PeterPen

Dario Montagnini, Antonio Musolino,
Manuel Prandini, Giovanni Varricchione

Sommario

Lo scopo che ci siamo posti con questo progetto è stato la creazione di una penna che permettesse il riconoscimento biometrico di un utente tramite la sua scrittura. A tal fine, è stata progettata e realizzata una penna che permettesse la cattura dei dati di scrittura degli utenti, inviandoli ad un server tramite una connessione Wi-Fi. La penna è stata realizzata con chip economici e la scocca tramite una stampante 3D. Sono stati implementati due diversi modelli per il riconoscimento biometrico, uno basato su un algoritmo di deep learning ed uno su distanze fra segnali. Sul dataset disponibile sono state osservate buone performance da entrambi i modelli, con risultati molto interessanti soprattutto per il modello di deep learning.

Scrittura

La scrittura è uno dei tanti tratti che può essere usato in un sistema biometrico. È categorizzato come un tratto biometrico *comportamentale*, in quanto è basato su un'azione appresa dall'utente, oltre al suo stato umorale nel momento della cattura dei dati (altri esempi di questi tratti sono il modo in cui si cammina e la battitura dei tasti). Generalmente, ha una bassa *accuracy* ma un'elevata *acceptability*.
Il tratto è principalmente influenzato da caratteristiche quali la calligrafia, la pressione esercitata e l'inclinazione della penna durante la scrittura.

Stato dell'arte

Tipicamente i sistemi di riconoscimento della scrittura si basano sul riconoscimento della firma dell'utente (ad esempio Karouni et al. [2]). Tali sistemi sono anche detti di "*signature recognition*", l'idea alla base di quest'approccio è che la firma varia notevolmente da individuo a individuo e quindi può essere sfruttata come tratto identificativo. Naturalmente, esistono anche sistemi che permettono il riconoscimento utilizzando qualsiasi parola, in questo caso si parla di "*handwriting recognition*". Mentre i primi sono vulnerabili ad attacchi di *spoofing*, i secondi sono più resistenti ma potrebbero avere performance più basse.
Ci sono due approcci principali al riconoscimento della scrittura: il primo, detto "*statico*" o "*off-line*", si basa su tecniche di *image processing*. La firma viene catturata e poi trasformata in un'immagine, che successivamente sarà analizzata dal modello (esempi sono Sharif et al.[3] e Souza V., Oliveira A. e Sabourin R. [4]). Il secondo approccio, detto "*dinamico*" o "*on-line*", si basa invece sui segnali che vengono catturati tramite dei sensori posti su una penna. Tipici segnali analizzati sono la pressione, l'accelerazione e la rotazione, in quanto caratterizzano la scrittura. In genere, quest'ultimi sono più performanti rispetto ai primi.
Un approccio dinamico molto simile a quello che verrà presentato è stato proposto con la BioPen da De Marsico M., Ponzi F., Scozzafava F. e Tortora G. [1], tuttavia vi sono delle differenze nei modelli proposti. Innanzitutto, nella PeterPen il modulo di Machine Learning usa le LSTM, mentre nella BioPen le SVM. La pressione, inoltre, è usata in modi diversi: mentre nella BioPen veniva misurata la pressione posta dall'utente nella presa della penna, nella PeterPen viene misurata la pressione esercitata sulla punta durante la scrittura. Questo ci ha permesso anche di automatizzare il processo di raccolta dei dati, considerando solo i periodi in cui la pressione misurata superava un certo threshold. Chiaramente questi due approcci all'utilizzo della pressione hanno anche ripercussioni differenti nel riconoscimento dell'utente.

Capitolo 1

Architettura del Sistema

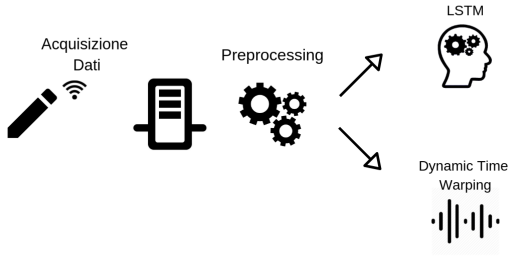

Figura 1.1: Schema dell'architettura

L'architettura del sistema è formata da 4 componenti: la penna, un server per l'acquisizione dei dati, e i moduli di *preprocessing* dei dati e di classificazione. La penna raccoglie i dati tramite i sensori e li invia al server attraverso il modulo Wi-Fi del chip. I dati vengono salvati in appositi file JSON sul server (ad ogni utilizzo della penna è associato un file diverso) e possono essere quindi usati dal modulo di *preprocessing*. Quest'ultimo si occupa principalmente di normalizzare i dati, in modo da renderli utilizzabili dai modelli di classificazione. Un modulo di classificazione è composto da una rete neurale che utilizza delle LSTM, mentre l'altro è basato sull'algoritmo *Dynamic Time Warping*.

1.1 PeterPen

La penna è composta dal seguente hardware:

- un chip NODE MCU ESP8266;
- un accelerometro;
- un giroscopio;
- un sensore di pressione;
- un LED verde;
- un condensatore;
- due resistenze;
- una batteria da 9V;
- un involucro realizzato in PLA.

L'ESP8266, il chip principale della penna, è formato da un micro-controllore e da un modulo Wi-Fi utilizzato per la comunicazione con il server. A questo sono collegati accelerometro e giroscopio (contenuti all'interno di un unico chip), il sensore di pressione e il LED.
Il sensore di pressione misura la forza applicata sulla punta della penna grazie all'ausilio di una molla. Subito dopo il sensore di pressione è posizionato il chip dell'accelerometro e del giroscopio e, dopo questo, si trova il microcontrollore.

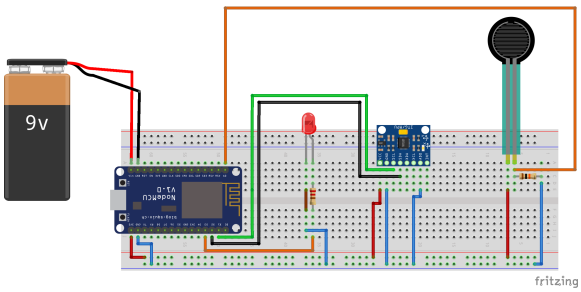

Figura 1.2: Circuito della PeterPen

Figura 1.3: Design della PeterPen, si ringraziano Cristian Cianfarani e Tiziano Grossi

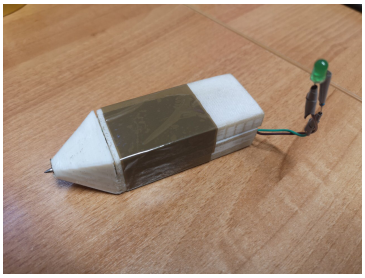

Figura 1.4: PeterPen realizzata

1.1.1 Protocollo di acquisizione

È stato ideato un protocollo per semplificare l'acquisizione e l'invio dei dati. All'accensione della penna (che può essere alimentata o tramite un cavo USB o tramite batteria, usandone o una da 9V (fig. 1.2) o una da 3.3V o 3.6V, alimentandola però tramite il pin 3v3) il LED lampeggia durante la connessione alla rete e al server, o in caso di errore. Una volta connessa, il LED smette di lampeggiare e rimane acceso, questo indica all'utente che può iniziare a scrivere. Durante la scrittura la luce rimane spenta e, se non viene rilevata alcuna pressione per più di 2 secondi, si riaccende indicando la fine della scrittura della parola precedente e l'invio dei dati. È stato inoltre impostato come tempo massimo di scrittura 10 secondi, nel caso in cui l'utente superi tale limite il LED comincerà a lampeggiare, per segnalare l'errore e nessun dato sarà inviato.

1.1.2 Codice sorgente della penna

Il microcontrollore ESP8266 è programmato con un linguaggio derivato dal C, lo stesso che viene usato nella programmazione dei chip Arduino. Tale linguaggio richiede necessariamente l'implementazione di due funzioni in ogni programma. La prima, `void setup()`, è eseguita un'unica volta all'inizio dell'esecuzione, ed è utilizzata per inizializzare le variabili. La seconda, `void loop()`, rappresenta il kernel del programma ed è eseguito continuamente finché il chip è attivo, nel nostro caso l'esecuzione avviene ogni 10 millisecondi.
Nel nostro caso, nella funzione `setup`, vengono invocate altre due funzioni: `mpu6050Begin` che inizializza il chip del giroscopio e dell'accelerometro e `connect` che si occupa di effettuare la connessione al server. La funzione `loop` ad ogni iterazione legge il valore della pressione e, in base allo stato (che è uno fra `READY`, `WRITING` e `ERROR`) e alla variabile `counter_writing` si comporterà diversamente.
Se la pressione registrata è sufficiente e lo stato è `READY`, allora la penna passa allo stato `WRITING`, acquisisce i dati e inizializza il contatore `counter_writing`. Invece, se la pressione è maggiore di un certo threshold, lo stato è `WRITING` e sono passati meno di 10 secondi dall'inizio della scrittura allora acquisiti i dati dai sensori; nel caso in cui sono passati 10 o più secondi allora la penna si sposterà nello stato `ERROR`. Infine, se lo stato è `WRITING` e sono passati almeno 2 secondi da quando non si rileva abbastanza pressione, vengono inviati i dati al server e viene impostato lo stato `READY`.

1.2 Server

Il server è programmato in Node.js, in ascolto sulla porta 8080 mediante una socket TCP. Ad ogni nuova connessione crea un nuovo file JSON all'interno della quale andrà a salvare tutti i dati ricevuti dalla PeterPen, effettuandone precedentemente il parsing. Nel momento in cui la sessione viene chiusa, il server si occupa di gestire la chiusura del file relativo alla stessa.

1.2.1 File JSON

Il file contiene un array di parole, ed ogni parola è composta da una sequenza di lunghezza variabile (ma lunga al più 1000) di vettori composti dalle 7 feature.
Le feature raccolte sono le seguenti:

1. componenti x, y e z dell'accelerometro;
2. componenti x, y e z del giroscopio;
3. valore della pressione.

Capitolo 2

Classificazione

2.1 Preprocessing

Nella fase di preprocessing i dati sono stati normalizzati, effettuando il rescaling delle singole feature, ed è stato aggiunto del padding per rendere i vettori delle singole parole lunghi esattamente 1000. Il padding è stato eseguito per entrambi i modelli, mentre il rescaling solo per il DTW (poiché peggiorava le prestazioni dell'LSTM, abbiamo deciso di non usare i dati scalati in quel caso).

2.1.1 Normalizzazione

I dati sono stati normalizzati utilizzando la funzione di rescaling:

$$rescaling(x) = \frac{x - min}{max - min}$$

A tal fine, come valori minimi e massimi abbiamo usato:

- ± 250 per i dati dell'accelerometro;
- ± 2 per i dati del giroscopio;
- 0 e 1024 per i dati del sensore di pressione.

2.2 DTW

L'algoritmo DTW (*Dynamic Time Warping*) è un algoritmo di programmazione dinamica che permette di misurare la similarità fra due sequenze temporali; la particolarità dell'algoritmo è che riconosce anche sequenze che variano in velocità. Nel nostro caso, un esempio immediato è dato da un individuo che, scrivendo la stessa parola, impiega meno tempo a scriverla nel primo caso che nel secondo.
Il DTW calcola l'*optimal matching* fra due sequenze con le seguenti regole e restrizioni:

1. ogni elemento di una sequenza deve essere associato ad almeno un elemento dell'altra (ma non necessariamente ad uno solo);
2. il primo elemento della prima sequenza deve essere associato al primo della seconda;
3. l'ultimo elemento della prima sequenza deve essere associato all'ultimo della seconda;
4. il matching deve essere *monotono* e *crescente*, ossia, dati due indici della prima sequenza i, j tali che $i < j$, allora non possono esistere due indici della sequenza k, q tali che $k < q$, q è stato associato a i e k è stato associato a j.

Ad ogni *matching* è associato un costo (tipicamente la somma delle distanze euclidee fra ogni coppia di indici associati, che abbiamo usato nella nostra implementazione), e quello ottimale è quello il cui costo è minimo.

L'algoritmo costruisce una matrice $n \times m$, dove n ed m sono rispettivamente le lunghezze delle due sequenze, e riempie ogni cella i, j della matrice con la distanza minima per raggiungerla dalla cella $[0, 0]$; l'output è dato quindi dal valore nella cella $[n, m]$. Si può chiaramente osservare che la complessità dell'algoritmo è $O(NM)$
L'algoritmo è stato implementato in due versioni; per la prima, che esegue l'algoritmo sopra scritto, abbiamo usato la libreria disponibile su GitHub al seguente link: github.com/pierre-rouanet/dtw. La seconda invece consiste in un'approssimazione dell'algoritmo, che ha un'esecuzione più rapida (ha una complessità di $O(N + M)$) ma può chiaramente portare a performance peggiori; per la libreria abbiamo usato la seguente, disponibile su GitHub: ithub.com/rtavenar/cydtw. In seguito ci riferiremo a questa versione con il nome di *Fast*-DTW (FDTW).

Algorithm 1: DTW

Data: $s_1 : [1 \dots n], s_2 : [1 \dots m]$
Result: distanza minima fra le due sequenze
DTW $:= [0 \dots n, 0 \dots m]$
for *i := 0 to n* **do**
  DTW$[i, 0] := \infty$
end
for *j := 0 to m* **do**
  DTW$[0, j] := \infty$
end
DTW$[0, 0] = 0$
for *i := 1 to n* **do**
  for *j := 1 to m* **do**
    cost $:= d(s_1[i], s_2[j])$
    DTW$[i, j] =$ cost $+$
    $\min($DTW$[i-1, j]$,
    DTW$[i, j-1]$, DTW$[i-1, j-1])$
  end
end
return DTW$[n, m]$

2.3 LSTM

Le LSTM (*Long Short Term Memory*) sono un'architettura di RNN (*Recurrent Neural Network*). Le RNN sono una classe di reti neurali in cui l'output ad un certo istante t è passato anche in input alla rete nell'istante successivo $t + 1$. Uno degli strati al suo interno è utilizzato come memoria, e questo permette alla rete di riconoscere sequenze di valori, utilizzando sia il valore di input all'istante t sia lo stato interno (che è ottenuto dalla sequenza di input fino all'istante $t - 1$).

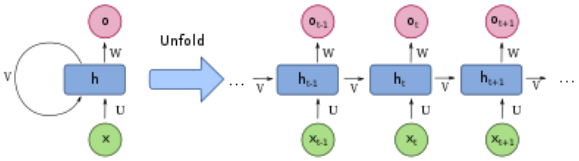

Figura 2.1: Una RNN "srotolata"

Il problema delle RNN tuttavia è che, utilizzando la *back-propagation* per effettuare l'aggiornamento dei pesi, i gradienti possono svanire fino a raggiungere 0 o esplodere. Nel primo caso, possiamo immaginarci che la RNN si scordi le informazioni che ha visto "troppo tempo fa".

Per risolvere il primo problema, le LSTM permettono ai gradienti di non essere modificati quando vengono passati in input alla rete nell'istante successivo. In particolare, la rete utilizza uno strato per decidere quanto "ricordare" una determinata informazione, moltiplicando il valore per un numero compreso fra 0 e 1 (si noti in figura la moltiplicazione in alto a sinistra nella cella).

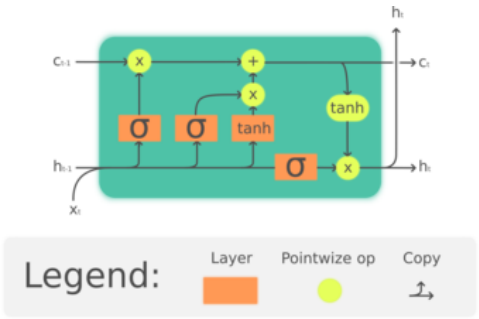

Figura 2.2: Architettura di una LSTM

Come possiamo vedere dalla figura, una cella LSTM in realtà è formata da più strati diversi, 3 con attivazione a sigmoide e 1 con attivazione a tangente iperbolica. In particolare:

- lo strato più a sinistra è detto "*forget gate*", e l'output regola la modifica del gradiente precedente;
- i due strati centrali sono detti "*input gate*", quello attivato dalla sigmoide decide quali valori andranno aggiunti alla memoria della rete, mentre quello attivato dalla tangente iperbolica dà in output i nuovi valori che potrebbero essere aggiunti. La moltiplicazione *point-wise* successiva darà il vettore di valori che saranno aggiunti alla memoria;
- lo strato più a destra è detto "*output gate*", e modifica l'output finale della rete.

2.3.1 Architettura del Modello

Il modello di *deep learning* implementato è formato da 5 strati, di cui 2 LSTM (fig. 2.3):

1. un primo strato `Masking`;
2. due strati `LSTM`;
3. uno strato `Dropout`;
4. uno strato finale `Dense`.

Masking

Lo strato `Masking` ha come obiettivo quello di eliminare il padding, in modo da rendere meno costoso il learning.

LSTM

I due strati `LSTM` sono consecutivi, il primo ha l'opzione `return_sequences` attivata, in modo da dare in output allo strato successivo tutti i vettori della sequenza, e non solo l'ultimo. Inizialmente utilizzavamo soltanto uno strato `LSTM`, tuttavia abbiamo osservato che questa versione con due strati offre performance migliori.; inoltre, l'aggiunta del secondo strato, ci ha permesso anche di poter avere uno strato che restituisse in output tutta la sequenza elaborata, in modo da poterla analizzare in un altro strato `LSTM`.

Dropout

Lo strato `Dropout` modifica alcuni input in modo casuale settandoli a 0, per tentare di prevenire l'*overfitting*. La probabilità di impostare un input a 0 è 0.5.

Dense

Lo strato `Dense` ha come scopo quello di restituire la classificazione finale, usando come funzione di attivazione la sigmoide.

```python
def create_model():
    model = Sequential()
    model.add(Masking(mask_value=0.0))
    model.add(LSTM(input_shape=(1000, 7), units=64, activation="sigmoid", return_sequences=True,
                   recurrent_activation="hard_sigmoid"))
    model.add(LSTM(units=128, activation="sigmoid", return_sequences=False, recurrent_activation="hard_sigmoid"))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
    return model
```

Figura 2.3: Definizione del modello

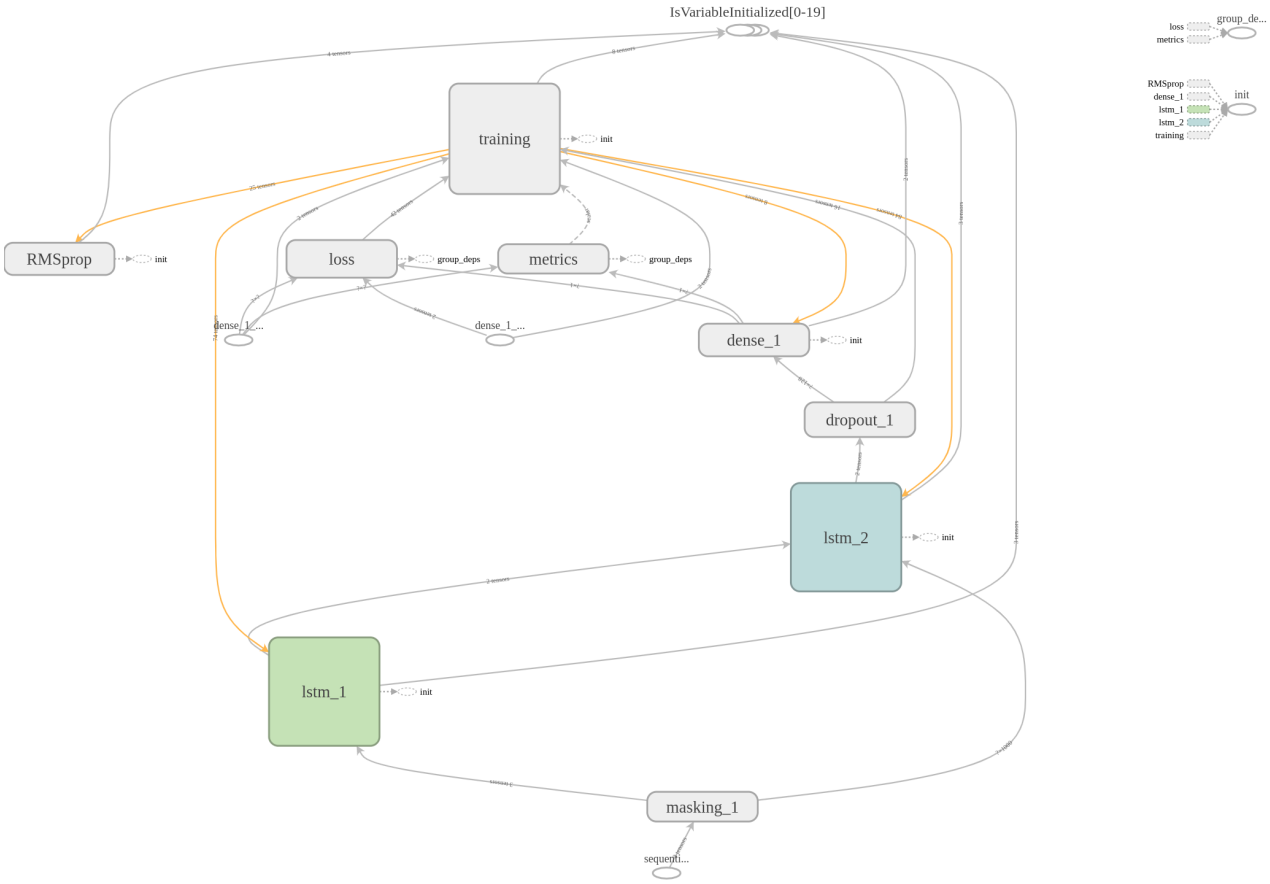

Figura 2.4: Modello durante la fase di training, rappresentazione da TensorFlow

Capitolo 3

Training e Test

Il dataset raccolto contiene parole scritte da 23 persone diverse, per un totale di 845 sample. La parola scelta da far scrivere agli utenti è "Computer", in corsivo.
Sia il training che i vari test sono stati eseguiti su un computer fisico con le seguenti specifiche:

- CPU: Intel Core i7 8700K;
- GPU: NVIDIA GeForce GTX 1070;
- RAM: 32 GB @ 3200 MHz.

Docker

Al fine di poter sfruttare la potenza calcolatrice della GPU abbiamo utilizzato il sistema Docker. Docker è un sistema di virtualizzazione che permette di creare delle macchine virtuali, detti "*container*", che condividono il kernel (in modo da offrire prestazioni migliori rispetto ad una semplice macchina virtuale). Quel che abbiamo fatto è stato creare un container al cui interno viene eseguito un Jupyter Notebook. Inoltre, ci ha permesso anche di facilitare il deploy, rendendolo possibile su qualsiasi macchina (poiché i Jupyter Notebook possono essere eseguiti in remoto) e fornendo anche maggior protezione da eventuali attacchi informatici (in quanto il programma viene eseguito all'interno di una macchina virtuale).

3.1 Training

È stato effettuato un training apposito per riconoscere alcuni degli utenti che sono stati registrati, generandone i relativi modelli.
Ogni training ha previsto 110 *epochs*, con un *batch* di 32 elementi; inoltre è stata effettuata, in ogni *epoch*, una fase di *validation* usando un terzo delle parole presenti nell'insieme di training.
Chiaramente, il dataset risulta ancora abbastanza contenuto, quindi i dati che saranno presentati nella successiva sezione non sono indicativi delle reali performance del sistema, per adesso. Nonostante ciò, ha senso analizzarli per avere una prima impressione dei modelli sviluppati.

3.2 Test

3.2.1 DTW

Per quanto riguarda il DTW abbiamo potuto effettuare un numero limitato di test, soprattutto per problemi di tempo. Poiché Python non supporta esecuzioni in multithread, il programma è stato eseguito su un singolo processore, rendendolo purtroppo più lento.
Nonostante ciò, abbiamo ottenuto alcuni risultati abbastanza promettenti:

```
AcX:
giovanni_1 vs dario_1 0.004921937874999999
giovanni_2 vs dario_2 0.004878329624999995
giovanni_1 vs giovanni_2 0.002709991375000001
dario_1 vs dario_2 0.00308468637499998
AcY:
giovanni_1 vs dario_1 0.003356265125000006
giovanni_2 vs dario_2 0.004203702
giovanni_1 vs giovanni_2 0.002556119125
dario_1 vs dario_2 0.003842376749999985
AcZ:
giovanni_1 vs dario_1 0.004122558499999999
giovanni_2 vs dario_2 0.004450192624999997
giovanni_1 vs giovanni_2 0.003453706625000022
dario_1 vs dario_2 0.00393582137499996
Pres:
giovanni_1 vs dario_1 0.00906591796875
giovanni_2 vs dario_2 0.010311035156251
giovanni_1 vs giovanni_2 0.00509619140625
dario_1 vs dario_2 0.00590185546875
GyX:
giovanni_1 vs dario_1 0.0020400611081899998
giovanni_2 vs dario_2 0.002146076259999995
giovanni_1 vs giovanni_2 2.9923656000000104e-05
dario_1 vs dario_2 3.651915299999858e-05
GyY:
giovanni_1 vs dario_1 0.003132091607999998
giovanni_2 vs dario_2 0.003351045822999999
giovanni_1 vs giovanni_2 0.003099580134999997
dario_1 vs dario_2 0.002572809150000012
GyZ:
giovanni_1 vs dario_1 0.003444519099999992
giovanni_2 vs dario_2 0.004467610739999999
giovanni_1 vs giovanni_2 0.002949771015000024
dario_1 vs dario_2 0.003298564887999999

Out[9]: [0.030083351285749998,
         0.033807992265251,
         0.019895283337250005,
         0.022672633163749994]
```

Figura 3.1: Risultati del test del DTW su due coppie di parole di due utenti registrati

Per ogni test del DTW (questo è solo uno di quelli che abbiamo fatto) consideriamo due parole per ogni utente e le confrontiamo con altre due parole di un altro utente. Per ogni componente (accelerometro, pressione, giroscopio) stampiamo in output le distanze per ogni combinazione fra le quattro parole; infine, stampiamo un array che contiene la somma delle rispettive distanze per ognuna delle quattro coppie di parole.
Idealmente, le coppie composte da parole di due utenti diversi dovrebbero avere una somma delle distanze maggiore rispetto alla somma di distanze di coppie di parole dello stesso utente. Ciò che abbiamo potuto osservare, limitatamente ai pochi test che abbiamo effettuato, è che quest'assunzione sembra essere vera in tutti i casi.
In un secondo tipo di test, che abbiamo potuto effettuare un'unica volta in quanto troppo lento (l'esecuzione è durata fra le 6 e le 7 ore), abbiamo selezionato casualmente una parola dal dataset ed abbiamo calcolato la distanza d (come media delle distanze di ognuna delle 7 componenti) di tutte le parole del dataset da essa. Il test ha avuto un esito molto positivo, generando una curva ROC la cui AUC è molto vicina a 1:

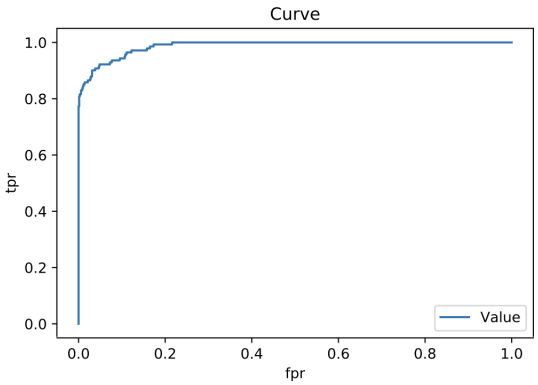

Figura 3.2: Curva ROC ottenuta considerando la "vicinanza" $(1 - d)$; AUC $\cong 0.99$

Visto che le feature sono state scalate fra 0 e 1, la distanza massima è proprio 1, e di conseguenza la vicinanza è data come la differenza fra 1 e la distanza calcolata.

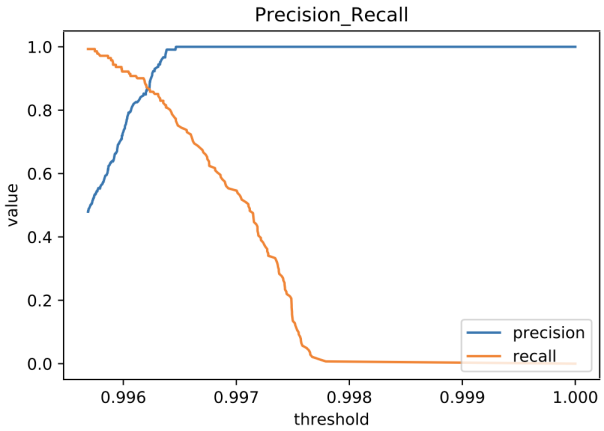

Figura 3.3: Curve di *precision* e *recall* al variare del threshold

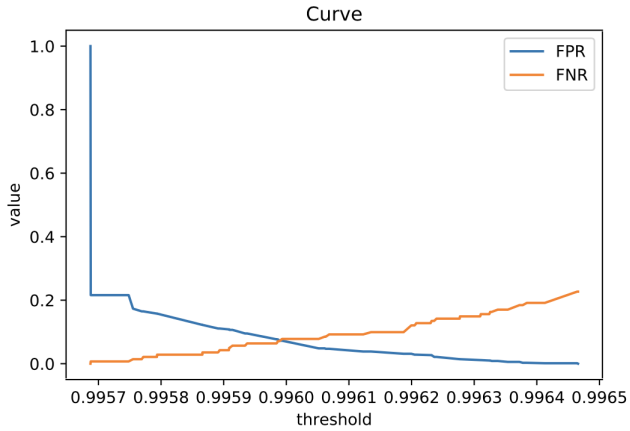

Figura 3.4: Curve di FPR e FNR al variare del threshold, si noti che l'EER è meno di 0.1, con il relativo threshold di poco inferiore a 0.996

FDTW

Con il *Fast*-DTW abbiamo potuto effettuare molti più test.
La nostra ipotesi che avesse performance peggiori del DTW è stata confermata, tuttavia abbiamo osservato che questo fatto è dipeso in gran parte dalla qualità dei dati, ossia da quanto rumore ci fosse al loro interno. In pratica, l'FDTW accentua le problematiche del DTW, ossia che non può "correggere" dati errati. Se un utente scrive in modo molto diverso la stessa parola, questi algoritmi che ne analizzano i segnali chiaramente avranno performance peggiori, mentre un algoritmo di deep learning può riuscire a comprendere che la persona è sempre la stessa, nonostante i dati siano piuttosto diversi fra di loro. Per la "vicinanza" fra due parole s_1, s_2 abbiamo usato l'equazione $\frac{1}{1+d(s_1,s_2)}$ in questo caso.

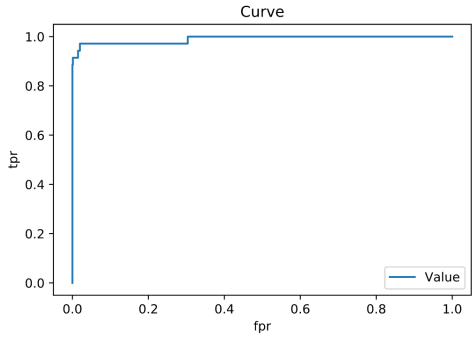

Figura 3.5: Curva ROC di uno dei casi in cui l'FDTW ha ottenuto performance migliori, AUC ≅???

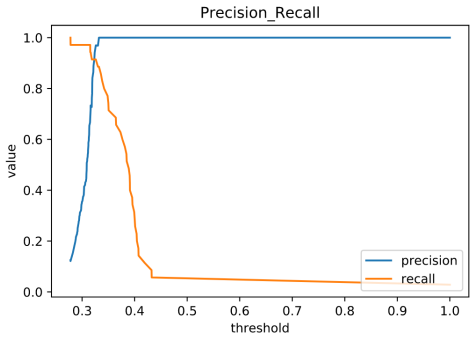

Figura 3.6: Curve di *precision* e *threshold* per lo stesso utente

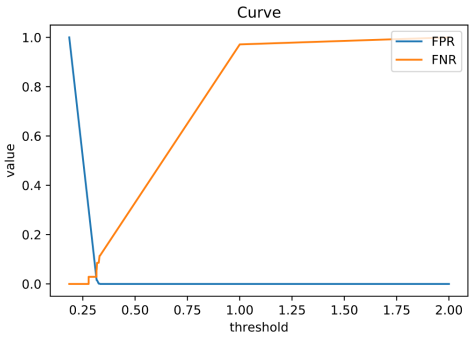

Figura 3.7: Curve di FPR e FNR dello stesso utente, l'EER è meno di 0.01 ed il relativo threshold è circa 0.3

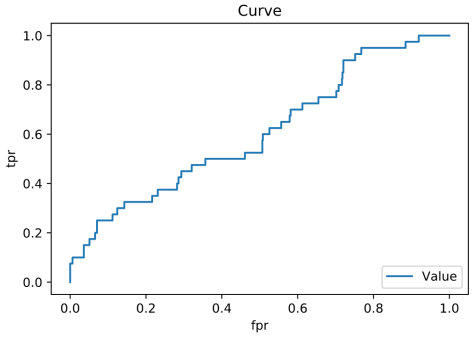

Figura 3.8: Curva ROC di uno dei casi in cui l'FDTW ha ottenuto performance peggiori, AUC ≅???

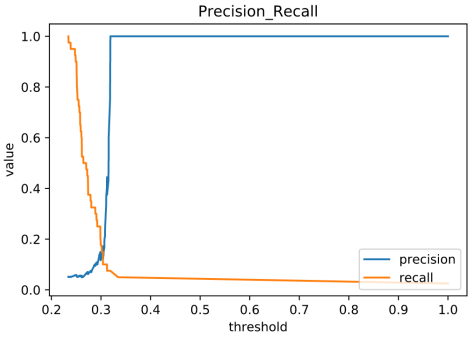

Figura 3.9: Curve di *precision* e *threshold* per lo stesso utente

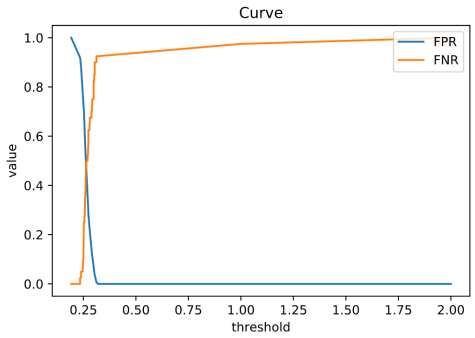

Figura 3.10: Curve di FPR e FNR dello stesso utente, l'EER è poco meno di 0.5[!!!] ed il relativo threshold è circa 0.3

3.2.2 LSTM

Il modello basato sulle LSTM ha raggiunto ottime prestazioni in poche epoch di training, tenendo comunque in considerazione che il dataset è relativamente piccolo. Includiamo solo uno dei modelli che abbiamo allenato ed i relativi test effettuati, in quanto altri esempi risulterebbero ridondanti.

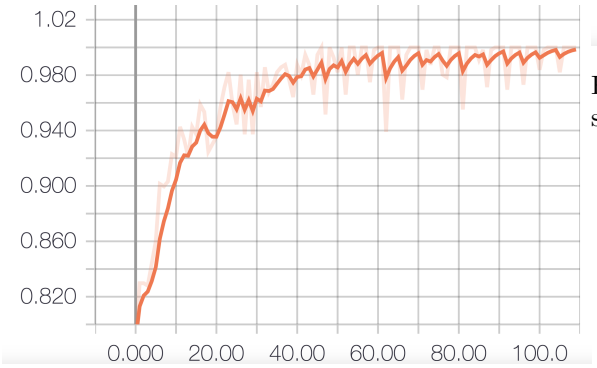

Figura 3.11: Accuracy del modello sul training set

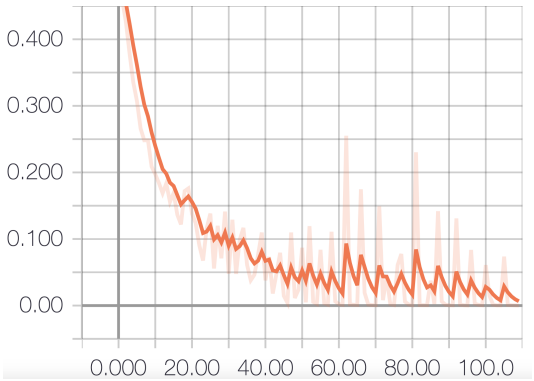

Figura 3.12: Loss del modello sul training set

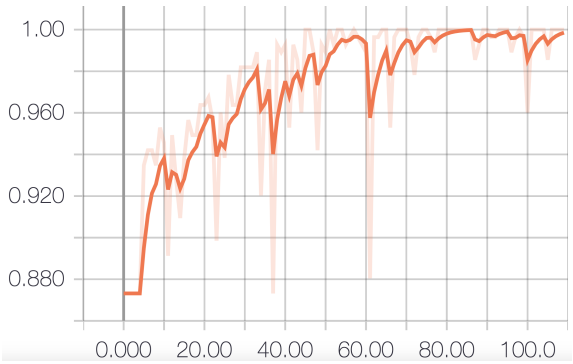

Figura 3.13: Accuracy del modello sul validation set

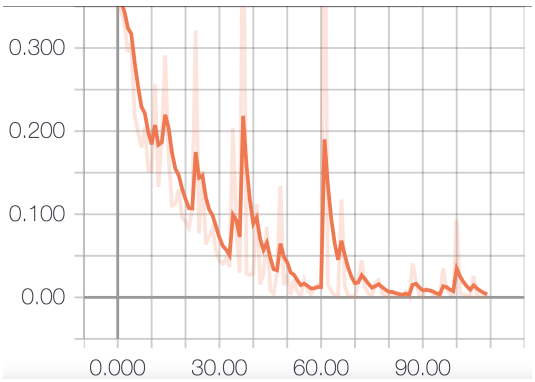

Figura 3.14: Loss del modello sul validation set

```
In [47]: with open(os.path.join('../dati/scrittura_di_computer/' ,'antonio_76_volte_pad.json'), 'r') as f:
             check = json.load(f)

         check = np.array(check)
         print(check.shape)

         model_antonio_reloaded.evaluate(check, np.ones(check.shape[0]))

         (76, 1000, 7)
         76/76 [==============================] - 2s 20ms/step
Out[47]: [0.06141953983981358, 0.9868421084002444]
```

Figura 3.15: Predizioni del modello su parole scritte in un altro momento rispetto a quelle usate nel training

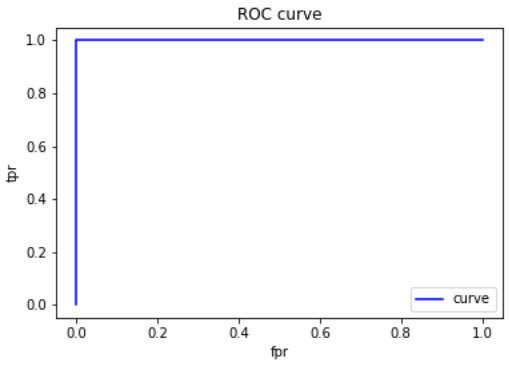

Figura 3.16: Curva ROC del modello allenato

Particolarmente interessante è la figura sulle predizioni, che in sostanza rappresenta il task principale della PeterPen, ossia riconoscere una persona dai dati che si raccolgono mentre scrive, sfruttando un modello allenato su altri dati dello stesso utente ma ottenuti in un momento precedente. Dai test che abbiamo effettuato sembra che il modello riesca effettivamente a riconoscere l'utente, con un accuracy di quasi il 99% (fig. 3.15).
Il modello inoltre non sembra essere afflitto da problemi di *overfit* o *underfit*, in quanto raggiunge accuracy molto simili nelle ultime epoche per il training set (fig. 3.11) ed il validation set (fig. 3.13), oltre al 99% di accuracy sul nuovo set di parole come scritto sopra.

	precision	recall	f1-score	support
rejection	1.00	1.00	1.00	704
acceptance	1.00	0.99	1.00	141
micro avg	1.00	1.00	1.00	845
macro avg	1.00	1.00	1.00	845
weighted avg	1.00	1.00	1.00	845

Figura 3.17: Report delle statistiche del modello

Capitolo 4

Lavori futuri

Innanzitutto, vorremmo effettuare una stampa della scocca più precisa. Abbiamo incontrato varie difficoltà nel montare la penna, dovute principalmente alle imprecisioni della stampante.
Inoltre, andrebbe sviluppato anche un chip dedicato, in quanto la maggior parte dei pin sull'ESP8266 è inutilizzato.
Per quanto riguarda il lato software, ci sono due possibili sviluppi.
Dal lato DTW, si potrebbe provare a dare dei pesi alle singole feature nel calcolo del costo. Questa possibile modifica nasce dall'osservazione che, con molta probabilità, nel calcolo della distanza fra due parole scritte, ci sono alcune componenti dei dati che potrebbero essere più importanti di altre. Inoltre, vanno necessariamente fatti più test, per poter verificare in modo più attendibile le prestazioni.
Infine, abbiamo osservato che il modello di deep learning, allenato soltanto per riconoscere la parola "Computer" scritta da un solo utente, ha "erroneamente" riconosciuto anche dei "Ciao" scritti dallo stesso utente (fig. 4.1). Questo ci fa pensare che, con alte probabilità, il modello si potrebbe prestare anche al riconoscimento della scrittura dell'utente, allenandolo su varie parole diverse scritte dallo stesso.

Figura 4.1: Come si può osservare, il modello rifiuta (correttamente) gli impostori, mentre circa nel 10% dei casi riesce a riconoscere correttamente l'utente, nonostante non abbia mai visto quella sua parola

Bibliografia

[1] M. De Marsico, F. Ponzi, F. Scozzafava, and G. Tortora, "Biopen - fusing password choice and biometric interaction at presentation level," *Pattern Recognition Letters*, 04 2018.

[2] A. Karouni, B. Daya, and S. Bahlak, "Offline signature recognition using neural networks approach," *Procedia Computer Science*, vol. 3, pp. 155 – 161, 2011, world Conference on Information Technology.

[3] M. Sharif, M. Khan, M. Faisal, M. Yasmin, and S. Lawrence Fernandes, "A framework for offline signature verification system: Best features selection approach," *Pattern Recognition Letters*, 02 2018.

[4] V. L. F. Souza, A. L. I. Oliveira, and R. Sabourin, "A writer-independent approach for offline signature verification using deep convolutional neural networks features," *CoRR*, vol. abs/1807.10755, 2018.