



**Politecnico
di Torino**

DIPARTIMENTO DI AUTOMATICA E INFORMATICA (DAUIN)

Corso di Laurea Magistrale in Ingegneria Informatica

**Miriana Martini
Roberta Macaluso**

**Prototipo Real-Time di un Assistente Visivo Virtuale della
Lingua dei Segni Internazionale tramite MediaPipe**

Tesina di Image Processing and Computer Vision

Professori: **Bartolomeo Montrucchio
Luigi De Russis**

ANNO ACCADEMICO 2022/2023

Dicembre 2022

INDICE

1. Abstract
2. Linguaggio dei Segni
3. MediaPipe
 - 3.1. MediaPipe Hands
4. Obiettivo (Risultato Atteso)
5. Implementazione
 - 5.1. HandTrackingModule.py
 - 5.2. Recogniser.py
 - 5.3. StoreGesture.py
 - 5.4. RecogniserSigned.py
 - 5.5. StoreGestureSigned.py
6. Valutazione dei Metodi Implementati
7. Risultato Ottenuto
8. Requisiti di Funzionamento
 - 8.1. Condizioni di Luminosità
 - 8.2. Requisiti Hardware e Software

1. Abstract

Ciò che ci ha spinto nella realizzazione di questo progetto è il desiderio di permettere alle persone non udenti di comunicare con più facilità. Per cui abbiamo deciso di dare un piccolo contributo implementando un prototipo di un assistente visivo virtuale real-time in grado di interpretare i gesti dell'alfabeto della Lingua dei Segni Internazionale.

Quest'assistente, implementato mediante l'utilizzo del framework MediaPipe e della libreria OpenCV, è capace di riconoscere e tradurre con grande semplicità i gesti statici prodotti dalla mano sinistra in real-time, in quanto richiede il solo utilizzo di una webcam.

Tuttavia, essendo un prototipo, andrebbe migliorato in termini di precisione nel riconoscimento dei gesti e ampliato all'interpretazione di intere espressioni o frasi più complesse che richiedono il movimento della mano nel tempo.

2. Linguaggio dei Segni

Le lingue dei segni sono lingue che veicolano i propri significati attraverso un sistema codificato di segni delle mani, espressioni del viso e movimenti del corpo. Sono utilizzate dalle comunità dei segnanti, a cui appartengono, in maggioranza, persone non udenti.

La comunicazione attraverso la lingua dei segni avviene tramite il canale visivo-gestuale, producendo dei segni precisi, compiuti con una o entrambe le mani, aventi un significato specifico e associato, come avviene per le parole.

I segni di ogni lingua dei segni possono essere scomposti in 4 componenti essenziali:

- Movimento,
- Orientamento,
- Configurazione,
- Luogo

(ossia le quattro componenti manuali del segno)

Le lingue dei segni sono lingue complete con una propria grammatica e un proprio dizionario, ma ne parliamo al plurale in quanto, ad ogni nazione, corrispondono diverse lingue dei segni; in Italia troviamo la **Lingua dei Segni Italiana (LIS)**, negli Stati Uniti la **Lingua dei Segni Americana (ASL)**, nel Regno Unito la **Lingua dei Segni Britannica (BSL)**, ... e così via.

Motivo per cui è stato deciso di codificare una lista di segni "internazionali", che facilitasse il superamento delle barriere linguistiche.

Prende il nome di Signuno o Gestuno ed è la **Lingua dei Segni Internazionale**, sviluppata dalla World Federation of Deaf negli anni '50, per permettere la comunicazione tra persone non udenti anche se di diversa nazionalità, e quindi usanti diverse lingue dei segni.

Trattandosi di una proposta esclusivamente lessicale, fu utilizzato, talvolta, in contesti di incontri internazionali, senza però acquisire mai le caratteristiche di una vera e propria lingua.

In Fig. 2.1. è possibile osservare i segni rappresentanti l'alfabeto del Signuno.



Fig. 2.1

3. MediaPipe

MediaPipe è una libreria multiplatforma sviluppata da Google che fornisce incredibili soluzioni di Machine Learning pronte all'uso per attività di visione artificiale. Più specificatamente è un framework basato su grafi per la creazione di pipeline di machine learning multimodali (video, audio e sensori).

Tramite l'uso di MediaPipe e OpenCV è stato possibile implementare un prototipo scalabile di un Riconoscitore di gesti che abbiamo poi adattato al riconoscimento delle lettere dell'alfabeto della lingua dei segni internazionale.

3.1. MediaPipe Hands

Nonostante, per una persona, il riconoscimento di una mano e i suoi movimenti sia un task piuttosto semplice, ciò non è altrettanto corretto dal punto di vista di una macchina, infatti la percezione della mano in tempo reale è un compito di Computer Vision decisamente impegnativo, poiché le mani spesso si occludono a vicenda (ad es. occlusioni di dita/palmo, tremori e orientamento della mano) e ci si ritrova spesso in mancanza di pattern ad alto contrasto.

A tal proposito **MediaPipe Hands** è una soluzione di tracciamento di mani e dita ad alta fedeltà. Utilizza il Machine Learning (ML) per dedurre 21 punti di riferimento 3D di una mano da un solo fotogramma.

3.1.1. Pipeline di Machine Learning

MediaPipe Hands utilizza una pipeline di Machine Learning composta da più modelli che lavorano insieme:

- Un **Palm Detection Model** che opera sull'immagine completa e restituisce un riquadro di delimitazione della mano orientato.
- Un **Hand Landmark Model** che opera sulla regione dell'immagine ritagliata definita dal rilevatore del palmo e restituisce punti chiave della mano 3D ad alta fedeltà.

Fornire l'immagine della mano accuratamente ritagliata all'**Hand Landmark Model** riduce drasticamente la necessità di aumentare i dati (ad esempio rotazioni, traslazione e scala) e consente invece alla rete di dedicare la maggior parte della sua capacità all'accuratezza della previsione delle coordinate.

3.1.2. Palm Detection Model

La mancanza di motivi ad alto contrasto nelle mani rende relativamente difficile rilevarle in modo affidabile solo dalle loro caratteristiche visive. Invece, fornire un contesto aggiuntivo, come le caratteristiche del braccio, del corpo o della persona, aiuta la localizzazione accurata della mano.

Il **Palm Detection Model** nasce dal fatto che stimare i riquadri di delimitazione di oggetti rigidi come palmi e pugni è significativamente più semplice rispetto al rilevamento di mani con dita articolate. Inoltre, i palmi possono essere modellati utilizzando riquadri di delimitazione quadrati ("*ancore*" nella terminologia ML) ignorando altre proporzioni e quindi riducendo il numero di ancoraggi di un fattore 3-5. In secondo luogo, un estrattore di funzionalità codificatore-decodificatore viene utilizzato per una maggiore consapevolezza del contesto della scena anche per piccoli oggetti. Infine, la perdita focale viene ridotta al minimo durante l'allenamento per supportare una grande quantità di ancoraggi risultanti dalla varianza su larga scala.

Con le tecniche di cui sopra, si ottiene una precisione media del 95,7% nel rilevamento del palmo.

3.1.3. Hand Landmark Model

Dopo il rilevamento del palmo sull'intera immagine, il successivo **Hand Landmark Model** esegue una precisa localizzazione dei punti chiave di 21 coordinate 3D delle nocche della

mano all'interno delle regioni della mano rilevate tramite regressione, ovvero la previsione diretta delle coordinate. Il modello apprende una rappresentazione coerente della posa della mano interna ed è robusto anche per mani parzialmente visibili e in presenza di auto-occlusioni.

Per ottenere il risultato sono state annotate manualmente ~30.000 immagini del mondo reale con 21 coordinate 3D, come mostrato in Fig.3.1 (il valore Z viene preso dalla mappa di profondità dell'immagine, se esiste, per ogni coordinata corrispondente). Per coprire meglio le possibili pose della mano e fornire un'ulteriore supervisione sulla natura della geometria della mano, viene eseguito anche il rendering di un modello di mano sintetico di alta qualità su vari sfondi e viene mappato alle corrispondenti coordinate 3D.

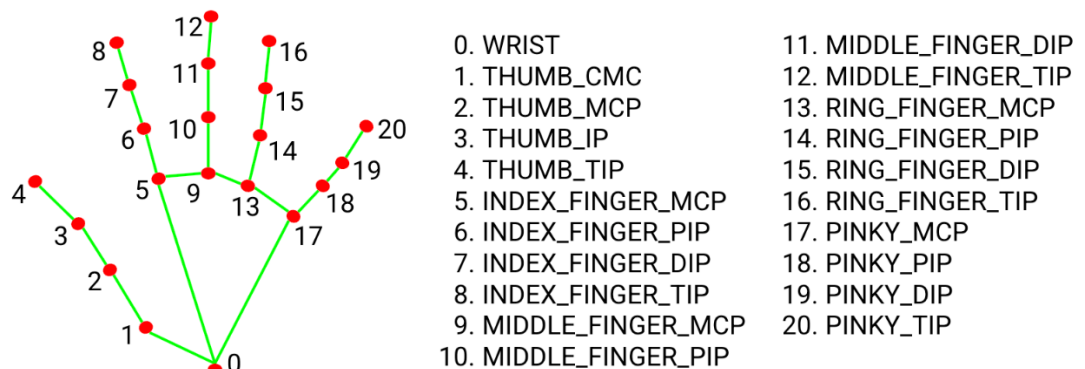


Fig. 3.1

3.1.4. Opzioni di Configurazione

- **STATIC_IMAGE_MODE:** Se impostato su *false*, la soluzione tratta le immagini di input come un flusso video. Tenterà di rilevare le mani nelle prime immagini di input e, in caso di rilevamento riuscito, localizzerà ulteriormente i punti di riferimento della mano. Nelle immagini successive, una volta rilevate tutte le mani e localizzati i corrispondenti punti di riferimento delle mani, tiene semplicemente traccia di tali punti di riferimento senza invocare un altro rilevamento fino a quando non perde traccia di tutte le mani. Ciò riduce la latenza ed è ideale per l'elaborazione di fotogrammi video. Se impostato su *true*, il rilevamento delle mani viene eseguito su ogni immagine di input, ideale per l'elaborazione di un batch di immagini statiche, possibilmente non correlate. Il valore di default è *false*.
- **MAX_NUM_HANDS:** Il numero massimo di mani da rilevare. Il valore di default è 2.
- **MODEL_COMPLEXITY:** Complessità dell'hand landmark model. Il valore può essere 0 o 1. Con l'aumento della complessità del modello, l'accuratezza del punto di riferimento e la latenza dell'interferenza, di solito, aumentano. Il valore di default è 1.
- **MIN_DETECTION_CONFIDENCE:** Rappresenta la misura di quanto il rilevamento della mano sia considerato riuscito da parte del modello. È un valore compreso nell'intervallo [0.0, 1.0]. Il valore di default è 0.5.
- **MIN_TRACKING_CONFIDENCE:** Valore di confidenza minimo (nell'intervallo [0.0, 1.0]) richiesto dal modello di tracciamento dei punti di riferimento affinché questi siano considerati tracciati correttamente, altrimenti il rilevamento della mano verrà richiamato automaticamente sull'immagine di input successiva. Impostandolo su un valore più elevato è possibile aumentare la robustezza della soluzione, a scapito di una latenza più elevata. Questo valore è ignorato se *STATIC_IMAGE_MODE* è settato a *True*, dove il rilevamento delle mani viene eseguito semplicemente su ogni immagine. Il valore predefinito è 0.5.

3.1.5. Output

- **MULTI_HAND_LANDMARKS:** Raccolta di mani rilevate/tracciate, in cui ogni mano è rappresentata come un elenco di 21 punti di riferimento, ciascuno di essi composto da 3 coordinate (**x**, **y**, **z**). 'x' e 'y' sono normalizzati a **[0.0, 1.0]** rispettivamente per la larghezza e l'altezza dell'immagine. 'z' rappresenta la profondità del punto di riferimento con la profondità al polso come origine: minore è il valore più il punto di riferimento è vicino alla fotocamera. La grandezza di z utilizza, all'incirca, la stessa scala di 'x'.
- **MULTI_HAND_WORLD_LANDMARKS:** Raccolta di mani rilevate/tracciate, dove ciascuna mano è rappresentata come un elenco di 21 punti di riferimento nelle coordinate del mondo. Ogni punto di riferimento è composto da tre coordinate (x, y e z): coordinate 3D (in metri) del mondo reale con l'origine nel centro geometrico approssimativo della mano.
- **MULTI_HANDEDNESS:** Raccolta della manualità delle mani rilevate/tracciate (cioè si tratta di una mano sinistra o destra?). Ogni mano è composta da un'etichetta e un punteggio. "**label**" è una stringa di valore "*Left*" o "*Right*". "**score**" è la probabilità stimata della manualità prevista ed è sempre ≥ 0.5 (e la manualità opposta ha una probabilità stimata di $1 - \text{score}$).

Si noti che la manualità è determinata assumendo che l'immagine in ingresso sia speculare, cioè scattata con una fotocamera frontale/selfie con immagini capovolte orizzontalmente. In caso contrario, è necessario scambiare l'output della manualità nell'applicazione.

4. Obiettivo

L'obiettivo che ci siamo prefissati di raggiungere in questa tesina consiste nella realizzazione di un **assistente visivo virtuale** che, mediante l'utilizzo di una webcam, sarà in grado di riconoscere e tradurre i segni riprodotti dalla mano sinistra, in real-time.

Tale assistente visivo virtuale sarà un programma OpenCV, con utilizzo del Framework Mediapipe per migliorare il modello di riconoscimento della mano, capace di tradurre i gesti riprodotti dalla mano in lettere dell'alfabeto della lingua dei segni internazionale (Fig. 2.1).

Questo progetto è il primo passo nella creazione di un potenziale traduttore delle lingue dei segni, tale da poter riconoscere la comunicazione in lingua dei segni e tradurla in lingua scritta istantaneamente.

Un tale traduttore potrebbe ridurre notevolmente la barriera tra molte persone non udenti e ipoudenti in modo che possano migliorare la comunicazione con gli altri nelle loro attività quotidiane.

5. Implementazione

5.1.HandTrackingModule.py

5.1.1. Configurazione di MediaPipe

Nella nostra applicazione il modello Mediapipe è stato configurato con i valori in tabella.

Opzioni di Configurazione	Valore	Intervallo di Riferimento
STATIC_IMAGE_MODE	False	True/False
MAX_NUM_HANDS	1	1, 2, ..., n → unsigned int
MODEL_COMPLEXITY	1	0/1
MIN_DETECTION_CONFIDENCE	0.5	[0.0, 1.0]
MIN_TRACKING_CONFIDENCE	0.6	[0.0, 1.0]

```
class HandDetector:
    def __init__(self, mode=False, max_hands=1, model_complex=1, detection_con=0.5, track_con=0.6):
        self.mode = mode
        self.max_hands = max_hands
        self.model_complex = model_complex
        self.detection_con = detection_con
        self.track_con = track_con
```

Fig. 5.1

5.1.2. Rilevamento della Mano Sinistra

Per i settaggi scelti su MediaPipe, il nostro sistema sarà in grado di individuare una mano alla volta, quindi se all'interno dell'inquadratura saranno visibili entrambe le mani o più mani, verrà riconosciuta la prima mano che entrerà nel campo visivo.

La funzione che si occupa di rilevare la mano è **find_hands()**. La quale, tramite *MULTI_HAND_LANDMARK*, che è la raccolta di mani rilevate/tracciate, e *MULTI_HANDEDNESS*, che è la raccolta della manualità delle mani rilevate/tracciate, (viste più nel dettaglio nel Paragrafo 3.1.5), sarà in grado di riconoscere la mano una volta entrata nell'inquadratura e sapere distinguere tra destra e sinistra. Nel caso in cui venisse rilevata la mano sinistra, questa funzione ritornerà all'utente un feedback visivo del tracciamento della mano.

```
def find_hands(self, img):
    imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    self.results = self.hands.process(imgRGB)

    if self.results.multi_hand_landmarks:
        for handLms in self.results.multi_hand_landmarks:
            for idx, classification in enumerate(self.results.multi_handedness):
                if classification.classification[0].label == 'Left':
                    self.mp_draw.draw_landmarks(img, handLms, self.mp_hands.HAND_CONNECTIONS)
    return img
```

Fig. 5.2

5.1.3. Rilevamento della Posizione della Mano

Il rilevamento dei dati inerenti alla mano è possibile tramite la funzione **find_position()**, una funzione che trasforma tutte le coordinate dei nodi trovate tramite *MULTI_HAND_LANDMARK* mappandole in base alla dimensione del frame catturato. La funzione ritorna l'oggetto **lmList** la cui struttura verrà spiegata in dettaglio nel Paragrafo 5.2.

```

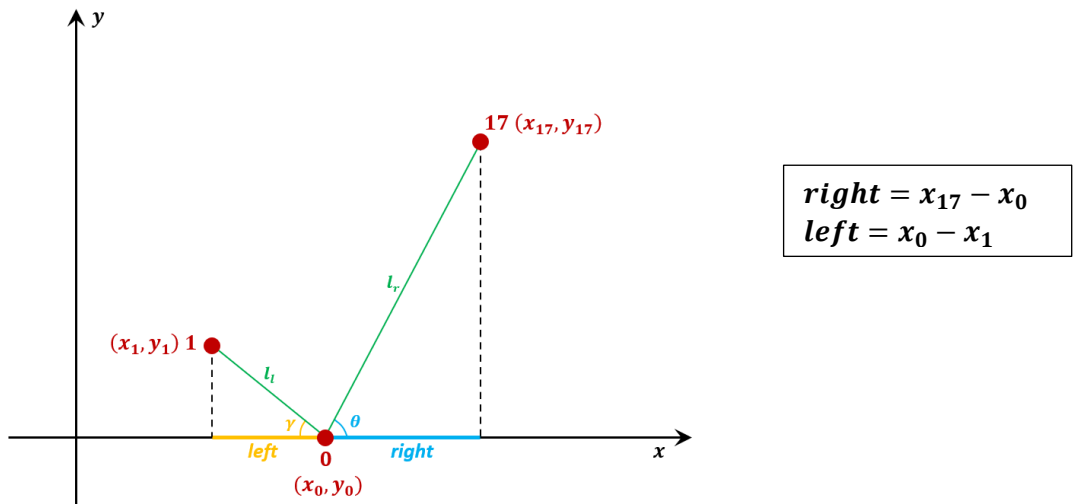
def find_position(self, img, hand_no=0, draw=True):
    lmList = []
    if self.results.multi_hand_landmarks:
        myHand = self.results.multi_hand_landmarks[hand_no]
        for idn, lm in enumerate(myHand.landmark):
            # print(idn, lm)
            h, w, c = img.shape
            cx, cy = int(lm.x * w), int(lm.y * h)
            # print(idn, cx, cy)
            lmList.append([idn, cx, cy])
            if draw:
                cv2.circle(img, (cx, cy), 15, (255, 0, 255), cv2.FILLED)
        return lmList

```

Fig. 5.3

5.1.4. Orientamento

Per semplicità consideriamo il seguente sistema di riferimento, rappresentando i nodi 0, 1 e 17 della mano, l'asse x è coerente col modello utilizzato da MediaPipe, mentre l'asse y no, in quanto la massima coordinata y, in MediaPipe, si raggiunge in corrispondenza del punto 0. Tuttavia, ai fini del calcolo dell'orientamento, questa differenza è ininfluente.



Per il Teorema di Pitagora abbiamo che:

$$right^2 + (y_{17} - y_0)^2 = l_r^2$$

$$left^2 + (y_1 - y_0)^2 = l_l^2$$

Quindi si ha che:

$$(x_{17} - x_0)^2 + (y_{17} - y_0)^2 = l_r^2 = \left(\frac{right}{\cos\theta}\right)^2$$

$$(x_0 - x_1)^2 + (y_1 - y_0)^2 = l_l^2 = \left(\frac{left}{\cos\gamma}\right)^2$$

Sostituendo:

$$(x_{17} - x_0)^2 + (y_{17} - y_0)^2 = \frac{(x_{17} - x_0)^2}{\cos^2\theta} \rightarrow \frac{(x_{17} - x_0)^2}{(x_{17} - x_0)^2 + (y_{17} - y_0)^2} = \cos^2\theta$$

$$(x_0 - x_1)^2 + (y_1 - y_0)^2 = \frac{(x_0 - x_1)^2}{\cos^2\gamma} \rightarrow \frac{(x_0 - x_1)^2}{(x_0 - x_1)^2 + (y_1 - y_0)^2} = \cos^2\gamma$$

Il coseno al quadrato è una funzione limitata nell'intervallo $[0, 1]$ per cui basta trovare il giusto intervallo entro cui la mano risulta orientata più o meno verticalmente.

$$0 \leq \cos^2 \theta = \cos_{right} \leq 0.4$$

$$0 \leq \cos^2 \gamma = \cos_{left} \leq 0.9$$

L'intervallo di \cos_{left} è più ampio in quanto la distanza che intercorre tra i nodi 0 e 1 è minore rispetto a quella fra i nodi 0 e 17.

Tutto ciò è implementato nella funzione **orientation()** (Fig. 5.4)

```
def orientation(self):
    if self.results.multi_handedness: # if a hand has been found
        flag = False
        for hand in self.results.multi_hand_landmarks:
            cos_right = (hand.landmark[17].x-hand.landmark[0].x)**2/((hand.landmark[17].x-hand.landmark[0].x)**2
                        + (hand.landmark[17].y-hand.landmark[0].y)**2)
            cos_left = (hand.landmark[0].x - hand.landmark[1].x) ** 2/((hand.landmark[0].x-hand.landmark[1].x)**2
                        + (hand.landmark[1].y-hand.landmark[0].y)**2)
            if hand.landmark[0].y > hand.landmark[1].y and hand.landmark[0].y > hand.landmark[17].y:
                if (0 <= cos_right <= 0.4) and (0 <= cos_left <= 0.9):
                    flag = True
                else:
                    flag = False
            else:
                flag = False
        return flag
```

Fig. 5.4

5.2. Recogniser.py

Nel file **Recogniser.py** è stata implementata la funzione **recogniser()**. Essa implementa un algoritmo che, mediante l'utilizzo della webcam, è in grado di riconoscere le lettere dell'alfabeto della lingua dei segni. Sarà necessario posizionarsi in modo tale che la webcam riprenda la nostra mano sinistra, quindi effettuare i possibili gesti. Di seguito ne viene descritto il funzionamento più nel dettaglio.

Per ogni frame catturato viene chiamata la funzione **grab_frame()** che effettua alcune operazioni fondamentali:

- b. Legge il frame.
- c. Effettua il mirror orizzontale del frame catturato attraverso la funzione **flip()**, per facilitare il rilevamento della mano sinistra.
- d. Rileva la mano e setta un flag che controlla che la mano rilevata sia quella sinistra.
- e. Crea una struttura dati contenente tutte le coordinate dei nodi della mano rispetto al frame catturato. Si tratta di una lista di 21 elementi, ciascun elemento rappresenta un nodo con un id e due coordinate che ne rappresentano la posizione nel frame:

lmList:

[[id0, cx, cy], [id1, cx, cy], ..., [id20, cx, cy]]
 Nodo 0 Nodo 1 Nodo 20

- f. Se tutte le Operazioni elencate prima vanno a buon fine e la mano rilevata è la sinistra allora si prosegue col calcolo delle distanze mediante la funzione **find_distances()** (Fig.5.5). Viene creata una struttura dati di 21 elementi, ciascun elemento rappresenta un nodo che contiene una matrice di 21 distanze di Eulero, tra il nodo in questione e tutti gli altri (sé stesso compreso).

distMatrix:

[[d(0,0), d(0,1), ..., d(0,20)], ..., [d(20,0), d(20,1), ..., d(20,20)]]
 Nodo 0 Nodo 20

Le distanze vengono calcolate e successivamente normalizzate rispetto alla lunghezza del palmo per permettere al programma di riconoscere i segni anche se la mano si trova a distanze diverse dalla webcam.

```
def find_distances(lmList):
    distMatrix = np.zeros([len(lmList), len(lmList)], dtype='float')
    palmSize = ((lmList[0][1]-lmList[9][1])**2+(lmList[0][2]-lmList[9][2])**2)**(1./2.)
    for row in range(0, len(lmList)):
        for column in range(0, len(lmList)):
            distMatrix[row][column] = (((lmList[row][1]-lmList[column][1])**2 +
                                           (lmList[row][2]-lmList[column][2])**2)**(1./2.))/palmSize
    return distMatrix
```

Fig. 5.5

g. Calcolate le distanze di Eulero fra i nodi della mano inerenti al gesto rilevato, vengono quindi confrontate con le distanze calcolate e salvate per ognuno dei simboli noti (Vedi il paragrafo *StoreGesture.py*). Per far questo viene chiamata la funzione **find_gesture()**:

- Per ognuno dei simboli noti (A, B, ..., Z) viene calcolato un errore tra le distanze del simbolo noto e quelle del simbolo rilevato, tramite la funzione **find_error()** (Fig. 5.6).
- L'errore non viene calcolato per ogni nodo, ma solo per alcuni, quelli più significativi, i **keyPoints**, precedentemente definiti (Fig. 5.6 e punti in rosso in Fig. 5.7).

```
keyPoints = [0, 4, 5, 9, 13, 17, 8, 12, 16, 20, 2, 6, 10, 14, 18]
def find_error(gestureMatrix, unknownMatrix):
    error = 0
    for row in keyPoints:
        for column in keyPoints:
            error = error + abs(gestureMatrix[row][column] - unknownMatrix[row][column])
    return error
```

Fig. 5.6

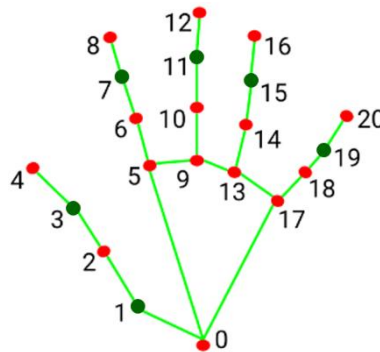


Fig. 5.7

- L'errore risultante è la somma di tutti quelli calcolati tra un keyPoint e l'altro.
- Ottenuto questo valore rispetto ad ogni gesto, se ne calcola il minimo.

```
# finds the min in error array
for i in range(0, len(errorArray), 1):
    if errorArray[i] < errorMin:
        errorMin = errorArray[i]
        minIndex = i
```

Fig. 5.8

- Successivamente, l'errore minimo viene sottoposto ad un ulteriore controllo di tolleranza, cioè, il calcolo del minimo non è sufficiente per stabilire che il gesto rilevato sia effettivamente corrispondente a quello trovato, ma è necessario stabilire una soglia entro cui un determinato risultato può essere accettato o meno.

Abbiamo deciso di aumentare la soglia di tolleranza per alcune lettere, in quanto più difficili da riconoscere.

```

if errorMin < tol_max:
    if gestNames[minIndex] == 'I' or gestNames[minIndex] == 'S' or gestNames[minIndex] == 'W':
        gesture = gestNames[minIndex]
    elif errorMin < tol:
        gesture = gestNames[minIndex]
    elif errorMin >= tol:
        gesture = 'Unknown'

```

tol = 20
tol_max = 35

Fig. 5.9

- Il gesto che supera tutti i controlli sarà quello restituito dalla funzione altrimenti verrà visualizzata la scritta “Unknown”.
- h. Dopo aver trovato il gesto corrispondente a quello rilevato, viene effettuato un ulteriore controllo sull’orientamento della mano, ampiamente spiegato nel Paragrafo 5.1.5. Quindi se la mano è orientata nel giusto modo il simbolo verrà riconosciuto e mostrato a video altrimenti l’utente verrà sollecitato a riposizionare la mano correttamente.

5.3.StoreGestures.py

Al fine di creare un “dizionario” di simboli noti da confrontare con quelli rilevati real-time è stato scritto lo script **StoreGesture.py** che utilizza MediaPipe per rilevare la mano e salvare i dati inerenti a uno specifico simbolo tramite i seguenti passaggi base:

- a. Come prima cosa viene chiesto da terminale all’utente quale lettera vuole salvare.
- b. Una volta specificata la lettera viene eseguito MediaPipe e attivata la webcam per rilevare la mano e permettere all’utente di scegliere una giusta posizione e delle corrette condizioni di luminosità (Vedi Capitolo 8).
- c. Quando l’utente è pronto può inserire “S” da terminale per avviare lo storing dei dati per quello specifico simbolo (Fig. 5.10).

```

Which Letter? --> A
Press S when Ready -->
S

```

Fig. 5.10

- d. Lo storing avviene considerando un numero di campioni pari a 100, ognuno dei quali è un frame (che viene considerato solo se è stata rilevata una mano sinistra nel frame). Per ognuno dei 100 frame, durante i quali l’utente deve mantenere la posizione corretta, vengono salvate le distanze tra i nodi della mano creando una struttura dati **unknown_gesture_samples** che sarà una lista di 100 matrici di distanze (cioè la **distMatrix**, vista nel Paragrafo 5.2), tramite la funzione **find_distances()**.

```

unknown_gesture_samples:
[distMatrix1, distMatrix2, ..., distMatrix100]
  Sample 1      Sample 2      Sample 100

```

- e. Salvati i 100 campioni, viene poi calcolata la media aritmetica tra questi tramite la funzione **calculate_average()** (Fig. 5.11) al fine di generare un’unica **distMatrix** che sarà poi quella che verrà salvata in un file json il cui formato è **NOME_LETTERA.json** (Fig. 5.12).

```

def calculate_average(samplesList):
    if len(samplesList) > 0:
        sumG = samplesList[0]
        for i in range(1, len(samplesList)):
            sumG = np.add(sumG, samplesList[i])
        avg = sumG/len(samplesList)
        return avg
    else:
        return -1

```

Fig. 5.11

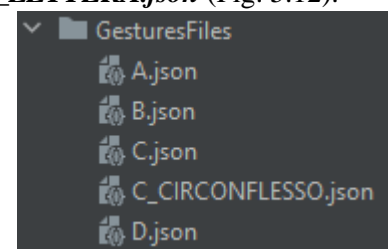


Fig. 5.12

Il codice viene mostrato in Fig. 5.13.

```
if start is True:
    if i < samples:
        cv2.putText(frame, 'Storing ' + _letter.upper(), (2, 50), cv2.FONT_HERSHEY_SIMPLEX, 1.5,
                     (0, 0, 125), 3, cv2.LINE_AA)
        unknown_gesture_sample = find_distances(lmList) # save the sample
        unknown_gesture_samples.append(unknown_gesture_sample) # add the sample to the list of samples
        i = i + 1
    else:
        if averageFlag is True:
            res = calculate_average(unknown_gesture_samples) # calculate the average
            save_in_file(res, letter) # save the result into a file
            averageFlag = False
        else:
            return
```

Fig. 5.13

5.4. RecogniserSigned.py

Al fine di migliorare il nostro algoritmo è stato implementato il **RecogniserSigned.py** che segue gli stessi passaggi del *Recogniser.py*, ma calcola le distanze in un modo diverso. Infatti le distanze di Eulero calcolate prima non tengono conto della posizione di un nodo rispetto a un altro, quindi abbiamo implementato una funzione **find_distances()** che, dati due nodi e le loro coordinate x, y, vada a calcolare la differenza tra le coordinate, preservando quindi il segno dell'operazione e salvando due valori piuttosto che uno solo (Fig. 5.14).

```
def find_distances(lmList):
    dist_matrix_x = np.zeros([len(lmList), len(lmList)], dtype='float')
    dist_matrix_y = np.zeros([len(lmList), len(lmList)], dtype='float')
    palmSize = ((lmList[0][1] - lmList[9][1]) ** 2 + (lmList[0][2] - lmList[9][2]) ** 2) ** (1. / 2.)

    for row in range(0, len(lmList)):
        for column in range(0, len(lmList)):
            dist_matrix_x[row][column] = (lmList[row][1] - lmList[column][1]) / palmSize
            dist_matrix_y[row][column] = (lmList[row][2] - lmList[column][2]) / palmSize

    distMatrix = [dist_matrix_x, dist_matrix_y]
    return distMatrix
```

Fig. 5.14

Ci saranno adesso due **distMatrix**, una per ogni coordinata 2D: **distMatrix_x** e **distMatrix_y**.

distMatrix: [**distMatrix_x**, **distMatrix_y**]

Dove:

distMatrix_x:
[[(x0-x0), ..., (x0-x20), ..., [(x20-x0), ..., (x20-x20)]]
 Nodo 0 Nodo 20

distMatrix_y:
[[(y0-y0), ..., (y0-y20), ..., [(y20-y0), ..., (y20-y20)]]
 Nodo 0 Nodo 20

Il calcolo dell'errore verrà dunque calcolato per ogni coordinata e il valore risultante sarà una coppia di valori (Fig. 5.15):

error: [**error_x**, **error_y**]

```
def find_error(gestureMatrix, unknownMatrix):
    errorX = 0
    errorY = 0

    for row in keyPoints:
        for column in keyPoints:
            errorX = errorX + abs(gestureMatrix[0][row][column] - unknownMatrix[0][row][column])
            errorY = errorY + abs(gestureMatrix[1][row][column] - unknownMatrix[1][row][column])

    error = [errorX, errorY]
    return error
```

Fig. 5.15

Anche il confronto verrà effettuato per ognuna delle coordinate. Anche in questo caso abbiamo più di un livello di tolleranza, in base ai vari test effettuati (Fig. 5.16).

```
if errorMin[0] < tol_ISE and errorMin[1] < tol_ISE:
    if gestNames[minIndex] == 'I' or gestNames[minIndex] == 'S' or gestNames[minIndex] == 'E':
        gesture = gestNames[minIndex]
    elif errorMin[0] < tol_max and errorMin[1] < tol_max:
        if gestNames[minIndex] == 'K' or gestNames[minIndex] == 'O' or gestNames[minIndex] == 'P' \
            or gestNames[minIndex] == 'C_CIRCONFLESSO' \
            or gestNames[minIndex] == 'G' or gestNames[minIndex] == 'M' or gestNames[minIndex] == 'T' \
            or gestNames[minIndex] == 'N' or gestNames[minIndex] == 'Q' or gestNames[minIndex] == 'E':
            gesture = gestNames[minIndex]
        elif errorMin[0] < tol and errorMin[1] < tol:
            gesture = gestNames[minIndex]
        elif errorMin[0] >= tol and errorMin[1] >= tol:
            gesture = 'Unknown'
    return gesture
```

tol = 25
tol_max = 40
tol_ISE = 50

Fig. 5.16

5.5.StoreGestureSigned.py

Il file **StoreGestureSigned.py** è identico allo **StoreGesture.py**, ovviamente adesso tutti i calcoli saranno effettuati per ogni coordinata e i file json conterranno quindi due matrici delle distanze, una per la coordinata x e una per la coordinata y.

In Fig. 5.17 è mostrata la funzione **calculate_average()** che effettua la media per x e per y.

```
def calculate_average(samplesList):
    if len(samplesList) > 0:
        sumX = samplesList[0][0]
        sumY = samplesList[0][1]
        for i in range(1, len(samplesList)):
            sumX = np.add(sumX, samplesList[i][0])
            sumY = np.add(sumY, samplesList[i][1])
        avgX = sumX/len(samplesList)
        avgY = sumY/len(samplesList)

        avg = [avgX, avgY]
        return avg
    else:
        return -1
```

Fig. 5.17

6. Valutazione dei Metodi Implementativi

Nel capitolo precedente abbiamo illustrato i due approcci che abbiamo usato per lo sviluppo del **Recogniser.py** e del **RecogniserSigned.py**.

Ritenendo entrambi gli approcci validi e con prestazioni più o meno equivalenti, in quanto il primo riconosce i gesti con più facilità ma ha meno accuratezza mentre il secondo li riconosce con meno facilità ma in modo più distintivo, abbiamo sviluppato dei test per poter stabilire in modo oggettivo quale dei due approcci portasse risultati migliori.

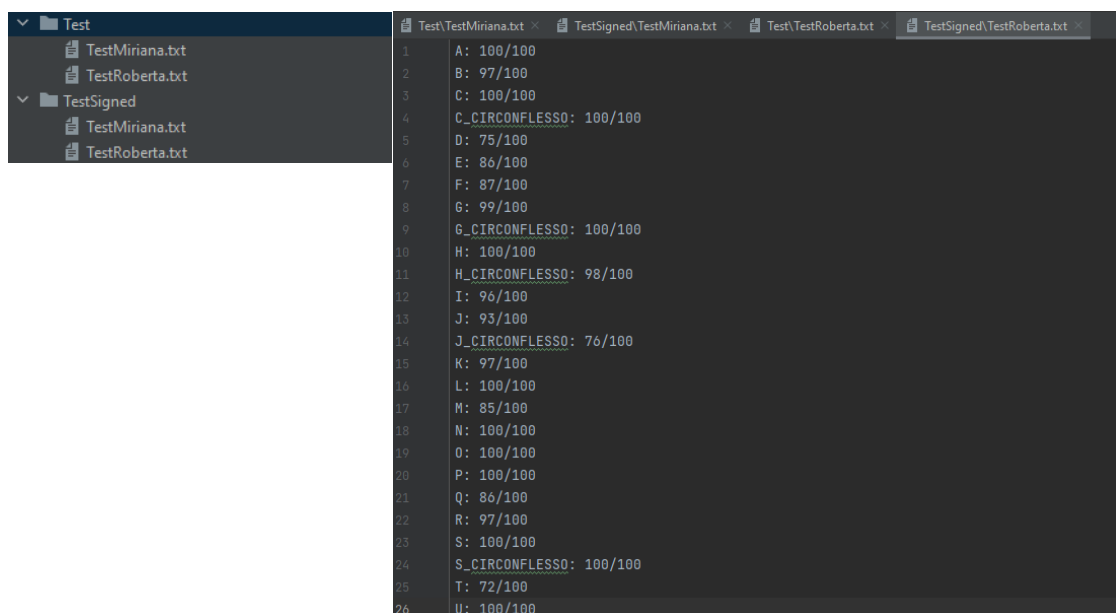
Il test che abbiamo effettuato sui due metodi è molto semplice in quanto va a vedere per ogni lettera quante volte, su un campione di 100 frames, il programma riconosce la lettera dato il gesto corrispettivo, in poche parole determina i true positive.

Il pezzo di codice in Fig.6.1, che spiega come abbiamo calcolato i true positive, è estratto da **Test.py** (equivalente in **TestSigned.py**) all'interno della funzione **test()**:

```
if start is True:
    if i < samples:
        cv2.putText(frame, 'Testing ' + _letter.upper(), (2, 50), cv2.FONT_HERSHEY_SIMPLEX, 1.5,
                     (0, 0, 125), 3, cv2.LINE_AA)
        if myGesture == _letter:
            true_positive += 1
        i = i + 1
    else:
        print("Correct Guess: {}/{}".format(true_positive, samples))
        save_in_file(test_path, _letter, true_positive)
    return
```

Fig. 6.1

Dopo aver determinato, a testa, la percentuale di riuscita di tutte le lettere dell'alfabeto, per entrambi gli approcci, ed averli salvati in dei file la cui tipologia è mostrata in Fig.6.2, abbiamo calcolato la riuscita media, tramite l'**AverageCalculator.py**, ottenendo una percentuale di riuscita del **Recogniser.py** del **90.99%** e del **RecogniserSigned.py** del **92.04%**.



The image shows a file explorer on the left with a tree view containing 'Test' and 'TestSigned' folders, each with 'TestMiriana.txt' and 'TestRoberta.txt' files. On the right, a text editor displays the contents of 'TestSigned/TestMiriana.txt', which lists recognition results for letters A through U. Each line shows the letter followed by a fraction of correct guesses out of 100.

Letter	Correct Guesses / Total
A	100/100
B	97/100
C	100/100
C_CIRCONFLESSO	100/100
D	75/100
E	86/100
F	87/100
G	99/100
G_CIRCONFLESSO	100/100
H	100/100
H_CIRCONFLESSO	98/100
I	96/100
J	93/100
J_CIRCONFLESSO	76/100
K	97/100
L	100/100
M	85/100
N	100/100
O	100/100
P	100/100
Q	86/100
R	97/100
S	100/100
S_CIRCONFLESSO	100/100
T	72/100
U	100/100

Fig. 6.2

In conclusione, prendendo in considerazione solo i test compiuti da noi, possiamo quindi evincere che entrambi i metodi sono abbastanza performanti, ma il migliore è il **RecogniserSigned.py**.

7. Risultato Ottenuto

L'obiettivo che ci eravamo prefissati di ottenere con questa tesina è stato raggiunto, in quanto abbiamo ottenuto un *assistente visivo virtuale* che, in real-time, è in grado di riconoscere tutti i gesti rappresentanti le lettere dell'alfabeto del Signuno.

In seguito, alcune immagini dimostrative rappresentanti il riconoscimento di qualche gesto del Signuno, usando il **Recogniser Signed**:

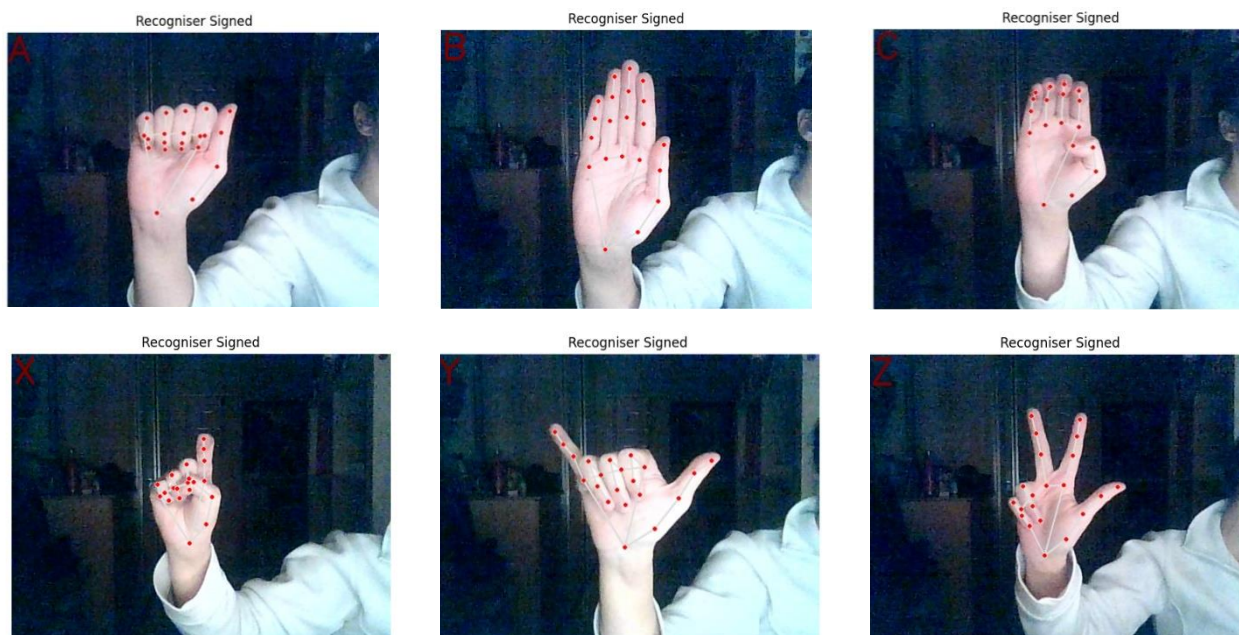


Fig. 7.1

Ciò che potrebbe essere migliorato è il *grado di precisione del riconoscimento dei gesti*, in quanto non sempre il programma è in grado di riconoscerli correttamente. A volte, infatti, pur facendo il gesto correttamente, come riportato in figura 2.1, quest'ultimo non viene riconosciuto o scambiato per un altro, mentre altre volte pur non eseguendo il gesto nel modo delineato in figura 2.1, il programma riesce a riconoscerlo.

8. Requisiti di Funzionamento

8.1. Condizioni di Luminosità

I modelli MediaPipe sono addestrati per funzionare bene in "condizioni normali" di buona illuminazione e buona qualità della fotocamera.

Abbiamo notato che, in condizioni di basso contrasto, il modello ha difficoltà a rilevare la mano e dunque a riconoscere i segni. È dunque necessario che ci sia un alto contrasto tra la mano e lo sfondo, una situazione come in Fig.7.1 è considerata accettabile.

8.2. Requisiti Hardware e Software

Le caratteristiche hardware e software dei computer che abbiamo usato per lo sviluppo e test del progetto sono le seguenti:

- **CPU:** Intel Core i7 8th Gen
- **GPU:** Nvidia GeForce GTX 1050
- **Webcam:** HD 720p
- **Sistema operativo:** Windows 10 Home, x64
- Python versione 3.9
- **IDE:** PyCharm
- **Librerie:** Mediapipe, OpenCV, Matplotlib e altre più basiche