



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

CORSO DI LAUREA MAGISTRALE
IN INFORMATICA

MACHINE LEARNING E APPLICAZIONI - MODULO B
VISIONE COMPUTAZIONALE

Studio dell'articolo “Extracting and Composing Robust Features with Denoising Autoencoders”, sviluppo e valutazione di una rete neurale dello stesso tipo

Autori:

Paolo Nicola Perrotta N97000266
Crispino Cicala N97000264

Professori:

Francesco Isgrò
Roberto Prevete

14 novembre 2018

Indice

| | |
|---------------------------------------------------|-----------|
| 1 Introduzione | 3 |
| 1.1 Descrizione lavoro svolto | 3 |
| 1.1.1 MNIST | 3 |
| 1.2 Denoising autoencoder | 4 |
| 2 Implementazione | 5 |
| 2.1 Caricamento dataset | 5 |
| 2.2 Creazione set | 6 |
| 2.3 Creazione rete neurale FFML | 6 |
| 2.4 Funzioni di errore | 8 |
| 2.4.1 Somma dei quadrati | 9 |
| 2.4.2 Cross entropy | 9 |
| 2.5 Funzioni di attivazione | 10 |
| 2.5.1 Sigmide | 10 |
| 2.5.2 Identità | 10 |
| 2.5.3 ReLU | 11 |
| 2.6 Addestramento | 11 |
| 2.6.1 Addestramento batch | 12 |
| 2.6.2 Addestramento autoencoder | 13 |
| 2.7 Discesa del gradiente | 16 |
| 2.7.1 Backpropagation | 17 |
| 2.7.2 Calcolo derivata pesi | 17 |
| 2.7.3 Aggiornamento dei pesi | 18 |
| 2.8 Forward Propagation | 18 |
| 2.9 Aggiunta rumore | 19 |
| 2.10 Testing | 21 |
| 2.10.1 Test globale | 21 |
| 2.10.2 Test singola configurazione | 22 |
| 2.10.3 Valutazione rete | 24 |
| 2.10.4 Gestione rappresentazione output | 25 |
| 3 Test e Risultati | 25 |
| 3.1 Struttura test | 25 |
| 3.2 Esempi di ricostruzione | 26 |
| 3.3 Valutazione accuratezza | 29 |
| 4 Appendice | 33 |

1 Introduzione

1.1 Descrizione lavoro svolto

Seguendo le indicazioni riportate nella parte A dei progetti di MLEA-B, sono state progettate ed implementate funzioni per la simulazione e la propagazione in avanti di una rete neurale multi-strato con: n strati, sigmoide come funzione di output dei nodi interni ed identità come funzione di output dei nodi di output. Inoltre, tramite gli script realizzati, è possibile implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di output per ciascun strato. Per concludere la parte A, sono state progettate ed implementate funzioni per la realizzazione della backpropagation adatta alle reti descritte, è possibile, inoltre, utilizzare la backpropagation con un numero qualsiasi di strati interni, con funzione di output a piacimento per ciascuno strato e con qualsiasi funzione di errore derivabile rispetto all'output. Dopo aver studiato l'articolo *“Extracting and composing robust features with denoising autoencoders”* [1] è stato utilizzato il linguaggio MATLAB per implementare le parti che vanno a comporre la creazione, la gestione e l'analisi dei risultati di una rete neurale multistrato feed-forward, al fine di replicare il lavoro descritto nel suddetto articolo. In particolare, vanno a formare il progetto finale la creazione di una rete neurale per la classificazione degli elementi del dataset MNIST ed un “denoising autoencoder” per la ricostruzione delle immagini sottoposte a rumore. L'obiettivo finale è quello di valutare l'accuratezza della rete per la classificazione in risposta a diversi tipi di input. Gli input forniti alla rete sono rappresentati dagli elementi originali del dataset MNIST 1.1.1, dagli stessi elementi sottoposti a rumore e infine dagli elementi ricostruiti dall'autoencoder. Il tipo di rumore utilizzato dagli autori dell'articolo è generato azzerando in maniera randomica un certo numero di pixel dell'immagine. È stato inoltre deciso di valutare le prestazioni delle reti anche con altri tipi di rumore.

1.1.1 MNIST

Sulla base della scelta effettuata dagli autori dell'articolo, si è scelto come dataset per la sperimentazione un sottoinsieme del dataset MNIST. Il dataset è composto da 60.000 immagini fornite come training set e 10.000 immagini fornite come test set. Le immagini hanno una dimensione fissa di 28x28 pixel e rappresentano le cifre da 0 a 9 scritte a mano, ad ognuna di queste è associata una label che indica quale cifra è rappresentata all'interno dell'immagine. Per rendere il dataset più esaustivo possibile, le cifre, oltre che in forma standard, vengono ruotate o leggermente deformate. Le immagini sono fornite come array da 784 elementi, ognuno dei quali ha un valore compreso tra 0 e 1 e rappresenta l'intensità del singolo pixel. Non si tratta di immagini binarie, visto che non sono previsti i soli valori 0 e 1, ma anche le sfumature di grigio che si trovano tra questi due valori. I primi 28 elementi rappresentano la prima riga dell'immagine, quelli da 29 a 56 la seconda e così via. In totale, come già accennato, ogni immagine avrà 784 elementi, 28 righe da 28 elementi.

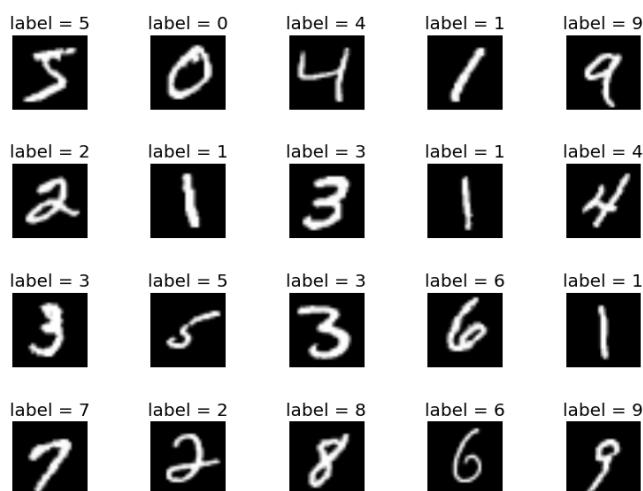


Figura 1: Esempio dataset MNIST

1.2 Denoising autoencoder

Nell'articolo su cui si basa il lavoro svolto, gli autori si sono posti come obiettivo principale quello di ideare un nuovo approccio per l'addestramento delle reti neurali, tale da riuscire ad estrarre caratteristiche robuste rispetto al rumore applicato sull'input. A tale scopo, l'idea è stata quella di utilizzare un autoencoder che, così utilizzato, è stato denominato “*Denoising Autoencoder*”. Un *autoencoder* è una rete neurale composta da almeno tre livelli, uno di input, uno hidden ed uno di output, una struttura descritta dalla figura 2.

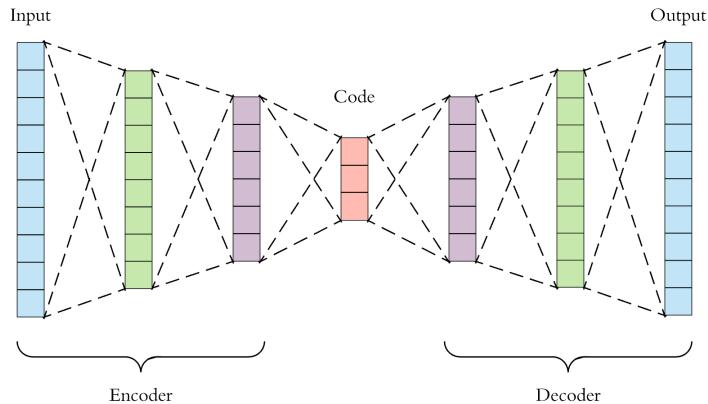


Figura 2: Struttura Autoencoder

Dato un certo input X , inizialmente la rete lo mappa in una rappresentazione codificata negli y nodi del livello hidden, in particolare avremo che $y = f_\theta(x) = s(Wx + b)$, la funzione di mapping è parametrizzata da $\theta = \{W, b\}$ con W matrice dei pesi e b è il vettore dei bias. La rappresentazione codificata viene poi decodificata per cercare di ricostruire al meglio l'input originale, avremo quindi un input ricostruito z calcolato come $z = g_{\theta'}(y) = s(W'y' + b')$ con $\theta' = \{W', b'\}$. Utilizzando la tecnica dei *pesi legati*, si può porre $W' = W^T$. La rete autoencoder cercherà, durante la fase di addestramento, di minimizzare il più possibile l'errore di ricostruzione, essendo un problema di regressione, generalmente la funzione di errore (*loss*) utilizzata è la *somma dei quadrati*, anche se è possibile adattare diverse funzioni di errore. Un Denoising Autoencoder, invece, è un particolare tipo di autoencoder utilizzato allo scopo di ricostruire dei dati sporcati da un certo rumore eliminando quest'ultimo.

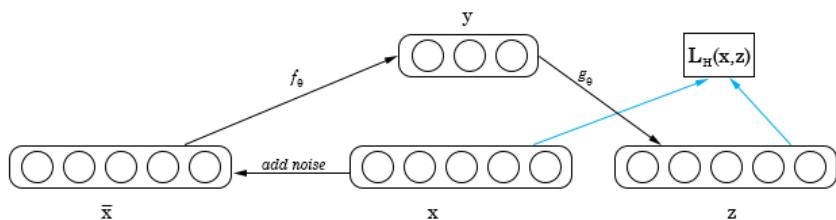


Figura 3: Schema Denoising Autoencoder

Come si vede in figura 3, l'input iniziale x viene parzialmente distrutto applicando forzatamente del rumore, da questo processo si ottiene l'input sporco \tilde{x} , questo input viene passato alla rete in modo da addestrarla a trovare internamente una certa rappresentazione codificata $y = f_\theta(\tilde{x}) = s(W\tilde{x} + b)$ dalla quale sia possibile ricostruire, all'uscita della rete, uno z corrispondente all'input originale senza rumore x , in formula: $z = g_{\theta'}(y) = s(W'y' + b') = x$.

2 Implementazione

Il progetto è stato sviluppato utilizzando Matlab, dividendo procedure e funzioni in script che sono poi stati utilizzati, attraverso un script principale, per testare una singola routine che comprende le fasi di addestramento, ricostruzione e classificazione. Inoltre, per studiare il comportamento delle reti su più iterazioni, è stato implementato uno script che richiama più volte l'esecuzione dell'esperimento e ne stima i risultati in media. In questa sezione verranno presentate le procedure e le funzioni implementate, descrivendone le logiche sulle quali si basano e le scelte di progettazione.

2.1 Caricamento dataset

La funzione `caricaDataset` carica le immagini contenute nei file del dataset MNIST, inserendole nelle strutture dati più “comode” di matlab in modo da processarle come vettori di caratteristiche. Le immagini caricate saranno memorizzate in una matrice 60000x784, nella quale ogni colonna rappresenta un’immagine, e le righe i valori di intensità associati ai pixel. La funzione si basa a sua volta sulle funzioni realizzate dall’Università di Stanford e rese disponibili dagli stessi realizzatori.

```
function [immaginiCifre, etichetteCifre] =
caricaDataset(percorsoImmagini, percorsoEtichette)

immaginiCifre = (loadMNISTImages(percorsoImmagini))';

etichetteCifre = loadMNISTLabels(percorsoEtichette);
end

function images = loadMNISTImages(filename)

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2051, ['Bad magic number in ', filename, '']);

numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
numCols = fread(fp, 1, 'int32', 0, 'ieee-be');

images = fread(fp, inf, 'unsigned char');
images = reshape(images, numCols, numRows, numImages);
images = permute(images, [2 1 3]);

fclose(fp);

images = reshape(images, size(images, 1) *
size(images, 2), size(images, 3));

images = double(images) / 255;

end

function labels = loadMNISTLabels(filename)

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2049, ['Bad magic number in ', filename, ']);
```

```

numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');

labels = fread(fp, inf, 'unsigned char');

assert(size(labels,1) == numLabels, 'Mismatch in label count');

fclose(fp);
end

```

2.2 Creazione set

Lo script “creaSet” è stato utilizzato per generare i sottoinsiemi di *training*, *validation* e *test* del dataset MNIST. Prende in input la dimensione dell’insieme che si vuole generare, una struttura chiamata *indiceElemUtilizzati* e due insiemi *matEtichette* e *matImmagini* contenenti immagini e relative etichette del MNIST. Restituisce in output l’insieme delle immagini e relative etichette scelte, più la struttura *indiceElemUtilizzati* aggiornata. Ci si è orientati verso la creazione di un insieme di elementi uniformemente distribuiti, ottenendo, all’interno dell’insieme finale, 10 sottoinsiemi (uno per ogni cifra) della stessa grandezza. In questo modo la probabilità di estrazione di una cifra dall’insieme è uguale a tutte le altre. Tra le strutture utilizzate *contCifre* è un array di interi da 10 elementi, ognuno dei quali indica quante cifre sono presenti all’interno dell’insieme per la relativa cifra. Durante l’iterazione che va a riempire l’insieme, è condizione necessaria che il numero di elementi inseriti, per una cifra pescata casualmente, sia inferiore a quello determinato dal partizionamento.

```

while contElementi < dimensioneSet
    % Prendo una posizione casuale all'interno del dataset
    randPos=floor((60000-1).*rand(1)+1);
    % Controllo che l'elemento non sia già stato inserito
    if indiceElemUtilizzati(randPos)==0
        % Controllo quanti elementi di una cifra ho inserito
        if contCifre(matEtichette(randPos)+1)<partizioneCifre
            % Aggiorni contatori e strutture
            contCifre(matEtichette(randPos)+1)=
                contCifre(matEtichette(randPos)+1)+1;
            contElementi=contElementi+1;
            indiceElemUtilizzati(randPos)=1;
            % Posso inserire questo elemento nell'insieme
            setEtichette(contElementi,matEtichette(randPos)+1)=1;
            setImmagini(contElementi,:)=matImmagini(randPos,:);
        end
    end
end

```

Un’altra struttura degna di approfondimento è *indiceElemUtilizzati*, un array da 60’000 elementi (tanti quanti ne ha il MNIST) che tiene traccia degli elementi già pescati dal dataset, spostando un bit di attivazione da 0 a 1 nel momento in cui l’elemento viene inserito nell’insieme finale. Con questo meccanismo si evita di utilizzare più volte uno stesso elemento, e si è sicuri di utilizzare ogni elemento una sola volta.

2.3 Creazione rete neurale FFML

“creaReteFFML” è lo script utilizzato per la creazione della struttura che rappresenta una rete neurale feed-forward multi-layer. Sono richieste in input le informazioni riguardanti:

- il numero di nodi del livello di input,
- il numero di nodi del livello di output,
- un puntatore alla funzione di attivazione usata per i nodi di output,

- una struttura che contiene per ogni sua istanza il numero di nodi e la funzione di attivazione di un livello hidden,
- il valore minimo e il valore massimo che può assumere un peso generato randomicamente.

| | |
|------------------------|----------------|
| numNodiInput | 784 |
| numNodiOutput | 10 |
| numLivelliHidden | 1 |
| funDiAttivazioneOutput | @funIdentita |
| numLivelli | 3 |
| m | 180 |
| b | 1x2 cell |
| W | 1x2 cell |
| g | 1x2 cell |
| X | 250x784 double |
| a | 1x2 cell |
| z | 1x2 cell |
| delta | 1x2 cell |

Figura 4: Esempio struttura rete.

La struttura finale della rete neurale è rappresentata dalla figura 4. Nel caso d'esempio è rappresentata la struttura di una rete a tre livelli che riceve in input 250 elementi del MNIST e li classifica, impostando ad 1 uno dei dieci nodi di uscita e zero gli altri.

- numNodiInput contiene il numero di *nodi del primo livello*, quello di input;
- numNodiOutput contiene il numero di *nodi dell'ultimo livello*, output;
- numLivelliHidden contiene il numero di livelli diversi da quelli di input e output;
- funDiAttivazioneOutput contiene il puntatore alla *funzione d'attivazione dell'ultimo livello*, in questo caso una funzione identità;
- numLivelli indica il numero di *livelli totali della rete*: input, hidden e output;
- m indica il numero di *nodi dei livelli hidden*, nel caso d'esempio c'è un solo livello hidden e quindi un solo intero, in altri casi ci sarà una struttura contenente per ogni livello hidden il relativo numero di nodi;
- b contiene la struttura per i *bias*, in questo caso, essendoci tre livelli e due strati di pesi, i bias sono memorizzati in una struttura da due elementi, il primo contiene i bias del primo strato di pesi, il secondo quelli dell'ultimo. In particolare il primo sarà una struttura da 180 elementi, mentre il secondo da 10;
- W contiene gli strati di pesi, nel caso d'esempio il primo elemento della struttura avrà 784x180 elementi (connessioni da strato input a hidden) mentre la seconda 180x10 (connessioni da strato hidden ad output);
- g contiene le funzioni di attivazione dei livelli;
- X rappresenta l'input della rete, in questo caso sono stati forniti alla rete 250 elementi e visto che gli elementi hanno lunghezza 784 ci saranno 250 istanze da 784 unità;
- a contiene l'input passato ai singoli nodi della rete, in questo caso conterrà due strutture da 250x180 e 250x10;
- z contiene l'output del singolo nodo ottenuto attraverso la funzione di attivazione, conterrà quindi due strutture da 250x180 e 250x10;
- delta contiene i δ utilizzati per la backpropagation.

Il codice della funzione è il seguente:

```

function reteNeurale = creaReteFFML(numNodiInput,numNodiOutput,
funDiAttivazioneOutput,strutturaLivelliHidden,minPeso,
maxPeso,numLivHidden)

% Viene uniformata la rappresentazione della matrice
if size(strutturaLivelliHidden,1)>size(strutturaLivelliHidden,2)
    strutturaLivelliHidden=strutturaLivelliHidden';
end

% Inizializzazione struttura output
reteNeurale.numNodiInput=numNodiInput;
reteNeurale.numNodiOutput=numNodiOutput;
reteNeurale.numLivelliHidden=numLivHidden;
reteNeurale.funDiAttivazioneOutput=funDiAttivazioneOutput;
reteNeurale.numLivelli=size(strutturaLivelliHidden,2)+2;
for i=1:numLivHidden
    reteNeurale.m(i)=strutturaLivelliHidden(i).dimLivello;
end

% Generazione casuale dei pesi
% *** PRIMO LIVELLO ***
reteNeurale.b{1}=(maxPeso-minPeso).*rand(1,reteNeurale.m(1))+minPeso;
reteNeurale.W{1}=(maxPeso-minPeso).*rand(reteNeurale.m(1),
numNodiInput)+minPeso;
reteNeurale.g{1}=strutturaLivelliHidden(1).funDiAttivazione;
% Generazione casuale dei pesi
% *** LIVELLI HIDDEN ***
if reteNeurale.numLivelliHidden>=2
    for i=2:reteNeurale.numLivelliHidden
        reteNeurale.b{i}=(maxPeso-minPeso).*rand(1,reteNeurale.m(i))+minPeso;
        reteNeurale.W{i}=
        (maxPeso-minPeso).*rand(reteNeurale.m(i),
        reteNeurale.m(i-1))+minPeso;
        reteNeurale.g{i}=strutturaLivelliHidden(i).funDiAttivazione;
    end
end
% Generazione casuale dei pesi
% *** ULTIMO LIVELLO ***
reteNeurale.b{reteNeurale.numLivelli-1}=
(maxPeso-minPeso).*rand(1,numNodiOutput)+minPeso;
reteNeurale.W{reteNeurale.numLivelli-1}=
(maxPeso-minPeso).*rand(numNodiOutput,
reteNeurale.m(reteNeurale.numLivelliHidden))+minPeso;
reteNeurale.g{reteNeurale.numLivelli-1}=funDiAttivazioneOutput;
end

```

2.4 Funzioni di errore

Per il calcolo dell'errore, essenziale per il metodo della discesa del gradiente, sono state implementate due diverse funzioni. A seconda del tipo di problema affrontato, classificazione o regressione, è stata utilizzata la funzione di errore appropriata. Tra le scelte implementative, per rendere più generica la funzione, si è scelto di aggiungere

una flag per il calcolo della derivata, qualora tale flag sia impostata come vera, non verrà calcolata la funzione standard ma la sua derivata.

2.4.1 Somma dei quadrati

La funzione somma dei quadrati è una funzione di errore utilizzata principalmente in problemi di regressione, quei problemi in cui l'obiettivo è quello di approssimare il meglio possibile una determinata funzione matematica. Tale funzione, applicata ad un insieme di elementi, è definita come:

$$E^n = \sum_{k=1}^c \frac{1}{2} \cdot (y_k^n - b_k^n)$$

mentre la derivata rispetto all'ultimo nodo y_k si calcola come

$$\frac{\delta E^n}{\delta y_k} = y_k - t_k$$

la funzione è stata implementata in questo modo:

```
function z = funSommaQuadrati(x,y,daDerivare)
    if (exist('daDerivare','var'))
        z=x-y;
    else
        z=0.5*sum(((x-y).^2));
    end
end
```

2.4.2 Cross entropy

La cross entropy si definisce come:

$$E = \sum_{k=1}^N E^N \quad \text{con} \quad E^N = - \sum_{k=1}^c t_k^n \cdot \log(y_k^n)$$

è una funzione di errore utilizzata principalmente per problemi di classificazione, cioè quei problemi nei quali si deve assegnare ogni elemento dell'input X^N ad una specifica classe contenuta nell'insieme delle classi C . In questi problemi, l'output della funzione può essere visto come la probabilità $P(c | x)$, la probabilità che un certo input x appartenga alla classe c . A tale scopo all'output della rete viene applicata la funzione softmax nel caso la relativa flag sia attivata, tale funzione è definita come:

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

inoltre quando si va a calcolare la derivata della cross entropy con softmax la formula risultante diventa:

$$\frac{\delta E^n}{\delta y_k} = y_k - t_k$$

segundo le formule elencate l'implementazione è stata la seguente:

```
function z = funCrossEntropy(x,y,daDerivare)
    if exist('daDerivare','var')
        z=x-y;
    else
        y(x>0)=y(x>0) .* log(x(x>0));
        y(x==0)=y(x==0)*(-708);
        z=-sum(y);
    end
end
```

2.5 Funzioni di attivazione

Come funzioni di attivazione per i nodi degli strati di input, hidden e di output, sono state testate diverse configurazioni che hanno richiesto l'implementazione di tre funzioni differenti. Come per le funzioni di errore, anche in questo caso si è scelto di utilizzare una flag per decidere se calcolare la funzione o la sua derivata. In questo modo si è realizzato solo uno script per ognuna delle funzioni.

2.5.1 Sigmoid

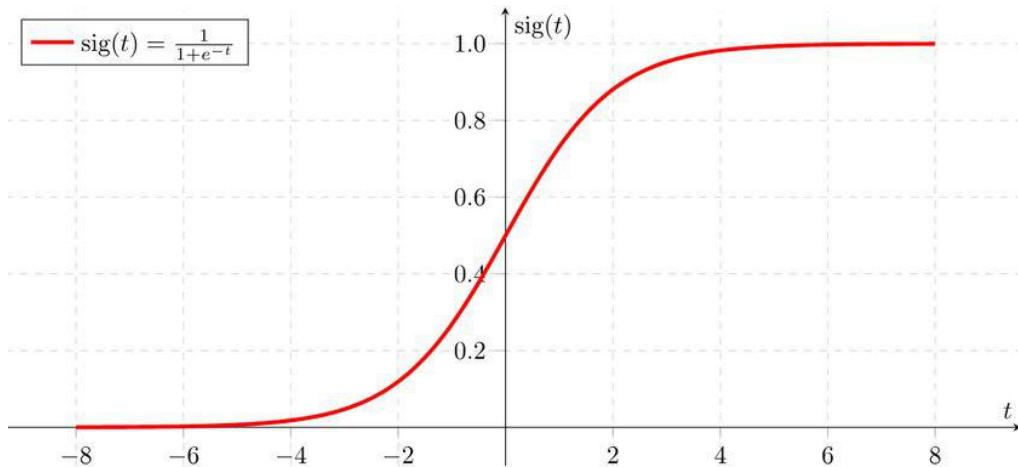


Figura 5: Grafico della funzione Sigmoid

La funzione sigmoid è stata utilizzata come funzione di attivazione per i nodi hidden della rete di classificazione, mentre per l'autoencoder è stata utilizzata come funzione di attivazione dei nodi dello strato di output. La funzione è stata implementata come segue:

```
function z = funSigmoide(x,daDerivare)
    if exist('daDerivare','var')
        z=funSigmoide(x).*(1-funSigmoide(x));
    else
        z=1.0 ./ (1.0+exp(-x));
    end
end
```

2.5.2 Identità

La funzione identità, utilizzata come funzione di attivazione dei nodi del livello di output della rete di classificazione, è stata implementata come segue:

```
function z = funIdentita(x,daDerivare)
    if exist('daDerivare','var')
        z=ones(size(x));
    else
        z=x;
    end
end
```

2.5.3 ReLU

La rete che costituisce il Denoising Autoencoder comprende tre livelli hidden come quella illustrata nell'articolo di riferimento. Trattandosi, quindi, di deep learning si è deciso di utilizzare come funzione di attivazione per i livelli hidden la funzione ReLU (Rectified Linear Unit). Questa funzione è definita come $f(x) = \max(0, x)$ e viene utilizzata per ridurre il fenomeno del *vanishing gradient* presente nelle reti deep; in questi casi, infatti, aumentando gli strati della rete si facilita la ricerca della soluzione ma, allo stesso tempo, aumentano i minimi locali. La ReLU aiuta a gestire questo problema, la funzione, infatti, presenta un andamento di questo tipo:

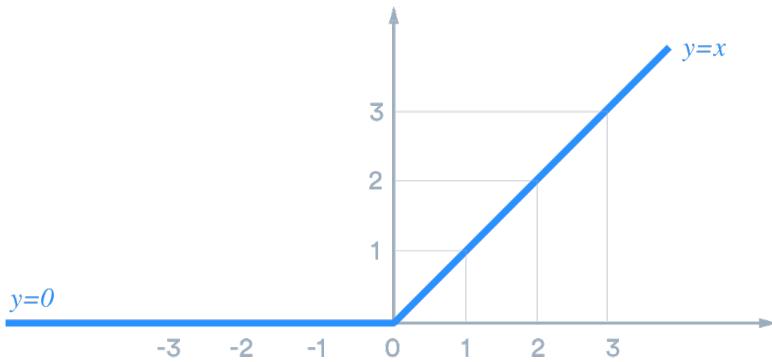


Figura 6: Grafico della funzione ReLU

La derivata della funzione è $= 0$ per $x < 0$ e non va mai a 0 per $x > 0$. Su matlab il tutto è stato implementato in questo modo:

```
function out = ReLU(x,daDerivare)
    if exist('daDerivare','var')
        if x < 0
            out=0;
        else
            out=1;
        end
    else
        if x < 0
            out=0;
        else
            out=x;
        end
    end
end
```

2.6 Addestramento

Per l'addestramento delle reti si è utilizzata la tecnica della discesa del gradiente 2.7. Per questo scopo sono state realizzate diverse funzioni, la funzione principale `addestraRete` provvede ad iterare le diverse epoche di addestramento ed ad inizializzazione e gestione delle strutture necessarie per memorizzare i risultati ottenuti ad ogni epoca. In questo script viene gestito anche il criterio di *early stopping* delle epoche. Quando, per una certa soglia arbitraria di epoche, l'errore rispetto al validation set continua a crescere, viene interrotto prematuramente il ciclo delle epoche; in caso questo non avvenga il processo terminerà ad un numero di epoche passato in input allo script. Di seguito viene riportato il codice della funzione:

```
function [reteNeurale,arrayErroriTTS,arrayErroriVTS] =
addestraRete(reteNeurale,epoche,tipoaddestramento,trainingSetImg,
```

```

validationSetImg , trainingSetLabel , validationSetLabel , funErrore ,
eta , flagSoftmax , fissapesi)

arrayErroriTTS=zeros(1,epoche);
arrayErroriVS=zeros(1,epoche);

epocheNecessarie=floor(epoche/3);
reteNeuraleCandidata=reteNeurale;
erroreVS=realmax;

for e = 1:epoche
    tempReteNeurale=reteNeurale;
    switch tipoAddestramento
        case 'batch'
            [reteNeurale , arrayErroriTTS(e) , arrayErroriVS(e)]=
            addestramentoBatch(reteNeurale , trainingSetImg ,
            validationSetImg , trainingSetLabel , validationSetLabel ,
            funErrore , eta , flagSoftmax , fissapesi);
        case 'online'
            [reteNeurale , arrayErroriTTS(e) , arrayErroriVS(e)]=
            addestramentoOnline(reteNeurale , trainingSetImg ,
            validationSetImg , trainingSetLabel , validationSetLabel ,
            funErrore , eta , flagSoftmax);
        otherwise
            error('Tipo di addestramento supportato: [batch] [online]');
    end
    contatoreErrore=0;

    if arrayErroriVS(e)<erroreVS
        contatoreErrore=0;
        erroreVS=arrayErroriVS(e);
        reteNeuraleCandidata=tempReteNeurale;
    else
        if e>=epocheNecessarie
            contatoreErrore=contatoreErrore+1;
            if contatoreErrore>30
                break;
            end
        end
    end
    fprintf("epoca corrente :%d\n",e);
end

reteNeurale=reteNeuraleCandidata;

if e<epoche
    arrayErroriTTS=arrayErroriTTS(1:e);
    arrayErroriVS=arrayErroriVS(1:e);
end
end

```

2.6.1 Addestramento batch

Il tipo di addestramento utilizzato, sia per la rete di classificazione sia per il Denoising Autoencoder, è stato quello batch. Con questo tipo di addestramento i pesi della rete vengono aggiornati solo alla fine del processo, dopo

aver calcolato l'errore. L'implementazione di questo tipo di addestramento è data dalla combinazione delle funzioni `addestraRete` e `addestramentoBatch`, la seconda provvede, inizialmente, ad effettuare una propagazione in avanti sulla rete, successivamente, a calcolare l'errore effettuato dalla rete e, infine, tramite backpropagation, vengono calcolate le derivate parziali con cui si aggiornano i pesi.

```

function [reteNeurale, erroreTS, erroreVS] = addestramentoBatch(reteNeurale,
    trainingSetImg, validationSetImg, trainingSetLabel, validationSetLabel,
    funErrore, eta, flagSoftmax, fissapesi)
%addestramentoBatch
% Addestra la rete neurale applicando un approccio di tipo batch, ovvero
% l'aggiornamento dei pesi avviene alla fine, dopo aver calcolato l'errore.
erroreTS=0;
erroreVS=0;

% Applicazione propagazione in avanti per training set e validation set
forwardPropTS=forwardProp(reteNeurale,trainingSetImg,flagSoftmax);
forwardPropVS=forwardProp(reteNeurale,validationSetImg,flagSoftmax);

% Calcolo errore training set e validation set, vengono divisi per la
erroreTS=sum(funErrore(forwardPropTS.z{forwardPropTS.numLivelliHidden+1},
    trainingSetLabel))/size(trainingSetImg,1);
erroreVS=sum(funErrore(forwardPropVS.z{forwardPropVS.numLivelliHidden+1},
    validationSetLabel))/size(validationSetImg,1);
% backpropagation su training set
forwardPropTS=backProp(forwardPropTS,trainingSetLabel,funErrore);
% Calcolo derivate parziali bias e pesi
[derBiasTS,derPesITS]=derivaPesi(forwardPropTS);
% Aggiorna pesi rete neurale
forwardPropTS=aggiornaPesi(forwardPropTS,derBiasTS,derPesITS,eta,fissapesi);

reteNeurale=forwardPropTS;
end

```

2.6.2 Addestramento autoencoder

Il denoising autoencoder è stato creato ed addestrato attraverso `addestraAutoEncoder`. Seguendo le linee guida dell'articolo, la funzione procede, prima, ad addestrare una rete autoencoder da un solo livello hidden, `autoEncoder1LV`, passandole il dataset con rumore aggiunto e le immagini originali come target. Successivamente, la funzione crea ed addestra un secondo autoencoder `autoEncoder2LV`, costituito da una rete deep con tre livelli hidden, uno di input e uno di output. I pesi ottenuti dall'addestramento del primo autoencoder vengono utilizzati per inizializzare il primo livello hidden del secondo autoencoder e vengono fissati durante il periodo di addestramento passando una specifica flag alla funzione che aggiorna i pesi. Il secondo autoencoder viene poi restituito allo script principale. Per entrambi gli autoencoder sono stati fatti test con o senza tecnica dei *pesi legati* anche se non sono state riscontrate particolari differenze dovute all'utilizzo o meno della tecnica. La configurazione che ha dato i migliori risultati è stata:

- addestramento batch;
- funzione di attivazione nodi hidden: ReLU;
- funzione di attivazione nodi output: Sigmide;
- funzione di errore: somma dei quadrati;
- epoche : 500;
- η autoencoder di inizializzazione = 0.0002;

- η autoencoder con tre livelli hidden = 0.00002;

La struttura degli autoencoder è descritta nella figura seguente:

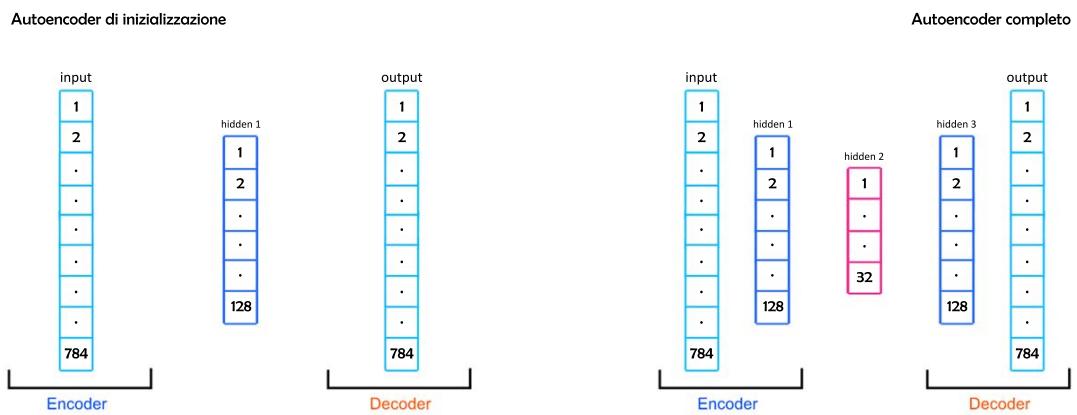


Figura 7: Struttura degli autoencoder realizzati, a sinistra quello ad 1 livello hidden, a destra quello completo. Entrambi gli autoencoder sono full-connected.

Codice della funzione:

```

function [autoEncoder2LV , AE2LVarrayErroriT , AE2LVarrayErroriVS] =
addestraAutoEncoder(trainingSetSporcato , validationSetSporcato ,
trainingSetImg , validationSetImg , testSetSporcato)

AE1LVfunNodiHidden = @ReLU;
AE1LVfunNodiOutput = @funSigmoide;
AE2LVfunNodiHidden = @ReLU;
AE2LVfunNodiOutput = @funSigmoide;
funErrore = @funSommaQuadrati;
AE1LVnodiHidden = 128;
AE2LVnodiHidden = 32;
AE1LVepoche = 500;
AE2LVepoche = 500;
tipoAddestramento = "batch";
AE1LVminPeso = -0.09;
AE1LVmaxPeso = 0.09;
AE2LVminPeso = -0.09;
AE2LVmaxPeso = 0.09;
AE1LVlvHidden = 1
AE2LVlvHidden = 3;
AE1LV_eta = 0.0002;
AE2LV_eta = 0.00002;
flagSoftmax = false;

% Creazione e inizializzazione dei livelli hidden del primo autoEncoder
strutturaLivelliHidden(1)=struct('dimLivello',AE1LVnodiHidden,
'funDiAttivazione',AE1LVfunNodiHidden);

% Creazione rete neurale feed-forward multi-layer
autoEncoder1LV = creaReteFFML(784,784,AE1LVfunNodiOutput ,

```

```

strutturaLivelliHidden ,AE1LVminPeso ,AE1LVmaxPeso ,AE1LVlvHidden);
autoEncoder1LV.W{2}=autoEncoder1LV.W{1}';

% Addestramento autoencoder AE1LV

[autoEncoder1LV , arrayErroriTS , arrayErroriVS] =
addestraRete(autoEncoder1LV ,AE1LVepoch
, tipoAddestramento ,trainingSetSporcato ,validationSetSporcato ,
trainingSetImg ,validationSetImg ,funErrore ,AE1LV_eta ,flagSoftmax ,false);

% creo autoencoder a 2 lv
% Creazione e inizializzazione dei livelli hidden
AE2LVstrutturaLivelliHidden(1)=struct('dimLivello',
AE1LVnodiHidden , 'funDiAttivazione' ,AE2LVfunNodiHidden);
AE2LVstrutturaLivelliHidden(2)=struct('dimLivello',AE2LVnodiHidden ,
'funDiAttivazione' ,AE2LVfunNodiHidden);
AE2LVstrutturaLivelliHidden(3)=struct('dimLivello',AE1LVnodiHidden ,
'funDiAttivazione' ,AE2LVfunNodiHidden);

% Creazione rete neurale feed-forward multi-layer
autoEncoder2LV = creaReteFFML(784,784,AE2LVfunNodiOutput ,
AE2LVstrutturaLivelliHidden ,AE2LVminPeso ,AE2LVmaxPeso ,AE2LVlvHidden);
% recupero pesi autoencoder 1 lv
autoEncoder2LV.W{1}=autoEncoder1LV.W{1};
% pesi legati
autoEncoder2LV.W{3}=autoEncoder2LV.W{2}';
autoEncoder2LV.W{4}=autoEncoder2LV.W{1}';
% addestro autoencoder 2LV
[autoEncoder2LV , AE2LVarrayErroriTS , AE2LVarrayErroriVS] =
addestraRete(autoEncoder2LV ,AE2LVepoch ,tipoAddestramento ,
trainingSetSporcato ,validationSetSporcato ,trainingSetImg ,
validationSetImg , funErrore ,AE2LV_eta ,flagSoftmax ,true);
end

```

Di seguito sono riportati i grafici relativi all'errore calcolato durante la fase di addestramento dell'autoencoder su validation set e training set, su training set da 1000 immagini e da 10000 immagini e 500 epoches:

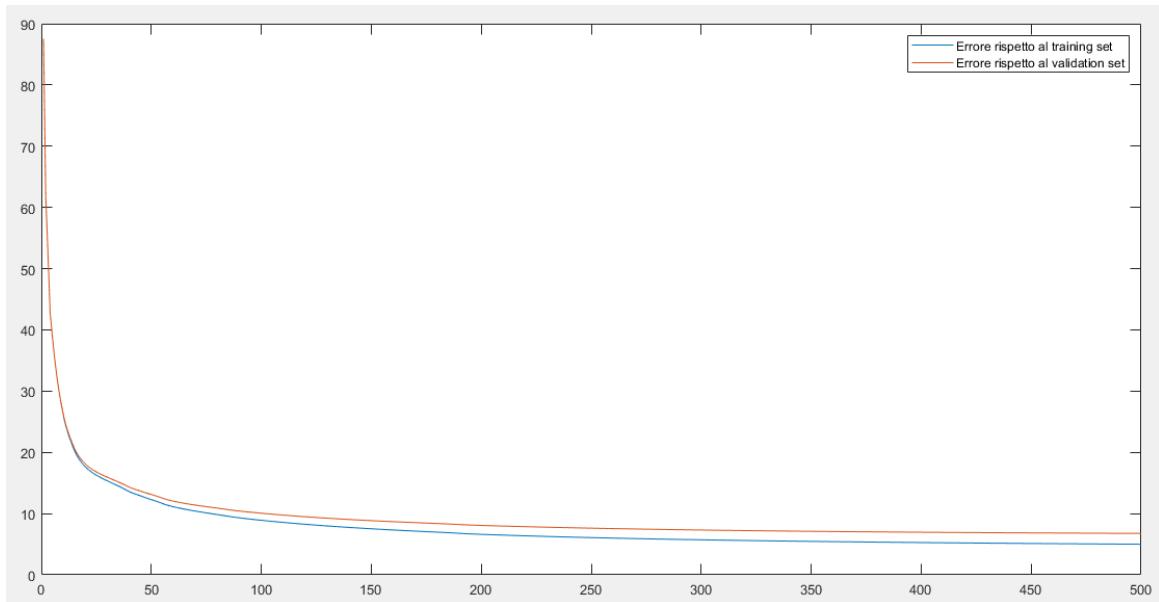


Figura 8: Errore di ricostruzione nell’autoencoder, su training set da 1000 immagini, all’aumentare delle epoche.

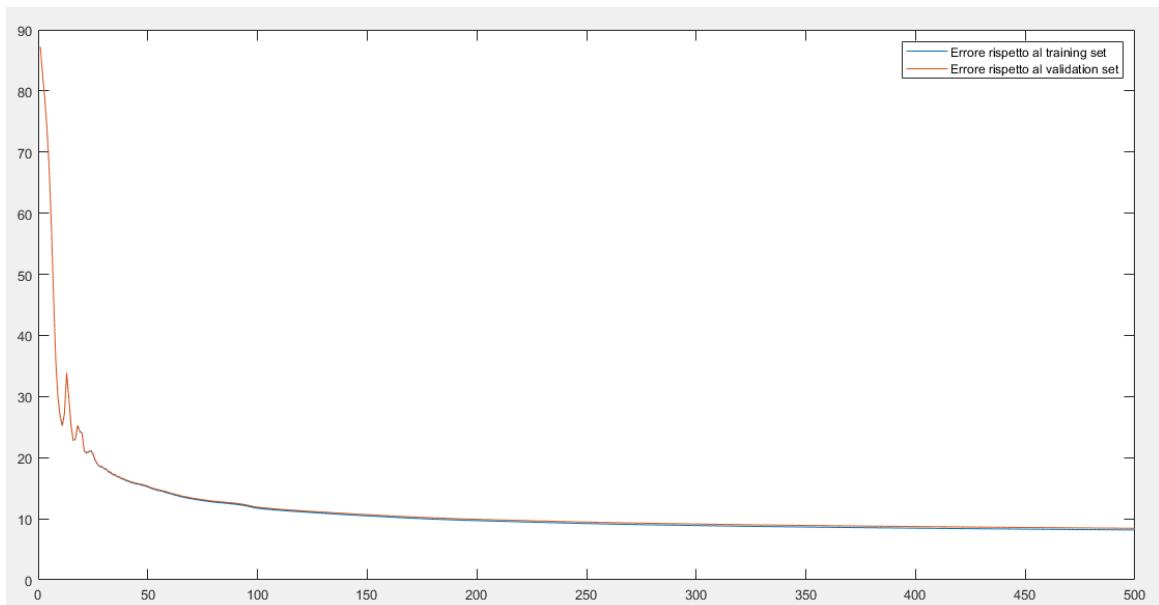


Figura 9: Errore di ricostruzione nell’autoencoder, su training set da 10000 immagini, all’aumentare delle epoche.

2.7 Discesa del gradiente

L’algoritmo della discesa del gradiente si basa sull’idea di minimizzare il valore della funzione di errore sull’output della rete rispetto a quello atteso. Tale errore, una volta fissato l’input, dipende soltanto da pesi e bias, bisogna quindi aggiornare questi valori cercando di raggiungere un minimo nella funzione di errore. La procedura, implementata nei diversi algoritmi di addestramento 2.6, calcola per ogni epoca l’output della rete, valuta l’errore e la derivata della funzione di errore rispetto a pesi e bias; modifica pesi e bias cercando di spostarsi sulla curva dell’errore verso un minimo, questo comporta che la funzione di errore debba essere continua e

derivabile. Inoltre, la derivata della funzione di errore rispetto ai pesi ci fornisce informazioni riguardo al verso dello scostamento della funzione rispetto al minimo, per questo motivo lo scostamento viene moltiplicato per un coefficiente η che serve a regolare la distanza percorsa sulla funzione ad ogni iterazione. Un η basso permetterà di evitare scostamenti troppo lunghi che potrebbero far allontanare dal minimo. In formule, considerando il singolo peso w_{ij} il Δ_{ij} viene calcolato come:

$$\Delta_{ij} = \frac{\delta E}{\delta w_{ij}} \cdot \eta$$

e l'aggiornamento dei pesi avviene come: $w_{ij}^t = w_{ij}^{t-1} - \Delta_{ij}$ con t epoca corrente.

2.7.1 Backpropagation

La backpropagation è una tecnica che permette di calcolare le derivate della funzione di errore rispetto ai valori di attivazione, questi valori risultano poi necessari per calcolare la derivata della funzione di errore rispetto ai pesi e ai bias. Le derivate intermedie calcolate dalla backpropagation vengono indicate con δ e sono definite in modo ricorsivo come:

$$\begin{aligned}\delta_i^L &= g'_L(a_i^L) \cdot E' z_i^L, t_i \\ \delta_i^l &= g'_l(a_i^l) \cdot \delta_{l+1} W_{l+1}\end{aligned}$$

Con L ultimo strato della rete, ed $l \neq L$ strato diverso dall'ultimo.

Il codice della funzione segue i passi della formula sopra descritta, calcola prima i δ dell'ultimo strato utilizzando la prima formula e, successivamente, per tutti gli altri strati, utilizza il δ degli strati successivi per calcolare quello degli strati precedenti.

```
function reteNeurale = backProp(reteNeurale, T, funErrore)
% backProp

indiceStratoOutput=reteNeurale.numLivelliHidden+1;
reteNeurale.delta{indiceStratoOutput}=
.funDiAttivazioneOutput(reteNeurale.a{indiceStratoOutput},true).*%
funErrore(reteNeurale.z{indiceStratoOutput},T,true);
for i=indiceStratoOutput-1:-1:1
    prodDeltaPesi = reteNeurale.delta{i+1}*reteNeurale.W{i+1};
    %calcolo intermedio
    reteNeurale.delta{i} =
    reteNeurale.g{i}(reteNeurale.a{i},true) .* prodDeltaPesi;
end
end
```

2.7.2 Calcolo derivata pesi

Per calcolare le derivate di pesi e bias è stata realizzata la funzione `derivaPesi`, come si può notare dal breve codice, la funzione calcola le derivate dei pesi seguendo le seguenti formule:

$$D_{w^1} = \delta^{1T} X \text{ per lo strato iniziale}$$

$$D_{w^l} = \delta^{1T} z_{l-1} \text{ per lo strato } l$$

e quelle dei bias come:

$$D_{b^1} = \sum_n^N \delta_n^l \text{ per lo strato iniziale}$$

$$D_{bl} = \sum_n^N \delta_n^l \text{ per lo strato } l$$

La funzione prende in input: la rete neurale su cui è stata effettuato il calcolo della backpropagation e restituisce in output due vettori contenenti derivate di pesi e bias.

```
function [derBias,derPesi] = derivaPesi(reteNeuraleBP)
% derivaPesi
    derBias{1}=sum(reteNeuraleBP.delta{1},1);
    derPesi{1}=reteNeuraleBP.delta{1}' * reteNeuraleBP.X;

    for i=2 : reteNeuraleBP.numLivelliHidden+1
        derBias{i}=sum(reteNeuraleBP.delta{i},1);
        derPesi{i}=reteNeuraleBP.delta{i}' * reteNeuraleBP.z{i-1};
    end
end
```

2.7.3 Aggiornamento dei pesi

Come già detto nella sezione 2.7 l'aggiornamento dei pesi avviene come $w_{ij}^t = w_{ij}^{t-1} - \Delta_{ij}$ con t epoca corrente. Di seguito viene riportato il codice della funzione `aggiornaPesi` implementata per applicare l'aggiornamento come descritto dalla formula. Data la necessità di fissare, durante l'addestramento del secondo autoencoder, i pesi ottenuti dal primo 2.6.2; è stata utilizzata una flag in modo da annullare l'aggiornamento per il primo livello. La funzione prende in input la rete neurale, le derivate di bias e pesi, la costante η e la flag sopra citata.

```
function reteNeurale = aggiornaPesi(reteNeurale,derBias,derPesi,
eta,fissaPesiPrimoLivello)
% AggiornaPesi

for i=1:reteNeurale.numLivelliHidden+1
    if fissaPesiPrimoLivello
        if (i ~= 1)
            reteNeurale.b{i}=reteNeurale.b{i}-(eta*derBias{i});
            reteNeurale.W{i}=reteNeurale.W{i}-(eta*derPesi{i});
        end
    else
        reteNeurale.b{i}=reteNeurale.b{i}-(eta*derBias{i});
        reteNeurale.W{i}=reteNeurale.W{i}-(eta*derPesi{i});
    end
end
end
```

2.8 Forward Propagation

L'output della rete neurale feed-forward realizzata per la classificazione è stato calcolato tramite la funzione “forwardProp”. La funzione prende in input:

- la rete neurale realizzata;
- la matrice X contenente N elementi aventi ciascuno F caratteristiche (i pixel delle immagini);
- una flag per decidere se utilizzare o meno la normalizzazione softmax.

Per calcolare l'output della rete la funzione memorizza nei livelli interni i valori di attivazione e le uscite di ogni nodo utilizzando due cell array, a e z. Il primo, “a”, conterrà la somma di tutti i valori in ingresso al nodo moltiplicati per i relativi pesi e sommati ai bias. Il secondo, “z”, conterrà il valore ottenuto applicando la

funzione di attivazione al contenuto di a . Consideriamo $X = (x_1, \dots, x_d)$ un unico elemento di input, il valore di attivazione degli m_1 nodi del primo strato sarà $a^1 = (a_1^1, \dots, a_m^1)$ e verrà calcolato come:

$$a_i^1 = \sum_{k=1}^d (x_k \cdot w_{ik}^1) + b_i$$

e nell'implementazione avremo $a^1 = x \cdot W^{1T} + b^1$ con b^1 vettore dei bias per il primo strato e X^T matrice del primo strato di pesi. Si può generalizzare la formula per tutti i livelli, in questo caso si ha $a^l = z^l - 1 \cdot W^{1T} + b^l$. La matrice X passata alla funzione viene assegnata come input per la rete, successivamente, per ogni livello hidden, viene effettuato il calcolo basato sulla formula precedente, al termine della funzione la struttura della rete neurale conterrà per ogni livello i dati di X, a, z, b , in particolare l'output della forward propagation si troverà nell'ultimo strato della matrice z .

Codice relativo al metodo:

```
function reteNeurale = forwardProp(reteNeurale, X, flagSoftmax)
% forwardProp

% Inizializzo strato di input
reteNeurale.X=X;
% Array dei valori di output del livello precedente
zPrec=X; % Al primo livello coincide con l'input

% Per ogni livello (escluso input) viene calcolato l'input e l'output del
% livello
for i=1 : reteNeurale.numLivelliHidden+1

    reteNeurale.a{i}=(zPrec*reteNeurale.W{i})';
    reteNeurale.a{i}=reteNeurale.a{i}+reteNeurale.b{i};
    reteNeurale.z{i}=reteNeurale.g{i}(reteNeurale.a{i});
    zPrec=reteNeurale.z{i};
end

% Se e' attiva la flag softmax
if flagSoftmax
    % Il calcolo del softmax e' stato implementato seguendo le indicazioni
    % date durante il corso.
    softmax=exp(reteNeurale.z{reteNeurale.numLivelliHidden+1}) ./%
        sum(exp(reteNeurale.z{reteNeurale.numLivelliHidden+1}), 2);
    reteNeurale.z{reteNeurale.numLivelliHidden+1}=softmax;
end
```

2.9 Aggiunta rumore

Lo script ‘‘sporcaInput’’ è stato utilizzato per applicare del rumore ad un set di immagini passato in input. Un secondo parametro di input indica in che percentuale si vuole “disturbare” l’immagine, mentre un terzo sta ad indicare che tipo di rumore si vuole applicare. I tipi di rumore implementati sono quattro:

- Standard, è il tipo di rumore utilizzato nell’articolo [1];
- GaussianStandard, rumore gaussiano standard implementato dalla funzione `imnoise(–, ’gaussian’)` di matlab;
- GaussianManual, filtro gaussiano implementato senza utilizzare la funzione di matlab;
- SaltnPepper, rumore sale e pepe già implementato in funzione `imnoise`.

Con \tilde{x} si fa riferimento all'input x sporcato, ed è l'insieme restituito in output dopo l'elaborazione. Il numero di pixel da sporcare viene calcolato sulla base della percentuale passata in input e l'elaborazione viene determinata dalla stringa passata.

- Standard: per ogni immagine dell'insieme passato viene utilizzato un array di zero da 784 elementi, questo array serve a tenere traccia dei pixel sporcati. Per sporcare i pixel viene scelto casualmente un numero da 1 a 784 e quel pixel posto uguale a 0 se la corrispondente posizione nell'array è uguale a 0, altrimenti si genera un diverso numero. Si termina nel momento in cui il numero di pixel modificati è uguale alla percentuale precedentemente calcolata.
- GaussiaStandard: viene utilizzata la funzione `imnoise` per ogni elemento dell'insieme passando 'gaussian' come rumore.
- GaussianManual: i pixel che compongono l'immagine vengono modificati utilizzando il rumore gaussiano, che data la deviazione standard modifica i pixel aggiungendo all'intensità originale di ogni pixel la deviazione standard moltiplicata per un valore casuale compreso tra 0 e 1. Questo tipo di rumore non è stato testato perché si è preferito utilizzare il rumore gaussiano fornito da matlab.
- SaltNpepper: viene utilizzata la funzione `imnoise` per ogni elemento dell'insieme, parametrizzata per il rumore sale e pepe e come intensità di rumore un valore tale da disturbare l'immagine senza però comprometterla.

```

function setImgTilde = sporcaInput(setImgInput,percDistruzione,tiporumore)
    setImgTilde=setImgInput;
    numImmagini=size(setImgInput,1);
    numPxDaSporcare=floor((percDistruzione*784)/100);
    switch tiporumore
        case 'Standard'
            for i=1:numImmagini
                arrayPxSporcati=zeros(1,784);
                numPxSporcati=0;
                while numPxSporcati<numPxDaSporcare
                    randPx=floor((784-1).*rand(1)+1);
                    if arrayPxSporcati(randPx)==0
                        setImgTilde(i,randPx)=0;
                        arrayPxSporcati(randPx)=1;
                        numPxSporcati=numPxSporcati+1;
                    end
                end
            end
        case 'GaussianStandard'
            for i=1:numImmagini
                digit = reshape(setImgInput(i,:), [28,28]);
                digitTilde = imnoise(digit,'gaussian');
                setImgTilde(i,:) = reshape(digitTilde,[784,1]);
            end
        case 'GaussianManual'
            for i=1:numImmagini
                digit = reshape(setImgInput(i,:), [28,28]);
                digitTilde = double(digit) +
                (percDistruzione/1000)*randn(size(digit));
                setImgTilde(i,:) = reshape(digitTilde,[784,1]);
            end
    end
end

```

```

case 'SaltnPepper'
    for i=1:numImmagini
        digit = reshape(setImgInput(i,:), [28,28]);
        digitTilde = imnoise(digit,'salt & pepper',
            percDistruzione/1000);
        setImgTilde(i,:) = reshape(digitTilde,[784,1]);
    end
otherwise
    fprintf("\n il tipo di rumore richiesto non e' supportato,
non verrà applicato rumore al dataset");
end
end

```

2.10 Testing

Per testare la rete sono state eseguite due principali configurazioni di test, l'accuratezza per singolo esperimento è stata calcolata tramite una funzione specifica, mentre l'accuratezza media, accompagnata da deviazione standard, è stata calcolata ripetendo l'intero esperimento un certo numero di volte.

2.10.1 Test globale

Per eseguire l'intero esperimento, ed ottenere *accuratezza media* e *deviazione standard*, è stato realizzato uno script che ripete un determinato numero di volte un singolo test e, per ogni configurazione significativa, calcola media e deviazione standard dell'accuratezza ottenuta. Di seguito viene riportato l'algoritmo utilizzato per testare ogni configurazione, per evitare di appesantire la documentazione è stato riportato il codice solo di alcuni test, lo script completo verrà inserito nell'appendice finale di questo documento.

algoritmo:

```

// per ogni configurazione
inizializza a 0 Accuratezze,
inizializza a 0 AccuratezzeDatasetSporcato,
inizializza a 0 AccuratezzeDatasetRicostruito,
per ogni i da 0 fino a NumeroRipetizioni
//calcola accuratezze
testSingolaConfig(trainingSet, validationSet,
testSet, "Rumore selezionato", "intensità");
//Salva accuratezze nei relativi array
//riempie la matrice finale con media e deviazione standard
M(esecuzione,1) = mean(tempAccNorm);
M(esecuzione,2) = std(tempAccNorm);
M(esecuzione,3) = mean(tempAccSporc);
M(esecuzione,4) = std(tempAccSporc);
M(esecuzione,5) = mean(tempAccRicostruito);
M(esecuzione,6) = std(tempAccRicostruito);

```

Frammento del codice:

```

clearvars;
numeroRipetizioni = 5;
M = zeros(8,6);
%script che testa l'accuratezza della rete nelle diverse configurazioni
trainingSet = 1000;
validationSet = 250;
testSet = 250;
%senza rumore, standard 0

```

```

row=1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostituito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostituito] =
testSingolaConfig(trainingSet, validationSet,
testSet, "Standard", 0);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostituito(i) = AccRicostituito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostituito);
M(row,6) = std(tempAccRicostituito);

%e così via per tutte le configurazioni
.
.
.
.

%GaussianStandard
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostituito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostituito] =
testSingolaConfig(trainingSet, validationSet,
testSet, "GaussianStandard", 0);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostituito(i) = AccRicostituito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostituito);
M(row,6) = std(tempAccRicostituito);
%la matrice con i risultati viene salvata in un file .csv
csvwrite('AccuratezzeMedie.txt',M);

```

2.10.2 Test singola configurazione

Per testare l'accuratezza della rete su una specifica configurazione è stata realizzata la funzione `testSingolaConfig`, la funzione prende in input:

- dimensione del training set,
- dimensione del test set,
- dimensione validation set,

- tipo di rumore codificato da una stringa,
- v intensità del rumore.

La funzione, inizialmente, prepara le strutture necessarie per il processo di addestramento, carica il dataset MNIST, ne estrae una sotto-porzione per ottenere training set, validation set e test set; crea e addestra una rete neurale per la classificazione delle immagini del dataset e ne misura l'accuratezza. Successivamente, applica del rumore al dataset e misura l'accuratezza della rete nel classificare le immagini sottoposte al rumore. infine, crea ed addestra il denoising autoencoder per ricostruire immagini sottoposte a quello specifico rumore, anche in questo caso viene calcolata l'accuratezza nel classificare le immagini ricostruite tramite autoencoder. Di seguito viene riportato il codice della funzione:

```

function [accuratezzaClassPura ,
accuratezzaClassSenzaDenoising ,accuratezzaClassConDenoising] =
testSingolaConfig(dimensioneTrainingSet ,
dimensioneTestSet ,dimensioneValidationSet ,tipoRumore ,v)
tic
addpath('./risorse/');
Class_funNodiOutput = @funIdentita;
Class_funNodiHidden = @funSigmoid;
Class_funErrore = @funCrossEntropy;
Class_nodiHidden = 180;
Class_epoche = 300;
tipoAddestramento = "batch";
ClassminPeso = -0.09;
ClassmaxPeso = 0.09;
ClasslvHidden = 1;
Class_eta = 0.0004;
flagSoftmax = true;
[matImmagini , matEtichette] =
caricaDataset('./risorse/train-images-idx3-ubyte',
 './risorse/train-labels-idx1-ubyte');

dimensioneTrainingSet = floor(dimensioneTrainingSet/10)*10;
dimensioneTestSet = floor(dimensioneTestSet/10)*10;
dimensioneValidationSet = floor(dimensioneValidationSet/10)*10;

indiceElemUtilizzati=zeros(1,60000);
% Creazione training set
[trainingSetImg ,trainingSetLab ,indiceElemUtilizzati]=
creaSet(dimensioneTrainingSet ,
indiceElemUtilizzati , matEtichette ,matImmagini);
% Creazione validation set
[validationSetImg ,validationSetLab ,indiceElemUtilizzati]=
creaSet(dimensioneValidationSet ,
indiceElemUtilizzati , matEtichette ,matImmagini);
% Creazione test set
[testSetImg ,testSetLab ,indiceElemUtilizzati]=
creaSet(dimensioneTestSet ,
indiceElemUtilizzati , matEtichette ,matImmagini);
% Creazione e inizializzazione dei livelli hidden
strutturaLivelliHidden(1)=
struct('dimLivello',Class_nodiHidden,'funDiAttivazione',
Class_funNodiHidden);
% Creazione rete neurale feed-forward per la classificazione
reteNeuraleClass = creaReteFFML(784,10,Class_funNodiOutput ,
strutturaLivelliHidden ,ClassminPeso ,ClassmaxPeso ,ClasslvHidden);

```

```
% Addestramento rete neurale
fprintf("\n Inizio addestramento della rete di classificazione\n");
[reteNeuraleClass, arrayErroriT, arrayErroriVS] =
addestraRete(reteNeuraleClass, Class_epoche,
tipoAddestramento, trainingSetImg, validationSetImg,
trainingSetLab, validationSetLab, Class_funErrore, Class_eta,
flagSoftmax, false);
%propagazione in avanti utilizzando come input il test set
[reteNeuraleClass] = forwardProp(reteNeuraleClass, testSetImg, true);
%viene calcolata l'accuratezza della valutazione della rete rispetto al
%test set
[accuratezzaClassPura] = valutazioneRete(reteNeuraleClass.z
{reteNeuraleClass.numLivelliHidden+1}, testSetLab);
%Sporco training e validation set per addestrare l'autoencoder
trainingSetSporcato = sporcaInput(trainingSetImg, v, tipoRumore);
validationSetSporcato = sporcaInput(validationSetImg, v, tipoRumore);
testSetSporcato = sporcaInput(testSetImg, v, tipoRumore);
%costruzione e addestramento dell'autoencoder
[denoisingAutoEncoder, AE2LVarrayErroriT, AE2LVarrayErroriVS]=
addestraAutoEncoder(trainingSetSporcato, validationSetSporcato,
trainingSetImg, validationSetImg, testSetSporcato);
%test senza autoencoder
reteNeuraleClassTest = forwardProp(reteNeuraleClass, testSetSporcato, true);
%calcolo accuratezza senza autoencoder
[accuratezzaClassSenzaDenoising] = valutazioneRete(reteNeuraleClassTest.z
{reteNeuraleClassTest.numLivelliHidden+1}, testSetLab);
%l'input sporcato viene passato all'autoencoder e poi alla rete di
%classificazione
denoisingAutoEncoder = forwardProp(denoisingAutoEncoder,
testSetSporcato, false);
%classificazione dell'input ricostruito tramite denoising
reteNeuraleConDenoising = forwardProp(reteNeuraleClass,
denoisingAutoEncoder.z
{denoisingAutoEncoder.numLivelliHidden+1}, true);
%valutazione classificazione della reteNeuraleConDenoising
[accuratezzaClassConDenoising]=
valutazioneRete(reteNeuraleConDenoising.z
{reteNeuraleConDenoising.numLivelliHidden+1},
testSetLab);
end
```

2.10.3 Valutazione rete

‘‘valutazioneRete’’ è stato utilizzato per valutare l’accuratezza della rete neurale di classificazione. Dato un insieme di elementi da testare, sono necessari, come parametri di input, la risposta della rete neurale e l’insieme delle etichette usate come base di verità. I due insiemi non sono inizialmente confrontabili perché rappresentati differentemente: l’insieme delle etichette è composto da celle contenenti solo valori uguali a 1 o 0, la cella con valore 1 identifica il reale valore della cifra; l’insieme delle risposte della rete contiene valori compresi tra 0 e 1, e la cella con valore maggiore indica quale sia stata la risposta della rete. Per poter confrontare i due insiemi si è scelto di adattare la risposta della rete neurale allo standard usato nel dataset MNIST. Per farlo è stato utilizzato lo script ‘‘adattaRisposta’’ 2.10.4.

```
function [accuratezza] = valutazioneRete(Y,T)
if(size(Y,1) ~= size(T,1)) || (size(Y,2) ~= size(T,2))
```

```

        error("Le dimensioni dei parametri non coincidono");
    end
    rispostaClasse = adattaRisposta(Y);
    corrette = nnz(rispostaClasse .* T);
    accuratezza = corrette/size(Y,1);
end

```

Una volta ottenuti i due insiemi si procede ad effettuare una moltiplicazione elemento per elemento. Questo tipo di moltiplicazione restituirà 1 solo quando per un elemento entrambi gli 1 nei due insiemi di confronto sono nella stessa posizione, 0 altrimenti. Con la funzione `nnz` si vanno a contare il numero di 1 presenti nella matrice ottenuta, in altre parole le risposte corrette. Dividendo questo numero per il numero totale di elementi viene calcolata l'accuratezza della rete.

2.10.4 Gestione rappresentazione output

```

function [rispostaClasse] = adattaRisposta(Y)
    rispostaClasse = zeros(size(Y,1), size(Y,2));
    for i = 1 : size(Y,1)
        [~, argmax] = max(Y(i,:));
        rispostaClasse(i,argmax) = 1;
    end
end

```

Lo script ‘‘adattaRisposta’’ è stato utilizzato per adattare la risposta della rete neurale alla rappresentazione usata dal dataset. Prende in input l’ultimo livello della rete neurale di classificazione e ritorna in output una struttura della stessa dimensione, quindi dieci elementi, uno per ogni cifra. Inizialmente tutti gli elementi hanno valore 0, successivamente, scorrendo tutti gli elementi dell’output della rete, viene presa la posizione dell’elemento con valore maggiore, e impostata a 1 la medesima cella della struttura di output.

3 Test e Risultati

In questa sezione verranno illustrati i test effettuati ed i risultati ottenuti.

3.1 Struttura test

I test effettuati sono stati strutturati in questo modo: per ogni tipo di rumore implementato e per diverse intensità di ognuno, è stato eseguito un esperimento completo che comprende addestramento e valutazione dell’accuratezza della rete nel classificare le cifre del dataset MNIST, addestramento del Denoising Autoencoder e valutazione accuratezza della rete di classificazione su dataset con rumore e dataset ricostruito tramite autoencoder. I test sono stati eseguiti sia con dimensione del training set 1000 sia con dimensione 10000. Dati i lunghi tempi di esecuzione dell’esperimento, per ogni test le varie accuratezze e la deviazione standard sono state calcolate su 5 ripetizioni dell’esperimento, la rete di classificazione ha mantenuto ottimi risultati al crescere del training set, per il Denoising Autoencoder, invece, è stato necessario cambiare η al crescere del dataset. Le configurazioni che hanno dato i risultati migliori sono riportate nella tabella seguente:

| Iperparametro | Rete Classificazione | DnAutoencoder-TS 1000 | DnAutoencoder-TS 10000 |
|---------------------------------|----------------------|-----------------------|------------------------|
| Training Set | 1000—10000 | 1000 | 10000 |
| Validation Set | 250—2500 | 250 | 2500 |
| Test Set | 250—2500 | 250 | 2500 |
| Livelli Hidden | 1 | 3 | 3 |
| Nodi Hidden | 180 | 128—32—128 | 128—32—128 |
| Funzione attivazione hidden | Sigmoide | ReLU | ReLU |
| Funzione attivazione output | Identità | Sigmoide | Sigmoide |
| Funzione di errore | Cross Entropy | Somma Dei Quadrati | Somma Dei Quadrati |
| Intervallo dei pesi di partenza | [-0.9,+0.9] | [-0.9,+0.9]* | [-0.9,+0.9]* |
| Epoche | 300 | 500 | 500 |
| η | 0.0004 | 0.00002 | 0.000002 |

*il primo livello dell'autoencoder è stato inizializzato eseguendo prima una fase di training con 1 livello solo, i pesi per quel livello sono stati poi tenuti fissi durante l'addestramento con 3 livelli.

Fissati i migliori iperparametri, sono state testate diverse configurazioni di rumore e intensità, lo studio nell'articolo ha utilizzato un determinato tipo di rumore, durante il progetto si è pensato di testare il funzionamento del Denoising Autoencoder con altri tipi di rumore per osservarne il comportamento.

Configurazioni Testate:

- Nessun rumore;
- Rumore **Standard** con intensità 25%, cioè quello illustrato nel paper di riferimento, forza una percentuale dei pixel dell'immagine ad avere intensità 0;
- Rumore **Standard** con intensità 50%;
- Rumore **Standard** con intensità 75%;
- Rumore **Sale e Pepe** con intensità 25%;
- Rumore **Sale e Pepe** con intensità 50%;
- Rumore **Sale e Pepe** con intensità 75%;
- Rumore **Gaussiano** con intensità standard settata da Matlab.

I risultati ottenuti sono mostrati nella sezione [3.3](#)

3.2 Esempi di ricostruzione

In questa sezione verranno forniti esempi visivi delle ricostruzioni effettuate dall'autoencoder rispetto ai diversi tipi di rumore.



Figura 10: Esempio di ricostruzione immagini con intensità 25% e rumore standard

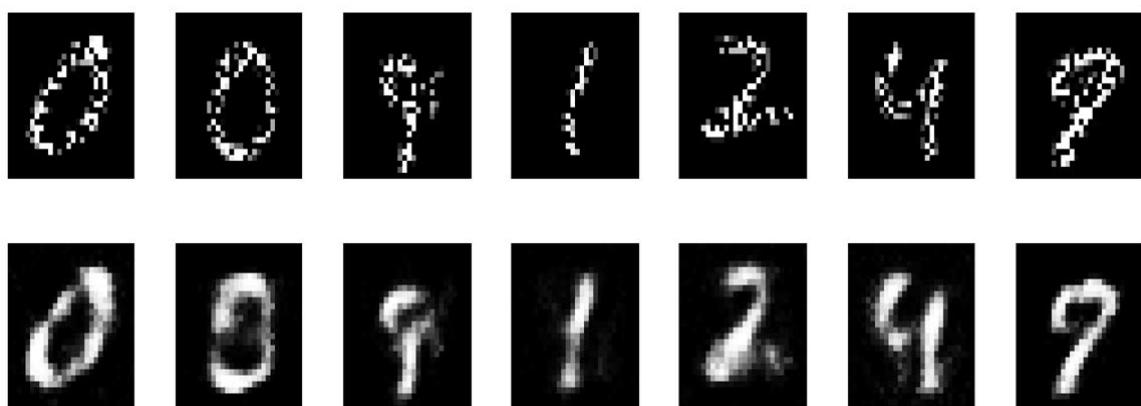


Figura 11: Esempio di ricostruzione immagini con intensità 50% e rumore standard

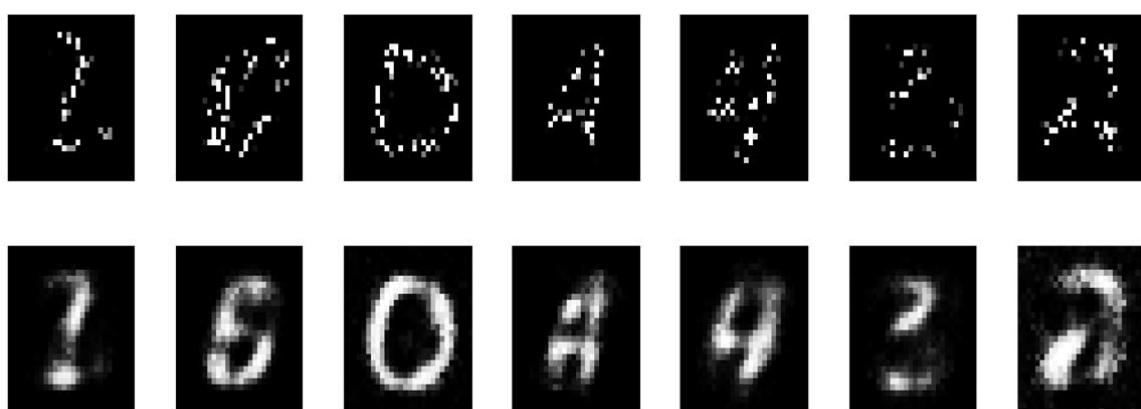


Figura 12: Esempio di ricostruzione immagini con intensità 75% e rumore standard

In figura 12 mostra come tramite denoising autoencoder si riesca a ricostruire l'immagine anche con intensità di rumore elevata.

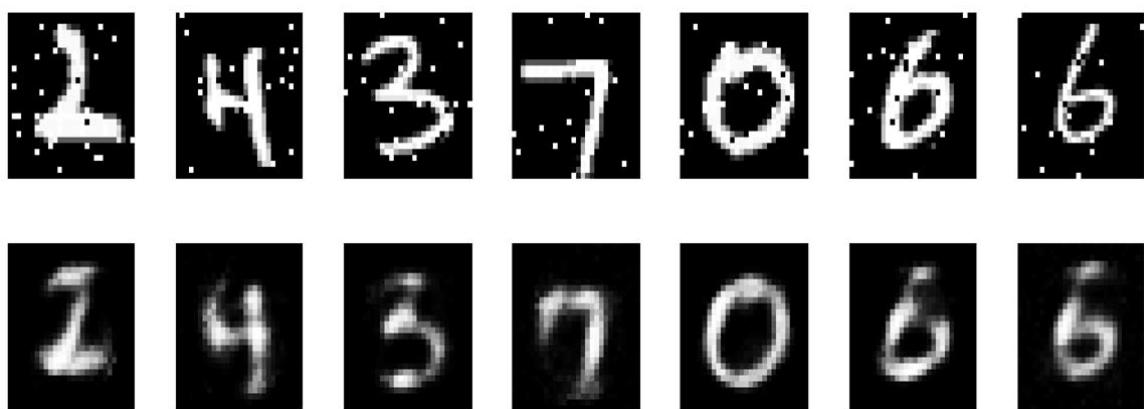


Figura 13: Esempio di ricostruzione immagini con intensità 50% e rumore sale e pepe

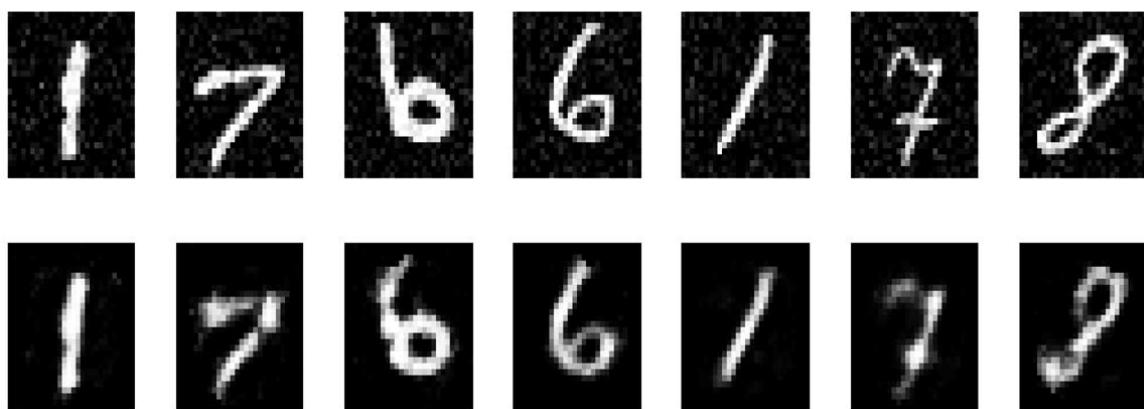


Figura 14: Esempio di ricostruzione immagini con intensità 50% e rumore gaussiano

Di seguito sono riportati anche esempi di ricostruzione con training set composto da 10000 immagini sottoposto a rumore basato sull'articolo di riferimento (standard).

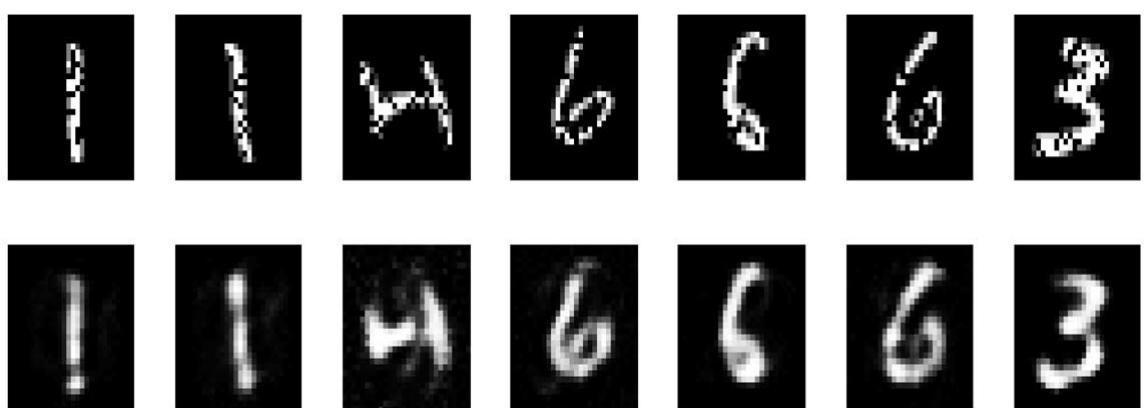


Figura 15: Esempio di ricostruzione immagini con training set da 10000 elementi sottoposto a rumore standard con intensità 25%



Figura 16: Esempio di ricostruzione immagini con training set da 10000 elementi sottoposto a rumore standard con intensità 50%

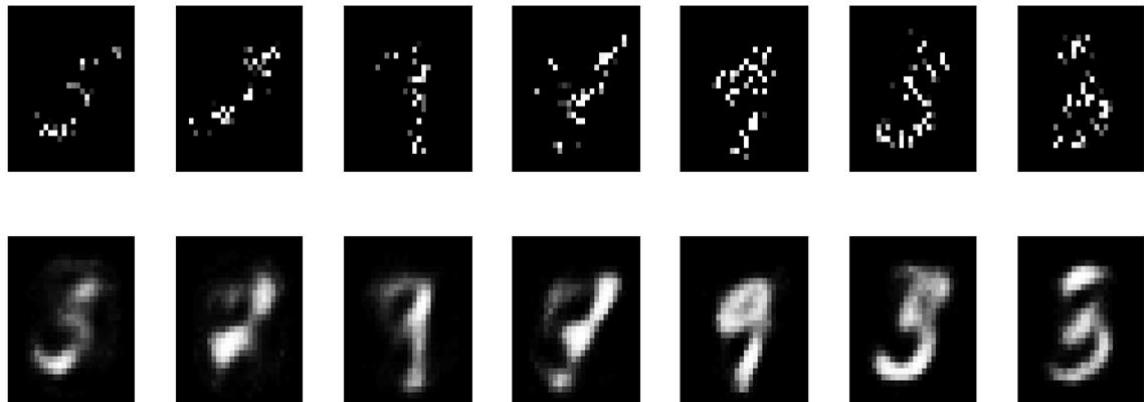


Figura 17: Esempio di ricostruzione immagini con training set da 10000 elementi sottoposto a rumore standard con intensità 75%

3.3 Valutazione accuratezza

Per tutte le configurazioni riportate nella sezione 3.1 sono state calcolate accuratezza media nella classificazione su 5 esecuzioni e la relativa deviazione standard, la tabella seguente riporta i risultati ottenuti:

| Accuracy classificatore | | 1'000 | | | | | | | | 10'000 | | | | | | | |
|-------------------------|------------------|--------|---------|-----------------|---------|-------------------|---------|--------|---------|-----------------|---------|-------------------|---------|-------|---------|-------|---------|
| | | MNIST | | MNIST Sporcatto | | MNIST Ricostruito | | MNIST | | MNIST Sporcatto | | MNIST Ricostruito | | | | | |
| Tipo rumore | Intensità rumore | Media | Dev std | Media | Dev std | Media | Dev std | Media | Dev std | Media | Dev std | Media | Dev std | Media | Dev std | Media | Dev std |
| NESSUNO | - | 87,20% | 0,012 | 87,20% | 0,012 | 84,88% | 0,01 | 90,80% | 0,015 | 90,80% | 0,015 | 88,44% | 0,01 | | | | |
| STANDARD | 25% | 88,24% | 0,0067 | 84,88% | 0,0201 | 84,24% | 0,0231 | 92,52% | 0,012 | 89,88% | 0,033 | 86,44% | 0,032 | | | | |
| | 50% | 85,84% | 0,0189 | 74,24% | 0,018 | 77,12% | 0,0322 | 92,40% | 0,009 | 79,12% | 0,024 | 81,20% | 0,024 | | | | |
| | 75% | 86,88% | 0,0252 | 47,92% | 0,0613 | 64,48% | 0,0232 | 92,96% | 0,019 | 43,36% | 0,061 | 69,96% | 0,026 | | | | |
| SALT AND PEPPER | 25% | 86,88% | 0,0325 | 87,44% | 0,0337 | 85,04% | 0,0341 | 93,60% | 0,0233 | 92,80% | 0,0262 | 91,00% | 0,029 | | | | |
| | 50% | 87,04% | 0,0342 | 85,92% | 0,0343 | 81,92% | 0,0371 | 92,20% | 0,0257 | 90,68% | 0,0321 | 88,20% | 0,041 | | | | |
| | 75% | 87,92% | 0,0148 | 86,32% | 0,0246 | 82,24% | 0,0326 | 93,80% | 0,0184 | 91,72% | 0,041 | 81,04% | 0,035 | | | | |
| GAUSS | - | 88,56% | 0,0159 | 88,88% | 0,0148 | 77,36% | 0,0213 | 93,60% | 0,03 | 92,32% | 0,015 | 79,48% | 0,019 | | | | |

Figura 18: Risultati test accuratezza

Come si nota dalla tabella, la classificazione senza rumore presenta risultati migliori al crescere del dataset, 88% circa con training set da 1000 e tra 90% e 92% con training set da 10000. Un altro dato fondamentale che si evince dalla tabella è che, applicando lo stesso tipo di rumore utilizzato dagli autori dell'articolo, la classificazione del dataset ricostruito migliora con intensità del rumore maggiore e uguale al 50%, mentre per gli altri tipi di rumore implementati, l'autoencoder non riesce a migliorare la classificazione della rete. I grafici successivi illustrano confronti significativi tra le diverse accuratezze ottenute:

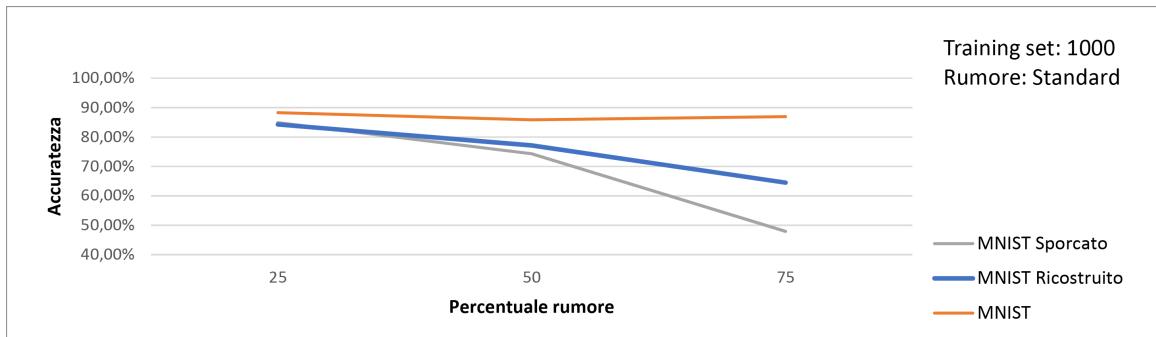


Figura 19: Accuratezza media valutata su training set da 1000 elementi con rumore standard ad intensità diverse.

Dal grafico si possono evincere diverse cose, in primis vediamo che la classificazione senza rumore presenta un'accuratezza costante (a meno di variazione nelle performance della rete nei singoli casi). Possiamo inoltre osservare come, quando l'intensità del rumore è al intorno al 25% le performance con o senza autoencoder sono simili. Risulta interessante, però, osservare come al crescere dell'intensità con cui viene applicato il rumore, si ottengono risultati risultati migliorati di circa il 20% utilizzando la rete con denoising autoencoder.

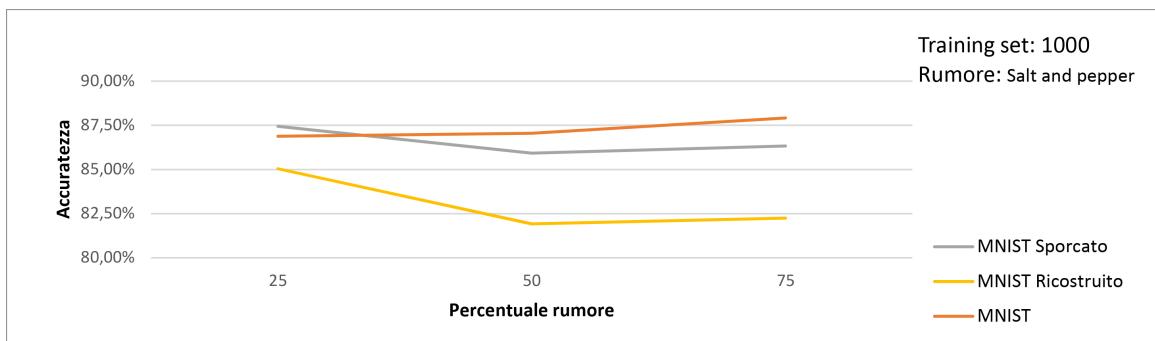


Figura 20: Accuratezza media valutata su training set da 1000 elementi con rumore sale e pepe ad intensità diverse.

A differenza del grafico precedente, possiamo notare come l'autoencoder non migliori le prestazioni in caso di rumore sale e pepe già partendo dal 25% di intensità.

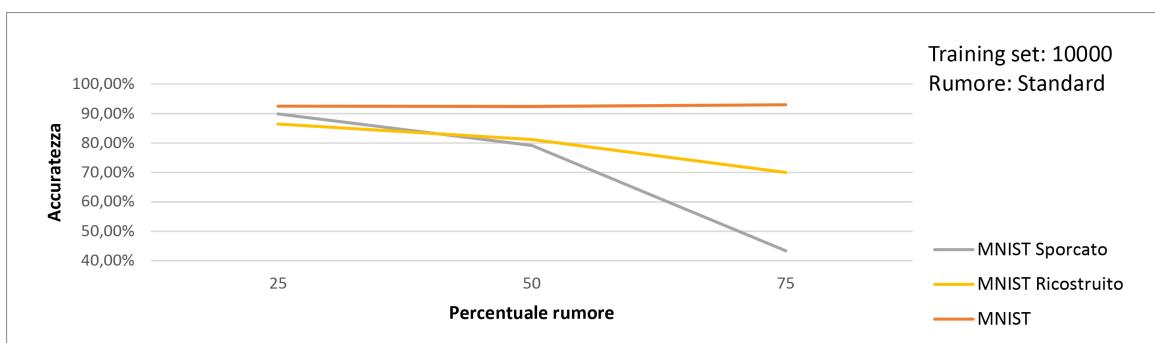


Figura 21: Accuratezza media valutata su training set da 10000 elementi con rumore standard ad intensità diverse.

Aumentando l'intensità del rumore sale e pepe vediamo come le prestazioni con l'autoencoder si riducono drasticamente. Allo stesso modo, nei grafici successivi osserviamo un risultato simile per il rumore sale e pepe al 75% di intensità e per il rumore gaussiano.

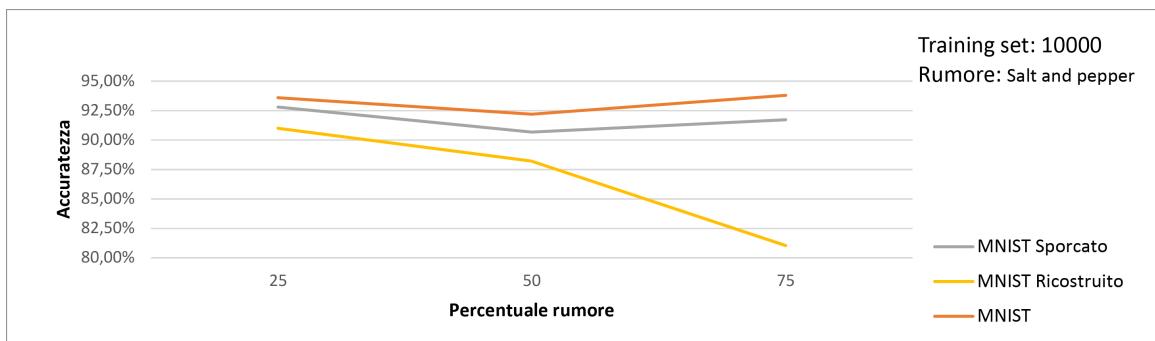


Figura 22: Accuratezza media valutata su training set da 10000 elementi con rumore sale e pepe ad intensità diverse.



Figura 23: Accuratezza media valutata su training set da 1000 elementi con rumore della funzione “*imnoise - gaussian*” di matlab.



Figura 24: Accuratezza media valutata su training set da 10000 elementi con rumore della funzione “*imnoise - gaussian*” di matlab.



Figura 25: Accuratezza media valutata su training set da 1000 elementi al quale non è stato applicato rumore.

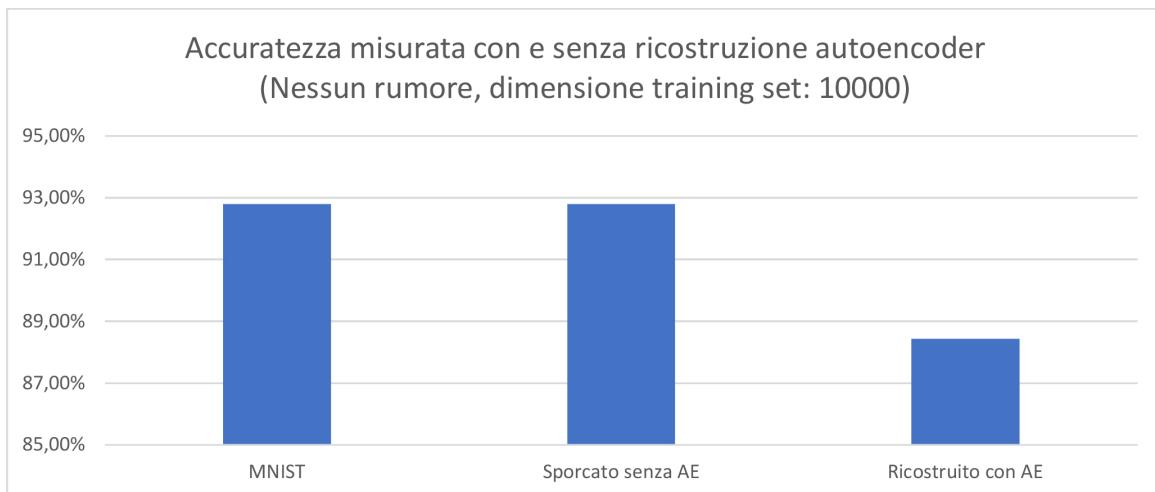


Figura 26: Accuratezza media valutata su training set da 10000 elementi al quale non è stato applicato rumore.

4 Appendice

```

function [rispostaClasse] = adattaRisposta(Y)
%adattaRisposta:
%Converte nel formato scelto la classificazione effettuata dalla
%rete.
%Richiede in input:
%- la matrice di output della rete
%Restituisce in output:
%- una matrice contenente l'output della rete adattato, l'elemento (i,j)
%contiene l'i-esimo dato in input per la classe j.

rispostaClasse = zeros(size(Y,1), size(Y,2));
for i = 1 : size(Y,1)
    %viene individuata la classe per la quale la rete ha fornito la

```

```

%risposta maggiore.
[~, argmax] = max(Y(i,:));
%adatto al formato della risposta
rispostaClasse(i,argmax) = 1;
end
end

function [autoEncoder2LV ,AE2LVarrayErroriTS ,AE2LVarrayErroriVS] =
    ↪ addestraAutoEncoder(trainingSetSporcato ,validationSetSporcato ,
    ↪ trainingSetImg ,validationSetImg ,testSetSporcato)
% Utilizzato come script per implementare la struttura dell'autoencoder.
% Input:
% - trainingSetSporcato , immagini di input a cui stato applicato del
% rumore;
% - validationSetSporcato , immagini usate come validation set;
% - trainingSetImg , immagini di input senza rumore, utilizzate come target;
% - validationSetImg , immagini del validation set senza rumore, utilizzate
% come target;
% - testSetSporcato , test set di immagini.
% Output:
% - autoEncoder2LV , la struttura autoencoder a due livelli
% - AE2LVarrayErroriTS , array errori della rete su training set;
% - AE2LVarrayErroriVS , array errori della rete su validation set;

% Configurazione iperparametri per i due autoencoder (un livello, due
    ↪ livelli)
AE1LVfunNodiHidden = @ReLU;
AE1LVfunNodiOutput = @funSigmoide;
AE2LVfunNodiHidden = @ReLU;
AE2LVfunNodiOutput = @funSigmoide;
funErrore = @funSommaQuadrati;
AE1LVnodiHidden = 128;
AE2LVnodiHidden = 32;
AE1LVepoche = 500;
AE2LVepoche = 500;
tipoAddestramento = "batch";
AE1LVminPeso = -0.09;
AE1LVmaxPeso = 0.09;
AE2LVminPeso = -0.09;
AE2LVmaxPeso = 0.09;
AE1LVlvHidden = 1;
AE2LVlvHidden = 3;
AE1LV_eta = 0.0002;
AE2LV_eta = 0.00002;
flagSoftmax = false;

% Creazione e inizializzazione dei livelli hidden del primo autoEncoder
strutturaLivelliHidden(1)=struct('dimLivello',AE1LVnodiHidden ,
    ↪ funDiAttivazione',AE1LVfunNodiHidden);

% Creazione rete neurale feed-forward multi-layer
autoEncoder1LV = creaReteFFML(784,784,AE1LVfunNodiOutput ,
    ↪ strutturaLivelliHidden ,AE1LVminPeso ,AE1LVmaxPeso ,AE1LVlvHidden);
autoEncoder1LV.W{2}=autoEncoder1LV.W{1}';


```

```
% Addestramento autoencoder AE1LV
[autoEncoder1LV, arrayErroriTS, arrayErroriVS] = addestraRete(autoEncoder1LV
    ↪ ,AE1LVepoch, tipoAddestramento, trainingSetSporcato,
    ↪ validationSetSporcato, trainingSetImg, validationSetImg, funErrore,
    ↪ AE1LV_eta, flagSoftmax, false);

% Creo autoencoder a 2 lv
% Creazione e inizializzazione dei livelli hidden
AE2LVstrutturaLivelliHidden(1)=struct('dimLivello',AE1LVnodiHidden ,
    ↪ funDiAttivazione',AE2LVfunNodiHidden);
AE2LVstrutturaLivelliHidden(2)=struct('dimLivello',AE2LVnodiHidden ,
    ↪ funDiAttivazione',AE2LVfunNodiHidden);
AE2LVstrutturaLivelliHidden(3)=struct('dimLivello',AE1LVnodiHidden ,
    ↪ funDiAttivazione',AE2LVfunNodiHidden);

% Creazione rete neurale feed-forward multi-layer
autoEncoder2LV = creaReteFFML(784,784,AE2LVfunNodiOutput,
    ↪ AE2LVstrutturaLivelliHidden,AE2LVminPeso,AE2LVmaxPeso,AE2LVlvHidden);
% Recupero pesi autoencoder 1 lv
autoEncoder2LV.W{1}=autoEncoder1LV.W{1};
% Pesi legati
autoEncoder2LV.W{3}=autoEncoder2LV.W{2}';
autoEncoder2LV.W{4}=autoEncoder2LV.W{1}';
% Addestro autoencoder 2LV
[autoEncoder2LV, AE2LVarrayErroriTS, AE2LVarrayErroriVS] = addestraRete(
    ↪ autoEncoder2LV,AE2LVepoch, tipoAddestramento, trainingSetSporcato,
    ↪ validationSetSporcato, trainingSetImg, validationSetImg, funErrore,
    ↪ AE2LV_eta, flagSoftmax, true);

% Vengono creati i grafici che mostrano l'andamento della funzione di errore
% rispetto al training e validation set dell'autoencoder a 2LV
figure('Name','Grafico errore AE2LV');
plot(AE2LVarrayErroriTS);
hold
plot(AE2LVarrayErroriVS);
legend('Errore rispetto al training set', 'Errore rispetto al validation set
    ↪ ');

% Pesi legati
autoEncoder2LVtest=forwardProp(autoEncoder2LV, testSetSporcato, true);

% Stampa immagini
figure('units','normalized','outerposition',[0 0 1 1], 'Name','Input - output
    ↪ denAutoencoder');
colormap(gray);

for i=1:10
    subplot(4,10,i+10)
    digit = reshape(autoEncoder2LVtest.X(i,:), [28,28]);
    imagesc(digit)
    axis off
    subplot(4,10,i+20)
    digit2 = reshape(autoEncoder2LVtest.z{autoEncoder2LV.numLivelliHidden
        ↪ +1}(i,:), [28,28]);
end
```

```

imagesc(digit2)
axis off
end

fprintf("\n struttura Denoising Autoencoder\n");
fprintf("\n struttura nodi hidden : %d - %d - %d -%d ",784,
    ↪ AE1LVnodiHidden ,AE2LVnodiHidden ,AE1LVnodiHidden ,784)
fprintf("\n eta : %f\n Numero di livelli hidden : %d",AE2LV_eta ,
    ↪ AE2LVLvHidden );
fprintf("\n tipo addestramento : %s\n Numero di epoch : %d\n",
    ↪ tipoAddestramento ,AE2LVEpoche);

```

```

function [reteNeurale,erroreTS,erroreVS] = addestramentoBatch(reteNeurale,
    ↪ trainingSetImg,validationSetImg,trainingSetLabel,validationSetLabel,
    ↪ funErrore,eta,flagSoftmax,fissapesi)
% addestramentoBatch
% Addestra la rete neurale applicando un approccio di tipo batch, ovvero
% l'aggiornamento dei pesi avviene alla fine, dopo aver calcolato l'errore.
% Ha bisogno in input di:
% - reteNeurale
% - trainingSetImg
% - validationSetImg
% - trainingSetLabel
% - validationSetLabel
% - funErrore, funzione di errore utilizzata per l'addestramento
% - eta, costante moltiplicativa utilizzata nella discesa del gradiente
% - flagSoftMax, se flag vera viene utilizzato softmax
% - fissapesi, se flag utilizzata per bloccare i pesi del primo livello
% dell'autoencoder a due livelli

% Inizializzazione variabili per calcolo errore training set e validation
    ↪ set
erreoreTS=0;
erreoreVS=0;

% Applicazione propagazione in avanti per training set e validation set
forwardPropTS=forwardProp(reteNeurale,trainingSetImg,flagSoftmax);
forwardPropVS=forwardProp(reteNeurale,validationSetImg,flagSoftmax);

% Calcolo errore training set e validation set, vengono divisi per la
erroreTS=sum(funErrore(forwardPropTS.z{forwardPropTS.numLivelliHidden+1},
    ↪ trainingSetLabel))/size(trainingSetImg,1);
erroreVS=sum(funErrore(forwardPropVS.z{forwardPropVS.numLivelliHidden+1},
    ↪ validationSetLabel))/size(validationSetImg,1);
% Back propagation su training set
forwardPropTS=backProp(forwardPropTS,trainingSetLabel,funErrore);
% Calcolo derivate parziali bias e pesi
[derBiasTS,derPesITS]=derivaPesi(forwardPropTS);
% Aggiorna pesi rete neurale
forwardPropTS=aggiornaPesi(forwardPropTS,derBiasTS,derPesITS,eta,fissapesi);

reteNeurale=forwardPropTS;
end

```

```

function [reteNeurale,sommatoriaErroriTS,sommatoriaErroriVS] =
    ↪ addestramentoOnline(reteNeurale,trainingSetImg,validationSetImg,
    ↪ trainingSetLabel,validationSetLabel,funErrore,eta,flagSoftmax)
%addestramentoOnline
% Addestra la rete neurale applicando un approccio di tipo online, ovvero
% l'aggiornamento dei pesi avviene durante le iterazioni del ciclo di
% addestramento.
% Ha bisogno in input di:
% - reteNeurale
% - immagini training set
% - etichette training set
% - immagini validation set
% - etichette validation set
% - eta
% - funzione di errore
% - flag per uso softmax
% Restituisce la rete neurale aggiornata e due array contenenti gli errori
% del training set e del validation set

% Inizializzazione variabili per calcolo errore training set e validation
% set
erroreTS=0;
sommatoriaErroriTS=0;
erroreVS=0;
sommatoriaErroriVS=0;

%Implementazione addestramento online
for n=1:size(trainingSetImg,1)
    % Applicazione propagazione in avanti, calcolo errore e sommo errori
    % del validation set
    forwardPropTS=forwardProp(reteNeurale,trainingSetImg(n,:),flagSoftmax);
    erroreTS=funErrore(forwardPropTS.z{forwardPropTS.numLivelliHidden+1},
        ↪ trainingSetLabel(n,:));
    sommatoriaErroriTS=sommatoriaErroriTS+erroreTS;

    % Applicazione propagazione in avanti, calcolo errore e sommo errori
    % del validation set
    if (n<=size(validationSetImg,1))
        forwardPropVS=forwardProp(reteNeurale,validationSetImg(n,:),,
            ↪ flagSoftmax);
        erroreVS=funErrore(forwardPropVS.z{forwardPropVS.numLivelliHidden
            ↪ +1},validationSetLabel(n,:));
        sommatoriaErroriVS=sommatoriaErroriVS+erroreVS;
    end

    % Back propagation su training set
    forwardPropTS=backProp(forwardPropTS,trainingSetLabel(n,:),funErrore);
    % Calcolo derivate parziali bias e pesi
    [derBiasTS,derPesiTS]=derivaPesi(forwardPropTS);
    % Aggiorna pesi rete neurale
    forwardPropTS=aggiornaPesi(forwardPropTS,derBiasTS,derPesiTS,eta);
end
reteNeurale=forwardPropTS;

end

```

```

function [reteNeurale ,arrayErroriTS ,arrayErroriVS] = addestraRete(
    ↪ reteNeurale ,epoches ,tipoAddestramento ,trainingSetImg ,validationSetImg ,
    ↪ trainingSetLabel ,validationSetLabel ,funErrore ,eta ,flagSoftmax ,
    ↪ fissapesi)
%addestraRete
% Addestramento rete neurale con discesa del gradiente

% Inizializzazione strutture di supporto:
    % Array errori training set
    arrayErroriTS=zeros(1,epoches);
    % Array errori validation set
    arrayErroriVS=zeros(1,epoches);

% Condizioni di fermata
    epochesNecessarie=floor(epoches/3);
    reteNeuraleCandidata=reteNeurale;
    erroreVS=realmax;

% Implementazione addestramento
for e = 1:epoches
    tempReteNeurale=reteNeurale;
    switch tipoAddestramento
        case 'batch'
            [reteNeurale ,arrayErroriTS(e) ,arrayErroriVS(e)]=
                ↪ addestramentoBatch(reteNeurale ,trainingSetImg ,
                ↪ validationSetImg ,trainingSetLabel ,validationSetLabel ,
                ↪ funErrore ,eta ,flagSoftmax ,fissapesi);
        case 'online'
            [reteNeurale ,arrayErroriTS(e) ,arrayErroriVS(e)]=
                ↪ addestramentoOnline(reteNeurale ,trainingSetImg ,
                ↪ validationSetImg ,trainingSetLabel ,validationSetLabel ,
                ↪ funErrore ,eta ,flagSoftmax);
        otherwise
            error('Tipo di addestramento supportato: [batch][online]');
    end
    contatoreErrore=0;
    % Controllo se l'errore decresce
    if arrayErroriVS(e)<erroreVS
        % Variabile che tiene conto del numero di volte in cui l'errore del
        % validation set cresce.
        contatoreErrore=0;
        erroreVS=arrayErroriVS(e);
        reteNeuraleCandidata=tempReteNeurale;
    else
        % Se l'errore cresce viene aumentato il contatore
        if e>=epochesNecessarie
            contatoreErrore=contatoreErrore+1;
            % Se l'errore cresce troppe volte consecutive superando una
            % soglia arbitraria
            if contatoreErrore>30
                % Viene interrotto il ciclo delle epoches
                break;
            end
        end
    end
end

```

```

    end
    fprintf("epoca corrente :%d\n",e);
end

% Ritorno la rete migliore
reteNeurale=reteNeuraleCandidata;

% All'occorrenza la dimensione degli array di errore viene ridotta
if e<epoch
    arrayErroriT$=arrayErroriT$(1:e);
    arrayErroriVS=arrayErroriVS(1:e);
end
end

```

```

function reteNeurale = aggiornaPesi(reteNeurale,derBias,derPesi,eta,
    ↪ fissaPesiPrimoLivello)
% AggiornaPesi
% La funzione aggiorna pesi e bias della rete utilizzando i valori
% calcolati precedentemente tramite le derivate.
% L'aggiornamento sar modulato dal parametro eta che regoler l'intervalllo
% di scarto da considerare tra un valore candidato e il successivo
% (spostamento sul grafico della curva).
% Prende in input:
% - reteNeurale
% - derivate bias
% - derivate pesi
% - eta
% - flag fissaPesiPrimoLivello
% Restituisce in output la rete neurale con pesi e bias aggiornati.

for i=1:reteNeurale.numLivelliHidden+1
    % se flag fissapesi attiva, non vengono aggiornati i pesi del primo
    % livello della rete
    if fissaPesiPrimoLivello
        if (i ~= 1)
            reteNeurale.b{i}=reteNeurale.b{i}-(eta*derBias{i});
            reteNeurale.W{i}=reteNeurale.W{i}-(eta*derPesi{i});
        end
    else
        reteNeurale.b{i}=reteNeurale.b{i}-(eta*derBias{i});
        reteNeurale.W{i}=reteNeurale.W{i}-(eta*derPesi{i});
    end
end
end

```

```

function reteNeurale = backProp(reteNeurale,T,funErrore)
% backProp
% La funzione effettua la propagazione all'indietro della rete, per ogni
% strato vengono calcolati i relativi delta partendo dai nodi di output
% fino a quelli di input.
% Ha bisogno di:
% - reteNeurale
% - T: Target con cui confrontare i valori di output (etichette)
% - funErrore: funzione per il calcolo dell'errore
% Restituisce in output la rete aggiungendo ad essa i delta calcolati

```

```
% utilizzando le derivate parziali.
% Per calcolare il delta del livello di output vengono moltiplicate la
% derivata della funzione di attivazione sull'input del nodo di output e la
% derivata parziale della funzione di errore ottenuta confrontando y e t
% (output ultimo nodo e label).

% All'interno di un ciclo verrano calcolati tutti i delta intermedi sui
% livelli hidden, ognuno si ottiene moltiplicando la derivata della
% funzione di attivazione sull'input del nodo ed il prodotto tra la matrice
% dei delta del livello successivo e la matrice dei pesi del livello
% successivo.
% La matrice ottenuta dal prodotto precedente viene poi moltiplicata punto
% per punto con il vettore contenente i valori restituiti dalla derivata della
% funzione di attivazione sull'input di quel livello.

indiceStratoOutput=reteNeurale.numLivelliHidden+1;
reteNeurale.delta{indiceStratoOutput}=reteNeurale.funDiAttivazioneOutput(
    ↪ reteNeurale.a{indiceStratoOutput},true).* funErrore(reteNeurale.z{
    ↪ indiceStratoOutput},T,true);
for i=indiceStratoOutput-1:-1:1
    prodDeltaPesi = reteNeurale.delta{i+1}*reteNeurale.W{i+1}; %calcolo
        ↪ intermedio
    reteNeurale.delta{i} = reteNeurale.g{i}(reteNeurale.a{i},true) .*%
        ↪ prodDeltaPesi;
end
end
```

```
function [immaginiCifre,etichetteCifre] = caricaDataset(percorsoImmagini,
    ↪ percorsoEtichette)
% caricaDataset:
% Funzione utilizzata per caricare il dataset MNIST.
% Il dataset formato da 60000 immagini che rappresentano
% le cifre da 0 a 9 scritte a mano. Ogni immagine rappresentata
% attraverso una matrice di pixel 28x28 (784px) in cui
% ogni cella rappresenta l'intensità del singolo pixel
% in una rappresentazione a scala di grigi (quindi da 0 a 255).
% Ad ogni immagine associata un'etichetta che indica la cifra
% rappresentata dall'immagine.
% Questa funzione utilizza le funzioni realizzate
% dall'università di Stanford e permettono di ricavare due
% matrici che contengono immagini e labels.

% La matrice immaginiCifre una matrice 60000x784, nella quale
% ogni colonna rappresenta un'immagine, e le righe i valori di
% intensità associati ai pixel.
immaginiCifre = (loadMNISTImages(percorsoImmagini))';

% La matrice etichetteCifre una matrice 60000x1 in cui ogni riga i
% contiene la cifra che corrisponde all'i-esima immagine.
etichetteCifre = loadMNISTLabels(percorsoEtichette);
end

function images = loadMNISTImages(filename)
%loadMNISTImages returns a 28x28x[number of MNIST images] matrix containing
%the raw MNIST images
```

```

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2051, ['Bad magic number in ', filename, '']);

numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
numCols = fread(fp, 1, 'int32', 0, 'ieee-be');

images = fread(fp, inf, 'unsigned char');
images = reshape(images, numCols, numRows, numImages);
images = permute(images,[2 1 3]);

fclose(fp);

% Reshape to #pixels x #examples
images = reshape(images, size(images, 1) * size(images, 2), size(images, 3))
    ↵ ;
% Convert to double and rescale to [0,1]
images = double(images) / 255;

end

function labels = loadMNISTLabels(filename)
%loadMNISTLabels returns a [number of MNIST images]x1 matrix containing
%the labels for the MNIST images

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2049, ['Bad magic number in ', filename, '']);

numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');

labels = fread(fp, inf, 'unsigned char');

assert(size(labels,1) == numLabels, 'Mismatch in label count');

fclose(fp);

end

```

```

function reteNeurale = creaReteFFML(numNodiInput,numNodiOutput,
    ↵ funDiAttivazioneOutput,strutturaLivelliHidden,minPeso,maxPeso,
    ↵ numLivHidden)
% creaReteFFML
% Crea una rete neurale feed-forward multi-layer.
% Richiede in input:
% - numNodiInput: numero dei nodi in input.
% - numNodiOutput: numero dei nodi in output.
% - funDiAttivazioneOutput: puntatore alla funzione di attivazione.

```

```
% - strutturaLivelliHidden: array contenente i nodi hidden con relativa
    ↪ funzione di
% attivazione.
% - minPeso: valore minimo di un peso.
% - maxPeso: valore massimo di un peso.
%
% In output restituisce la struttura che descrive la rete neurale.

% Viene uniformata la rappresentazione della matrice
if size(strutturaLivelliHidden,1)>size(strutturaLivelliHidden,2)
    strutturaLivelliHidden=strutturaLivelliHidden';
end

% Inizializzazione struttura output
reteNeurale.numNodiInput=numNodiInput;
reteNeurale.numNodiOutput=numNodiOutput;
reteNeurale.numLivelliHidden=numLivHidden;
reteNeurale.funDiAttivazioneOutput=funDiAttivazioneOutput;
reteNeurale.numLivelli=size(strutturaLivelliHidden,2)+2;
for i=1:numLivHidden
    reteNeurale.m(i)=strutturaLivelliHidden(i).dimLivello;
end

% Generazione casuale dei pesi
% *** PRIMO LIVELLO ***
reteNeurale.b{1}=(maxPeso-minPeso).*rand(1,reteNeurale.m(1))+minPeso;
reteNeurale.W{1}=(maxPeso-minPeso).*rand(reteNeurale.m(1),numNodiInput)+
    ↪ minPeso;
reteNeurale.g{1}=strutturaLivelliHidden(1).funDiAttivazione;
% Generazione casuale dei pesi
% *** LIVELLI HIDDEN ***
if reteNeurale.numLivelliHidden>=2
    for i=2:reteNeurale.numLivelliHidden
        reteNeurale.b{i}=(maxPeso-minPeso).*rand(1,reteNeurale.m(i))+minPeso
            ↪ ;
        reteNeurale.W{i}=(maxPeso-minPeso).*rand(reteNeurale.m(i),
            ↪ reteNeurale.m(i-1))+minPeso;
        reteNeurale.g{i}=strutturaLivelliHidden(i).funDiAttivazione;
    end
end
% Generazione casuale dei pesi
% *** ULTIMO LIVELLO ***
reteNeurale.b{reteNeurale.numLivelli-1}=(maxPeso-minPeso).*rand(1,
    ↪ numNodiOutput)+minPeso;
reteNeurale.W{reteNeurale.numLivelli-1}=(maxPeso-minPeso).*rand(
    ↪ numNodiOutput,reteNeurale.m(reteNeurale.numLivelliHidden))+minPeso;
reteNeurale.g{reteNeurale.numLivelli-1}=funDiAttivazioneOutput;
end
```

```
function [setImmagini, setEtichette, indiceElemUtilizzati] = creaSet(
    ↪ dimensioneSet, indiceElemUtilizzati, matEtichette, matImmagini)
    % creaSet: Partendo da un dataset crea un sotto-insieme, stata
    % utilizzata per generare training set, validation set e test set.

    % Calcolo dimensione singola partizione
```

```

partizioneCifre = floor(dimensioneSet/10);
% Inizializzazione struttura di output
setImmagini=zeros(dimensioneSet,784);
setEtichette=zeros(dimensioneSet,10);
% Inizializzazione struttura di supporto
% Contatore numero di elementi inseriti per ogni cifra
contCifre=zeros(1,10);
% Contatore elementi inseriti nell'insieme
contElementi=0;
% Finche non viene riempito l'insieme...
while contElementi < dimensioneSet
    % Prendo una posizione casuale all'interno del dataset
    randPos=floor((60000-1).*rand(1)+1);
    % Controllo che l'elemento non sia già stato inserito
    if indiceElemUtilizzati(randPos)==0
        % Controllo quanti elementi di una cifra ho inserito
        if contCifre(matEtichette(randPos)+1)<partizioneCifre
            % Aggiorni contatori e strutture
            contCifre(matEtichette(randPos)+1)=contCifre(matEtichette(
                ↪ randPos)+1)+1;
            contElementi=contElementi+1;
            indiceElemUtilizzati(randPos)=1;
            % Viene inserito questo elemento nell'insieme
            setEtichette(contElementi,matEtichette(randPos)+1)=1;
            setImmagini(contElementi,:)=matImmagini(randPos,:);
        end
    end
end
end

```

```

function [derBias,derPesi] = derivaPesi(reteNeuraleBP)
% derivaPesi
% La funzione calcola, per ogni livello, le derivate parziali della
% funzione di errore rispetto ai pesi e bias del livello.
% Richiede in input:
% - reteNeuraleBP: rete neurale sulla quale stata effettuato il calcolo
% della back propagation.
% In output restituisce i vettori contenenti le derivate dei pesi e dei
% bias.

% Inizialmente vengono calcolate le derivate parziali del primo livello
% hidden. Per i bias le derivate vengono sommate tra loro, mentre per i
% pesi necessario effettuare un prodotto tra la matrice dei delta e X
% (input). Per effettuare il prodotto la matrice dei delta viene trasposta.

% Per i restanti livelli (hidden+output) il calcolo verr effettuato allo
% stesso modo (tra il livello i e i-1)

derBias{1}=sum(reteNeuraleBP.delta{1},1);
derPesi{1}=reteNeuraleBP.delta{1}' * reteNeuraleBP.X;

for i=2 : reteNeuraleBP.numLivelliHidden+1
    derBias{i}=sum(reteNeuraleBP.delta{i},1);
    derPesi{i}=reteNeuraleBP.delta{i}' * reteNeuraleBP.z{i-1};
end

```

```
end
```

```

function reteNeurale = forwardProp(reteNeurale,X,flagSoftmax)
% forwardProp
% Propagazione in avanti della rete neurale
% Prende come input:
% - reteNeurale
% - X: matrice contenente le immagini di input
% - flagSoftmax: flag per stabilire se applicare softmax
% Restituisce:
% - a: valori di input ricevuti dell'intero layer i dato il vettore di input
%   ↪ j
% - z: output dei nodi
% - X: come sopra

% Inizializzo strato di input
reteNeurale.X=X;
% Array dei valori di output del livello precedente
zPrec=X; % Al primo livello coincide con l'input

% Per ogni livello (escluso input) viene calcolato l'input e l'output del
% livello
for i=1 : reteNeurale.numLivelliHidden+1
    % Per calcolare la a (input) associata al livello corrente avremo da
    % considerare un numero di pesi pari al prodotto tra i nodi di questo
    % livello e i nodi del livello precedente.
    % Facendo il prodotto tra la matrice dei valori correnti trasposta e
    % gli output del livello precedente si ottiene una matrice che avr
    % come righe il numero di nodi di input sul layer attuale e come colonne
    % il numero di vettori per cui stata moltiplicata.
    % Per uniformarsi agli standard il prodotto finale viene trasposto.
    % Per calcolare invece la z (output) associata al livello corrente
    % abbiamo bisogno di sommare il bias del layer corrente ad ogni riga
    % della matrice di input.

    reteNeurale.a{i}=(zPrec*reteNeurale.W{i})';
    reteNeurale.a{i}=reteNeurale.a{i}+reteNeurale.b{i};
    reteNeurale.z{i}=reteNeurale.g{i}(reteNeurale.a{i});
    zPrec=reteNeurale.z{i};
end

% Controllo flag softmax
if flagSoftmax
    % Il calcolo del softmax stato implementato seguendo le indicazioni
    % date durante il corso.
    softmax=exp(reteNeurale.z{reteNeurale.numLivelliHidden+1}) ./ sum(exp(
        ↪ reteNeurale.z{reteNeurale.numLivelliHidden+1}),2);
    reteNeurale.z{reteNeurale.numLivelliHidden+1}=softmax;
end

```

```

% mainScript:
% Questo script viene utilizzato per addestrare una rete utilizzando il
% criterio di minimizzazione dell'errore della discesa del gradiente.
% Successivamente viene addestrato un autoencoder e viene testata la rete
% con input uguale all'uscita dell'autoencoder applicato ad un dataset

```

```
% con rumore.

% Vengono cancellate le variabili d'ambiente di eventuali esecuzioni
%   ↪ precedenti
clearvars;
tic
% Aggiunge al workspace il percorso alla cartella resources.
% Al suo interno:
% - le funzioni di attivazione
% - il dataset MINST (immagini e label)
addpath('./risorse/');

% Funzione di attivazione dei nodi di output
Class_funNodiOutput = @funIdentita;
% Funzione di attivazione dei nodi interni
Class_funNodiHidden = @funSigmoid;
% Funzione d'errore
Class_funErrore = @funCrossEntropy;
% Numero di nodi interni
Class_nodiHidden = 180;
% Numero di epoch
Class_epoches = 300;
% Tipo di addestramento: "online" o "batch", altre stringhe sono causa
% d'errore
tipoAddestramento = "batch";
% Numero di elementi del training set
dimensioneTrainingSet = 1000;
% Numero di elementi del test set
dimensioneTestSet = 250;
% Numero di elementi del validation set
dimensioneValidationSet = 250;
% Valore casuale minimo del peso
ClassminPeso = -0.09;
% Valore casuale massimo del peso
ClassmaxPeso = 0.09;
% numero di livelli hidden della rete
ClasslvHidden = 1;
% Valore eta, utilizzato per la discesa del gradiente
% migliore era 0.0004
Class_eta = 0.0004;
% Softmax, funzione esponenziale normalizzata, utilizzabile per rimappare
% l'intervallo di classificazione in modo da utilizzare la sigmoide per
% problemi con pi di due classi
flagSoftmax = true;

% Tipo rumore utilizzato,
% sono supportati -> Standard rumore utilizzato nel paper di riferimento
%                   -> GaussianStandard rumore gaussiano standard matlab
%                   -> GaussianManual rumore gaussiano implementato
%   ↪ manualmente con intensit percDistruzione/1000
%                   -> SaltNPepper rumore sale e pepe con intensit
%   ↪ percDistruzione/1000
tipoRumore = "Standard";

% Percentuale di distruzione dell'input utilizzata
```



```

    ↵ Class_eta,flagSoftmax ,false);

% Propagazione in avanti utilizzando come input il test set
[reteNeuraleClass] = forwardProp(reteNeuraleClass , testSetImg , true);

% Viene calcolata l'accuratezza della valutazione della rete rispetto al
% test set
[accuratezzaClassPura] = valutazioneRete(reteNeuraleClass.z{reteNeuraleClass
    ↵ .numLivelliHidden+1}, testSetLab);

% Funzione che stampa a video il confronto tra le label del test set e i
% risultati dati sulle stesse immagini dalla rete, il numero di immagini
% selezionate dato dall'ultimo parametro
riscontroVisivo(testSetImg ,reteNeuraleClass.z{reteNeuraleClass .
    ↵ numLivelliHidden+1},testSetLab ,25);
pause(3);
fprintf("\n Fase addestramento autoencoder ");

% Sporco training e validation set per addestrare l'autoencoder
trainingSetSporcato = sporcaInput(trainingSetImg ,v, tipoRumore);
validationSetSporcato = sporcaInput(validationSetImg ,v, tipoRumore);
testSetSporcato = sporcaInput(testSetImg ,v, tipoRumore);

% Costruzione e addestramento dell'autoencoder
[denoisingAutoEncoder ,AE2LVarrayErrorITS ,AE2LVarrayErroriVS]=
    ↵ addestraAutoEncoder(trainingSetSporcato ,validationSetSporcato ,
    ↵ trainingSetImg ,validationSetImg ,testSetSporcato);

pause(2);
fprintf("\n Fine addestramento autoencoder ");

% Test rete senza autoencoder su input sporcato
fprintf("\n Classificazione input sporcato senza autoencoder");

% Test senza autoencoder
reteNeuraleClassTest = forwardProp(reteNeuraleClass , testSetSporcato ,true);
% Calcolo accuratezza senza autoencoder
[accuratezzaClassSenzaDenoising] = valutazioneRete(reteNeuraleClassTest.z{
    ↵ reteNeuraleClassTest.numLivelliHidden+1}, testSetLab);

% Stampa i dati utilizzati per l'addestramento della rete e
% l'accuratezza della rete rispetto al test set.
fprintf("\n Dimensione Training Set : %d\n Dimensione Validation Set : %d",
    ↵ dimensioneTrainingSet,dimensioneValidationSet);
fprintf("\n Struttura Rete di classificazione");
fprintf("\n Dimensione Test Set: %d\n Nodi hidden per livello: %d",
    ↵ dimensioneTestSet , Class_nodiHidden);
fprintf("\n eta : %f\n Numero di livelli hidden: %d",Class_eta ,
    ↵ Class_lvHidden);
fprintf("\n tipo addestramento: %s\n Numero di epoch: %d",tipoAddestramento
    ↵ ,Class_epoch);
fprintf("\n Accuratezza classificazione pura: %d%%\n ", int16(
    ↵ accuratezzaClassPura*100));

fprintf("\n Intensita rumore: %d%%",v);

```

```

fprintf("\n Tipo di rumore utilizzato: %s", tipoRumore);
fprintf("\n Risultato classificazione input sporcato senza autoencoder ");
fprintf("\n Accuratezza: %d%%\n ", int16(accuratezzaClassSenzaDenoising*100)
    ↪ );

% L'input sporcato viene passato all'autoencoder e poi alla rete di
% classificazione
denoisingAutoEncoder = forwardProp(denoisingAutoEncoder, testSetSporcato,
    ↪ false);

% Classificazione dell'input ricostruito tramite denoising
reteNeuraleConDenoising = forwardProp(reteNeuraleClass, denoisingAutoEncoder
    ↪ .z{denoisingAutoEncoder.numLivelliHidden+1},true);

% Valutazione classificazione della reteNeuraleConDenoising
[accuratezzaClassConDenoising] = valutazioneRete(reteNeuraleConDenoising.z{
    ↪ reteNeuraleConDenoising.numLivelliHidden+1}, testSetLab);

% Funzione che stampa a video il confronto tra le label del test set e i
% risultati dati sulle stesse immagini dalla rete, il numero di immagini
% selezionate dato dall'ultimo parametro
riscontroVisivo(testSetSporcato,reteNeuraleConDenoising.z{
    ↪ reteNeuraleConDenoising.numLivelliHidden+1},testSetLab,10);

% Stampa i dati utilizzati per l'addestramento della rete e
% l'accuratezza della rete rispetto al test set.
fprintf("\n Risultato finale classificazione rete con autoencoder ");
fprintf("\n Accuratezza : %d%%\n ", int16(accuratezzaClassConDenoising*100))
    ↪ ;
fprintf("\n Tempo impiegato per addestramento e test della rete: %f secondi
    ↪ \n",toc);

```

```

function riscontroVisivo(imgCifre,outputRete,labelCorrette,numImmagini)
% Stampa le immagini contenente le cifre accompagnate dall'output del
% relativo output della rete e la label corretta dell'immagine.

rispostaClasse=adattaRisposta(outputRete);
figure('units','normalized','outerposition',[0 0 1 1], 'Name','Risultati
    ↪ classificazione');
colormap(gray);
for i=1:numImmagini
    j=1;
    z=1;
    while rispostaClasse(i,j) ~=1
        j=j+1;
    end
    while labelCorrette(i,z) ~=1
        z=z+1;
    end

    subplot(ceil(sqrt(numImmagini)),ceil(sqrt(numImmagini)),i)
    digit = reshape(imgCifre(i,:), [28,28]);
    imagesc(digit)
    title(sprintf("Out rete: %d / Label: %d", int16(j-1),int16(z-1)),'
        ↪ FontSize',8)

```

```

    axis off
end

end

function setImgTilde = sporcaInput(setImgInput,percDistruzione,tiporumore)
%sporcaInput
% Aggiunge rumore alle immagini passate.
% Prende in input:
% - setImgInput, insieme delle immagini da sporcare
% - percDistruzione, percentuale di distruzione delle immagini
% - tiporumore : permette di selezionare che tipo di rumore applicare ,
% sono supportati -> "Standard" rumore utilizzato nel paper di riferimento
% -> "GaussianStandard" rumore gaussiano standard matlab ,
%   ↪ media 0 e stdev 0.01
%           -> "GaussianManual" rumore gaussiano implementato
%   ↪ manualmente con intensit percDistruzione/1000
%           -> "SaltNPepper" rumore sale e pepe con intensit
%   ↪ percDistruzione/1000
% Ritorna l'insieme delle immagini sporcate

setImgTilde=setImgInput;
% Estraie il numero di immagini da sporcare
numImmagini=size(setImgInput,1);
% In proporzione alla percentuale di distruzione data in input calcola
% quanti pixel sia necessario sporcare per ogni immagine
numPxDaSporcare=floor((percDistruzione*784)/100);
% Switch che gestisce i diversi tipi di rumore supportati
switch tiporumore
    case 'Standard'
        % Per ogni immagine genera un numero di 0 uguale alla percentuale di
        % distruzione passata (percDistruzione)
        for i=1:numImmagini
            % Array contenente indice pixel sporcati
            arrayPxSporcati=zeros(1,784);
            % Contatore pixel sporcati
            numPxSporcati=0;
            while numPxSporcati<numPxDaSporcare
                % Scelgo casualmente quale pixel sporcare
                randPx=floor((784-1).*rand(1)+1);
                if arrayPxSporcati(randPx)==0
                    setImgTilde(i,randPx)=0;
                    arrayPxSporcati(randPx)=1;
                    numPxSporcati=numPxSporcati+1;
                end
            end
        end
    case 'GaussianStandard'
        %"GaussianStandard" rumore gaussiano standard matlab, media 0 e stdev
        ↪ 0.01
        for i=1:numImmagini
            digit = reshape(setImgInput(i,:), [28,28]);
            digitTilde = imnoise(digit,'gaussian');
            setImgTilde(i,:) = reshape(digitTilde,[784,1]);
        end
    end
end

```

```

    end

    case 'GaussianManual'
    %"GaussianManual" rumore gaussiano implementato manualmente con intensit
        → percDistruzione/1000
    for i=1:numImmagini
        digit = reshape(setImgInput(i,:), [28,28]);
        digitTilde = double(digit) + (percDistruzione/1000)*randn(size(
            → digit));
        setImgTilde(i,:) = reshape(digitTilde,[784,1]);
    end

    case 'SaltnPepper'
    %"SaltnPepper" rumore sale e pepe con intensit percDistruzione/1000
    for i=1:numImmagini
        digit = reshape(setImgInput(i,:), [28,28]);
        digitTilde = imnoise(digit,'salt & pepper', percDistruzione
            → /1000);
        setImgTilde(i,:) = reshape(digitTilde,[784,1]);
    end
    otherwise
        fprintf("\nIl tipo di rumore richiesto non supportato, non verr
            → applicato rumore al dataset");
    end
end

```

```

% Script utilizzato per la sperimentazione orientata al calcolo della media
% dell'accuratezza e della deviazione standard della rete di
% classificazione.

% Inizializzazione variabili
clearvars;
numeroRipetizioni = 5;
M = zeros(8,6);
trainingSet = 1000;
validationSet = 250;
testSet = 250;

%senza rumore, standard 0
row=1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    → validationSet, testSet, "Standard", 0);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostruito(i) = AccRicostruito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);

```

```

M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

%standard 25
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    ↪ validationSet, testSet, "Standard", 25);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostruito(i) = AccRicostruito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

%standard 50
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    ↪ validationSet, testSet, "Standard", 50);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostruito(i) = AccRicostruito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

%Standard 75
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    ↪ validationSet, testSet, "Standard", 75);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostruito(i) = AccRicostruito;
end

```

```

M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

%SaltnPepper 25
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    ↪ validationSet, testSet, "SaltnPepper", 25);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostruito(i) = AccRicostruito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

%SaltnPepper 50
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    ↪ validationSet, testSet, "SaltnPepper", 50);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostruito(i) = AccRicostruito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

%SaltnPepper 75
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    ↪ validationSet, testSet, "SaltnPepper", 75);

```

```

tempAccNorm(i) = AccNorm;
tempAccSporc(i) = AccSporc;
tempAccRicostruito(i) = AccRicostruito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

%GaussianStandard
row=row+1;
tempAccNorm = zeros([1,numeroRipetizioni]);
tempAccSporc = zeros([1,numeroRipetizioni]);
tempAccRicostruito = zeros([1,numeroRipetizioni]);
for i = 1:numeroRipetizioni
[AccNorm,AccSporc,AccRicostruito] = testSingolaConfig(trainingSet,
    validationSet, testSet, "GaussianStandard", 0);
    tempAccNorm(i) = AccNorm;
    tempAccSporc(i) = AccSporc;
    tempAccRicostruito(i) = AccRicostruito;
end
M(row,1) = mean(tempAccNorm);
M(row,2) = std(tempAccNorm);
M(row,3) = mean(tempAccSporc);
M(row,4) = std(tempAccSporc);
M(row,5) = mean(tempAccRicostruito);
M(row,6) = std(tempAccRicostruito);

csvwrite('AccuratezzeMedie.txt',M);

fprintf("\n FINE\n");

```

```

function [accuratezzaClassPura,accuratezzaClassSenzaDenoising,
    accuratezzaClassConDenoising] = testSingolaConfig(
    dimensioneTrainingSet,dimensioneTestSet,dimensioneValidationSet,
    tipoRumore,v)
% Questo script viene utilizzato per testare una singola configurazione
% della rete con denoising autoencoder
% prende in input la dimensione del training set, quella del test set,
% quella del validation set e il tipo di rumore
% che si vuole applicare per sporcare le immagini, infine il parametro v
% (parametro di distruzione) che indica in percentuale quanto vede essere
% sporcata l'immagine.

% Successivamente viene addestrato un autoencoder e viene testata la rete
% con input uguale all'uscita dell'autoencoder applicato ad un dataset
% con rumore.
% cancello le variabili d'ambiente di eventuali esecuzioni precedenti

% Aggiunge al workspace il percorso alla cartella resources.
% Al suo interno:
% - le funzioni di attivazione

```

```
% - il dataset MINST (immagini e label)
addpath('./risorse/');

% Funzione di attivazione dei nodi di output
Class_funNodiOutput = @funIdentita;
% Funzione di attivazione dei nodi interni
Class_funNodiHidden = @funSigmoide;
% Funzione d'errore
Class_funErrore = @funCrossEntropy;
% Numero di nodi interni
Class_nodiHidden = 180;
% Numero di epoche
Class_epoche = 300;
% Tipo di addestramento: "online" o "batch", altre stringhe sono causa
% d'errore
tipoAddestramento = "batch";
% Numero di elementi del training set

% Valore casuale minimo del peso
ClassminPeso = -0.09;
% Valore casuale massimo del peso
ClassmaxPeso = 0.09;
% numero di livelli hidden della rete
ClasslvHidden = 1;
% Valore eta, utilizzato per la discesa del gradiente
% migliore era 0.0004
Class_eta = 0.0004;
% Softmax, funzione esponenziale normalizzata, utilizzabile per rimappare
% l'intervallo di classificazione in modo da utilizzare la sigmoide per
% problemi con pi di due classi
flagSoftmax = true;

% Inizializzazione matrici di immagini ed etichette del dataset MNIST
[matImmagini, matEtichette] = caricaDataset('./risorse/train-images-idx3-
    ↪ ubyte', './risorse/train-labels-idx1-ubyte');

% Gli elementi che costituiscono training set, validation set e test
% set vengono selezionati casualmente in modo da diversificare i test
% effettuati con gli stessi input.
% Verranno a tal fine effettuate delle chiamate alla funzione rand()
% ottenendo dei numeri casuali da 1 a 60000 (dimensione MNIST).
% Per evitare di considerare pi volte una stessa immagine, si tiene
% traccia degli indici gi "pescati" attraverso una struttura dati.
% Il numero di elementi presi per ogni cifra deve essere uguale per
% tutte le cifre.
% Essendoci nel dataset 10 cifre diverse il numero di elementi preso
% per ognuna deve essere uguale ad un decimo della dimensione del set
% indicata in input. (Es. dim. training set=1000 -> ogni cifra 100
% elementi). Se viene inserita una dimensione che non sia un multiplo di
% dieci, questa viene arrotondata al valore multiplo di dieci che la
% precede.

% Vengono aggiornate le dimensioni dei set in modo da poter avere dieci
% partizioni della stessa dimensione.
dimensioneTrainingSet = floor(dimensioneTrainingSet/10)*10;
```

```

dimensioneTestSet = floor(dimensioneTestSet/10)*10;
dimensioneValidationSet = floor(dimensioneValidationSet/10)*10;

indiceElemUtilizzati=zeros(1,60000);

% Creazione training set
[trainingSetImg,trainingSetLab,indiceElemUtilizzati]=creaSet(
    ↪ dimensioneTrainingSet,indiceElemUtilizzati,matEtichette,matImmagini);
% Creazione validation set
[validationSetImg,validationSetLab,indiceElemUtilizzati]=creaSet(
    ↪ dimensioneValidationSet,indiceElemUtilizzati,matEtichette,matImmagini
    ↪ );
% Creazione test set
[testSetImg,testSetLab,indiceElemUtilizzati]=creaSet(dimensioneTestSet,
    ↪ indiceElemUtilizzati,matEtichette,matImmagini);

% Creazione e inizializzazione dei livelli hidden
strutturaLivelliHidden(1)=struct('dimLivello',Class_nodiHidden,
    ↪ funDiAttivazione',Class_funNodiHidden);

% Creazione rete neurale feed-forward multi-layer per la classificazione
reteNeuraleClass = creaReteFFML(784,10,Class_funNodiOutput,
    ↪ strutturaLivelliHidden,ClassminPeso,ClassmaxPeso,ClasslvHidden);

%timestamp utilizzato per misurare il tempo impiegato per training e test
%tic;

% Addestramento rete neurale
fprintf("\n Inizio addestramento della rete di classificazione\n");

[reteNeuraleClass, arrayErroriTS, arrayErroriVS] = addestraRete(
    ↪ reteNeuraleClass,Class_epocha, tipoAddestramento, trainingSetImg,
    ↪ validationSetImg, trainingSetLab, validationSetLab, Class_funErrore,
    ↪ Class_eta, flagSoftmax, false);

%propagazione in avanti utilizzando come input il test set
[reteNeuraleClass] = forwardProp(reteNeuraleClass, testSetImg, true);

%viene calcolata l'accuratezza della valutazione della rete rispetto al
%test set
[accuratezzaClassPura] = valutazioneRete(reteNeuraleClass.z{reteNeuraleClass
    ↪ .numLivelliHidden+1}, testSetLab);

fprintf("\n Fase addestramento autoencoder ");

%Sporco training e validation set per addestrare l'autoencoder
trainingSetSporcato = sporcaInput(trainingSetImg,v, tipoRumore);
validationSetSporcato = sporcaInput(validationSetImg,v, tipoRumore);
testSetSporcato = sporcaInput(testSetImg,v, tipoRumore);

%costruzione e addestramento dell'autoencoder
[denoisingAutoEncoder,AE2LVarrayErroriTS,AE2LVarrayErroriVS]=
    ↪ addestraAutoEncoder(trainingSetSporcato,validationSetSporcato,
    ↪ trainingSetImg,validationSetImg,testSetSporcato);

```

```

pause(2);
fprintf("\n Fine addestramento autoencoder ");

%test rete senza autoencoder su input sporcato
fprintf("\n Classificazione input sporcato senza autoencoder");

%test senza autoencoder
reteNeuraleClassTest = forwardProp(reteNeuraleClass, testSetSporcato,true);
%calcolo accuratezza senza autoencoder
[accuratezzaClassSenzaDenoising] = valutazioneRete(reteNeuraleClassTest.z{
    ↪ reteNeuraleClassTest.numLivelliHidden+1}, testSetLab);

%l'input sporcato viene passato all'autoencoder e poi alla rete di
%classificazione
denoisingAutoEncoder = forwardProp(denoisingAutoEncoder, testSetSporcato,
    ↪ false);

%classificazione dell'input ricostruito tramite denoising
reteNeuraleConDenoising = forwardProp(reteNeuraleClass, denoisingAutoEncoder
    ↪ .z{denoisingAutoEncoder.numLivelliHidden+1},true);

%valutazione classificazione della reteNeuraleConDenoising
[accuratezzaClassConDenoising] = valutazioneRete(reteNeuraleConDenoising.z{
    ↪ reteNeuraleConDenoising.numLivelliHidden+1}, testSetLab);

end

```

```

function [accuratezza] = valutazioneRete(Y,T)
%valutazioneRete: funzione che valuta l'accuratezza complessiva della rete
%nel classificare le cifre.
%richiede in input:
%- la matrice di output Y, contenente il risultato restituito dalla rete
%- T, matrice contenente le label associate al set%
%restituisce in output:
% -L'accuratezza complessiva della rete nella classificazione.

%Check degli input
if(size(Y,1) ~= size(T,1)) || (size(Y,2) ~= size(T,2))
    error("Le dimensioni dei parametri non coincidono");
end

%ricavo la risposta della rete
rispostaClasse = adattaRisposta(Y);

%viene effettuato il calcolo del numero di risposte corrette
corrette = nnz(rispostaClasse .* T);

%e viene calcolata l'accuratezza
accuratezza = corrette/size(Y,1);

end

```

```

function z = funCrossEntropy(x,y,daDerivare)
% funCrossEntropy:

```

```
% Una delle possibili funzioni utilizzate per il
% calcolo dell'errore. Dati due valori x e y viene
% calcolato il valore della funzione cross entropy
% rispetto ai valori di input.
% Se richiesto, viene calcolata e restituita la derivata
% della funzione.

% Viene richiesta la derivata della cross entropy
if exist('daDerivare','var')
    z=x-y;
% Viene la calcolata la cross entropy senza derivata
else
    % Distinguiamo il calcolo per i valori maggiori
    % o uguali di 0. Quelli uguali a 0 danno problemi
    % per il calcolo del logaritmo.

    % Valori >0:
    y(x>0)=y(x>0) .* log(x(x>0));

    % Valori =0;
    y(x==0)=y(x==0)*(-708); %Logaritmo del valore pi
                               %piccolo rappresentabile.
    z=-sum(y);
end
end
```

```
function z = funIdentita(x,daDerivare)
% funIdentita:
% Prende in input un valore x e restituisce lo
% stesso valore.
if exist('daDerivare','var')
    z=ones(size(x));
else
    z=x;
end
end
```

```
function z = funSigmoide(x,daDerivare)
% funSigmoide:
% Prende in input un valore x e restituisce il risultato
% della funzione sigmoide nel caso in cui la flag per la
% derivata sia inattiva, il valore derivato altrimenti.

% Viene richiesta la derivata
if exist('daDerivare','var')
    z=funSigmoide(x).*(1-funSigmoide(x));
else % Viene calcolato il valore della sigmoide
    z=1.0 ./ (1.0+exp(-x));
end
end
```

```
function z = funSommaQuadrati(x,y,daDerivare)
%funSommaQuadrati
% Dati due valori x e y calcola la somma dei quadrati.
```

```
% Se richiesto ritorna la derivata di questa funzione.
```

```
% Viene richiesta la derivata
if (exist('daDerivare','var'))
    z=x-y;
else
    z=0.5*sum((x-y).^2));
end
end
```

```
function out = ReLU(x,daDerivare)
%Funzione di attivazione rectified linear unit.
if exist('daDerivare','var')
    if x < 0
        out=0;
    else
        out=1;
    end
else
    if x < 0
        out=0;
    else
        out=x;
    end
end
end
```

Riferimenti bibliografici

- [1] Pascal Vincent et al. «Extracting and composing robust features with denoising autoencoders». In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 1096–1103.