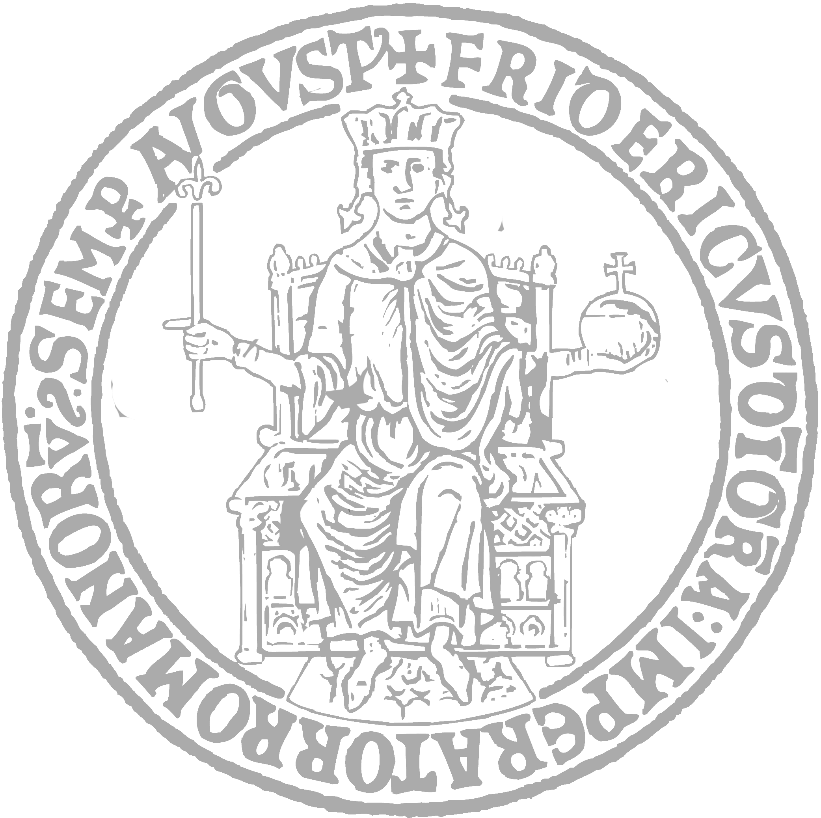
MLeA – Mod. B 2017/2018  
Progetto 5

Prof. Roberto Prevete

Djihad Boukara – N97000275

Emanuele Cioffi – N97000277



# Indice Generale

Sommario

[1. Indice Generale 2](#_Toc535941042)

[2. Indice delle figure 3](#_Toc535941043)

[3. Traccia 4](#_Toc535941044)

[4. Descrizione del problema 4](#_Toc535941045)

[5. Dataset 5](#_Toc535941046)

[6. Implementazione 5](#_Toc535941047)

[A. La rete neurale 5](#_Toc535941048)

[B. L’addestramento 6](#_Toc535941049)

[7. Test Plan 11](#_Toc535941050)

[8. K-Folding 13](#_Toc535941051)

[A. Dati Risultato 13](#_Toc535941052)

[B. Osservazioni sul K-Folding 15](#_Toc535941053)

[9. Test finali 15](#_Toc535941054)

[10. Conclusioni 17](#_Toc535941055)

[11. Appendice 18](#_Toc535941056)

[A. Kfold.m 18](#_Toc535941057)

[B. Singolo Training (Mnist.m) 20](#_Toc535941058)

[C. PlotBar.m 22](#_Toc535941059)

# Indice delle figure

[Figura 1 - Esempio di creazione rete neurale 5](#_Toc535923971)

[Figura 2 – Train 6](#_Toc535923972)

[Figura 3 – BackPropagation 7](#_Toc535923973)

[Figura 4 - Forward Propagation 7](#_Toc535923974)

[Figura 5 – SoftMax 8](#_Toc535923975)

[Figura 6 - Cross Entropy 9](#_Toc535923976)

[Figura 7 - Calcolo Derivate dei pesi 9](#_Toc535923977)

[Figura 8 - Discesa del gradiente 10](#_Toc535923978)

[Figura 9 – K-fold 11](#_Toc535923979)

[Figura 10 - K-fold: sigmoide-identità 13](#_Toc535923980)

[Figura 11 K-fold: sigmoide - sigmoide 13](#_Toc535923981)

[Figura 12 K-fold: tangente iperbolica - identità 14](#_Toc535923982)

[Figura 13 K-fold: tangente iperbolica - ReLU 14](#_Toc535923983)

[Figura 14 K-fold: migliori 10 risultati per accuracy 15](#_Toc535923984)

[Figura 15 - Errore: online-minibatch 16](#_Toc535923985)

[Figura 16 - Validation Error: minibatch 16](#_Toc535923986)

[Figura 17 Validation Error: online 17](#_Toc535923987)

[Figura 18 - Risultati finali 17](#_Toc535923988)

# Traccia

**PARTE A.**

Progettazione ed implementazione di funzioni per simulare la propagazione in avanti di una rete neurale multistrato con almeno:

* due strati di pesi, con la sigmoide come funzione di output dei nodi interni e l'identità come funzione di output dei nodi di output.

(FACOLTATIVO: permettere all'utente di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di output per ciascun strato).

Progettazione ed implementazione di funzioni per la realizzazione della back-propagation per reti neurali multistrato con almeno:

* due strati di pesi, con la sigmoide come funzione di output dei nodi interni e l'identità come funzione di output dei nodi di output, con la somma dei quadrati come funzione di errore.

(FACOLTATIVO: permettere all'utente di realizzare la backpropagation con più di uno strato di nodi interni, con qualsiasi funzione di output per ciascun strato e con qualsiasi funzione di errore derivabile rispetto all'output).”

**PARTE B.**

Si consideri come input le immagini raw del dataset mnist. Si ha, allora, un problema di classificazione a C classi, con C=10. Si estragga opportunamente un dataset globale di N coppie (ad esempio, N=200). Si fissi la discesa del gradiente come algoritmo di aggiornamento dei pesi, ed una rete neurale con un unico strato di nodi interni. Si scelgano gli iper-parametri del modello, cioè eta della regola di aggiornamento ed il numero di nodi interni, sulla base di un approccio di cross-validation k-fold (ad esempio k=5).

Scegliere e mantenere invariati tutti gli altri "parametri" come, ad esempio, le funzioni di output e la funzione di errore. Se è necessario, per questioni di tempi computazionali e spazio in memoria, si possono ridurre (ad esempio dimezzarle) le dimensioni delle immagini raw del dataset mnist (ad esempio utilizzando in MATLAB la funzione imresize).

# Descrizione del problema

Per risolvere il problema di classificazione con apprendimento supervisionato, si vuole implementare un modello di rete neurale artificiale feed forward, multistrato e full connected, che permetta all’utente di specificare gli iperparametri e le funzioni di attivazione ed errore desiderate. La rete sarà sviluppata in maniera quanto più generale possibile, nell’ottica di poter accogliere input di diverse tipologie. L’ambiente di sviluppo scelto per implementare tale modello è MATLAB. Si vuole inoltre studiare tramite la tecnica di cross validation K-Fold, il miglioramento o il peggioramento delle performance di riconoscimento della rete neurale al variare degli iperparametri: learning rate, funzioni di attivazione (per ciascuno strato) e numero di nodi interni dello strato hidden.

Utilizzando il dataset per caratteri scritti a mano MNIST, dove le classi target discrete consistono dei caratteri da ‘0’ a ‘9’, verranno svolti test in ottica di classificazione (e non regressione), utilizzando come funzione d’errore la Cross Entropy, combinata con la funzione Soft Max per il post processing.

# Dataset

Il dataset impiegato per l’addestramento della rete è il database di caratteri scritti a mano MNIST. Tale dataset è composto da 60.000 esempi di caratteri numerici (da 0 a 9) corredati di label compilate a mano e di 10.000 esempi ulteriori per il testing. Il dataset è pubblicamente disponibile, e per leggerne i dati ed acquisire in campioni in formato leggibile per MATLAB, ci si è serviti delle funzioni sviluppate all’università di Stanford (allegate al progetto, si veda il file *loadMNISTImages.m*). Tali funzioni, una volta letto l’intero dataset, restituiscono una matrice di dimensioni 784\*60000 (per il training set, 784\*10000 per il test set), in cui le colonne rappresentano i singoli campioni, e le righe rappresentano le feature dei campioni. Le 784 feature rappresentano il totale dei pixel per singola immagine grande 28x28, e descrivono tramite un valore numerico compreso tra 0 (il bianco) e 255 (il nero) il contenuto del pixel (in scala di grigi). Un singolo campione del training/test può quindi essere visto come un vettore colonna della matrice generata. Tale vettore viene poi diviso valore per valore, per 255, per normalizzare i valori di input da consegnare alla rete.

# Implementazione

La rete neurale è stata quindi implementata in un’ottica di flessibilità e facilità di riutilizzo, specie considerando le necessità dettate dalla Cross Validation di poter eseguire l’addestramento su reti sempre diverse (nel senso di scelta di iperparametri). Perciò, la rete sviluppata permette all’utilizzatore di scegliere un numero arbitrario di strati hidden tramite l’utilizzo di array contenitori, di specificare le funzioni di attivazione per tali strati e per lo strato di output, ed infine di specificare la funzione d’errore da utilizzare. Sebbene gli script corredati per il testing siano pensati per l’utilizzo della rete con il dataset MNIST, nulla vieta di utilizzare la rete con altri tipi di dataset a patto di fornire alla rete i dati nel formato giusto.

## La rete neurale

Il cuore della rete neurale è osservabile nella funzione *neuralNet.m*, che permette di creare la rete neurale specificando i parametri desiderati:

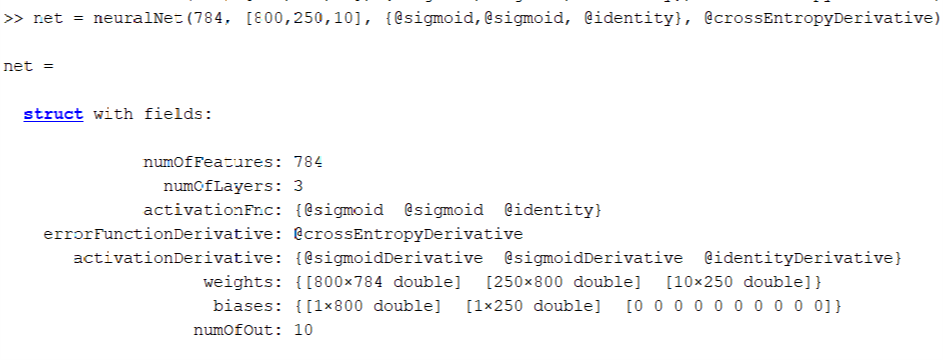


Figura 1 - Esempio di creazione rete neurale

In Figura 1 un esempio di creazione di rete neurale con due strati hidden rispettivamente formati da 800 e 250 nodi interni utilizzanti la funzione d’attivazione Sigmoide, e uno strato d’output da 10 nodi con funzione di attivazione Identità (il numero di nodi in output è in questo caso quello necessario al riconoscimento dei dati MNIST). La Cross Entropy è la funzione d’errore utilizzata in questo caso, ma è stata allegata l’implementazione anche per la somma dei quadrati.

Data la necessità di dover parametrizzare il calcolo della derivata (diverso per ogni funzione di attivazione) necessario nella backpropagation, si era pensato in un primo momento di adoperare uno dei toolbox di MATLAB per la valutazione delle espressioni e il calcolo delle derivate su queste; tale soluzione si è rivelata però troppo dispendiosa in termini di tempo, in un contesto dove i tempi di esecuzione sono già lunghi di per sé. Pertanto, si è scelto di adottare una soluzione diversa, ossia di fornire in allegato al codice alcune funzioni di attivazione (Sigmoide, Identità, Tangente Iperbolica, ReLU), e le relative implementazioni delle derivate, che l’algoritmo potrà poi chiamare a seconda dei casi. I codici relativi alle funzioni d’attivazione sono stati allegati al progetto (es: *sigmoid.m)*, insieme alle rispettive funzioni per la derivata (es: *sigmoidDerivative.m)*. L’utente dovrà quindi solo specificare le funzioni d’attivazione scelte per ogni strato tramite l’handle MATLAB (@nomeFunzione), e il sistema provvederà ad accoppiargli le relative funzioni per il calcolo della derivata all’atto della creazione della rete.

Per gestire i pesi della rete, è stato creato un array cell di matrici *weights*, che comprende una matrice di pesi per ogni layer della rete. Ognuna di queste matrici conta una riga per ogni nodo del layer relativo alla matrice, ed una colonna per ogni nodo del layer precedente. Osservando quindi tale matrice riga per riga, si otterrà l’insieme dei pesi/archi entranti nel nodo relativo alla riga, se invece la si osserva per colonna, si otterranno tutti gli archi uscenti del nodo dello strato precedente relativo alla colonna della matrice (ed entranti nei rispettivi nodi dello strato corrente). I bias sono rappresentati tramite pesi fittizi, come matrici con una riga e numero di colonne pari al numero di nodi del layer relativo per facilitare i calcoli.

## L’addestramento

La rete è stata sviluppata in modo da poter essere addestrata tramite l’algoritmo della discesa del gradiente e gli script corredati per l’utilizzo permettono di specificare un learning di tipo **Online, MiniBatch e Batch**.

Il cuore dell’addestramento è visibile nella funzione presente nel file *train.m*.



Figura 2 – Train

Come visibile in Figura 2, è possibile partizionare il Training Set in porzioni grandi *batchSize* ed effettuare l’addestramento della rete su tali porzioni. Un singolo training consisterà quindi di una chiamata alla funzione *backPropagation.m,* che dopo aver effettuato forward propagation (*forwardPropagation.m)* e calcolo dei delta, ritornerà le derivate rispetto ai pesi e i bias che verranno poi impiegate dalla funzione *gradientDescent.m* per effettuare l’aggiornamento dei pesi.



Figura 3 – BackPropagation



Figura 4 - Forward Propagation

In Figura 4 è leggibile l’implementazione della **Forward Propagation**. Il codice pre-alloca due cell array per un numero di elementi pari a quello degli strati della rete e calcolerà strato per strato i valori di A e Z, rispettivamente l’input e l’output di ogni nodo. Nel dettaglio, le formule implementate risultano essere:

E

Con ***l*** indice di layer e ***g***funzione d’attivazione per lo strato (con z che assumerà valore x per lo strato 1).  
Il calcolo da effettuare in forma matriciale è descritto da:

L’implementazione è quindi stata effettuata tramite moltiplicazioni tra matrici (caso batch/minibatch) o vettori per matrice (caso online) così ottenuti. Per chiarire, ilcell array ***a*** conterrà in prima posizione, la matrice risultante dal prodotto tra la matrice di BatchSize righe e 784 (l’input) colonne PER la matrice di righe pari al numero di nodi del layer 1 e 784 colonne da trasporre, i cui elementi saranno sommati al bias.

Vi è poi la possibilità di applicare una funzione di post process ai nodi z così ottenuti, in particolare nel nostro caso questa applicherà il softmax per una più chiara classificazione degli esempi rispetto alle label formate da un vettore di dieci componenti, di cui nove impostate al valore 0 ed una al valore 1, in corrispondenza dell’indice corrispondente al valore assegnato.



Figura 5 – SoftMax

Il Softmax viene impiegato quindi come post processing su funzioni di attivazione lineari per normalizzare la somma dei valori (nel nostro caso dei 10 output) portandola a 1, dando così maggiore *peso* in termini di influenza ai valori di grandezza superiore all’interno dei 10 possibili target.

In Figura 3 è possibile osservare l’implementazione della **Back Propagation**. Questa si compone di una chiamata alla forward propagation, del calcolo dei delta dello strato di output e di quelli precedenti a ritroso ed infine del calcolo delle derivate rispetto ai pesi. Il calcolo dei delta dello strato di output è effettuato secondo la formula:

Dove rappresenta la derivata della funzione d’attivazione dello strato , il valore di uscita di un nodo dello strato i-esimo prima della funzione d’attivazione. La derivata dell’errore sull’esempio rispetto all’output Z è stata quindi resa parametrica per permettere di specificare diverse funzioni d’errore. Nel caso d’utilizzo in questione (con Cross Entropy come funzione d’errore su un output a cui è stato applicato softmax), la funzione passata come parametro (*net*.*errorFunctionDerivative*) sarà quindi quella visibile in Figura 6.



Figura 6 - Cross Entropy

Per i restanti delta da calcolare negli strati interni, la formula implementata risulta essere:

Dove rappresenta la derivata della funzione d’attivazione per lo strato i-esimo, applicata sul valore pre-attivazione del nodo i allo strato l, e i termini della sommatoria sono i prodotti dei pesi uscenti dal nodo in questione per i delta associati al nodo dello strato successivo. I singoli delta vengono calcolati a livello vettore/strato da MATLAB (*delta{layer*} nel codice) moltiplicando in ciascuno la matrice dei pesi dello strato successivo per quella dei delta già calcolati al layer +1. Prima di terminare, la funzione back propagation calcolerà le derivate parziali rispetto ai pesi.



Figura 7 - Calcolo Derivate dei pesi

In Figura 7 è possibile vedere il codice per il calcolo delle derivate rispetto ai pesi. Il calcolo effettuato per ottenere tali derivate è:

Dove Znk assumerà valore Xi (ossia input) nel caso di n=1.



Figura 8 - Discesa del gradiente

In Figura 8 l’implementazione della discesa del gradiente, che aggiorna pesi e bias secondo:

# Test Plan

Per valutare quali iperparametri e quali combinazioni di funzioni di attivazione risultino più performanti in fase di classificazione, si è realizzata una procedura di cross validation su un sottoinsieme ristretto del training set, per testare la capacità di generalizzare le valutazioni da parte della rete. In particolare, si è utilizzata una strategia di K-Folding, visibile nello script *kfold.m*. Tale strategia consiste nel dividere un training set campione in k parti, fissare una delle parti come test set e utilizzare le restanti k-1 come porzione per l’addestramento, ripetendo tale operazione ruotando k volte la k-esima porzione di test. I risultati così ottenuti vengono poi mediati tra le varie rotazioni per fornire un risultato meno influenzato. Il k-fold è stato quindi impiegato per scegliere i migliori iperparametri tra un campione di combinazioni da valutare.

Il training set è stato limitato a 3200 esempi, e per i test effettuati si è scelto *dieci* come valore per il parametro K, formando così 10 raggruppamenti contenenti 320 esempi ciascuno, di cui 9 utilizzati per l’addestramento (in blu in Figura 9), ed uno per il test (in giallo), ruotando a turno su tutte e 10 le posizioni.

Figura 9 – K-fold

Il test plan programmato prevede di far spaziare i seguenti parametri, tra i valori di fianco riportati:

* Numero di **nodi** per lo strato interno:
  + 250
  + 500
  + 800
* Coppia di **funzioni** **d’attivazione** per strato interno ed output:
  + Tangente Iperbolica (tanH) – Unità Lineare Rettificata (ReLU)
  + Sigmoide – identità
  + Tangente Iperbolica – Identità
  + Sigmoide - Sigmoide
* **Eta** (Learning Rate)
  + 0.1
  + 0.01
  + 0.001
  + 0.0001

Tali parametri sono stati scelti a seguito di una breve scrematura preventiva, effettuata osservando il funzionamento delle funzioni su determinati parametri nelle fasi iniziali di sviluppo della rete; sono quindi stati scelti i parametri che forniscono un quadro d’insieme più descrittivo e che abbracci una variegata serie di casistiche, escludendo casi limite o poco interessanti, ad esempio learning rate troppo piccoli che risultano sicuramente meno efficaci di vari ordini di grandezza rispetto a tutti gli altri.

Le 9 fasi di addestramento e la singola di test che contraddistinguono un k-folding sono quindi eseguite per un totale di 4x3x4 volte, incrociando le possibili combinazioni, utilizzando la modalità di apprendimento minibatch.

# K-Folding

## Dati Risultato

Di seguito vengono mostrati i grafici che descrivono i dati estrapolati dalla cross validation per la ricerca degli iperparametri migliori. I colori delle barre rappresentano il learning rate utilizzato mentre i raggruppamenti sull’asse delle ascisse indicano il numero di hidden nodes. Il valore numerico ottenuto rappresenta la accuracy (percentuale di esempi correttamente classificati) media risultato della cross validation (quindi mediata su 10 addestramenti sulle rispettive 10 porzioni di training). I quattro grafici elencano i risultati delle quattro coppie di funzioni attivazione/output utilizzate come campione d’esame.

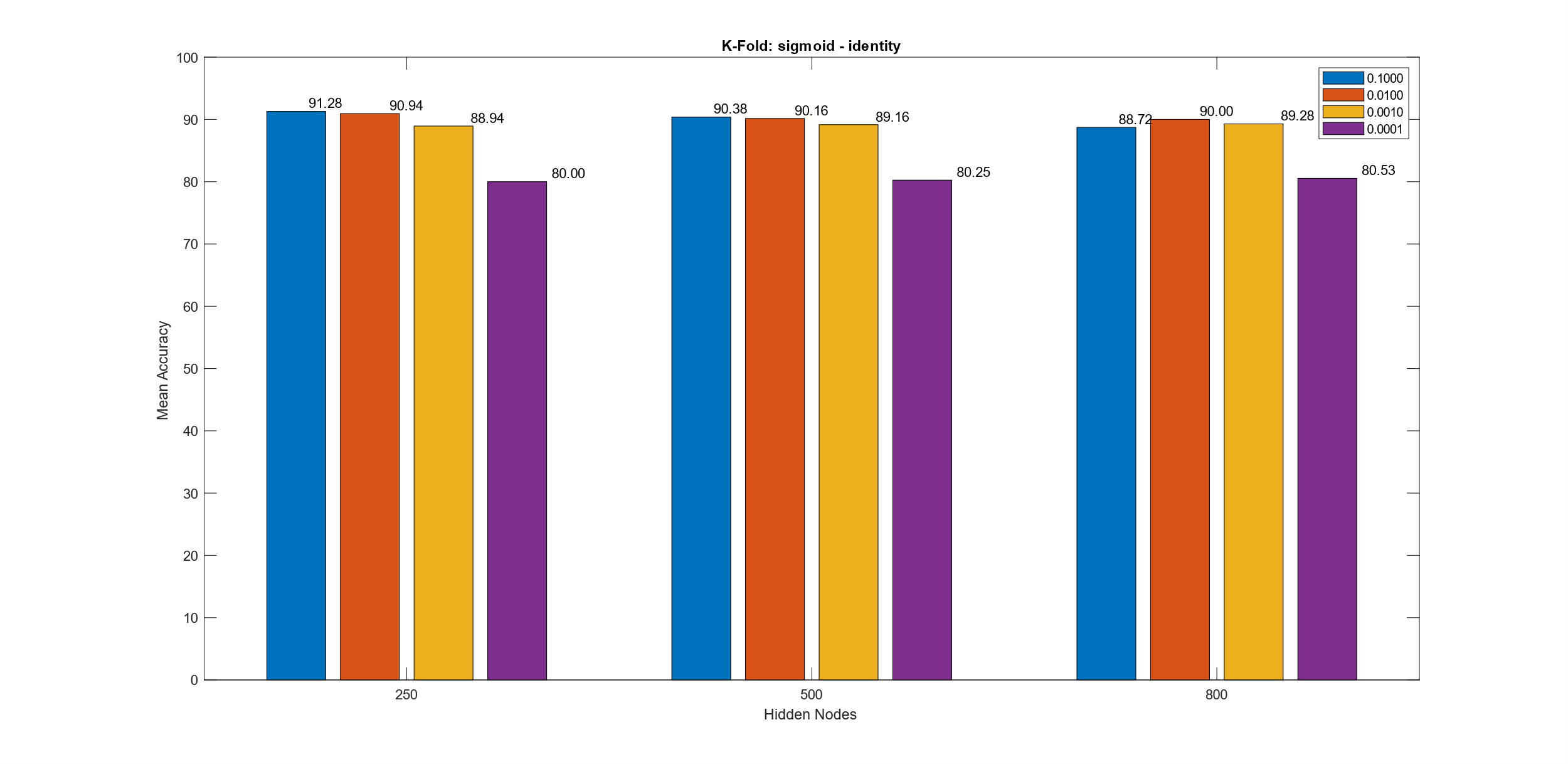


Figura 10 - K-fold: sigmoide-identità

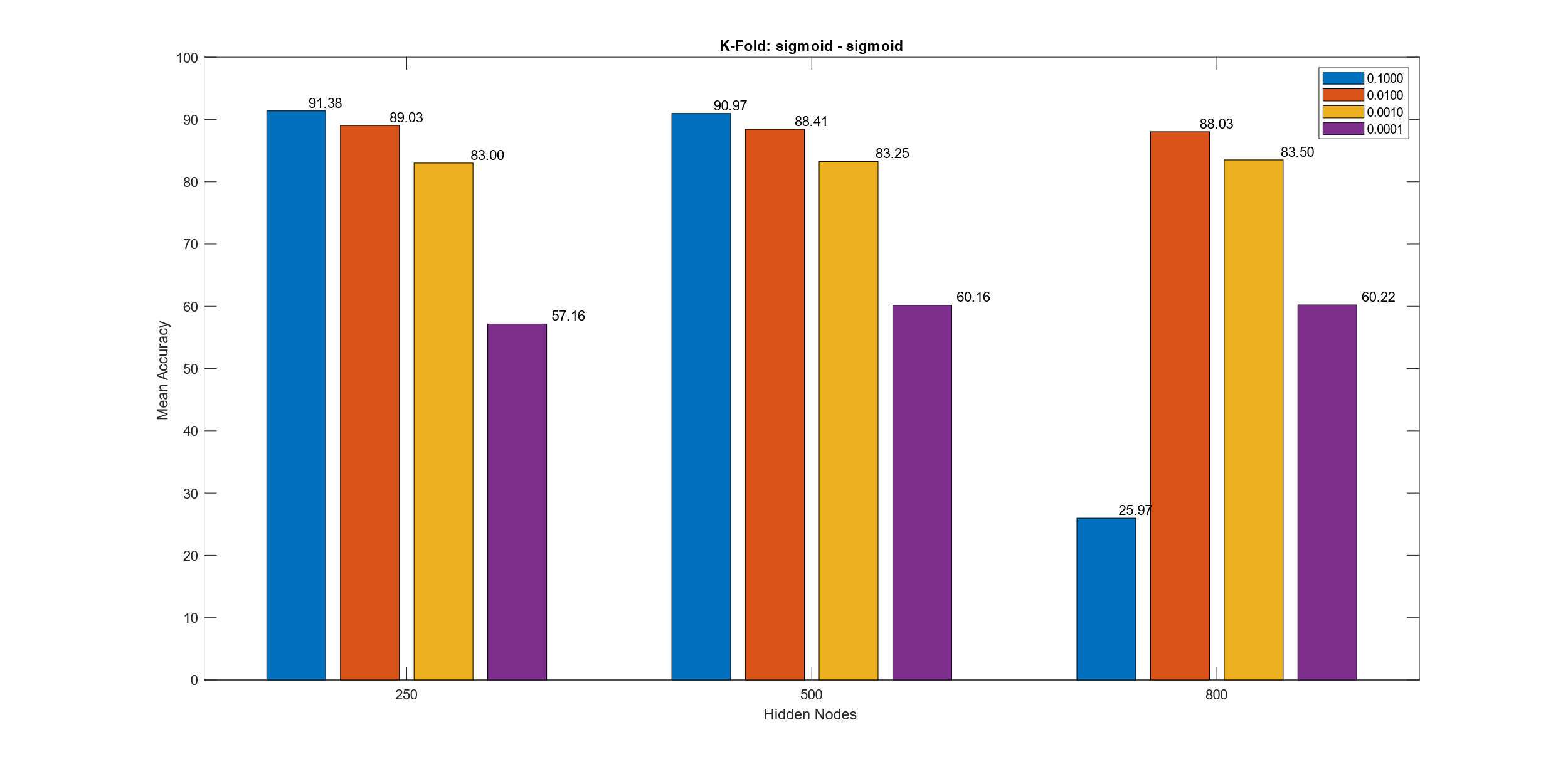


Figura 11 K-fold: sigmoide - sigmoide

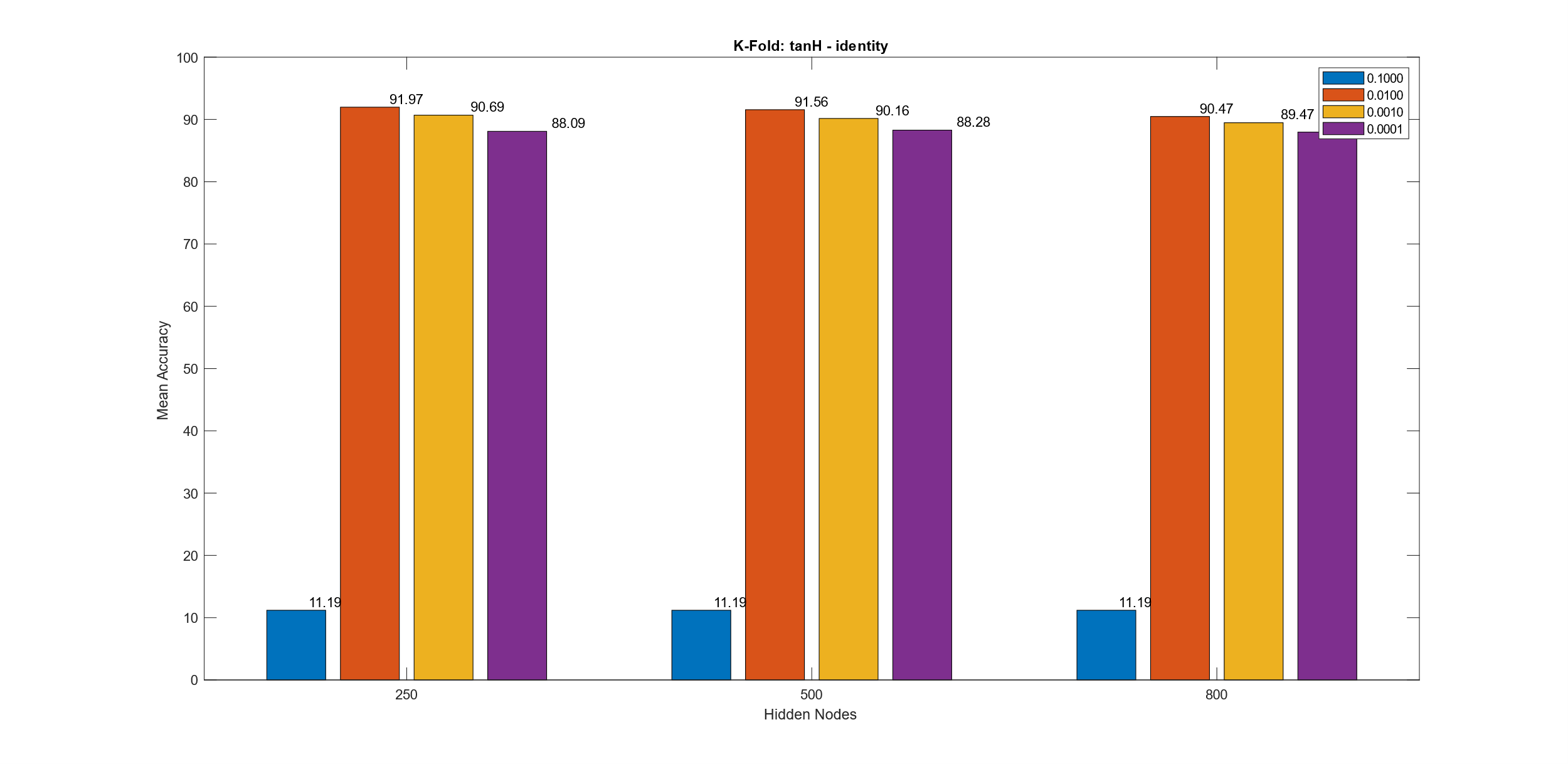


Figura 12 K-fold: tangente iperbolica - identità

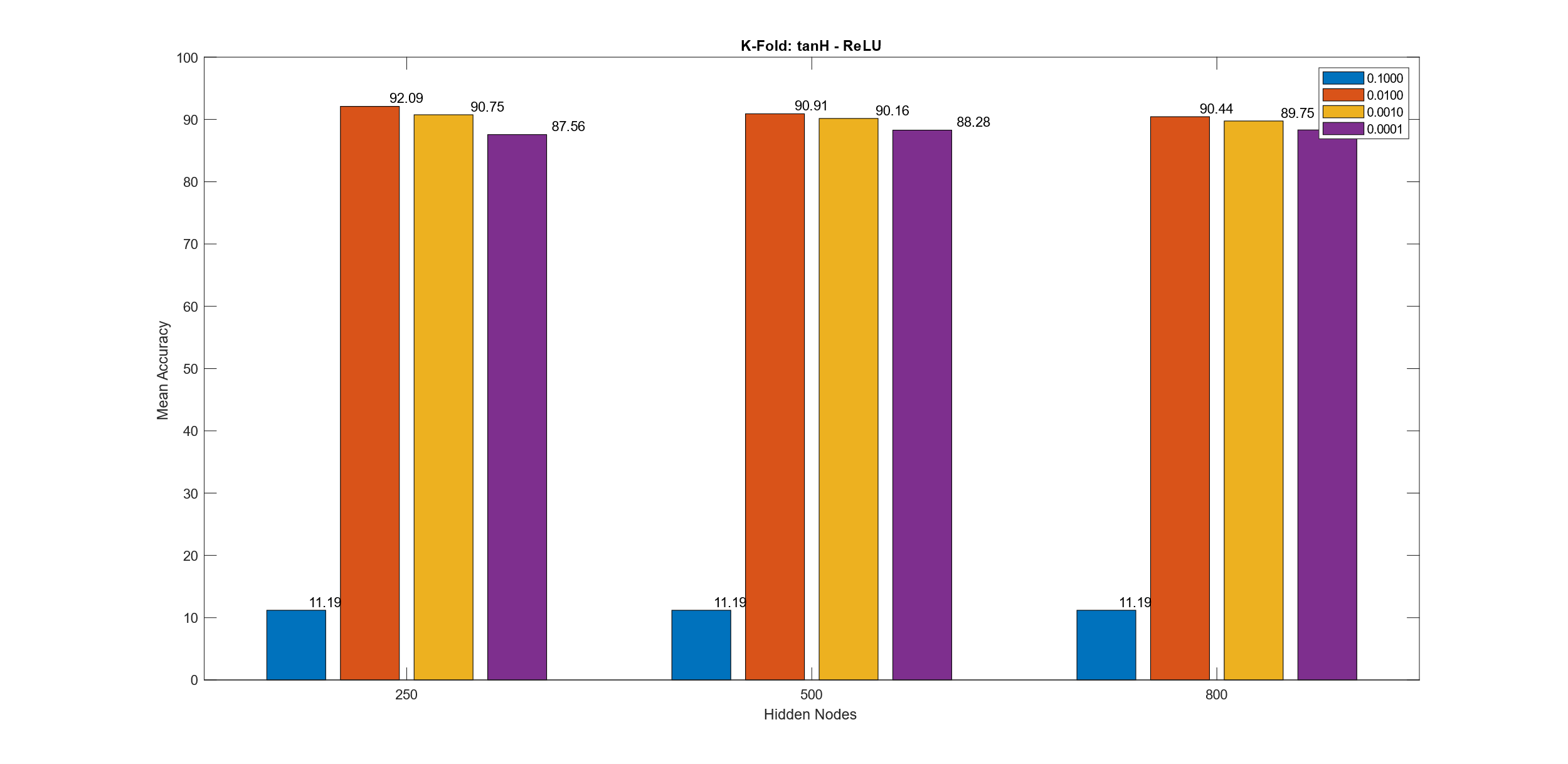


Figura 13 K-fold: tangente iperbolica - ReLU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hidden fun. - Output fun. | Eta | Hidden nodes | Mean Accuracy | Mean error | Std dev Acc | Std dev Error |
| **tanH-ReLU** | **0,01** | **250** | **92,09** | **130,99** | **1,66** | **47,25** |
| tanH-identity | 0,01 | 250 | 91,97 | 130,07 | 1,89 | 49,34 |
| tanH-identity | 0,01 | 500 | 91,56 | 153,54 | 1,47 | 54,9 |
| sigmoid-sigmoid | 0,1 | 250 | 91,38 | 497 | 1,84 | 5,57 |
| sigmoid-identity | 0,1 | 250 | 91,28 | 135,43 | 2,23 | 41,7 |
| sigmoid-sigmoid | 0,1 | 500 | 90,97 | 498,62 | 1,93 | 5,65 |
| sigmoid-identity | 0,01 | 250 | 90,94 | 117,62 | 1,98 | 37,18 |
| tanH-ReLU | 0,01 | 500 | 90,91 | 156,02 | 1,73 | 58,51 |
| tanH-ReLU | 0,001 | 250 | 90,75 | 105,85 | 1,85 | 28,62 |
| tanH-identity | 0,001 | 250 | 90,69 | 106,73 | 1,65 | 27,96 |

Figura 14 K-fold: migliori 10 risultati per accuracy

## Osservazioni sul K-Folding

Tra le considerazioni che è possibile effettuare osservando i grafici, c’è la differenza che si evince confrontando i test effettuati utilizzando la sigmoide come funzione d’attivazione oppure di ouput, e quando non la si usa. La presenza della sigmoide permette infatti il funzionamento della rete anche con un learning rate alto (0.1) e a differenza dei casi in cui non la si usa, l’apprendimento può avvenire correttamente. Lo stesso learning rate utilizzato nelle coppie tanH / identità (Figura 12) e tanH / ReLU (Figura 13) dà luogo al fallimento dell’apprendimento, causato dal problema dell’esplosione del gradiente, che produce un valore fisso di accuracy all’11%, e un errore totale propagato di 0.

Altra considerazione che è possibile fare è che l’aumento del numero di nodi interni non si traduce automaticamente in un miglioramento dei valori, ma anzi la variazione tra i risultati risulta minima, con valori che tendono a decrescere all’aumentare del numero di nodi, se non nel caso particolare della coppia sigmoide/sigmoide (Figura 11) dove un eta di 0.1 risulta troppo elevato se combinato ad un numero di nodi troppo grande (800), che porta ad un apprendimento fallato.

Per poter quindi estrarre una terna di parametri (coppia di funzioni, learning rate, numero di hidden nodes), si è scelto come fattore discriminante principale per scremare i 48 risultati ottenuti, l’accuratezza maggiore mediata sulle 10 rotazioni del k-fold. La misurazione per ogni terna è stata effettuata però anche sull’errore propagato (in media sulle 10 rotazioni), e sulla deviazione standard dell’accuracy e dell’errore. Ottenuti quindi i primi 10 risultati ordinati per accuratezza (visibili in Figura 14) si è cercata tra le possibili candidate, una terna che oltre ad avere ottenuto un punteggio alto in percentuale sull’accuracy, esibisse una deviazione standard della stessa non eccessiva rispetto alle altre, in modo da garantire la maggiore stabilità della rete. Il primo risultato per accuracy è risultato un buon compromesso tra questi criteri, esibendo la percentuale di accuracy più alta, ed avendo la terza deviazione standard più bassa tra le dieci esaminate.

Fatte tali considerazioni, si è scelto di eleggere la terna:

* Funzione attivazione= Tangente Iperbolica, Funzione di output= ReLU
* Eta= 0.01
* Hidden nodes= 250

Come migliore rappresentante della rete sviluppata, in termini di precisione della classificazione, stabilità, e capacità di generalizzare sul training set. Si è quindi proceduto ad effettuare dei test conclusivi per analizzare il comportamento della rete addestrata su un training set di maggiori dimensioni.

# Test finali

Per valutare la rete sviluppata, dopo aver effettuato il k-fold ed aver scelto i parametri di apprendimento più significativi, sono stati sviluppati dei test in modalità Online e MiniBatch su un training set più vasto, e utilizzando un validation set per evitare l’overfitting.

Nella modalità di apprendimento Online, l’aggiornamento dei pesi segue dopo ogni addestramento su singolo esempio, realizzando una rete in grado di convergere in minor numero di epoche, impiegando però un tempo d’esecuzione maggiore. La modalità minibatch invece effettua l’addestramento contemporaneamente su porzioni di esempi di grandezza fissata (ad esempio 32 esempi per addestramento). La modalità batch (qui non trattata pur essendo stata sviluppata) può essere vista come un caso particolare della minibatch in cui l’aggiornamento dei pesi viene effettuato una singola volta per epoca sull’intero training set. Per i nostri test è stato scelto un batchSize (grandezza delle partizioni per il minibatch learning) di 32 ed un training set grande complessivamente 32000 esempi. La dimensione del batch size è stata scelta in potenza di 2 per facilitare la parallelizzazione dei calcoli di MATLAB. Per l’addestramento della rete, si è inoltre utilizzato un validation set di ulteriori 8000 esempi, in modo da studiare l’andamento della curva che descrive l’errore di classificazione dopo ogni epoca, su un set che non fosse quello di training, in modo da evitare l’overfitting. Per ciò è stato anche implementato un criterio di early stopping, che controlla ogni cinque epoche di addestramento, che l’errore generato sul validation set all’epoca corrente non sia superiore a quello generato nella quinta epoca precedente, e in caso contrario, termini l’addestramento anticipatamente.

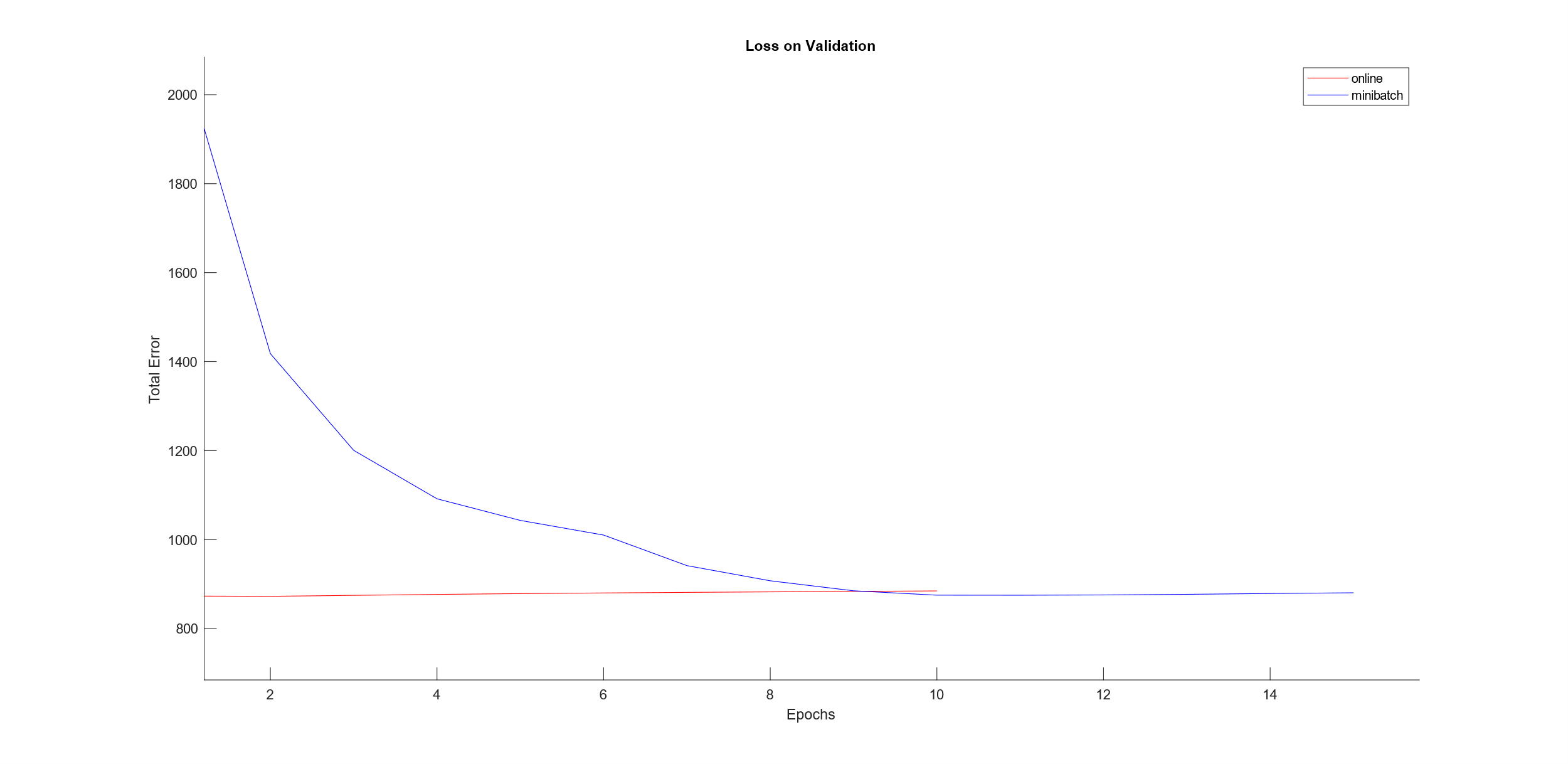


Figura 15 - Errore: online-minibatch

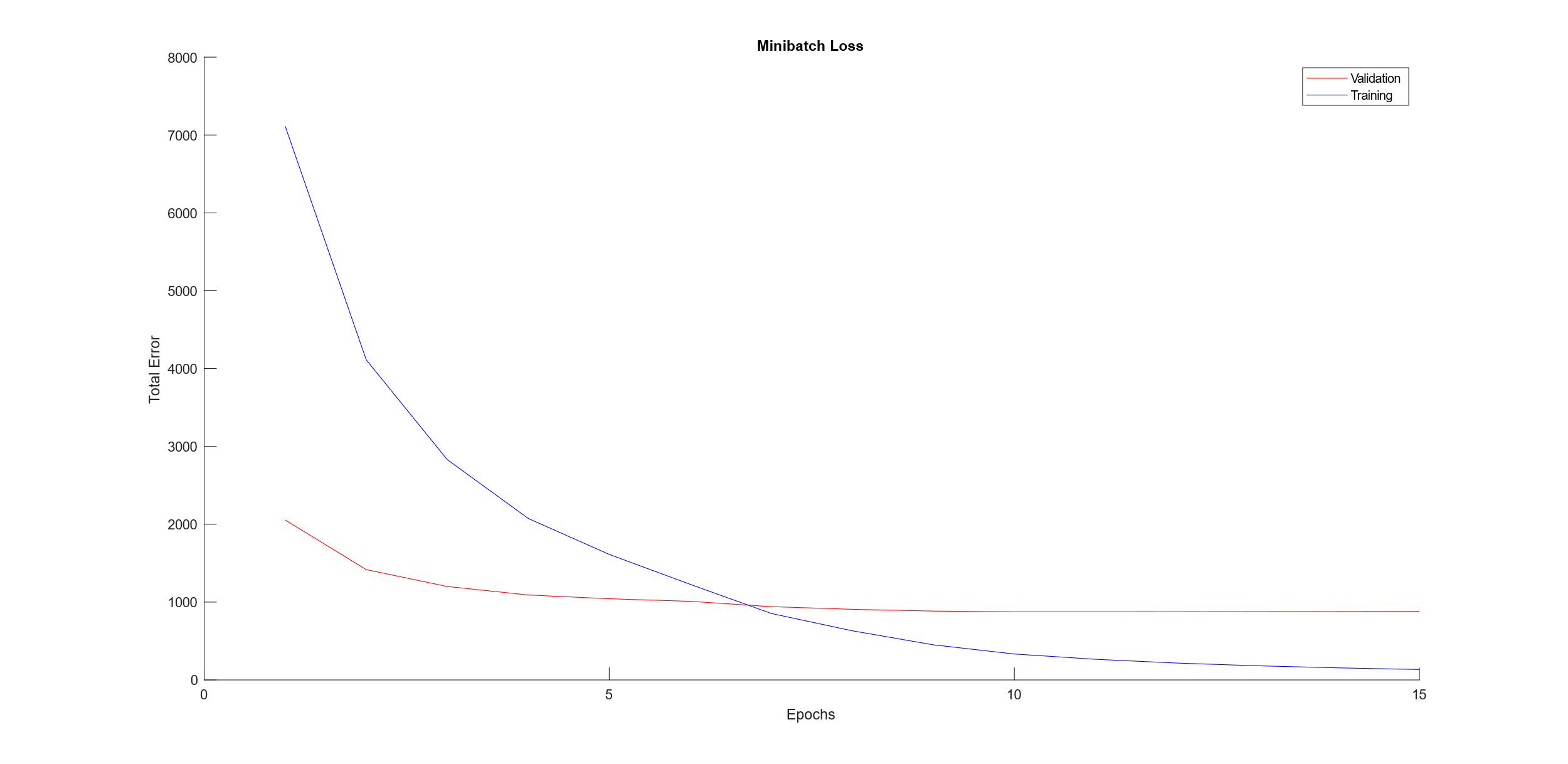


Figura 16 - Validation Error: minibatch

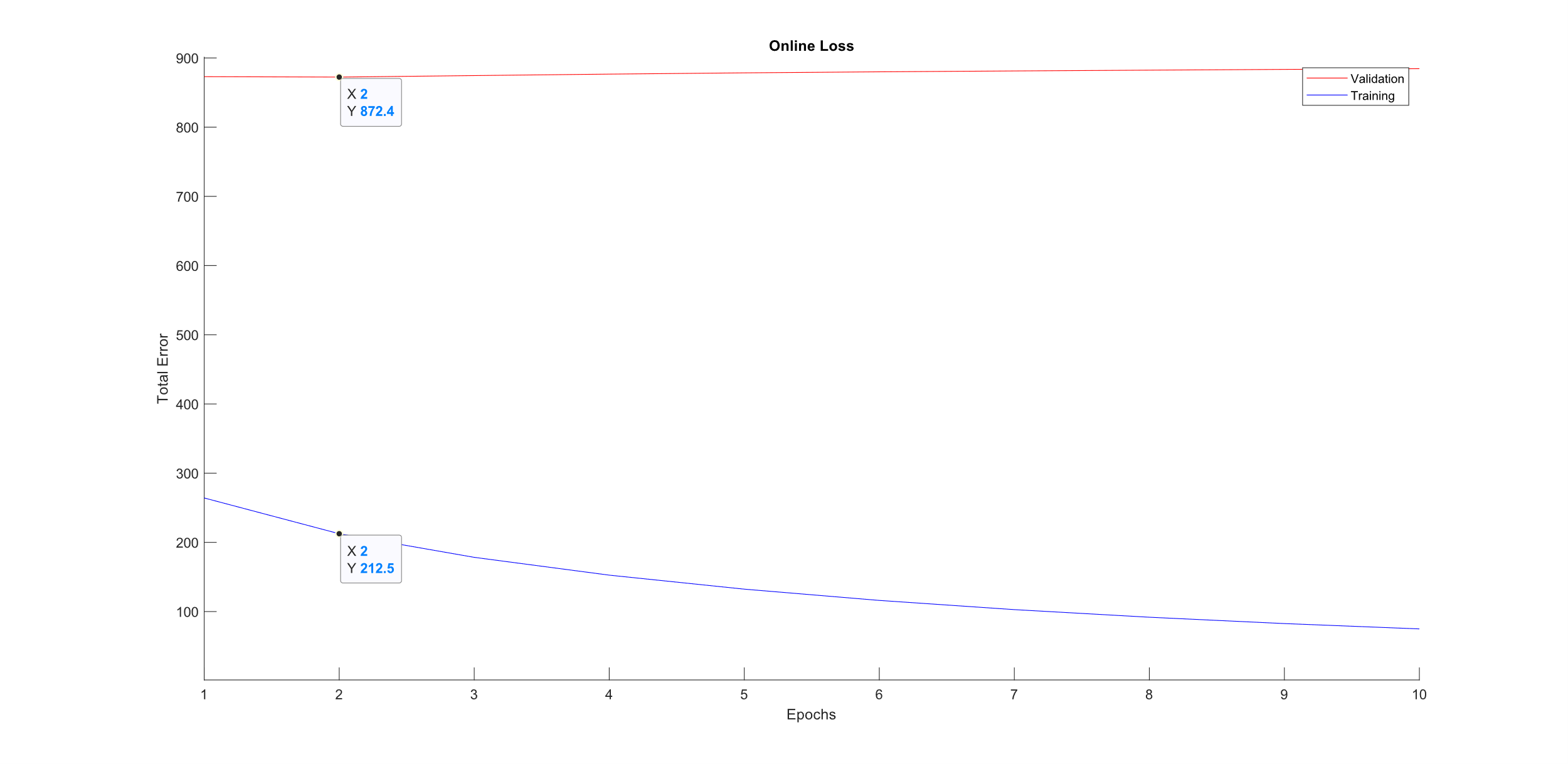


Figura 17 Validation Error: online



Figura 18 - Risultati finali

# Conclusioni

In Figura 15 sono stati messe a confronto le curve dell’errore ottenuto dalla rete sul validation set nei casi di Online learning e di MiniBatch learning. Dal grafico risulta immediatamente chiara la differenza in termini di tempo necessario alla rete per convergere al minimo quando si utilizzano i due approcci proposti. La modalità online riesce già dalla prima epoca di addestramento a fornire un risultato vicino a quello finale, mentre la modalità minibatch richiede un numero di epoche di molto superiore. In questa esecuzione, la rete addestrata in modalità online ha richiesto un numero di epoche pari a dieci per attivare il criterio di early stopping (con criteri più articolati probabilmente anche in numero minore), mentre la modalità minibatch ne ha richieste almeno 15 prima di poter terminare. Come visibile in Figura 18, a fronte di un margine di miglioramento dell’accuracy (calcolata sull’intero test set da diecimila esempi) trascurabile tra il risultato finale ottenuto tra online learning e minibatch, il tempo d’esecuzione impiegato tra i due (< 1 minuto in minibatch contro i 9 dell’online) favorisce non di poco l’utilizzo della modalità minibatch.

In Figura 16 sono state messe a confronto le curve dell’errore propagato su validation set e training set dalla rete. E’ possibile notare il punto in cui l’errore sul validation set viene minorato da quello sul training in corrispondenza dell’intervallo tra le epoche 5-10, ossia il punto in cui possiamo cominciare ad osservare un overfitting della rete sul training set. Il nostro criterio basilare di early stopping continua però fino alla quindicesima epoca di addestramento.

In Figura 17 sono stati fatti ragionamenti analoghi per la modalità online. Risulta in maniera evidente, come tale modalità restituisca valori accettabili già alla prima epoca di addestramento (l’esecuzione è stata forzata a calcolare almeno le prime dieci epoche in ogni caso, da cui l’assenza di stop alla quinta epoca).

In Figura 18 sono stati quindi elencati i risultati finali ottenuti.

Il valore di accuratezza migliore, ottenuto addestrando la nostra rete su un training set contenente 320000 esempi, e testando su un test set di 10000 esempi, ammonta quindi a 97.16% di precisione.

# Appendice

Seguono estratti dal codice sorgente degni di menzione. Vengono omesse per brevità porzioni di codice non interessanti.

## Kfold.m

Script utilizzato per la valutazione del k-folding. Il cuore dello script è rappresentato dai 4 cicli for innestati che variano tra le combinazioni di iperparametri e ruotano tra le k “fette” del training/validation set.

%load the training set

% 60000 (examples) x 784 (features)

train\_im = loadMNISTImages('train-images.idx3-ubyte')';

%load the training set labels

% 60000 x 10

train\_lb = loadMNISTLabels('train-labels.idx1-ubyte');

train\_lb = train\_lb';

train\_lb(train\_lb==0) = 10;

train\_lb = dummyvar(train\_lb);

%load the test set

% 784 x 10000

test\_im = loadMNISTImages('t10k-images.idx3-ubyte')';

% 10000 x 10

test\_lb = loadMNISTLabels('t10k-labels.idx1-ubyte');

test\_lb = test\_lb';

test\_lb(test\_lb==0) = 10;

test\_lb = dummyvar(test\_lb);

% Dimension of total training+validation set for k-fold, default: 300

ts\_size = 3200;

% Number of folds

k = 10;

epochNumber=50;

batchSize=32;

%compute slice size for k-folding

slice\_size = int32(ts\_size / k);

resized\_im = train\_im(1:ts\_size, :);

resized\_lb = train\_lb(1:ts\_size, :);

% Fixed hyperparams

errorDerivative = @crossEntropyDerivative;

errorFnc = @crossEntropy;

% Hyperparameters to test

netFnc = {{@tanH, @identity}, {@sigmoid, @identity}, ...

{@tanH, @ReLU},{@sigmoid,@sigmoid}};

netNodes = [250, 500, 800];

netEtas = [0.1,0.01, 0.001, 0.0001];

fprintf("Hidden function-Output function; Eta; Hidden nodes; Mean Accuracy; Mean error; Std Acc; Std Error\n");

bestAcc = -Inf;

bestErr = Inf;

elapsedTime = 0;

tic

%for each hyper param:

for fnc = 1: length(netFnc)

%used to store current error values for plotting

accNode = cell(length(netNodes),1);

nodeCounter=0;

for node = netNodes

%used to store current error values for plotting

nodeCounter=nodeCounter+1;

accEtas=cell(length(netEtas),1);

etaCounter = 0;

for eta = netEtas

etaCounter=etaCounter+1;

%store current error and accuracy for the k slice

k\_error = zeros(k, 1);

k\_accuracy = zeros(k, 1);

%rotate the leave one out k validation part

for i = 0: k-1

% Calculate the slice index for testing and

% validation part

start\_idx = slice\_size \* i + 1;

stop\_idx = start\_idx + slice\_size - 1;

%cut the submatrix for the training set

k\_train\_im = [resized\_im(1:start\_idx-1, :); resized\_im(stop\_idx+1:ts\_size, :)];

k\_train\_lb = [resized\_lb(1:start\_idx-1, :); resized\_lb(stop\_idx+1:ts\_size, :)];

%cut the submatrix for the validation set

k\_test\_im = resized\_im(start\_idx:stop\_idx, :);

k\_test\_lb = resized\_lb(start\_idx:stop\_idx, :);

%train the network with current params

net = neuralNet(784, [node, 10], netFnc{fnc}, errorDerivative);

%net = train(net, k\_train\_im, k\_train\_lb, eta, size(k\_train\_im, 1), 1);

for epoch = 1: epochNumber

net = train(net, k\_train\_im, k\_train\_lb, eta, size(k\_train\_im, 1), batchSize);

end

%correctly predicted examples

guessed = 0;

%value storing computed error

currError = 0;

%test on the validation part

[~, z] = forwardPropagation(net, k\_test\_im, @softmax);

%compute guessed examples and errore

for n = 1: size(z{1,2}, 1)

[val, idx] = max(z{1,2}(n,:));

if( idx == find( k\_test\_lb(n, :) ) )

guessed = guessed + 1;

end

currError = currError + sum(errorFnc(z{1,2}(n,:),k\_test\_lb(n, :)));

end

%store current error for compute average between k rotations later

k\_error(i+1) = currError;

k\_accuracy(i+1) = guessed / size(k\_test\_im, 1) \* 100;

end

%compute mean between al the k rotations

mean\_accuracy =mean(k\_accuracy);

mean\_error = mean(k\_error);

stdAcc = std(k\_accuracy);

stdError = std(k\_error);

fnc1 = func2str(netFnc{fnc}{1});

fnc2 = func2str(netFnc{fnc}{2});

str = sprintf("%s-%s; %.5f; %d; %.2f; %.2f; %.2f; %.2f\n", fnc1, fnc2, eta, node, mean\_accuracy, mean\_error,stdAcc,stdError);

fprintf("%s", str);

accEtas{etaCounter} = mean\_accuracy;

if (mean\_accuracy > bestAcc)

bestAccString = str;

bestAcc = mean\_accuracy;

end

if (mean\_error < bestErr && mean\_error > 0)

bestErrString = str;

bestErr = mean\_error;

end

end

accNode{nodeCounter}=accEtas;

end

elapsedTime = elapsedTime + toc;

%plot the current functions

plotBar(fnc1, fnc2, netNodes, netEtas, accNode);

end

fprintf('\nBest Accuracy: %s\n', bestAccString);

fprintf('Best Error: %s\n', bestErrString);

fprintf('Execution time: %d minutes\n', floor(elapsedTime/60));

## Singolo Training (Mnist.m)

Tramite la funzione che segue è possibile effettuare un addestramento completo in modalità online, e batch/minibatch ed ottenere i grafici (mostrati in documentazione) corrispondenti.

. . .

% 60000 x 784

train\_im = loadMNISTImages('train-images.idx3-ubyte')';

% 60000 x 10

train\_lb = loadMNISTLabels('train-labels.idx1-ubyte');

train\_lb = train\_lb';

train\_lb(train\_lb==0) = 10;

train\_lb = dummyvar(train\_lb);

% 784 x 10000

test\_im = loadMNISTImages('t10k-images.idx3-ubyte')';

% 10000 x 10

test\_lb = loadMNISTLabels('t10k-labels.idx1-ubyte');

test\_lb = test\_lb';

test\_lb(test\_lb==0) = 10;

test\_lb = dummyvar(test\_lb);

%size of training set and validation set

TsSize = 32000;

VsSize = 8000;

%cut the training/val set

trainingSet=train\_im(1:TsSize,:);

trainingSetLabel=train\_lb(1:TsSize,:);

validationSet=train\_im(TsSize+1:TsSize+1+VsSize,:);

validationSetLabel=train\_lb(TsSize+1:TsSize+1+VsSize,:);

%activation function for hidden layer

hiddenFnc = @tanH;

%activation function for output layer

outputFcn = @ReLU;

errorDerivative = @crossEntropyDerivative;

errorFnc = @crossEntropy;

batchSize = 32;

eta = 0.01;

epochNumber = 500;

hiddenNodes = 250;

%epoch number cointainer for early stopping

finalEpochsMiniB = epochNumber;

finalEpochsOnline = epochNumber;

% Create neural network

net = neuralNet(784, [hiddenNodes, 10], {hiddenFnc, outputFcn}, errorDerivative);

% used to revert to previous net in early stopping

lastNet=net;

lastError=Inf;

%store error and accuracy computed after each epoch on all test set

errorsOnlineValidation = zeros(epochNumber, 1);

errorsOnlineTraining = zeros(epochNumber, 1);

errorsBatchValidation = zeros(epochNumber, 1);

errorsBatchTraining = zeros(epochNumber, 1);

accuracyOnline = zeros(epochNumber, 1);

accuracyBatch = zeros(epochNumber, 1);

tic

%start training for each epoch (miniBatch)

fprintf('MiniBatch Training\n');

for epoch = 1: epochNumber

net = train(net, trainingSet, trainingSetLabel, eta, TsSize, batchSize);

% Evaluate Network

%evaluate accuracy on current network

correct = 0;

[~, out] = forwardPropagation(net, validationSet, @softmax);

%compute correctly guessed exemples

for i = 1: size(out{1,2}, 1)

[~, idx] = max(out{1,2}(i,:));

if( idx == find( validationSetLabel(i, :) ) )

correct = correct + 1;

end

end

%store accuracy for this epoch

accuracyBatch(epoch)=(correct/size(validationSet, 1))\*100;

%evaluate error on validation

errorsBatchValidation(epoch) = calculateError(out{1,2}, validationSetLabel, errorFnc);

%evaluate error on training

[~, out] = forwardPropagation(net, trainingSet, @softmax);

errorsBatchTraining(epoch) = calculateError(out{1,2}, trainingSetLabel, errorFnc);

fprintf('epoch: %3d; accuracy: %3.2f%%; errorVal: %3f,errorTrain: %3f\n', epoch,accuracyBatch(epoch) ,errorsBatchValidation(epoch),errorsBatchTraining(epoch));

%check early stopping every 5 epoch

if(mod(epoch,5)==0)

if(errorsBatchValidation(epoch)>lastError)

%early stop

net=lastNet;

finalEpochsMiniB=epoch;

break;

else

lastNet=net;

lastError=errorsBatchValidation(epoch);

end

end

end

elapsedTime = toc;

fprintf("time elapsed for execution: %.2f minutes, epochs:%d\n", floor(elapsedTime/60),finalEpochsMiniB);

% plotting loss on validation and training in minibatch mode

figure('Name', 'Error');

hold on

warning off;

legend('Error');

title('Minibatch Loss');

xlabel('Epochs');

ylabel('Total Error');

axis auto;

plot(errorsBatchValidation(1:finalEpochsMiniB), 'r','DisplayName','Validation');

plot(errorsBatchTraining(1:finalEpochsMiniB), 'b','DisplayName','Training');

hold off

drawnow;

%accuracy test with minibatch network

correct = 0;

[~, out] = forwardPropagation(net, test\_im, @softmax);

%compute correctly guessed exemples

for i = 1: size(out{1,2}, 1)

[~, idx] = max(out{1,2}(i,:));

if( idx == find( test\_lb(i, :) ) )

correct = correct + 1;

end

end

fprintf('Minibatch accuracy on test set %2f\n', (correct/10000)\*100);

lastNet=net;

lastError=Inf;

%start training for each epoch (online learning)

tic

fprintf('Online Training\n');

for epoch = 1: epochNumber

%train the network with a batchSize of 1 (online)

net = train(net, trainingSet, trainingSetLabel, eta, TsSize, 1);

% Evaluate Network

%evaluate accuracy on current network

correct = 0;

[~, out] = forwardPropagation(net, validationSet, @softmax);

%compute correctly guessed exemples

for i = 1: size(out{1,2}, 1)

[~, idx] = max(out{1,2}(i,:));

if( idx == find( validationSetLabel(i, :) ) )

correct = correct + 1;

end

end

%store accuracy for this epoch

accuracyOnline(epoch)=(correct/size(validationSet, 1))\*100;

%evaluate error on validation

errorsOnlineValidation(epoch) = calculateError(out{1,2}, validationSetLabel, errorFnc);

%evaluate error on training

[~, out] = forwardPropagation(net, trainingSet, @softmax);

errorsOnlineTraining(epoch) = calculateError(out{1,2}, trainingSetLabel, errorFnc);

fprintf('epoch: %3d; accuracy: %3.2f%%; errorVal: %3f,errorTrain: %3f\n', epoch,accuracyOnline(epoch) ,errorsOnlineValidation(epoch),errorsOnlineTraining(epoch));

%check early stopping every 5 epoch

if(mod(epoch,5)==0)

if(errorsOnlineValidation(epoch)>lastError)

%early stop

net=lastNet;

finalEpochsOnline=epoch;

break;

else

lastNet=net;

lastError=errorsOnlineValidation(epoch);

end

end

end

fprintf("time elapsed for execution: %.2f minutes, epochs:%d\n", floor(toc/60),finalEpochsOnline);

%accuracy test with online network

correct = 0;

[~, out] = forwardPropagation(net, test\_im, @softmax);

%compute correctly guessed exemples

for i = 1: size(out{1,2}, 1)

[~, idx] = max(out{1,2}(i,:));

if( idx == find( test\_lb(i, :) ) )

correct = correct + 1;

end

end

fprintf('online accuracy on test set %2f\n', (correct/10000)\*100);

% plotting loss on validation and training in online mode

figure('Name', 'Error');

hold on

warning off;

legend('Error');

title('Online Loss');

xlabel('Epochs');

ylabel('Total Error');

axis auto;

plot(errorsOnlineValidation(1:finalEpochsOnline), 'r','DisplayName','Validation');

plot(errorsOnlineTraining(1:finalEpochsOnline), 'b','DisplayName','Training');

hold off

drawnow;

% plotting loss of online and batch learning with the same number of epochs

figure('Name', 'Error');

hold on

warning off;

legend('Error');

title('Loss on Validation');

xlabel('Epochs');

ylabel('Total Error');

axis auto;

plot(errorsOnlineValidation(1:finalEpochsOnline), 'r','DisplayName','online');

plot(errorsBatchValidation(1:finalEpochsMiniB), 'b','DisplayName','minibatch');

hold off

drawnow;

## PlotBar.m

function plotBar(fnc1, fnc2, netNodes, netEtas, deviationsNodes)

%Plot K-fold error measure for given functions and hyperparams

% the plot describes the cross validation results grouped by hidden nodes

% number used, with each bar represeting the error standard deviation on

% given learning rate

%

%fnc1: activation function for hidden nodes layer

%

%fnc2: activation function for output nodes layer

%

%netNodes: cell array with number of hidden nodes used for the CV,

% ex:({250},{500},{800})

%

%netEtas: cell array containing the learning rates used in the CV

%

%deviationNodes: counter for how many hidden nodes number are there

graphName = sprintf('K-Fold: %s - %s', fnc1, fnc2);

figure('Name', graphName);

c = categorical(netNodes);

y = zeros(length(netNodes), length(netEtas));

for devNodesIndex = 1 : length(deviationsNodes)

tmp = cell2mat(deviationsNodes{devNodesIndex})';

y(devNodesIndex, :) = tmp;

end

b = bar(c,y);

title(graphName);

l = cell(1, length(netEtas));

for i = 1: length(netEtas)

l{i} = sprintf('%.4f',netEtas(i));

end

legend(b, l);

xlabel('Hidden Nodes');

ylabel('Mean Accuracy');

a = (1:size(y,1)).';

x = [a-0.20 a a+0.20 a+0.40];

for k=1:size(y,1)

for m = 1:size(y,2)

text(x(k,m),y(k,m),num2str(y(k,m),'%0.2f'),...

'HorizontalAlignment','center',...

'VerticalAlignment','bottom')

end

end

drawnow;

end

Funzione utilizzata per creare i grafici del k-folding.