Indice

1 — Introduzione MediaPipe

L'applicazione sfrutta le potenzialità offerte dalla libreria `MediaPipe`, che utilizza soluzioni basate su Machine Learning nel contesto dello streaming multimediale. In particolare, noi abbiamo sfruttato la soluzione per il riconoscimento del movimento delle mani: **MediaPipe Hands**.

Il nostro scopo è stato quello di interpretare ed associare movimenti delle mani a determinate gesture in modo tale da poter gestire la lettura di un documento testuale, come un pdf.
Le funzionalità che abbiamo scelto di implementare sono:

- zoom
- scorrimento verticale e laterale delle pagine
- screenshot

Abbiamo deciso di realizzare l'applicazione per ambiente Android, tramite l'utilizzo di Java ed Android Studio.

1.1 MediaPipe Hands

MediaPipe Hands offre funzionalità per il tracciamento delle mani e delle dita, utilizzando il ML per individuare 21 punti di riferimento a 3 dimensioni di una mano da un solo fotogramma.
Viene sfruttata una pipeline composta da due modelli che lavorano insieme:

1. un modello per il **rilevamento del palmo** che opera sull'immagine completa e restituisce un riquadro di delimitazione orientato (*ritaglio*) della mano.
2. un modello che opera sulla regione dell'immagine ritagliata definita dal modello precedente e restituisce i **punti di riferimento della mano** a 3 dimensioni.

Il fatto di fornire al modello per i punti di riferimento un'immagine della mano accuratamente ritagliata diminuisce di gran lunga le elaborazioni necessarie, aumentando

l'efficienza.
Inoltre, nella pipeline i *ritagli* della mano possono essere generati anche in base ai punti di riferimento identificati nel frame precedente, e solo quando il modello del punto di riferimento non è più in grado di identificare la presenza della mano viene invocato il modello di rilevamento del palmo.

Specifica delle pipeline

La pipeline è implementata come un **grafo** che utilizza un **sottografo per il tracking dei punti** dal modulo `landmark` e renderizza usando un sottografo dedicato al rendering.
Il sottografo per il tracking dei punti utilizza al suo interno un'altro sottografo per il tracking e un **sottografo per il rilevamento della mano** dal modulo `palm detection`.

1.1.1 Modelli

Palm Detection Model

Per la rilevazione delle posizione iniziali delle mani, viene adottato un modello di rilevamento *single-shot* ottimizzato per l'utilizzo in tempo reale.
La rilevazione delle mani è però un compito complesso per via della loro omogeneità visiva, a differenza per esempio del viso, che invece ha dei pattern a più alto contrasto come nelle zone degli occhi e della bocca. Viene utilizzato perciò un contesto aggiuntivo, come possono essere la posizione del braccio o del corpo di una persona, in modo tale da poter individuare la mano più facilmente, aumentando l'efficienza.

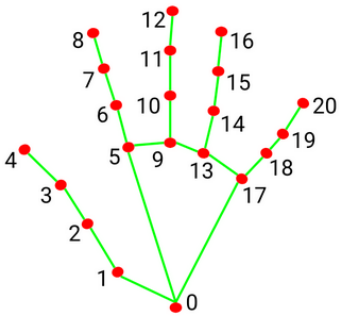
Il primo componente utilizzato è un **rilevatore del palmo**, piuttosto che della mano, in quanto *ritagli* di oggetti rigidi come palmi e pugni sono nettamente più semplici rispetto a quelli delle dita.
In secondo luogo, viene utilizzato un de/codificatore per l'estrazione di caratteristiche dall'immagine in modo da avere 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 derivanti dalla varianza su scala elevata.

Viene stimata una precisione di circa il 96% nella rilevazione del palmo.

Hand Landmark Model

Dopo il rilevamento del palmo, questo modello esegue, all'interno delle regioni rilevate, la localizzazione precisa di 21 punti della mano tramite **regressione**, ovvero la previsione diretta delle coordinate.
Il modello apprende quindi una rappresentazione della posizione del'interno della mano coerente ed è robusto anche a mani parzialmente visibili.

Per arrivare a questo risultato sono state annotate manualmente circa 30 mila immagini dal mondo reale con 21 coordinate 3D (valore Z preso tramite mappa di profondità). E' stato inoltre utilizzato un modello di mano sintetica su vari sfondi, mappato alle corrispondenti coordinate, in modo tale da avere un accuratezza maggiore.



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

1.1.2 API

1.1.3 Opzioni di configurazione

`STATIC_IMAGE_MODE` Se settata a `false` (default), le immagini in ingresso vengono trattate come un flusso video, cercando di rilevare le mani nelle prime immagini di input e localizzando poi i punti di riferimento della mano. Nelle immagini successive, una volta che tutte le mani `MAX_NUM_HANDS` vengono rilevate e i corrispondenti punti di riferimento della mano sono localizzati, si limita a tracciare quei punti di riferimento senza invocare un altro rilevamento fino a quando non perde traccia di una qualsiasi delle mani. Questo riduce la latenza ed è ideale per l'elaborazione di fotogrammi video.
Se impostato a `true`, il rilevamento manuale viene eseguito su ogni immagine di input, ideale per elaborare un batch di immagini statiche, magari non correlate.

`MAX_NUM_HANDS` Massimo numero di mani da rilevare (2 a default).

`MODEL_COMPLEXITY` Complessità del modello di landmark: 0 o 1 (default). La precisione del punto di riferimante e la latenza aumentano con la complessità.

`MIN_DETECTION_CONFIDENCE` Valore minimo di fiducia $[0.0, 1.0]$ nei confronti del modello di rilevamento della mano per considerare il rilevamento della mano corretto. A defult è 0.5.

`MIN_TRACKING_CONFIDENCE` Valore minimo di fiducia $[0.0, 1.0]$ nei confronti del modello di tracking per considerare il rilevamento dei punti di riferimento corretto. Altrimenti, il modello di rilevamento della mano (palmo) verrà richiamato all'immagine in input successiva. Di conseguenza, aumentando questo valore avremo una robustezza maggiore, a discapito di un aumento di latenza. A defult è 0.5.

Output

`MULTI_HAND_LANDMARKS` Raccolta di mani rilevate/tracciate, ognuna rappresentata come una lista di 21 punti di riferimento a 3 dimensioni. In ogni punto `x` e `y` sono normalizzati ($[0.0, 1.0]$) rispetto alla larghezza e all'altezza dell'immagine. La coordinata `z` rappresenta invece la profondità del punto rispetto a quella del polso (considerato l'origine). Minore è il valore della `z`, più il punto è considerato vicino alla fotocamera.

`MULTI_HAND_WORLD_LANDMARKS` Come il precedente, ma i punti sono valutati come coordinate 3D reali in metri, con l'origine al centro geometrico approssimativo della mano.

`MULTI_HAND_HANDEDNESS` Raccolta delle *handedness* della mani (destra/sinistra). Ogni mano è composta da un'etichetta (`label`) e da un punteggio (`score`). L'etichetta è una stringa di valore `left` o `right`, mentre il punteggio rappresenta la stima della probabilità di predizione corretta della mano (sempre ≥ 0.5, mentre l'*handedness* opposta è $1-$`score`). L'*handedness* viene valutata considerando che l'immagine in input è specchiata.

OpenGL ES

Per il rendering dell'input e dell'output viene utilizzato `OpenGL ES`, un sottoinsieme delle librerie grafiche di `OpenGL` pensato per dispositivi embedded.
Nella porzione di codice sottostante viene mostrato un esempio di utilizzo.

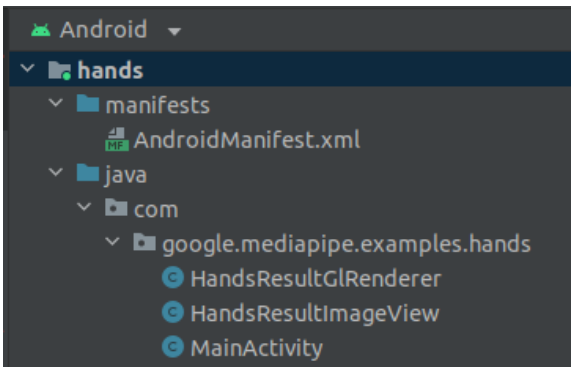
```
// Vengono prima settate le opzioni di configrazione citate precedentemente
HandsOptions handsOptions =
    HandsOptions.builder()
        .setStaticImageMode(false)
```

```
            .setMaxNumHands(2)
            .setRunOnGpu(true).build();
    Hands hands = new Hands(this, handsOptions);
    hands.setErrorListener(
        (message, e) -> Log.e(TAG, "MediaPipe Hands error:" + message));

    // Inizializzato una nuova istanza di CameraInput
    // e connessione alla soluzione MediaPipe Hands
    CameraInput cameraInput = new CameraInput(this);
    cameraInput.setNewFrameListener(
        textureFrame -> hands.send(textureFrame));

    // Inizializzato una istanza di GlSurfaceView tramite
    // ResultGlRenderer<HandsResult> che fornisce un interfaccia
    // per eseguire codice di rendering OpenGL definito dall'utente
    SolutionGlSurfaceView<HandsResult> glSurfaceView =
        new SolutionGlSurfaceView<>(
            this, hands.getGlContext(), hands.getGlMajorVersion());
    glSurfaceView.setSolutionResultRenderer(new HandsResultGlRenderer());
    glSurfaceView.setRenderInputImage(true);

    hands.setResultListener(
        handsResult -> {
        if (result.multiHandLandmarks().isEmpty()) {
            return;
        }
        NormalizedLandmark wristLandmark =
            handsResult.multiHandLandmarks().get(0).getLandmarkList()
                .get(HandLandmark.WRIST);
        Log.i(
            TAG,
            String.format(
                "MediaPipe Hand wrist normalized coordinates
                (value range: [0, 1]): x=%f, y=%f",
                wristLandmark.getX(), wristLandmark.getY()));
        // Richiesta di rendering a OpenGL
        glSurfaceView.setRenderData(handsResult);
        glSurfaceView.requestRender();
        });

    // Avvio della fotocamera dopo che GLSurfaceView si è connesso
    glSurfaceView.post(
        () -> cameraInput.start(
                this,
                hands.getGlContext(),
                CameraInput.CameraFacing.FRONT,
                glSurfaceView.getWidth(),
                glSurfaceView.getHeight()));
```

2 — Struttura del progetto

2.1 Overview

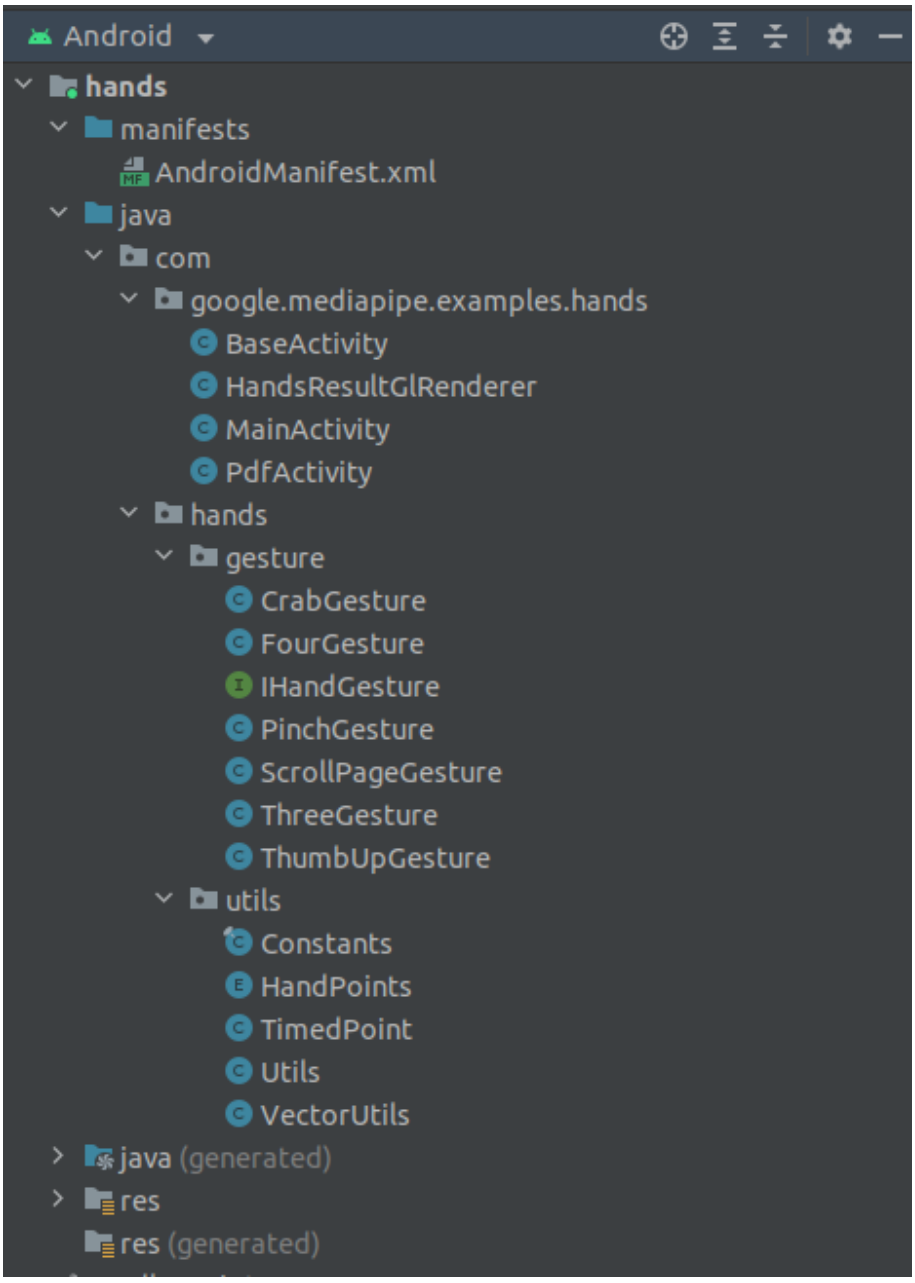
Come input alla realizzazione della nostra applicazione, siamo partiti da una demo offerta direttamente da MediaPipe. In questa applicazione viene data la possibilità all'utente di visualizzare la mappatura delle proprie mani scegliendo diverse modalità di acquisizione dell'immagine. Oltre a file ausiliari come il *manifest* o gli *xml* di configurazione, il cuore del progetto è rappresentato da 3 file nella cartella `hands/java/com/google/mediapipe/examples/hands`.



Ovviamente, `MainActivity` corrisponde all'entry point dell'applicazione ed eredita infatti da `AppCompatActivity`.

Nonostante noi ci siamo concentrati solo sull'input da streaming video, l'applicazione mette a disposizione 3 diverse modalità di acquisizione dell'input, rappresentate dall'enumerativo `InputSource` all'interno del quale troviamo `CAMERA`, `VIDEO` e `IMAGE`.

2.2 Applicazione finale



All'interno della cartella `hands/java/com/google/mediapipe/examples/hands` abbiamo potuto eliminare la classe `HandsResultImageView`, utile solo con acquisizione dell'input tramite immagine statica; mentre la classe `HandsResultGlRenderer`, per il rendering su schermo, è stata lasciata sostanzialmente inalterata.
Sono state invece applicate delle modifiche consistenti alla classe `MainActivity` (aggiungendo anche altre due classi *Activity*) per implementare la logica applicativa.

2.2.1 BaseActivity, MainActivity e PdfActivity

Mentre prima era `MainActivity` ad ereditare da `AppCompatActivity`, ora questo avviene per `BaseActivity`, da cui erediteranno poi la classe `PdfActivity` e la nuova `MainActivity`.
Abbiamo scelto di adottare questa strategia implementativa per suddividere in modo chiaro e pulito le funzionalità esposte dalle due classi:

- `MainActivity`: si occupa di definire il layout dell'applicazione nel caso in cui si voglia visualizzare su tutto lo schermo la propria mano, con alcune `TextView` per mostrare il riconoscimento di determinate **gesture**.

- `PdfActivity`: corrisponde al cuore dell'applicazione, definendo un layout nel quale è presente un file `pdf` da leggere ed una piccola vista della fotocamera interna dello smartphone. Quest'ultima ha lo scopo di mostrare all'utente il movimento della propria mano, tramite la quale avrà la possibilità di interagire con il file.

Entrambe le *activities* hanno quindi lo scopo di mostrare la mano ed il riconoscimento di gesture, la prima *"loggando"* l'eventuale riconoscimento, la seconda interagendo con il pdf.
Per fare questo, all'interno di ognuna, viene definito un **listener** che controlla continuamente l'eventuale associazione di un movimento della mano ad una gesture. Sono state infatti create 6 classi, una per gesture, che si occupano di effettuare il controllo sulla base dei **landmarks** passati in input dalle *activities*.

Prima di concentrarci su questE classi, di seguito vengono esposte caratteristiche più dettagliate di ognuna delle *activitites* presentate sopra.

BaseActivity

Questa classe funge da contenitore per tutte quelle variabili e funzioni comuni sia a `MainActivity` sia a `PdfActivity` (perciò dichiarate con la clausola `protected`). Questa permette di evitare la ridondanza di codice.

Variabili Alcune delle variabili comuni definite sono:

- `Hands`, tramite la quale viene recuperato il **listener** nominato sopra, da cui poi otteniamo i risultati del mapping effettuato dalla pipeline di riconoscimento di MediaPipe.

- `InputSource`, che rappresenta la modalità di acquisizione dell'immagine (da noi sarà sempre `CAMERA`).

- `CameraInput`, che rappresenta la ricezione dell'input dalla fotocamera.

Abbiamo poi `SolutionGlSurfaceView<HandsResult>` e `HandsResultGlRenderer` per la definizione del layout e per il rendering.
Infine, sono presenti le istanze di tutte quelle classi che effettuano il rilevamento delle gesture, che, come detto in precedenza, sono necessarie ad entrambe le `activities`, anche se con scopi diversi.

Metodi Essendo che questa classe eredita da `AppCompatActivity`, è necessario effettuare l'`override` dei seguenti metodi:

- `onCreate`, accetta un `Bundle` in input, utilizzato per chiamare la `onCreate` della classe estesa. In Android Studio i *Bundle* si utilizzano solitamente per il passaggio di dati da un attività all'altra: in questo caso abbiamo un istanza di uno stato salvato precedentemente (e.g. l'applicazione era stata messa in background). All'interno di questa funzione è solito definire il layout della pagina, ma abbiamo scelto di farlo solo nelle *activities specializzate* che erediteranno da `BaseActivity`.

- `onPause`, nella quale viene chiamata la `onPause` della classe estesa per poi nascondere la visibilità del layout (`glSurfaceView`) e fermare infine il flusso video in input dalla fotocamera.

- `onResume`, dove dopo essere stata chiamata la `onResume` della classe estesa, si istanzia un nuovo `CameraInput` (passandogli l'*activity* corrente) per poi settare un nuovo *FrameListener*. Successivamente viene utilizzata l'istanza di `SolutionGlSurfaceView<HandsResult>` per chiamare il metodo interno `post`, passandogli la funzione interna `startCamera()` (il cui scopo è quello di inizializzare lo streaming video). In questo modo, tramite la `post` ereditata dalla classe `View`, viene eseguita la funzione passata ed esposto l'output nell'interfaccia utente.

Sono presenti poi ulteriori funzioni che vengono utilizzate da `MainActivity` e `PdfActivity` per inizializzare la pipeline di acquisizione dell'input e per terminarla. Verranno analizzate in seguito.

MainActivity

Abbiamo detto che lo scopo di questa *activity* è quello di mostrare su schermo l'eventuale riconoscimento di determinate gesture. Perciò, oltre alle variabili definite

nella classe estesa, le uniche variabili globali presenti sono una serie `TextView`, una per ogni gesture.
Viene effettuato l'`override` delle funzioni ereditate da `AppCompatActivity` e, mentre `onPause` e `onResume` chiamano solamente le relative funzioni della classe `BaseActivity`, la `onCreate` si occupa inoltre di inizializzare lo streaming video e di definire il layout della pagina, istanziando anche un nuovo bottone per accedere alla funzionalità di interazione coi file `pdf`.

Per quanto riguarda l'inizializzazione dello streaming video, vengono utilizzate due funzioni:

- `setupLiveDemoUiComponents`, all'interno della quale viene definito un bottone per attivare lo streaming che, una volta cliccato, causerà la chiamata alla stessa funzione della classe estesa (il cui scopo è quello di fermare la pipeline di acquisizione attuale (nel caso in cui la modalità di acquisizione sia diversa da `CAMERA`)). Successivamente, viene chiamata la funzione che implementa effettivamente la pipeline, mostrata qui sotto.
- `setupStreamingModePipeline`, questa funzione è divisa sostanzialmente in 3 parti, oltre ad una preliminare nella quale vengono inizializzate le `TextView`.
    1. chiamata alla funzione `firstSetupStreamingModePipeline` di `BaseActivity` che crea delle nuove istanze di `HandsResultGlRenderer`, `Hands` e `glSurfaceView`. Tramite essa inizia lo streaming dalla fotocamera e, concorrentemente, la pipeline per il mapping dei **landamrks** della mano.
    2. settato il **listener** per il riconoscimento delle gesture, all'interno del quale vengono effettuate delle chiamate alle classi per il controllo e mostrato l'eventuale riconoscimento colorando le `Textview` definite prima.
    3. chiamata alla funzione `lastSetupStreamingModePipeline` di `BaseActivity` che chiama il metodo `post` di `glSurfaceView` visto in precedenza e definisce il nuovo `FrameLayout` con i dati trovati nella fase precedente.

PdfActivity

Il funzionamento di questa classe è analogo a quella mostrata in precedenza. Viene ovviamente differenziato il layout della pagina e, all'interno del *listener* della funzione `setupStreamingModePipeline`, vengono associate le varie gesture riconosciute ad azioni che permettono di interagire con il file `pdf` che si sta visualizzando.

3 — Metodologie di riconoscimento delle gesture

Come già specificato in **??**, il framework mette a disposizione due collezioni distinte di punti tridimensionali:

- la lista dei landmark a coordinate globali con origine nel centro geometrico approssimato della mano
- la lista dei landmark con coordinate X e Y normalizzate all'intervallo [0.0 - 1.0] per le dimensioni dell'immagine in input, la coordinata Z trova il suo centro nella posizione del landmark del polso

Abbiamo utilizzato entrambe le strutture per la rilevazione delle gesture, insieme a una combinazione delle le tecniche descritte di seguito, impiegando l'una o l'altra a seconda del tipo di azione alla quale la particolare gesture è collegata (vedi **??**).

3.1 Distanze tra landmark

Il primo metodo che abbiamo implementato per discriminare una gesture dall'altra consiste semplicemente nel considerare la distanza tra landmark impiegando un sottoinsieme di punti rilevanti per la particolare gesture.

Abbiamo rilevato le distanze tra landmark caratterizzanti una particolare posizione delle dita che forma la gesture voluta in modo empirico, sfruttando opportune stringhe di log per visualizzare in tempo reale le distanze dei punti, per poi compilare una struttura dati usata come riferimento per identificare la gesture.

In seguito a queste analisi abbiamo notato, come ipotizzato in precedenza, le piccole ma non trascurabili fluttuazioni dei valori ottenuti dal framework, che impedivano un accurato riconoscimento della gesture; questo ha portato a considerare una soglia di errore sulle distanze rilevate, effettivamente individuando dei range di posizionamento delle dita che costituivano le gesture.

Aggiungendo nuove gesture alla nostra applicazione, abbiamo riscontrato un livello di inaccuratezza sempre più alto, manifestato nella forma di gesture rilevate contemporaneamente ad altre o, in certi casi, non rilevate affatto.

3.2 Normalizzazione a livelli

Dati i risultati della prima fase di test, abbiamo realizzato che molti fallimenti nella rilevazione di una gesture erano dovuti alla naturale differenza di dimensione tra le nostre mani.

Il passo successivo è stato quindi quello di trovare un modo per tentare di normalizzare le misure effettuate, questo ci ha portato a identificare, dopo una serie di prove, una distanza caratteristica da utilizzare come misura per uniformare le altre. Accordando sulla misura della distanza dalla punta del dito medio al landmark del polso, siamo riusciti a migliorare nettamente il riconoscimento di una determinata gesture effettuate da mani differenti.

Questa distanza è calcolata sommando le distanze tra i quattro landmark del dito medio (le tre falangi e la base) e la distanza tra la base e il landmark polso.

Abbiamo poi accoppiato questa tecnica con la mappatura delle misure a un range di livelli discreto, in modo da ridurre l'impatto dell'instabilità dei valori forniti dal framework, a scapito di un minimo livello di accuratezza.

3.3 Lo spazio tridimensionale

Le tecniche implementato fino ad ora non tengono veramente conto del fatto che le gesture hanno una forte caratteristica tridimensionale, in quanto riducono lo stato della mano a un insieme di parametri monodimensionali, abbiamo infatti notato che molte posizioni che vogliamo identificare presentano notevoli similitudini nello spazio delle distanze tra landmark.

Un parametro che abbiamo scelto di adottare nel sistema di riconoscimento è quello dell'allineamento nello spazio tridimensionale di un certo dito, o più in generale di un insieme di landmark.

Per ottenere questa informazione, abbiamo implementato una serie di funzioni di utilità che permettono di identificare il discostamento che una serie di punti ha dalla retta 3D formata da altri due punti dati; il calcolo di questa misura, unito a una lista di valori di errore ottenuti in modo sperimentale, permette di quantificare il livello di ripiegamento di un certo dito.

Grazie a questa tecnica, unita alle metodologie sopra descritte, siamo riusciti a ottenere una identificazione molto più accurata della posizione delle dita di una mano e a ridurre significativamente il grado di sovrapposizione tra gesture.

Anche in questo caso abbiamo implementato la normalizzazione a livelli delle misure effettuate, prendendo però in considerazione una distanza di riferimento variabile, ovvero la lunghezza di ogni falange del dito considerato.

3.4 Rilevamento velocità e direzione

L'utilità di un sistema di riconoscimento di gesture non si ferma soltanto alla rilevazione di posizioni statiche, è infatti desiderabile poter ottenere anche informazioni relative alla velocità e direzione di certi landmark significativi, utilizzando poi questi dati per abilitare l'utente a una gamma più vasta di interazioni.

Per una delle gesture implementate, si è reso necessario creare una apposita classe `TimedPoint`, che mette a disposizione funzionalità di calcolo della velocità media e della direzione tra due `TimedPoint`, questi dati vengono poi utilizzati per abilitare l'identificazione di gesture in seguito a un movimento definito a una certa velocità media.

Questi calcoli ci hanno permesso di definire gesture **dinamiche**, che devono essere rilevate **in seguito a un movimento** effettuato a una certa velocità e in una certa direzione, e non in seguito all'avverarsi di una posizione statica.

E' importante specificare che anche le gesture cosiddette *statiche*, in certi casi vengono utilizzate per ottenere parametri che vengono poi applicati in particolari modi all'applicazione, il fatto che questi parametri sono dinamici non le categorizza come gesture dinamiche.

4 — ...

5 — Osservazioni finali

5.1 Analisi delle performance

5.1.1 Limiti dell'applicazione

Limiti imposti da MediaPipe

1. **Framerate limitato**: comprensibile contando che siamo su dispositivi mobile, meno elaborazioni facciamo meno batteria viene consumata.

2. **Dati imprecisi quando non tutte le dita vengono mostrate**: se si mostra una gesture che non espone tutte le dita bene in vista (pollice in su) e si muove la mano cambiando punto di vista, le coordinate rilevate dalla libreria non rimangono coerenti.

3. **Applicazione basata su immagini ben illuminate**: se nello stream video manca di luminosità l'accuratezza del rilevamento cala drasticamente (problema noto nel mondo di computer vision)

Limiti della nostra soluzione

1. **Consumo batteria relativamente elevato**: per una app mobile porre attenzione ai consumi è fondamentale, la nostra applicazione consuma un 2% ogni 5 minuti circa su una batteria 4000mAh. Questo consumo è comprensibile considerando il lavoro che svolge; un fattore critico è rappresentato dall'impossibilità di avere momenti di stand-by in quanto il tracking delle mani lavora anche quando non vi sono mani da riconoscere, portando i momenti morti (dove non accade nulla ma l'applicazione è attiva) ad una fonte di spreco risorse.

2. **Lentezza di gesture**: essendo il framerate limitato bisogna effettuare le gesture con calma per stare dietro alle elaborazioni.

3. **Gesture non eccessivamente elastiche**: l'applicazione utilizza le proporzioni usando come riferimento la lunghezza fisica del dito medio fino ad arrivare

al centro del polso; tutte le altre misure rilevate sono comparate con quella. Dal punto di vista teorico dovrebbe funzionare su tutte le mani, ma, dovendo discriminare le gesture in modo abbastanza rigido per evitare che gesture differenti si confondano tra di loro, per effettuare correttamente i segni in alcuni casi le dita vanno orientate in un modo specifico (`crab` gesture = il pollice deve essere particolarmente inclinato (vincolo posto per non confondere tale gesture con il `pinch`)).

5.1.2 Profiling con Android Studio

I dati presentati in seguito sono stati raccolti dal profiler di Android Studio su un dispositivo con le seguenti specifiche:

- **GPU**: ARM MALI-G72 MP3 (850MHz)
- **CPU**: Octa-core, 2 processori: 4x 2.3GHz ARM Cortex-A73 (Quad-core), 4x 1.7GHz ARM Cortex-A53 (Quad-core)
- **RAM**: 6GB

5.2 Opinioni e sviluppi futuri

La possibilità di poter interagire con un dispositivo tramite una videocamera introduce un metodo alternativo di interfacciamento con la macchina. Nel corso degli anni i computer si sono evoluti sempre di più mentre il metodo per interagire con essi è rimasto sempre lo stesso: la tastiera.
Solo nell'ultimo decennio, grazie ad una sufficiente maturazione della tecnologia, si è cominciato concretamente a pensare a mezzi differenti per comunicare con i dispositivi. D'altronde siamo di fronte ad una situazione dove due macchine computazionali potenti (cervello e computer) nella loro collaborazione sono ostacolate dall'interfacciamento Input/Output che gli si pone in mezzo ed è per questo che soluzioni che rendono più immediato e naturale il trasferimento dei dati saranno probabilmente l'argomento di discussione principale degli anni a seguire.

Il nostro team si è occupato di analizzare un "proof-of-concept" di una applicazione PdfReader gestita non dall'interfacciamento proposto dello schermo touch ma dalla videocamera frontale che rileva specifiche gesture. Svolgendo i test finali siamo rimasti soddisfatti del risultato; l'unico tallone d'Achille è il fatto di avere un riconoscimento gesti in alcuni casi abbastanza rigido, dovuto proprio ai dati non molto precisi che Mediapipe pone quando non tutte le dita vengono mostrate.

Un'idea di risoluzione sarebbe porre due videocamere che riprendono due angolazioni diverse e poi elaborate insieme per ricostruire la posizione delle dita. Questa proposta però non avrebbe molti punti a favore considerando che nei dispositivi già sul mercato non sarebbe compatibile oltre al fatto di svolgere il doppio delle elaborazioni, ma per prodotti non mobile ancora da inserire sul mercato come una lavagna LIM che adotta questa tecnica non sarebbe una idea da scartare a priori.

In questo progetto ci si è limitati a gestire un pdf, ma una volta che la logica di gesture detection è stata messa a punto e anche il consumo migliorato ecco che cambiare scenario diviene immediato: è sufficiente collegare le gesture a determinate azioni. Un esempio potrebbe essere la simulazione del cursore tramite gesture, permettendo quindi di navigare nell'interfaccia proposta dallo schermo. Combinando poi il riconoscimento delle mani con altri tipi di riconoscimenti (face, body, ...) potrebbe nascere qualcosa di interessante. Rimanendo però nel mondo mobile, potendo consumare poca batteria e di conseguenza non avendo una grande potenza computazionale a disposizione, il framerate limitato non permette di riconoscere accuratamente gesture dinamiche veloci, ciò nonostante si riescono a realizzare applicazioni con interazioni più naturali rispetto ai classici pulsanti. Sarebbe valido proporre una soluzione dove l'applicazione non elabora direttamente i dati, ma delega un server remoto con una potenza di calcolo molto maggiore.