

# “GESTURE REC”

## Riconoscimento del linguaggio dei segni

A.A. 2021/2022



*Luca Berardi*

*Marco Motamed*

*Giuseppe Sergi*

## Sommario

|  |           |
|--|-----------|
| <b>Abstract .....</b>                                    | <b>3</b>  |
| <b>1. Lingua dei segni .....</b>                         | <b>4</b>  |
| <b>2. Dataset .....</b>                                  | <b>5</b>  |
| <b>3. Architettura e Addestramento del modello .....</b> | <b>7</b>  |
| <b>3.1 Strumenti e tecniche utilizzati.....</b>          | <b>7</b>  |
| <b>3.2 Il modello.....</b>                               | <b>8</b>  |
| <b>3.3 Addestramento .....</b>                           | <b>10</b> |
| <b>4. Implementazione su Android .....</b>               | <b>12</b> |
| <b>4.1 Conversione del modello in Tflite .....</b>       | <b>12</b> |
| <b>4.2 Funzionalità dell'applicazione .....</b>          | <b>12</b> |
| <b>4.3 Implementazione .....</b>                         | <b>18</b> |
| <b>4.4 Prestazioni .....</b>                             | <b>22</b> |
| <b>5 Conclusioni .....</b>                               | <b>25</b> |
| <b>6 Riferimenti.....</b>                                | <b>26</b> |

## Abstract

L'applicazione Gesture Rec è un'applicazione per dispositivi Android pensata per il supporto alle persone affette da mutismo e/o sordità. Essa permette, infatti, di riconoscere i vari gesti del linguaggio dei segni utilizzando la fotocamera del dispositivo e di trascrivere ciò che viene detto.

L'applicazione è composta principalmente da tre funzionalità:

1. *Modalità foto*: l'utente, attraverso la fotocamera, inquadra il gesto e scatta una foto che verrà elaborata e verrà mostrato a schermo il significato del gesto.
2. *Modalità video*: l'utente, attraverso la fotocamera, registra un video in cui vengono eseguiti vari gesti. Quest'ultimo verrà elaborato e a schermo sarà mostrato il significato attribuito alla parola o alla frase precedentemente registrata.
3. *Modalità audio*: l'utente registra un audio e quanto detto viene riportato a schermo (funzionalità pensata per permettere ad un utente di parlare in maniera rapida ad un soggetto affetto da sordità).

# 1. Lingua dei segni

Le lingue dei segni sono lingue che veicolano i propri significati attraverso un sistema codificato di segni delle mani. L'applicazione Gesture Rec si basa sulla ASL (American Sign Language), sviluppatasi dalla seconda metà del secolo XIX.

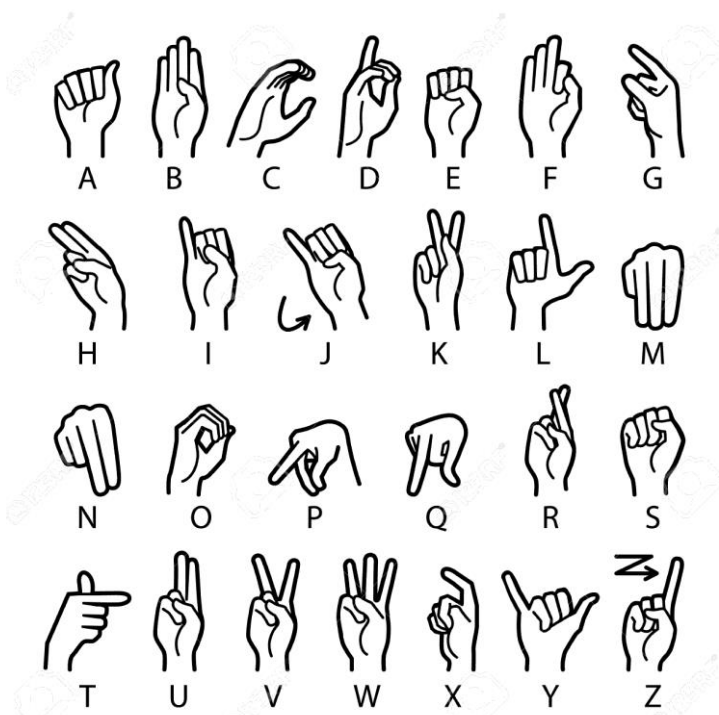


Figura 1: lingua dei segni.

Fonte: <https://www.pinterest.it>

La comunicazione avviene producendo dei precisi segni compiuti con le mani, segni che hanno un significato specifico e codificato come avviene per le lettere.

## 2. Dataset

La rete neurale descritta nei capitoli successivi viene addestrata su un dataset composto da 29 classi ognuna indicante un segno differente. Ogni classe è composta da circa 7760 immagini per un totale di 225040 immagini. Di seguito sono riportati i nomi delle classi:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, del, nothing, space.



Figura 2: esempio di immagini presenti in ciascuna classe.

Di queste 29 classi, 26 rappresentano le varie lettere dell'alfabeto e le restanti 3 rappresentano rispettivamente:

- del: la classe che rappresenta il gesto con cui eliminare la lettera precedente.
- nothing: la classe che non comprende alcun gesto della lingua dei segni.
- space: la classe che rappresenta il gesto con cui separare una parola dall'altra.

Le immagini presenti all'interno del dataset sono di varie dimensioni e in formato ".jpg", esse rappresentano i gesti realizzati sia con la mano destra che con la mano sinistra.

Nella fase iniziale era stato utilizzato un dataset composto da 3000 immagini per ognuna delle 29 classi, per un totale di 87000 immagini. Successivamente è stato deciso di unirlo con altri dataset al fine di ottenere sia un numero più consistente di immagini, sia immagini diverse tra loro (tipi di mani, sfondi, luminosità, distanza, qualità dell'immagine) in modo da rendere più efficiente il modello.

I dataset sopracitati sono stati scaricati dal sito Kaggle [2][3]. Inizialmente, all'interno di uno dei dataset utilizzati, per ciascuna classe erano presenti circa 750 immagini realizzate ruotando alcune delle immagini già presenti nella classe. Queste sono state rimosse poiché potevano provocare confusione nel riconoscimento del gesto visto che alcuni di essi sono molto simili e differiscono unicamente per la posizione della mano.

### **3. Architettura e Addestramento del modello**

#### **3.1 Strumenti e tecniche utilizzati**

Per lo sviluppo della rete è stato utilizzato TensorFlow 2.7.0 con Keras. Inizialmente si è optato per realizzare un modello sequenziale con vari livelli ma, data la scarsa accuratezza delle previsioni, è stato deciso di utilizzare dapprima ResNet50 (l'aumento dell'accuratezza era già notevole) ed infine MobileNet, vista anche la sua predisposizione per i dispositivi mobili data la leggerezza e l'uso limitato di risorse.

Per l'addestramento del modello è stata utilizzato sia Google Colab, sia una scheda grafica NVIDIA GeForce RTX 2060 con supporto a CUDA 11.2 e cuDNN 8.1.1.

Per quanto riguarda le scelte implementative inizialmente è stata utilizzata la tecnica di data augmentation, che si basa sulla generazione di ulteriori dati ricavati da leggere modifiche su quelli già esistenti, al fine di aumentare la mole di file utili per l'addestramento settando uno zoom ed una rotazione casuale.

Successivamente, questa scelta è stata accantonata poiché la rete neurale produceva previsioni errate scambiando le lettere tra loro.

Al fine di uniformare gli elementi del dataset utilizzati per l'addestramento, tutte le immagini sono state ridimensionate con una dimensione pari a 200x200.

Per l'addestramento della rete il dataset è stato così suddiviso:

- 80% training set
- 20% validation set

## 3.2 Il modello

Il modello utilizzato è MobileNet:

```
mobile=tf.keras.applications.MobileNet(  
    input_shape=input_shape,  
    alpha=1.0,  
    include_top=True,  
    weights=None,  
    classes=num_classes,  
    classifier_activation="softmax"  
)
```

Figura 3: estratto del codice, modello MobileNet.

Come mostrato in figura 3 è presente l'attributo *classifier\_activation="softmax"* che consente di introdurre la funzione Softmax utile per convertire un vettore di valori in una distribuzione di probabilità, ottenendo così per ciascuna classe un valore compreso tra 0 e 1.

Infine, il modello finale, visibile in figura 4, è un modello sequenziale in cui è presente un livello di preelaborazione che ridimensiona i vari input in un nuovo intervallo e successivamente il modello MobileNet sopra mostrato.

```
# Keras will add a input for the model behind the scene.  
model = models.Sequential([  
    layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),  
    mobile  
)
```

Figura 4: estratto del codice, modello finale.

L'immagine seguente (figura 5) mostra il livello di preelaborazione (input layer) con shape (200,200,3), questa shape viene indicata nel successivo livello di Rescaling (figura 4) del modello sequenziale, seguito infine da mobileNet:



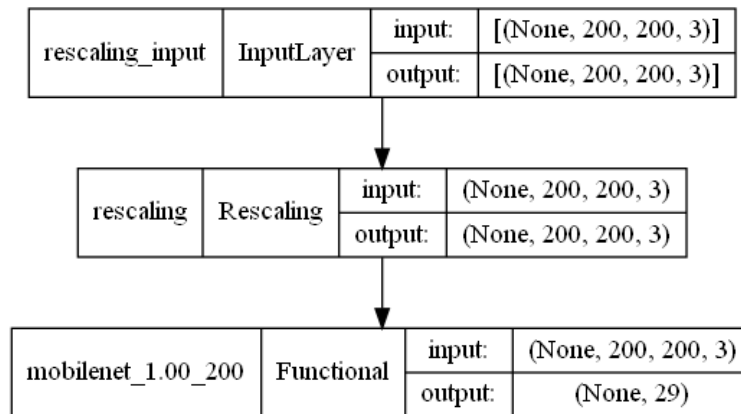


Figura 5: rappresentazione del modello finale.

Inoltre, nella fase di preelaborazione, impostando il valore di “shuffle” a “True” i dati usati per training e validation vengono mescolati prima di ogni epoca in modo da aumentare l’efficienza della rete prodotta.

Per ottimizzare il processo di addestramento è stato deciso di salvare il modello ad ogni miglioramento monitorando la perdita totale del modello (*checkpoint*). È stato deciso di interrompere l’addestramento qualora, al passare di 5 epoche (*patience=5*), non vi siano ulteriori miglioramenti (*earlyStop*). Inoltre, se non si verificano miglioramenti sul “val\_loss” dopo 3 epoche il tasso di apprendimento viene ridotto di un fattore 0.2 evitandone così il ristagno (*reduce\_lr*). Quanto detto è mostrato nelle callbacks rappresentate in figura 5.

```
checkpoint = ModelCheckpoint("NeuralNetworkUltima.h5",
                             monitor="val_loss",
                             mode="min",
                             save_best_only=True,
                             verbose=1)

earlystop = EarlyStopping(monitor='val_loss',
                           restore_best_weights=True,
                           patience=5,
                           verbose=1)

reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                                                  patience=3, min_lr=0.001)

callbacks = [reduce_lr,earlystop,checkpoint]
```

Figura 5: callbacks.

Inoltre, utilizziamo la funzione `keras.losses.SparseCategoricalCrossentropy` in quanto nel modello sono presenti ventinove classi di etichette. Inserendo l'attributo `from_logits=False` il risultato della rete codificherà una distribuzione di probabilità.

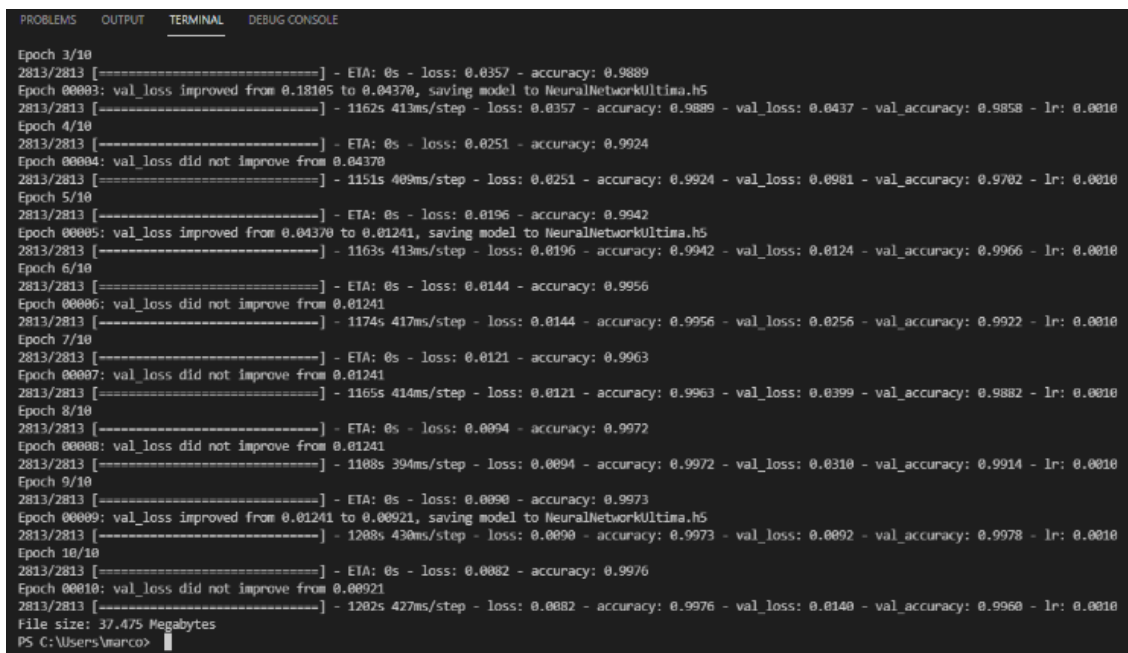
```
model.compile(optimizer = 'adam',  
              loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),  
              metrics = ["accuracy"])
```

Figura 6: model.compile.

### 3.3 Addestramento

La rete è stata addestrata con una scheda grafica NVIDIA GeForce RTX 2060. L'addestramento è stato svolto per 10 epoche, con una durata complessiva di 4 ore. I risultati, nella fase iniziale dell'addestramento, tendono a migliorare velocemente per poi assestarsi verso la fase finale.

Anche se il numero di epoche è abbastanza contenuto, i risultati sono soddisfacenti, raggiungendo un'accuratezza finale pari al 99%.



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE  
Epoch 3/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0357 - accuracy: 0.9889  
Epoch 00003: val_loss improved from 0.18105 to 0.04370, saving model to NeuralNetworkUltima.h5  
2813/2813 [=====] - 1162s 413ms/step - loss: 0.0357 - accuracy: 0.9889 - val_loss: 0.0437 - val_accuracy: 0.9858 - lr: 0.0010  
Epoch 4/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0251 - accuracy: 0.9924  
Epoch 00004: val_loss did not improve from 0.04370  
2813/2813 [=====] - 1151s 409ms/step - loss: 0.0251 - accuracy: 0.9924 - val_loss: 0.0981 - val_accuracy: 0.9702 - lr: 0.0010  
Epoch 5/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0196 - accuracy: 0.9942  
Epoch 00005: val_loss improved from 0.04370 to 0.01241, saving model to NeuralNetworkUltima.h5  
2813/2813 [=====] - 1163s 413ms/step - loss: 0.0196 - accuracy: 0.9942 - val_loss: 0.0124 - val_accuracy: 0.9966 - lr: 0.0010  
Epoch 6/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0144 - accuracy: 0.9956  
Epoch 00006: val_loss did not improve from 0.01241  
2813/2813 [=====] - 1174s 417ms/step - loss: 0.0144 - accuracy: 0.9956 - val_loss: 0.0256 - val_accuracy: 0.9922 - lr: 0.0010  
Epoch 7/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0121 - accuracy: 0.9963  
Epoch 00007: val_loss did not improve from 0.01241  
2813/2813 [=====] - 1165s 414ms/step - loss: 0.0121 - accuracy: 0.9963 - val_loss: 0.0399 - val_accuracy: 0.9882 - lr: 0.0010  
Epoch 8/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0094 - accuracy: 0.9972  
Epoch 00008: val_loss did not improve from 0.01241  
2813/2813 [=====] - 1108s 394ms/step - loss: 0.0094 - accuracy: 0.9972 - val_loss: 0.0310 - val_accuracy: 0.9914 - lr: 0.0010  
Epoch 9/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0090 - accuracy: 0.9973  
Epoch 00009: val_loss improved from 0.01241 to 0.00921, saving model to NeuralNetworkUltima.h5  
2813/2813 [=====] - 1208s 430ms/step - loss: 0.0090 - accuracy: 0.9973 - val_loss: 0.0092 - val_accuracy: 0.9978 - lr: 0.0010  
Epoch 10/10  
2813/2813 [=====] - ETA: 0s - loss: 0.0082 - accuracy: 0.9976  
Epoch 00010: val_loss did not improve from 0.00921  
2813/2813 [=====] - 1202s 427ms/step - loss: 0.0082 - accuracy: 0.9976 - val_loss: 0.0140 - val_accuracy: 0.9960 - lr: 0.0010  
File size: 37.475 Megabytes  
PS C:\Users\marco>
```

Figura 7: epoche.

Di seguito sono riportati due grafici riguardanti l'andamento dei valori di accuracy (figura 8) e di loss (figura 9) in riferimento alle epoche:

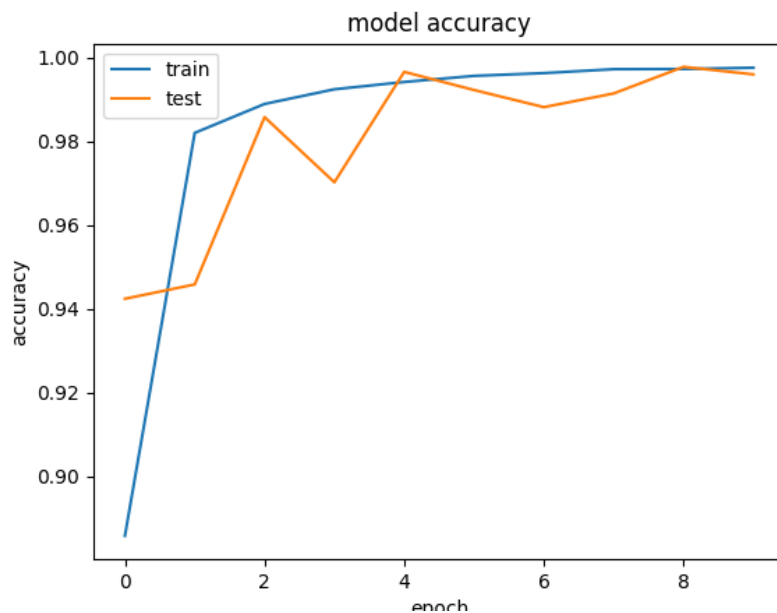


Figura 8: grafico accuracy.

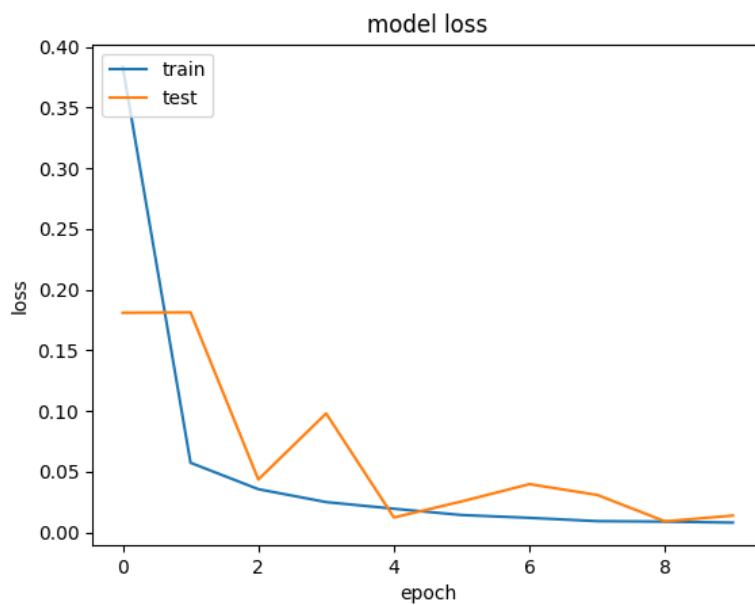


Figura 9: grafico loss.

## 4. Implementazione su Android

### 4.1 Conversione del modello in Tflite

Il primo passo è stato quello di convertire il modello (formato .h5) in un modello TensorflowLite (formato .tflite) al fine di garantire una dimensione ridotta e un'inferenza più rapida, grazie alla possibilità di accedere ai dati senza passaggi di parsing.

Difatti la dimensione del modello generato è di 12 MB a scapito dei 38 MB iniziali.

### 4.2 Funzionalità dell'applicazione

L'applicazione Gesture Rec è articolata in tre principali funzionalità:

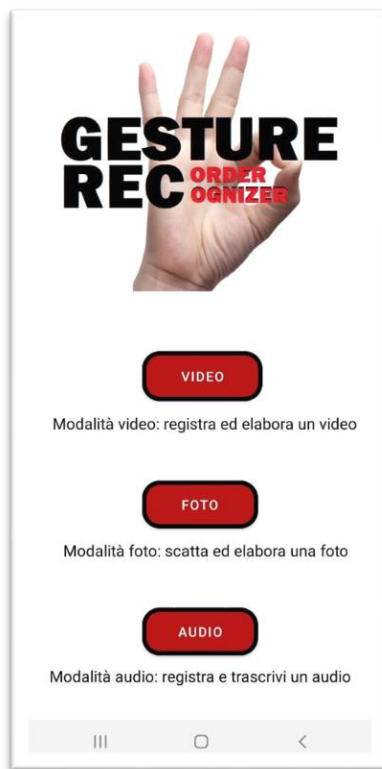


Figura 10: schermata principale dell'applicazione.

## Foto

Premendo il bottone “foto” verrà visualizzata una nuova schermata in cui sarà possibile scattare una foto come mostrato in figura 11. In questa schermata sarà mostrato in tempo reale ciò che viene inquadrato dalla fotocamera ed un bottone per scattare la foto. Una volta scattata la foto, comparirà il bottone “elabora” (figura 12) che una volta premuto reindirizzerà l’utente su una nuova schermata in cui saranno visualizzati i tre migliori risultati della previsione con le relative percentuali e la foto appena scattata, come visibile in figura 13. Inoltre, in questa schermata sono presenti due bottoni che consentono di tornare alla schermata principale o di scattare una nuova foto.



Figura 11: schermata della funzionalità “foto”.

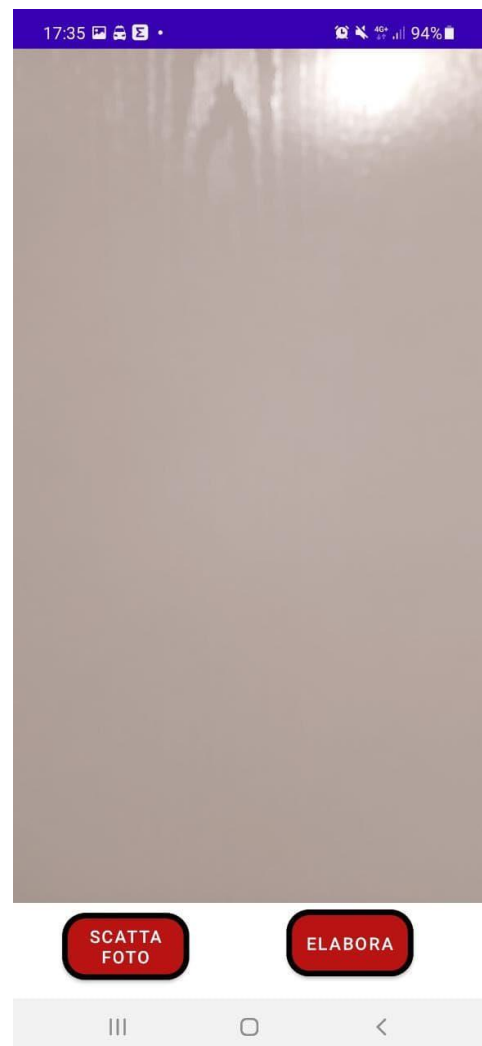


Figura 12: schermata con bottone “elabora”.

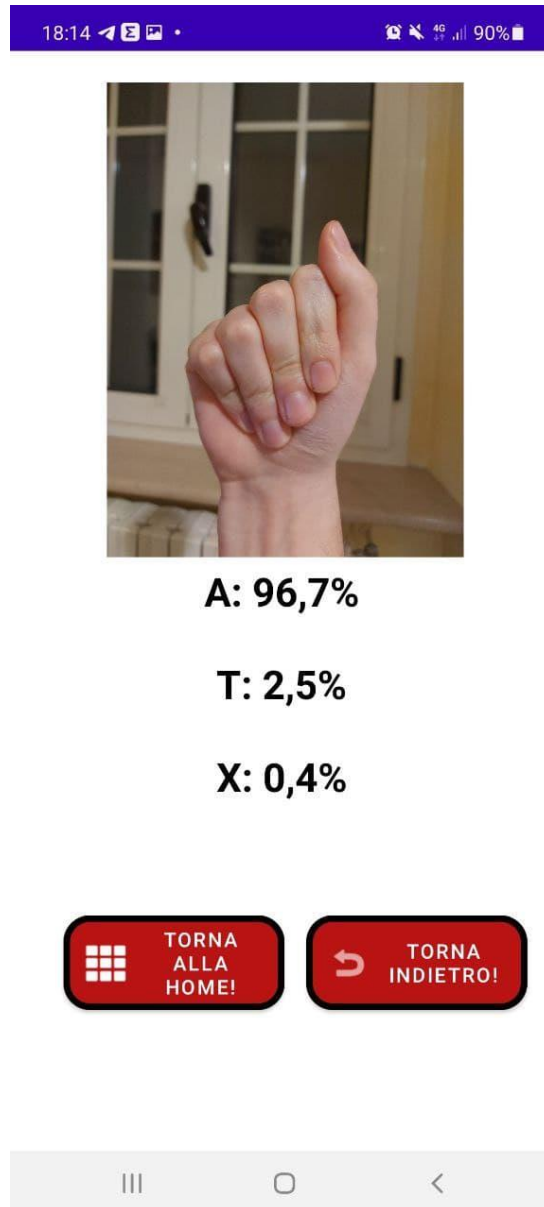


Figura 13: schermata in cui sono visualizzati i risultati.

## Video

Premendo il bottone “video” verrà visualizzata una nuova schermata in cui sarà possibile registrare un video (figura 14). In questa schermata sarà mostrato in tempo reale ciò che viene inquadrato dalla fotocamera e tre bottoni.

Premendo il bottone “play” (bottone centrale) sarà avviata la registrazione del video e contemporaneamente i bottoni “mostra” ed “elabora”, rispettivamente alla sinistra ed alla destra del tasto centrale, non verranno visualizzati (figura 15).

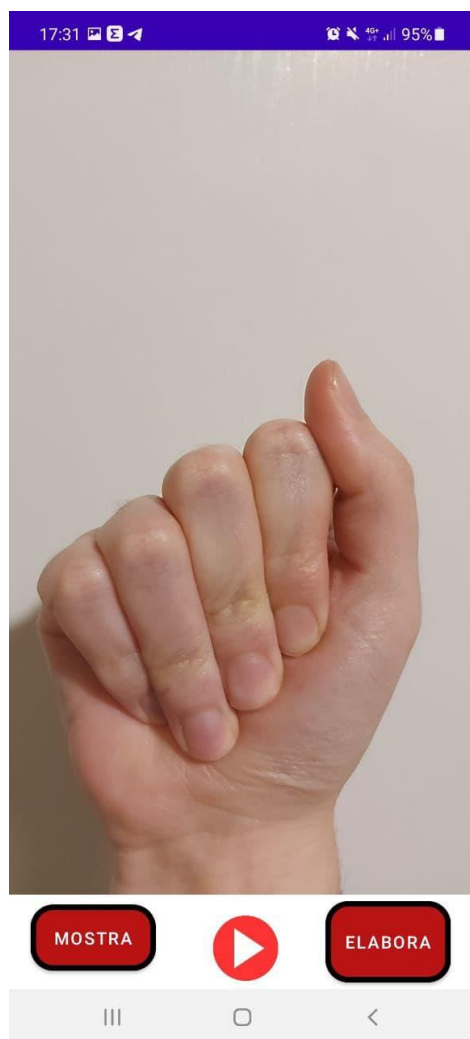


Figura 14: schermata di registrazione del video



Figura 15: registrazione avviata.

Cliccando sul bottone stop il video registrato verrà salvato e compariranno i bottoni “mostra” ed “elabora”. Premendo il bottone “mostra” sarà possibile visualizzare la registrazione come mostrato in figura 16. Premendo il bottone “elabora”, invece, il video verrà elaborato e sarà mostrato a schermo il suo

significato in una schermata da cui, attraverso due bottoni, sarà anche possibile tornare alla schermata principale o registrare un nuovo video (figura 17).

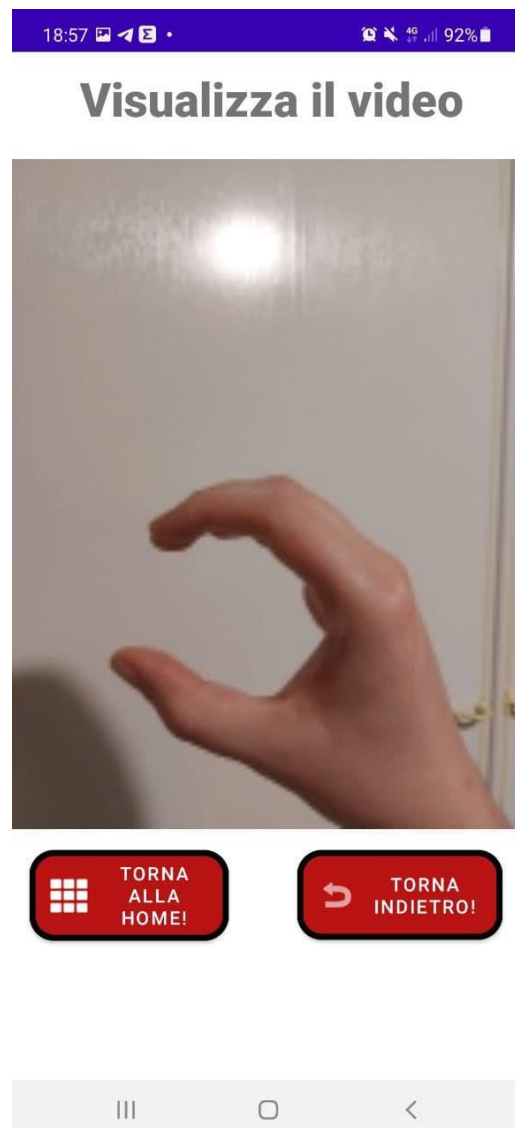


Figura 16: schermata in cui viene mostrato il video.

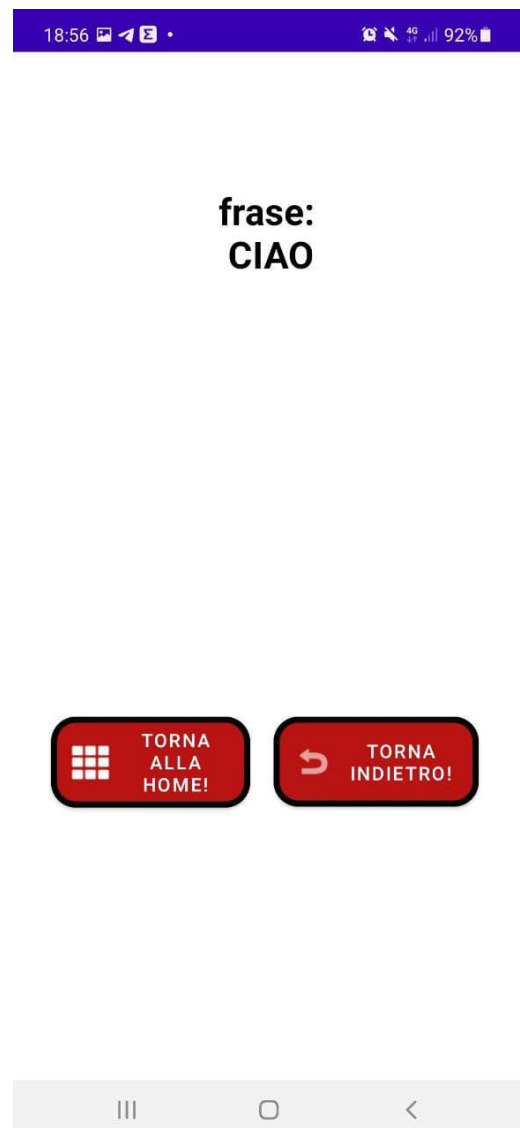


Figura 17: schermata in cui viene mostrato il risultato dell'elaborazione.

## Audio

Premendo il bottone “audio” si aprirà una schermata in cui, tramite il bottone elabora (figura 18), sarà possibile registrare un audio vocale (figura 19) che sarà successivamente trascritto come mostrato in figura 20.





Figura 18: schermata che permette la registrazione audio.

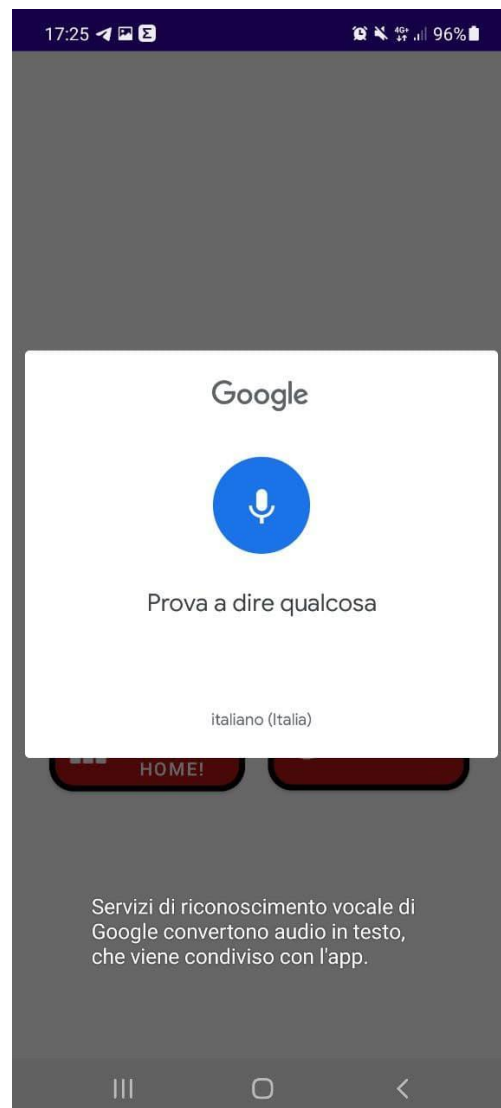


Figura 19: registrazione audio.



## TRASCRIZIONE:

**Ciao, come va?**



Figura 20: risultato.

### 4.3 Implementazione

In questo paragrafo viene descritto come l'applicazione è stata implementata, per la sua realizzazione sono stati utilizzati: Android Studio come ambiente di sviluppo integrato, Java come linguaggio e alcune risorse in formato xml.

Nell'immagine seguente sono riportate tutte le activities presenti nell'applicazione, ognuna implementa una schermata:

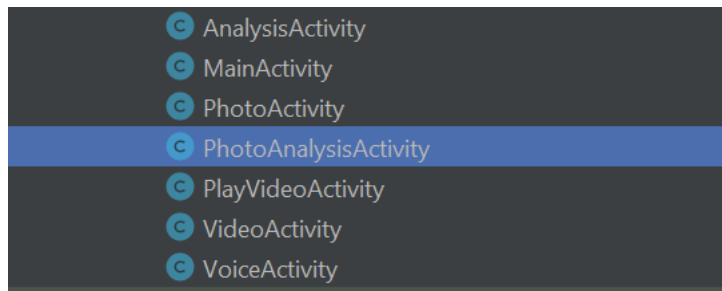


Figura 21: activities.

Successivamente saranno mostrati i punti salienti delle varie classi al fine di rappresentare le tecniche di implementazione utilizzate.

Per quanto riguarda *photoActivity* la foto viene scattata utilizzando *ImageCapture* di Androidx.

L'activity *PhotoAnalysisActivity* si occupa dell'elaborazione dell'immagine. L'immagine viene elaborata come Bitmap e dato che su dispositivi diversi questo risultava ruotato diversamente sarà ruotato con la giusta angolazione, ottenuta tramite l'implementazione di un'apposita funzione. Una volta inizializzato il modello Tensorflow Lite e creata un'istanza della classe Interpreter, per eseguire l'inferenza del modello, vengono caricati i nomi delle classi da predire da un file *labels.txt*.

```
...  
int imageTensorIndex = 0;  
DataType imageDataType = tflite.getInputTensor(imageTensorIndex).dataType();  
TensorImage tfImage = new TensorImage(imageDataType);  
tfImage.load(bitmap);  
tflite.run(tfImage.getBuffer(), labelProbArray);  
...
```

Figura 22: estratto da photoAnalysisActivity.

I risultati dell'inferenza verranno salvati in un array che rappresenta l'output della funzione *run* visibile in figura 22. Una volta fatto questo, sarà possibile reperire le probabilità di ciascuna classe grazie alla seguente funzione:

```
protected float getProbability(int labelIndex) {  
    return labelProbArray[0][labelIndex];  
}
```

Figura 23: funzione *getProbability()*.

Al fine di non occupare memoria, l'immagine, dopo essere stata elaborata, viene cancellata.

L'activity *VideoActivity* permette di registrare un video utilizzando la funzione *startRecording()* della classe *VideoCapture* di Androidx come mostrato in figura 24.

```
videoCapture.startRecording(  
    new VideoCapture.OutputFileOptions.Builder(vidFile).build(),  
    getExecutor(),  
    new VideoCapture.OnVideoSavedCallback() {  
        @Override  
        public void onVideoSaved(@NonNull VideoCapture.OutputFileResults outputFileResults) {  
            Toast.makeText(context: VideoActivity.this, text: "Video has been saved successfully.",  
                Toast.LENGTH_SHORT).show();  
        }  
        @Override  
        public void onError(int videoCaptureError, @NonNull String message, @Nullable Throwable cause) {  
            Toast.makeText(context: VideoActivity.this, text: "Error saving video: " + message,  
                Toast.LENGTH_SHORT).show();  
        }  
    }  
);
```

Figura 24: *startRecording()*.

Per la registrazione del video, al fine di aumentare le prestazioni e la velocità di elaborazione, è stato deciso di fissare il frame rate ed abbassare la risoluzione a 200x200 già al momento della registrazione. Così facendo, il tempo di elaborazione del video risultava più che dimezzato. Ad esempio, per un video di 5 secondi, senza impostare la risoluzione sopra descritta, il tempo di elaborazione era di circa 6 secondi, successivamente il tempo è stato ridotto a circa 1.8 secondi.

In *AnalysisActivity* viene elaborato il video precedentemente registrato. Utilizzando *VideoCapture* di OpenCV è stato deciso di non analizzare ogni singolo frame del video ma di analizzare cinque frame per ogni secondo, in questo modo il carico di lavoro risulta ridotto e più sostenibile.

Una prima soluzione prevedeva di leggere ogni frame in arrivo tramite la funzione *read(frame)* di *VideoCapture* e, successivamente, elaborare ogni 5 frame (figura 25). Nella figura 26, è invece riportata la soluzione finale, con un’ottimizzazione notevole dei tempi.

```
if(videoCapture.isOpened()){
    ...
    while(videoCapture.read(frame)){
        if(i%6==0){
            ...
            frase += classifyFrame(bitmap);
            ...
        }
        i++;
    }
    ...
}
```

Figura 25: prima soluzione.

```
if(cap.isOpened()) {
    ...
    while (videoCapture.grab()) {
        if (i % 6 == 0) {
            videoCapture.read(frame);
            ...
            frase += classifyFrame(bitmap);
            ...
        }
        i++;
    }
    ...
}
```

Figura 26: soluzione finale.

Una volta analizzati tutti i frame interessati, per ovviare al problema dei frame di transizione tra un gesto ed un altro è stato deciso di assumere come gesto significativo (e quindi trascritto nella schermata in figura 17) soltanto quello che risultava identico anche nei successivi due frame. Ovviamente, per evitare ripetizioni dello stesso gesto, qualora venga considerato “valido”, tutti i successivi frame rappresentanti il gesto in questione verranno scartati. Nel caso delle “doppie”, ovvero una parola che presenta due lettere (e quindi due gesti) uguali in successione, queste saranno distinte dalla presenza di uno o più frame di transizione; questo permetterà alle due lettere di essere trascritte correttamente. Di seguito è riportato un esempio per comprenderne meglio il funzionamento:

È stato registrato un video in cui veniva registrata la parola “CASSA”. L’output prodotto dall’elaborazione di tutti i frame risulta essere: “CCCCHZAAAAAAAAAAJASSSSSSSFESSSSSSSHTAAAAAAAAAA”, i gesti significativi (le lettere sottolineate) risultano comparire consecutivamente per un numero maggiore o uguale a tre frame. Grazie all’algoritmo sopra descritto, infatti, viene mostrata a schermo la stringa corretta: “CASSA”.

Anche qui, come nel caso dell'activity precedente, l'elaborazione dell'immagine viene effettuata utilizzando il modello Tensorflow Lite importato creando un'istanza della classe Interpreter, per eseguirne l'inferenza.

Inizialmente era stato deciso di analizzare soltanto un frame ogni secondo, questo però risultava inefficiente in quanto poteva accadere che il frame analizzato non fosse quello desiderato ma uno derivato dalla transizione tra un gesto e l'altro. Inoltre, non era possibile gestire il caso delle doppie.

Un'ulteriore scelta implementativa è stata quella di non utilizzare *MediaMetadataRetriever* in quanto l'elaborazione risultava più lenta e meno efficiente rispetto a *VideoCapture* di OpenCV.

Infine, come nel caso della foto, è stato scelto di eliminare il video dopo l'elaborazione per non occupare memoria inutilmente.

Nell'activity *VoiceActivity* è implementata la trascrizione vocale. È stata utilizzata la libreria di Android *Speech* che, tramite la creazione di un nuovo Intent implicito in cui viene specificata l'operazione da eseguire (riconoscimento vocale) come visibile in figura 27.

```
speak = (Button) findViewById(R.id.speak);
speak.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent( action: "android.speech.action.RECOGNIZE_SPEECH");
        startActivityForResult(intent, requestCode: 0);
    }
});
```

Figura 27: Intent implicito.

## 4.4 Prestazioni

In questo paragrafo sono riportate alcune analisi delle performance, per effettuarle è stato utilizzato il tool Android Profiler disponibile in Android Studio,

grazie ad esso è stato possibile analizzare il consumo di CPU e di memoria durante l'esecuzione dell'app.

Di seguito è riportata la prima sessione inerente all'avvio dell'applicazione. Si può notare come in una prima fase entrambi riportino valori maggiori rispetto a quelli assunti una volta che l'applicazione è stata avviata:

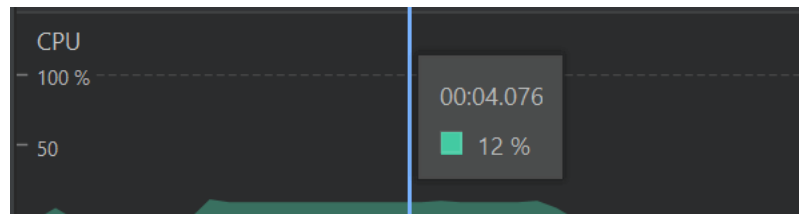


Figura 28: consumo CPU all'avvio dell'applicazione.

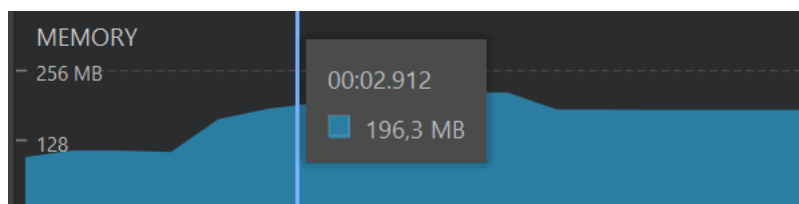


Figura 29: consumo di memoria all'avvio dell'applicazione.

Nell'immagine seguente sono riportati i consumi durante il processo di scatto ed elaborazione della foto, si può notare come il consumo di CPU cresca al passaggio tra una activity e l'altra ed al click dei bottoni, mentre la memoria rimanga costante ad eccezione della *PhotoAnalysisActivity* in cui avviene l'elaborazione dell'immagine:

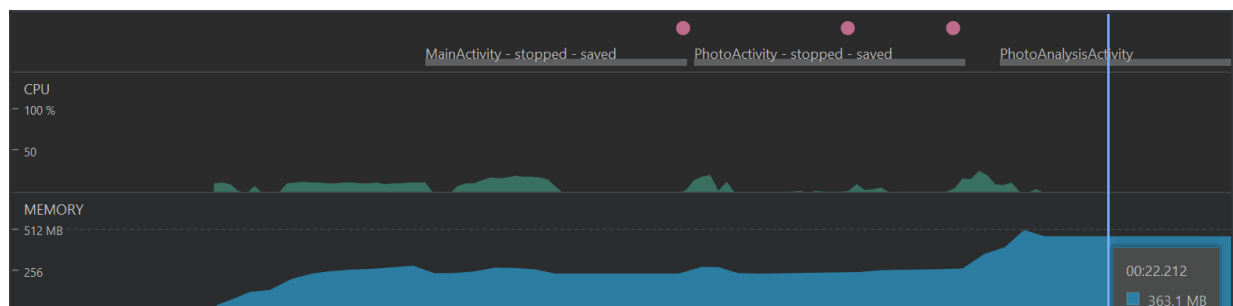


Figura 30: consumi durante l'uso della funzionalità Foto.

In figura 31 vengono riportati i consumi derivanti dall'uso della funzionalità Video (registrazione ed elaborazione di un video di cinque secondi):

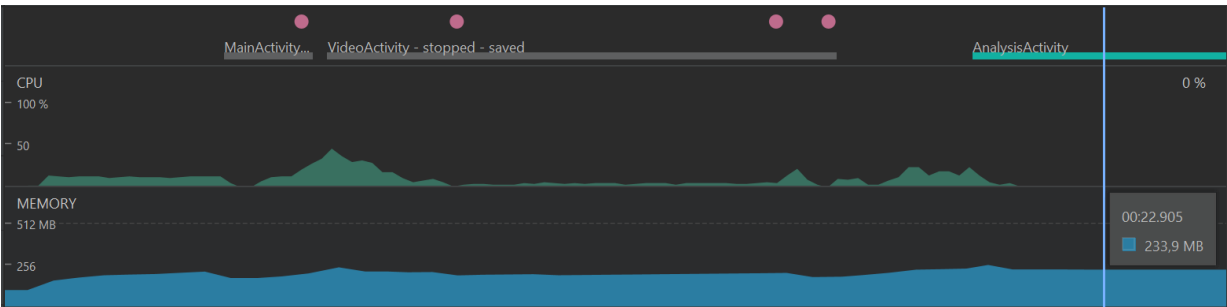


Figura 31: consumi durante l'uso della funzionalità Video.

Nell'ultima figura, invece, sono riportati i consumi inerenti all'uso della funzionalità Audio:

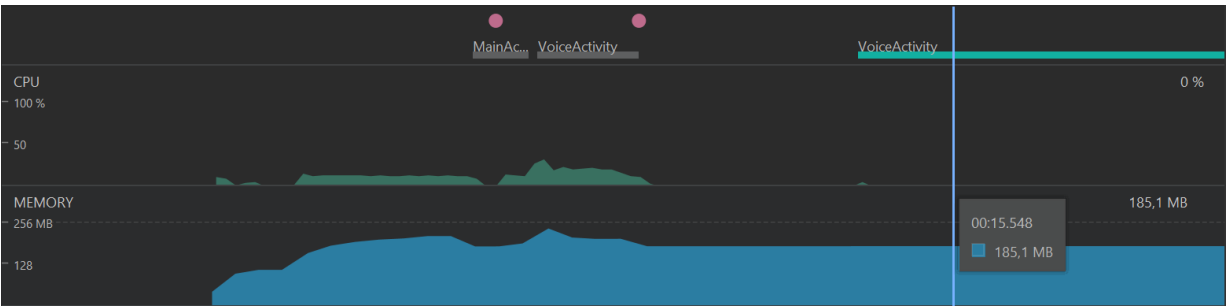


Figura 32: consumi durante l'uso della funzionalità Audio.



## 5 Conclusioni

Il progetto è stato realizzato con successo, con un'ottima velocità di esecuzione e fluidità dell'applicazione sullo smartphone (testata su: Samsung Galaxy A71, Samsung Galaxy S10 lite, Honor 10 lite) e con una dimensione dell'apk abbastanza contenuta (meno di 70 MB), che in futuro potrebbe essere ottimizzata ulteriormente. Anche il livello di accuratezza della rete risulta abbastanza soddisfacente. Tuttavia, in alcune circostanze, soprattutto dovute al rumore di fondo e alla forte somiglianza tra alcune lettere, la precisione delle previsioni risulta meno ottimale.

Un'estensione futura possibile, al fine di ovviare a questo problema, potrebbe concernere nell'aumento del numero di immagini fornite per ciascuna classe del dataset oppure nell'addestramento del modello con un numero più elevato di epoche.

## 6 Riferimenti

- [1] <https://www.tensorflow.org/>
- [2] <https://www.kaggle.com/debashishsau/aslamerican-sign-language-aplhabet-dataset>
- [3] <https://www.kaggle.com/kapillondhe/american-sign-language>
- [4] <https://stackoverflow.com/>
- [5] <https://github.com/>
- [6] <https://www.pinterest.it/pin/908953137264735331/>