Università Politecnica delle Marche

FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Analisi sperimentale di reti neurali profonde per l'identificazione di volti senza vincoli

Experimental analysis of deep neural networks for unconstrained face identification

Relatore:
Prof. Aldo Franco Dragoni

Candidato:
Gianmarco Pigini

Correlatore:
Dott. Paolo Sernani

Anno Accademico 2019/2020

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Lo scopo di questa tesi è confrontare, in termini di accuratezza e tempi di esecuzione, le prestazioni di reti neurali pre-addestrate al riconoscimento facciale. Per riconoscimento facciale si intende il problema di risalire all'identità di persone in un'immagine o in un video partendo dai volti.
Il riconoscimento facciale può essere eseguito in due modi:

- verifica (o autenticazione)
- identificazione (o riconoscimento)

Per verifica si intende un confronto tra un volto non noto e un volto di cui è conosciuta l'identità, con l'obiettivo di capire se appartengano o meno alla stessa persona.
L'identificazione consiste invece in un confronto di un volto non noto con più immagini, contenute in un database, per risalire all'identità del volto sconosciuto.[3]
Un sistema di riconoscimento facciale si basa in genere su quattro moduli, rappresentati in figura 1.1: detection, alignment, feature extraction e matching.
Nella fase di *detection* vengono identificati e isolati i volti all'interno di un'immagine.
Il *face alignment* è mirato a identificare le componenti principali dei volti, come naso, occhi, bocca e contorno del viso. Il volto viene quindi normalizzato tramite trasformazioni geometriche, rispettando proprietà fotometriche come l'illuminazione e la scala di grigi.
In seguito viene eseguita la *feature extraction* per ottenere un vettore contenente le informazioni effettivamente utili per distinguere i volti di persone differenti.

Infine, tramite il *matching*, il vettore estratto dal volto di input è confrontato con i volti contenuti in un database. Si riceve quindi in output l'identità del volto quando si ottiene una corrispondenza con buona confidenza, altrimenti il volto viene indicato come non riconosciuto [3].

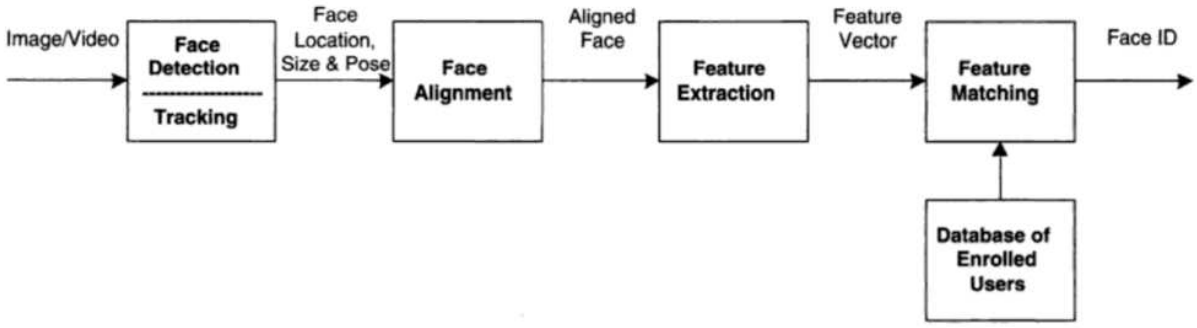


Figura 1.1: Processo di riconoscimento facciale [3]

All'interno di questa tesi è stato preso in considerazione il problema dell'identificazione. In particolare, per approcciarmi a tale problema, ho utilizzato versioni pre-addestrate delle seguenti reti neurali:

- FaceNet[4]
- VGGFace2[5]
- OpenFace[6]
- InsightFace [7]

le cui strutture sono spiegate nel capitolo 2.
In seguito, nel capitolo 3, sono elencati i test effettuati per ciascuna rete con i relativi risultati ottenuti in termini di accuratezza nel riconoscimento e di tempi di esecuzione.
Infine, nel capitolo 4, sono riportate le conclusioni derivate dal confronto delle reti.

Capitolo 2

Stato dell'Arte

Il machine learning è una delle tecnologie più interessanti dei giorni d'oggi. A partire dal riconoscimento facciale fino ad arrivare a macchine con guida autonoma, questa tecnologia sta avendo un ruolo sempre più impattante nella vita di tutti i giorni [1]. Infatti, con l'aumentare dei dati disponibili e delle capacità computazionali per processarli, è sempre più diffusa la tendenza ad avere sistemi che apprendono da soli, piuttosto che programmati per un problema specifico.
Kevin P. Murphy [2] definisce il machine learning come un insieme di metodi che permettono di identificare automaticamente un modello in un insieme di dati, per poi usare tale modello per prevedere dati futuri o per prendere altri tipi di decisioni in condizioni di incertezza. Esistono due possibili tecniche di machine learning:

- apprendimento supervisionato (supervised learning) o predittivo: l'obiettivo è determinare un modello partendo da dei dati di addestramento (training), con i quali poter fare previsioni su dati non disponibili o futuri.

- apprendimento non supervisionato (unsupervised learning) o descrittivo: l'obiettivo è determinare modelli o corrispondenze interessanti nei dati di input. Questo approccio viene chiamato anche *ricerca di conoscenza* (knowledge discovery).

Nel campo del machine learning stanno assumendo sempre più importanza le reti neurali artificiali.

2.1 Reti Neurali Artificiali

Che cos'è una rete neurale? Per iniziare, è necessario definire un tipo di neurone artificiale chiamato *Percettrone*. Tale neurone artificiale venne sviluppato tra gli anni 1950 e 1960 da Frank Rosenblatt, ispirato dai precedenti lavori di Warren McCulloch e Walter Pitts [8]. Un percettrone riceve input binari $x_1, x_2, ..., x_n$ e produce un singolo output binario.

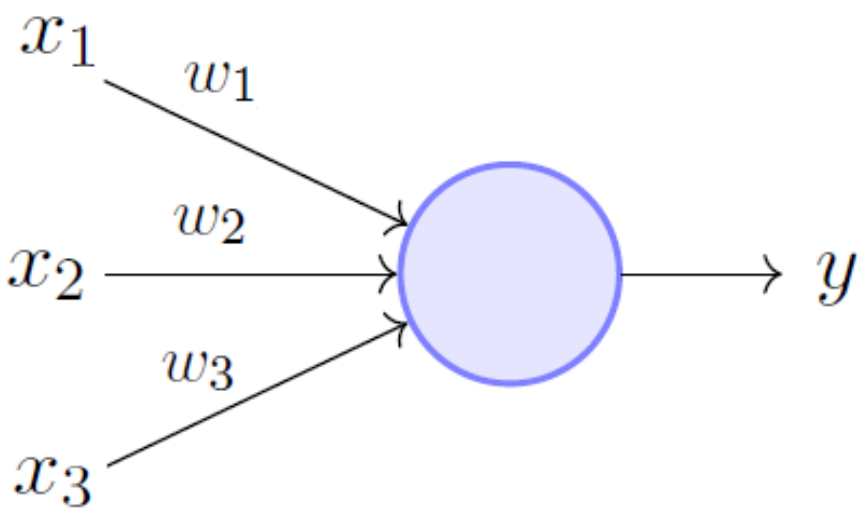


Figura 2.1: Struttura di un percettrone [8]

Per produrre l'output il percettrone utilizza dei pesi $w_1, w_2, ..., w_n$, numeri reali che rappresentano l'importanza di ciascun input per l'output del neurone.
L'output del neurone, 1 o 0, viene determinato dalla somma dei prodotti di ogni input per il relativo peso in base ad una soglia. Se tale somma è maggiore della soglia, l'output ha valore 1, altrimenti ha valore 0.

$$output = \begin{cases} 0 \ se \ \sum_j \ w_j x_j \leq soglia \\ 1 \ se \ \sum_j \ w_j x_j > soglia \end{cases} \tag{2.1}$$

Variando i pesi e le soglie si ottengono diversi tipi di modelli decisionali. Ovviamente un singolo percettrone non basta a simulare un modello decisionale umano, proprio per questo si usano reti neurali composte da più percettroni opportunamente collegati tra loro.

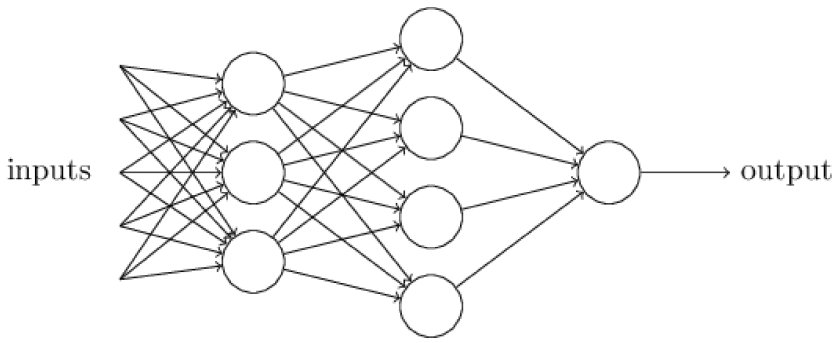


Figura 2.2: Rete di percettroni [8]: Gli output dei percettroni di un livello vengono utilizzati come input dei percettroni dei livelli successivi.

Il percettrone presenta però un problema fondamentale, quello di avere un output binario, per cui un singolo cambiamento di un peso o di una soglia può cambiare completamente il valore dell'output. Per tale motivo si è iniziato ad utilizzare un altro tipo di neurone artificiale, chiamato *sigmoide*.

Il neurone sigmoide ha una struttura simile a quella del percettrone, ma con differenze tali da risolvere il problema precedentemente introdotto. Proprio come un percettrone, infatti, un neurone sigmoide riceve in input $x_1, x_2, ..., x_n$ segnali che possono però assumere valori compresi tra 0 e 1. Di conseguenza anche l'output del neurone sarà compreso tra 0 e 1.

Infatti, tale output è determinato tramite la funzione sigmoide (da cui il neurone prende il nome) definita nel seguente modo:

$$\frac{1}{1 + exp(-\sum_j \ w_j x_j - b)}$$

dove per b si intende la soglia (bias).

Quindi, riassumendo, una rete neurale artificiale è un sistema di elaborazione dell'informazione, il cui funzionamento trae ispirazione dai sistemi biologici nervosi [9].

Una rete neurale possiede molte unità di elaborazione connesse tra di loro, ognuna delle quali prende il nome di nodo.

Come mostrato in fig. 2.3, i nodi possono essere di tre tipi:

- input
- output
- hidden(nascosti)

I nodi hidden possono essere strutturati in più livelli, determinando una maggiore profondità della rete. Ogni nodo simula il ruolo di un neurone all'interno delle reti neurali biologiche. Il compito di ciascun nodo all'interno di una rete neurale consiste nell'attivarsi se la quantità di segnale ricevuta in input supera la propria soglia di attivazione. Se un'unità diventa attiva, essa emette un segnale che viene trasmesso ai nodi cui è connessa.

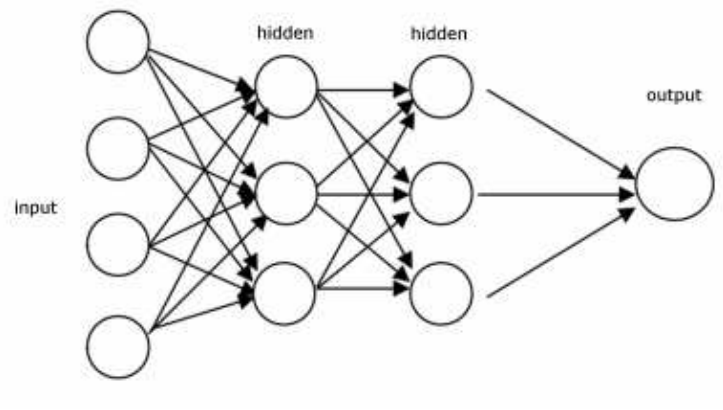


Figura 2.3: Struttura rete neurale

2.2 Apprendimento di una rete neurale

Avendo introdotto la struttura di una rete neurale, è fondamentale capire come funzioni il processo di apprendimento di una rete.
Come già visto in precedenza, l'apprendimento può essere supervisionato o non supervisionato. Di particolare interesse per il problema preso in questione è l'apprendimento supervisionato.
Il suo funzionamento si basa sul fornire alla rete delle coppie (x,y) di informazioni note appartenenti ad un dataset detto di *training* [9].

Tali coppie sono formate da x_i input da fornire alla rete e dai corrispondenti y_i output noti. Il compito della rete è apprendere quale legame ci sia tra input e output, in modo da poter poi determinare legami ingresso-uscita per input non appartenenti al dataset di training.
Questo tipo di apprendimento è detto supervisionato in quanto, per ogni output generato, si procede a correggere la rete per modificare i pesi e migliorare la risposta. Viene infatti utilizzato un algoritmo di Error-Back-propagation che valuta l'errore commesso dalla rete, cioè la differenza tra output ottenuto e output desiderato. Si hanno quindi più iterazioni, chiamate *epoche*, durante la fase di training, almeno fino a quando l'errore non scende sotto una soglia di accettabilità. Ogni epoca prevede una modifica dei pesi interni alla rete, comportando una riduzione dell'errore ad ogni iterazione.
Una volta terminata la fase di training si può passare alla fase successiva di test, in cui si utilizza la rete addestrata su nuovi input.

2.3 Reti Neurali utilizzate

All' interno di questa tesi sono state utilizzate quattro reti neurali pre-addestrate, quindi già sottoposte alla fase di training e pronte per essere usate per il testing.

2.3.1 FaceNet

FaceNet è un sistema di riconoscimento facciale sviluppato nel 2015 dai ricercatori di Google. Tale sistema, ricevuta in input l'immagine di un volto, estrae le caratteristiche fondamentali ad alta qualità del volto e predice un vettore di 128 elementi che rappresentano tali caratteristiche, chiamato face embedding [4]. Il vettore di face embedding incorpora un'immagine x in uno spazio Euclideo *d*-dimensionale.
Il modello consiste in una rete neurale convoluzionale addestrata tramite una funzione *Triplet Loss*. L'obiettivo di tale funzione è garantire che un'immagine x_i^a (anchor) di un'identità specifica sia più vicina alle altre immagini x_i^p (positive) della stessa identità rispetto a tutte le altre x_i^n (negative) immagini appartenenti ad altre identità (fig. 2.4) [11].

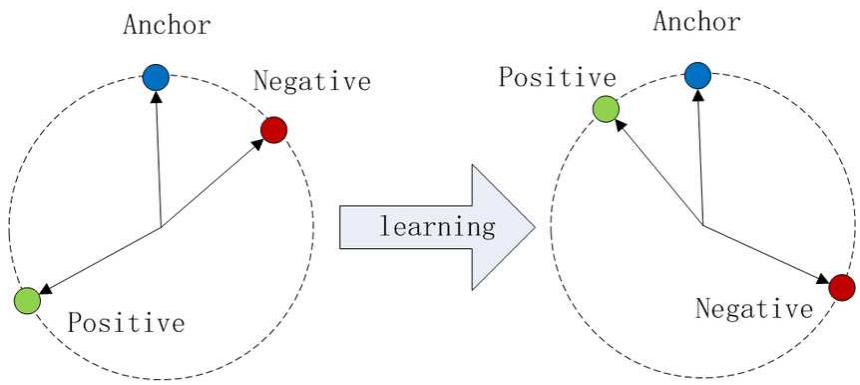


Figura 2.4: Triplet Loss [11]

In pratica ciò che si vuole ottenere è:

$$||x_i^a - x_i^p||_2^2 + \alpha < ||x_i^a - x_i^n||_2^2, \forall (x_i^a, x_i^p, x_i^n) \in T$$

dove α è un margine introdotto tra le coppie positive e negative, T è l'insieme di tutte le possibili triplette ed ha cardinalità N.
Di conseguenza, la perdita che si vuole minimizzare è:

$$L = \sum_i^N [||f(x_i^a) - f(x_i^p)||_2^2 - ||f(x_i^a) - f(x_i^n)||_2^2 + \alpha]$$

Per addestrare il modello sono state utilizzate tra i 100 e 200 milioni di immagini appartenenti a 8 milioni di identità. Il modello è stato poi valutato sul dataset LFW [10] dove ha ottenuto, usando l'allineamento dei volti, un'accuratezza di classificazione del 99.63%.

2.3.2 VGGFace2

Per VGG si intendono una serie di dataset per il riconoscimento facciale sviluppati da dei membri del Visual Geomety Group (VGG) dell'università di Oxford [5]. Esistono due principali modelli VGG che sono VGGFace e VGGFace2.
Gli sviluppatori del modello descrivono VGGFace2 come un dataset realizzato con l'intento di addestrare e valutare modelli per il riconoscimento facciale. Spiegano infatti come il dataset contenga 3.31 milioni di immagini di 9131 soggetti, con una media di 362.6 immagini per persona. Tali immagini presentano soggetti con differenze in posa, età, illuminazione, etnia e professione [12].

Nonostante l'intento fosse quello di creare un dataset di grosse dimensioni, VGGFace2 è diventato il nome di riferimento per i modelli pre-addestrati che lo hanno utilizzato per la fase di training.
In particolare, in questa tesi è stato utilizzato come modello ResNet50 [13].
ResNet50 appartiene alla famiglia delle reti neurali residuali. Tali reti si basano sui blocchi residuali che eseguono su un input x operazioni di convoluzione e attivazione. Il risultato è un output F(x) a cui viene sommato l'input x (fig. 2.5). La rete ResNet50 usa due blocchi residuali. Questa rete è stata valutata su dataset standard per il riconoscimento facciale, raggiungendo le performance delle migliori reti nell'ambito del riconoscimento facciale.

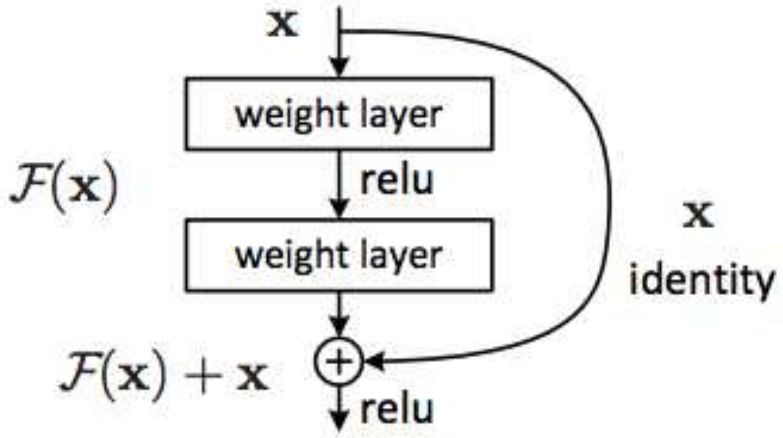


Figura 2.5: Blocchi residuali [13]

2.3.3 OpenFace

OpenFace è una rete neurale sviluppata a partire da FaceNet. A differenza di FaceNet tale rete prevede un'azione di preprocessamento dei volti ricevuti in input, in modo tale che ogni immagine presenti le caratteristiche principali (occhi, bocca, naso, contorno viso) in posizioni simili [14]. Per fare ciò è necessario un algoritmo che ricerchi in ogni volto queste caratteristiche principali. L'algoritmo usato appartiene alla libreria *Dlib* e determina 68 punti di riferimento (landmarks) in ogni volto.
Una volta trovati i landmarks, si procede all'allinemento del volto in modo tale che tali punti di riferimento risultino centrati rispetto all'immagine di dimensione 96x96.

Openface è stata addestrata con oltre 500.000 immagini appartenenti a due dataset combinati, CASIA-WebFace [YLLL14] e FaceScrub [NW14].
Nonostante sia stata addestrata con dataset di dimensioni molto inferiori rispetto a FaceNet, Openface ha comunque ottenuto una buona accuratezza di riconoscimento sul dataset LFW, raggiungendo le performance delle migliori reti nell'ambito del riconoscimento facciale.
Inoltre tale rete presenta anche il vantaggio di avere ottime prestazioni in termini di tempo di esecuzione.

2.3.4 InsightFace

InsightFace è un modello basato su ArcFace [15]. Questa rete usa per l'addestramento una versione modificata di una delle funzioni più utilizzate, *SoftMax Loss*:

$$L_1 = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{W_{y_i}^T x_i + b_{y_i}}}{\sum_{j=1}^{n} e^{W_j^T x_i + b_j}}$$

dove gli x_i rappresentano le caratteristiche dell'esempio i, appartenente alla classe y.
I vettori di embeddings d generati da tale rete hanno dimensione 512.
Come affermato però dagli autori del paper [15], la funzione *Softmax Loss* non ottimizza gli embeddings appartenenti alla stessa classe ad essere più simili e quelli di classi diverse ad essere meno simili. Per questo motivo è stata introdotta una normalizzazione e una scalatura a s degli embeddings.
La normalizzazione delle caratteristiche degli embeddings e dei pesi fa sì che la predizione dipenda solo dall'angolo formato tra di essi.
Le caratteristiche apprese degli embeddings sono poi distribuite su una ipersfera di raggio *s*.

$$L_2 = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\cos\theta_{y_i}}}{e^{s\cos\theta_{y_i}} + \sum_{j=1, j\neq y_i}^{n} e^{s\cos\theta_j}}$$

É stato quindi aggiunto un margine angolare additivo *m* per migliorare la compattezza tra gli embeddings della stessa classe e la discrepanza tra gli embeddings di classi diverse.

$$L_3 = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s(\cos(\theta_{y_i}+m))}}{e^{s(\cos(\theta_{y_i}+m))} + \sum_{j=1, j\neq y_i}^{n} e^{s\cos\theta_j}}$$

Questo metodo è stato chiamato ArcFace in quanto il margine m introdotto è uguale al margine di distanza geodetica di un'ipersfera normalizzata.
Come mostrato in fig. 2.6, la funzione ArcFace aumenta il gap tra classi diverse vicine [15].
La rete InsightFace, una volta addestrata, ha ottenuto accuratezze pari al 99.83% su LFW.

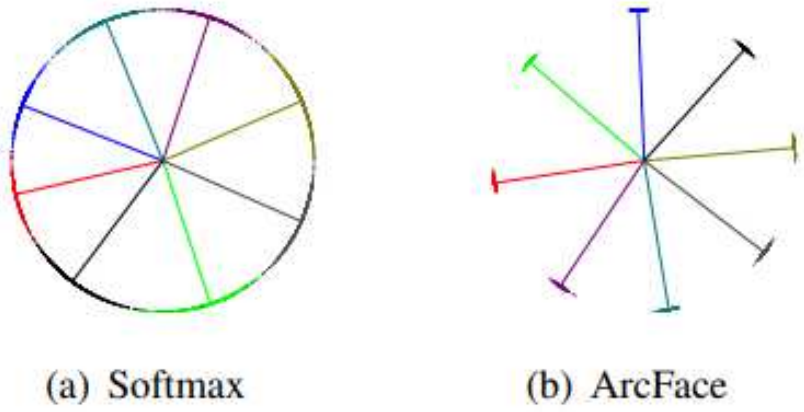


Figura 2.6: Differenze SoftMax e ArcFace [15]

Capitolo 3

Lavoro svolto e risultati

In questo capitolo sono riportati e analizzati i test effettuati.
Nella sezione 3.1 sono brevemente elencati gli strumenti softare utilizzati. Nella sezione 3.2 è spiegato il codice scritto in Python per ogni rete. Infine, nella sezione 3.3, sono riportati i risultati ottenuti.

3.1 Strumenti utilizzati

3.1.1 Keras

Keras è una libreria opensource ad alto livello per l'apprendimento automatico e le reti neurali, scritta in Python e funziona da interfaccia a Tensorflow, Theano e CNTK.
Lo scopo di Keras è quello di sviluppare e sperimentare velocemente nell'ambito del machine learning e del deep learning.
I vantaggi principali forniti da Keras sono:

- Velocità: Keras basa la propria forza su semplicità e velocità di implementazione. Con poche righe di codice si possono sviluppare modelli complessi.
- Modularità: Permette di realizzare modelli come sequenze di moduli configurabili che possono essere assemblati insieme con poche restrizioni.

- Estensibilità: Permette di aggiungere facilmente nuovi moduli (come classi e funzioni). Inoltre fornisce ampi esempi per i moduli esistenti.

3.1.2 Tensorflow

TensorFlow è una libreria software open source per l'apprendimento automatico che fornisce moduli testati e ottimizzati utili alla realizzazione di algoritmi per diversi tipi di compiti percettivi e di comprensione del linguaggio.
Fu sviluppata dal team Google Brain e resa disponibile il 9 novembre 2015, nei termini della licenza open source Apache 2.0.
TensorFlow si configura come un framework per la definizione e il calcolo di operazioni che coinvolgono tensori, rappresentazioni generiche di vettori e matrici a dimensioni maggiori. Nel 2017 è stato introdotto il supporto per Keras: il modulo tf.keras è l'implementazione delle API Keras come libreria di alto livello per TensorFlow.

3.1.3 Google Colab

Google Colab è una piattaforma che permette di scrivere ed eseguire codice Python direttamente sul cloud con i seguenti vantaggi:

- Nessuna configurazione necessaria
- Accesso gratuito alle gpu
- Condivisione semplificata

Inoltre è facilmente accessibile tramite un account Google.

3.2 Lavoro svolto

Il lavoro di confronto delle prestazioni delle reti precedentemente introdotte ha richiesto uno step preliminare.
Infatti, prima di poter utilizzare le reti, ho dovuto preparare il database su cui valutarle. Per fare ciò sono partito da un database standard molto utilizzato nell'ambito del riconoscimento facciale, LFW [10], che contiene oltre 13000 immagini di 1680 personaggi famosi.
A questo punto sono andato a realizzare quattro database di dimensioni inferiori secondo il seguente criterio:

1. Nel primo database ho inserito tutte le identità del database di partenza contenenti almeno dieci immagini per persona. Per uniformare il database ho eliminato le foto in eccesso, ottenendo quindi un database di **113** identità con esattamente **10** foto per persona.

2. Nel secondo database ho inserito tutte le identità del database di partenza contenenti almeno venti immagini di persona. Per uniformare il database ho eliminato le foto in eccesso, ottenendo quindi un database di **46** identità con esattamente **20** foto per persona.

3. Nel terzo database ho inserito tutte le identità del database di partenza contenenti almeno trenta immagini per persona. Per uniformare il database ho eliminato le foto in eccesso, ottenendo quindi un database di **23** identità con esattamente **30** foto per persona.

4. Nel quarto database ho inserito tutte le identità del database di partenza contenenti almeno quaranta immagini per persona. Per uniformare il database ho eliminato le foto in eccesso, ottenendo quindi un database di **14** identità con esattamente **40** foto per persona.

Successivamente ho suddiviso ciascun database in 5 parti di uguali dimensioni, in modo tale da poter ottenere più valutazioni dell'accuratezza della rete tramite lo strumento della *validazione incrociata* (cross-validation).
La validazione incrociata consiste nella suddivisione dell'insieme dei dati di partenza in k parti uguali (in questo caso *k=5*). Ad ogni passo, viene utilizzata la k-esima parte dell'insieme dei

dati come test, mentre le restanti k-1 parti vengono utilizzate per la fase di training.

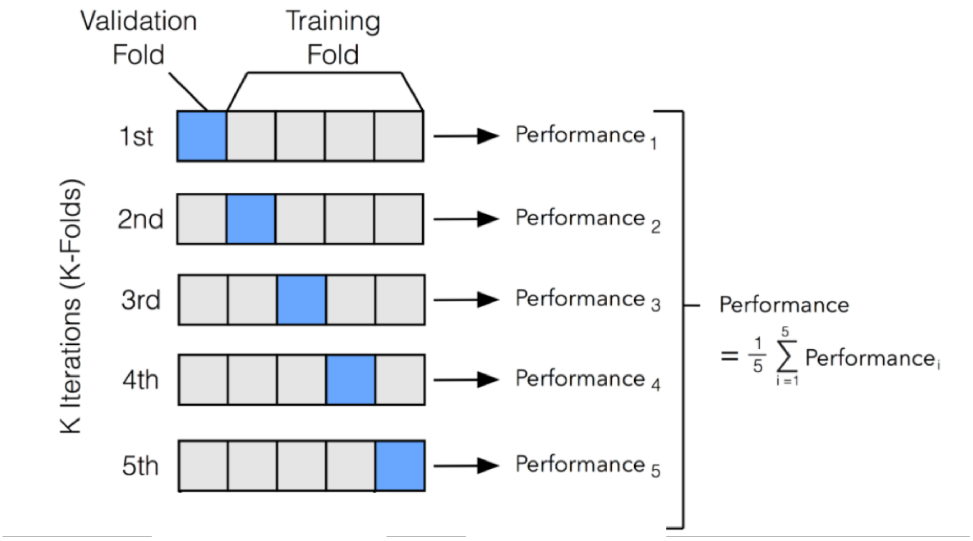


Figura 3.1: Validazione incrociata

In tal modo si allena il modello per ognuna delle k parti e si ottengono k valutazioni dell'accuratezza della rete sul database (fig. 3.1).
Terminata questa fase di preparazione, ho caricato i database ottenuti sul mio account di Google Drive in modo da potervi accedere tramite Google Colab.
A questo punto sono passato alla fase di scrittura del codice, in parte ispirato dagli esempi di Jason Brownlee, riportati nell'ebook *Machine Learning Mastery*[4][5].
Il codice realizzato è articolato in tre blocchi principali:

1. Identificazione volti: In questa parte sono utilizzati modelli per identificare volti. Tali modelli sono applicati ai database. I volti identificati vengono poi estratti ed opportunamente ridimensionati ed adattati in base alla rete utilizzata. Successivamente i volti estratti sono salvati su un file compresso. Questo processo è stato applicato a tutti i database di train e test utilizzati.

2. Generazione embeddings: In questa parte di codice i file compressi generati al punto 1 sono forniti in input alla rete neurale. La rete, elaborando gli input, genera i corrispet-

tivi *embeddings* (già introdotti nel capitolo 2), cioè vettori contenenti le caratteristiche estratte da ciascun volto. Questi vettori vengono a loro volta salvati su un file compresso.

3. Classificazione volti: In questo ultimo blocco sono caricati i file compressi creati nel punto 2, contenenti i vettori di embeddings. Questi vettori, dopo essere stati normalizzati, vengono utilizzati per addestrare un classificatore SVC. Il classificatore genera quindi una predizione per ogni esempio presente nei dataset di train e test. In seguito, tramite la funzione *accuracy_score*, si valuta l'accuratezza delle predizioni del classificatore. Tale accuratezza è il rapporto tra le identità esattamente predette rispetto alle identità totali.

Nelle sezioni seguenti sono analizzati nel dettaglio i test effettuati per ciascuna rete utilizzata.

3.2.1 FaceNet

Come anticipato nel capitolo 3.2, il codice realizzato è strutturato in tre blocchi. La parte relativa alla classificazione di volti sarà trattata a parte nella sezione 3.2.5.

Identificazione volti

Come modello per identificare volti è stato utilizzato MTCNN. Sono state definite tre funzioni: *estrai_faccia*, *carica_facce*, *carica_dataset*.

```
#estrae una faccia da una data fotografia
def estrai_faccia(nomefile, dimensione_richiesta=(160,160)):
   #carica l'immagine dal file
   immagine = Image.open(nomefile)
   #converte l'immagine in RGB
   immagine = immagine.convert('RGB')
   #converte l'immagine in un vettore
   pixels = asarray(immagine)
   #crea il modello di riconoscimento, usando pesi di default
   riconoscitore = MTCNN()
   #riconosce le facce dalle immagini
   risultato = riconoscitore.detect_faces(pixels)
   #estrae un quadrato contenente la faccia
   x1, y1, larghezza, altezza = risultato[0]['box']
   #correzione
   x1, y1 = abs(x1), abs(y1)
   x2, y2 = x1 + larghezza, y1 + altezza
   #estrae le facce
   faccia = pixels[y1:y2, x1:x2]
   #adatta la dimensione dei pixels a quella del modello
   immagine = Image.fromarray(faccia)
   immagine = immagine.resize(dimensione_richiesta)
   facearray = asarray(immagine)
   return facearray
```

Listing 3.1: *estrai_faccia*

La funzione *estrai_faccia* utilizza il modello MTCNN sull'immagine fornita in input.

I volti identificati vengono estratti e, dopo essere stati ridimensionati in base all'input richiesto da FaceNet (160x160), vengono convertiti in un vettore tramite la funzione *asarray*. Tale vettore è l'output restituito dalla funzione *estrai_faccia*.

```
#carica immagini ed estrae facce per ogni immagine della directory
def carica_facce(directory):
   facce = list()
   for gruppo in listdir(directory):
     path = directory + '/' + gruppo
     for filename in listdir(path):
       new_path = path + '/' + filename
     #ottiene faccia
       faccia = estrai_faccia(new_path)
      #carica faccia
       facce.append(faccia)
   return facce
```

Listing 3.2: *carica_facce*

La funzione *carica_facce* richiama la funzione *estrai_faccia* per tutte le immagini di uno stesso individuo, contenute nella directory fornita in input.

```
def carica_dataset(directory):
  X,y = list(), list()
  for nome_personaggio in listdir(directory):
    path = directory + '/' + nome_personaggio
    #salta i file che dovrebbero essere nella directory
    if not isdir(path):
      continue
    #carica tutte le facce nella sottodirectory
    facce = carica_facce(path)
    #crea etichette
    labels = [nome_personaggio for _ in range(len(facce))]
    #elenca i progressi
    print('caricati %d esempi per la classe: %s' % (len(facce), nome
   _personaggio))
     #memorizza
```

```
    X.extend(facce)
    y.extend(labels)
  return asarray(X), asarray(y)
```

Listing 3.3: *carica_dataset*

Infine la funzione *carica_dataset* riceve in input la directory di un database e richiama la funzione *carica_facce* per ogni sottodirectory contenuta nel database. Quindi in output restituisce due array: uno contenente i volti estratti e uno contenente i nomi (labels) delle identità interne al database.

Gli array ottenuti dalla funzione *carica_dataset* vengono salvati in un file compresso.

```
#carica il dataset di training
trainX, trainy = carica_dataset('directory-dataset-train')
#salva gli array in un file compresso
savez_compressed('filename.npz', trainX, trainy)
```

Listing 3.4: salvataggio file

Questo processo è stato applicato a tutti i database di train e test.

Generazione embeddings

In questa parte di codice vengono caricati i file compressi generati durante la fase di identificazione dei volti, contenenti i volti dei database di train e di test.

```
#carica i dataset di facce precedentemente trovate
data_train = load('filename-train.npz')
trainX, trainy = data_train['arr_0'], data_train['arr_1']
data_test = load('filename-test.npz')
testX, testy = data_test['arr_0'], data_test['arr_1']
```

Listing 3.5: loading dataset

Inoltre, tramite la funzione *load_model*, si carica il modello pre-addestrato di FaceNet, precedentemente salvato su Google Drive.

```
#si carica il modello FaceNet per prevedere l'identita' della faccia
model = load_model('/content/drive/My Drive/keras-facenet/model/
    facenetkeras.h5')
```

Listing 3.6: *load_model*

A questo punto è stata definita la funzione *ottieni_embedding*.

Tale funzione riceve in input un volto che, dopo essere stato adattato, viene utilizzato dal modello per fare una predizione e generare un embedding.

```
def ottieni_embedding(model, pixel_faccia):
   #scala i valori dei pixel
   pixel_faccia = pixel_faccia.astype('float32')
   #standardizza i valori dei pixel attraverso canali globali
   mean, std = pixel_faccia.mean(), pixel_faccia.std()
   pixel_faccia = (pixel_faccia - mean) / std
   #per fare la predizione, bisogna espandere la dimensione in modo
    tale che il face array sia un campione
   campioni = expand_dims(pixel_faccia, axis=0)
   #si usa il modello per fare una predizione ed estrarre i vettori di
    embedding
   yhat = model.predict(campioni)
   return yhat[0]
```

Listing 3.7: *ottieni_embedding*

Tramite un ciclo *for* si applica la funzione *ottieni_embedding* a tutti i volti contenuti nei file compressi precedentemente caricati, ottenendo quindi un vettore di embeddings di train e un vettore di embeddings di test.

```
#converte ogni faccia nel train set in un embedding
newTrainX = list()
for pixel_faccia in trainX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTrainX.append(embedding)
newTrainX = asarray(newTrainX)
```

```
#converte ogni faccia nel test set in un embedding
newTestX = list()
for pixel_faccia in testX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTestX.append(embedding)
newTestX = asarray(newTestX)
```

Listing 3.8: Creazione vettori di embeddings

Questi vettori vengono salvati in due diversi file compressi, insieme ai vettori contenenti i nomi delle identità del rispettivo database.

```
#salva gli array in un file in formato compresso
savez_compressed('embeddings_train.npz', newTrainX, trainy)
savez_compressed('embeddings_test.npz', newTestX, testy)
```

Listing 3.9: salvataggio file

Durante questa fase di generazione embeddings è stato misurato il tempo di esecuzione del programma, in modo da poter confrontare il tempo necessario alla rete per generare gli embeddings con il tempo necessario alle altre reti. Per poter valutare il tempo di esecuzione è stata utilizzata la funzione *time.monotonic*. Tale funzione restituisce (in frazioni di secondo) il valore di un orologio monotonico. Prendendo gli istanti di inizio e fine di esecuzione del programma si è poi misurato il tempo totale di esecuzione calcolando la differenza tra i due istanti. Il tempo ottenuto è stato poi stampato a schermo con la funzione *print*.
I risultati ottenuti sono riportati nel capitolo 3.3.

```
import time
start = time.monotonic()
time.sleep(0.1)
```

Listing 3.10: start time

```
end = time.monotonic()
print('tempo di esecuzione programma : -:9.2f '.format(end-start))
```

Listing 3.11: end time

3.2.2 VGGFace2

Come anticipato nel capitolo 3.2, il codice realizzato è strutturato in tre blocchi. La parte relativa alla classificazione di volti sarà trattata a parte nella sezione 3.2.5.

Identificazione volti

Anche in questo caso, come per FaceNet, è stato utilizzato MTCNN. Sono state definite tre funzioni: *estrai_faccia*, *carica_facce*, *carica_dataset*.

```
#estrae una faccia da una data fotografia
def estrai_faccia(nomefile, dimensione_richiesta=(224,224)):
   #carica l'immagine dal file
   immagine = Image.open(nomefile)
   #converte l'immagine in RGB
   immagine = immagine.convert('RGB')
   #converte l'immagine in un vettore
   pixels = asarray(immagine)
   #crea il modello di riconoscimento, usando pesi di default
   riconoscitore = MTCNN()
   #riconosce le facce dalle immagini
   risultato = riconoscitore.detect_faces(pixels)
   #estrae un quadrato contenente la faccia
   x1, y1, larghezza, altezza = risultato[0]['box']
   #correzione
   x1, y1 = abs(x1), abs(y1)
   x2, y2 = x1 + larghezza, y1 + altezza
   #estrae le facce
   faccia = pixels[y1:y2, x1:x2]
   #adatta la dimensione dei pixels a quella del modello
   immagine = Image.fromarray(faccia)
   immagine = immagine.resize(dimensione_richiesta)
   facearray = asarray(immagine)
   return facearray
```

Listing 3.12: *estrai_faccia*

La funzione *estrai_faccia* utilizza il modello MTCNN sull'immagine fornita in input. I volti identificati vengono estratti e, dopo essere stati ridimensionati in base all'input richiesto da VGGFace2 (224x224), vengono convertiti in un vettore tramite la funzione *asarray*. Tale vettore è l'output restituito dalla funzione *estrai_faccia*.

```
#carica immagini ed estrae facce per ogni immagine della directory
def carica_facce(directory):
  facce = list()
  for gruppo in listdir(directory):
    path = directory + '/' + gruppo
    for filename in listdir(path):
      new_path = path + '/' + filename
    #ottiene faccia
      faccia = estrai_faccia(new_path)
     #carica faccia
      facce.append(faccia)
  return facce
```

Listing 3.13: *carica_facce*

La funzione *carica_facce* richiama la funzione *estrai_faccia* per tutte le immagini di uno stesso individuo, contenute nella directory fornita in input.

```
def carica_dataset(directory):
 X,y = list(), list()
 for nome_personaggio in listdir(directory):
   path = directory + '/' + nome_personaggio
   #salta i file che dovrebbero essere nella directory
   if not isdir(path):
     continue
   #carica tutte le facce nella sottodirectory
   facce = carica_facce(path)
   #crea etichette
   labels = [nome_personaggio for _ in range(len(facce))]
   #elenca i progressi
   print('caricati %d esempi per la classe: %s' % (len(facce), nome
  _personaggio))
    #memorizza
```

```
    X.extend(facce)
    y.extend(labels)
  return asarray(X), asarray(y)
```

Listing 3.14: *carica_dataset*

Infine la funzione *carica_dataset* riceve in input la directory di un database e richiama la funzione *carica_facce* per ogni sottodirectory contenuta nel database. Quindi in output restituisce due array: uno contenente i volti estratti e uno contenente i nomi (labels) delle identità interne al database. Gli array ottenuti dalla funzione *carica_dataset* vengono salvati in un file compresso.

```
#carica il dataset di training
trainX, trainy = carica_dataset('directory-dataset-train')
#salva gli array in un file compresso
savez_compressed('filename.npz', trainX, trainy)
```

Listing 3.15: salvataggio file

Questo processo è stato applicato a tutti i database di train e test.

Generazione embeddings

In questa parte di codice vengono caricati i file compressi generati durante la fase di identificazione dei volti, contenenti i volti dei database di train e di test.

```
#carica i dataset di facce precedentemente trovate
data_train = load('filename-train.npz')
trainX, trainy = data_train['arr_0'], data_train['arr_1']
data_test = load('filename-test.npz')
testX, testy = data_test['arr_0'], data_test['arr_1']
```

Listing 3.16: loading dataset

Il modello VGGFace2 può essere creato su Keras usando la funzione costruttore *VGGFace()* e specificando il tipo di modello da creare tramite l'argomento. In questo caso è stato utilizzato il modello 'resnet50'.

```
model = VGGFace(model='resnet50')
```

Listing 3.17: modello VGGFace2

A questo punto è stata definita la funzione *ottieni_embedding*. Tale funzione riceve in input un volto che, dopo essere stato adattato, viene utilizzato dal modello per fare una predizione e generare un embedding.

```
def ottieni_embedding(model, pixel_faccia):
   #scala i valori dei pixel
   pixel_faccia = pixel_faccia.astype('float32')
   #prepara il volto al modello
   pixel_faccia = preprocess_input(pixel_faccia, version = 2)
   #per fare la predizione, bisogna espandere la dimensione in modo
    tale che il face array sia un campione
   campioni = expand_dims(pixel_faccia, axis=0)
   #si usa il modello per fare una predizione ed estrarre i vettori di
    embedding
   yhat = model.predict(campioni)
   return yhat[0]
```

Listing 3.18: *ottieni_embedding*

Tramite un ciclo *for* si applica la funzione *ottieni_embedding* a tutti i volti contenuti nei file compressi precedentemente caricati, ottenendo quindi un vettore di embeddings di train e un vettore di embeddings di test.

```
#converte ogni faccia nel train set in un embedding
newTrainX = list()
for pixel_faccia in trainX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTrainX.append(embedding)
newTrainX = asarray(newTrainX)

#converte ogni faccia nel test set in un embedding
newTestX = list()
for pixel_faccia in testX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTestX.append(embedding)
newTestX = asarray(newTestX)
```

Listing 3.19: creazione vettori di embeddings

Questi vettori vengono salvati in due diversi file compressi, insieme ai vettori contenenti i nomi delle identità del rispettivo database.

```
#salva gli array in un file in formato compresso
savez_compressed('embeddings_train.npz', newTrainX, trainy)
savez_compressed('embeddings_test.npz', newTestX, testy)
```

Listing 3.20: salvataggio file

Durante questa fase di generazione embeddings è stato misurato il tempo di esecuzione del programma, in modo da poter confrontare il tempo necessario alla rete per generare gli embeddings con il tempo necessario alle altre reti. Per poter valutare il tempo di esecuzione è stata utilizzata la funzione *time.monotonic*. Tale funzione restituisce (in frazioni di secondo) il valore di un orologio monotonico. Prendendo gli istanti di inizio e fine di esecuzione del programma si è poi misurato il tempo totale di esecuzione calcolando la differenza tra i due istanti. Il tempo ottenuto è stato poi stampato a schermo con la funzione *print*.
I risultati ottenuti sono riportati nel capitolo 3.3.

```
import time
start = time.monotonic()
time.sleep(0.1)
```

Listing 3.21: start time

```
end = time.monotonic()
print('tempo di esecuzione programma : -:9.2f '.format(end-start))
```

Listing 3.22: end time

3.2.3 OpenFace

Come anticipato nel capitolo 3.2, il codice realizzato è strutturato in tre blocchi. La parte relativa alla classificazione dei volti sarà trattata a parte nella sezione 3.2.5.

Identificazione volti

A differenza dei casi precedenti, per l'identificazione dei volti non è stato utilizzato MTCNN, ma la funzione *align* della classe *AlignDlib*. Tale funzione, oltre a identificare i volti, li ridimensiona in base all'input richiesto da OpenFace (96x96) ed esegue un allineamento dei volti in modo che i punti fondamentali (naso, occhi, bocca, contorno viso) appaiano nella stessa posizione in tutte le immagini. Sono state quindi definite tre funzioni: *estrai_faccia*, *carica_facce*, *carica_dataset*.

```
def estrai_faccia(nomefile):
  # carica l'immagine dal file
  img = cv2.imread(nomefile,1)
  # inverte l'ordine dei canali di colore perche' OpenCV carica le
    immagini con l'ordine BGR anziche' RGB
  img = img[...,::-1]
  # ottiene il delimitatore del volto piu' grande
  bb = alignment.getLargestFaceBoundingBox(img)
  # allinea e ridimensiona l'immagine
  img = alignment.align(96, img, bb, landmarkIndices=AlignDlib.OUTER_EYES
    _AND_NOSE)
  return img
```

Listing 3.23: *estrai_faccia*

La funzione *estrai_faccia* utilizza la funzione *align* sull'immagine fornita in input e restituisce il volto allineato in forma di vettore.

```
def carica_facce(directory):
   facce = list()
   for gruppo in listdir(directory):
     path = directory + '/' + gruppo
     for filename in listdir(path):
       new_path = path + '/' + filename
```

```
    #ottiene faccia
      faccia = estrai_faccia(new_path)
     #carica faccia
      facce.append(faccia)
  return facce
```

Listing 3.24: *carica_facce*

La funzione *carica_facce* richiama la funzione *estrai_faccia* per tutte le immagini di uno stesso individuo, contenute nella directory fornita in input.

```
def carica_dataset(directory):
  X,y = list(), list()
  for nome_personaggio in listdir(directory):
    path = directory + '/' + nome_personaggio
    #salta i file che dovrebbero essere nella directory
    if not isdir(path):
      continue
    #carica tutte le facce nella sottodirectory
    facce = carica_facce(path)
    #crea etichette
    labels = [nome_personaggio for _ in range(len(facce))]
    #elenca i progressi
    print('caricati %d esempi per la classe: %s' % (len(facce), nome
   _personaggio))
     #memorizza
    X.extend(facce)
    y.extend(labels)
  return asarray(X), asarray(y)
```

Listing 3.25: *carica_dataset*

Infine la funzione *carica_dataset* riceve in input la directory di un database e richiama la funzione *carica_facce* per ogni sottodirectory contenuta nel database. Quindi in output restituisce due array: uno contenente i volti estratti e uno contenente i nomi (labels) delle identità interne al database. Gli array ottenuti dalla funzione *carica_dataset* vengono salvati in un file compresso.

```
#carica il dataset di training
trainX, trainy = carica_dataset('directory-dataset-train')
#salva gli array in un file compresso
savez_compressed('filename.npz', trainX, trainy)
```

Listing 3.26: salvataggio file

Questo processo è stato applicato a tutti i database di train e test.

Generazione embeddings

In questa parte di codice vengono caricati i file compressi generati durante la fase di identificazione dei volti, contenenti i volti dei database di train e di test.

```
#carica i dataset di facce precedentemente trovate
data_train = load('filename-train.npz')
trainX, trainy = data_train['arr_0'], data_train['arr_1']
data_test = load('filename-test.npz')
testX, testy = data_test['arr_0'], data_test['arr_1']
```

Listing 3.27: loading dataset

Inoltre, tramite la funzione *load_model*, si carica il modello pre-addestrato di OpenFace, precedentemente salvato su Google Drive.

```
with CustomObjectScope('tf': tf):
  model = load_model('/content/drive/My Drive/Kerasopenface/modello-
    OpenFace.h5')
```

Listing 3.28: *load_model*

A questo punto è stata definita la funzione *ottieni_embedding*. Tale funzione riceve in input un volto che, dopo essere stato adattato, viene utilizzato dal modello per fare una predizione e generare un embedding.

```
def ottieni_embedding(model, pixel_faccia):
   pixel_faccia = (pixel_faccia/ 255.).astype(np.float32)
   #per fare la predizione, bisogna espandere la dimensione in modo
    tale che il face array sia un campione.  Si usa il modello per fare
    una predizione ed estrarre i vettori di embedding
```

```
  yhat = model.predict(np.expand_dims(pixel_faccia,axis=0))
  return yhat[0]
```

Listing 3.29: *ottieni_embedding*

Tramite un ciclo *for* si applica la funzione *ottieni_embedding* a tutti i volti contenuti nei file compressi precedentemente caricati, ottenendo quindi un vettore di embeddings di train e un vettore di embeddings di test.

```
#converte ogni faccia nel train set in un embedding
newTrainX = list()
for pixel_faccia in trainX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTrainX.append(embedding)
newTrainX = asarray(newTrainX)

#converte ogni faccia nel test set in un embedding
newTestX = list()
for pixel_faccia in testX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTestX.append(embedding)
newTestX = asarray(newTestX)
```

Listing 3.30: creazione vettori di embeddings

Questi vettori vengono salvati in due diversi file compressi, insieme ai vettori contenenti i nomi delle identità del rispettivo database.

```
#salva gli array in un file in formato compresso
savez_compressed('embeddings-train.npz', newTrainX, trainy)
savez_compressed('embeddings-test.npz', newTestX, testy)
```

Listing 3.31: salvataggio file

Durante questa fase di generazione embeddings è stato misurato il tempo di esecuzione del programma, in modo da poter confrontare il tempo necessario alla rete per generare gli embeddings con il tempo necessario alle altre reti. Per poter valutare il tempo di esecuzione è stata utilizzata la funzione *time.monotonic*. Tale funzione restituisce (in frazioni di secondo) il valore di un orologio monotonico. Prendendo gli istanti di inizio e fine di esecuzione del

programma si è poi misurato il tempo totale di esecuzione calcolando la differenza tra i due istanti. Il tempo ottenuto è stato poi stampato a schermo con la funzione *print*.

I risultati ottenuti sono riportati nel capitolo 3.3.

```
import time
start = time.monotonic()
time.sleep(0.1)
```

Listing 3.32: start time

```
end = time.monotonic()
print('tempo di esecuzione programma : -:9.2f '.format(end-start))
```

Listing 3.33: end time

3.2.4 InsightFace

Come anticipato nel capitolo 3.2, il codice realizzato è strutturato in tre blocchi. La parte relativa alla classificazione dei volti sarà trattata a parte nella sezione 3.2.5.

Identificazione volti

Anche in questo caso, come per OpenFace, per l'identificazione dei volti è stato utilizzata la funzione *align* della classe *AlignDlib*. Tale funzione, oltre a identificare i volti, li ridimensiona in base all'input richiesto da InsightFace (112x112) ed esegue un allineamento dei volti in modo che i punti fondamentali (naso, occhi, bocca, contorno viso) appaiano nella stessa posizione in tutte le immagini. Sono state definite tre funzioni: *estrai_faccia*, *carica_facce*, *carica_dataset*.

```
def estrai_faccia(nomefile):
  # carica l'immagine dal file
  img = cv2.imread(nomefile,1)
  # inverte l'ordine dei canali di colore perche' OpenCV carica le
    immagini con l'ordine BGR anziche' RGB
  img = img[...,::-1]
  # ottiene il delimitatore del volto piu' grande
  bb = alignment.getLargestFaceBoundingBox(img)
  # allinea e ridimensiona l'immagine
  img = alignment.align(112, img, bb, landmarkIndices=AlignDlib.OUTER_EYES
    _AND_NOSE)
  return img
```

Listing 3.34: *estrai_faccia*

La funzione *estrai_faccia* utilizza la funzione *align* sull'immagine fornita in input e restituisce il volto allineato in forma di vettore.

```
def carica_facce(directory):
   facce = list()
   for gruppo in listdir(directory):
     path = directory + '/' + gruppo
     for filename in listdir(path):
       new_path = path + '/' + filename
     #ottiene faccia
```

```
8        faccia = estrai_faccia(new_path)
9        pyplot.imshow(faccia)
10       if faccia is None:
11         print(filename)
12      #carica faccia
13       facce.append(faccia)
14   return facce
```

Listing 3.35: *carica_facce*

La funzione *carica_facce* richiama la funzione *estrai_faccia* per tutte le immagini di uno stesso individuo, contenute nella directory fornita in input.

```
1 def carica_dataset(directory):
2   X,y = list(), list()
3   for nome_personaggio in listdir(directory):
4     path = directory + '/' + nome_personaggio
5     #salta i file che dovrebbero essere nella directory
6     if not isdir(path):
7       continue
8     #carica tutte le facce nella sottodirectory
9     facce = carica_facce(path)
10    #crea etichette
11    labels = [nome_personaggio for _ in range(len(facce))]
12    #elenca i progressi
13    print('caricati %d esempi per la classe: %s' % (len(facce), nome
    _personaggio))
14     #memorizza
15    X.extend(facce)
16    y.extend(labels)
17
18  return asarray(X), asarray(y)
```

Listing 3.36: *carica_dataset*

Infine la funzione *carica_dataset* riceve in input la directory di un database e richiama la funzione *carica_facce* per ogni sottodirectory contenuta nel database. Quindi in output restituisce due array: uno contenente i volti estratti e uno contenente i nomi (labels) delle identità in-

terne al database. Gli array ottenuti dalla funzione *carica_dataset* vengono salvati in un file compresso.

```
#carica il dataset di training
trainX, trainy = carica_dataset('directory-dataset-train')
#salva gli array in un file compresso
savez_compressed('filename.npz', trainX, trainy)
```

Listing 3.37: salvataggio file

Questo processo è stato applicato a tutti i database di train e test.

Generazione embeddings

In questa parte di codice vengono caricati i file compressi generati durante la fase di identificazione dei volti, contenenti i volti dei database di train e di test.

```
#carica i dataset di facce precedentemente trovate
data_train = load('filename-train.npz')
trainX, trainy = data_train['arr_0'], data_train['arr_1']
data_test = load('filename-test.npz')
testX, testy = data_test['arr_0'], data_test['arr_1']
```

Listing 3.38: loading dataset

Inoltre, tramite la funzione *load_model*, si carica il modello pre-addestrato di InsightFace, precedentemente salvato su Google Drive.

```
model = load_model('/content/insightface.h5')
```

Listing 3.39: *load_model*

A questo punto è stata definita la funzione *ottieni_embedding*. Tale funzione riceve in input un volto che, dopo essere stato adattato, viene utilizzato dal modello per fare una predizione e generare un embedding.

```
def ottieni_embedding(model, pixel_faccia):
   pixel_faccia = pixel_faccia.astype('float32')
   #per fare la predizione, bisogna espandere la dimensione in modo
    tale che il face array sia un campione.  Si usa il modello per fare
    una predizione ed estrarre i vettori di embedding
```

```
  samples = expand_dims(pixel_faccia, axis=0)
  yhat = model.predict(samples)
  return yhat[0]
```

Listing 3.40: *ottieni_embedding*

Tramite un ciclo *for* si applica la funzione *ottieni_embedding* a tutti i volti contenuti nei file compressi precedentemente caricati, ottenendo quindi un vettore di embeddings di train e un vettore di embeddings di test.

```
#converte ogni faccia nel train set in un embedding
newTrainX = list()
for pixel_faccia in trainX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTrainX.append(embedding)
newTrainX = asarray(newTrainX)

#converte ogni faccia nel test set in un embedding
newTestX = list()
for pixel_faccia in testX:
  embedding = ottieni_embedding(model, pixel_faccia)
  newTestX.append(embedding)
newTestX = asarray(newTestX)
```

Listing 3.41: creazione vettori di embeddings

Questi vettori vengono salvati in due diversi file compressi, insieme ai vettori contenenti i nomi delle identità del rispettivo database.

```
#salva gli array in un file in formato compresso
savez_compressed('embeddings_train.npz', newTrainX, trainy)
savez_compressed('embeddings_test.npz', newTestX, testy)
```

Listing 3.42: salvataggio file

Durante questa fase di generazione embeddings è stato misurato il tempo di esecuzione del programma, in modo da poter confrontare il tempo necessario alla rete per generare gli embeddings con il tempo necessario alle altre reti. Per poter valutare il tempo di esecuzione è stata utilizzata la funzione *time.monotonic*. Tale funzione restituisce (in frazioni di secondo)

il valore di un orologio monotonico. Prendendo gli istanti di inizio e fine di esecuzione del programma si è poi misurato il tempo totale di esecuzione calcolando la differenza tra i due istanti. Il tempo ottenuto è stato poi stampato a schermo con la funzione *print*.

I risultati ottenuti sono riportati nel capitolo 3.3.

```
import time
start = time.monotonic()
time.sleep(0.1)
```

Listing 3.43: start time

```
end = time.monotonic()
print('tempo di esecuzione programma : -:9.2f '.format(end-start))
```

Listing 3.44: end time

3.2.5 Classificazione volti

Dopo aver ottenuto i vettori di embeddings generati da ciascuna rete, è stato possibile passare alla fase di classificazione dei volti. In questa ultima parte sono stati caricati i file compressi creati durante la fase di generazione degli embeddings. In seguito, i vettori di embeddings caricati sono stati normalizzati tramite la classe *Normalizer* della libreria *scikit-learn*. Inoltre, i vettori contenenti i nomi delle identità sono stati convertiti in interi tramite la classe *LabelEncoder*. A questo punto è stato addestrato il classificatore SVC, fornendo al modello i vettori di embeddings di train e di test. Il classificatore ha quindi generato una predizione per ciascun elemento dei vettori. Infine, è stata valutata la percentuale di predizioni corrette usando la funzione *accuracy_score*. Le percentuali ottenute rappresentano l'accuratezza della rete valutata sui database di train e test.

I risultati ottenuti sono riportati nel capitolo 3.3.

```
# carica il dataset
data_train= load('embeddings-train.npz')
data_test = load('embeddings-test.npz')
trainX, trainy = data_train['arr_0'], data_train['arr_1']
testX, testy = data_test['arr_0'], data_test['arr_1']
print('Dataset: train=%d, test=%d' % (trainX.shape[0], testX.shape[0]))
# normalizza i vettori
inencoder = Normalizer(norm='l2')
trainX = inencoder.transform(trainX)
testX = inencoder.transform(testX)
# converte i nomi delle classi in interi
out_encoder = LabelEncoder()
out_encoder.fit(trainy)
trainy = out_encoder.transform(trainy)
testy = out_encoder.transform(testy)
# addestra il modello
model = SVC(kernel='linear', probability=True)
model.fit(trainX, trainy)
# predizione
yhat_train = model.predict(trainX)
yhat_test = model.predict(testX)
```

```
# accuratezza
score_train = accuracy_score(trainy, yhat_train)
score_test = accuracy_score(testy, yhat_test)
# stampa i risultati
print('Accuracy: train=%.3f, test=%.3f' % (score_train*100, score_test*100))
```

Listing 3.45: valutazione accuratezza

Durante questa fase è stato valutato anche un altro parametro, ovvero il tempo impiegato da ogni rete per riconoscere un singolo volto, scelto in modo casuale dal database di test. Per valutare il tempo di una singola classificazione è stata utilizzata la funzione *time.monotonic*, già vista precedentemente nella fase di generazione degli embeddings.
Il volto casuale da classificare è stato scelto tramite la funzione *choice*.

```
selection = choice([i for i in range(testX.shape[0])])
random_face_pixels = testX_faces[selection]
random_face_emb = testX[selection]
random_face_classe = testy[selection]
random_face_name = out_encoder.inverse_transform([random_face_classe])
```

Listing 3.46: scelta casuale dell'esempio da classificare

Una volta ottenuto l'esempio da classificare, sono stati presi il corrispondente embedding, la classe predetta attesa e il corrispondente nome della classe. L'embedding è stato poi usato per fare una singola predizione con il classificatore. Sono state predette anche la classe e la probabilità della predizione.

```
samples = expand_dims(random_face_emb, axis=0)
yhat_classe = model.predict(samples)
yhat_prob = model.predict_proba(samples)
```

Listing 3.47: predizione

Una volta ottenuta la predizione, il programma ha terminato l'esecuzione, riportando in output il tempo di esecuzione, l'esito della classificazione e il volto scelto.
In fig. 3.2 è riportato un esempio di una classificazione effettuata da FaceNet sul database di 40 immagini per individuo.

Per ogni database sono stati effettuati 10 test di classificazione con ciascuna rete, ottenendo in tal modo i tempi medi di classificazione di un singolo volto, riportati nel cap. 3.3.

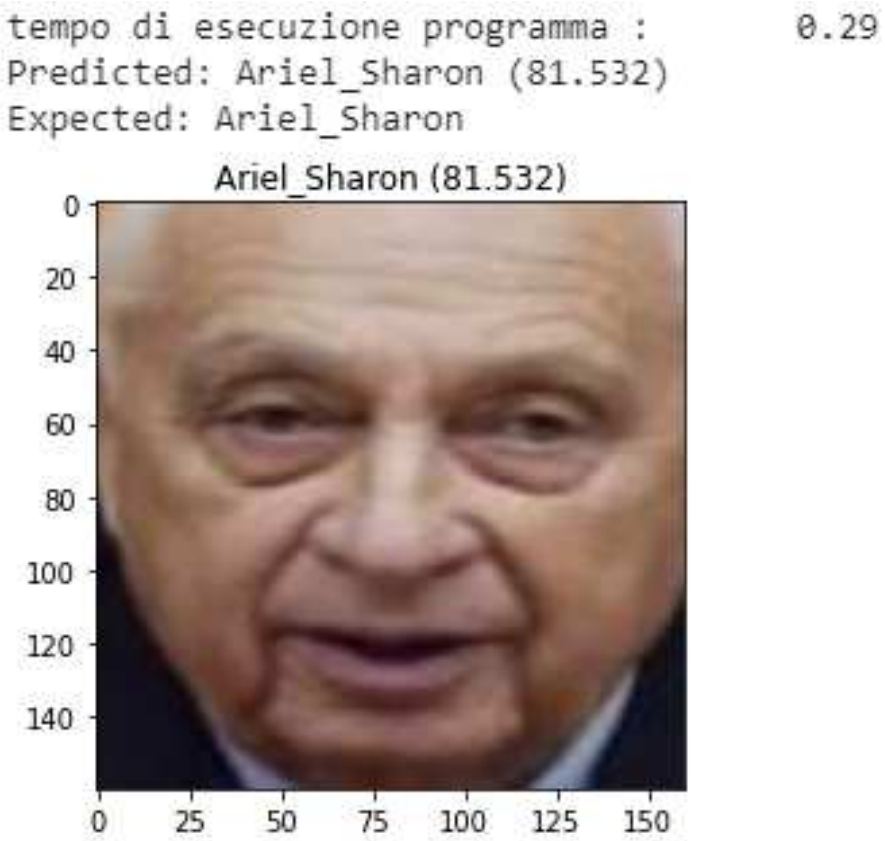


Figura 3.2: Esempio di classificazione

3.3 Risultati ottenuti

I risultati ottenuti da ogni rete durante i test effettuati sono riportati nelle tabelle 3.1, 3.2, 3.3 e 3.4. Ogni tabella riporta, in base al numero di immagini per identità contenute nel database, i seguenti valori:

- Accuratezza Train: É l'accuratezza media ottenuta dalla rete sul database di Train. É stata calcolata come la media aritmetica tra i cinque valori di accuratezza ottenuti su ciascun database di train usando la tecnica della validazione incrociata.
- Accuratezza Test: É l'accuratezza media ottenuta dalla rete sul database di Test. É stata calcolata come la media aritmetica tra i cinque valori di accuratezza ottenuti su ciascun database di test usando la tecnica della validazione incrociata.
- Generazione embeddings: É il tempo medio di esecuzione impiegato dalla rete nel generare gli embeddings. É stato calcolato come la media aritmetica tra i cinque tempi di esecuzione ottenuti su ciascun database usando la tecnica della validazione incrociata.
- Classificazione: É il tempo necessario alla rete per classificare un singolo volto preso dal database di test. É stato calcolato come la media aritmetica dei tempi ottenuti in 10 diversi test di classificazione, effettuati su ciascun database.

FaceNet				
Database	Accuratezza train	Accuratezza test	Generazione embeddings [s]	Classificazione [s]
10 volti	98.761%	99.558%	228,54s	1.96s
20 volti	99.891%	99.674%	86.74s	0.81s
30 volti	100.000%	99.855%	71.67s	0.42s
40 volti	100.000%	99.821%	55.34s	0.30s

Tabella 3.1: FaceNet

VGGFace2				
Database	Accuratezza train	Accuratezza test	Generazione embeddings [s]	Classificazione [s]
10 volti	96.350%	91.239%	269.69s	91.51s
20 volti	99.212%	97.283%	196.02s	43.52s
30 volti	99.167%	97.826%	142.74s	17.55s
40 volti	100.000%	99.464%	118.03s	9.46s

Tabella 3.2: VGGFace2

OpenFace				
Database	Accuratezza train	Accuratezza test	Generazione embeddings [s]	Classificazione [s]
10 volti	96.659%	91.327%	60.65s	1.66s
20 volti	99.103%	96.739%	46.19s	0.66s
30 volti	99.167%	97.826%	34.71s	0.34s
40 volti	98.884%	98.036%	28.76s	0.26s

Tabella 3.3: OpenFace

InsightFace				
Database	Accuratezza train	Accuratezza test	Generazione embeddings [s]	Classificazione [s]
10 volti	94.934%	65.752%	604.31s	7.01s
20 volti	95.896%	79.130%	489.28s	3.89s
30 volti	97.572%	85.797%	373.29s	1.85s
40 volti	97.857%	87.143%	305.31s	1.09s

Tabella 3.4: InsightFace

Per poter confrontare meglio le prestazioni delle reti, sono stati realizzati quattro grafici.
In ogni grafico sono riportati nelle ascisse i database utilizzati, ordinati in base al numero di immagini per identità contenute nel database. Nel grafico 1 sono riportati i valori di accuratezza ottenuti da ciascuna rete sui database di train. Nel grafico 2 sono riportati i valori di accuratezza ottenuti da ciascuna rete sui database di test. Nel grafico 3 sono riportati i tempi medi impiegati per generare gli embeddings ottenuti da ciascuna rete sui database utilizzati. Infine nel grafico 4 sono riportati i tempi medi impiegati per la classificazione di un volto, ottenuti da ciascuna rete sui database utilizzati.

Grafico 1: Accuratezza delle reti rispetto al database di train

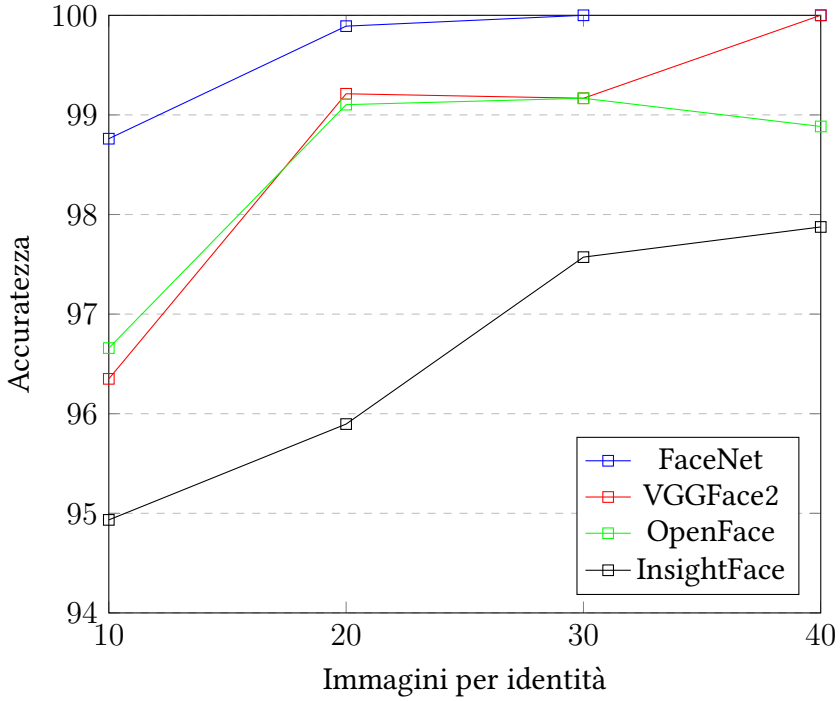


Grafico 2: Accuratezza delle reti rispetto al database di test

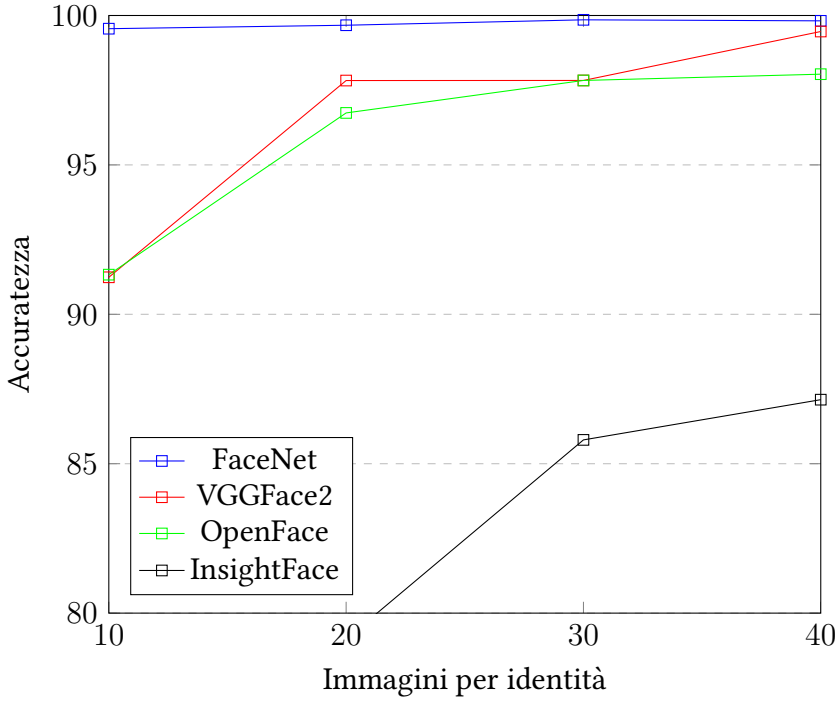


Grafico 3: Tempo di esecuzione generazione degli embeddings

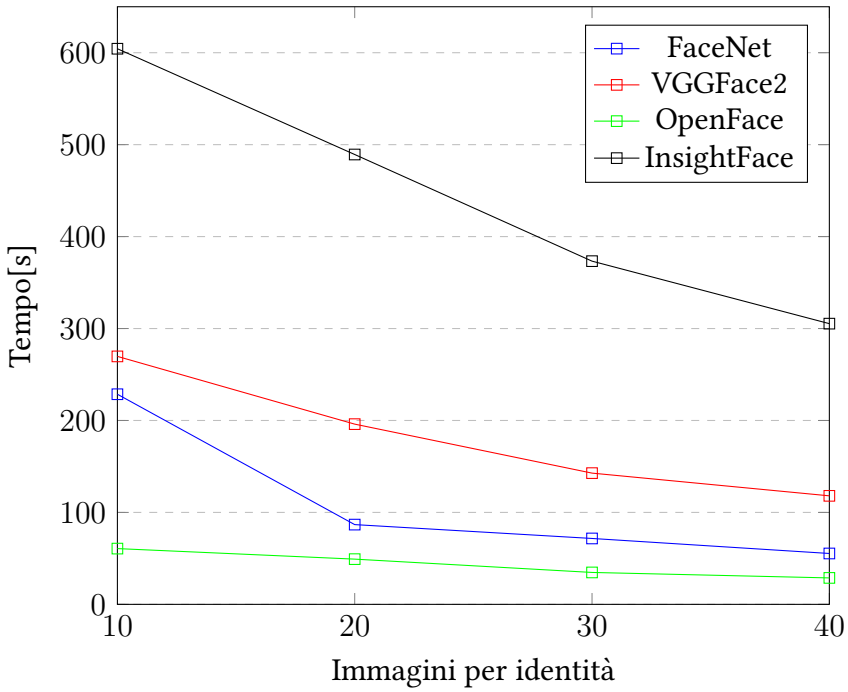
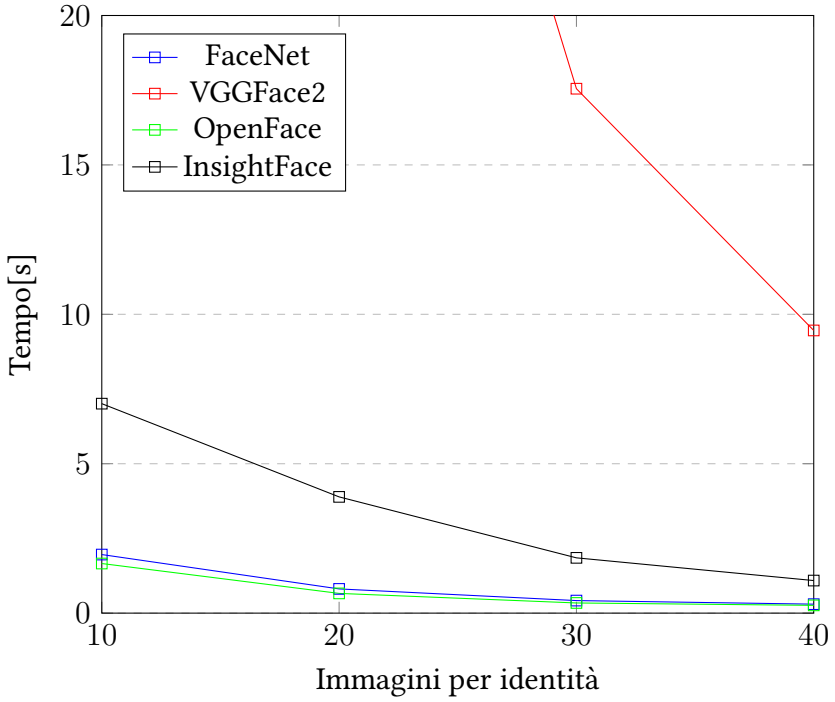


Grafico 4: Tempo di classificazione di un volto



Dai valori riportati nelle tebelle e nei grafici si possono evincere delle tendenze comuni a tutte le reti. Infatti, osservando i grafici 1 e 2, si può notare che all'aumentare del numero di immagini per identità, a cui corrisponde una riduzione del numero di identità, l'accuratezza nel riconoscimento tende ad aumentare. Questo risultato in realtà non è del tutto inaspettato. Era infatti prevedibile che al diminuire delle identità da classificare diminuisse la probabilità di errore nella classificazione; a maggior ragione se si considera che, parallelamente al ridursi delle identità, si ha un incremento del numero di esempi per individuo con cui addestrare la rete.

Inoltre, dai grafici 3 e 4, si può notare che all'aumentare del numero di immagini per individuo si ha una riduzione dei tempi necessari sia a generare gli embeddings sia a classificare un volto. Questo risultato è dovuto al fatto che, pur aumentando il numero di immagini per individuo, la rete deve processare un numero totale inferiore di immagini, in quanto diminuiscono il numero di individui.

Un risultato inaspettato ottenuto durante i test è stato quello relativo all'accuratezza della rete InsightFace. Infatti, come si evince dalla tabella 3.4, è stata ottenuta una differenza significativa tra i valori di accuratezza sul database di train e quelli sul database di test. Il motivo di questi

valori inattesi risiede in un problema riscontrato durante la fase di test e diffuso nel campo del machine learning, denominato *overfitting* (sovradattamento). Per overfitting si intende quella situazione per cui il modello si adatta eccessivamente al database di train, prendendo in considerazione dati x irrilevanti per la decisione y. Quanto spiegato è proprio ciò che è accaduto alla rete InsightFace che, adattandosi eccessivamente ai dati di training, ha ottenuto buone prestazioni sul database di train mentre lo stesso non è accaduto sul database di test.
Il confronto tra le prestazioni delle reti è riportato nel capitolo 4.

Capitolo 4

Conclusioni

In questo capitolo sono riassunte le conclusioni a cui sono giunto confrontando i dati ottenuti nei test, riportati nel capitolo 3.3.
Prendendo in considerazione l'accuratezza nel riconoscimento, FaceNet è risultata essere la rete più performante sui database utilizzati, ottenendo i risultati migliori sia sui database di train che di test. Interessanti sono i risultati relativi all'accuratezza sui database di test, dove ha ottenuto percentuali mai inferiori al 99.5%. Non vanno comunque sottovalutate le prestazioni di OpenFace e VGGFace2. Le due reti hanno infatti ottenuto buoni risultati nel riconoscimento, in particolare VGGFace2 che, nel database contenente 40 volti per identità, ha raggiunto un'accuratezza di test molto vicina a quella di FaceNet. Dai test effettuati, la rete con prestazioni peggiori nel riconoscimento è risultata essere InsightFace, soggetta al problema di *overfitting* visto nel capitolo 3.3.
Considerando invece i dati relativi ai tempi necessari a generare gli embeddings, OpenFace è stata la rete che ha ottenuto tempi di esecuzione minori su tutti i database in cui è stata testata, richiedendo tempi nettamente più bassi rispetto a FaceNet, la seconda rete più veloce. Anche in questo caso, come già visto per l'accuratezza, la rete meno performante è risultata essere InsightFace, con tempi di esecuzione eccessivamente elevati rispetto alle altre reti.
Infine, valutando i tempi medi impiegati nella classificazione di un singolo volto, la rete con prestazioni migliori è stata OpenFace, ottenendo risultati, seppur di poco, migliori rispetto a FaceNet. In questo caso InsightFace ha ottenuto risultati non troppo distanti dalle due reti più performanti, mentre VGGFace2 ha richiesto tempi estremamente elevati, mostrando una bassa

rapidità nella classificazione.

Complessivamente, dai test effettuati, si può affermare che la rete con prestazioni migliori è risultata essere FaceNet. Infatti, tale rete ha ottenuto la miglior accuratezza nel riconoscimento, oltre ad aver richiesto tempi di esecuzione non eccessivamente elevati. Dal confronto è emerso anche che, seppur con dei compromessi, OpenFace e VGGFace2 possono essere considerate valide alternative a FaceNet. OpenFace ha infatti garantito la miglior rapidità di esecuzione, a discapito di un'accuratezza nel riconoscimento non ottimale. Al contrario VGGFace2 ha dimostrato di avere un'accuratezza migliore rispetto a OpenFace, ma ha richiesto tempi di esecuzione più elevati nella classificazione. Infine, InsightFace si è rivelata essere, dai test effettuati, la rete con prestazioni complessivamente peggiori, anche a causa del problema di overfitting incontrato. Va però sottolineato il dato relativo ai tempi necessari a classificare un singolo volto, in cui ha ottenuto prestazioni non troppo distanti da FaceNet e OpenFace. Non è quindi da escludere che, una volta risolto il problema di overfitting, InsightFace non possa rivelarsi una valida alternativa alle altre reti.

Bibliografia

[1] Ethem Alpaydin. *Introduction to Machine Learning*. Cambridge, Massachussets: The MIT press, 2020.

[2] Kevin P. Murphy. *Machine Learning: a probabilistic perspective*. pag 1-13. Massachusetts Institute of Technology, 2012.

[3] Stan Z. Li ,Anil K. Jain. *Handbook of Face Recognition*. Springer, 2011.

[4] Jason Brownlee. *How to Develop a Face Recognition System Using FaceNet in Keras*. https://machinelearningmastery.com/how-to-develop-a-face-recognition-system-using-facenet-in-keras-and-an-svm-classifier/.

[5] Jason Brownlee. *How to Perform Face Recognition With VGGFace2 in Keras*. https://machinelearningmastery.com/how-to-perform-face-recognition-with-vggface2-convolutional-neural-network-in-keras/.

[6] iwantooxxoox. *Keras-OpenFace*. https://github.com/iwantooxxoox/Keras-OpenFace.

[7] Jia Guo, Jiankang Deng. *InsightFace: 2D and 3D Face Analysis Project*. https://github.com/deepinsight/insightface.

[8] Michael Nielsen. *Neural Networks and Deep Learning*. 2019. http://neuralnetworksanddeeplearning.com.

[9] Dario Floreano, Claudio Mattiussi. *Manuale sulle reti neureli*. Il Mulino. 1996.

[10] University of Massachusetts. *Labeled faces in the wild.* http://vis-www.cs.umass.edu/lfw.

[11] Florian Schroff, Dmitry Kalenichenko, James Philbin. *FaceNet: A Unified Embedding for Face Recognition and Clustering.*

[12] Qiong Cao, Li Shen, Weidi Xie, Omkar M. Parkhi, Andrew Zisserman. *VGGFace2: A dataset for recognising faces across pose and age.* Visual Geometry Group, Department of Engineering Science, University of Oxford

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition*

[14] Brandon Amos, Bartosz Ludwiczuk, Mahadev Satyanarayanan. *OpenFace: A general-purpose face recognition library with mobile applications.*

[15] Jiankang Deng, Jia Guo, Niannan Xue, Stefanos Zafeiriou. *ArcFace: Additive Angular Margin Loss for Deep Face Recognition.*