

Biometric Gate Access

Secure access through facial recognition

Relazione del progetto di Biometric Systems

Anno accademico 2023/2024

Jessica Frabotta – 1758527

Alessio Ferrone – 1751859

Indice

Introduzione	3
Architettura del sistema	6
Librerie, datasets e scelta dell'hardware	13
Valutazione del modulo di face detection	23
Addestramento e valutazione del modulo di antispoofing	27
Valutazione del modulo di face recognition	36
Implementazione del sistema	45
Materiale del progetto	59

Introduzione

In un contesto in cui la ricerca di comodità, senza compromettere la sicurezza, è diventata un elemento fondamentale della vita quotidiana, l'uso della biometria offre una soluzione ideale ed efficiente.

Questi requisiti ci hanno portato a ideare un'applicazione Android in grado di aprire o chiudere cancelli, nonché di bloccare o sbloccare porte, in modo rapido tramite il riconoscimento facciale, offrendo una soluzione sicura, comoda ed automatizzata.

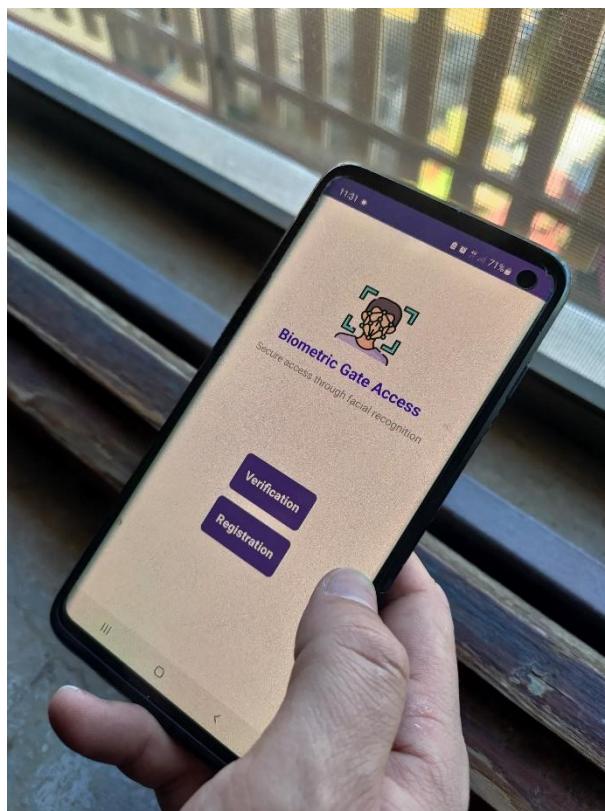


Figura 1: l'applicazione realizzata per questo progetto

L'applicazione sfrutta tre moduli chiave per garantire la robustezza e la sicurezza del sistema:

- **Face Detection:** individua la presenza di un volto nelle immagini acquisite dalla fotocamera del dispositivo.

- **Liveness Detection:** è un modulo di antispoofing che verifica se il volto rilevato appartiene effettivamente a una persona viva, prevenendo tentativi di spoofing tramite foto o video. In particolare, mira a contrastare i *print attacks*, che sfruttano una copia cartacea della foto dell'utente genuino per ingannare il sistema, e i *replay attacks*, che consistono nella registrazione di un'interazione precedente con il sistema, come l'acquisizione di un'immagine facciale autorizzata o di una sequenza di movimenti del viso, e nella riproduzione successiva per ottenere l'accesso non autorizzato.

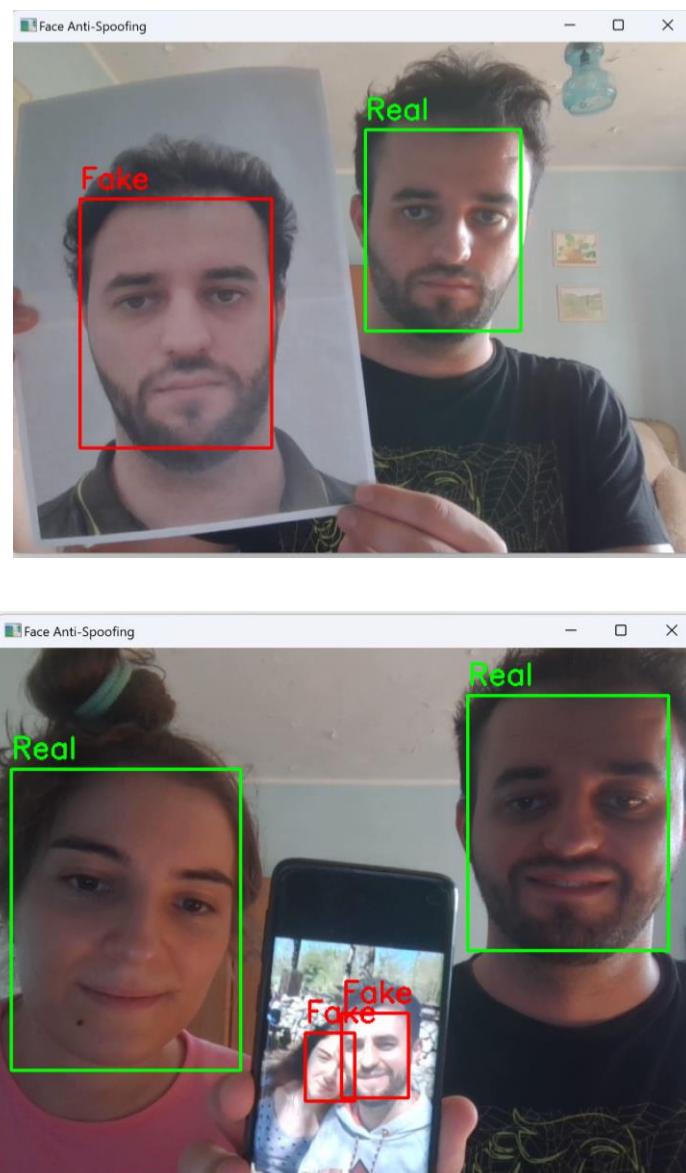


Figura 2: testing in real-time dei moduli di liveness detection e face detection

- **Face Recognition:** confronta il volto rilevato con i template dell'identità dichiarata, memorizzati su un server. Si tratta di un processo di *verification*, poiché viene effettuato un confronto uno ad uno per confermare se l'identità affermata dall'individuo è corretta. Se il confronto è positivo e le caratteristiche biometriche corrispondono al template memorizzato, l'identità è accettata; altrimenti, viene rifiutata. In questo progetto il confronto si basa sulle caratteristiche facciali estratte attraverso un modello di deep learning, rappresentate da *embeddings*.

L'applicazione Android comunica con un Raspberry Pi, che, tramite un driver motore, controlla l'estensione o la retrazione di un attuatore lineare. Questo attuatore consente di aprire e chiudere un cancello, premendo il relativo pulsante del citofono, e di inserire o rimuovere i ganci da una porta.



Figura 3: hardware utilizzato nel progetto

L'obiettivo di questa relazione è descrivere il processo di progettazione e sviluppo dell'applicazione, con un focus particolare sulla valutazione dei tre moduli biometrici utilizzati e sull'analisi dei risultati ottenuti.

Architettura del sistema

Il nostro sistema vede la comunicazione di tre componenti principali:

- **Applicazione per smartphone:** il dispositivo Android consente all'utente di acquisire immagini del volto tramite la fotocamera. Queste immagini vengono inviate al server, che esegue i moduli di face detection, liveness detection e face recognition. Una volta completata l'elaborazione, lo smartphone riceve una risposta con la decisione di accettazione o rifiuto.
- **Server:** esegue i tre moduli biometrici e memorizza i template corrispondenti a ciascun soggetto che esegue l'enrollment (solo se viene individuato correttamente un volto in tutte le immagini inviate). Se la verifica ha successo (ossia, il volto è individuato correttamente in tutte le immagini, il controllo di liveness viene superato e c'è una corrispondenza con i template dell'identità dichiarata), il server invia un messaggio di accettazione allo smartphone. Altrimenti, invia un messaggio di rifiuto.
- **Raspberry Pi:** se la verifica è andata a buon fine, lo smartphone invia una richiesta HTTP al Raspberry Pi. Quest'ultimo è collegato a un attuatore lineare e, una volta ricevuta la richiesta, aziona il motore per eseguire l'operazione scelta dall'utente. Il Raspberry Pi risponde poi con un segnale di conferma che l'operazione è stata eseguita correttamente.

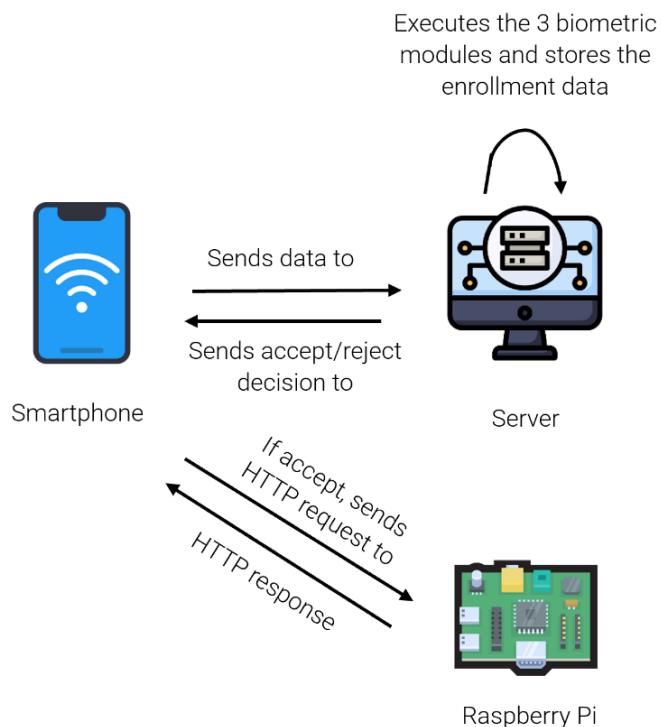


Figura 4: comunicazione tra i tre componenti principali

Nei grafici seguenti vengono analizzate nel dettaglio le due fasi fondamentali che costituiscono l'architettura di un sistema biometrico: l'enrollment e il riconoscimento (in questo caso specifico, si tratta di verification).

La policy adottata per la nostra applicazione prevede la cattura di tre immagini del volto per l'**enrollment**. I volti vengono memorizzati sul server dopo aver eseguito la face detection: solo se tutte e tre le immagini contengono un volto si procede con l'estrazione delle caratteristiche e la creazione dei template; in caso contrario, viene richiesto all'utente di ripetere la cattura delle foto. Un processo di preprocessing viene effettuato per preparare i dati alla face detection. L'identità associata ai template è determinata dall'Android ID, un identificatore univoco associato a ciascun dispositivo Android.

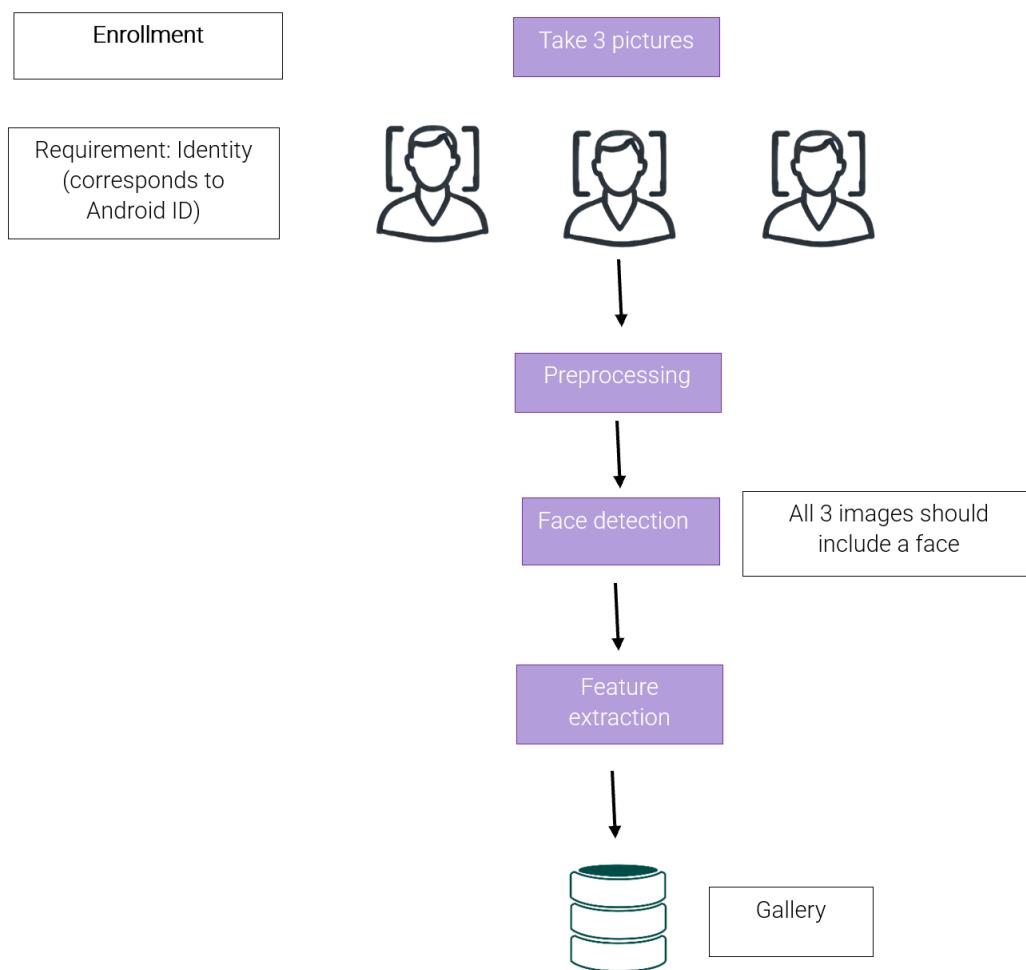


Figura 5: enrollment

Anche nel caso della **verification**, la nostra policy prevede la cattura di tre foto del volto. Le immagini passano attraverso una fase di preprocessing che include il ridimensionamento e l'applicazione di tecniche di normalizzazione. A questo punto, viene applicato il primo modulo, quello di rilevamento del volto. Se non tutte le tre immagini contengono un volto, la pipeline si interrompe e viene restituita immediatamente la decisione di "reject". Altrimenti, si procede con il modulo di liveness detection. Anche in questo caso, se almeno una delle tre foto viene rilevata come "spoof", la decisione finale sarà "reject" e il processo si ferma. In caso contrario, si passa al riconoscimento facciale, che include l'estrazione delle feature e il confronto con i template recuperati dalla gallery e corrispondenti all'identità dichiarata (determinata sempre dall'Android ID). Se coincidono, viene emessa una decisione finale di "accept", altrimenti "reject".

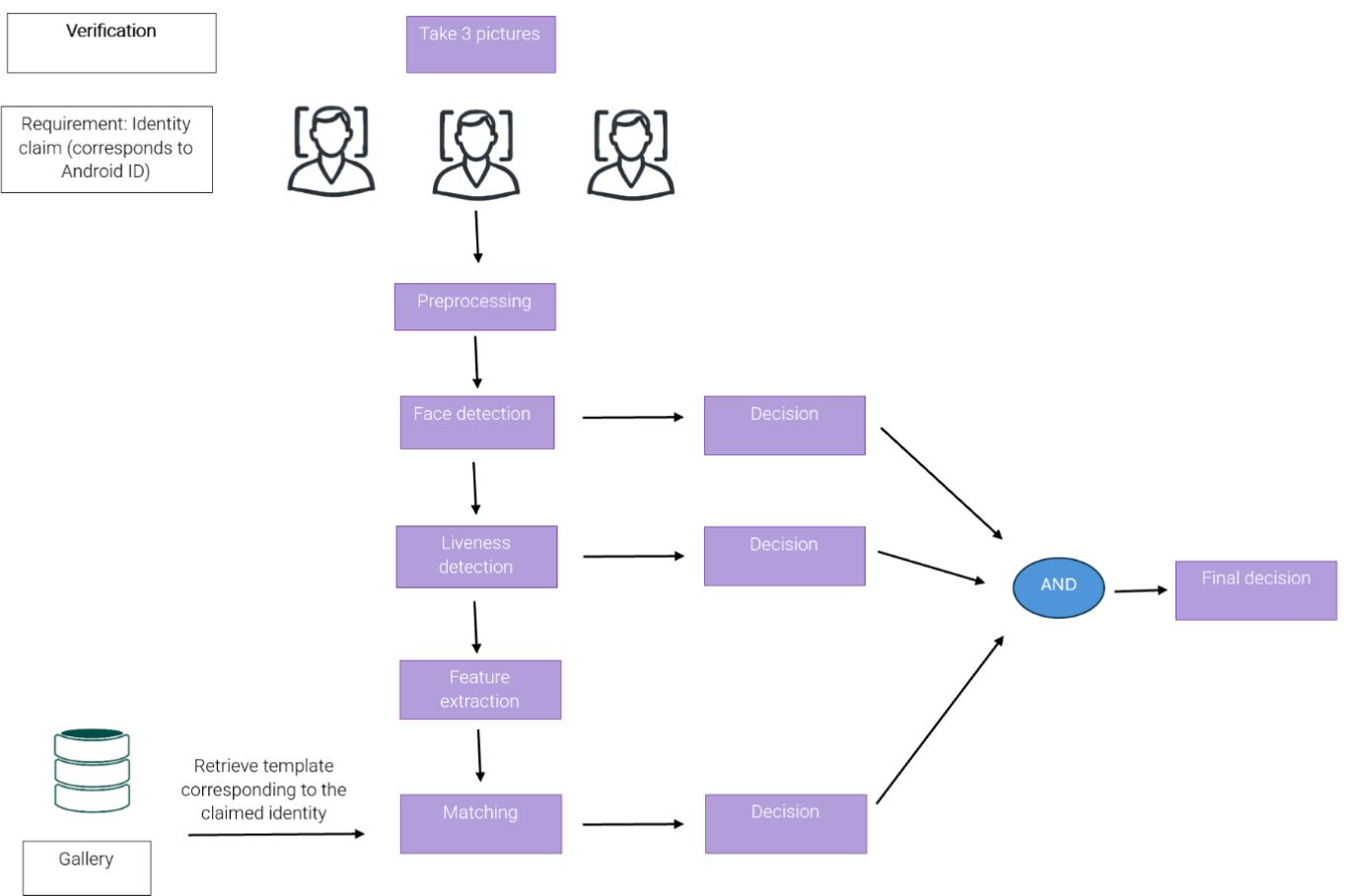


Figura 6: verification

In generale nella progettazione di un sistema, il processo inizia con la definizione delle funzionalità chiave che il sistema deve avere. Le funzionalità rappresentano le capacità generali del sistema. Successivamente, si utilizzano queste funzionalità come blocchi fondamentali per creare i casi d'uso, che rappresentano scenari specifici in cui gli utenti o i sistemi interagiscono con il sistema per raggiungere obiettivi specifici.

Un **diagramma dei casi d'uso** è una rappresentazione visuale di questi scenari, mostrando gli attori (utenti o sistemi esterni) coinvolti e le azioni che compiono per raggiungere i loro scopi. Queste sono le funzionalità che il sistema deve eseguire:

1. Connessione ad internet
2. Enrollment
 - 2.1. Apertura della fotocamera
 - 2.2. Cattura delle immagini del volto
 - 2.3. Invio delle foto al server
 - 2.4. Face detection
 - 2.5. Ricattura delle immagini del volto se la face detection fallisce
3. Verification
 - 3.1. Apertura della fotocamera
 - 3.2. Cattura delle immagini del volto
 - 3.3. Invio delle foto al server
 - 3.4. Face detection
 - 3.5. Liveness detection
 - 3.6. Features extraction e matching (face recognition)
 - 3.7. Ricezione della decisione
 - 3.7.1. Apertura/chiusura del cancello o lock/unlock della porta se la decisione è "accept"
 - 3.7.2. Invio del comando a Raspberry Pi
 - 3.7.2.1. Estensione dell'attuatore lineare
 - 3.7.2.2. Contrazione dell'attuatore lineare
 - 3.7.3. Ricattura delle immagini del volto se la decisione è "Reject"

Da queste funzionalità abbiamo ricavato il seguente diagramma dei casi d'uso:

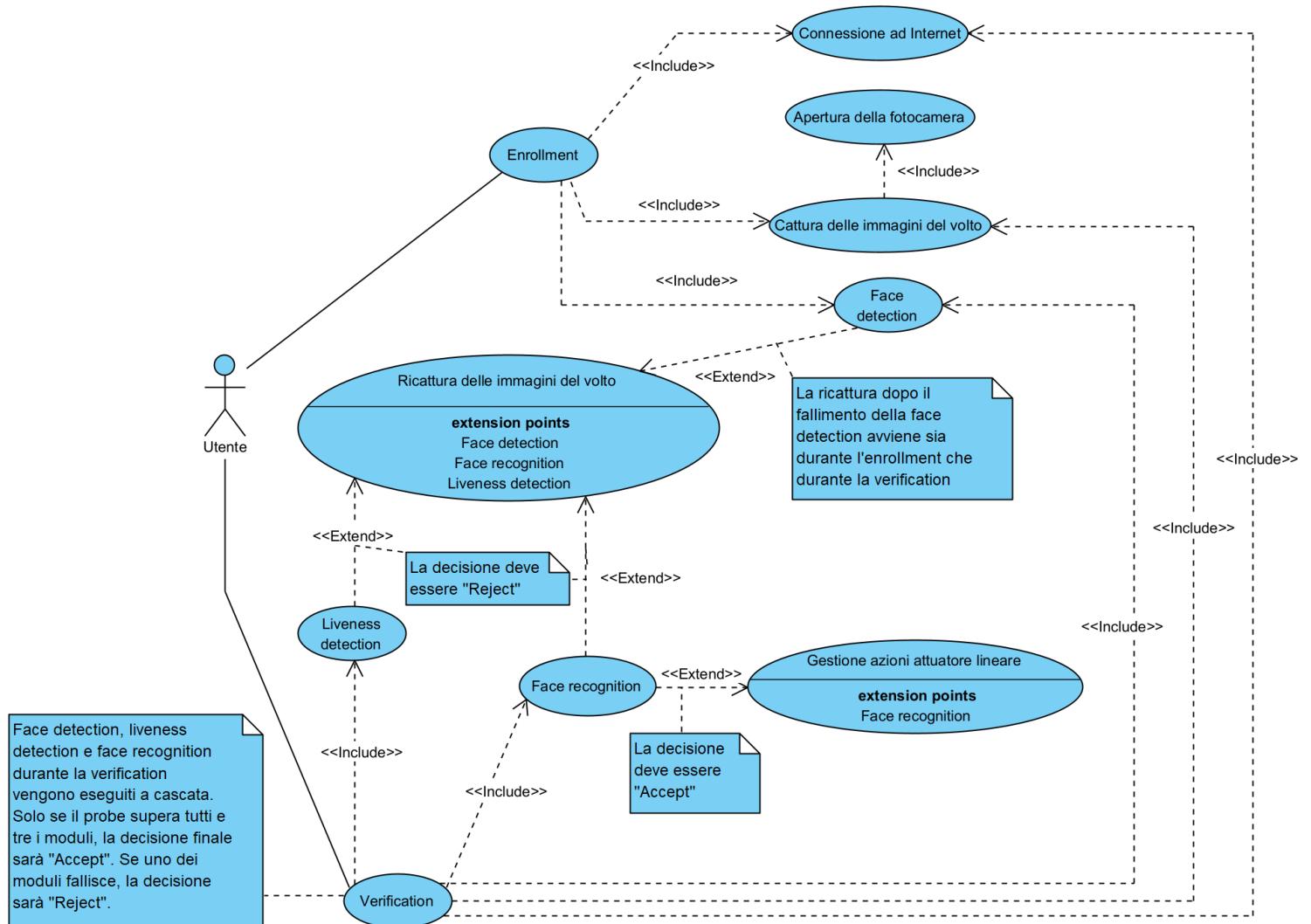


Figura 7: diagramma dei casi d'uso

I **diagrammi di attività** servono a rappresentare visivamente il flusso di lavoro o le sequenze di azioni all'interno di un sistema o di un processo. Sono utilizzati per modellare le attività, le decisioni, le condizioni e le interazioni tra gli oggetti o gli attori in un sistema.

I diagrammi di attività che verranno presentati sono:

- Enrollment
- Verification

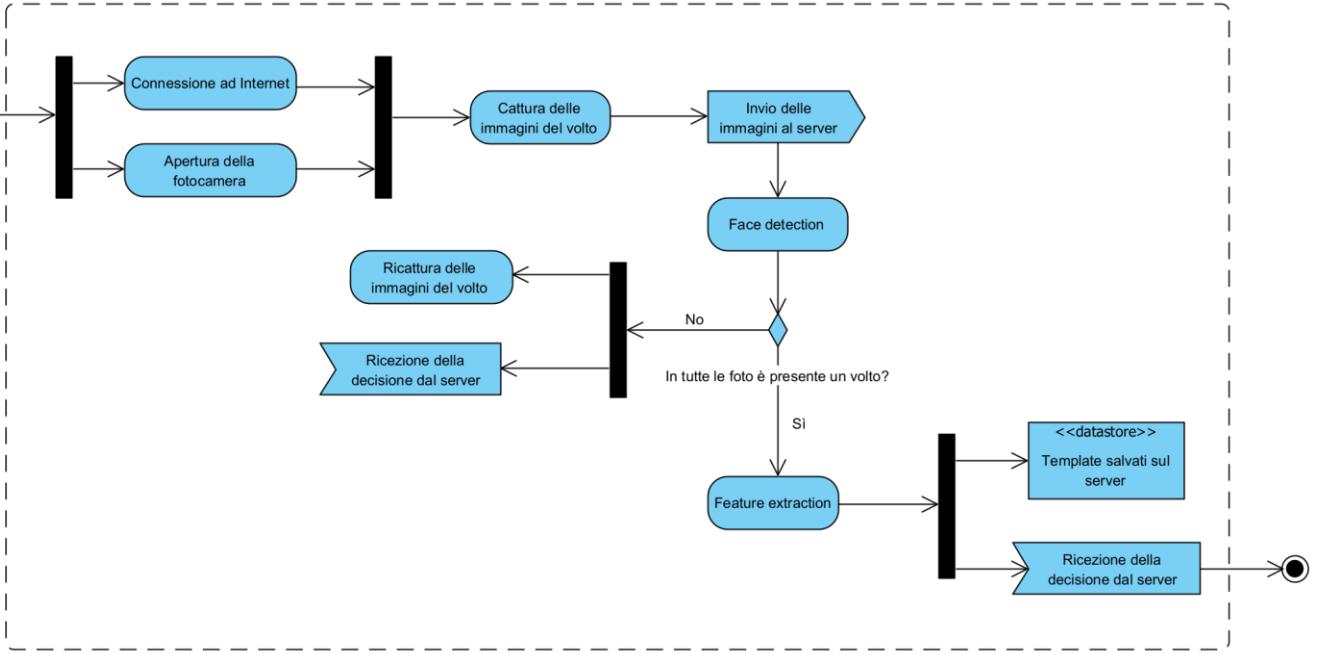


Figura 8: diagramma attività "enrollment"

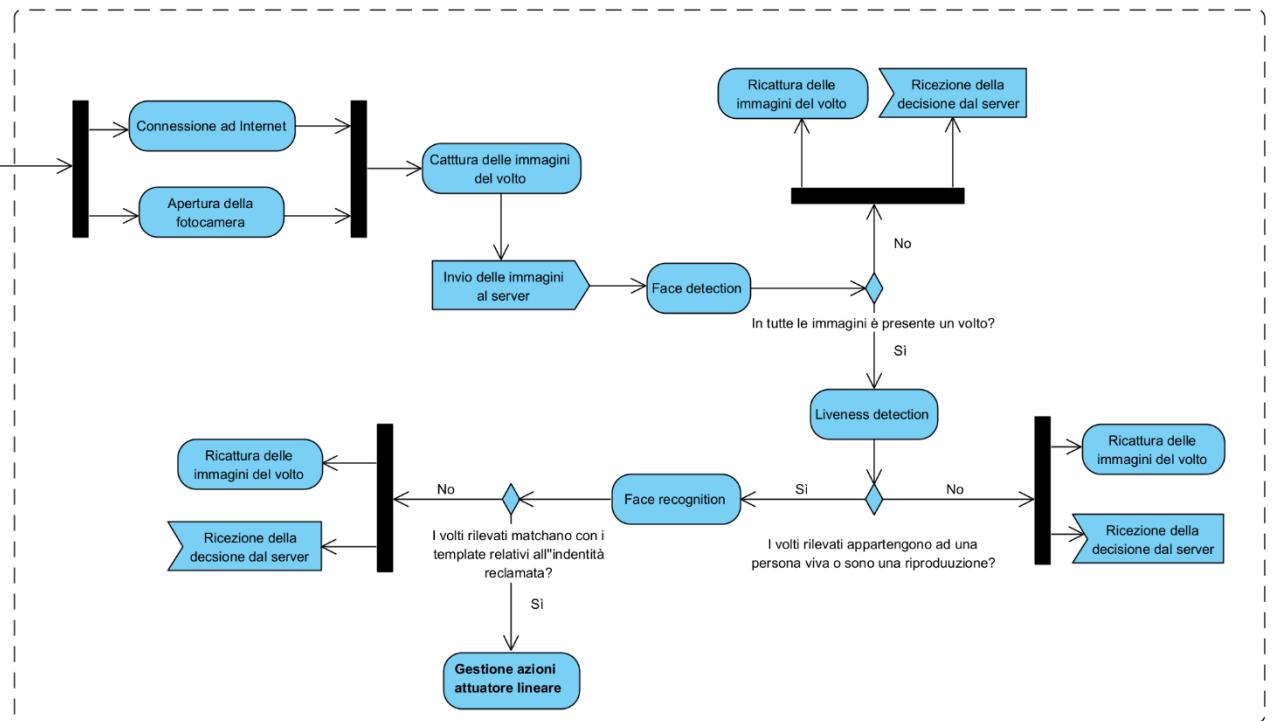


Figura 8: diagramma attività "verification"

I **diagrammi di sequenza** servono a rappresentare visivamente l'interazione tra gli oggetti o le componenti di un sistema software nel tempo. Essi mostrano

come gli oggetti comunicano tra loro attraverso messaggi, evidenziando l'ordine.

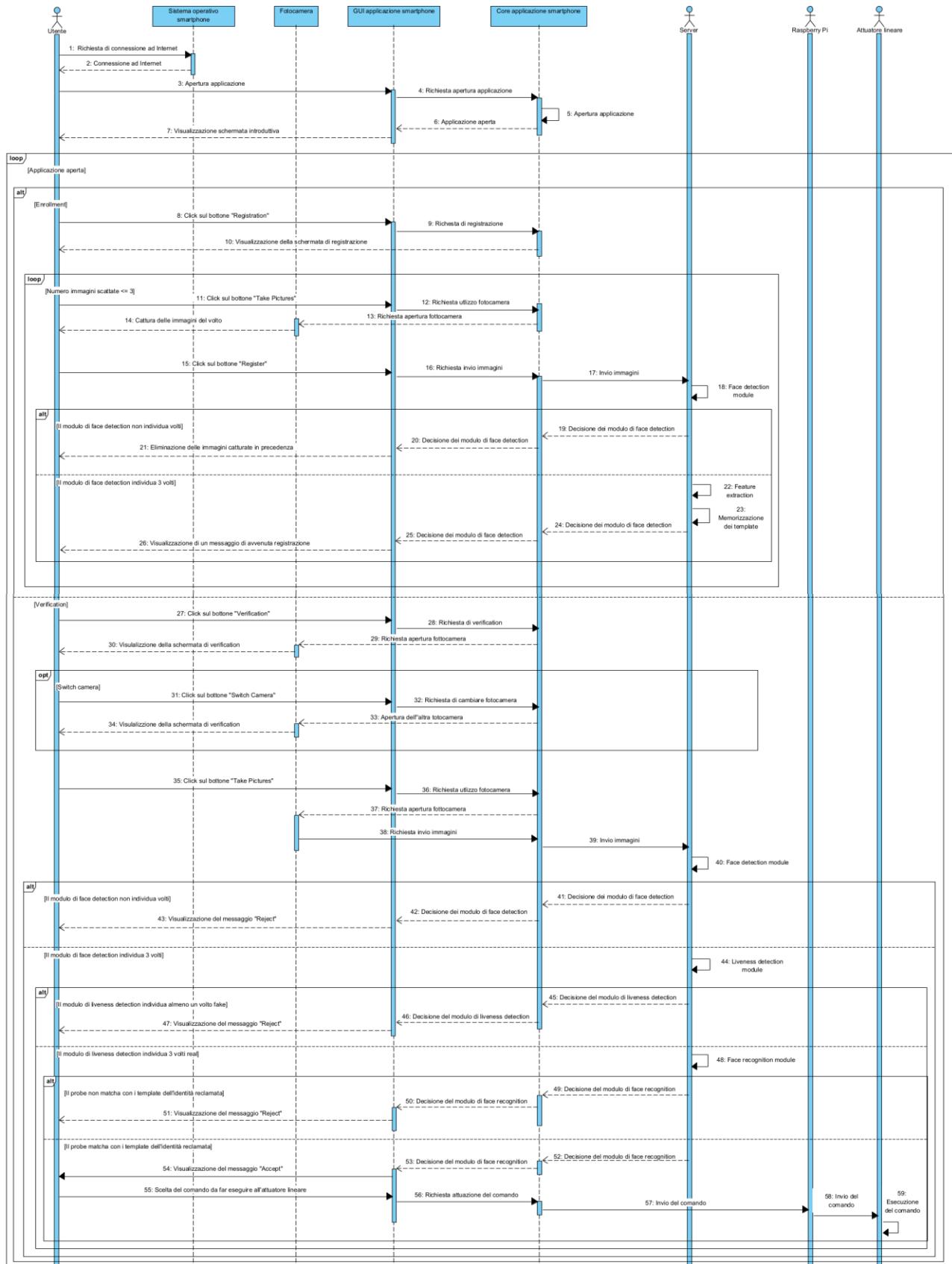


Figura 9: diagramma di sequenza per il sistema

Librerie, datasets e scelta dell'hardware

I linguaggi di programmazione utilizzati per la realizzazione di questo progetto sono **Python** e **Java**. Python è stato impiegato in tutte le fasi di addestramento e valutazione dei modelli, nonché per scrivere il codice relativo ai server necessari al funzionamento del sistema, tra cui quello creato su Raspberry Pi per la ricezione dei comandi da eseguire sull'attuatore lineare. Java è stato utilizzato esclusivamente per l'applicazione Android sviluppata tramite l'IDE Android Studio.

Per l'addestramento e la valutazione dei modelli integrati nel nostro sistema è stata utilizzata **PyTorch**, una libreria open source per il machine learning e il deep learning che consente di utilizzare tensori per rappresentare dati multidimensionali, gestibili sia su CPU che su GPU, accelerando il calcolo. La libreria include diversi modelli pre-addestrati, tra cui MobileNet_v2, che è stato utilizzato per realizzare il nostro modulo di liveness detection, riaddestrando il suo ultimo layer sui dataset che verranno presentati in seguito.

MobileNetV2 è una rete neurale convoluzionale (CNN) progettata per essere efficiente dal punto di vista computazionale. Si basa su due concetti chiave: **depthwise separable convolutions** e **inverted residual blocks**.

Le depthwise separable convolutions riducono il numero di operazioni dividendo una normale convoluzione in due fasi: convoluzione per singolo canale (*depthwise*) seguita da una convoluzione 1x1 per combinare i risultati (*pointwise*). Gli **inverted residuals** espandono i canali di input tramite convoluzioni 1x1, eseguono convoluzioni depthwise, e poi riducono i canali con un'altra convoluzione 1x1.

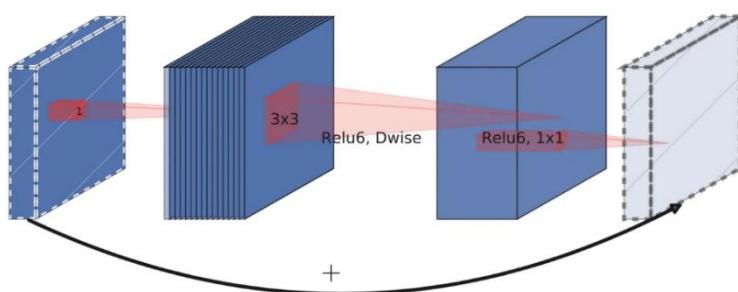


Figura 10: visualizzazione delle feature maps intermedie nell'inverted residual layer

In caso di uguaglianza nel numero di canali tra input e output, vengono usate le **skip connections** per il passaggio diretto delle informazioni. Inoltre, il modello usa un **linear bottleneck** alla fine di ogni blocco per evitare di perdere informazioni durante la compressione.

La libreria di elaborazione delle immagini **scikit-image** è stata utilizzata per applicare **Local Binary Pattern (LBP)** alle immagini nei dataset per l'antispoofing. LBP funziona confrontando i pixel di un'immagine con i suoi vicini per creare una rappresentazione binaria basata sulle differenze di intensità. Questa tecnica è spesso usata per l'analisi delle texture.

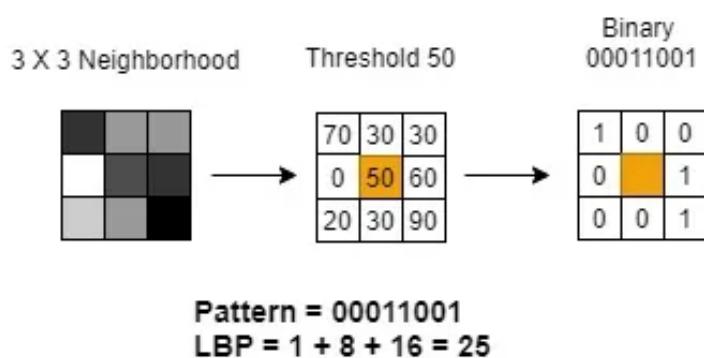


Figura 11: funzionamento di LBP

Ecco cosa fa, in sintesi:

1. Per ogni pixel, si seleziona una finestra (di solito 3x3) centrata su quel pixel.
2. Si confrontano i pixel circostanti con il valore del pixel centrale.
3. Se un pixel circostante è più grande o uguale al pixel centrale, viene assegnato un valore binario di 1, altrimenti 0.
4. Si ottiene così un codice binario che rappresenta la texture locale.

FaceNet è un'architettura di rete neurale convoluzionale profonda (CNN) progettata per compiti di riconoscimento facciale. Impara a mappare le immagini facciali in uno spazio di caratteristiche ad alta dimensionalità, dove la distanza euclidea tra gli embedding facciali corrisponde alla loro similarità. FaceNet utilizza la **triplet loss** come loss function durante l'addestramento per garantire che gli embedding del volto della stessa persona siano vicini nello spazio delle caratteristiche, mentre gli embedding di persone diverse siano distanti.

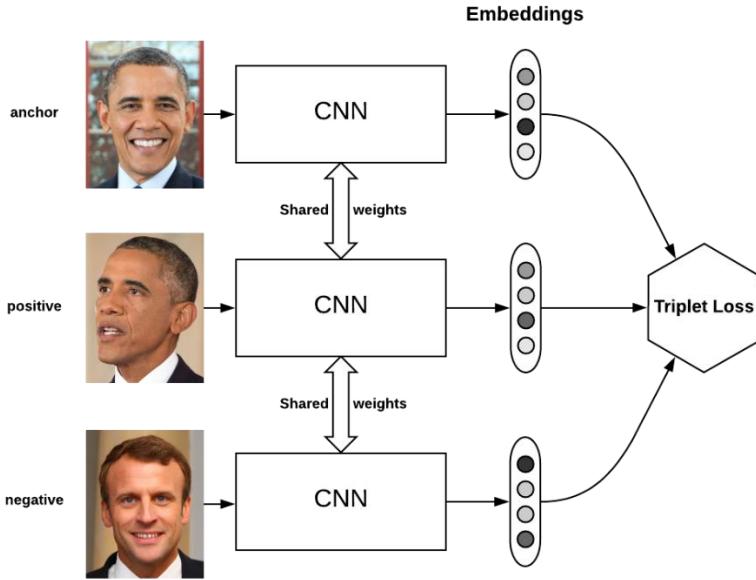


Figura 12: triplet loss

Utilizziamo questo modello nel nostro modulo di riconoscimento facciale attraverso la libreria **FaceNet-PyTorch**, che fornisce un'implementazione del modello FaceNet in PyTorch. L'architettura di base usata in questa implementazione è Inception Resnet (V1) pre-addestrata sui dataset **VGGFace2** e **CASIA-WebFace**. Inoltre, FaceNet-PyTorch fornisce anche un'implementazione in PyTorch di **MTCNN**, modello che utilizziamo per il nostro modulo di face detection.

MTCNN (Multi-task Cascaded Convolutional Neural Networks) è un modello di face detection composto da una struttura a cascata, il che significa che il processo di rilevamento facciale è suddiviso in più fasi, ognuna delle quali esegue un compito specifico. La rete a cascata consente di affinare progressivamente i risultati, partendo da una localizzazione approssimativa del volto fino a raggiungere una rilevazione precisa. In particolare, MTCNN è costituito da tre reti:

- **P-Net (Proposal Network)**: una rete che propone regioni candidate che potrebbero contenere volti.
- **R-Net (Refinement Network)**: affina le regioni proposte da P-Net, eliminando falsi positivi e migliorando la precisione.
- **O-Net (Output Network)**: fornisce le coordinate finali delle bounding box dei volti e localizza i punti di riferimento (occhi, naso, bocca).

Questa struttura a cascata consente di ridurre il numero di falsi positivi, migliorando l'accuratezza complessiva del modello, poiché ogni rete successiva perfeziona i risultati della precedente.

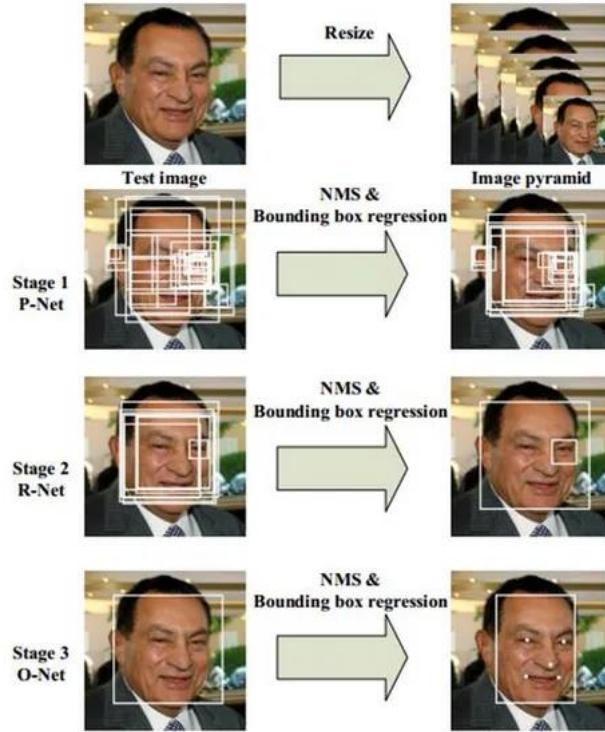


Figura 13: funzionamento di MTCNN

Il sistema include due server HTTP: uno eseguito su un PC, necessario per l'esecuzione dei moduli biometrici, e uno su Raspberry Pi per la commessione con l'applicazione. Entrambi i server sono realizzati utilizzando **Flask**, un framework web leggero per Python, che gestisce le richieste HTTP (come GET, POST, PUT, DELETE) e può restituire risposte al client.

Per elaborare le immagini durante il preprocessing, viene utilizzata anche **OpenCV**, la libreria open-source per la visione artificiale probabilmente più celebre e comune.

Inoltre, si utilizza **NumPy**, che consente di creare array multidimensionali e di sfruttare una vasta gamma di funzioni matematiche avanzate, e **Matplotlib**, che permette la visualizzazione sotto forma di grafici dei risultati ottenuti dalla valutazione dei vari moduli.

Andiamo ora ad esplorare le caratteristiche dei dataset utilizzati durante il progetto.

Per la valutazione del modulo di face detection è stato creato manualmente un dataset contenente istanze della classe da identificare, in questo caso facce umane (**esempi positivi**), e istanze di immagini che non contengono alcun oggetto della classe ma potrebbero causare un errore (**esempi negativi**). Le classi sono state denominate “**face**” e “**no face**”. Complessivamente, il dataset contiene 6000 immagini, 3000 per la classe “**face**” e 3000 per la classe “**no face**”.

Gli esempi negativi sono stati prelevati randomicamente da un dataset contenente immagini di animali (cani, gatti, elefanti, cavalli e leoni) e da uno contenente palle di diversi sport, come football americano, baseball, biliardo, bowling, calcio, golf, hockey, rugby, ping-pong, tennis, pallavolo e basket.

Questi dataset sono disponibili su Kaggle [\[1\]](#) [\[2\]](#).



Figura 13: alcune immagini della classe “no face”

L’idea di includere immagini di questo tipo nella classe ‘no face’ nasce dal desiderio di mettere alla prova il modello, presentandogli immagini che potrebbero ingannarlo e generare **falsi positivi**.

Nella classe ‘face’ sono state incluse immagini provenienti dagli altri tre dataset utilizzati in questo progetto, ovvero **Labelled Faces in the Wild (LFW)**, **NUAA Photograph Imposter Database** e **MSU-MFSD**. In particolare, sono state selezionate casualmente 1000 immagini da ciascuno dei dataset menzionati (per gli ultimi due, sono state selezionate solo immagini di volti autentici).

Per allenare e valutare il modulo di liveness detection sono stati utilizzati due dataset:

- **NUAA Photograph Imposter Database:** è un dataset comunemente utilizzato per la ricerca sull'anti-spoofing in ambito di riconoscimento facciale, in particolare per la rilevazione di print attack. Esso è composto da 15 soggetti, comprendendo un totale di 5.105 immagini di volti genuini e 7.509 *presentation attacks* raccolti tramite una webcam generica a 20 fps con una risoluzione di 640 x 480 pixel. I soggetti sono stati catturati in tre sessioni diverse, in luoghi e condizioni di illuminazione differenti. La produzione dei samples di attacco è stata realizzata scattando una fotografia ad alta risoluzione con una fotocamera digitale Canon.



Figura 14: alcune immagini del dataset NUAA

- **MSU-MFSD:** contiene 280 registrazioni video di volti genuini e di attacchi spoofing. Alla creazione di questo database hanno partecipato 35 individui. Per gli accessi genuini, ciascun individuo ha due video registrati rispettivamente con le telecamere di un laptop e di un dispositivo Android. Sono stati anche ripresi dei video con una Canon e un iPhone che sono stati poi riprodotti su uno schermo iPad Air o sull'iPhone stesso per generare replay attacks HD. I print attack sono stati prodotti stampando le foto dei 35 soggetti su fogli A3 usando una stampante a colori HP. Le registrazioni video dei 35 individui sono state suddivise

rispettivamente in training set (15 soggetti con 120 video) e test set (40 soggetti con 160 video).

Abbiamo estratto tutti i frames dai video che compongono il dataset ottenendo un dataset composto da 57.925 immagini per la classe "attack" e 19.746 immagini per la classe "real".



Figura 15: alcune immagini del dataset MSU

Per valutare il modulo di riconoscimento facciale è stato utilizzato il dataset **Labelled Faces in the Wild (LFW)**. Si tratta di un database di fotografie di volti progettato per lo studio del riconoscimento facciale in condizioni non controllate. Essendo foto scattate in condizioni "naturali", le immagini presentano variazioni in termini di illuminazione, espressioni facciali, età, angoli di ripresa e sfondi, rendendo il riconoscimento facciale una sfida più realistica rispetto a dataset più controllati. Il dataset contiene oltre 13.000 immagini di volti raccolte dal web, con ogni volto etichettato con il nome della persona rappresentata. 1.680 delle persone ritratte hanno due o più foto distinte all'interno del dataset.



Figura 16: alcune immagini del dataset Labelled Faces in the Wild

Per realizzare la parte del sistema che consente di aprire e chiudere un cancello o di inserire e rimuovere i ganci da una porta è chiaramente necessario dell'hardware specifico.

In particolare, il “cervello” che permette ai vari componenti di funzionare come richiesto è il **Raspberry Pi**.

Il Raspberry Pi è un mini-computer a basso costo e dalle dimensioni compatte, sviluppato dalla Raspberry Pi Foundation. È molto apprezzato per la sua versatilità, poiché può essere utilizzato come PC base, server domestico, controller per dispositivi IoT, media center e molto altro. Non dispone di memoria interna, quindi utilizza una scheda microSD come unità di archiviazione principale. È dotato di quattro porte USB, due porte micro HDMI e una porta Ethernet. Include inoltre 40 pin GPIO per controllare dispositivi elettronici esterni e interfacciarsi con sensori e attuatori.

Il modello in nostro possesso è il 4B, nella configurazione con 4 GB di RAM.



Figura 17: Raspberry Pi 4B

Il Raspberry Pi è alimentato tramite un **UPS (Uninterruptible Power Supply)**, che previene danni causati da interruzioni e sbalzi di corrente. L'UPS assicura che, in caso di interruzione dell'alimentazione principale, una batteria o un'altra fonte di energia subentri, sostituendo l'alimentazione principale e permettendo al dispositivo di continuare a funzionare normalmente senza interruzioni. Quando l'alimentazione principale viene ripristinata, l'UPS si ricarica, pronto per la prossima emergenza.



Figura 18: Raspberry Pi UPS

I GPIO del Raspberry Pi vengono utilizzati per stabilire una connessione con il motor driver tramite dei jumper wires. Il driver che abbiamo scelto è L298N, ampiamente utilizzato in progetti di elettronica e robotica per controllare motori DC (a corrente continua). Ha vari pin per connettersi a microcontrollori come Arduino, a Raspberry Pi e a motori.

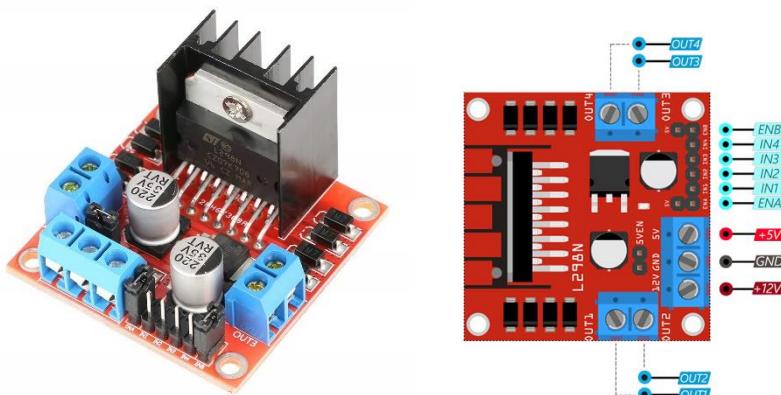


Figura 19: L298N

Un **attuatore lineare** è un dispositivo che trasforma l'energia in movimento lineare, cioè un movimento in una direzione rettilinea. Il nostro è un attuatore lineare 12 volt DC che utilizza una tensione di 12 volt in corrente continua (DC) per funzionare. Per questo è necessario anche un alimentatore che fornisce 12 volt in corrente continua.

In generale l'attuatore lineare è collegato ai pin OUT1 e OUT2 di L298N mostrati in figura 19. L'alimentatore è collegato al driver attraverso i pin +12V e GND. Quest'ultimo è un pin di massa comune e deve essere collegato a 0V.

Un jumper wire deve collegare il GND del driver anche a un pin GND del Raspberry Pi, per garantire un riferimento di massa comune tra i dispositivi. I collegamenti rimanenti riguardano i pin ENA, IN1, e IN2 del L298N, che controllano rispettivamente la velocità e la direzione dell'attuatore lineare. Questi pin sono collegati ai GPIO 22, 17, e 27 del Raspberry Pi per gestire il controllo completo dell'attuatore.

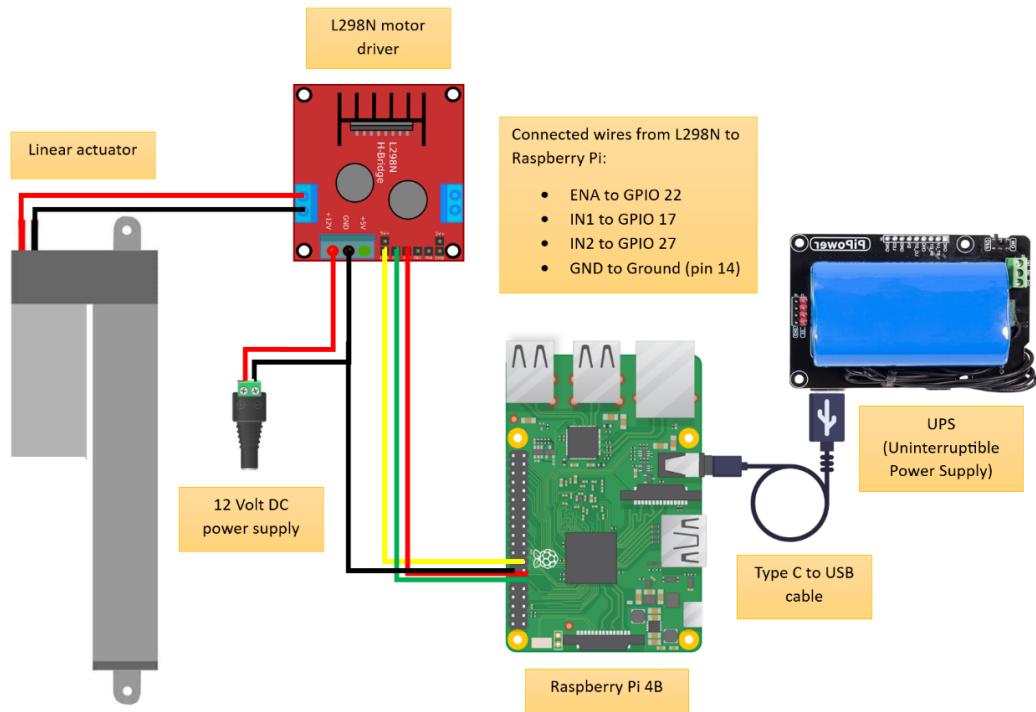


Figura 20: diagramma che riassume tutti i collegamenti

Valutazione del modulo di face detection

Come accennato nel capitolo precedente, nel modulo di face detection abbiamo deciso di utilizzare MTCNN (Multi-task Cascaded Convolutional Neural Networks). Questo modulo sarà integrato all'interno di un'applicazione mobile che opererà in contesti con possibili variazioni ambientali, quindi deve dimostrare un'elevata accuratezza e robustezza nella rilevazione dei volti.

Nel rilevare i volti, ci concentreremo soprattutto su quelli posizionati nella parte centrale e in primo piano delle immagini. Una volta rilevata la faccia, il modulo estrarrà la regione corrispondente ritagliandola attorno al bounding box predetto, con un margine di 14 pixel, per assicurare che il volto venga completamente incluso.

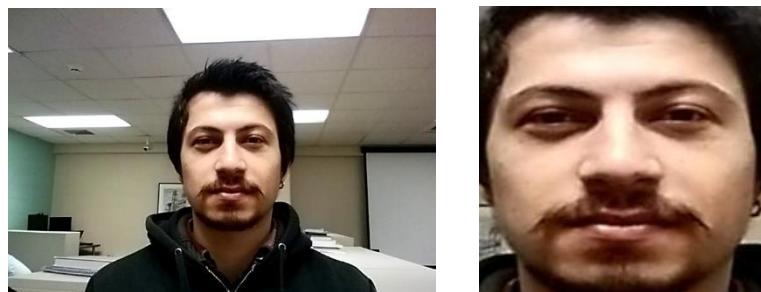


Figura 21: stessa immagine prima e dopo l'applicazione di MTCNN

Come descritto in precedenza, il dataset che abbiamo creato manualmente per valutare MTCNN è composto da immagini organizzate in due cartelle: 'face' e 'no face'. Le immagini nella cartella 'face' rappresentano la classe positiva (classe 1), mentre quelle nella cartella 'no face' rappresentano la classe negativa (classe 0). Abbiamo popolato la cartella 'no face' con immagini di animali (cani, gatti, leoni, cavalli, elefanti) e oggetti di forma rotonda o ovale (palloni, sfere), per introdurre elementi che potessero confondere il modello, simulando strutture simili al volto umano.

Abbiamo quindi valutato MTCNN utilizzando diverse metriche:

- **False Positives:** numero di immagini senza volto in cui il modello ha rilevato erroneamente un volto.
- **Missed Faces:** numero di immagini con un volto presente in cui il modello non ha rilevato alcun volto.

- **True Positives**: numero di immagini con volto in cui il modello ha correttamente rilevato un volto.
- **True Negatives**: numero di immagini senza volto in cui il modello ha correttamente indicato l'assenza di un volto.

Con queste informazioni abbiamo calcolato:

- **False Positive Rate**: percentuale di immagini senza volto classificate erroneamente come contenenti un volto.
- **Missed Face Rate**: percentuale di immagini con un volto in cui il modello non ha rilevato alcun volto.

Prima di arrivare alla valutazione vera e propria, abbiamo effettuato una fase di preprocessing dei dati, organizzando le immagini in un DataLoader con batch size uguale a 16. All'interno del DataLoader, ogni immagine è associata alla rispettiva etichetta: 1 per le immagini contenenti un volto (classe "face") e 0 per quelle senza volto (classe "no face").

Per ogni batch di immagini, è stato applicato MTCNN per rilevare i volti. Il modulo analizza ogni immagine e restituisce il bounding box del volto rilevato. Se il modello non rileva alcun volto, restituisce **None**. L'interpretazione di questo risultato dipende dall'etichetta associata all'immagine:

- Se il bounding box è **None** e l'etichetta è 1 (quindi l'immagine contiene un volto), abbiamo una **missed detection**. Il modello non ha individuato il volto presente, quindi viene incrementato il conteggio delle missed faces.
- Se il bounding box è **None** e l'etichetta è 0 (quindi l'immagine non contiene un volto), abbiamo un **true negative**, poiché il modello ha correttamente rilevato l'assenza di volti.

Nel caso in cui MTCNN restituisca un bounding box, ossia il modello ha rilevato un volto, ci sono due possibilità:

- Se l'etichetta associata è 0 (non dovrebbe esserci un volto), si tratta di un **false positive**, poiché il modello ha erroneamente individuato un volto dove non c'è.
- Se l'etichetta è 1 (quindi c'è effettivamente un volto), si tratta di un **true positive**, poiché il modello ha correttamente rilevato il volto.

I risultati dell'applicazione di MTCNN sulle 3000 immagini della classe "face" e sulle 3000 immagini della classe "no face" sono stati utilizzati per calcolare i seguenti risultati:

- False Positive Rate: 2,47%
- Missed Face Rate: 0,00%

Il Missed Face Rate pari a 0,00% dimostra che il modello non ha fallito nella rilevazione di alcun volto umano nelle immagini della classe "face". Il False Positive Rate di 2,47%, seppur basso, riflette l'inclusione di immagini volutamente complicate nella classe "no face", come quelle di animali o oggetti sferici che possono somigliare a un volto umano.

Quelle mostrate in figura 22 sono alcune delle immagini che il modello classifica erroneamente come volti umani: nel primo caso, l'errore è probabilmente legato al fatto che la forma triangolare può ricordare i contorni di un volto, e le palline possono ricordare la zona degli occhi; nel secondo caso possiamo notare come i volti di alcune razze di cani possano richiamare il volto umano, come il cane in foto, che può ricordare un uomo anziano con la barba.



Figura 22: alcuni false positive

Abbiamo anche calcolato la confusion matrix dei risultati ottenuti:

- 2926 immagini della classe "no face" sono state correttamente classificate come senza volto.
- 0 immagini della classe "face" sono state erroneamente classificate come senza volto.
- 74 immagini della classe "no_face" sono state erroneamente classificate come contenenti un volto.

- 3000 immagini della classe "face" sono state correttamente classificate come contenenti un volto.

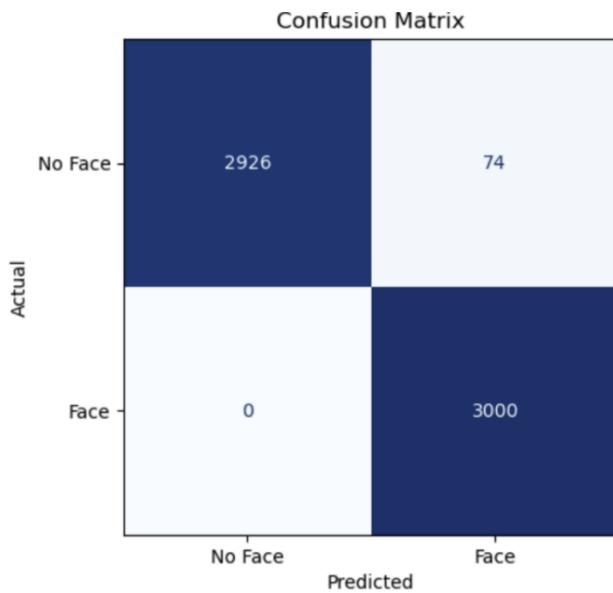


Figura 23: confusion matrix

Questa matrice conferma che il modello ha un'elevata precisione nel rilevamento delle facce, con una perfetta identificazione delle immagini contenenti volti e una bassa percentuale di falsi positivi. In conclusione, MTCNN si è dimostrato un modello robusto ed efficace per il nostro scenario applicativo.

Addestramento e valutazione del modulo di antispoofing

La presenza di un modulo di antispoofing nel nostro sistema mira a prevenire le tipologie di presentation attack più comuni: i **print attack** e i **replay attack**.

Prima di addestrare il modulo di liveness detection e valutarne le prestazioni, è stato effettuato un preprocessing sui due dataset utilizzati, MSU-MFSD e NUAA Photograph Imposter Database. Per velocizzare il processo di addestramento e valutazione del modello, tutte le immagini sono state prima convertite in NumPy array, facilitandone la successiva conversione in tensori di PyTorch. Prima di questa conversione, ogni immagine è stata ritagliata a una dimensione di 224x224 pixel utilizzando MTCNN. MTCNN ci ha permesso di estrarre solo la regione di interesse del volto, eliminando lo sfondo. Ciò consente al modello di distinguere i volti reali dai presentation attack esclusivamente sulla base delle caratteristiche facciali e di texture, evitando elementi di bias.

Un ulteriore problema affrontato è stato lo sbilanciamento del dataset MSU-MFSD, che contiene molte più immagini di spoof rispetto a quelle di volti reali. Per risolvere questo problema, sono state selezionate randomicamente 19.000 immagini di volti reali e 19000 di volti spoof, per un totale di 38000 istanze, creando così un dataset bilanciato di dimensioni (38000, 224, 224, 3). A ciascuno dei due NumPy array, X_{nuaa} e X_{msu} , è stato poi associato un altro array che contiene le rispettive etichette, S_{nuaa} e S_{msu} , di dimensioni (38.000,). A ogni immagine è stata assegnata l'etichetta 1 se appartiene alla categoria spoof, e 0 se rappresenta un volto reale.

A questo punto, abbiamo addestrato MobileNet_v2 con e senza il preprocessing delle immagini effettuato tramite LBP.

MobileNet_v2 è un modello pre-addestrato, al quale abbiamo modificato l'ultimo layer per adattarlo alla classificazione binaria tra immagini reali e spoof. L'ultimo layer della rete è stato riaddestrato sui nostri dati per 10 epoch, con un batch size di 32 e un learning rate di 0,0001. Abbiamo deciso di sperimentare l'uso di LBP (Local Binary Pattern) come metodo di preprocessing poiché enfatizza i dettagli delle texture e può individuare difetti

nella qualità di stampa degli artefatti fotografici usati nei print attack, che riflettono la luce e presentano proprietà superficiali (ad esempio colori) differenti rispetto a quelle di un volto reale.

Sono stati effettuati i seguenti esperimenti:

- MobileNet_v2 è stata addestrata sul dataset NUAA Photograph Imposter Database e successivamente testata su MSU-MFSD.
- MobileNet_v2 è stata addestrata sul dataset MSU-MFSD e poi testata su NUAA Photograph Imposter Database.
- Le immagini sono state preprocessate con LBP, quindi MobileNet_v2 è stata addestrata sul dataset NUAA Photograph Imposter Database e testata su MSU-MFSD.
- Le immagini sono state preprocessate con LBP, quindi MobileNet_v2 è stata addestrata sul dataset MSU-MFSD e testata su NUAA Photograph Imposter Database.
- MobileNet_v2 è stata addestrata su una combinazione dei due dataset e testata su una partizione di immagini non viste durante la fase di addestramento, sempre provenienti dalla combinazione dei due dataset.
- Le immagini sono state preprocessate con LBP, quindi MobileNet_v2 è stata addestrata su una combinazione dei due dataset e testata su una partizione di immagini non viste durante la fase di addestramento, sempre provenienti dalla combinazione dei due dataset.

Nei primi quattro esperimenti, il modello è stato testato anche sul test set del dataset con cui era stato addestrato, oltre a svolgere una valutazione **cross-dataset**.

Per valutare il sistema sono state utilizzate le seguenti metriche:

- **APCER (Attack Presentation Classification Error Rate)**: è la proporzione di immagini spoof classificate erroneamente come reali.
È calcolato con la formula $\frac{FP}{(FP+TN)}$.
- **BPCER (Bona Fide Presentation Classification Error Rate)**: è la proporzione di immagini reali classificate erroneamente come spoof.
È calcolato con la formula $\frac{FN}{(FN+TP)}$.
- **ACER (Average Classification Error Rate)**: è la media tra APCER e BPCER.

- **Accuracy**: è la proporzione di predizioni corrette sul totale delle predizioni.
- **Recall o True Positive Rate**: indica la capacità del modello di identificare correttamente i volti reali. È calcolato con la formula $\frac{TP}{(TP+FN)}$.
- **Precision**: indica la precisione del modello nell'identificare i volti reali tra tutte le predizioni di volti reali. È calcolato con la formula $\frac{TP}{(TP+FP)}$.
- **F1-Score**: è la media armonica tra precisione e recall.

```

def test_model.loaded_model, test_loader):
    threshold = 0.5
    results = []

    with torch.no_grad():
        corrects = 0
        total = 0
        TP = FP = TN = FN = 0

        for images, labels in test_loader:
            images, labels = images.to(device), labels.float().to(device).unsqueeze(1)

            outputs = loaded_model(images)
            preds = torch.sigmoid(outputs) > threshold

            TP += ((preds == 0) & (labels == 0)).sum().item()
            TN += ((preds == 1) & (labels == 1)).sum().item()
            FP += ((preds == 0) & (labels == 1)).sum().item()
            FN += ((preds == 1) & (labels == 0)).sum().item()

            corrects += (preds == labels).sum().item()
            total += labels.size(0)

            APCER = FP / (FP + TN) if (FP + TN) > 0 else 0
            BPCER = FN / (FN + TP) if (FN + TP) > 0 else 0
            ACER = (APCER + BPCER) / 2
            accuracy = corrects / total
            recall = TP / (TP + FN) if (TP + FN) > 0 else 0
            precision = TP / (TP + FP) if (TP + FP) > 0 else 0
            f1_score = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

            results.append({
                'accuracy': accuracy,
                'APCER': APCER,
                'BPCER': BPCER,
                'ACER': ACER,
                'precision': precision,
                'recall': recall,
                'f1_score': f1_score,
                'TP': TP,
                'TN': TN,
                'FP': FP,
                'FN': FN
            })

    print(f'APCER: {APCER:.2f}, BPCER: {BPCER:.2f}, ACER: {ACER:.2f}')
    print(f'Accuracy: {accuracy:.2f}, Precision: {precision:.2f}, Recall: {recall:.2f}, F1-Score: {f1_score:.2f}')

    return loaded_model, results, TP, TN, FP, FN

```

Figura 24: codice che calcola le metriche

Quando si implementa un modulo di antispoofing, l'obiettivo principale è mantenere il più bassi possibile i valori di APCER, BPCER e ACER. Questo perché desideriamo che il sistema sia in grado di distinguere correttamente tra un utente genuino e un impostore, evitando sia di rigettare erroneamente un utente legittimo sia di accettare un falso positivo.

Nel primo esperimento, abbiamo allenato MobileNet esclusivamente sulle immagini del dataset NUAA, per poi valutare le prestazioni del modello su una porzione dello stesso dataset, utilizzando istanze che il modello non aveva mai visto durante l'addestramento. I risultati ottenuti sono stati eccellenti: le metriche classiche, tra cui l'accuracy, non hanno mostrato segni di overfitting nelle 10 epoche, dimostrando che il modello è in grado di generalizzare perfettamente su dati mai visti del medesimo dataset. Anche le metriche specifiche per l'antispoofing, come APCER, BPCER e ACER, hanno mostrato valori praticamente pari a zero, il che significa che ogni immagine è stata correttamente classificata. Per valutare la robustezza del modello allenato su NUAA, abbiamo eseguito una valutazione cross-dataset utilizzando il dataset MSU. Dopo aver calcolato le metriche abbiamo notato un significativo calo delle prestazioni rispetto alla valutazione su NUAA.

In seguito, abbiamo allenato nuovamente MobileNet, questa volta sul dataset MSU, valutandolo su una porzione di immagini che il modello non aveva visto durante l'addestramento. Anche in questo caso, i risultati sono stati eccellenti per tutte le metriche di valutazione. Tuttavia, quando abbiamo testato il modello su NUAA, abbiamo riscontrato un calo delle prestazioni, simile a quello osservato nell'esperimento precedente.

Per migliorare le prestazioni, abbiamo deciso di sperimentare l'uso di LBP. Dopo aver applicato LBP a tutte le immagini del dataset, abbiamo allenato MobileNet su NUAA e testato le sue prestazioni su istanze non viste del medesimo dataset. Anche in questo caso, i risultati sono stati eccellenti. Quando abbiamo eseguito la valutazione cross-dataset su MSU, abbiamo riscontrato un miglioramento del 10% in tutte le metriche calcolate, sebbene le prestazioni fossero ancora insufficienti per l'uso in contesti reali (valori di APCER e BPCER troppo alti e accuracy del 62%).

Sempre dopo aver applicato LBP, abbiamo addestrato MobileNet su immagini di MSU. Anche in questo caso, le prestazioni del modello su immagini non viste

del dataset su cui è addestrato sono state ottime, ma sono diminuite quando valutate su NUAA, sebbene con miglioramenti rispetto agli esperimenti precedenti.

L'andamento della valutazione cross-dataset era prevedibile: i due dataset presentano differenze significative in termini di illuminazione, tipologia di persone e dispositivi utilizzati per la cattura delle immagini (smartphone, tablet, laptop). Ad esempio, NUAA contiene prevalentemente immagini di persone di origine asiatica, mentre MSU presenta una maggiore varietà demografica. Inoltre, le immagini di MSU provengono da frame estratti dai video originali che componevano il dataset, mentre quelle di NUAA no.

Le discrepanze si sono ulteriormente confermate quando abbiamo testato i modelli in real-time, dove le prestazioni tendevano a seguire le stesse fluttuazioni osservate nelle valutazioni. Anche una minima variazione nelle condizioni ambientali portava il modello a commettere errori.

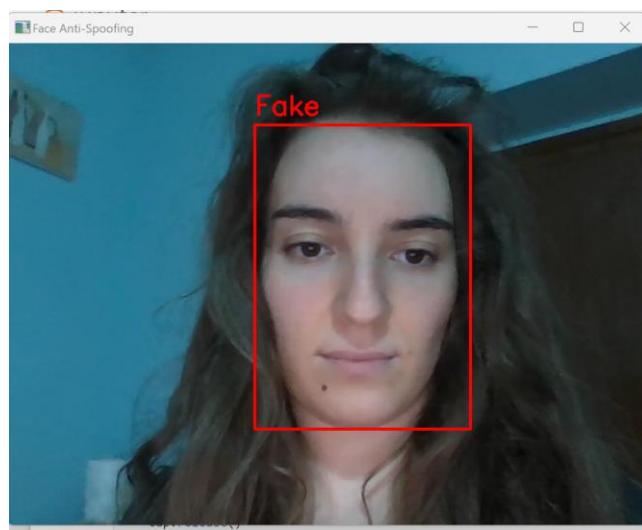


Figura 25: test in real-time del modello allenato su MSU senza preprocessing con LBP

Infine, date le differenze intrinseche tra i due dataset, abbiamo deciso di combinarli, cercando di catturare il maggior numero possibile di variazioni ambientali. Abbiamo quindi addestrato MobileNet senza il preprocessing con LBP utilizzando il dataset combinato e lo abbiamo testato su una porzione di dati non visti, provenienti da entrambi i dataset. I risultati ottenuti, come mostrano i grafici e le metriche calcolate, hanno raggiunto punteggi elevati, dimostrando che la combinazione dei due dataset ha migliorato

significativamente la capacità del modello di generalizzare su dati eterogenei, rendendolo più robusto e adatto a situazioni reali.

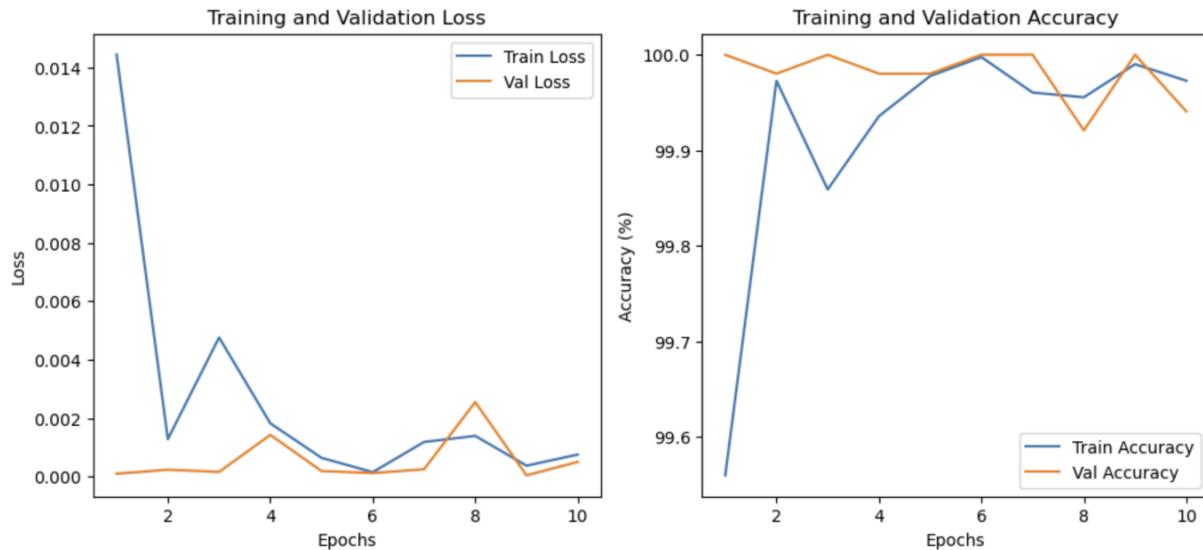


Figura 26: Evoluzione di training e validation loss ed evoluzione di training e validation accuracy al variare delle epoche

Abbiamo ottenuto risultati eccellenti anche per quanto riguarda le metriche APCER, BPCER e ACER, tutte con valori pari a zero.

APCER: 0.00, BPCER: 0.00, ACER: 0.00
 Accuracy: 1.00, Precision: 1.00, Recall: 1.00, F1-Score: 1.00

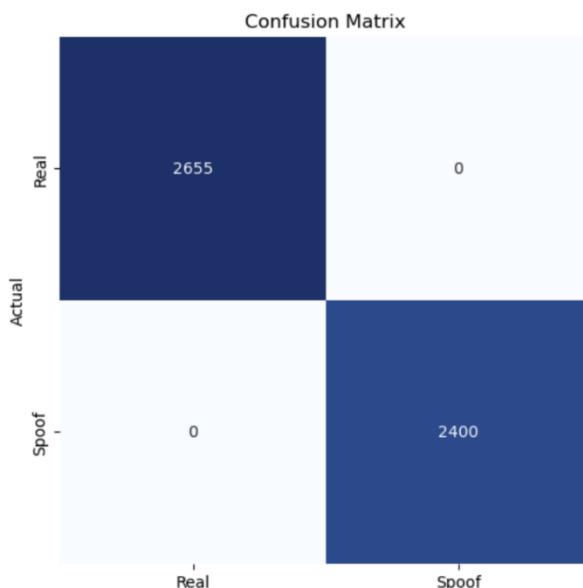


Figura 27: stampa dei valori delle metriche e confusion matrix

Per completare il quadro, abbiamo allenato MobileNet v2 sulle immagini del nuovo dataset combinato preprocessato con LBP. Anche in questo caso, i risultati sono stati quasi perfetti, con valori di loss sia in fase di addestramento che di test vicini allo zero, dati che dimostrano l'efficacia del modello.

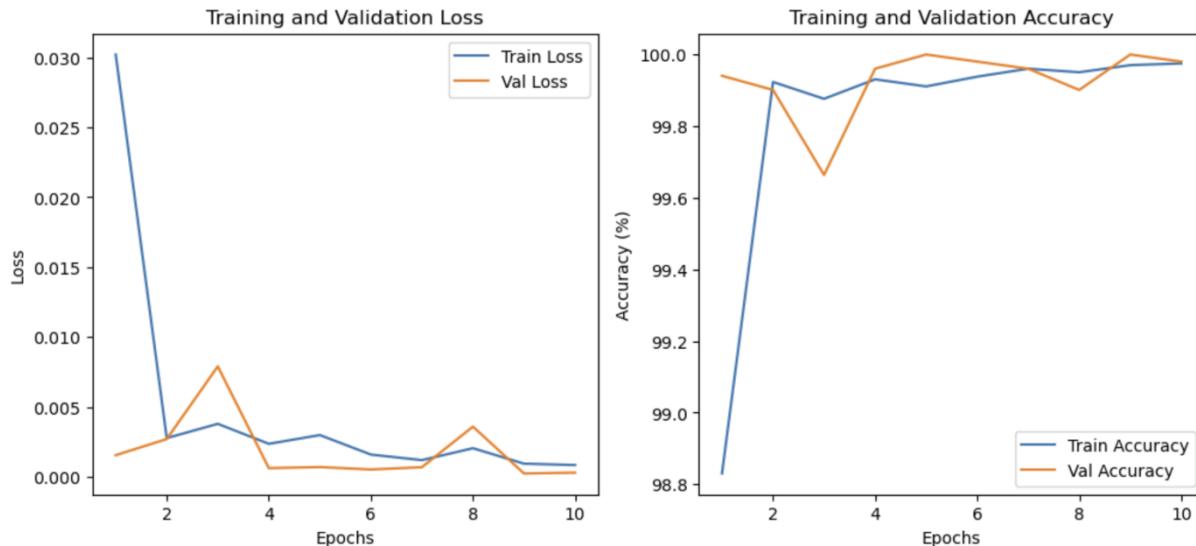


Figura 28: stampa del valore delle metriche e confusion matrix

Analizzando la confusion matrix, abbiamo osservato che il modello ha commesso un solo errore su 5055 tentativi, classificando erroneamente un volto genuino come spoof.

```
APCER: 0.00, BPCER: 0.00, ACER: 0.00
Accuracy: 1.00, Precision: 1.00, Recall: 1.00, F1-Score: 1.00
```

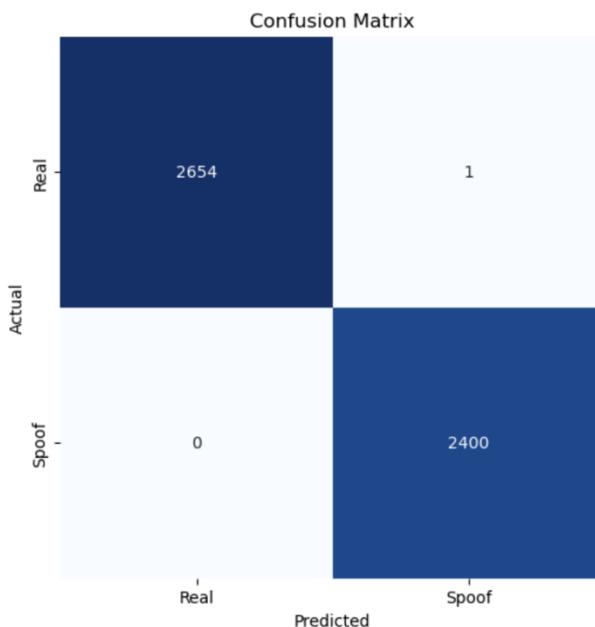


Figura 29: stampa del valore delle metriche e confusion matrix

Abbiamo quindi eseguito dei test in tempo reale su questi nuovi modelli. In particolare, il modello di MobileNet preprocessato con LBP e allenato sul dataset combinato ha mostrato una maggiore robustezza alle variazioni di posa ed espressione: è in grado di mantenere buone prestazioni anche con il volto inclinato o ruotato. Tuttavia, questo modello è risultato meno preciso nel distinguere tra volti genuini e spoof, con un aumento dei falsi positivi, ovvero casi in cui un utente impostore viene classificato erroneamente come real.

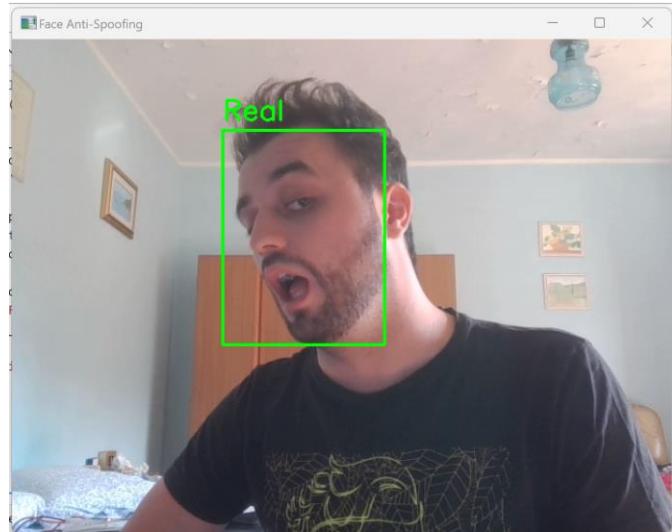


Figura 29: il modello preprocessato con LBP è più resistente alle variazioni di posa ed espressione ma meno ottimale nell'individuazione dei volti fake

D'altro canto, il modello allenato sul dataset combinato senza l'uso del preprocessing LBP si è dimostrato più sensibile alle variazioni di posa, ma molto più accurato nella distinzione tra volti genuini e spoof. Entrambi i modelli, però, hanno mostrato una certa vulnerabilità alle variazioni di illuminazione, soprattutto in condizioni di luce molto intensa, come in caso di vicinanza a una lampada o in presenza di aree del volto illuminate e altre in ombra. Le performance peggiorano leggermente al tramonto, mentre le migliori prestazioni si ottengono con luce naturale, anche se la luce artificiale uniforme durante il giorno genera comunque buoni risultati.

Dopo queste considerazioni, abbiamo deciso di dare priorità alla capacità di distinguere con precisione le due classi, sacrificando in parte la robustezza alle variazioni di posa, scegliendo di utilizzare nella nostra applicazione il modello MobileNet_v2 addestrato sul dataset combinato senza il preprocessing LBP.

Questo perché, in un'applicazione mobile come la nostra, è generalmente richiesta una ripresa frontale del volto, senza grosse variazioni di posa.

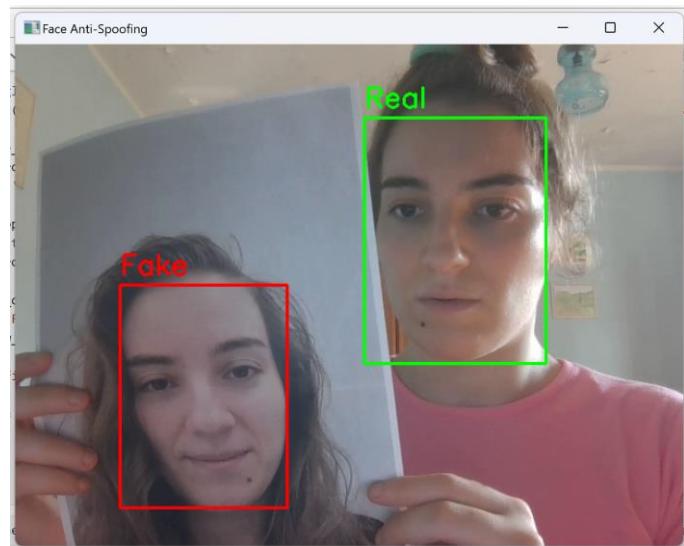


Figura 30: test in real-time del modello scelto per l'applicazione

Valutazione del modulo di face recognition

Il terzo modulo biometrico che compone la nostra pipeline è quello di riconoscimento facciale. L'uso della libreria FaceNet-PyTorch ci consente di sfruttare la potenza della GPU per eseguire operazioni complesse come il calcolo della matrice ALL-against-ALL. Questa scelta ha permesso una drastica riduzione dei tempi di calcolo, rendendo possibili un maggior numero di esperimenti e ottimizzando le prestazioni complessive.

Come già accennato, l'architettura di base utilizzata da FaceNet-PyTorch nella sua implementazione è Inception ResNet (V1), pre-addestrata sui dataset VGGFace2 e CASIA-WebFace. Per ottenere una valutazione accurata del nostro sistema abbiamo optato per una valutazione cross-dataset utilizzando il celebre dataset Labelled Faces In The Wild (LFW).

Le immagini sono state pre-elaborate utilizzando MTCNN, che ha eseguito la face detection applicando un margine di 14 pixel, catturando completamente le caratteristiche del volto. Successivamente, le immagini sono state ritagliate attorno al bounding box del volto e ridotte alla dimensione 224x224. Abbiamo utilizzato il parametro *selection_method* = "center_weighted_size" per garantire la selezione del volto più centrale e prominente.

Una volta elaborate le immagini con MTCNN, abbiamo organizzato il dataset in un DataLoader di tensori float, suddivisi in piccoli batch da 16. Ad ogni immagine è stata assegnata un'etichetta univoca (un intero crescente da 0 al numero totale di identità differenti, pari a 5749), derivata dal nome della cartella originale da cui l'immagine è stata estratta. Così, due immagini provenienti dalla stessa cartella hanno ricevuto la stessa etichetta.

Successivamente, abbiamo applicato la *fixed_image_standardization*: ogni canale dell'immagine (R, G, B) è stato normalizzato sottraendo un valore medio fisso di 127.5, con successiva divisione dei pixel per 128, standardizzando i valori tra -1 e 1. Questo tipo di normalizzazione è cruciale per stabilizzare le prestazioni del modello durante l'addestramento e l'inferenza.

A questo punto, abbiamo caricato il modello InceptionResNetV1 per ottenere soltanto gli embedding delle immagini, senza effettuare classificazioni.

Grazie alla funzione `create_labels_and_embeddings()` mostrata in figura 31, ogni immagine nel dataloader è stata processata dal modulo di riconoscimento facciale, generando un embedding di 512 caratteristiche.

```
# Extract embeddings and labels from the dataset using the InceptionResnetV1 model
def create_labels_and_embeddings(resnet, embed_loader, device):
    labels = [] # List to store labels (subject IDs)
    embeddings = [] # List to store embeddings (feature vectors)
    resnet.eval() # Set the model to evaluation mode
    with torch.no_grad():
        for xb, yb in embed_loader: # Iterate over batches of images and labels
            xb = xb.to(device)
            b_embeddings = resnet(xb) # Extract embeddings using the ResNet model
            b_embeddings = b_embeddings.to('cpu').numpy()
            labels.extend(yb.numpy()) # Store the labels for the current batch
            embeddings.append(b_embeddings) # Store the embeddings for the current batch
    # Convert lists of labels and embeddings to PyTorch tensors and move to GPU
    labels = torch.tensor(labels).cuda()
    embeddings = torch.tensor(embeddings).cuda()
    return labels, embeddings
```

Figura 31: funzione `create_labels_and_embeddings()`

Abbiamo condotto 8 esperimenti differenti per valutare il modulo di Face Recognition utilizzando sia il modello pre-addestrato su CASIA-Webface sia quello su VGGFace2. La funzione `select_identities_with_S_templates(labels, S)`, mostrata in figura 32, seleziona il numero desiderato di template per ogni soggetto, fondamentale per il calcolo della matrice ALL-against-ALL nel caso di verification *single-template*.

```
# Select subjects with exactly S templates for evaluation
def select_identities_with_S_templates(labels, S):
    selected_indices = [] # List to store the selected image indices
    select_labels_list = [] # List to store the corresponding labels
    total_labels_number = torch.max(labels).item() + 1 # Get the number of unique subjects
    distinct_labels = torch.arange(total_labels_number).cuda() # Create a tensor of all label values

    # Iterate through all subjects
    for label in distinct_labels:
        indices = torch.where(labels == label.item())[0] # Find the indices for the current label
        if len(indices) < S: # If there are fewer than S images, skip to the next label
            continue
        if len(indices) >= S: # If there are at least S images
            # Randomly select S indices from the available ones
            selected_indices.extend(indices[torch.randperm(len(indices))[:S]].tolist())
            select_labels_list.extend([label] * S) # Assign the label S times to the selected images

    # Convert the list into a tensor on CUDA
    selected_indices_tensor = torch.tensor(selected_indices, device='cuda')
    select_labels_tensor = torch.tensor(select_labels_list, device='cuda')
    # Extract the embeddings for the selected images
    selected_embeddings = embeddings[selected_indices_tensor].cuda()
    return selected_indices_tensor, select_labels_tensor, selected_embeddings
```

Figura 32: funzione `select_identities_with_S_templates()`

Ad esempio, impostando $S=2$, la funzione seleziona solo le identità con almeno due embeddings e ne prende esattamente due a caso per ciascuna identità. Nello specifico sono stati eseguiti i seguenti esperimenti con diversi valori di S :

- **$S=2$** : sono stati presi in considerazione 1680 soggetti, ciascuno con 2 embeddings, per un totale di 3360 embeddings.
- **$S=3$** : sono stati presi in considerazione 901 soggetti, ciascuno con 3 embeddings, per un totale di 2703 embeddings.
- **$S=9$** : sono stati presi in considerazione 184 soggetti, ciascuno con 9 embeddings, per un totale di 1656 embeddings.
- **$S=70$** : sono stati presi in considerazione 7 soggetti, ciascuno con 70 embeddings, per un totale di 490 embeddings.

A questo punto per ciascun valore di S , abbiamo costruito una matrice ALL-against-ALL dove tutti i template vengono confrontati con ogni altro template, calcolando la distanza euclidea tra ogni coppia di embeddings escludendo gli elementi sulla diagonale principale che corrispondono al match tra un template e sé stesso. Le distanze sono state successivamente normalizzate attraverso la min-max normalization, con valori compresi tra 0 e 1.

Nella **verification**, un soggetto viene accettato se e solo se la distanza tra il probe e i template associati all'identità reclamata è minore o uguale a certa soglia (threshold). Quindi si possono presentare 4 output:

- **GA (Genuine Acceptance)**: l'identità reclamata è vera e il soggetto viene accettato dal sistema.
- **FR (False Rejection)**: l'identità reclamata è vera ma il soggetto viene rifiutato dal sistema (errore di tipo 1).
- **FA (False Acceptance)**: l'identità reclamata è falsa e il soggetto viene accettato dal sistema. (errore di tipo 2).
- **FR (Genuine Rejection)**: l'identità reclamata è falsa e il soggetto viene rifiutato dal sistema.

In questo tipo di sistema, l'utente viene considerato un impostore in due scenari:

1. L'utente è registrato ma dichiara un'identità falsa.
2. L'utente non ha effettuato l'enrollment.

Dal punto di vista computazionale, non c'è differenza tra questi due scenari, poiché in entrambi i casi l'utente dichiara un'identità falsa. Al contrario, un soggetto viene definito genuino se l'identità dichiarata corrisponde a quella reale.

Alla luce di queste considerazioni possiamo ricavare le seguenti metriche:

- **GAR (Genuine Acceptance Rate)**: probabilità di utente genuino di essere accettato dal sistema
- **FAR (False Acceptance Rate)**: probabilità di un impostore di essere accettato dal sistema.
- **FRR (False Rejection Rate)**: probabilità di utente genuino di essere rifiutato dal sistema.
- **GRR (Genuine Rejection Rate)**: probabilità di un impostore di essere rifiutato dal sistema.

Inoltre, è stata definita una funzione per calcolare il numero di Total Genuine Attempts (TG) e il numero di Total Impostor Attempts (TI) considerati nella matrice.

```
# Calculate the All-Against-All parameters (number of subjects, genuine and impostor attempts)
def get_ALL_against_ALL_parameters(selected_labels_tensor, selected_embeddings, S):
    N = selected_labels_tensor.size(0)//S # Number of unique subjects
    TG = selected_embeddings.size(0) * (S - 1) # Total number of genuine attempts
    TI = selected_embeddings.size(0) * S * (N - 1) # Total number of impostor attempts
    return N, TG, TI
```

Figura 33: get_ALL_against_ALL_parameters()

Per calcolare la matrice ALL-against-ALL, abbiamo sviluppato una funzione chiamata evaluate_ALL_against_ALL_distance_matrix(). La funzione originale era progettata per confrontare gli embeddings su un insieme di 100 soglie, variabili tra 0 e il valore massimo della matrice delle distanze normalizzate (quindi 1). Per ogni soglia, si iterava su tutte le possibili combinazioni di embeddings (usando itertools.combinations) e si confrontavano le etichette associate agli embeddings con la loro distanza corrispondente. Se la distanza tra due embeddings era inferiore o uguale alla soglia corrente e le etichette corrispondevano, si incrementava il contatore delle Genuine Acceptances (GA). Al contrario, se le etichette non corrispondevano, veniva incrementato il contatore delle False Acceptances (FA). Allo stesso modo, se la distanza tra i due embeddings superava la soglia e le etichette corrispondevano, si

registrava un False Rejection (FR), mentre per due embeddings diversi correttamente rifiutati si contava un Genuine Rejection (GR).

```

def evaluate_ALL_against_ALL_distance_matrix1(normalized_matrix, selected_labels_tensor, TG, TI):
    # Define the thresholds
    normalized_matrix = normalized_matrix.cpu()
    selected_labels_tensor = selected_labels_tensor.cpu()
    thresholds = torch.linspace(0, normalized_matrix.max().item(), steps=100).cpu()

    # Initialize counters for the metrics
    GA = torch.zeros(thresholds.size(), device='cpu') # True Accepts
    FA = torch.zeros(thresholds.size(), device='cpu') # False Accepts
    GR = torch.zeros(thresholds.size(), device='cpu') # True Rejects
    FR = torch.zeros(thresholds.size(), device='cpu') # False Rejects

    for idx, t in enumerate(thresholds):
        for i, j in tqdm(itertools.combinations(range(normalized_matrix.size(0)), 2),
                        desc=f"Threshold {t:.3f}", unit=" pairs", leave=True):
            if i != j:
                if normalized_matrix[i, j] <= t:
                    if selected_labels_tensor[i] == selected_labels_tensor[j]:
                        GA[idx] += 1
                    else:
                        FA[idx] += 1
                else:
                    if selected_labels_tensor[i] == selected_labels_tensor[j]:
                        FR[idx] += 1
                    else:
                        GR[idx] += 1

        GA[idx] = GA[idx] * 2
        FA[idx] = FA[idx] * 2
        FR[idx] = FR[idx] * 2
        GR[idx] = GR[idx] * 2

    GAR = GA / TG # Genuine Acceptance Rate
    FAR = FA / TI # False Acceptance Rate
    FRR = FR / TG # False Rejection Rate
    GRR = GR / TI # Genuine Rejection Rate

    return GAR, FAR, FRR, GRR, GA, FA, FR, GR, thresholds

```

Figura 34: vecchia funzione evaluate_ALL_against_ALL_distance_matrix

Questo codice anche se calcolava correttamente tutte le metriche, aveva un grave problema di efficienza. Ogni esperimento richiedeva circa un giorno intero per essere completato, principalmente a causa della complessità computazionale. Per risolvere questo problema, abbiamo riscritto il codice per sfruttare al massimo la potenza computazionale della nostra GPU Nvidia RTX 4060.

Il nuovo approccio sfrutta in modo ottimale i tensori di PyTorch. Ecco come funziona in dettaglio:

1. **Inizializzazione delle thresholds sulla GPU:** usando lo stesso criterio usato nel primo approccio, vengono generate 100 thresholds tra 0 e 1.

Questa volta però sono create direttamente sulla GPU, utilizzando la funzione `torch.linspace` con l'opzione `device="cuda"`. Questo permette di evitare costosi trasferimenti di dati tra CPU e GPU.

2. **Preparazione delle etichette e delle maschere:** le etichette delle immagini vengono confrontate tra loro in parallelo utilizzando il tensore `selected_labels_tensor` trasformato in una matrice booleana `labels_same`, che rappresenta se due immagini appartengono alla stessa persona. Anche questa operazione avviene interamente sulla GPU. Inoltre, viene generata una **maschera** diagonale (`diag_mask`) per escludere i confronti di un embedding con sé stesso.
3. **Applicazione delle soglie in batch:** anziché confrontare ogni coppia di embeddings per tutte le soglie in un'unica operazione, il nuovo codice elabora le soglie in piccoli batch di dimensione 10, riducendo così la quantità di memoria utilizzata dalla GPU in ogni iterazione. In ogni batch, viene creata una matrice di distanze booleana (chiamata `distances`), che indica per ogni coppia di embeddings se la loro distanza è inferiore o uguale alla soglia corrente. Anche questa operazione sfrutta la potenza parallela della GPU.
4. **Calcolo parallelo delle metriche:** le metriche GA, FA, FR, e GR vengono calcolate direttamente utilizzando operazioni di somma vettoriali (`torch.sum`) sulla GPU. Le metriche finali GAR, FAR, FRR, e GRR vengono poi calcolate dividendo i contatori per i valori totali di TG e TI.

Grazie a questa ottimizzazione, il tempo di esecuzione di un singolo esperimento è stato ridotto drasticamente, da un'intera giornata a pochi minuti.

Analizziamo ora i vari esperimenti effettuati. Per ciascun esperimento, oltre alle metriche indicate in precedenza, sono stati visualizzati grafici che mostrano la variazione di FAR e FRR al variare della soglia (e il punto di Equal Error Rate che si verifica quando $\text{FAR} = \text{FRR}$); la curva ROC (Receiver Operating Characteristic), che rappresenta la probabilità che un soggetto venga correttamente identificato al variare del FAR; la curva logaritmica DET (Detection Error Tradeoff), che rappresenta la probabilità che un soggetto venga erroneamente non identificato al variare del FAR.

```

def evaluate_ALL_against_ALL_distance_matrix(normalized_matrix, selected_labels_tensor, TG, TI, batch_size=10):
    thresholds = torch.linspace(0, normalized_matrix.max().item(), steps=100, device='cuda')

    GA = torch.zeros(len(thresholds), device='cuda')
    FA = torch.zeros(len(thresholds), device='cuda')
    FR = torch.zeros(len(thresholds), device='cuda')
    GR = torch.zeros(len(thresholds), device='cuda')

    labels_same = (selected_labels_tensor.unsqueeze(0) == selected_labels_tensor.unsqueeze(1)).cuda()
    diag_mask = ~torch.eye(normalized_matrix.size(0), device='cuda').bool().cuda()

    for start in range(0, len(thresholds), batch_size):
        end = min(start + batch_size, len(thresholds))
        current_batch = thresholds[start:end]

        distances = (normalized_matrix.unsqueeze(0) <= current_batch.unsqueeze(1).unsqueeze(1)).cuda()

        GA[start:end] = torch.sum(distances & labels_same & diag_mask, dim=(1, 2)).cuda()
        FA[start:end] = torch.sum(distances & ~labels_same & diag_mask, dim=(1, 2)).cuda()
        FR[start:end] = torch.sum(~distances & labels_same & diag_mask, dim=(1, 2)).cuda()
        GR[start:end] = torch.sum(~distances & ~labels_same & diag_mask, dim=(1, 2)).cuda()

    GAR = GA / TG
    FAR = FA / TI
    FRR = FR / TG
    GRR = GR / TI

    return GAR, FAR, FRR, GRR, GA, FA, FR, GR, thresholds

```

Figura 34: vecchia evaluate_ALL_against_ALL_distance_matrix che ottimizza il calcolo su GPU

Nel primo esperimento, abbiamo valutato il modello pre-addestrato su VGGFace2 limitandoci a calcolare gli embeddings per i soggetti con due template ciascuno.

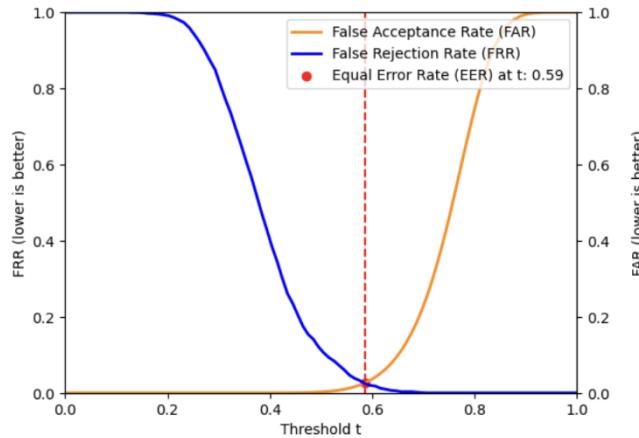


Figura 35: curva che rappresenta la variazione di FAR e FRR al variare della threshold

Come evidenziato dalla figura 35, la threshold in cui si verificano contemporaneamente i migliori valori combinati di FAR e FRR è 0,59. In questo punto, i valori minimi di FAR e FRR sono rispettivamente 0,026 e 0,024, molto vicini allo 0, mentre i valori massimi di GAR e GRR raggiungono

rispettivamente 0,976 e 0,974, molto vicini a 1, indicando un'eccellente capacità di riconoscimento.

La curva ROC conferma questo risultato con un'area sotto la curva (AUC) pari a 1, segno di prestazioni pressoché perfette. Anche la curva DET mostra che sia il FRR che il FAR rimangono bassi, indicando pochi errori.

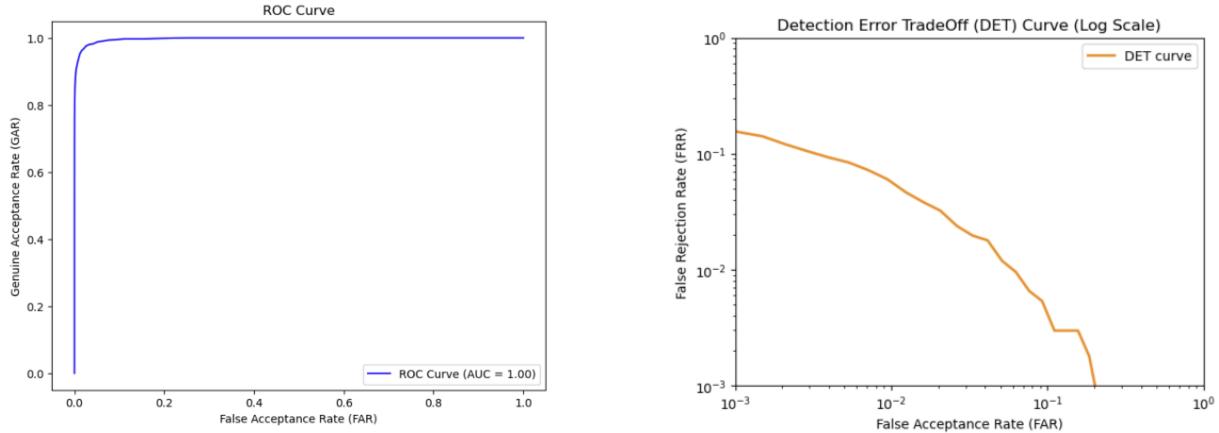


Figura 36: ROC e DET per il modello pre-allenato su VGGFace2 considerando S=2

Successivamente, abbiamo effettuato la valutazione del modello pre-allenato su VGGFace2 per altri valori di S (nello specifico 3, 9, 70). I risultati sono rimasti costanti, con soglie simili per l'EER e metriche comparabili a quelle osservate nel primo esperimento.

Quando abbiamo ripetuto gli stessi esperimenti utilizzando lo stesso modello però pre-addestrato su CASIA-Webface, osservando un calo delle prestazioni su tutti i valori di S presi in considerazione. Il degrado delle performance è stato particolarmente evidente all'aumentare del numero di template per soggetto. I grafici evidenziano chiaramente questo deterioramento: nella figura 37, a sinistra si possono osservare le performance considerando 2 template per soggetto, mentre a destra è mostrato il caso di S=70. Con l'aumento del numero di template, si nota un progressivo incremento dei valori di FAR e FRR, e una riduzione dei valori di GAR e GRR.

In conclusione, il modello addestrato su VGGFace2 ha dimostrato una notevole stabilità, non risentendo significativamente delle variazioni intra-classe e inter-classe al variare del numero di template per soggetto. Al contrario, il modello addestrato su CASIA-Webface ha mostrato performance inferiori per tutti i valori di S presi in considerazione, con

prestazioni progressivamente decrescenti all'aumentare di S. Questo suggerisce che il modello fatica a gestire in modo ottimale le variazioni intraclasse, come cambiamenti di posa, illuminazione ed espressioni per lo stesso soggetto che sono particolarmente comuni in un dataset con immagini catturate in condizioni non controllate, come Labeled Faces in the Wild.

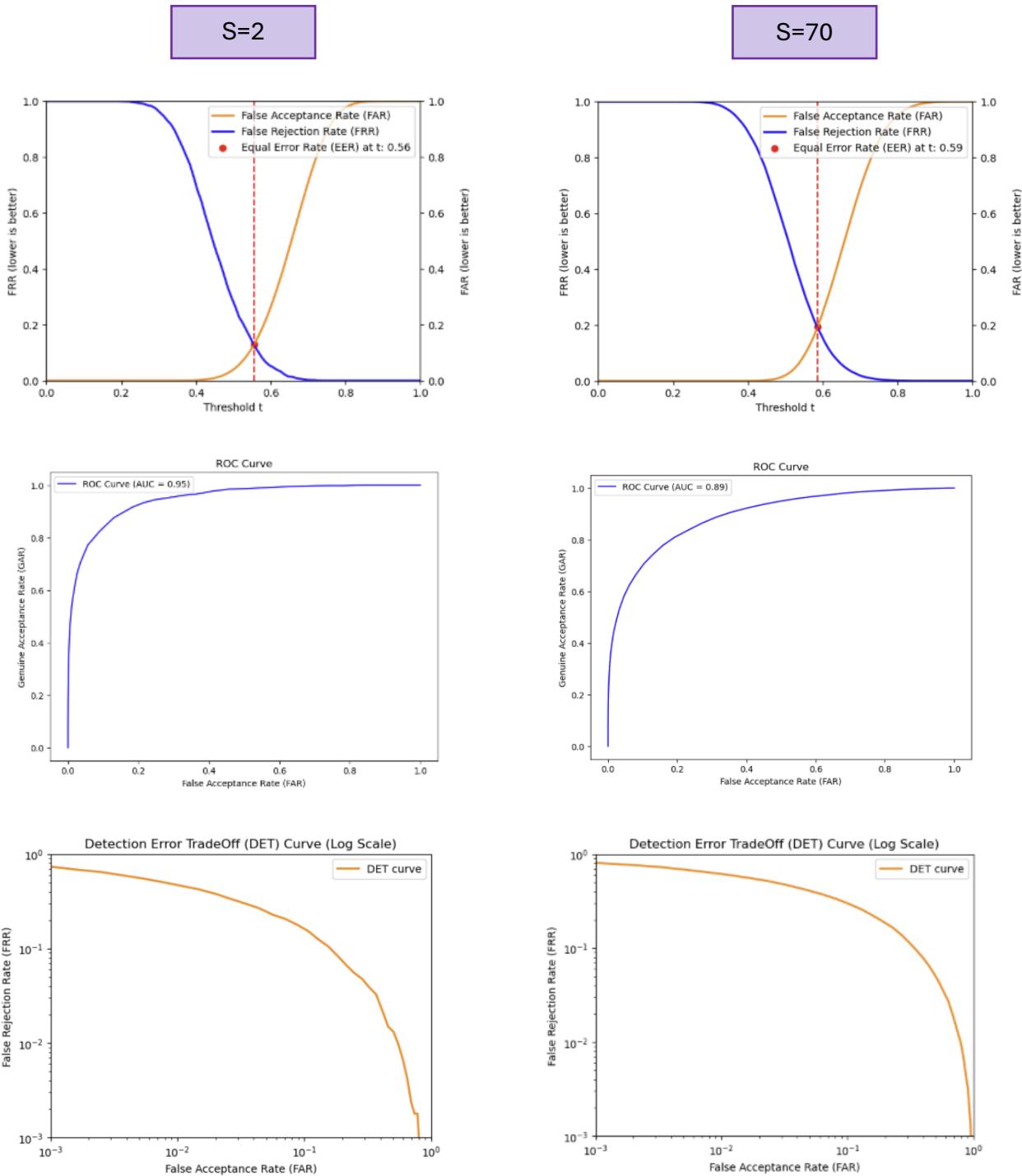


Figura 37: variazione delle figures of merit per S=2 e S=70 per il modello pre-allenato su CASIA-WebFace

Implementazione del sistema

L'applicazione Android sviluppata per questo progetto è composta da quattro activity principali: HomeActivity, RegistrationActivity, VerificationActivity e UserAcceptedActivity.

Al suo avvio, viene mostrata la HomeActivity, in cui nella parte superiore si trova il logo dell'app seguito dal nome dell'applicazione e il suo slogan. Più in basso sono presenti due pulsanti viola, con testo bianco: "Verification" e "Registration", che permettono di navigare rispettivamente verso le activity di verifica e di registrazione. Questi pulsanti consentono all'utente di eseguire i due task principali dell'applicazione: effettuare l'enrollment di un nuovo utente nel sistema e verificare le caratteristiche facciali di una persona per accedere alla possibilità di aprire/chiudere il cancello o bloccare/sbloccare una porta.

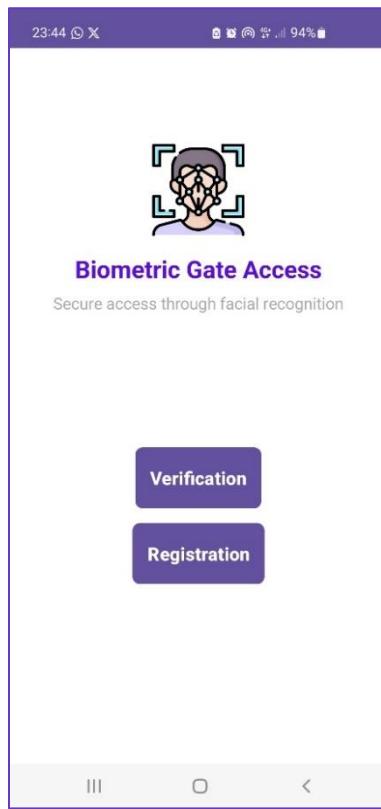


Figura 38: HomeActivity

Premendo il pulsante "Registration" viene avviato il processo di enrollment e si apre RegistrationActivity. Nella parte alta della schermata c'è un messaggio che invita l'utente a scattare tre foto del proprio volto. Al centro della

schermata sono presenti tre riquadri grigi che fungono da placeholder per le foto, mentre in basso si trova il pulsante "Take Pictures", che attiva la fotocamera che può essere cambiata da frontale a posteriore a seconda della preferenza dell'utente. Dopo lo scatto, se la foto non è soddisfacente, è possibile riprovare a catturarla premendo il pulsante "Riprova". Se invece la foto va bene, l'utente può confermarla premendo "OK", e la foto apparirà in uno dei riquadri.

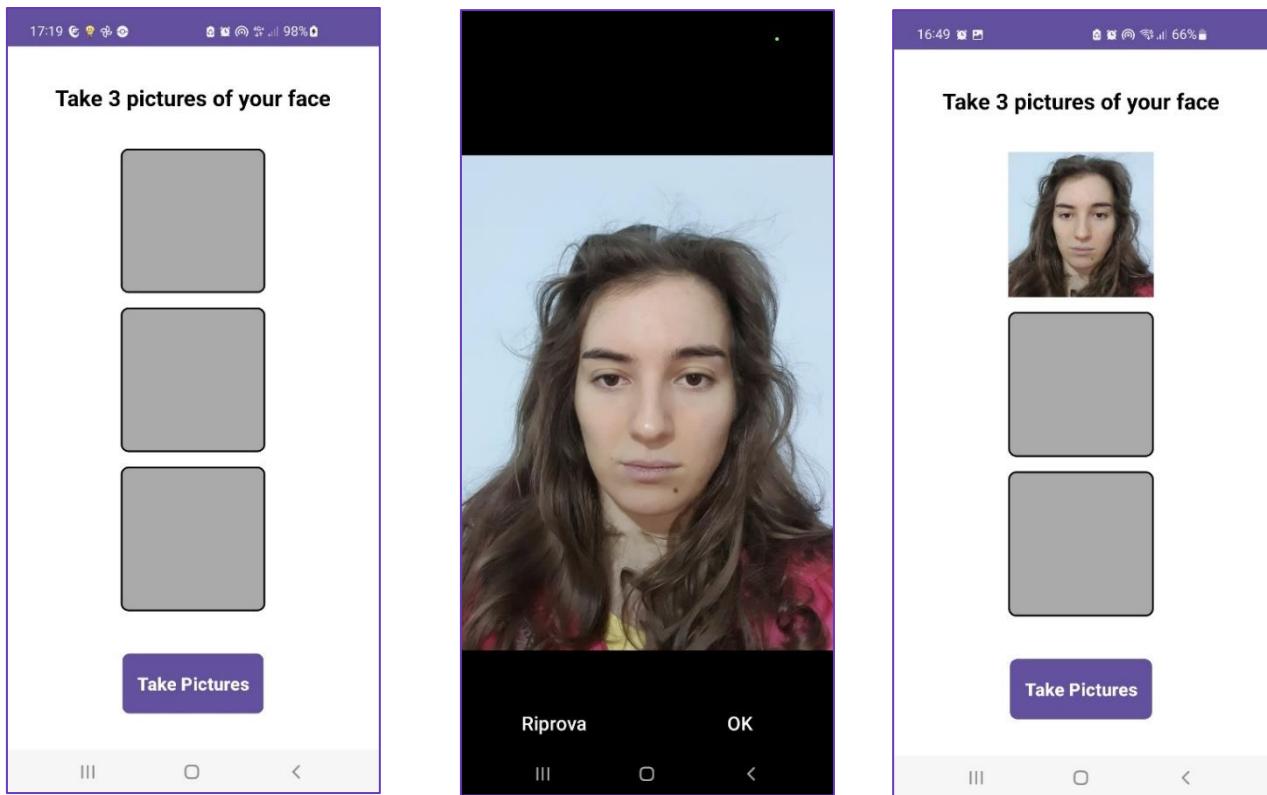


Figura 39: flusso di enrollment

L'utente deve ripetere il processo per altre due foto. Al termine degli scatti, il pulsante 'Take Pictures' viene sostituito da 'Register'. Cliccando su questo pulsante, viene inviata una richiesta HTTP contenente le tre immagini al nostro server. È stato necessario utilizzare un server perché i tre modelli di machine learning sono troppo pesanti per essere eseguiti in tempi efficienti su uno smartphone. Questa soluzione è ideale, poiché eseguendo i modelli su un PC è possibile sfruttare la GPU, garantendo prestazioni elevate e tempi di elaborazione rapidi. Se la richiesta di registrazione ha esito positivo, ovvero la connessione al server è andata a buon fine e il face detector ha individuato un volto in tutte e tre le foto, l'utente riceverà un messaggio tramite un Toast con la scritta "You registered successfully!". Se non è stato rilevato un volto in

almeno una delle immagini il server restituisce all'app un messaggio di errore che invita l'utente a ripetere la registrazione.

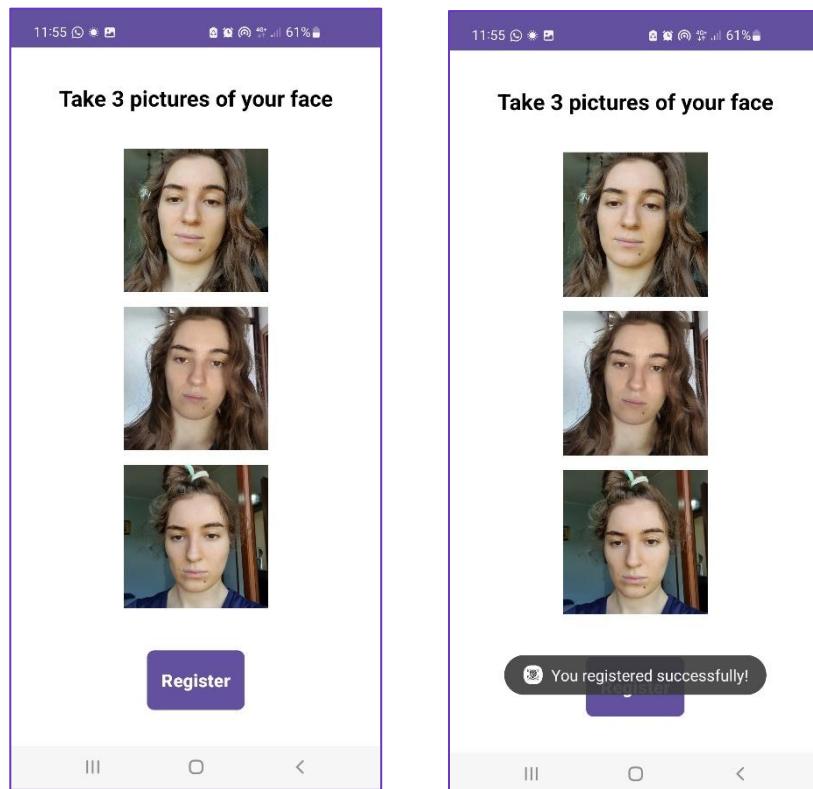


Figura 39: registrazione andata a buon fine

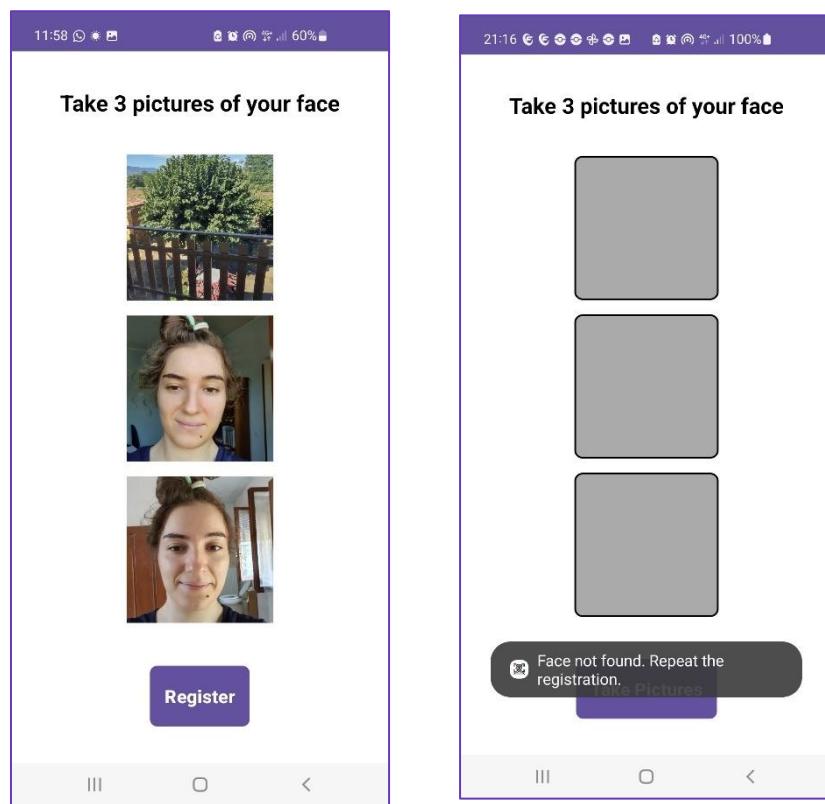


Figura 40: registrazione fallita

In caso di registrazione fallita, viene invocata la funzione `resetRegistration()`, che ripristina tutti i placeholder, rimuove le immagini inserite e riporta il pulsante al suo stato iniziale, permettendo all'utente di ricominciare da capo.

```
private void resetRegistration() {
    photoUrIs.clear();

    // Reset the ImageViews to their initial gray placeholder
    for (ImageView imageView : imageViews) {
        imageView.setImageDrawable(null); // Remove the current image
        imageView.setBackgroundResource(R.drawable.photo_placeholder);
    }

    // Reset the photo index
    currentPhotoIndex = 0;

    // Reset the button text and click listener
    takePicturesButton.setText("Take Pictures");
    takePicturesButton.setOnClickListener(v -> {
        if (currentPhotoIndex < 3) {
            dispatchTakePictureIntent();
        }
    });

    // Reset the bad registration count
    bad_registration_count = 0;
}
```

Figura 41: metodo `resetRegistration()`

Solo quando il volto viene rilevato correttamente in tutte e tre le immagini, il sistema estrae gli embeddings facciali, che vengono poi associati all'ID Android del dispositivo utilizzato per la registrazione.

L'invio delle immagini al server per inizializzare il processo di registrazione viene effettuato all'interno di `RegistrationActivity`. Il codice in figura 42 ne mostra i punti salienti.

```
private void sendImagesToServer(List<File> imageFiles) { 1 usage
    @SuppressLint("HardwareIds") String androidID = Settings.Secure.getString(getApplicationContext(),
        Settings.Secure.ANDROID_ID);

    OkHttpClient client = new OkHttpClient.Builder()
        .connectTimeout(5, TimeUnit.MINUTES) // Maximum time to establish a connection
        .writeTimeout(5, TimeUnit.MINUTES) // Maximum time to send data
        .readTimeout(5, TimeUnit.MINUTES) // Maximum time to read the response
        .build();

    MultipartBody.Builder requestBodyBuilder = new MultipartBody.Builder()
        .setType(MultipartBody.FORM)
        .addFormDataPart("android_id", androidID);

    // Add all the images to the request
    for (int i = 0; i < imageFiles.size(); i++) {
        File file = imageFiles.get(i);
        requestBodyBuilder.addFormDataPart("image" + i, file.getName(),
            RequestBody.create(MediaType.parse("image/jpeg"), file));
    }

    RequestBody requestBody = requestBodyBuilder.build();

    Request request = new Request.Builder()
        .url("http://192.168.227.181:5000/registration")
        .post(requestBody)
        .build();
}
```

```

client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(@NonNull Call call, @NonNull IOException e) {
        e.printStackTrace();
        runOnUiThread(() -> {
            Toast.makeText(context, RegistrationActivity.this, text: "Failed to register images",
            |           Toast.LENGTH_SHORT).show();
            resetRegistration();
        });
    }

    @Override 4 usages
    public void onResponse(@NonNull Call call, @NonNull Response response) throws IOException {
        if (response.isSuccessful()) {
            runOnUiThread(() -> {
                if (bad_registration_count == 0) {
                    Toast.makeText(context, RegistrationActivity.this, text: "You registered successfully!",
                    |           Toast.LENGTH_SHORT).show();
                }
            });
        } else {
            bad_registration_count += 1;
            runOnUiThread(() -> {
                Toast.makeText(context, RegistrationActivity.this, text: "Face not found. Repeat the registration.",
                |           Toast.LENGTH_SHORT).show();
                resetRegistration();
            });
        }
    }
});

```

Figura 42: metodo sendImagesToServer()

Abbiamo utilizzato la libreria OkHttp per inviare la lista delle immagini al server in modo asincrono. Di seguito viene spiegato il funzionamento del codice:

- *private void sendImagesToServer(List<File> imageFiles)*: definisce un metodo privato che prende in input una lista di file immagine (*imageFiles*) che rappresentano le 3 immagini scattate convertite in formato .jpeg e li invia a un server.
- *String androidID = Settings.Secure.getString(getApplicationContext(), Settings.Secure.ANDROID_ID)*: recupera l'ID univoco del dispositivo Android usandolo come parte della richiesta HTTP (che sarà di tipo multipart/form-data).
- *OkHttpClient client = new OkHttpClient.Builder()...build()*: crea un oggetto OkHttpClient configurato con un timeout massimo di 5 minuti per la connessione, la scrittura e la lettura.
- *RequestBodyBuilder.addFormDataPart("image" + i, file.getName(), RequestBody.create(MediaType.parse("image/jpeg"), file))*: aggiunge ciascuna immagine al corpo della richiesta.
- *Request request = new Request.Builder()...build()*: crea un oggetto Request, specificando l'URL del server e l'uso di HTTP POST per inviare il corpo della richiesta.
- *client.newCall(request).enqueue(new Callback() { ... })*: avvia la richiesta HTTP in modo asincrono. Il metodo enqueue consente di eseguire la

chiamata in background, evitando di bloccare il thread principale dell'interfaccia utente.

- *public void onFailure(Call call, IOException e)*: questo metodo viene eseguito se la richiesta fallisce e mostra un messaggio di errore all'utente con un Toast e resettando il processo di registrazione.
- *public void onResponse(Call call, Response response) throws IOException*: questo metodo viene eseguito quando il server risponde. Verifica se la risposta è stata gestita con successo: se la risposta del server è positiva viene visualizzato un Toast con un messaggio che conferma l'avvenuta registrazione. Se la risposta del server non è positiva (il server non ha trovato un volto in tutte le immagini), incrementa una variabile chiamata *bad_registration_count* e informa l'utente con un Toast che la registrazione non è riuscita, chiedendogli di ripetere il processo.

Il server riceve la richiesta HTTP e attiva la route /registration, che gestisce il processo di caricamento delle immagini. All'interno di questa route, il server estrae le tre immagini dalla richiesta, le decodifica e le salva temporaneamente in una lista.

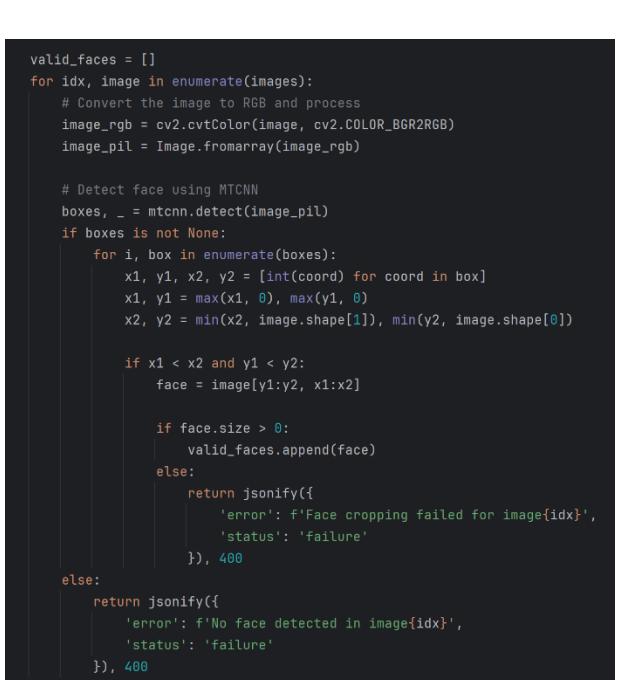
```
@app.route('/registration', methods=['POST'])
def upload():
    images = []
    android_id = request.form.get('android_id')

    # Check for the presence of multiple image files
    for i in range(3):
        if f'image{i}' in request.files:
            file = request.files[f'image{i}']
            if file.filename != '':
                # Read the image directly from the request
                image = np.fromstring(file.read(), np.uint8)
                image = cv2.imdecode(image, cv2.IMREAD_COLOR)
                images.append(image)
            else:
                return jsonify({'error': f'No selected file for image{i}'}), 400
        else:
            return jsonify({'error': f'No image{i} part in the request'}), 400
```

Figura 43: route /registration sul server

Successivamente, viene applicato MTCNN per rilevare i volti e, se tutte le immagini contengono volti validi, viene eseguita la fase di estrazione degli embeddings facciali. Una volta che i volti sono stati correttamente estratti da tutte e tre le immagini, ciascuna viene convertita in un tensore utilizzando la libreria PyTorch. Successivamente, le immagini vengono normalizzate secondo la procedura di *fixed_image_standardization*, garantendo così la compatibilità con il modulo di riconoscimento facciale. Per ogni immagine,

viene quindi applicato il modulo di estrazione degli embeddings, che cattura le features del volto. Questi embeddings vengono memorizzati come array NumPy, ciascuno salvato con un nome nel formato "embedding_photo_{timestamp in millisecondi}.npy". I file vengono archiviati in una cartella associata all'ID Android del dispositivo che ha originato la richiesta. Questi embeddings saranno utilizzati in fase di verification per confrontarli con quelli presentati come probe dall'utente che vuole autenticarsi.



```

valid_faces = []
for idx, image in enumerate(images):
    # Convert the image to RGB and process
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image_pil = Image.fromarray(image_rgb)

    # Detect face using MTCNN
    boxes, _ = mtcnn.detect(image_pil)
    if boxes is not None:
        for i, box in enumerate(boxes):
            x1, y1, x2, y2 = [int(coord) for coord in box]
            x1, y1 = max(x1, 0), max(y1, 0)
            x2, y2 = min(x2, image.shape[1]), min(y2, image.shape[0])

            if x1 < x2 and y1 < y2:
                face = image[y1:y2, x1:x2]

                if face.size > 0:
                    valid_faces.append(face)
                else:
                    return jsonify({
                        'error': f'Face cropping failed for image{idx}',
                        'status': 'failure'
                    }), 400
            else:
                return jsonify({
                    'error': f'No face detected in image{idx}',
                    'status': 'failure'
                }), 400
    else:
        return jsonify({
            'error': f'No face detected in image{idx}',
            'status': 'failure'
        }), 400
else:
    return jsonify({
        'error': f'All images processed successfully, embeddings saved',
        'status': 'success'
    }), 200
else:
    return jsonify({
        'error': 'Failed to detect valid faces in all images',
        'status': 'failure'
    }), 400
}

```

```

# If all the images are real, save all the embeddings
if len(valid_faces) == 3:
    for idx, face in enumerate(valid_faces):
        face = cv2.resize(face, IMG_SIZE)
        trans = transforms.Compose([
            np.float32,
            transforms.ToTensor(),
            fixed_image_standardization
        ])

        face_tensor = trans(face).unsqueeze(0).cuda()
        resnet_model = load_model(pretrained_on_dataset='vggface2', device)

    # Create the face embeddings
    resnet_model.eval()
    with torch.no_grad():
        face_embedding = resnet_model(face_tensor).squeeze(0).cpu().numpy()

    # Save the embeddings as a numpy array
    embedding_filename = f"embedding_image{idx}_{android_id}.npy"
    embedding_filepath = os.path.join(android_folder, embedding_filename)
    np.save(embedding_filepath, face_embedding)

return jsonify({
    'message': 'All images processed successfully, embeddings saved',
    'status': 'success'
}), 200
else:
    return jsonify({
        'error': 'Failed to detect valid faces in all images',
        'status': 'failure'
    }), 400
}

```

Figura 44: applicazione del modulo face detection e estrazione delle features sulle foto inviate

L'utente, premendo il bottone “Verification” dalla schermata home, viene indirizzato alla pagina di autenticazione. All'apertura di questa nuova schermata, la fotocamera del dispositivo si attiva automaticamente, e vengono posizionati nella parte inferiore dello schermo due pulsanti: “Switch Camera”, per cambiare la telecamera attiva (frontale o posteriore) e “Take Pictures”. Premendo quest'ultimo pulsante, viene avviato un timer di 5 secondi che conta alla rovescia, offrendo all'utente il tempo necessario per posizionarsi correttamente davanti alla fotocamera. Quando il timer raggiunge lo zero, il dispositivo scatta automaticamente tre foto consecutive. Ogni scatto è accompagnato da un feedback visivo e sonoro. Le tre immagini appena scattate vengono immediatamente inviate al server, che le elabora per

verificare se il volto dell'utente corrisponde ai template registrati con quell'Android ID.

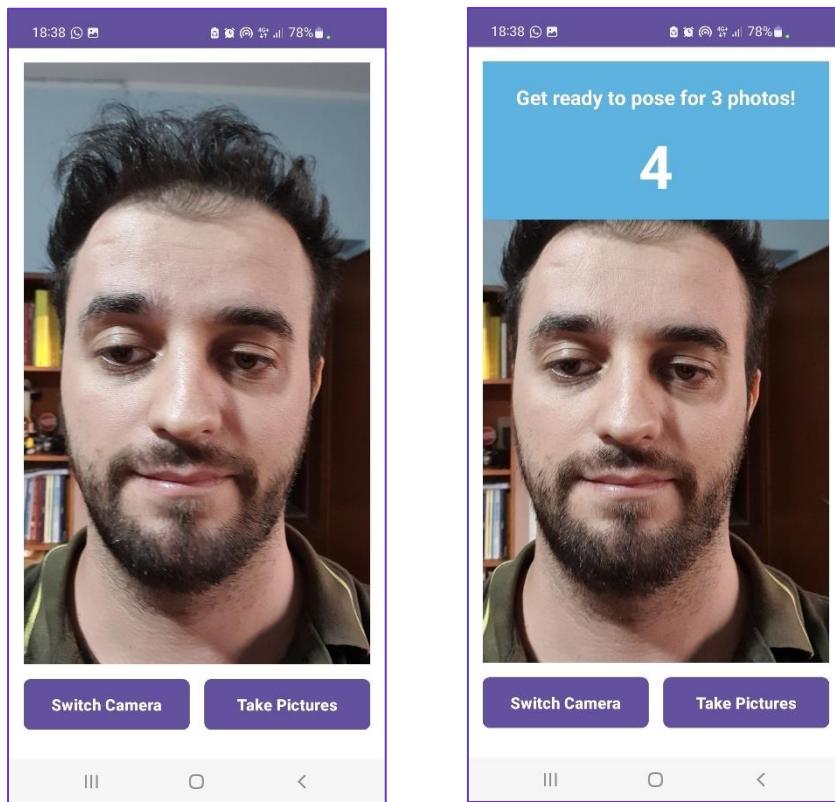


Figura 45: VerificationActivity

Il processo di invio delle immagini al server viene gestito dal metodo `uploadImage(File file)` all'interno della `VerificationActivity`. Questo metodo accetta come input un file immagine, quindi viene chiamato tre volte durante la fase di verification, una per ogni foto scattata. Viene recuperato l'identificativo univoco del dispositivo Android (AndroidID) tramite una chiamata a `Settings.Secure.getString`, che estrae questo valore dalle impostazioni del sistema operativo. Questo identificativo fungerà da identity claim per eseguire il matching con i template relativi presenti nella gallery. Abbiamo costruito poi il corpo della richiesta HTTP utilizzando `MultipartBody.Builder`. In questa sezione, vengono aggiunti i dati necessari: l'immagine da caricare, l'ID Android del dispositivo e il numero delle chiamate di verifica eseguite fino a quel momento. Il file immagine viene allegato specificando che il suo tipo è "image/jpeg", e tutto viene convertito in un formato compatibile con la trasmissione su rete. La richiesta viene costruita utilizzando un `Request.Builder`, che specifica l'URL del server e la route del metodo che

computa la verification. In aggiunta è stabilito che il metodo di trasmissione è di tipo POST. Una volta preparata la richiesta, essa viene inviata in modo asincrono attraverso il client HTTP.

```

private void uploadImage(File file) { 1 usage

    number_of_verification_calls = number_of_verification_calls + 1; // Increment the verification call counter
    System.out.println(number_of_verification_calls);
    @SuppressLint("HardwareIds") String androidID = Settings.Secure.getString(getApplicationContext(),
        Settings.Secure.ANDROID_ID);

    OkHttpClient client = new OkHttpClient.Builder()
        .connectTimeout( timeout: 5, TimeUnit.MILLISECONDS ) // Set connection timeout
        .writeTimeout( timeout: 5, TimeUnit.MILLISECONDS ) // Set write timeout
        .readTimeout( timeout: 5, TimeUnit.MILLISECONDS ) // Set read timeout
        .build();

    RequestBody requestBody = new MultipartBody.Builder()
        .setType(MultipartBody.FORM)
        .addFormDataPart( name: "image", file.getName(),
            RequestBody.create(MediaType.parse( "image/jpeg"), file))
        .addFormDataPart( name: "android_id", androidID)
        .addFormDataPart( name: "number_of_verification_calls", String.valueOf(number_of_verification_calls))
        .build();

    Request request = new Request.Builder()
        .url("http://192.168.227.181:5000/verification")
        .post(requestBody)
        .build();

    client.newCall(request).enqueue(new Callback() {
        @Override
        public void onFailure(@NotNull Call call, @NotNull IOException e) {
            Log.e( tag: "Upload", msg: "Image upload failed: " + e.getMessage(), e);
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText( context: VerificationActivity.this, text: "Image upload failed: " +
                        e.getMessage(), Toast.LENGTH_SHORT).show();
                }
            });
        }

        @Override 4 usages
        public void onResponse(@NotNull Call call, @NotNull Response response) throws IOException {
            if (!response.isSuccessful()) {
                Log.e( tag: "Upload", msg: "Image upload failed: " + response.message());
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        Toast.makeText( context: VerificationActivity.this, text: "REJECTED", Toast.LENGTH_SHORT).show();
                    }
                });
            } else {
                Log.d( tag: "Upload", msg: "Image upload succeeded: " + response.message());
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        imagesUploadedCount++;
                        if (imagesUploadedCount == 3) {
                            Intent intent = new Intent( packageContext: VerificationActivity.this, UserAcceptedActivity.class);
                            startActivity(intent);
                        }
                    }
                });
            }
        }
    });
    if (number_of_verification_calls == 3) { // Reset the counter after 3 verification calls
        number_of_verification_calls = 0;
    }
}

```

Figura 46: metodo uploadImage()

Nel caso in cui il caricamento fallisca, viene eseguito il metodo *onFailure*, che scrive un messaggio di errore nei log e aggiorna l'interfaccia utente mostrando un Toast. Se invece la richiesta ha successo, il metodo *onResponse* viene

invocato. In caso di arrivo di una decisione negativa dal server, viene mostrato all'utente un Toast con il massaggio "REJECTED". Altrimenti, se tutte e tre le immagini inviate superano ognuno dei 3 moduli (face detection, liveness detection, face Recognition), l'app avvia la UserAcceptedActivity, che indica all'utente che la verifica è andata a buon fine.

Ogni volta che viene chiamato, il server riceve una richiesta HTTP contenente sia il file dell'immagine che l'identificativo del dispositivo Android che ha effettuato la richiesta. Se il file immagine risulta corrotto o invalido, il processo di autenticazione fallisce istantaneamente e l'utente riceve un messaggio di errore. Se le immagini sono valide, ciascuna di esse viene elaborata singolarmente dal modello di face detection MTCNN in maniera analoga a quanto già visto per la registrazione. Se anche solo una delle tre immagini non contiene un volto rilevabile (MTCNN restituisce "None"), il processo di autenticazione termina immediatamente e il server invia all'utente un messaggio di rifiuto ("REJECTED"), informandolo del fallimento della verifica. Se invece i volti sono stati correttamente rilevati in tutte e tre le immagini, queste vengono inviate al modulo di liveness detection che esegue il modello MobileNetV2 addestrato sul dataset combinato contenente immagini sia di MSU-MFSD che di NUAA Photograph Imposter Database.

Il metodo *detect_real_or_fake(face, loaded_model, IMG_SIZE)* riceve in input l'immagine del volto ritagliata da MTCNN, il modello di anti-spoofing e la dimensione dell'immagine, che nel nostro caso è fissata a 224x224 pixel. L'immagine viene ulteriormente preprocessata, convertita in un tensore PyTorch e normalizzata. A questo punto, viene passata al modello di anti-spoofing che restituisce una probabilità compresa tra 0 e 1. Se questa probabilità supera la soglia di 0.5, la predizione sarà "Fake", altrimenti "Real".

```
1 usage
def detect_real_or_fake(image, model, target_size):
    model.eval()
    with torch.no_grad():
        processed_image = preprocess_image(image, target_size).to(device)
        output = model(processed_image)
        prediction = torch.sigmoid(output).item() # Binary prediction (Real or Spoof)
        return prediction, "Fake" if prediction > 0.5 else "Real"
```

Figura 46: funzione detect_real_or_fake()

Se anche una sola delle tre immagini viene identificata come spoof, il processo di autenticazione si interrompe, l'utente viene respinto e riceve il messaggio

“REJECTED”. In caso contrario, se tutte e tre le immagini vengono classificate come appartenenti a persone reali, si passa all’ultima fase del processo, l’esecuzione del modulo di face recognition.

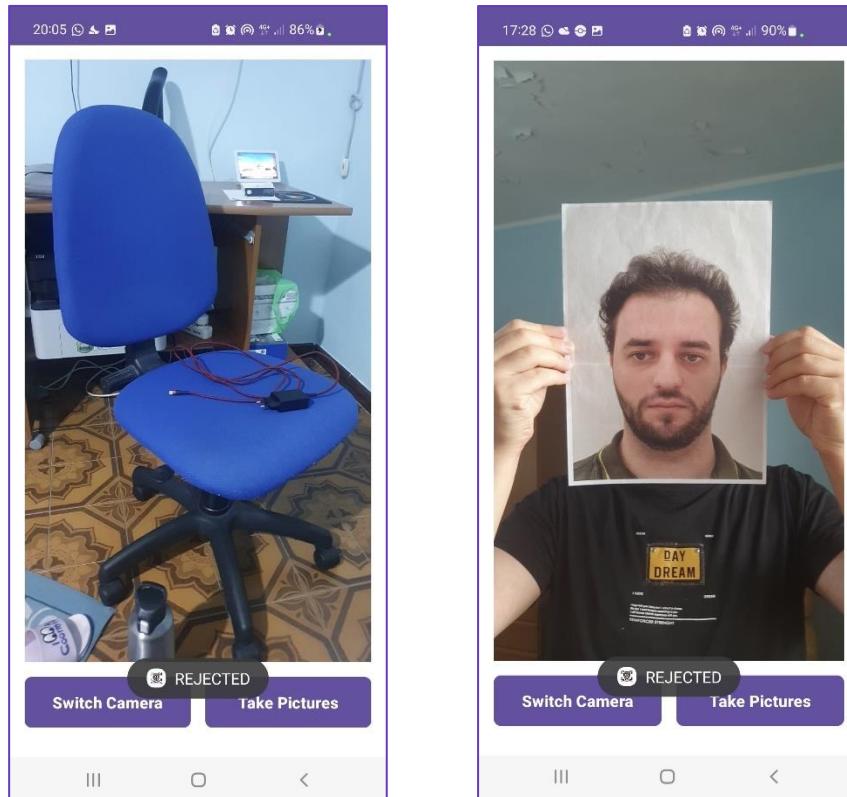


Figura 47: verification fallita per l’assenza di volti in almeno una delle tre immagini e verification fallita a causa di uno spoofing attack in almeno una delle tre immagini

In questa fase, le immagini dei volti, ora ridimensionate a 224x224 pixel e ritagliate attorno al bounding box rilevato, vengono preparate per l’elaborazione finale. Ogni immagine viene sottoposta a una standardizzazione attraverso il processo di `fixed_image_standardization`. Il modello di face recognition, a questo punto, estrae degli embeddings di dimensione 512 per ciascuna immagine. Questi embeddings sono rappresentazioni uniche delle caratteristiche facciali e costituiscono la base per il confronto con i template registrati. A questo punto, per ogni embedding generato viene calcolata la distanza euclidea rispetto agli embeddings salvati nel dispositivo durante la registrazione. Se ciascun embedding ha una distanza rispetto ai template salvati inferiore alla soglia di 0.58, il sistema riconosce l’utente come legittimo e lo autorizza ad accedere reindirizzandolo alla schermata di controllo dell’attuatore lineare. Se, invece, anche solo uno degli embeddings ha un valore

di distanza inferiore alla soglia, l’utente viene respinto e riceve il messaggio “REJECTED”. La soglia di 0.58 è stata scelta durante la fase di valutazione del modello di riconoscimento facciale, in quanto corrispondeva all’Equal Error Rate (EER) del modello addestrato su VGGFace2 considerando S=3. L’EER è una scelta molto popolare come threshold perché rappresenta un giusto compromesso tra il voler limitare le false acceptances e le false rejections ma senza compromettere il GAR e il GRR.

```

else:
    print("Real")
    trans = transforms.Compose([
        np.float32,
        transforms.ToTensor(),
        fixed_image_standardization
    ])

    face_tensor = trans(face).unsqueeze(0).cuda()
    resnet_model = load_model(pretrained_on_dataset='vggface2', device)
    resnet_model.eval()
    with torch.no_grad():
        face_embedding = resnet_model(face_tensor).squeeze(0).cpu()

    embeddings = []
    for embedding_file in embedding_files:
        embedding_path = os.path.join(REGISTERED_USER_FOLDER, embedding_file)
        embedding = np.load(embedding_path)
        embeddings.append(torch.tensor(embedding).cuda())

    image_embeddings = torch.stack(embeddings).cuda()
    distance_matrix = calculate_distance(face_embedding, image_embeddings)

    threshold = 0.58
    min_distance = torch.min(distance_matrix)

    if min_distance <= threshold:
        result = 1
        minimum_number_of_matches += result
    else:
        result = 0

    print(f"The result of the matching is: {result}")

```

Figura 48: codice che gestisce le operazioni da eseguire per effettuare il matching

Quando l’utente ottiene l’accesso, viene visualizzata la schermata UserAcceptedActivity. Per gestire la comunicazione tra l’app e il Raspberry Pi, è stato implementato un server HTTP Flask. Quando viene aperta la UserAcceptedActivity, nel metodo *onCreate(Bundle savedInstanceState)* vengono inizializzati tutti i componenti della schermata, e il metodo privato *fetchStatus()* invia una richiesta HTTP di tipo GET al server Flask per ottenere lo stato corrente dell’attuatore lineare. Se l’attuatore è esteso, significa che la porta è bloccata; se l’attuatore è ritratto, la porta è sbloccata.

```

private void fetchStatus() { 1 usage
    // Create a request to fetch the status from the server
    Request request = new Request.Builder()
        .url(BASE_URL + "/get_status")
        .build();
    // Asynchronously execute the request
    client.newCall(request).enqueue(new Callback() {
        @Override
        public void onFailure(@NonNull Call call, @NonNull IOException e) {
            e.printStackTrace();
        }

        @Override 4 usages
        public void onResponse(@NonNull Call call, @NonNull Response response) throws IOException {
            if (response.isSuccessful()) {
                try {
                    // Parse the response to get the status
                    assert response.body() != null;
                    JSONObject jsonObject = new JSONObject(response.body().string());
                    isDoorLocked = jsonObject.getBoolean("name: \"isDoorLocked\"");

                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            // Update the UI with the fetched status
                            updateGateUI();
                            updateDoorUI();
                        }
                    });
                } catch (JSONException e) {
                    e.printStackTrace();
                }
            }
        }
    });
}

```

Figura 49: metodo fetchStatus()

Questa informazione viene visualizzata nell’interfaccia utente: quando l’attuatore è disteso, la schermata mostrerà l’icona della porta chiusa, quando è contratto, verrà mostrata l’icona della porta aperta.

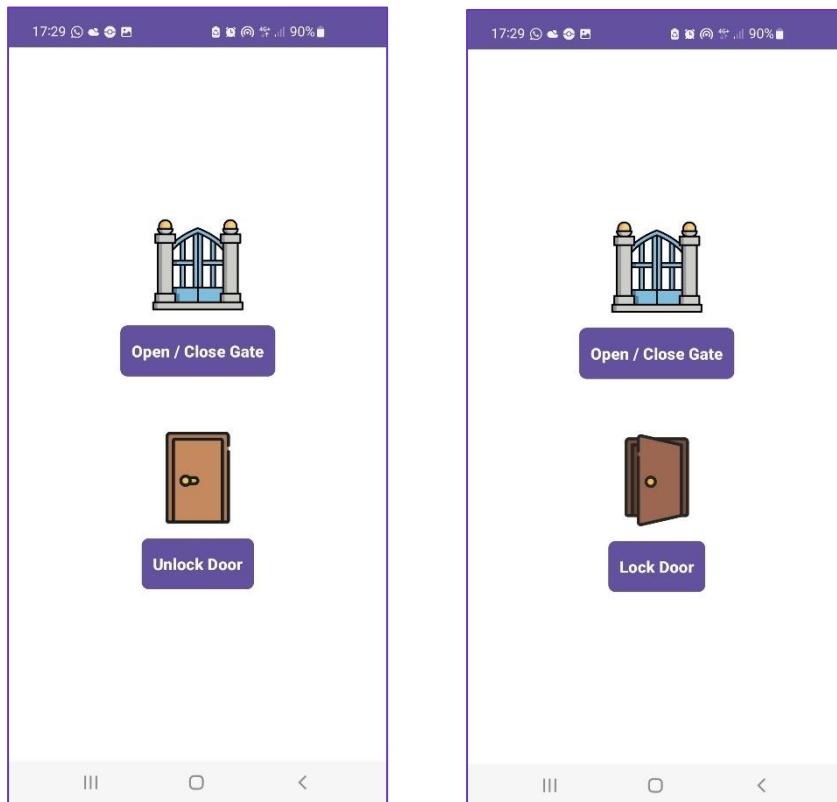


Figura 50: schermata UserAcceptedActivity al variare dello stato della porta

Il cancello, invece, ha un comportamento leggermente diverso. L'attuatore lineare apre e chiude il cancello tramite il pulsante del citofono. Per aprire il cancello, l'attuatore deve spingersi in avanti premendo il bottone del citofono e poi contrarsi velocemente, così da evitare di tenere il bottone premuto. Per chiudere il cancello, l'attuatore ripete lo stesso movimento. Per questo motivo, è sufficiente un solo bottone sull'app per gestire l'apertura e la chiusura del cancello: ogni pressione del pulsante invia un comando all'attuatore per compiere il ciclo di movimento avanti-indietro.

Viene utilizzato un insieme di comandi GPIO per far estendere, ritrarre e fermare l'attuatore lineare. Le funzioni che definiscono questi movimenti sono così definite:

- *move_forward()*: questa funzione attiva i pin IN1 e IN2 del motor driver rispettivamente con un segnale alto (GPIO.HIGH) e con un segnale basso (GPIO.LOW). Questo causa un movimento in avanti dell'attuatore.
- *move_backward()*: in questo caso, IN1 viene impostato su basso (GPIO.LOW) e IN2 su alto (GPIO.HIGH), permettendo all'attuatore di ritrarsi.
- *stop_actuator()*: entrambi i pin IN1 e IN2 vengono impostati su basso (GPIO.LOW), fermando il movimento dell'attuatore. Questa funzione è utilizzata per arrestare l'attuatore una volta completato il movimento desiderato.

```
# Function to move the motor forward (close the door).
def move_forward():
    GPIO.output(IN1, GPIO.HIGH)
    GPIO.output(IN2, GPIO.LOW)
    print("Forward")

# Function to move the motor backward (open the door).
def move_backward():
    GPIO.output(IN1, GPIO.LOW)
    GPIO.output(IN2, GPIO.HIGH)
    print("Backward")

# Function to stop the motor.
def stop_actuator():
    GPIO.output(IN1, GPIO.LOW)
    GPIO.output(IN2, GPIO.LOW)
    print("Stop")
```

Figura 51: funzioni che permettono il movimento dell'attuatore lineare

Materiale del progetto

In questa sezione viene fornita una panoramica completa del materiale prodotto durante lo sviluppo del progetto.

Tutti i dataset utilizzati nel progetto sono stati scaricati localmente per eseguire i test sui tre moduli biometrici. Complessivamente, tutti questi file pesano più di 30 GB. Abbiamo provato a caricarli su Google Drive, ma il processo avrebbe richiesto circa tre giorni a causa della nostra connessione Internet non molto performante, quindi abbiamo deciso di non effettuarne l'upload.

Il materiale del progetto è contenuto all'interno della cartella *Progetto_BS_Frabotta_Ferrone*, che si articola in quattro sottocartelle principali, ognuna delle quali contiene file e directory specifici:

- *Android Application* contiene:
 - La cartella “*BiometricGateAccess*” al cui interno si trovano tutti i file che compongono l'app Android.
 - La cartella “*BiometricServer*” che include il file app.py, utilizzato per avviare il server che esegue i moduli di face detection, liveness detection e face recognition. All'interno della cartella “*static*” sono presenti gli embeddings dei template delle identità registrate sull'app, salvati nelle cartelle associate ai dispositivi registrati.
- *Evaluation* contiene altre tre sottocartelle:
 - *Face_antispoofing* contiene:
 - La cartella “*Models*” con i due modelli migliori addestrati sul dataset combinato per il modulo di face antispoofing, in formato .pth.
 - *Antispoofing_module_training_and_evaluation.ipynb*: un notebook in cui viene eseguito il training e la valutazione del modello di liveness detection utilizzando MobileNetV2, addestrato sul dataset combinato che comprende immagini di MSU-MFSD e NUAA Photograph Imposter Database. Include anche l'utilizzo di Local Binary Patterns (LBP) nella fase di pre-processing.

- *Create_dataset_MSU_as_images.ipynb*: un notebook in cui vengono estratti tutti i frames dai video del dataset MSU.
 - *Create_numpy_arrays_from_images.ipynb*: questo codice processa e genera array NumPy dai dataset NUAA e MSU-MFSD che facilitano la successiva conversione dei dati in tensori di PyTorch.
 - *Previous_trials-antispoofing_module.ipynb*: è un notebook che riporta tutti gli altri esperimenti effettuati prima di arrivare all'allenamento e al test del modello definitivo scelto per il modulo di liveness detection dell'app.
 - *Real_time_face_detection_and_antispoofing.ipynb*: un notebook che implementa un sistema che esegue face detection e anti-spoofing in real-time sfruttando la webcam del computer. È stato utilizzato per scegliere il modello di liveness detection più adatto a diverse condizioni ambientali.
- *Face_detection* contiene:
 - *Face_detection_module_evaluation.ipynb*: un notebook che esegue la valutazione del modello di face detection basato su MTCNN.
- *Face_Recognition* contiene:
 - *Face_Recognition_Evaluation.ipynb*: un notebook che valuta il modulo di riconoscimento facciale basato su FaceNet, pre-addestrato sui dataset VGGFace2 e CASIA-Webface, utilizzando il dataset LFW.
- *Raspberry Pi Code* contiene:
 - *Linear_actuator_server.py*: codice Python che implementa il server per la comunicazione tra l'app Android e il Raspberry Pi, responsabile del controllo dell'attuatore lineare.
- *Video_demos* contiene:
 - Tutti i video dimostrativi relativi al progetto.
- Relazione del progetto
- Presentazione Power Point del progetto