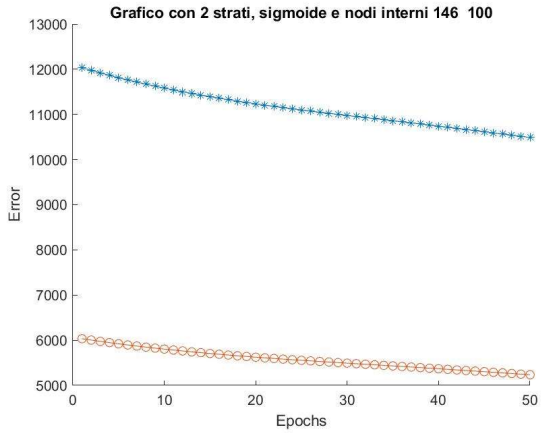


Grafico con 2 strati, sigmoide e nodi interni 146 100
Error
Epochs

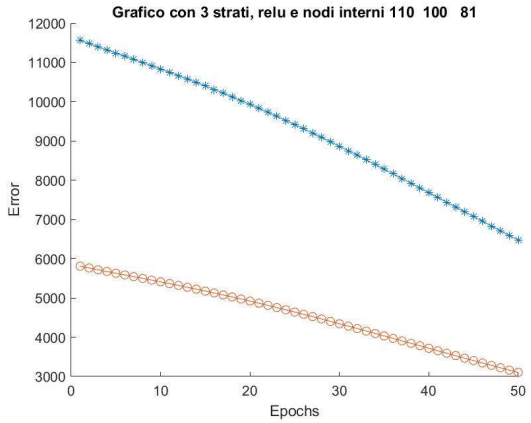


Grafico con 3 strati, relu e nodi interni 110 100 81
Error
Epochs

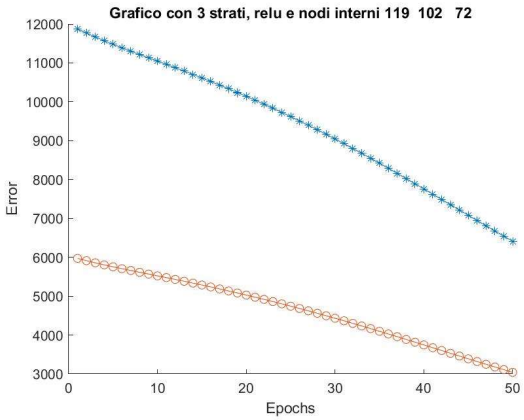


Grafico con 3 strati, relu e nodi interni 119 102 72
Error
Epochs

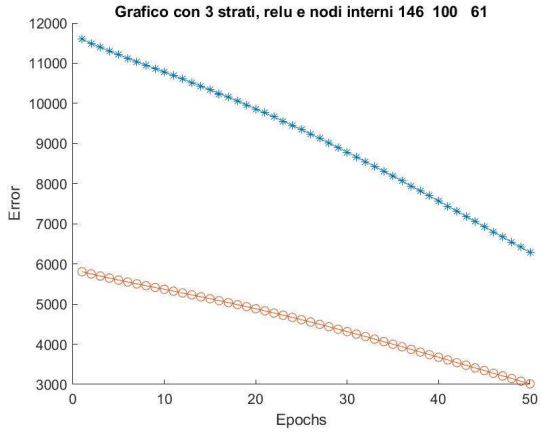


Grafico con 3 strati, relu e nodi interni 146 100 61
Error
Epochs

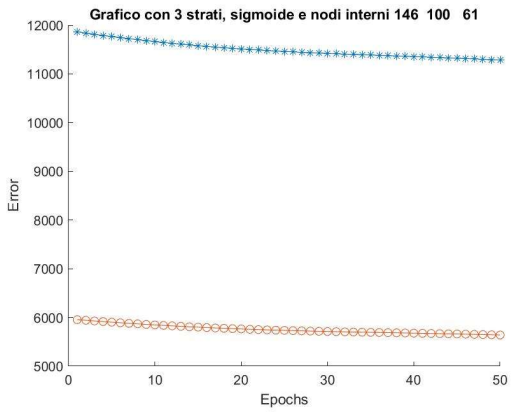


Grafico con 3 strati, sigmoide e nodi interni 146 100 61
Error
Epochs

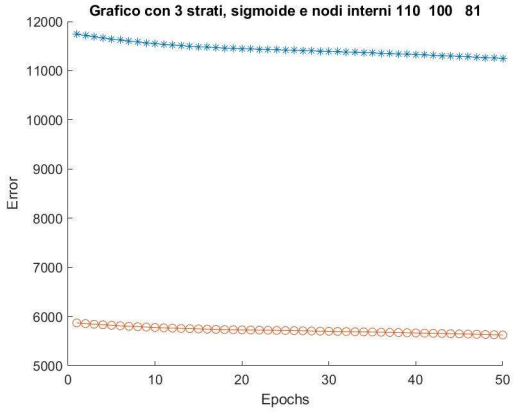


Grafico con 3 strati, sigmoide e nodi interni 110 100 81
Error
Epochs

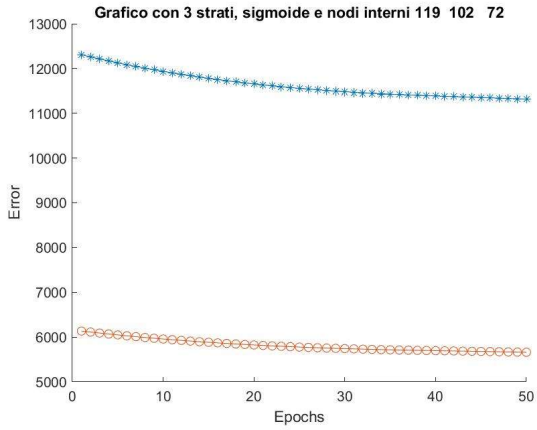


Grafico con 3 strati, sigmoide e nodi interni 119 102 72
Error
Epochs

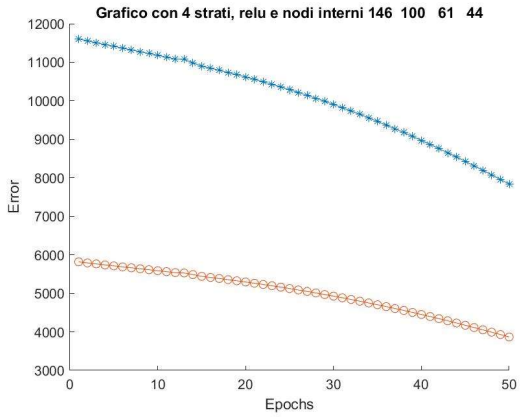


Grafico con 4 strati, relu e nodi interni 146 100 61 44
Error
Epochs

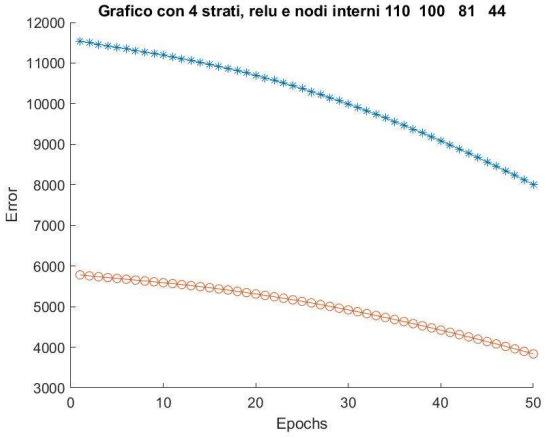


Grafico con 4 strati, relu e nodi interni 110 100 81 44
Error
Epochs

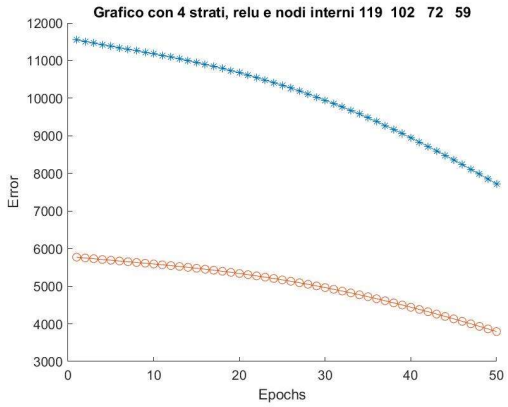


Grafico con 4 strati, relu e nodi interni 119 102 72 59
Error
Epochs

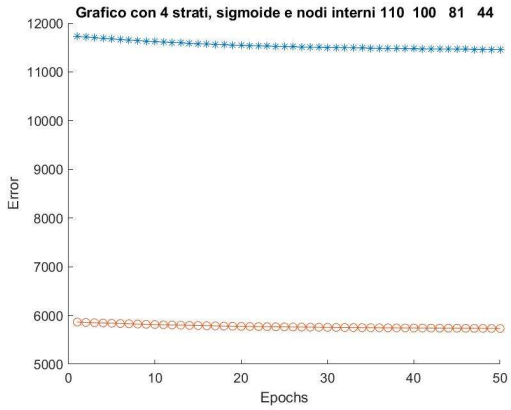


Grafico con 4 strati, sigmoide e nodi interni 110 100 81 44
Error
Epochs

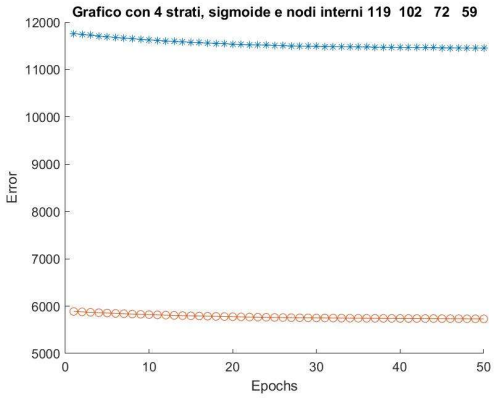


Grafico con 4 strati, sigmoide e nodi interni 119 102 72 59
Error
Epochs

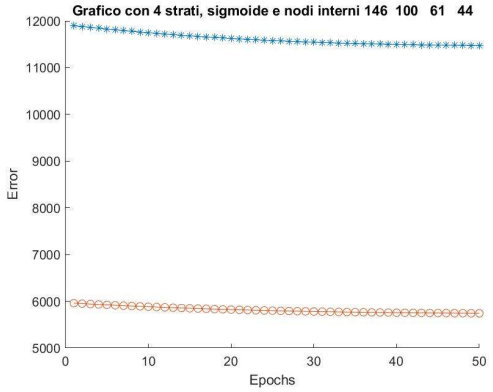


Grafico con 4 strati, sigmoide e nodi interni 146 100 61 44
Error
Epochs

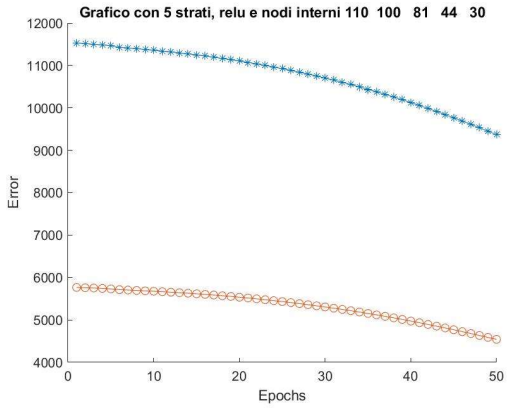


Grafico con 5 strati, relu e nodi interni 110 100 81 44 30
Error
Epochs

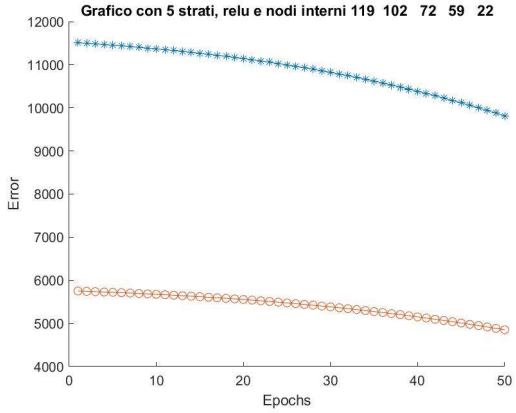


Grafico con 5 strati, relu e nodi interni 119 102 72 59 22
Error
Epochs

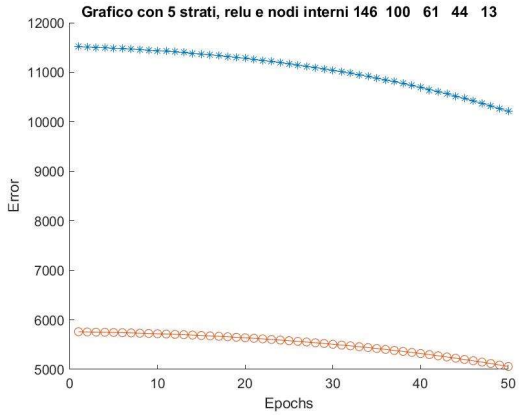


Grafico con 5 strati, relu e nodi interni 146 100 61 44 13
Error
Epochs

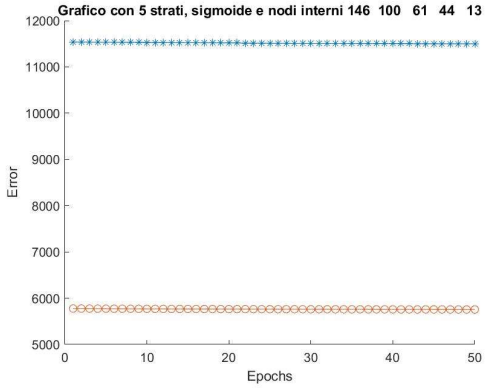


Grafico con 5 strati, sigmoide e nodi interni 146 100 61 44 13
Error
Epochs

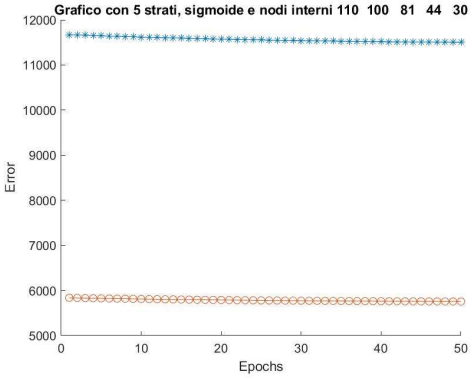


Grafico con 5 strati, sigmoide e nodi interni 110 100 81 44 30
Error
Epochs

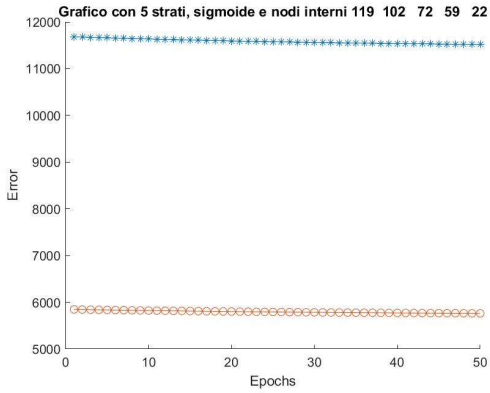


Grafico con 5 strati, sigmoide e nodi interni 119 102 72 59 22
Error
Epochs

Riferimenti

[1] Riedmiller M. et al. « A Direct Adaptive Method for Faster Backpropagation Learning:

The RPROP Algorithm » . https://paginas.fe.up.pt/~ee02162/dissertacao/RPROP%20paper.pdf