Assignment II

1920-2-F1801Q151, Advance Machine Learning, 2019-2020

Università degli studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione

Villa Giacomo `807462`

28 Ottobre 2019

Indice

1 Dataset e Operazioni preliminari

Il **dataset** proposto per questo assignment conteneva `14000` matrici rappresentati **caratteri scritti a mano**, per la precisione le lettera dalla `P` alla `Z` dell'alfabeto inglese (estremi inclusi). La rappresentazione di ogni record è caratterizzata da una matrice **28x28** contenente valori numeri nel range `0 - 255` volti ad indicare l'intensità luminosa del pixel:

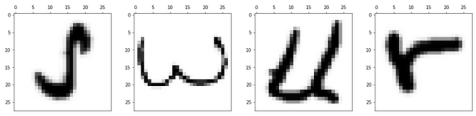

Figura 1: Esempio record da classificare

Veniva richiesto di **sviluppare una rete neurale per effettuare la classificazione** ed un **autoencoder** al fine di effettuare un'analisi visiva delle capacità di ricostruzione dell'input dello stesso.

Al fine di permettere il corretto funzionamento delle reti è stata **effettuata una fase di pre-processing** dove sostanzialmente si rimodellava il dato da matrice `28x28` a vettore di `784` elementi; in seguito, con l'obiettivo di **normalizzare**, si è proceduto a dividere ogni elemento per `255` (valore massimo assumibile da ogni elemento della matrice). Per ciò che riguarda la **tabella di verità** del dataset questa era composta da un valore compreso tra 16 e 26; ho proceduto a scalare di 16 così da ottenere un valore compreso tra 0 e 10 per poi ottenere una rappresentazione one-hot vector ponendo `1` laddove il valore scalato indicasse.

Per quanto riguarda le differenti classi (le lettere) queste presentano la seguente **distribuzione**:

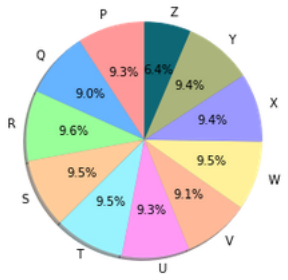

Figura 2: Distribuzione classi

In generale **la distribuzione delle classi non presenta particolari problematiche**, faccio notare semplicemente che la classe `Z` presenta meno istanza in generale rispetto alle altre classi.

2 Rete Neurale

Per implementare questa rete è stata utilizzata la libreria `Keras:The Python Deep Learning library`.

2.1 Struttura

In seguito alla rimodellazione della matrice in un vettore e alla normalizzazione dei valori di questo in un range compreso tra `0` ed `1`, ho **implementato una rete** con `784` **neuroni di input** (elementi vettore), **due layers nascosti** (rispettivamente di `512` e `256` neuroni) ed un **layer di output** con **11** neuroni rappresentanti le possibili classi. In generale ho:

```
Model: "sequential_308"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_916 (Dense)            (None, 512)               401920
_________________________________________________________________
dropout_543 (Dropout)        (None, 512)               0
_________________________________________________________________
dense_917 (Dense)            (None, 256)               131328
_________________________________________________________________
dropout_544 (Dropout)        (None, 256)               0
_________________________________________________________________
dense_918 (Dense)            (None, 11)                2827
_________________________________________________________________
activation_271 (Activation)  (None, 11)                0
=================================================================
Total params: 536,075
Trainable params: 536,075
Non-trainable params: 0
```

Figura 3: Distribuzione classi

La scelta del numero di neuroni e dei layers è stata guidata da una **serie di test** volti a comprendere la dimensione ottimale, ho seguito la tradizionale **logica del "triangolo"** con una decrescita dei neuroni verso il layer di output; la configurazione attuale, dati una serie di test empirici, risulta essere la più performante. Inoltre, data la numerosità dei record (14000), ritengo che **i parametri richiesti dalla rete possano essere correttamente appresi**.

Significato e motivazioni dei layers di Dropout verrà spiegato nella sottosezione `2.3 Ottimizzazione e Regolarizzazione` di questa documentazione.

2.2 Apprendimento della rete

Per quanto riguarda la **funzione di attivazione dei layers nascosti** ho deciso di utilizzare la funzione `ReLu`, la stessa scelta è stata presa anche nello scorso assignment; ancora una volta la motivo sottolineando come lo stato dell'arte la veda protagonista e di come permetta, ad ogni propagazione dell'errore, di non avere tutti i neuroni attivi. Questo rende tutto il **meccanismo di funzionamento più efficiente**.

Come **funzione di attivazione del layer di output** ho deciso di utilizzare una `Softmax` in quanto, il problema in questione, è una **classificazione multiclasse**; di fatto la funzione `Softmax` mi permette di ottenere un vettore che può essere trattato come un vettore di probabilità, che dunque mi permetterà di adottare la `Categorical Crossentropy` come **funzione di loss** al fine di minimizzare l'entropia tra ciò che dovrebbe essere (nella versione one-hot vector) e ciò che effettivamente il mio modello predice.

La **dimensione dei batch** è ricaduta su **256**, il valore mi sembra abbastanza corretto per far si che la discesa mediante gradiente segui un buon andamento. Ho **effettuato test** con differenti valori e non sembra cambiare le performance, reputo comunque corretto mantenere la dimensione definita; il numero relativamente alto di record fornisce un'ulteriore giustificazione alla dimensione di questo parametro.

Il **numero di epoche** è stato impostato a **200**; tuttavia, come verrà spiegato nella sezione successiva, **tale valore non viene mai effettivamente raggiunto**.

2.3 Ottimizzazione e Regolarizzazione

Per quanto riguarda la scelta della **funzione di ottimizzazione** questa è ricaduta su `Adadelta`; la scelta è **giustificata da un test** da me eseguito. Ho effettuato una 10 CV stratificata valutando i seguenti diversi ottimizzatori: `SGD`, `RMSprop`, `Adagrad`, `Adadelta`, `Adam`, `Adamax`, `Nadam`; sono andato a confrontare **l'accuracy media sui 10 fold di test** per ogni ottimizzatore. Il confronto sull'accuracy è giustificato in quanto il dataset non presenta situazioni di sbilanciamento per le classi, dunque la scelta può basarsi su questo indice.

Fornisco dunque il risultato del test da me effettuato:

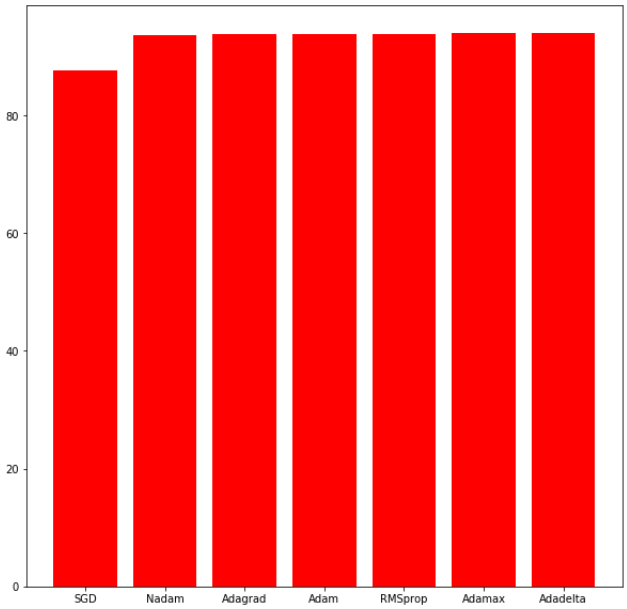

Figura 4: Livello accuracy medio sui fold di test

Possiamo notare come le d**differenze** tra i primi sei ottimizzatori (partendo dalla destra) **presentano differenze minime** e sicuramente non statisticamente rilevanti; ho tuttavia deciso di utilizzare quello che mi ha permesso di ottenere il risultato migliore nella sessione.

Per quanto riguarda le **operazioni di regolarizzazione** sono stati implementati dei layers di `Dropuot` (come mostra la figura nella sottosezione `2.1 Struttura` di questa documentazione); ho impostato un **coefficiente di 0.4** affinché, ad ogni iterazione, il 40% dei neuroni per ogni livello risulti essere addormentato. In questo modo cerco di **contrastare la possibilità di cadere in situazione di overfitting**.

Ho inoltre implementato un `Early Stopping` al fine **fermare la fase di fitting** nel momento in cui l'accuracy (sul validation set), per un periodo di 10 epoche consecutive, non registrava un miglioramento; tale scelta mi ha portato ad ottenere fasi di training lunghe dalle 17 alle 44 epoche.

2.4 Performance

Per quanto riguarda la **validazione del modello**, ho utilizzato una **10 CV** stratificata; questo è motivato dal fatto che il numero di record permette di adottare questa strategia, inoltre mi offre risultati più solidi per quanto riguarda la stima delle performance. Ad ogni **iterazione il test viene diviso** (80% e 20%) in **test set** effettivo e **validation set** al fine di poter osservare le curve di loss. Fornisco in tabella i valori medi sui fold di test di Precision e Recall data ogni classe:

Performance		
	Precision	**Recall**
P	*0.952*	*0.95*
Q	*0.912*	*0.931*
R	*0.926*	*0.919*
S	*0.977*	*0.972*
T	*0.938*	*0.947*
U	*0.919*	*0.925*
V	*0.886*	*0.908*
W	*0.964*	*0.963*
X	*0.959*	*0.932*
Y	*0.911*	*0.889*
Z	*0.961*	*0.957*

In generale le **performance risultano essere ottime**, attestandosi di fatto sopra il *90%* per la maggior parte dei casi; uniche eccezione la *Precision* di **V** che si ferma poco sotto (confondo **Y** e **U** per **V**) e la *Recall* di **Y** (dovuto tendenza di etichettare con **V** istanze di **Y**). Ulteriore conferma arriva dalle matrici di confusione dei vari fold:

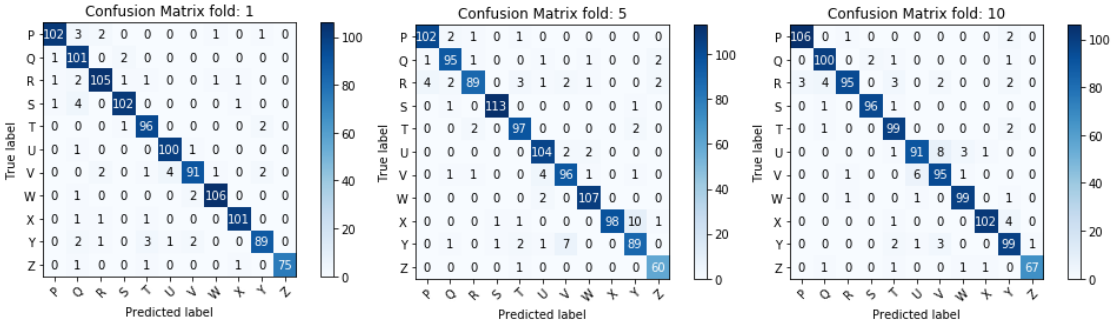

Figura 5: Matrici di confusione dei fold 1, 5 e 10

Per quanto riguarda l'**accuracy media** sui 10 fold di test questa è pari a `93.52%` mentre la f1-score weighted media è pari a `0.935%`.
NB: la numerosità delle istanze delle varie classi varia in quanto eseguo uno split sul fold di test al fine di valutare l'andamento durante il training mediante un validation set; ritengo la numerosità delle varie classi, dati i vari fold di test, in ogni caso abbastanza rappresentativa e in linea con le percentuali di numerosità del dataset originale.

Ho inoltre effettuato uno **studio** per quanto riguarda la **capacità discriminatoria** dei due livelli della mia rete neurale; ho **valutato** le capacità, rispettivamente, alla **fine del primo layer nascosto e alla fine del secondo** (di fatto prima dell'output); i risultati sono i seguenti:

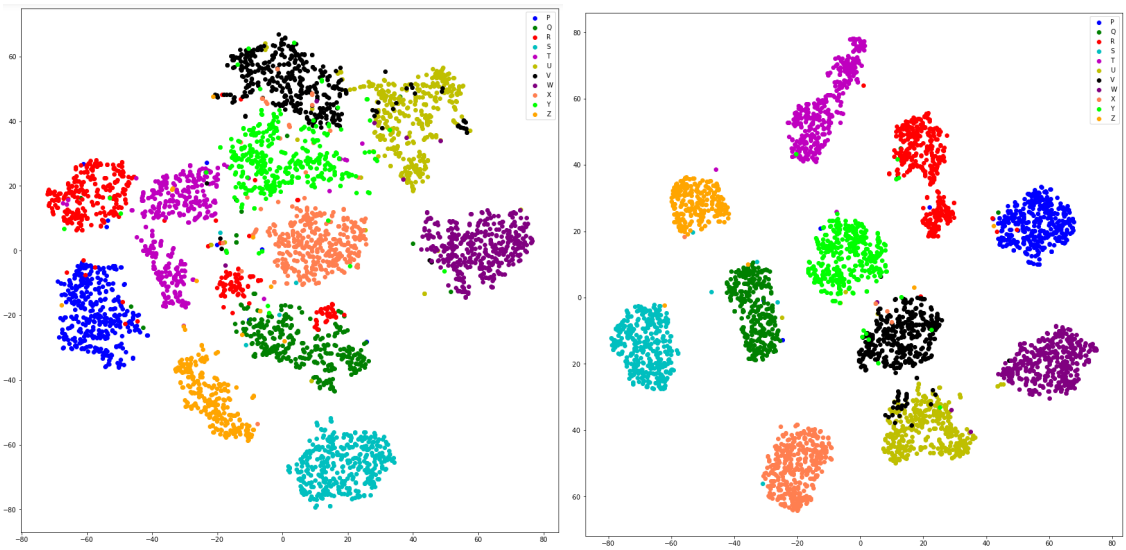

Figura 6: Capacità discriminatoria del modello dopo il primo e il secondo layer nascosto dati i primi 4000 record

Risulta possibile notare come **la capacità discriminatoria aumenti sensibilmente dal primo al secondo layer nascosto**; le immagini suggeriscono, come poi è accertato dai risultati delle varie performance, che **il modello ha una buona capacità discriminativa** (il modello lavora in dimensioni maggiori rispetto alla rappresentazione).

Fornisco anche le **curve** per quanto riguarda l'*accuracy* e *loss*:

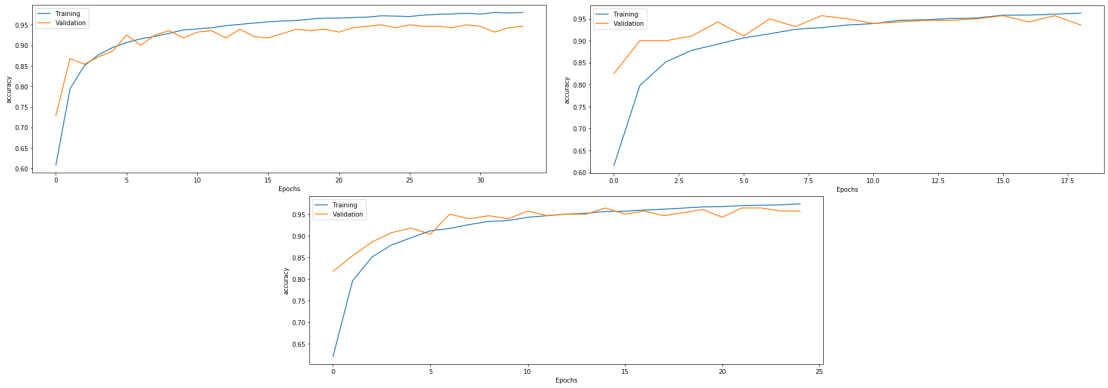

Figura 7: Accuracy sulle fold 1, 6, e 9

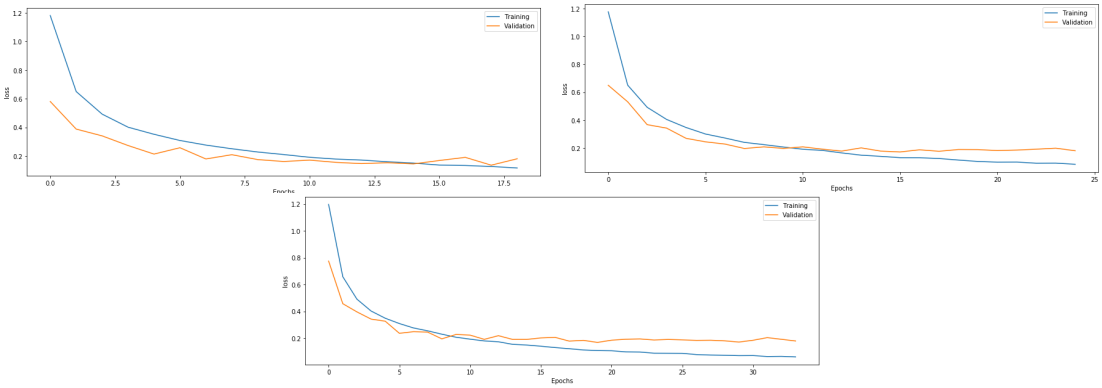

Figura 8: Loss sulle fold 1, 6, e 9

Notare come il **numero di epoche sull'asse delle x cambi**, questo è **dovuto all'early stopping**. In generale l'andamento segue una tendenza crescente (per l'*accuracy*) o discendente (per la *loss*) senza distaccarsi troppo dai valori di training con valori alquanto buoni; ulteriore conferma delle **buone performance del modello**.

3 Autoencoder

Per implementare l'autoencoders è stata utilizzata la libreria `Keras:The Python Deep Learning library`.

3.1 Struttura

Per la struttura dell'autoencoder ho implementato una rete con `784` **neuroni di input** (elementi vettore), **encoder** di rispettivamente `128` e `64` per poi passare a un **decoder** di `128` e infine `784` neuroni. In generale:

```
Model: "model_51"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_35 (InputLayer)        (None, 784)               0
_________________________________________________________________
dense_462 (Dense)            (None, 128)               100480
_________________________________________________________________
dense_463 (Dense)            (None, 64)                8256
_________________________________________________________________
dense_464 (Dense)            (None, 128)               8320
_________________________________________________________________
dense_465 (Dense)            (None, 784)               101136
=================================================================
Total params: 218,192
Trainable params: 218,192
Non-trainable params: 0
```

Figura 9: Summary dell'autoencoders

Il **numero di record permette di apprendere i parametri** richiesti dalla rete, ritengo inoltre più corretto effettuare una "discesa" passando in maniera più "dolce" verso la compressione effettiva. **Ritengo** inoltre **sufficiente una riduzione da 784 a 64** della dimensione dell'input.

3.2 Apprendimento della rete

Per quanto riguarda la **funzione di attivazione** utilizzata nella parte centrale dell'**autencoder** ho deciso di utilizzare una `ReLu` in quanto risulta essere molto utilizzata allo stato dell'arte inoltre, grazie alla sua natura, tende a rendere inattivi certi neuroni durante la fasi di aggiornamento pesi.

Per la **funzione di ouput** ho scelto una `Sigmoid` in quanto abbiamo valori in output riscalati nel range 0, 1. Inoltre questo mi permette di utilizzare la `binary crossentropy` come **funzione di loss** per il problema.

Per quanto riguarda la dimensione dei **batch** ho utilizzato un valore pari a `256` in quanto mi sembra il più corretto al fine di permettere una **corretta discesa lungo il gradiente**; inoltre la quantità di dati permette di adottare questo approccio.

Il **numero di epoche** impostato è pari a `200` ho comunque impostato un early stopping, come verrà spiegato nella sezione successiva.

3.3 Ottimizzazione e Regolarizzazione

Per quanto riguarda la **funzione di ottimizzazione** ho utilizzato `Adam` in quanto ho potuto osservare come questa **sembri registrare miglioramenti della loss più accentuati** rispetto ad altri ottimizzatori. Per la **regolarizzazione** ho utilizzato un **early stopping** monitorando la **loss del validation set** al fine di bloccare il fitting nel momento in cui questa non osservi miglioramenti per più di 10 epoche. Nella versione finale vado a monitorare la loss adottando la stessa logica.

3.4 Performance

Per quando riguarda la **valutazione delle performance** ho effettuato uno **splitting del train set** in **training set** e **validation set** al fine di osservare il comportamento su dati non visti; i **valori della loss** in generale, data la strutturazione mostrata nella sezione `3.1 Struttura` di questa documentazione, si attestano intorno al `0.12` prima di terminare. Data la configurazione così impostata, **allenando poi su tutti i dati disponibili**, ottengo i seguenti **risultati nella fase di ricostruzione**:

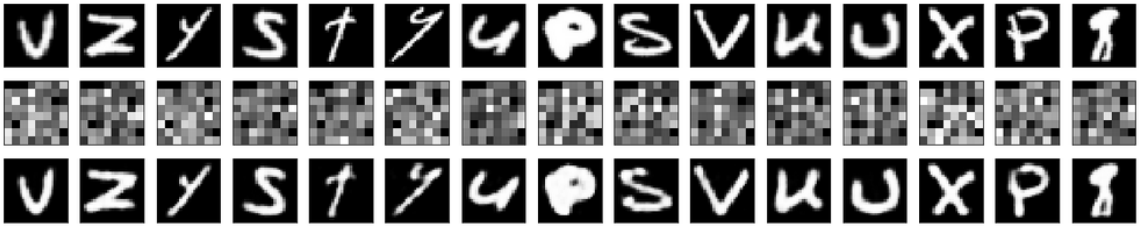

Figura 10: Ricostruzione immagini di test con strutturazione a due livelli e riduzione da 784 a 64

La **prima riga** mostra i caratteri in input, la **seconda riga** la rappresentazione, in questo caso, a 64 ed **ultima riga** la ricostruzione.

Sono andato, in via strettamente sperimentale, a **valutare diverse possibili configurazioni** per quanto riguarda la riduzione al fine di verificare le possibili differenze di rappresentazione.

I **test effettuati** si basano sempre **sull'autoencoder dotato di due livelli** di profondità, ho provato una **rappresentazione** `512` e `256` ed una a `64` e `32`; i risultati da me ottenuti sul test risultano essere i seguenti:

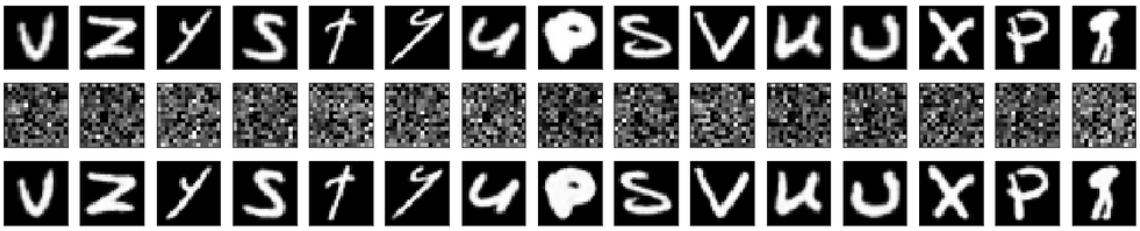

Figura 11: Ricostruzione immagini di test con strutturazione a due livelli e riduzione da 784 a 256

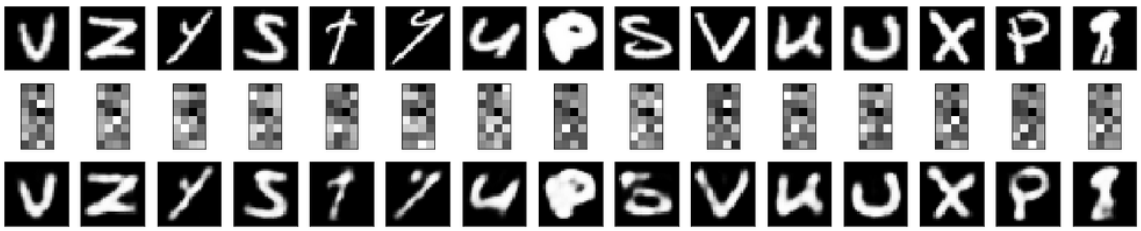

Figura 12: Ricostruzione immagini di test con strutturazione a due livelli e riduzione da 784 a 32

Ovviamente più si cerca di **ridurre lo spazio** di **rappresentazione** più la **ricostruzione** dell'immagine sarà **imperfetta**, **viceversa** più lo spazio di rappresentazione sarà simile allo spazio originale più l'**immagine** sarà **simile all'originale**.

4 Conclusioni

I **risultati ottenuti** dalla rete neurale sono **complessivamente buoni**, il testing sull'encoder ha permesso di valutare diverse possibili rappresentazioni dell'input.

Sarebbe interessante **provare** a valutare la rete neurale con una dimensione di

input ridotta (data presenza encoder) al fine di verificare la soglia per la quale le performance vedono una diminuzione e, al contempo, osservare se risulti essere possibile permettere una migliore discriminazione delle lettere più "problematiche"