



UNIVERSITÀ DEGLI STUDI DI TRENTO

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE

**Sviluppo di un sistema OCR con  
metodo probabilistico e Deep Neural  
Network**

**Corso di Computer Vision**

**Francesco Nicolini 209760  
Marco Dei Rossi 214649**

**Anno Accademico 2019-2020**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Metodo basato sulla probabilità</b>	<b>2</b>
2.1	Creazione mappa di probabilità . . . . .	2
2.1.1	Contornamento immagine . . . . .	3
2.1.2	Creazione matrice probabilità . . . . .	4
2.1.3	Implementazione confronto . . . . .	10
2.2	Risultati . . . . .	10
<b>3</b>	<b>MACHINE LEARNING &amp; DEEP LEARNING</b>	<b>11</b>
3.0.1	<i>HANDWRITING PREDICTION</i> . . . . .	20
3.1	CODICE . . . . .	21
3.2	codice LSTM . . . . .	22
3.3	Risultati . . . . .	26
<b>4</b>	<b>Confronto</b>	<b>29</b>
<b>5</b>	<b>STATE OF THE ART</b>	<b>31</b>

## 1 Introduzione

La presente relazione è atta alla presentazione del nostro lavoro per il riconoscimento e la digitalizzazione di un documento scritto a mano.

Sono quindi presentati due diversi algoritmi di lavoro per il rilevamento del segno scritto a mano.

Uno si basa su un confronto probabilistico, l'altro invece è sviluppato su un modello di apprendimento automatico.

Per capire al meglio ciò che abbiamo sviluppato, saranno presentati inoltre anche alcune pagine esplicative dei concetti basilari che permettono l'esistenza ed il funzionamento di questi modelli. Abbiamo inoltre cercato di presentare brevemente lo stato dell'arte di questo argomento.

I concetti presentati saranno: il *deep learning*, le reti neurali ricorrenti o *recurrent neural network* con particolare attenzione alle LSTM.

## 2 Metodo basato sulla probabilità

Riflettendo sulla metodologia con cui l'essere umano scrive, ci si può accorgere come i movimenti effettuati durante la scrittura siano dettati più dalla memoria muscolare che non dalla forma della lettera; si può riscontrare un simile comportamento anche in altre attività come ad esempio un violinista preme sulla tastiera senza riflettere, ma affidandosi al mero esercizio e quindi alla suddetta memoria.

Supponendo ora di avere delle immagini campione di una stessa lettera, contornate, in bianco e nero e scalate in una risoluzione standard, ipoteticamente per 100x100 pixel, si può presumere che la forma sia circa costante, data una persona con la propria scrittura. Fatta questa assunzione possiamo ipotizzare la creazione di una "mappa di probabilità" per ogni lettera o simbolo. Questo processo si basa sulla somma termine a termine di tutte le matrici campione di ogni singolo carattere, dividendo in seguito il risultato per il totale dei campioni, si normalizzano i valori rendendoli utilizzabili anche in presenza di una disparità di campionamento. Questa matrice, non essendo booleana, bensì con valori variabili da 0 a 1, rappresenta i punti con maggior o minor probabilità di intercettare un dato carattere. Ovvero i pixel la cui probabilità di intercettazione del tratto è prossima al 100 % assumono un valore normalizzato attorno ad 1, invece quelli che hanno probabilità 0 assumono valore nullo.

Ad esempio il carattere L avrà una mappa di probabilità con valori alti a sinistra e bassi in alto a destra.

Finito il processo di costruzione del *dataset* di probabilistico per ogni lettera, è possibile confrontare il simbolo da riconoscere con ogni elemento del suddetto *database* per stabilire quale lettera coincida meglio. Si pone l'accento sul fatto che la lettera con la maggior probabilità non è detto che sia quella reale bensì quella più simile. Questi e altri aspetti verranno approfonditi nelle conclusioni di questa sezione.

### 2.1 Creazione mappa di probabilità

Come detto precedentemente si genera una mappa di probabilità per ogni lettera avendo a disposizione un database. In questo caso abbiamo utilizzato il *MNIST DATABASE of handwritten digits*<sup>1</sup> per velocizzare il lavoro e non dover creare un database.

I primi passi da svolgere sono quelli che permettono un confronto equo tra tutte le immagini. Si rende quindi necessario standardizzare le immagini campioni seguendo i seguenti passi:

1. si contorna l'immagine in verticale
2. si contorna in orizzontale
3. si scala in una risoluzione standard
4. si converte in bianco e nero con un *threshold* adeguato

---

<sup>1</sup><http://yann.lecun.com/exdb/mnist/>

Successivamente si crea la mappa di probabilità assegnando a ogni pixel o elemento della matrice il valore dato dalla formula:

$$Mp_{ij} = \frac{1}{N} \sum_{k=1}^N I_{ij}(k) \quad (1)$$

Ovvero data una posizione  $i, j$  si sommano tutti i valori della specifica posizione di tutte le immagini campione  $P(k)$  e si divide per il totale dei campioni. Per la programmazione si è usato il *software* e le librerie di *image processing*.

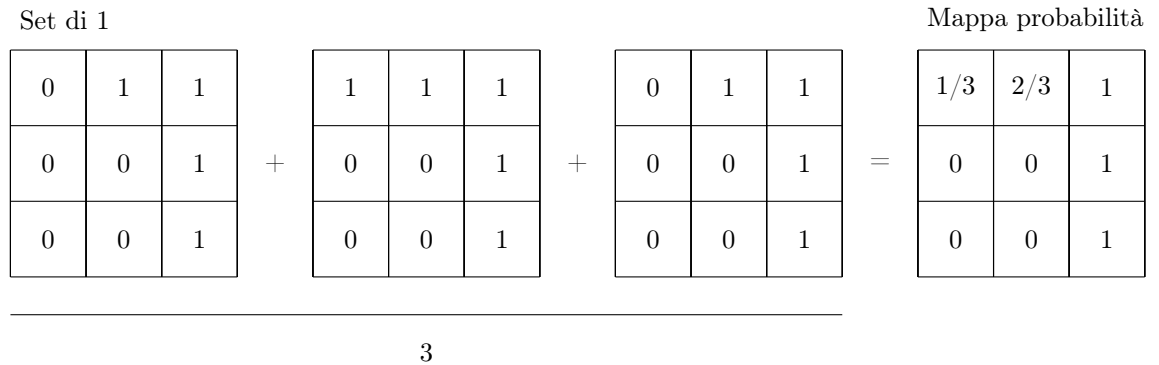


Figura 1: creazione mappa di probabilità

### 2.1.1 Contornamento immagine

Per un corretto confronto tra lettere è necessario che esse abbiano la stessa dimensione e risoluzione. Inizialmente siamo in possesso di un'immagine che rappresenta un'intera pagina con qualche riga scritta a mano. Questo file deve essere separato inizialmente in righe, quindi parole ed infine caratteri. I caratteri possono essere processati dal nostro codice, paragonandoli con le matrici probabilistiche. Come primo passo si rende necessario eliminare i bordi vuoti dell'immagine in tutte le direzioni. Considerando una situazione iniziale di immagine in bianco e nero *booleana*, dobbiamo togliere dalla suddetta immagine le righe e le colonne completamente bianche. Sono quindi ricopiate in una nuova immagine, che sovrascriverà quella vecchia che stiamo analizzando, solo le righe che presentano almeno un pixel nero. Per eliminare i bordi a destra e a sinistra è stato ripetuto il processo ruotando l'immagine di 90 gradi.

```

1  %Creazione nuova funzione cut
2  function [cutletter] = cut(letter)
3
4  %Definisco la dimensione
5  dim=size(letter);
6
7  %Defnisco matrice vuota
8  cutletter=[];
9  k=1;
10 sum=0;

```

```

11
12 for j=1:dim(1) %scorro righe
13     %Sommo tutti gli elementi sulla riga
14     for i=1:dim(2)
15         sum=sum+letter(j,i);
16     end
17
18     %Se il risultato e' diverso da zero
19     %copio la riga in cutletter
20     if sum~=0
21         cutletter(k,:)=letter(j,:);
22         k=k+1;
23     end
24
25     sum=0;
26 end
27
28 %riscalo l'immagine in formato standar
29 cutletter=imresize(cutletter,[28 28]);
30 end

```

### Commento

Riga 5: si trova la dimensione standard di ogni lettera del *database* che ci permette di stabilire le dimensioni definite come *standard*.

Riga 8: si crea una matrice vuota

Da riga 9 a riga 20: definiamo un indice iterativo *k* che ci permette di scorrere per righe. Qualora la somma di tutti i valori abbia come risultato 0 la riga non viene copiata. Nel caso invece in cui la somma sia diversa da 0, si iniziano a ricopiare le righe. Come accennato prima, lo stesso processo è anche iterabile sulle colonne semplicemente girando la matrice.

Riga 22: si scala l'immagine in una matrice 28x28, dimensione *standard*. La dimensione *standard* è definita dal *database* stesso che infatti contiene elementi con 785 colonne. Il che ci porta ad assumere che, tolta la prima riga di *labeling*, il resto delle colonne rappresenta tutti i pixel ordinati di una matrice 28X28 (784 valori).

### 2.1.2 Creazione matrice probabilità

Per la creazione delle matrici di probabilità abbiamo implementato il codice *train\_mnist.m*.

Esso crea un'immagine probabilistica per ogni carattere, in questo caso dalla cifra 0 alla 9, e crea uno *struct* con all'interno le varie matrici. Come accennato precedentemente si applica la formula 1 sulle lettere già standardizzate.

```

1     data = load ('mnist_train.csv');
2     labels = data(:,1);
3     dim=size(data);
4     images = data(:,2:dim(2));
5
6     %DATA 1
7
8     one_data=zeros(28,28);
9     k=0;

```

```
10
11     for i=1:dim(1)
12         if labels(i)==1
13             im=im2bw((reshape(images(i,:),28,28)'),0.5);
14             im=cut(im);
15             im=cut(im');
16             im=im';
17             one_data=one_data+im;
18             k=k+1;
19         end
20     end
21
22     for i=1:28
23         for j=1:28
24             one_data(i,j)= one_data(i,j)./k;
25         end
26     end
27
28     figure, imshow(one_data);
```

Al fine di creare la matrice di probabilità, una per ogni cifra, in scala di grigi definiti come media di tutti i pixel appartenenti a quella classe, definiamo la matrice *one-data* come matrice di zeri la cui dimensione è 28 X 28 pixel. Inoltre creiamo un contatore *k* che parta da 0.

Quindi, implementando un ciclo *for* che consente di elaborare tutti i valori della matrice in modo ordinato, applichiamo la formula 1 a tutte le matrici con stesso *label*. Nel nostro caso 1. Nello specifico della cifra 1 abbiamo definito la funzione *im* che consenta di condensare tutto questo processo in un unico comando usato per tutte le altre immagini.

Il valore finale di *k* permette di ricordare il giusto numero di esempi iterati. In tal modo si rende possibile la normalizzazione dell'immagine. Le funzioni *im2bw* e *reshape* sono funzioni standard Matlab. *im2bw* viene utilizzato per convertire un'immagine a colori in una binaria, con una soglia definita dal valore di *threshold*. La funzione *reshape* è necessaria per rimodellare tutte le immagini iniziali ad un unico standard (28X28).

Per una migliore comprensione del risultato ottenuto da questo codice, presentiamo un'immagine modificata e renderizzata a colori.

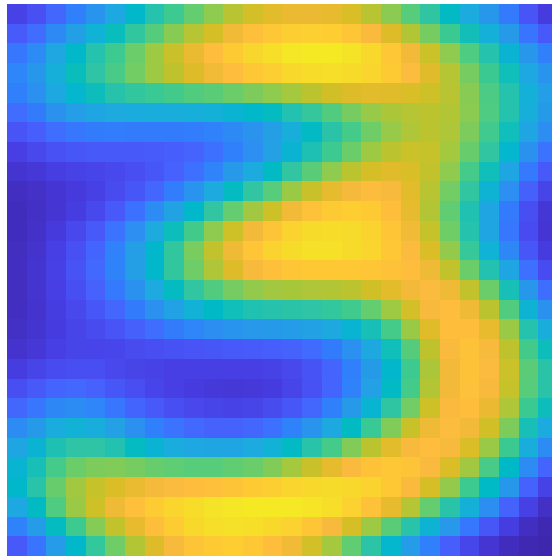


Figura 2:

Il processo è iterato per gli altri simboli.

```
1
2      %DATA 2
3
4      two_data=zeros(28,28);
5      k=0;
6
7      for i=1:dim(1)
8          if labels(i)==2
9              two_data=two_data+im2bw((reshape(images
10                 (i,:),28,28)') ,0.5);
11
12              k=k+1;
13          end
14      end
15
16      for i=1:28
17          for j=1:28
18              two_data(i,j)= two_data(i,j)./k;
19          end
20      end
21
22      figure , imshow(two_data);
23
24      %DATA 3
25
26      three_data=zeros(28,28);
27      k=0;
28
29      for i=1:dim(1)
30          if labels(i)==3
```



```
31     three_data=three_data+im2bw((reshape(images
32     (i,:),28,28)'),0.5);
33
34     k=k+1;
35     end
36     end
37
38     for i=1:28
39     for j=1:28
40     three_data(i,j)= three_data(i,j)./k;
41     end
42     end
43
44     figure, imshow(three_data);
45
46     %DATA 4
47
48     four_data=zeros(28,28);
49     k=0;
50
51     for i=1:dim(1)
52     if labels(i)==4
53     four_data=four_data+im2bw((reshape(images
54     (i,:),28,28)'),0.5);
55
56     k=k+1;
57     end
58     end
59
60     for i=1:28
61     for j=1:28
62     four_data(i,j)= four_data(i,j)./k;
63     end
64     end
65
66     figure, imshow(four_data);
67
68     %DATA 5
69
70     five_data=zeros(28,28);
71     k=0;
72
73     for i=1:dim(1)
74     if labels(i)==5
75     five_data=five_data+im2bw((reshape(images
76     (i,:),28,28)'),0.5);
77
78     k=k+1;
79     end
80     end
81
82     for i=1:28
83     for j=1:28
84     five_data(i,j)= five_data(i,j)./k;
```

```
85         end
86     end
87
88     figure, imshow(five_data);
89
90     %DATA 6
91
92     six_data=zeros(28,28);
93     k=0;
94
95     for i=1:dim(1)
96         if labels(i)==6
97             six_data=six_data+im2bw((reshape(images
98                 (i,:),28,28)')',0.5);
99
100         k=k+1;
101     end
102     end
103
104     for i=1:28
105         for j=1:28
106             six_data(i,j)= six_data(i,j)./k;
107         end
108     end
109
110     figure, imshow(six_data);
111
112     % DATA 7
113
114     seven_data=zeros(28,28);
115     k=0;
116
117     for i=1:dim(1)
118         if labels(i)==7
119             seven_data=seven_data+im2bw((reshape(images
120                 (i,:),28,28)')',0.5);
121
122         k=k+1;
123     end
124     end
125
126     for i=1:28
127         for j=1:28
128             seven_data(i,j)= seven_data(i,j)./k;
129         end
130     end
131
132     figure, imshow(seven_data);
133
134     %DATA 8
135
136     eight_data=zeros(28,28);
137     k=0;
138
```

```
139     for i=1:dim(1)
140     if labels(i)==8
141     eight_data=eight_data+im2bw((reshape(images
142     (i,:),28,28)')),0.5);
143
144     k=k+1;
145     end
146     end
147
148     for i=1:28
149     for j=1:28
150     eight_data(i,j)= eight_data(i,j)./k;
151     end
152     end
153
154     figure, imshow(eight_data);
155
156     %DATA 9
157
158     nine_data=zeros(28,28);
159     k=0;
160
161     for i=1:dim(1)
162     if labels(i)==9
163     nine_data=nine_data+im2bw((reshape(images
164     (i,:),28,28)')),0.5);
165
166     k=k+1;
167     end
168     end
169
170     for i=1:28
171     for j=1:28
172     nine_data(i,j)= nine_data(i,j)./k;
173     end
174     end
175
176     figure, imshow(nine_data);
177
178     %DATA 0
179
180     zero_data=zeros(28,28);
181     k=0;
182
183     for i=1:dim(1)
184     if labels(i)==0
185     zero_data=zero_data+im2bw((reshape(images
186     (i,:),28,28)')),0.5);
187
188     k=k+1;
189     end
190     end
191
192     for i=1:28
```

```
193     for j=1:28
194         zero_data(i,j)= zero_data(i,j)./k;
195     end
196 end
```

### 2.1.3 Implementazione confronto

Per confrontare il database con l'immagine da determinare, si moltiplica l'elemento  $i, j$  della matrice di probabilità con lo stesso elemento dell'immagine campione. Si sommano quindi tutti i valori. Si effettua questa procedura per tutte le immagini di probabilità precedentemente ottenute. Quella con il valore più alto avrà una probabilità maggiore e sarà la candidata.

A causa di alcune somiglianze tra lettere, ad esempio tra la lettera C e la O, abbiamo aggiunto un metodo per discriminarle:

se in un punto la matrice di probabilità ha valori alti e l'immagine incognita ha valore 0, si tolgono alcuni punti percentuali alla somma precedente in modo da favorire lettere con più punti in comune.

Con questa accortezza abbiamo diminuito l'errore medio di circa il 15%.

Questo metodo da noi sviluppato autonomamente, che prende in considerazione solamente due caratteristiche di discriminazione fra caratteri, ha dei valori di efficienza abbastanza buoni per alcune lettere facili da rappresentare, come alcuni valori particolarmente bassi per altri caratteri.

Possiamo quindi asserire che sebbene alcune cifre come lo 0 o l'1 abbiamo percentuali di individuazione positiva sopra l'attuale media dello stato dell'arte, in generale è un metodo non troppo affidabile nei confronti degli altri disponibili ad oggi.

## 2.2 Risultati

Sono quindi brevemente presentati i risultati del sopraesposto metodo:

Number	probability
0	93,06%
1	96,74%
2	70,64%
3	85,94%
4	75,87%
5	42,83%
6	81,21%
7	81,81%
8	74,44%
9	81,86%

Possiamo notare che per alcune cifre si ottiene un risultato più che soddisfacente, invece per altre sarebbero necessari dei controlli aggiuntivi atti a migliorarne la caratterizzazione.

Questo ci porta a concludere che tale metodo, seppur funzionante, non sarebbe facilmente riportabile nel riconoscimento delle lettere o ancor peggio delle parole per una presenza troppo vasta di singole variabili da caratterizzare. Tralasciando quindi questo metodo "analogico", siamo passati ad un programma più funzionale e in apparenza più semplice ma molto più efficace: LSTM.

### 3 MACHINE LEARNING & DEEP LEARNING

Avendo valutato che il metodo precedente non rispetta i parametri caratteristici dello *state of the art* abbiamo provato ad implementare un nuovo metodo sviluppato sul *machine learning* e più nello specifico sul *deep learning*.

Il *deep learning* per definizione non è né descrivibile come intelligenza artificiale né come *machine learning* anche se ha molte caratteristiche in comune.

Il *machine learning* è definita come una categoria di ricerca e di algoritmi il cui scopo è quello di trovare dei *pattern* comuni nei *set* di dati inseriti e di usare gli stessi per fare una successiva previsione.

In altre parole, lo scopo di un algoritmo di *machine learning* è quello di imparare a compiere un compito con una minima *performance* basata sul tipo di dati da processare. Usando una definizione in inglese possiamo dire: “A computer program is said to learn from experience *E* concerning some class of task *T* and a performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves because of experience *E*.” In questo modo, ad ogni iterazione il programma si evolverà, migliorando la propria accuratezza nello svolgimento del proprio compito basandosi sui nuovi dati inseriti.

Possiamo così introdurre alcuni principi base del *machine learning*.

Ogni algoritmo di *machine learning* è diviso in tre fasi di apprendimento:

- *supervised learning*
- *unsupervised learning*
- *reinforcement learning*

Nella prima fase, dato l'*output* desiderato, l'algoritmo trova dei metodi per far combaciare i dati con l'*output* iniziale salvato come primo elemento nel file *.csv*. In pratica forniamo un'immagine con già allegato il giusto label di assegnazione e l'algoritmo trova le caratteristiche che tutte le immagini con la stessa assegnazione hanno in comune e in opposizione con quelle diverse.

Nel processo successivo il codice deve sfruttare le regolarità nelle immagini fornite per riuscire a suddividerle nel modo corretto. Vengono quindi inseriti degli *input* senza l'*output* desiderato e basandoci sulle caratteristiche appena scoperte si provano a costruire delle classi inserendoci i nuovi *input*.

L'ultima fase è quindi di rinforzo delle caratteristiche appena descritte. Ovvero produce delle azioni forzate *ai* che cambiano l'immagine e viene assegnata una ricompensa in caso di successo *ri* e l'algoritmo procede in modo da massimizzare il suo valore finale.

L'apprendimento delle caratteristiche è un problema impegnativo per il *machine learning*, come per gli algoritmi di *computer vision*, per l'intelligenza artificiale e le neuroscienze. Il *deep learning* presuppone che sia possibile apprendere una descrizione gerarchica con astrazione crescente, trasformando e adattando i livelli allenandone le caratteristiche. Ad esempio, nel riconoscimento delle immagini possiamo passare i dati della riga del modulo, ai pixel, al orlo, alla trama, alla regione, all'oggetto e arrivare alla fine comprendendo l'intera scena. Più legati al nostro esempio, nell'analisi del testo, le lettere possono formare parole, che a loro volta compongono frasi creando un testo completo. A seconda della direzione del flusso di informazioni, possiamo avere architetture diverse per la gerarchia delle caratteristiche. La direzione delle informazioni feedforward utilizza reti multistrato o reti convoluzionali. La direzione delle informazioni di feedback utilizza invece una codifica sparsa sovrapposta o reti deconvoluzionali. Alla fine per un movimento bidirezionale di informazioni è preferibile utilizzare la macchina Boltzmann profonda o gli autoencoder.

Questo può anche portare a diversi tipi di *unsupervised learning* come: livelli puramente supervisionati, non supervisionati, o supervisionati solo inizialmente e senza supervisione finale.

Per comprendere meglio l'architettura dell'*artificial neural network* abbiamo scelto di partire dal caso più semplice per poi implementarlo con modelli più complessi.

Possiamo quindi iniziare con l'uso delle porte logiche. Ad esempio, l'*OR* può essere visto come una somma ponderata con un *threshold* che consente di differenziare almeno due casi.

Basandosi sull'apprendimento hebbiano possiamo dire che la forza di una sinapsi migliora insieme e simultaneamente all'attivazione di un input relativo del target desiderato.

Possiamo così definire il tasso di apprendimento di un algoritmo definito come  $\eta$  e viene utilizzato nelle formule:

$$W_i^k + 1 = W_i^k + \Delta W_i^k$$

$$\Delta W_i^k = \eta * X_i^k * t^k$$

Nel primo caso, il dataset è implementato linearmente, il che limita il processo nei casi più complessi. Nel caso invece dell'attuazione non lineare il *perceptron* non è più in grado di lavorare e abbiamo bisogno di soluzioni alternative.

Con un singolo operatore, il tipo di regione descritta è un *half bounded by hyperplanes*. Maggior è il numero di operatori, più la regione diventa complessa e di forme variegate. Possiamo quindi creare una regione convessa aperta o chiusa, come una regione arbitraria la cui complessità è solamente limitata dal numero di nodi che applichiamo. I nodi interni di solito sono nascosti e per questo motivo sono chiamati *hidden layer*. Il livello di input è collegato al livello di output con uno o più livelli nascosti. Maggiore è il loro numero, maggiore è la complessità di calcolo e quindi l'accuratezza della regione delimitata. Bisogna anche prestare attenzione a non overfittare l'errore usando troppi *hidden layer* o un *training set* troppo grande.

Il modello non lineare è caratterizzato da una serie di valori quali il numero di neuroni, le funzioni di attivazione e i valori del peso di ogni collegamento. I collegamenti sono caratterizzati tramite pesi  $W(i)$ . L'*output* di un nodo / neurone dipende solo dal livello precedente e dal peso del collegamento di connessione. La funzione di attivazione è definita come la legge che definisce l'*output* di quel nodo dato un *input* o un *set* di *input*. Deve essere differenziabile. Può

avere schemi diversi per usi diversi. In regressione viene utilizzata una funzione di attivazione lineare, mentre *Sigmoid* e *Tanh* vengono utilizzati per la classificazione. In regressione, l'*output* copre l'intero dominio  $\mathbb{R}$ , quindi utilizziamo una funzione di attivazione lineare per il neurone di *output*. Nella classificazione con due classi, usiamo l'attivazione dell'*output* di *Tanh* se le due classi sono -1, +1; invece usiamo l'attivazione *sigmoid* con 0, +1. Questi valori possono essere interpretati come probabilità posteriore di classe. Invece, se abbiamo a che fare con più classi nel numero  $k$  usiamo tanti neuroni quante sono le classi, questo metodo è chiamato *one-hot encoding*.

Ora che conosciamo le basi delle reti neurali possiamo adattare più nello specifico per i nostri scopi.

Le *neural network* possono essere viste come una generica approssimazione. Anche una rete neurale a *single hidden layer* con funzione di attivazione sigmoide può approssimare qualsiasi misura con qualsiasi grado di precisione desiderato su un set compatto. Indipendentemente dalla funzione che l'algoritmo sta imparando, l'utilizzo di un singolo livello non può significare che un algoritmo di apprendimento sia in grado di trovare i pesi necessari. Quindi può sussistere il caso in cui si necessiti di più di un livello intermedio. Nel peggiore dei casi potrebbe essere necessario un numero esponenziale di unità nascoste e il livello potrebbe essere eccessivamente grande non riuscendo ad apprendere e generalizzare il nostro evento. In questi casi, la classificazione richiede solo un livello aggiuntivo a quello precedente.

Di solito il processo di *training* è così composto:

Viene dato come input

$$D = \langle x_1, t_1 \rangle, \dots, \langle x_n, t_n \rangle.$$

L'algoritmo deve quindi trovare un modello che fornisca i parametri della nostra *neural network*, tali che per dati non precedentemente elaborati, otteniamo una funzione

$$y(x_n|\theta) \approx t_n.$$

Nel caso di una rete neurale quest'equazione può essere riscritta come

$$g(x_n|w) \approx t_n.$$

La funzione di errore da minimizzare diventa quindi la somma degli errori dati come sottrazione del valore atteso  $t_n$  e della funzione  $g(x_n | w)$  al quadrato.

$$E = \sum (t_n - g(x_n|w))^2$$

Quindi cerchiamo il modello matematico, in prima approssimazione lineare, che minimizzi la somma degli errori al quadrato. Ovvero il metodo dei minimi quadrati.

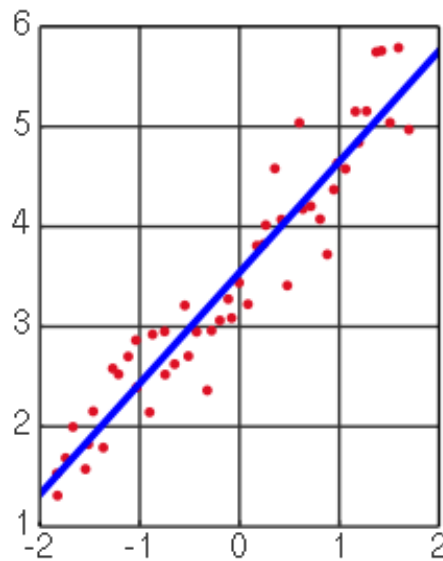


Figura 3: metodo dei minimi quadrati

Per trovare il minimo di una generica funzione di approssimazioni non lineari è sufficiente calcolare la derivata parziale della funzione e trovare i punti stazionari, ovvero quelli con valore nullo.  $\frac{dJ(w)}{dw}$ . Una soluzione in forma chiusa non è praticamente mai ottenibile, così si rende indispensabile l'uso di metodi iterativi numerici che convergano progressivamente al minimo globale.

Trovare i pesi della rete neurale è un processo di ottimizzazione non lineare che va ripetuto per diverse configurazioni iniziali. La scelta di molteplici punti di partenza evita l'ottenimento di punti di minimo locali.

Invece di scegliere un modello di funzione lineare, possiamo sceglierne uno polinomiale, che si adatti meglio ai punti empirici, avendo inoltre una funzione di errore che minimizzi meglio in intervalli di tempo minori. Il problema principale delle funzioni polinomiali con ordine crescente è che, non stiamo più studiando i procedimenti effettivi dell'evento, ma bensì prendendo in considerazione troppe incognite, stiamo *overfittando* l'errore. Significa che stiamo prendendo in considerazione nel nostro modello anche il modo in cui gli errori cambiano in funzione del tempo. Possiamo anche vederlo come un'applicazione del *Novacula Occami* o anche detto rasoio di Ockham. Praticamente stiamo assumendo che le nostre misure siano perfette e che le successive siano descritte dal modello con il maggior grado. Il che probabilmente non sarà rispettato, ovvero i nuovi punti non apparterranno alla curva con il grado massimo. Invece, usando una funzione più semplice, l'errore medio rimarrà costante anche con esempi non rilevati.

Possiamo ora inoltrarci nello studio più specifico delle reti neurali a noi più congeniali.

Esamineremo segnatamente le LSTM ovvero *long short time memory* che possono essere utilizzate per generare una sequenza complessa di dati generati basandosi anche su *input* pregressi.



Le *recurrent neural network* (RNN) sono le classi di algoritmi migliori che siano in grado di produrre *deep learning* con un modello dinamico che genera dati in tempo reale. Possono essere allenate per la generazione di sequenze elaborando dati in tempo reale cercando di prevedere, ad ogni passo, ciò che verrà dopo. Come detto prima, una RNN non estrae un modello specifico dai dati di *training*, ma utilizza piuttosto la sua rappresentazione interna per eseguire una *high-dimensional interpolation* tra esempi di *training*. Per questo motivo le RNN possono essere definite come "fuzzy". Ovvero nel caso in cui si dovesse calcolare due volte la stessa previsione si avrà bassa probabilità di ottenere lo stesso risultato. La previsione non è influenzata dalla *curse dimensionality*, che è definita come il numero crescente di esempi richiesti, conseguentemente ad un crescente numero di caratteristiche delle immagini necessarie per addestrare l'algoritmo.

Una RNN abbastanza grande dovrebbe quindi essere in grado di memorizzare e creare una sequenza di output ampi a piacimento.

Consideriamo il caso reale in cui la RNN non è in grado di memorizzare per un tempo sufficiente l'output. Questo provoca che solo alcuni *output* creano dipendenza sull'attuale uscita. Ciò porta a problemi di amnesia e instabilità. Se l'*output* si basa solo su poche uscite pregresse anchesse previste su pochi dati, abbiamo poche opportunità di riscontrare quindi correggere gli errori iniziali. Infatti, qualora si verificasse un errore da qualche parte nel calcolo, lo manterremo a lungo in quanto tutti gli *output* sono riferiti a quelli precedenti, ma già con pochi passaggi non siamo più in grado di tenerne traccia.

Avere una memoria lunga ha un effetto stabilizzante poichè gli errori sono distribuiti in intervalli di tempo più lunghi e possono essere corretti più facilmente. Per evitare il problema dell'instabilità una soluzione palliativa potrebbe essere quella di inserire degli *input* distorti con del rumore, solo con modelli condizionali, all'interno del processo di previsione prima di reimmetterli nel modello. Questo permette allo stesso di diventare più robusto con *input* imprevisti. Ma la soluzione migliore rimane comunque avere una memoria lunga. Per questo motivo, è meglio usare LSTM.

LSTM sono una RNN la cui architettura interna è stata riprogettata per archiviare meglio e dare accesso alle informazioni in periodi più lunghi. Possiamo inoltre dimostrare che LSTM può generare una sequenza con struttura a lungo raggio temporale. Dopo aver preso in considerazione l'architettura, il più generale possibile di una RNN, possiamo iniziare con la parte di *training* e *validation* convalida. La formazione è necessaria per scoprire il peso di ogni connessione tra ogni nodo. La convalida invece è un processo atto a dimostrare che il peso appena trovato è quello corretto o quantomeno quello che più si avvicina con una certa accuratezza ai risultati desiderati. L'ultima parte è il test in cui viene rilevata la precisione effettiva del modello. La convalida della parte può anche essere trascurata perché non è fondamentale per il successo del sistema. Sia il *training set* che il *validation set* hanno entrambi il label della classe corrispondente alla caratterizzazione effettiva contenuta nella parte appena successiva di codice. Invece, il set di test non ha l'etichetta di classe, ma solamente i valori ordinati di rappresentazione dell'oggetto. Durante la parte di training, tutte le righe, ovvero le immagini, vengono inserite nell'algoritmo e il peso delle connessioni viene aggiornato ad ogni passaggio. Alla fine del suddetto processo dovremmo aver raggiunto un algoritmo finale abbastanza buono, che può finalmente compiere una buona previsione con nuovi dati simili a quelli

precedenti.

Il *validation set* è simile, seppur completamente separato e diverso dal *training set*. Ovvero da un unico primitivo *dataset* contenente solo valori diversi gli uni dagli altri e composto da *label* e valori, sono ricavati i due *subset* di *validation* e *training*. L'obiettivo principale del processo di *validation* è quello di verificare che il modello appena trovato nel *training* sia ottimale. Inoltre ci aiuta fornendoci informazioni che possano aiutarci a regolare gli *hyperplanes*. Rielaborando quanto appena detto, in ogni fase dell'allenamento il modello viene addestrato sui dati nel *training set* e convalidato simultaneamente anche sui dati del *validation set*. Durante il processo di addestramento, il modello classificherà l'output per ciascun diverso input del *training set*. Dopo questa classificazione, l'errore verrà calcolato e il peso nel modello regolato. Quindi durante la fase successiva, verrà classificato nuovamente lo stesso input in modo da poter aver un confronto più veritiero. Inoltre, durante l'addestramento, il modello classificherà anche ogni input dal set di validazione. La classificazione si baserà solo su quanto appreso dai dati con cui è stato allenato nel set di addestramento e i pesi non verranno più aggiornati nel modello in base all'errore calcolato sui dati di validazione. Bisogna anche ricordare che i dati e il set di validazione sono separati da quelli del set di formazione, quindi durante la convalida del modello, i dati studiati non sono già stati implementati nel modello. Uno dei motivi principali per cui è necessario un set di convalida è garantire che il nostro modello non si adatti troppo ai dati nel set di *training*, ovvero per verificare che non si *overfitti* l'errore. Discuteremo successivamente in modo approfondito gli errori di *overfitting* e *underfitting* del modello. Possiamo comunque definire l'*overfitting* come il problema che il modello si adatti troppo ai dati forniti nel *training set* non riuscendo a prevedere con la stessa accuratezza anche i dati presenti nel *validation set*. Si può evincere la non presenza di *overfitting* se nel momento di validazione, l'accuratezza delle soluzioni è simile a quella ottenuta nel processo di *training*. Al contrario, se i risultati sui dati di allenamento sono particolarmente buoni, ma il risultato dei dati di convalida è scarso, allora possiamo assumere con elevata probabilità che ci sia *overfitting*. Spostandoci ora alla fase di test dobbiamo innanzitutto far notare che il *data set* non contiene i *label*. Questo processo è successivo alle fasi di *training* e *validation*. Il metodo di sviluppo dell'algoritmo è lo stesso delle fasi precedenti, mantenendo la stessa architettura interna. L'assenza della classificazione pregressa porta ad un'incertezza delle soluzioni. Lo scopo principale dell'intero codice è quello di riuscire nel processo di classificazione anche senza sapere anticipatamente i *label* dei casi studiati.

Per inoltrarci nella comprensione delle LSTM, dobbiamo immaginare graficamente i passaggi compiuti dagli *input* per l'ottenimento degli *output*. Ovvero bisogna tener conto della correlazione interna di tutti i nodi di un livello e di quella più 'superficiale' tra *output* dei livelli precedenti e *input* dell'attuale processo.

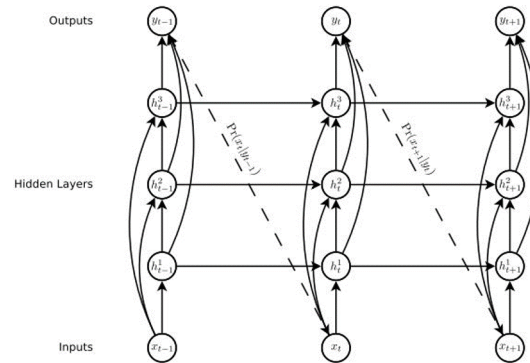


Figura 4: architettura generale struttura LSTM

Per meglio comprendere il funzionamento di tale architettura è utile procedere anche ad una trattazione scritta esemplificativa. Ad ogni istante temporale  $t$  vengono inseriti gli *input*  $x = (x_1, \dots, x_T)$  gli *input* vengono fatti passare attraverso tutti gli *hidden layer*  $hn = (hn_1, \dots, hn_T)$  con i diversi pesi rispettivamente associati. Alla fine degli *hidden layer* si giunge quindi alla formulazione di una previsione dell'*output*  $y = (y_1, \dots, y_T)$ . Ogni *output* viene inoltre usato per parametrizzare una distribuzione predittiva  $Pr(x_{t+1}|y_t)$  per tutti i possibili successivi *input*  $x_{t+1}$ . Il primo elemento  $x_1$  di ogni sequenza di *input* è sempre un vettore nullo le cui entrate quindi corrispondono tutte a zero. L'algoritmo produce quindi una predizione per la seconda serie di *input*  $x_2$ , ovvero il primo *input* realmente inserito nel programma senza però essere influenzato da informazioni pregresse.

Tutti i livelli intermedi tra il livello di inserimento degli *input* e il livello degli *output* sono definiti *hidden layer* in quanto non sono visibili, ne quanto meno modificabili esternamente. Per facilitare la fase di addestramento per le *deep networks*, gli input sono collegati a tutti i *layer* nascosti e quindi tutti i *layer* sono collegati agli *output*. In questo modo, riduciamo il numero di passaggi dal basso verso l'alto.

Possiamo descrivere quindi il livello nascosto con una funzione  $H$ , che è un'applicazione  $n$  in senso elementare di una funzione sigmoide. I livelli nascosti vengono calcolati integrando tale funzione che fornisce i pesi degli input. La funzione sigmoide è una semplice funzione matematica che ha una caratteristica curva la cui forma è ad S. Un esempio di una possibile funzione sigmoide è quella logistica. In generale, tutte le curve sigmoidi hanno un comportamento monotono e la loro derivata prima rispecchia un comportamento gaussiano o quantomeno a campana. Tuttavia, come abbiamo accennato prima, abbiamo bisogno di un algoritmo con una memoria più estesa di una singola iterazione, e LSTM ha un'architettura tale da essere ideale per il nostro caso. È utile trovare e sfruttare le dipendenze a lungo raggio nell'insieme di dati che dovremmo elaborare. L'algoritmo originario utilizza un metodo specifico per aggiornare il peso della singola connessione tra nodi che si basa su un calcolo approssimativo del gradiente. Ma un modello che invece utilizza il calcolo del gradiente esatto, sommato al processo di *backpropagation* su tutto il testo da studiare,

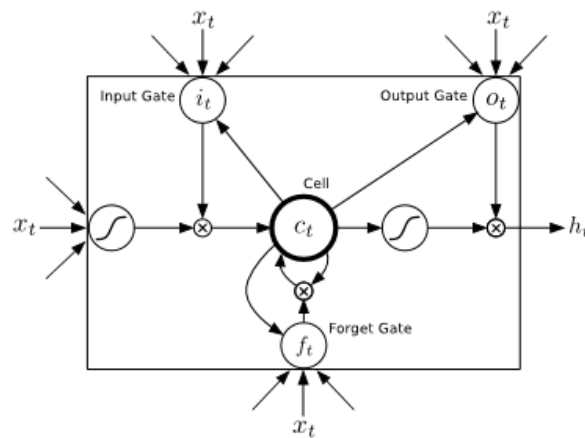


Figura 5: architettura basiale della cella dell'architettura LSTM

è più efficace sebbene di più difficile realizzazione. Spesso le derivate parziali utilizzate per calcolare il gradiente esatto potrebbero divergere anche in brevi intervalli di tempo. Quindi dovremmo considerare l'utilizzo di una limitazione del modulo per appunto evitare divergenze. Passando da tutta questa teoria al nostro progetto, l'LSTM viene utilizzato per la previsione del testo. I dati di testo vengono utilizzati come input discreto e inseriti in un vettore di input *one hot*. Per ricordare, il vettore *one-hot* in una rete neurale ricorrente con  $n$  classi, avrà una lunghezza di  $n$  elementi. Le voci del vettore sono tutte nulle per le classi che non rappresentano l'elemento, e la posizione della classe assume il valore 1. Le LSTM per il *text prediction* sono principalmente usate con due diversi modelli di classi. Queste sono:

- parole
- caratteri

Ci sono molti pro e contro per entrambe le classi. Ad esempio, la classe con le parole è sicuramente più efficiente nel rilevamento delle parole, perché può funzionare non solo con l'algoritmo basato sulle caratteristiche del tratto scritto, ma anche attraverso un algoritmo sottrattivo su un 'dizionario'. Al contrario, la classe di caratteri può predire il simbolo basandosi solamente su ciò che può vedere senza aiuti supplementari. Inoltre, con il modello a lettere si può incorrere nella creazione di parole non esistenti. Questo è appunto dovuto alla mancanza di un controllo a livello più alto sulle parole. Nel primo caso, il numero di elementi della classe è uguale al numero di parole del nostro dizionario e questo può portare a problemi in attività in *real time* in quanto dovrebbe processare un numero di classi che supera facilmente le 100.000 unità. Quindi, la classe con parole viene utilizzata se vogliamo prevedere la parola che stiamo scrivendo in *real time*, usando quindi un semplice algoritmo sottrattivo sul dizionario. Non è necessario un algoritmo di apprendimento automatico anche se facilmente implementabile. Se vogliamo produrre invece una previsione delle parole in tempo reale con le regole grammaticali o con l'uso di *machine learning*, potrebbe essere

più veloce, ma anche più impreciso l'uso della classe con i caratteri. L'algoritmo utilizzando la classe 'parole', basato un *database* completo del dizionario, sottrae contestualmente alla scrittura delle prime lettere tutte le parole dal dizionario che non hanno quell'ordine della lettera. Inoltre, il *database* dovrebbe essere "ponderato". Ovvero, una parola più comune dovrebbe avere un peso maggiore di una con frequenza d'uso minore (ciao è sicuramente più comune di ciangottare e anche se iniziano con le stesse tre lettere, l'algoritmo dovrebbe preferire quella più comune, in questo caso, ciao). A parte questo caso, che è molto limitante e tornando all'algoritmo di apprendimento automatico, la classe più comune è appunto quella con le lettere. Infatti maggiore è il numero di classi, maggiore sforzo dovrà essere fatto in ordine di quantità di dati di training per coprire adeguatamente tutti i possibili contesti per la parola. Il modello di *machine learning* sulle parole potrebbe essere utilizzato per la previsione delle parole all'interno di un contesto (vogliamo prevedere la parola successiva dati gli input precedenti). Un ulteriore problema per la classe delle parole è che non è applicabile ad un testo che contiene anche elementi non classificabili come parole. Un esempio potrebbero essere caratteri speciali (!?@ ) o le parole di altre lingue. L'algoritmo con la classe delle parole è stato considerato solo di recente nelle reti neurali e ha ancora prestazioni peggiori rispetto ai modelli basati sui singoli caratteri anche se sarebbe possibile inventare una parola e nuove stringhe.

Sono stati svolti numerosi esperimenti per misurare la potenza predittiva e la precisione dei diversi metodi di reti neurali. L'esperimento più significativo e degno di nota è il *Penn Treebank* che ha confrontato le prestazioni sia per le classi con parole che quelle con singoli caratteri con lo stesso LSTM composto da 1000 unità per il singolo *hidden layer*. Hanno usato lo stesso numero di elementi nel set di formazione, ma questo confronto è in qualche modo impari in quanto il numero di elementi per ogni classe è molto diverso. Inoltre, dato un piccolo set di dati di training, l'algoritmo supera facilmente i campioni (non ci sono abbastanza differenze da tutti gli elementi della stessa classe). E' qui di seguito mostrata la tabella di riepilogo per rappresentare i risultati e la tendenza dell'errore:

INPUT	REGULARISATION	DYNAMIC	BPC	PERPLEXITY	ERROR (%)	EPOCHS
CHAR	NONE	NO	1.32	167	28.5	9
CHAR	NONE	YES	1.29	148	28.0	9
CHAR	WEIGHT NOISE	NO	1.27	140	27.4	25
CHAR	WEIGHT NOISE	YES	1.24	124	26.9	25
CHAR	ADAPT. WT. NOISE	NO	1.26	133	27.4	26
CHAR	ADAPT. WT. NOISE	YES	1.24	122	26.9	26
WORD	NONE	NO	1.27	138	77.8	11
WORD	NONE	YES	1.25	126	76.9	11
WORD	WEIGHT NOISE	NO	1.25	126	76.9	14
WORD	WEIGHT NOISE	YES	1.23	117	76.2	14

Figura 6: risultati dell'esperimento Penn-Treebank

Tutti i dati sono stati valutati nel modo più equo possibile. Quindi, tutte le reti sono state addestrate con un gradiente stocastico discendente, utilizzando un tasso di apprendimento di 0,0001 e un momento di 0,99, infine, i derivati sono stati tagliati nell'intervallo  $[-1, n. 1]$ .

Di solito, le reti neurali usano il peso fisso, ma per i problemi predittivi, è possibile anche modificare i valori del peso. Mikolov ha chiamato questo metodo di valutazione dinamica. Il che ha portato ad un confronto più equo tra diversi

algoritmi di compressione. Inoltre non è stata applicata una separazione tra il *train set* ed il *test set* in quanto vengono stimati una sola volta. Dal momento che entrambe le reti overfittano hanno provato l'esperimento applicando al peso dei collegamenti sia del rumore fisso e diverso per ognuno *weight noise*, come del rumore ponderato adattivo *adaptive weighted noise*. Il primo applicava una deviazione standard fissa ai pesi iniziali. Nel secondo, invece, la varianza del rumore viene salvata insieme ai pesi utilizzandola in seguito all'interno di una funzione di perdita atta a minimizzare la sua lunghezza. La *Perplexity* è la misura delle prestazioni usuali per la modellazione linguistica. La tabella superiore mostra che una RNN con classe di parole ha prestazioni migliori rispetto alla rete con classe di caratteri, ma il divario tra le prestazioni a livello di parola e di carattere sembrava chiudersi quando viene utilizzata una qualsivoglia regolarizzazione. Alla fine, i risultati si confrontano favorevolmente con quelli raccolti nella tesi di Tomas Mikolov. È la prova finale che, anche se il livello delle parole è più difficile e più lento, ha prestazioni migliori e dovrebbe essere preferito per la previsione delle parole (non necessario per il riconoscimento delle parole come nel nostro caso).

### 3.0.1 HANDWRITING PREDICTION

In primo luogo, definiamo ciò che intendiamo con il termine *handwriting prediction*: "Dato un insieme di tratti scritti a mano, inseriti nel programma e registrati come una sequenza di posizioni della punta della penna (percorso dato da un paio di valori x,y), vogliamo che l'algoritmo preveda la lettera, la parola si cui essa fa parte e possibilmente anche la parola successiva, il tutto in *real time*". Si tratta quindi della generazione di una sequenza di *output* digitali partendo da una sequenza di *input* creati da un essere umano. Così, come abbiamo detto prima, possiamo usare un algoritmo LSTM che si adatta perfettamente al nostro caso. Per esempio è possibile usare la libreria IAM-OnDB. Questa è composta da tutti i caratteri estratti da 221 diversi scritti a mano dello stesso testo. I tratti della penna sono stati tracciati con un sistema ad infrarossi che salva la posizione della penna ad ogni istanza temporale con le due coordinate cartesiane. Due segni consecutivi sono divisi l'uno dall'altro quando la penna viene sollevata dalla tastiera. Per evitare errori inoltre, coloro che hanno formato il *database*, hanno sia tagliato i passi troppo lunghi, superiori ad un certo *threshold*, sia interpolato i tratti troppo brevi (dato da una lettura mancante dell'intera parola). Questo metodo, essendo di difficile applicazione, necessita di molta memoria e di un grande lasso temporale per calcolare tutti i passaggi. Soprattutto per le lettere composte da più di un singolo tratto, come t o i è richiesto un grande sforzo all'algoritmo. Il *set* di dati viene suddiviso automaticamente in un *set* di *training*, due *validation set* e un *set* di *test*; tutti con dimensioni diverse. Per la mancanza di un risultato storico di riferimento, possiamo unire i due *validation set* e il *training set* in un unico grande *database* in modo da ottenere un processo di formazione più forte, anche se affetto da *overfitting*. Il *set* di convalida più piccolo viene utilizzato nel caso in cui si renda necessario arrestare anticipatamente il processo. L'obiettivo che allo stesso tempo raffigura anche il problema principale è determinare una distribuzione predittiva adatta ad input reali.

Fino ad ora la nostra RNN era riconducibile abbastanza facilmente ad un modello base che dato un *input* produceva una predizione in *output* basata sul

processo di *training* pregresso. Ora possiamo implementare il nostro algoritmo in modo da voler ottenere non un valore certo in *output*, bensì una distribuzione di probabilità che ci indichi sia il valore più probabile di assegnazione dell'*output*, sia un indice di confidenza della nostra predizione. Solo inizialmente possiamo implementare la distribuzione come simil gaussiana pura, ma è facilmente iterabile il processo in modo da creare una così detta *mixture density output*. Ovvero la distribuzione probabilistica degli *output* finali è formata dalla somma pesata di più curve probabilistiche a campana, assumendo quindi una forma varia a piacimento. Il processo matematico formale per le *recurrent neural network* si riduce al mero calcolo di una probabilità composta e di qualche altra operazione computazionale.

Questo modello è attualmente molto usato.

Formalmente un sottoinsieme delle uscite viene utilizzato per definire i nuovi *mixture weights*, mentre le uscite rimanenti vengono utilizzate per parametrizzare i singoli componenti della *mixture*. Le uscite del *mixture weight* sono normalizzate con una funzione *softmax* per garantire che formino una distribuzione discreta valida e le altre uscite vengono passate attraverso funzioni adatte per mantenere i loro valori all'interno di una gamma significativa. Le reti che utilizzano tale metodo vengono addestrate per massimizzare la densità di probabilità logaritmica degli obiettivi nell'ambito delle distribuzioni indotte. Più semplicemente possiamo dire che otteniamo come output delle triplette, una per ogni valore della media, che ci forniscono il valore centrale della distribuzione, la deviazione standard e il peso percentuale di tale valore rispetto alle altre distribuzioni.

Il processo di *mixture density output* può essere utilizzato anche con reti neurali ricorrenti, come accennato precedentemente. In questo caso la distribuzione dell'*output* è condizionata non solo sull'input corrente, ma anche sulla cronologia degli input precedenti. In questa breve spiegazione ci siamo avvalsi solo di esempi di distribuzioni gaussiane, ma ovviamente sono utilizzabili anche distribuzioni di Bernulli o altre varie ed eventuali.

### 3.1 CODICE

Questo metodo cambia completamente punto di vista rispetto al precedente in quanto non siamo più noi a fornire un metodo di discriminazione tra i diversi caratteri, ma è lo stesso algoritmo che si evolve ed impara a distinguere le diverse lettere. Infatti non è più basato su una mera idea matematico-probabilistica, ma su ogni qualsivoglia caratteristica scoperta autonomamente dal programma.

Questo ci permette di avere indici di accuratezza decisamente più elevati e paragonabili allo *state of the art* o semplicemente la riduzione dei casi di falsi negativi e falsi positivi. Il codice è stato implementato in *python* e in modo del tutto simile a prima sono stati usati dei *train* e *test-set*. Per poter analizzare e dimostrare che il seguente metodo è valido anche per casi più complicati sono stati usati *dataset* diversi sviluppati sulle lettere e non sulle cifre. Tutte le immagini sono già riscalate e normalizzate su una grandezza standard, quindi abbiamo opportunamente ridotto le linee di codice per non appesantire l'algoritmo. L'intero *dataset* è stato quindi diviso in 2 gruppi, con percentuali diverse rispetto a prima per non sovrasaturare la fase di *training*. Il 20% dei dati è stato inserito nel *testing set* e l'80% nel *training*.

### 3.2 codice LSTM

```

1 import numpy as np
2 import sklearn
3 %matplotlib inline
4 import matplotlib.pyplot as plt
5 import pandas as pd
6 import math
7 from sklearn import metrics
8 from google.colab import files
9 uploaded = files.upload()
10
11 import io
12 X_train = pd.read_csv(io.BytesIO(uploaded['train-data.csv']),
13 header = None)
14 Y_train = pd.read_csv(io.BytesIO(uploaded['train-targets.csv']
15 ]), header = None)
16 X_test = pd.read_csv(io.BytesIO(uploaded['test-data.csv']),
17 header = None)
18 Y_test = pd.read_csv(io.BytesIO(uploaded['test-targets.csv'])
19 , header = None)
20
21 WE CHOOSE A DATASE: IMPORTING OPTICAL CHARACTER RECOGNITION
22 (ORC) DATASET:
23 x_train=np.array(X_train)
24 y_train1=np.array(Y_train)
25 x_test=np.array(X_test)
26 y_test1=np.array(Y_test)
27
28 y_train=np.ravel(y_train1,order='C')
29 y_test=np.ravel(y_test1,order='C')
30
31 OTHERWISE WE CAN USE ALTERNATIVE METHOD FO IMPORTING
32 # from numpy import genfromtxt
33 # x_train = genfromtxt("C:\\Users\\deiro_jgs0sqk\\Desktop
34 \\sklearn-lab-material\\sklearn-lab\\ocr\\train-data.
35 csv", delimiter=',')
36
37 # y_train = genfromtxt("C:\\Users\\deiro_jgs0sqk\\Desktop
38 \\sklearn-lab-material\\sklearn-lab\\ocr\\train-targets.
39 csv", delimiter=',')
40
41 # x_test = genfromtxt("C:\\Users\\deiro_jgs0sqk\\Desktop
42 \\sklearn-lab-material\\sklearn-lab\\ocr\\test-data.
43 csv", header = None)
44
45 # y_test = genfromtxt("C:\\Users\\deiro_jgs0sqk\\Desktop
46 \\sklearn-lab-material\\sklearn-lab\\ocr\\test-targets.
47 csv", header = None)
48
49 x_train.shape, x_test.shape , y_train.shape, y_test.shape
50
51 EXPERIMENTING WITH SVM CLASSIFIER
52

```



```
53 from sklearn.svm import SVC
54
55 clf = SVC(C=10, kernel='rbf', gamma=0.02)
56
57 # Training
58 clf.fit(x_train, y_train)
59
60 # Prediction
61 y_pred = clf.predict(x_test)
62
63 y_pred
64
65 #In order to see the first predicted letter(it should be a g)
66 #x = x_test[0].reshape((16, 8))
67
68 #plt.gray()          # use a grayscale
69 #plt.matshow(x)      # display a matrix of values
70 #plt.show()          # show the figure
71
72 metrics.accuracy_score(y_test, y_pred)
73
74 THE ACCURACY OBTAINED IS THE SAME WE CAN LOOK IN THE
75 TRAIN SET WE USED. IT IS 0.09673 (PRESENT IN THE READ ME)
76
77 TESTING THE SVM CLASSIFIER USING CROSS-VARIATION
78
79 try:
80     from sklearn.model_selection import KFold, cross_val_score
81     legacy = False
82 except ImportError:
83     from sklearn.cross_validation import KFold, cross_val_score
84     legacy = True
85
86 if legacy:
87     kf = KFold(len(y_train), n_folds=3, shuffle=True,
88               random_state=42)
89
90 else:
91     kf = KFold(n_splits=3, shuffle=True, random_state=42)
92
93 gamma_values = [0.1, 0.05, 0.02, 0.01]
94 accuracy_scores = []
95 precision_weighted_scores = []
96 recall_weighted_scores = []
97 f1_weighted_scores = []
98
99 # Do model selection over all the possible values of gamma
100 for gamma in gamma_values:
101
102     # Train a classifier with current gamma
103     clf = SVC(C=10, kernel='rbf', gamma=gamma)
104
105     # Compute cross-validated accuracy scores
106     if legacy:
```

```
107     scores = cross_val_score(clf, x_train, y_train,
108                               cv=kf, scoring='accuracy')
109
110     scores2 = cross_val_score(clf, x_train, y_train,
111                               cv=kf, scoring='precision_weighted')
112
113     scores3 = cross_val_score(clf, x_train, y_train,
114                               cv=kf, scoring='recall_weighted')
115
116     scores4 = cross_val_score(clf, x_train, y_train,
117                               cv=kf, scoring='f1_weighted')
118
119 else:
120     scores = cross_val_score(clf, x_train, y_train,
121                               cv=kf.split(x_train), scoring='accuracy')
122
123     scores2 = cross_val_score(clf, x_train, y_train,
124                               cv=kf.split(x_train), scoring='precision_weighted')
125
126     scores3 = cross_val_score(clf, x_train, y_train,
127                               cv=kf.split(x_train), scoring='recall_weighted')
128
129     scores4 = cross_val_score(clf, x_train, y_train,
130                               cv=kf.split(x_train), scoring='f1_weighted')
131
132 accuracy_score = scores.mean()
133 precision_weighted_score = scores2.mean()
134 recall_weighted_score = scores3.mean()
135 f1_weighted_score = scores4.mean()
136
137 accuracy_scores.append(accuracy_score)
138 precision_weighted_scores.append(precision_weighted_score)
139 recall_weighted_scores.append(recall_weighted_score)
140 f1_weighted_scores.append(f1_weighted_score)
141
142 # Get the gamma with highest mean accuracy
143 best_index = np.array(accuracy_scores).argmax()
144 best_gamma = gamma_values[best_index]
145
146
147 accuracy_score
148
149 precision_weighted_score
150
151 recall_weighted_score
152
153 f1_weighted_score
154
155 TRAINING YOUR CLASSIFIER OVER THE FULL TRAINING SET
156
157 # FULL TRAINING SET TRAINING (using best gamma)
158 clf = SVC(C=20, kernel='rbf', gamma=best_gamma);
159 clf.fit(x_train, y_train);
160
```

```
161 USE THE CLASSIFIER TO PREDICT THE EXAMPLES IN THE TEST SET
162
163 # PREDICT EXAMPLES IN THE TEST SET (with best gamma)
164 y_pred = clf.predict(x_test)
165
166
167 accuracy = metrics.accuracy_score(y_test, y_pred)
168 accuracy
169
170 from sklearn.metrics import precision_score
171 from sklearn.metrics import recall_score
172 from sklearn.metrics import f1_score
173
174 precision_score(y_test, y_pred, average='weighted')
175
176 recall_score(y_test, y_pred, average = 'weighted')
177
178 f1_score(y_test, y_pred, average='weighted')
179
180 PLACE THE LABELS IN A FILE, IN THE SAME ORDER AS YOU
181 READ THE TEST EXAMPLES AND IN THE SAME FORMAT OF THE
182 LABELS IN THE TRAINING SET
183
184 # writing predictions in file test-pred.csv
185 import csv
186 predictions = y_pred;
187 with open('test-pred.csv', 'w', newline='') as file:
188     writer = csv.writer(file)
189     writer.writerows(predictions)
190
191 # writing predictions in file test-pred.txt
192
193 predictions = np.array(y_pred)
194 predictions = predictions.T
195
196 with open("test-pred.txt", 'w+') as datafile_id:
197     #here you open the ascii file
198
199     np.savetxt(datafile_id, predictions, fmt=['%s'])
200
201 #Y_new = pd.read_csv("test-pred.csv", header = None)
202 #y_new=np.array(Y_new)
203 #l=y_new.shape[0]
204 #y_new=y_new.reshape(1,)
205 PLOTTING THE LEARNING TRAINING CURVE
206 try:
207     from sklearn.model_selection import learning_curve
208 except ImportError:
209     from sklearn.learning_curve import learning_curve
210
211
212 plt.figure()
213 plt.title("Learning curve")
214 plt.xlabel("Training examples")
```

```
215 plt.ylabel("Score")
216 plt.grid()
217
218 clf = SVC(C=10, kernel='rbf', gamma=best_gamma)
219
220 # Compute the scores of the learning curve
221 # by default the (relative) dataset sizes are:
222 10%, 32.5%, 55%, 77.5%, 100%
223 # The function automatically executes a Kfold
224 cross validation for each dataset size
225
226 train_sizes, train_scores, val_scores = learning_curve
227 (clf, x_train, y_train, scoring='accuracy', cv=3)
228
229 # Get the mean and std of train and validation scores
230 over the cv folds along the varying dataset sizes
231
232 train_scores_mean = np.mean(train_scores, axis=1)
233 train_scores_std = np.std(train_scores, axis=1)
234 val_scores_mean = np.mean(val_scores, axis=1)
235 val_scores_std = np.std(val_scores, axis=1)
236
237 # Plot the mean for the training scores
238 plt.plot(train_sizes, train_scores_mean, 'o-',
239 color="r", label="Training score")
240
241 # Plot the std for the training scores
242 plt.fill_between(train_sizes, train_scores_mean -
243 train_scores_std, train_scores_mean + train_scores_std,
244 alpha=0.1, color="r")
245
246 # Plot the mean for the validation scores
247
248 plt.plot(train_sizes, val_scores_mean, 'o-',
249 color="g", label="Cross-validation score")
250
251 # Plot the std for the validation scores
252
253 plt.fill_between(train_sizes, val_scores_mean -
254 val_scores_std, val_scores_mean + val_scores_std,
255 alpha=0.1, color="g")
256
257 plt.ylim(0.05,1.3) # set bottom and top limits for y axis
258 plt.legend()
259 plt.show()
260
261 best_gamma
```

### 3.3 Risultati

Riportiamo qui solamente i risultati più importanti che siamo riusciti ad ottenere dal nostro codice, per una visione più dettagliata tutti i singoli passaggi con i rispettivi risultati sono riportati nel codice.

L'accuratezza ottenuta con una semplice *SVM classifier* è già molto buona e migliore della baseline.

```
[ ] metrics.accuracy_score(y_test, y_pred)
```


 0.9044195187422107

The accuracy obtained is already higher than the baseline

Figura 7: precisione algoritmo con SVM classifier

I risultati sono stati anche testati e sono state calcolate le rispettive precisioni.


```
[ ] accuracy_score
```

 0.8782867141247813

```
[ ] precision_weighted_score
```

 0.880603158563776

```
[ ] recall_weighted_score
```

 0.8782867141247813

```
[ ] f1_weighted_score
```


 0.8776603222357244

Figura 8: accuratezza e precisione testando la SVM classifier usando una cross validation

Abbiamo anche allenato il nostro classificatore su tutti i valori del set di training e l'abbiamo utilizzato per predire i valori presenti nel set di test ottenendo una precisione molto elevata, soprattutto a confronto con i risultati del metodo precedentemente esposto.

```
[ ] accuracy = metrics.accuracy_score(y_test, y_pred)
accuracy
```

0.9061451442814688

```
[ ] from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
```

```
[ ] precision_score(y_test, y_pred, average='weighted')
```

0.9072529457776723

```
[ ] recall_score(y_test,y_pred,average = 'weighted')
```

0.9061451442814688

```
[ ] f1_score(y_test,y_pred,average='weighted')
```

0.9056122164381931

Figura 9: precisione con il classificatore allenato con tutti i valori di training e usato pr predire degli esempi del set di test

Abbiamo anche disegnato la curva di apprendimento del nostro algoritmo che sembra essere ottimale per il nostro scopo, riuscendo ad ottenere un buon risultato per l'intero problema.

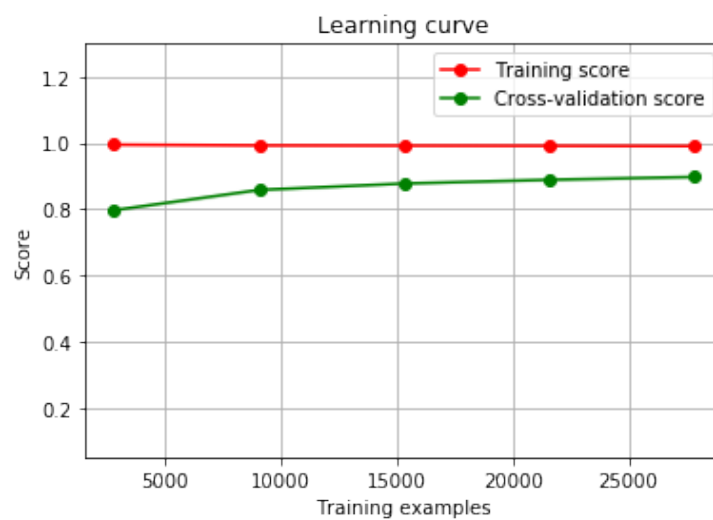


Figura 10:

## 4 Confronto

Quando ci siamo affacciati la prima volta al problema del riconoscimento della scrittura manuale, entrambi abbiamo subito riconosciuto la possibilità dello sviluppo di due algoritmi che potessero adempiere a questo scopo. Uno molto naif e l'altro più complicato ma anche più prestante.

Il primo che abbiamo sviluppato crea delle matrici di probabilità per ogni carattere studiato e le confronta con l'incognita del nostro problema. Inizialmente il confronto è basato solo sul livello di aderenza tra l'originale e l'input. Ovviamente è un algoritmo poco efficace e con molte incertezze. E' stato quindi introdotto un altro controllo per rafforzare il sistema basandoci sulla certezza delle zone vuote e piene. Il che ci ha permesso di migliorare decisamente le prestazioni del nostro algoritmo. Per semplicità di sviluppo abbiamo utilizzato un dataset MNIST sulle cifre, in modo da snellire la computazione del programma. Siamo arrivati ad ottenere delle percentuali di rivelazione con dei picchi molto alti, fino al 96,74%. Sono presenti però dei caratteri con percentuali di riuscita molto bassi (42,83% per la cifra 6). Questo ci fa capire che questo metodo all'attuale stato di sviluppo non è da ritenersi affidabile e andrebbe implementato con altri controlli più specifici atti a migliorare l'affidabilità di tale software.

Number	probability
0	93,06%
1	96,74%
2	70,64%
3	85,94%
4	75,87%
5	42,83%
6	81,21%
7	81,81%
8	74,44%
9	81,86%

Il secondo metodo, basato sul LSTM, cercando di scoprire in autonomia le caratteristiche peculiari di distinzione tra i diversi caratteri, è molto più affidabile e con molti meno casi di falsi positivi. Come ogni metodo di RNN non si possono però conoscere quali siano le caratteristiche distintive tra le diverse variabili, in questo caso le lettere.

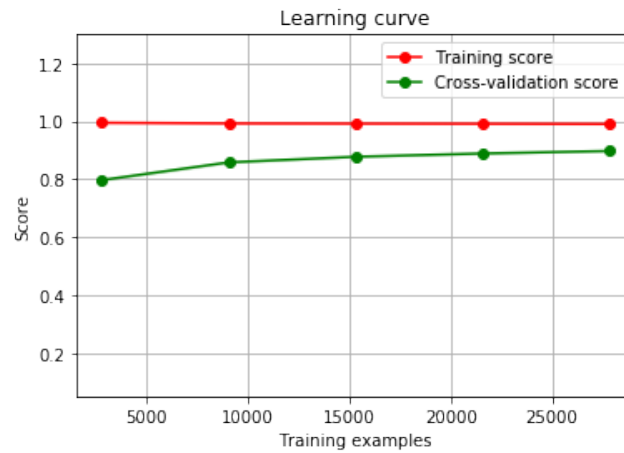


Figura 11:

I risultati non sono paragonabili per prestazioni, in quanto uno riguarda le cifre e l'altro le lettere. Comunque possiamo asserire che se il secondo ha delle prestazioni molto più elevate e decisamente accettabili rispetto allo stato dell'arte. Il metodo probabilistico essendo limitato dal punto di vista computazionale, se fosse apportata qualche miglioria tecnica potrebbe essere usato, come nel nostro caso, per il riconoscimento delle cifre. L'implementazione di questo stesso programma per le lettere maiuscole sarebbe molto difficile e probabilmente porterebbe a scarsi risultati. Sarebbe altrettanto difficile, se non impossibile, modificare tale algoritmo per il riconoscimento della scrittura in corsivo, soprattutto per il problema della separazione delle lettere e per il fatto che alcune assunzioni non sarebbero più valide (come il rettangolo che contenga tutta una lettera, visto che nella scrittura corsiva si tende a scrivere in obliquo).

Possiamo quindi asserire che il metodo basato sull'apprendimento automatico sia preferibile in tutto e per tutto a quello "analogico" basato sulle probabilità non solo per le prestazioni che riesce a fornirci, ma anche per semplicità di utilizzo e sviuppo.



## 5 STATE OF THE ART

Il primo paper che citiamo è:

*The state of the art in online handwriting recognition.*

Questo articolo è molto apprezzato dalla comunità scientifica internazionale come punto di partenza per lo studio dello stato dell'arte in materia, citato infatti ben 1212 volte. Bensì sia datato, è sempre attuale per comprendere le basi di questo argomento.

Questo estratto infatti è una raccolta completa di tutto ciò che riguarda *online handwriting recognition*. E' un'estesa raccolta di letteratura pregressa, articoli scientifici, discorsi di conferenze e brevetti. Presenta inoltre alcuni incisi sulla differenza tra *online ed offline recognition*, sulla tecnologia utilizzabile per digitalizzare il tratto scritto, sulle proprietà della calligrafia delle persone come di alcuni problemi nel riconoscimento della scrittura. Sono infine esaminati alcuni algoritmi di riconoscimento, di *preprocessing and postprocessing techniques*, sia sperimentali che in libero commercio.

Come estratto di questo *paper* riportiamo solamente le tabelle riassuntive dei vari esperimenti svolti negli anni e sui dispositivi / algoritmi in commercio a quei tempi che permettevano il processo di *handwritten recognition*.

EXPERIMENTAL SYSTEMS FOR HANDPRINTED CHARACTERS							
Author	Alphabet Size	Alphabet	Stroke number & order freedom	Segmentation	Recognition Method	Rate (percent)	Writer Dependent
Yhap '81 (IBM, USA)	2260	Chinese (square)		Boxes	Stroke code sequence	80	No
Nakagawa '82 (academia, Japan)	1162	Kana, Kanji (square)		Boxes	72 stroke codes	96 (with 3 tries)	No
Yoshida '82 (NEC, Japan)	2100	Kana, Kanji (square)	Number free	Boxes	Stroke code sequence	89 (Kanji)	No
Wakahara '84 (NTT, Japan)	1945	Kana, Kanji (square)	Number and order free	Boxes	29 stroke codes	90 (Katakana)	No
Ye '84 (acad., Switzerland)	500	Kana, Kanji (square)	Order free	Boxes	Elastic match directions	98.3	No
Murase '85 (NTT, Japan)	1991	Kanji (cursive)	Order free	Boxes	Elastic match x/y coordinates	95.2	No
Sato '85 (academia, Japan)	559	Chinese (square)	Order free	Boxes	Stroke code sequence	92 (order free)	No
Yurugi '85 (OKI, Japan)	3280	Hiragana	Order free	Structure analysis	99 (constrained)	96.3	No
Mandler '85 '87 (AEG, Germany)	39	Kanji (square)	Order free	Boxes	Elastic match x/y, directions	97.1	Yes
Tappert '87 (IBM, USA)	44	Kana	Number free	Boxes	Elastic match x/y coordinates	95.5	No
Shiau '88 (ERSO, Taiwan)	5400	Kanji (cursive)	Order free	Boxes	Stroke code sequence	96.2	Yes
		Alphanumeric	Order free	Boxes	76 stroke codes	97.2	Yes
		Special	Order free	Boxes	Elastic match, dir. Features	97.4	No
		Chinese (square)	Order free	Boxes	21 stroke codes	99.4(features)	No

Figura 12: experimental systems for handprinted characters

EXPERIMENTAL SYSTEMS FOR CURSIVE SCRIPT				
Author	Segmentation	Method	Rate (percent)	Writer Dependent
Harmon '61 '62 (BTL, USA)	External	features	60-90	No
Burr '80 (BTL, USA)	Special	directions	90	Yes
Tappert '82 (IBM, USA)	Internal	elastic match	100 (dictionary)	Yes
Higgins & Whitrow '84 '87 (academia, UK)	Internal	dictionary	97	Yes
		directions, heights		
		elastic match		
		letter digrams		
		features		
		letter quadgrams		
		dictionary		

Figura 13: experimental systems for cursive script

Un modello che nei primi anni 2000 forniva lo standard dello stato dell' arte è HMM, ovvero *hidden Markov model*.

Per meglio comprendere gli HMM usati per il riconoscimento della scrittura in *real time* abbiamo studiato due articoli:

- ***Real-time on-line unconstrained handwriting recognition using statistical methods***
- ***HMM based online handwriting recognition***

Entrambi gli articoli sono dello stesso periodo (1995, 1996).

Il primo affronta il problema del riconoscimento automatico del testo scritto a mano senza vincoli basandosi su metodi statistici, come *hidden Markov model* (HMM).

Nel riconoscimento on-line i dati vengono raccolti su una tavoletta elettronica che traccia il movimento della penna, conservando le informazioni temporali. Precedentemente venivano usati approcci al problema come TDNN per il riconoscimento di soli caratteri discreti. In quest'articolo vengono presentati metodi statistici, come i modelli Markov nascosti (HMM) che erano utilizzati con successo per il riconoscimento vocale e da loro applicati anche riconoscimento automatico della grafia. Molti altri articoli come *K. Nathan, J. R. Bellegarda, D. Nahamoo, and E. J. Bellegarda. "On-Line Handwriting Recognition Using Continuous Parameter Hidden Markov Models". In ICASSPSS, volume 5, pages 121-124, 1993.* utilizzano HMM con parametri continui che vengono utilizzati per riconoscere i caratteri scritti in modo isolato. Invece, *T. Starner, J. Makhou, R. Schwartz, and G. Chou. "On-Line Cursive Handwriting Recognition Using Speech Recognition Methods". In ICASSP94, volume 5, pages 125-128, 1994.* esegue il riconoscimento della scrittura a mano in stile puramente corsivo e dipendente da un unico utente utilizzando un sistema sviluppato per il riconoscimento vocale.

In questo documento viene descritto un sistema basato su HMM che non è limitato a uno stile particolare, cioè puramente corsivo o puramente stampatello. La scrittura può essere quindi una qualsiasi combinazione dei 2 stili, una situazione che si incontra molto frequentemente nella pratica. Gli autori si sono molto impegnati nella progettazione di un sistema in grado di funzionare in tempo reale su piccole piattaforme PC con memoria limitata. Sono stati fatti compromessi al fine di ridurre i requisiti di calcolo e di memoria anche a scapito della precisione. Il riconoscimento così sviluppato avviene in due fasi. Nel primo passaggio viene utilizzato un modello più semplice per generare un breve elenco di stringhe definite come potenziali candidate. Questa operazione viene definita corrispondenza rapida (FM). Successivamente, viene utilizzato un modello più costoso dal punto di vista computazionale per riordinare ogni parola nel sopraccitato elenco. Vengono inoltre illustrati i risultati nel modo più generale possibile ovvero indipendenti dall'utente finale e per diverse dimensioni del vocabolario.

Dal momento che per l'esperimento erano principalmente interessati alle *performance* indipendenti dallo scrittore, hanno creato un database con dati raccolti da un *pool* sufficientemente ampio di scrittori. Circa 100.000 elementi sono stati raccolti da circa 100 diverse persone. Il set di addestramento consisteva in parole scelte da un lessico di parole di oltre 20.000 parole e caratteri discreti scritti in modo non contiguo. Hanno richiesto ai soggetti di scrivere nel loro stile naturale richiedenogli di scrivere su una linea orizzontale. Non sono state

fornite altre istruzioni o indicazioni relative allo stile di scrittura. Il set di test era composto da dati raccolti da un insieme separato di 25 scrittori e consisteva elementi univoci scelti a caso dallo stesso lessico.

Come previsto, i dati hanno manifestato caratteristiche che hanno permesso di suddividerli in tre macro-categorie e precisamente:

- puramente stampatello
- miste stampatello e corsivo
- puramente corsivo.

Alcuni esempi sono illustrati nella Figura 9.

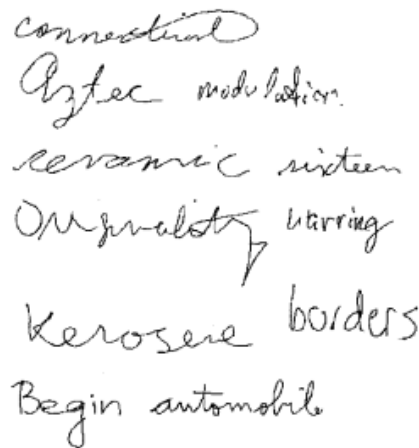
The image shows six lines of handwritten text in a cursive script. The words are: 'connecticut', 'Qztec modulation', 'ceramic system', 'Originality Warring', 'Kerosene borders', and 'Begin automobile'. The handwriting is fluid and connected, characteristic of cursive.

Figura 14: esempi di scrittura in corsivo

Il processo di *training* non vuole trovare una modellizzazione di ogni carattere diverso dagli altri, bensì si vuole costruire un modello per ogni variazione significativa di ogni simbolo. Ad esempio, un carattere può differire per il numero di tratti della penna, la direzione del movimento della penna o nella forma effettiva stessa. Il *training* è stato quindi implementato con una procedura automatica non supervisionata che viene utilizzata per identificare le variazioni che chiamiamo *lexemes*. Per questo set sono stati generati automaticamente circa 150 *lexemes*. Le singole forme di base, stati singoli e multipli, sono quindi stati addestrati per ogni *lexemes*. Dal momento che in media ogni HMM è costituito da 6 stati, questo si tradurrà in 900 stati distinti. Per ridurre i parametri nel sistema, i singoli stati sono condivisi tra e all'interno delle forme di base.

La scelta di sviluppare tale algoritmo è stata obbligata dalla mancata disponibilità degli utenti occasionali dei sistemi di riconoscimento della scrittura ad essere disposti a sottoporsi a sforzi e dedicare tempo per adattare un sistema *tailor made* ai propri fabbisogni. Inoltre esistono anche ambiti di utilizzo di tali algoritmi che non possono essere utilizzabili da una singola persona.

	<b>Small Vocab. (3K)</b>	<b>Medium Vocab. (12K)</b>	<b>Large Vocab. (21K)</b>
<b>Fast Match</b>	14.9%	25.1%	27.9%
<b>Detailed Match</b>	9.0%	14.9%	18.9%

Figura 15: Writer Independent word error rates

	<b>Small Vocab. (3K)</b>	<b>Medium Vocab. (12K)</b>	<b>Large Vocab. (21K)</b>
<b>Fast Match</b>	0.29	0.33	0.36
<b>Detailed Match</b>	0.40	0.45	0.48

Figura 16: Average recognition time in seconds for one word

La figura 10 riassume i risultati per varie dimensioni del vocabolario. I risultati dettagliati della corrispondenza sono stati ottenuti prendendo le prime ipotesi dalla corrispondenza veloce e presentandole al modello a più stati. Con un piccolo vocabolario, otteniamo un tasso di errore, indipendente dallo scrittore inferiore al 9%. Come previsto, questo aumenta con le dimensioni del lessico (o con la *perplexity*). Il tasso di errore con un vocabolario consistente è di circa il 19%. Nel testo viene previsto un calo dell'errore significativo, se il sistema di riconoscimento appena sviluppato viene implementato insieme a modelli di linguaggio statistico pre-esistenti.

La figura 11 consente di evincere i tempi di riconoscimento per una singola parola con i diversi modelli. I tempi di riconoscimento di una parola su una piattaforma workstation IBM RS/6000 vanno da 0,4 sec. per un vocabolario ridotto a 0,48 sec. per le attività di un vocabolario di grandi dimensioni. Sulle piattaforme PC standard di classe 486, osserviamo tempi di riconoscimento che sono superiori da 4 a 5 volte le suddette cifre, ma pur sempre sufficienti per la definizione come riconoscimento istantaneo.

Nel secondo articolo si parla sempre di HMM che attraverso un processo simile nei modi, ma non negli obiettivi a quello di prima giunge alle stesse conclusioni di ridefinizione del precedente stato dell'arte.

Infatti hanno implementato il riconoscimento della scrittura a mano sempre sul modello Markov nascosto (HMM), come prima, ma incorporandolo in un complesso modello linguistico stocastico per il riconoscimento della grafia. In questo algoritmo i HMM sono concatenati per formare i modelli delle lettere, che sono ulteriormente incorporati in un modello linguistico stocastico. Oltre a una migliore modellazione linguistica, hanno introdotto nuove funzionalità di riconoscimento della grafia di vario tipo. Alcune di queste funzionalità hanno proprietà di invarianza e alcune altre sono segmentali, coprendo un'area più ampia del modello di input. Abbiamo raggiunto un tasso di riconoscimento indipendente dallo scrittore del 94,5% su 3.823 campioni di parole scritte a mano da 18 scrittori diversi che coprono un vocabolario particolarmente ridotto a sole 32 parole.

Un altro articolo più recente esemplificativo per il nostro argomento e che ridefinisce gli standard rispetto all'HMM è:

### *Online handwriting recognition with support vector machines - a kernel approach*

In questo articolo è presentato, al contrario di prima, un solo nuovo approccio alla classificazione per il riconoscimento della calligrafia online. La tecnica combina il *dynamic time warping* (DTW) e *support vector machines* (SVM) creando un nuovo kernel SVM. Questo nuovo kernel viene chiamato *Gaussian DTW* (GDTW).

Questo approccio porta ad un vantaggio principale rispetto alle comuni tecniche HMM. Ovvero non viene usato un modello per le densità condizionali della classe generata. Invece, affronta direttamente il problema della discriminazione delle classi creando dei confini e quindi è meno sensibile alle ipotesi di modellizzazione.

Il metodo proposto in questo articolo può essere ed è applicato in modo diretto a tutti i problemi di classificazione, in cui un DTW fornisce una misura ragionevole della distanza. Come esemplificativo del nostro algoritmo mostriamo solo gli esperimenti sui dati di scrittura a mano con UNIPEN, ottenendo risultati paragonabili a una tecnica basata su *hidden Markov model* che, come detto prima, allora forniva lo stato dell'arte in questo ambito.

Le SVM sono classificatori discriminatori basati sul principio di minimizzazione del rischio strutturale di Vapnik. Possono infatti implementare confini decisionali flessibili in spazi delle caratteristiche di dimensioni elevate. La regolarizzazione implicita della complessità del classificatore evita l'overfitting e soprattutto questo porta a buone generalizzazioni. Alcune proprietà sono comunemente riconosciute come le ragioni per il successo delle SVM in problemi reali:

- l'ottimalità del risultato del training è garantita,
- esistono algoritmi di *training* veloci,
- è necessario un *training labeled set* non eccessivamente grande.

Gli algoritmi basati su HMM hanno dimostrato di modellare molto bene la struttura complessa dei dati di scrittura a mano in *real time*.

L'approccio generativo, quello di quest'esempio, è davvero ottimale solo se i modelli sottostanti sono accurati. Infatti nei casi di applicazioni pratiche si tende ad usare i modelli precedenti in quanto più affidabili che permettono di ottenere misure più realistiche.

In queste situazioni gli approcci discriminatori che non mirano a stimare le densità condizionali di classe, ma affrontano direttamente la discriminazione creando confini di classe, sono una scelta promettente. Come indicato in precedenza, le SVM appartengono a questa categoria di classificatori. L'esempio presentato nell'articolo citato si suppone sia il primo che utilizzi SVM nel HWR on-line.

Le tecniche SVM comuni sono state sviluppate per uno spazio di funzionalità con una dimensione fissa, mentre le sequenze di scrittura a mano hanno linee di lunghezza e durata variabile. Una soluzione ad hoc per superare questa incompatibilità è stata da loro trovata ad esempio in un ridimensionamento lineare della scrittura ad un numero fisso di campioni.

Gli esperimenti si basano sulla sezione 1a, 1b e 1c (rispettivamente cifre, caratteri maiuscoli e minuscoli) del database UNIPEN Train-R01/V07. Per

queste sezioni, la dimensione del set di dati è rispettivamente di 16K, 28K e 61K. Esempi di dati UNIPEN sono stati mostrati nella figura 12. Formazione e set di test sono stati presi in modo disgiunto. Va detto che UNIPEN è costituito da dati molto difficili a causa della varietà di scrittori e dati rumorosi o etichettati in modo errato. Abbiamo utilizzato il database senza pulizia per essere il più paragonabile possibile ad altri rapporti di classificazione.












$T$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$
					
$\phi^*$					
$D(T, P_j)$	0	0.20	0.71	0.99	10.04
$K(T, P_j)$	1	0.70	0.28	0.17	0.00

Figura 17: gaussian dynamic time warping GDTW kernel

Negli ultimi anni, dal 2013 in poi, sono stati sviluppati dei metodi basati sul *machine learning* e più nello specifico con LSTM.

Come esempio abbiamo scelto l'ultima fonte possibile, ovvero l'articolo ***Fast multi-language LSTM-based online handwriting recognition*** del 08/02/2020.

Loro descrivono un sistema di scrittura online in grado di supportare il riconoscimento di 102 lingue usando un *deep neural network architecture*. Questo nuovo sistema ha completamente sostituito tutti i sistemi precedenti basati su segmentazione e decodifica e ha ridotto il tasso di errore del 20–40% rispetto alla maggior parte delle lingue. Inoltre, riportano nuovi risultati allo stato dell'arte su IAM-OnDB sia per l'impostazione di set di dati aperti che chiusi. Il sistema combina metodi di riconoscimento sequenziale con una nuova codifica di input che utilizza curve di Bézier. Ciò porta a tempi di riconoscimento fino a 10 volte più veloci rispetto al nostro sistema precedente. Attraverso una serie di esperimenti, determiniamo la configurazione ottimale dei nostri modelli e riportiamo i risultati della nostra configurazione su una serie di set di dati pubblici aggiuntivi.

Successivamente ai modelli di Markov sono stati implementati degli approcci ibridi che combinano HMM e reti neurali feed-forward. I primi modelli privi di HMM erano basati su reti neurali ritardate (TDNN) e ricerche più recenti si concentrano su metodi derivati da variazioni di reti neurali ricorrenti (RNN) come le reti di memoria a breve termine (LSTM).

La rappresentazione dei dati calligrafici online è stato un argomento di ricerca per molto tempo. I primi approcci erano basati su funzionalità, in cui ogni punto è rappresentato usando una serie di funzionalità o utilizzando funzionalità globali per rappresentare interi caratteri.

Più recentemente, il *deep learning* ha rivoluzionato la maggior parte degli sforzi ingegneristici pregressi sulle caratteristiche e li ha sostituiti con rappresentazioni apprese in molti settori, ad esempio il discorso, la visione artificiale e l'elaborazione del linguaggio naturale.

Insieme ai cambiamenti dell'architettura delle reti neurali, sono cambiate anche le metodologie di addestramento, passando dall'affidamento alla segmentazione esplicita alla segmentazione implicita usando la perdita di classificazione

temporale connessa (CTC), o agli approcci *encoder-decoder* addestrati con la stima della massima verosimiglianza.

Il risultato del riconoscimento viene quindi ottenuto utilizzando un algoritmo di decodifica per la ricerca di percorsi ottimali sul reticolo delle ipotesi che incorpora fonti di conoscenza aggiuntive come i modelli linguistici. Questo sistema si basa su numerosi metodi euristici di preelaborazione, segmentazione ed estrazione delle caratteristiche che non sono più presenti nel sistema da loro presentato. Questo nuovo sistema riduce la quantità di personalizzazione richiesta ed è costituito da una semplice pila di LSTM bidirezionali (BLSTM), un singolo livello Logits e la perdita CTC

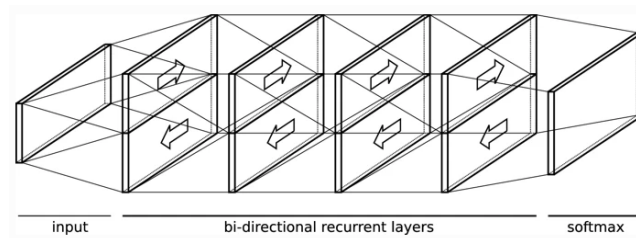


Figura 18: andamento dei layer tra input e softmax per LSTM bidirezionali

L'architettura dell'algoritmo implementato in questo articolo è simile a quella che spesso viene utilizzata nel contesto della modellizzazione acustica per il riconoscimento vocale, in cui viene definita CLDNN (Convoluzioni, LSTM e DNN), ma differisce da essa in quattro punti principali.

- non sono utilizzati livelli di convoluzione, che nella nostra esperienza non forniscono un valore aggiunto per reti di grandi dimensioni addestrate su set di dati altrettanto grandi con sequenze relativamente brevi (rispetto all'input vocale) che si vedono in genere nel riconoscimento della grafia.
- sono utilizzati LSTM bidirezionali, che a causa di vincoli di latenza non sono fattibili nei sistemi di riconoscimento vocale.
- quest'architettura non utilizza livelli aggiuntivi completamente collegati prima e dopo i livelli bidirezionali del LSTM.
- il processo di allenamento del sistema è sviluppato usando *CTC loss*.

In questo modo sono resi superflui molti componenti tipici dei sistemi precedenti come ad esempio l'estrazione e la segmentazione delle funzioni. Tutte le caratteristiche euristiche di difficile estrazione sono ora invece implicitamente apprese dai dati di *training*.

Il modello accetta come input una serie temporale di lunghezza  $T$  che codifica l'input dell'utente e lo passa attraverso diversi livelli del LSTM bidirezionali che apprendono durante il processo di *training* la struttura dei caratteri ( $v_1, \dots, v_T$ ) sotto forma di pesi dei collegamenti. L'output del layer LSTM finale viene passato attraverso un layer softmax che porta a un vettore i cui elementi sono una sequenza di distribuzioni di probabilità sui caratteri per ogni fase temporale.

Per lo specifico algoritmo sono state usate come caratteristiche in input 23 differenti *features* per punto, in quanto averne un numero più elevato non porta ad un incremento di prestazione proporzionale al lavoro svolto per ricavarle. Anzi, è stato provato anche che la rimozione di tali caratteristiche porta a risultati migliori.

Gli input usati in questo specifico esempio sono dei vettori penta-dimensionali  $(x_i, y_i, t_i, p_i, n_i)$  contenenti le coordinate del tocco della penna i-esimo, l'istante temporale del tocco in secondi,  $p_i$  indica se il punto corrisponde a un tratto di attacco o stacco della penna infine  $n_i$  indica l'inizio di un nuovo tratto. Entrambi i val  $p_i, n_i$  sono variabili booleane ovvero assumono solo valori 0 o 1.  $p_i$  assume valore 0 per lo stato *pen-up* invece assume valore 1 per *pen-down*.  $n_i = 1$  indica il caso di inizio di un nuovo tratto, altrimenti 0. Questi punti sono quindi interpolati con delle curve di Bézier.

Le LSTM sono diventate una delle tipologie di RNN più comunemente utilizzate perché sono facili da addestrare e danno buoni risultati. In tutti gli esperimenti riguardanti l'*handwriting recognition*, utilizziamo LSTM bidirezionali, ovvero elaboriamo la sequenza di input sia in avanti che indietro e uniamo gli stati di output di ciascun layer prima di inviarli al layer successivo. Il numero esatto di livelli e nodi viene determinato empiricamente per ogni script. L'output dei layer LSTM ad ogni timestep viene inserito in un layer softmax per ottenere una distribuzione di probabilità sui caratteri C possibili nello script (inclusi spazi, segni di punteggiatura, numeri o altri caratteri speciali), oltre all'etichetta vuota richiesta dal CTC perdita e decodificatore.

Come spiegato precedentemente possiamo utilizzare classi per LSTM diverse tra loro. Infatti risulta spesso conveniente, per le lingue che usano spazi per separare le parole, utilizzare un modello linguistico basato su parole. La classe dei caratteri è invece utile per quelle lingue caratterizzate piuttosto da ideogrammi. Per questa classe viene aggiunto un punteggio che aumenta il valore delle lettere dall'alfabeto. Questa funzione caratteristica fornisce un segnale forte per caratteri rari che potrebbero non essere riconosciuti con sicurezza dall'LSTM e che gli altri modelli linguistici potrebbero non pesare abbastanza da essere riconosciuti. Questa funzione è stata ispirata dal nostro sistema precedente.

Il processo di *training* avviene in due fasi con due *dataset diversi*

- *Training end-to-end* del modello di rete neurale usando la perdita CTC con un ampio set di dati
- Ottimizzazione dei pesi del decodificatore utilizzando l'ottimizzazione bayesiana attraverso i processi gaussiani di Vizier.

In questo articolo sono stati riportati, ove possibile, i risultati con *data set* pubblici e chiusi, ovvero i *dataset* per il *training* e il *testing* sono utilizzati solamente con protocolli standard. Inoltre, sono stati presentati i risultati anche per set di dati pubblici in modalità aperta, ovvero in cui il modello viene formato sui nostri dati.

### IAM-OnDB

Il set di dati IAM-OnDB è probabilmente il dataset più utilizzato per il riconoscimento della grafia online. È composto da 298.523 caratteri in 86.272 parole da un dizionario di 11.059 parole scritte da 221 scrittori. La separazione del set di dati IAM-OnDB è quella standard, ovvero un set di training, due set



di validazioni e un set di test contenente rispettivamente 5363, 1438, 1518 e 3859 righe di elementi.

Input	lstm	64 Nodes	128 Nodes	256 Nodes
Raw	1 Layer	6.1	5.95	5.56
	3 Layers	<b>4.03</b>	4.73	4.34
	5 Layers	4.34	<b>4.20</b>	<b>4.17</b>
Curves	1 Layer	6.57	6.38	6.98
	3 Layers	4.16	<b>4.16</b>	4.83
	5 Layers	<b>4.02</b>	4.22	<b>4.11</b>

Figura 19: Confronto dei tassi di errore dei caratteri (inferiore è migliore) sul set di test IAM-OnDB per diverse configurazioni di livelli LSTM.

E' stato eseguito anche uno studio più approfondito per stabilire il numero di layer e nodi per layer ottimale sia per i *raw input* che per i *curve input*.

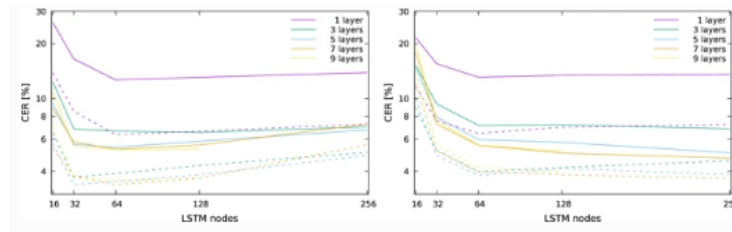


Figura 20: CER di modelli addestrati sul set di dati IAM-OnDB con diversi numeri di livelli LSTM e nodi LSTM utilizzando input non elaborati (a sinistra) e curve (a destra). Le linee continue indicano i risultati senza alcun modello di linguaggio o funzione nella decodifica e le linee tratteggiate indicano i risultati con il sistema completamente sintonizzato.

Sono stati eseguiti esperimenti solamente con le funzioni basilari (Figura 4, linee continue), e solo in seguito sono state implementate per valutarne gli effetti. Osserviamo che per entrambi i formati di input, sia a 3 che a 5 livelli, l'utilizzo di più funzioni non fornisce alcun miglioramento apprezzabile. Inoltre, l'uso di 64 nodi per livello è sufficiente, poiché le reti più ampie offrono solo piccoli miglioramenti.

System	CER (%)	WER (%)
Frinken et al. BLSTM [2]	12.3	25.0
Graves et al. BLSTM [15]	11.5	20.3
Liwicki et al. LSTM [32]	-	18.9
This work (curve, 5x64, no FF)	5.9	18.6
This work (curve, 5x64, FF)	<b>4.0</b>	<b>10.6</b>
Our previous BLSTM [25]*	8.8	26.7
Combination [32]*	-	13.8
Our segment-and-decode [25]*	4.3	10.4
This work (production system)*	<b>2.5</b>	<b>6.5</b>

Figura 21: Tassi di errore sul set di test IAM-OnDB rispetto allo stato dell'arte e al nostro sistema precedente.

Infine viene mostrato un confronto tra questo nuovo sistema e i precedenti sempre basati sul set di dati IAM-OnDB come da Figura 16.

Questo metodo stabilisce un nuovo risultato all'avanguardia nel caso in cui si faccia affidamento su dati chiusi utilizzando IAM-OnDB, nonché quando si fa affidamento sui nostri dati che utilizziamo per il nostro sistema di produzione, escludendo da qualsivoglia processo lo IAM- Dati OnDB.

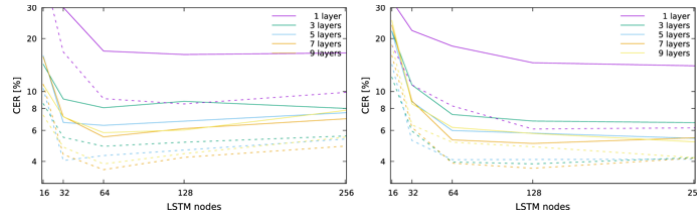


Figura 22: CER di modelli addestrati sul set di dati IBM-UB-1 con diversi numeri di livelli LSTM e nodi LSTM utilizzando input non elaborati (a sinistra) e curve (a destra). Le linee continue indicano i risultati senza alcun modello di linguaggio o funzione nella decodifica e le linee tratteggiate indicano i risultati con il sistema completamente sintonizzato

Per capire meglio da dove provengono i miglioramenti, discutiamo le differenze tra il sistema HMM all'avanguardia e il sistema LSTM in quattro punti:

- pre-elaborazione degli input ed estrazione delle caratteristiche,
- architettura della rete neurale,
- addestramento,
- decodifica CTC e metodologia di addestramento modello.

La preelaborazione dell'input nel metodo presentato differisce solo in maniera minore: la coordinata  $x$  utilizzata non viene trasformata utilizzando un filtro passa-alto, non vengono divise le righe di testo usando gli spazi vuoti, non vengono rimossi i tratti ritardati, e neppure vengono eseguire correzioni dell'inclinazione o altre preelaborazioni. La principale differenza deriva dall'estrazione delle funzionalità. Contrariamente alle 25 funzioni per punto utilizzate nei modelli pregressi, ne vengono sfruttate solo 5 (non elaborate) o 10 (curve). Mentre le 25 funzioni includevano sia le caratteristiche temporali (posizione nella serie temporale) sia quelle spaziali (rappresentazione offline), il nostro lavoro utilizza solo la struttura temporale. Contrariamente ai sistemi precedenti, l'uso di una rappresentazione più compatta (e la riduzione del numero di punti per le curve) consente di apprendere una rappresentazione di caratteristiche, compresa la struttura spaziale, nei primi *layers* della rete neurale. L'architettura della rete neurale differisce sia nella struttura interna della cella LSTM che nella configurazione generale.

Invece di fare affidamento su un singolo strato LSTM bidirezionale di larghezza 100, vengono sperimentate una serie di varianti di configurazione come raffigurato dettagliatamente in figura 15. Si nota inoltre che è particolarmente importante avere più di un livello per apprendere una rappresentazione significativa senza estrazione delle caratteristiche.

### IBM-UB-1

Un altro set di dati in lingua sempre inglese accessibile pubblicamente è il set di dati IBM-UB-1. Dai set di dati disponibili in esso, utilizziamo il set di

dati della *query* inglese, che consiste di 63.268 parole inglesi scritte a mano. Poiché questo set di dati non è stato usato spesso nella letteratura accademica, viene anche proposto un protocollo di valutazione. Abbiamo diviso questo set di dati in 4 parti con *ID writer* non sovrapposti: 47.108 articoli per il *training*, 4690 per l'ottimizzazione dei pesi, 6134 per la convalida e 5336 per il *test*. Vengono eseguite una serie simile di esperimenti, come abbiamo fatto per IAM-OnDB per determinare la giusta profondità e larghezza della nostra architettura di rete neurale. I risultati di questi esperimenti sono mostrati in Figura 17.

Le conclusioni tratte per questo set di dati sono simili a quelle che abbiamo elaborato per IAM-OnDB. Si utilizzano reti con 5 strati di LSTM bidirezionali ciascuno con 64 celle poiché questa configurazione fornisce una buona precisione. Le reti meno profonde e meno ampie hanno prestazioni sostanzialmente peggiori, ma le reti più grandi offrono solo piccoli miglioramenti a discapito di grandi potenze di calcolo necessarie. Ciò è vero indipendentemente dal metodo di elaborazione dell'input scelto e, di nuovo, non viene apprezzata una differenza significativa nella precisione tra la rappresentazione grezza e la curva.

System	CER (%)	WER (%)
This work (curve, 5x64, no FF)	6.0	25.1
This work (curve, 5x64, FF)	4.1	15.1
Segment-and-decode from [25]	6.7	22.2
This work (production system) (Sect. 5)*	4.1	15.3

Figura 23: Tassi di errore relativi sul set di test IBM-UB-1 rispetto al sistema precedente

Come fatto prima, riportiamo alcuni risultati caratteristici di questo modello con questo *dataset* esemplari per un confronto con un altro sistema di modellizzazione attuale, nonché i risultati per il sistema presentato precedentemente nella Figura 18.

Notiamo che il nostro sistema attuale è migliore di una percentuale che va dal 32% fino al 38% sia per CER che per WER<sup>2</sup>, rispetto al precedente approccio di segmentazione e decodifica.

La mancanza di miglioramento del tasso di errore durante la valutazione sul nostro sistema di produzione è dovuta al fatto che i nostri set di dati contengono degli spazi.

### Ottimizzazione dei parametri della rete neurale sui nostri dati interni

CONTROLLARE CE NON CI SIA UN PARAGRAFO DOVE DICONO CHE HANNO CREATO UN LORO DATASET

I nostri set di dati interni consistono in vari tipi di dati per il processo di *training*, la cui quantità varia in base allo script. Porta a set di dati più eterogenei rispetto a set di dati accademici come IBM-UB-1 o IAM-OnDB che sono stati raccolti in condizioni standardizzate. Il numero di campioni di addestramento varia da decine di migliaia a diversi milioni per script, a seconda della complessità e dell'utilizzo. Forniamo ulteriori informazioni sulla dimensione della nostra formazione interna e set di dati di test nella Figura 19.

<sup>2</sup>sono due variabili caratteristiche dei programmi di RNN o LSTM per il riconoscimento vocale o della scrittura il cui significato è crossover error rate (indica la percentuale di falsi positivi o veri negativi) e word error rate

Language	en	es	de	ar	ko	th	hi	zh
Internal test data (per language)								
Items	32,645	7136	14,408	11,617	22,951	23,608	9030	197,547
Characters	162,367	40,302	83,231	84,017	55,654	109,793	36,726	312,478
Internal training data (per script)								
Items	3,293,421			570,375	3,495,877	207,833	1,004,814	5,969,179
Characters	15,850,724			4,597,255	4,770,486	989,520	5,575,552	7,548,434
Unique supported characters	295			337	3524	195	197	12,726
System	CER (%)							
Segment-and-decode [25]	7.5	7.2	6.0	14.8	13.8	4.1	15.7	3.76
BLSTM (comparison) [25]	10.2	12.4	9.5	18.2	44.2	3.9	15.4	–
Model architecture (this work)	$5 \times 224$			$5 \times 160$	$5 \times 160$	$5 \times 128$	$5 \times 192$	$4 \times 192$
(2) BLSTM-CTC baseline curves	8.00	6.38	7.12	12.29	6.87	2.41	7.65	1.54
(3) + n-gram LM	6.54	4.64	5.43	8.10	6.90	1.82	7.00	1.38
(4) + character classes	6.60	4.59	5.36	7.93	6.79	1.78	7.32	1.39
(5) + word LM	6.48	4.56	5.40	7.87	–	–	7.42	–
Avg. latency per item	(ms)							
Segment-and-decode [25]	315	359	372	221	389	165	139	208
This work	23	25	26	14	20	13	19	30
Number of parameters (per script)								
Segment-and-decode [25]	5,281,061			5,342,561	8,381,686	6,318,361	9,721,361	–
This work	5,386,170			2,776,937	3,746,999	1,769,668	3,927,736	7,729,994

Figura 24: Tassi di errore dei caratteri sui dati di convalida utilizzando successivamente più componenti di sistema sopra descritti per inglese (en), spagnolo (es), tedesco (de), arabo (ar), coreano (ko), thailandese (th), hindi (zh) e cinese (zh) insieme al rispettivo numero di elementi e caratteri nei set di dati di test e addestramento. Le latenze medie per tutte le lingue e i modelli sono state calcolate su una CPU Intel Xeon E5-2690 a 2,6 GHz

La migliore configurazione per questo sistema è stata identificata eseguendo più esperimenti su una gamma di profondità e larghezze dei *layer* sui nostri *dataset* interni. In modo del tutto simile agli esperimenti illustrati nelle figure 15 e 17, aumentando la profondità e la larghezza dell'architettura di rete si ottengono rendimenti decrescenti abbastanza rapidamente. Tuttavia, l'*overfitting* è meno pronunciato probabilmente perché i nostri set di dati sono sostanzialmente più grandi dei set di dati disponibili pubblicamente. Per gli esperimenti con i nostri set di dati sono utilizzati gli input della curva di Bézier che offrono prestazioni leggermente migliori in termini di precisione rispetto alla codifica degli input non elaborati, ma sono molto più veloci da addestrare e valutare a causa di più brevi lunghezze di sequenza.

#### Prestazioni di sistema e discussione

L'impostazione descritta in questo documento ha ottenuto risultati migliori rispetto allo stato dell'arte poiché si basa sull'elaborazione dell'input con l'interpolazione della *spline* di Bézier, seguita da 4–5 strati di LSTM bidirezionali di larghezza variabile, seguiti a loro volta da uno strato finale di softmax. Per ogni script è stata determinata sperimentalmente la migliore configurazione attraverso più sessioni di addestramento.

Inoltre, sono stati mostrati i miglioramenti relativi dei tassi di errore nelle lingue per le quali disponiamo di set di dati di valutazione con più di 2000 elementi Fig.20. Questa nuova architettura offre prestazioni tra il 20 e il 40% (relative) migliori in quasi tutte le lingue rispetto a qualsivoglia modello precedente.

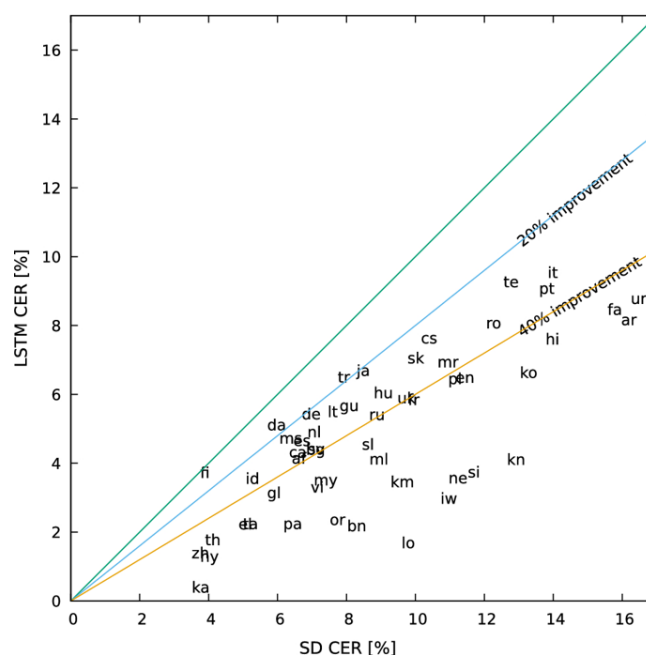


Figura 25: Un confronto tra i CER per l'LSTM e il sistema di segmentazione e decodifica (SD) per tutte le lingue sui nostri set di test interni con oltre 2000 articoli. Il diagramma a dispersione mostra il codice del linguaggio ISO in una posizione corrispondente al CER per il sistema SD (asse x) e il sistema LSTM (asse y). I punti sotto la diagonale sono miglioramenti di LSTM su SD. La trama mostra anche le linee di miglioramento relativo del 20% e del 40%

### Differenze tra IAM-OnDB, IBM-UB-1 e i nostri set di dati interni

Per capire come i diversi set di dati si relazionano tra loro, abbiamo eseguito una serie di esperimenti e valutazioni con l'obiettivo di evidenziare le differenze tra i set di dati.

Abbiamo allenato il sistema in vari esperimenti indipendenti gli uni dagli altri con i tre diversi set di allenamento, quindi abbiamo valutato ciascun sistema su tutti e tre i set di test (Figura 21).

L'architettura della rete neurale è la stessa di quella che abbiamo determinato in precedenza (LSTM bidirezionali a 5 strati di 64 celle ciascuno) con le stesse funzioni, con pesi sintonizzati sul set di dati di tuning corrispondenti per tutti gli esperimenti. Gli input vengono sempre elaborati utilizzando le curve di Bézier. Per riscontrare delle differenze possiamo innanzitutto mettere in risalto le differenze tra i due *dataset*. IBM-UB-1 ha una scrittura prevalentemente corsiva, mentre IAM-OnDB ha principalmente elementi in stampatello. Ma l'altra

Train/test	IAM-OnDB	IBM-UB-1	Own dataset
IAM-OnDB	3.8	17.7	31.2
IBM-UB-1	35.1	4.1	32.9
Own dataset	3.3	4.8	8.7

Figura 26: Confronto CER durante l'addestramento e la valutazione di IAM-OnDB, IBM-UB-1 e il nostro set di formazione / valutazione caratteri latini

differenza fondamentale che altera maggiormente il funzionamento dei programmi è il fatto che IBM-UB-1 contiene parole singole, mentre IAM-OnDB contiene prevalentemente frasi composte da parole separate da spazi. Ciò si traduce in modelli basati sul *dataset* IBM-UB-1 che non sono in grado di prevedere gli spazi poiché non sono presenti come classe di dati all'interno della classificazione. Risulta inoltre intuitivo che i due dataset, come gli stili non sono interscambiabili nei processi di *training* e *testing*. Infatti lo stile di scrittura stampatello di IAM-OnDB rende più difficile il riconoscimento quando viene valutato un esempio di scrittura corsiva. È anche probabile che la mancanza di struttura semantica e della grammatica, avendo usato un sistema di *training* composto di sole parole renda più difficile il riconoscimento di IAM-OnDB su un sistema addestrato su IBM-UB-1.

I sistemi addestrati su IBM-UB-1 o IAM-OnDB da soli hanno prestazioni significativamente peggiori dello stesso sistema allenato con il *dataset* proprietario, poiché la distribuzione di dati copre una vasta gamma di casi che non sono contemplati nei due set di dati accademici in quanto non seguono degli standard così rigidi. Alcuni esempi possono essere una pessima calligrafia, caratteri sovrapposti, frequenze di campionamento non uniformi e input parzialmente ruotati. La rete addestrata sul set di dati proprietari funziona bene su tutti e tre i modelli in quanto contiene tipi diffusi di classi. Riesce addirittura a funzionare meglio su IAM-OnDB rispetto al sistema addestrato solo su di esso, ma peggio per IBM-UB-1. Questo rende infatti deducibile il fatto che l'utilizzo delle sole parole cursive durante l'addestramento consenta alla rete di apprendere meglio le caratteristiche del campione, rispetto a quando si apprende la separazione dello spazio e altre proprietà della struttura non presenti in IBM-UB-1.