ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

WLASL CLASSIFICATION & SIGNER CLASSIFICATION

Elaborato in
VISIONE ARTIFICIALE E RICONOSCIMENTO

Presentata da
SIMONE CEREDI
GIULIA NARDICCHIA

Anno Accademico 2023 – 2024

Indice

Introduzione **v**

1 Pre-elaborazione dei dati **1**
1.1 Analisi dei dati 1
1.2 Elaborazione dei dati 3
1.2.1 Modifica del parametro frame_end 3
1.2.2 Region Of Interest (ROI) 4
1.2.3 Ridimensionamento e padding dei video 4
1.2.4 Rimozione del background 5
1.2.5 Risultati ottenuti 6

2 Hand-crafted features **7**
2.1 Feature di forma 7
2.1.1 Edge extractor 7
2.1.2 HoG extractor 9
2.2 Feature di texture 11
2.2.1 Haar-like features 11
2.2.2 Local Binary Pattern 14
2.3 Feature colore 16
2.3.1 Color histogram 16
2.4 Feature di movimento 18
2.4.1 Optical flow 18
2.5 Estrazione keypoint con MediaPipe 20

3 Classificazione video **22**
3.1 Dynamic Time Warping (DTW) 22
3.2 Classificazione tradizionale 23
3.3 Classificazione con MediaPipe 24
3.4 Classificazione con rete neurale e MediaPipe 26
3.5 Risultati a confronto 30
3.5.1 Metodo tradizionale VS soluzione con MediaPipe 30
3.5.2 Soluzione con MediaPipe VS soluzione proposta su Kaggle 31

4 Classificazione Signer **33**
4.1 Analisi del dataset e pre-elaborazione dei dati 33
4.2 Data augmentation 34
4.3 Classificazione tradizionale 35
4.4 Classificazione con rete neurale 37
4.5 Risultati a confronto 40

Conclusioni **42**

Introduzione

Il progetto è stato realizzato prendendo come dataset di riferimento: Word-Level American Sign Language, reperibile su Kaggle al seguente link. Si tratta del più esteso dataset video dedicato al riconoscimento della Lingua dei Segni Americana (ASL). Include circa 12.000 video che coprono le 2.000 parole più comuni in ASL.

Si è individuato un primo obiettivo di progetto, quale: la classificazione delle parole traducendo dal linguaggio dei segni all'inglese.

In seguito, dopo aver analizzato il dataset e avendo visto la complessità del problema, si è proceduto ad effettuare una seconda classificazione: il riconoscimento dei volti (*signer*).

Un ulteriore obiettivo di progetto è quello di mostrare le differenze di performance che si ottengono sfruttando i modelli tradizionali e le reti neurali per i problemi sopra descritti.

Si inizierà effettuando l'analisi e la pre-elaborazione dei dati. Si estrarranno le *features handcrafed* che verranno utilizzate per le successive classificazioni. Si proseguirà poi con la creazione dei modelli per la classificazione vera e propria, prima per la classificazione delle parole dei video e in seguito per il riconoscimento dei volti. Successivamente verranno raccolti i risultati e confrontati tra le varie soluzioni (tradizionale VS rete neurale). Infine verranno raccolte alcune considerazioni finali e possibili sviluppi futuri.

Capitolo 1

Pre-elaborazione dei dati

In questo capitolo viene descritto il processo che è stato seguito per l'analisi del dataset e la pre-elaborazione dei dati.

1.1 Analisi dei dati

Una prima analisi riguarda la struttura del dataset e i dati presenti all'interno. Oltre ai vari video in formato .mp4, nel dataset si trova un file .json, la cui struttura è descritta di seguito. Esaminando il file `WLASL_v0.3.json` si osserva che per ogni parola `"gloss"` vi è un elenco di `"instances"`, le istanze rappresentano ogni video del dataset e alcune informazioni a riguardo.
Le informazioni fornite per ogni video sono le seguenti:

- `bbox`: si tratta di una lista i cui valori rappresentano un rettangolo che delimita lo spazio di movimento del soggetto all'interno del video;
- `fps`: descrive il numero di frame per secondo a cui è stato registrato il video. Tutti i video sono registrati a 25 fps;
- `frame_end`: mostra a che frame termina il video, e ne determina quindi la durata del contenuto informativo;
- `frame_start`: mostra a che frame il contenuto informativo del video comincia;
- `instance_id`: è un indice che va ad incrementarsi sulla base di quante volte appare la parola, dato che per ciascuna si hanno più video in cui viene mostrata;
- `signer_id`: si tratta di un identificatore numerico che indica quale soggetto viene ripreso nel video;

- `source`: il dataset è stato creato unendo i video provenienti da 19 diverse sorgenti, il valore mostra la sorgente da cui è stato preso il video;
- `split`: il dataset è già stato diviso negli split di training, validation e test;
- `url`: si tratta di un link al quale è reperibile il video.
- `variation_id`: vengono parlati 3 diversi dialetti della lingua dei segni americana, questo indice indica quale di queste il soggetto sta utilizzando;
- `video_id`: si tratta di un identificatore numerico utilizzato per identificare il video a cui si sta facendo riferimento.

Di seguito è mostrata la struttura del dataset per uno specifico video preso come esempio, il video di riferimento è **videos/69241.mp4**.

```
[
  {
    "gloss": "book",
    "instances": [
      {
        "bbox": [
          385,
          37,
          885,
          720
        ],
        "fps": 25,
        "frame_end": 75,
        "frame_start": 1,
        "instance_id": 0,
        "signer_id": 118,
        "source": "aslbrick",
        "split": "train",
        "url": "http://aslbricks.org/New/ASL-Videos/book.mp4",
        "variation_id": 0,
        "video_id": "69241"
      }
    ]
  }
]
```

Listato 1.1: Struttura del dataset, come descritto sopra.

1.2 Elaborazione dei dati

A seguito dell'analisi 1.1 sono emersi alcuni problemi con la struttura attuale del dataset; di seguito è fornita una descrizione delle modifiche apportate.

1.2.1 Modifica del parametro frame_end

Come precedentemente analizzato, il file .json che descrive l'intero dataset espone il campo `frame_end`, il quale risulta però pari a -1 nella maggior parte dei casi. Si è quindi preferito modificare questo valore per rispecchiare il numero di frame di cui è composto il video. Esplicitare questo valore ha semplificato e reso meno prone a errori l'utilizzo del dataset; inoltre, questa variazione permette di modificare i video stessi, ad esempio andando ad aggiungere nuovi frames di padding, senza alterare il contenuto informativo iniziale.

```
class Dataset:
    # ...
    def update_videos(self) -> None:
        with open(self.filename) as f:
            data = json.load(f) # caricamento del json del dataset
        for word_data in data:
            gloss = word_data["gloss"]
            instances = word_data["instances"]
            for instance in instances:
                video = Video.from_instance(gloss, instance,
                    self._only_samples)
                if not video.is_missing():
                    instance["frame_end"] = video.get_end()
                else:
                    instance = None
        with open(self.filename, "w") as f:
            json.dump(data, f, indent=4)
    # ...
```

Listato 1.2: Codice per l'aggiornamento del file WLASL_v0.3.json

```
class Video:
    # ...
    def get_end(self) -> int:
        if self.frame_end != -1:
            return self.frame_end
        ret = self.get_video_capture().get(cv2.CAP_PROP_FRAME_COUNT)
        print(f"Video {self.video_id} has {ret} frames")
        self.get_video_capture().release()
        return int(ret)
    # ...
```

Listato 1.3: Codice per ottenere la lunghezza di un video

1.2.2 Region Of Interest (ROI)

Altro parametro che è risultato molto utile è la lista di valori della *bounding box*, questi valori, nell'ordine `(xmin, ymin, xmax, ,ymax)`, hanno permesso la rimozione di porzioni non interessanti dei video. Il secondo passo del processo di pre-elaborazione è infatti stato quello di ritagliare ogni frame del dataset, mantenendo solamente la porzione descritta all'interno del campo `bbox` del dataset.

```
class RoiExtractor:
    def extract(self, remove_background=False) -> List["np.ndarray"]:
        frames = []
        for frame in self.frames:
            frame = frame[self.bbox[1] : self.bbox[3], self.bbox[0] :
                self.bbox[2]]
            frame = self._resize(frame)
            frames.append(frame)
        return frames
    # ...
```

Listato 1.4: Codice per l'estrazione della ROI

1.2.3 Ridimensionamento e padding dei video

I video all'interno del dataset hanno dimensioni diverse, ora che in ogni frame è stata rimossa la porzione non contenente contenuto informativo è possibile unificare le dimensioni dei fotogrammi. Ogni frame di ogni video è quindi stato ridimensionato a 224x224 pixel, aggiungendo dove opportuno dei pixel

di padding, in modo tale da mantenere inalterato il rapporto tra altezza e larghezza dei video. Siccome alcuni algoritmi, come SVC, si aspettano vettori di features delle stesse dimensioni, sono stati aggiunti un tot di frames non significativi al termine di ogni video, portando la lunghezza a 240 frames. Questa modifica non impatterà gli algoritmi che permettono l'analisi di sequenze con lunghezze diverse, in quanto `frame_end` viene mantenuto pari alla lunghezza del video originale come descritto precedentemente 1.2.1.

```
class ROIExtractor:
    # ...
    def _resize(self, frame) -> "np.ndarray":
        if self.resize is None:
            return frame
        if frame.shape[0] >= frame.shape[1]:
            height = self.resize
            width = int(frame.shape[1] * self.resize / frame.shape[0])
        else:
            width = self.resize
            height = int(frame.shape[0] * self.resize /
                frame.shape[1])
        frame = cv2.resize(frame, (width, height),
            interpolation=cv2.INTER_LINEAR)
        frame = self._pad(frame)
        return frame

    def _pad(self, frame: "np.ndarray") -> "np.ndarray":
        if self.resize is None:
            return frame
        pad_width = self.resize - frame.shape[0]
        pad_height = self.resize - frame.shape[1]
        my_padded_image = np.pad(
            frame, ((0, pad_width), (0, pad_height), (0, 0)),
                "constant"
        )
        return my_padded_image
```

Listato 1.5: Codice per il resizing e padding dei video

1.2.4 Rimozione del background

Si è osservato che i video girati dalle stesse persone hanno il medesimo *background* e anche i vestiti sono gli stessi. Per il secondo problema preso in esame per questo progetto, ovvero il riconoscimento dei *signer* si è pensato che

questo potesse essere un problema e quindi si è proceduto a pre-processare di nuovo il dataset per ottenere i video senza sfondo.

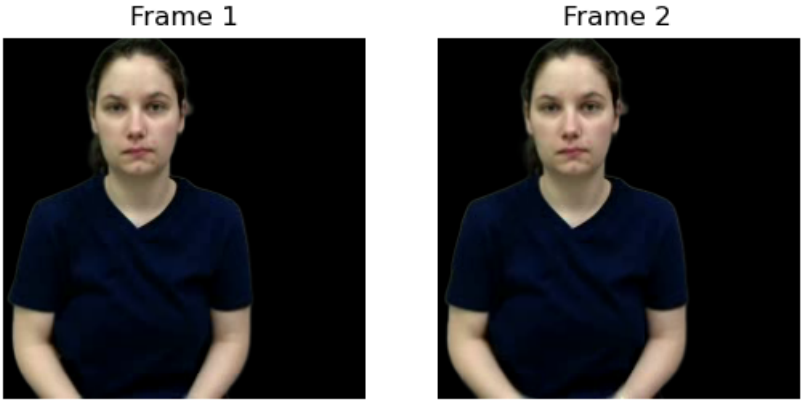
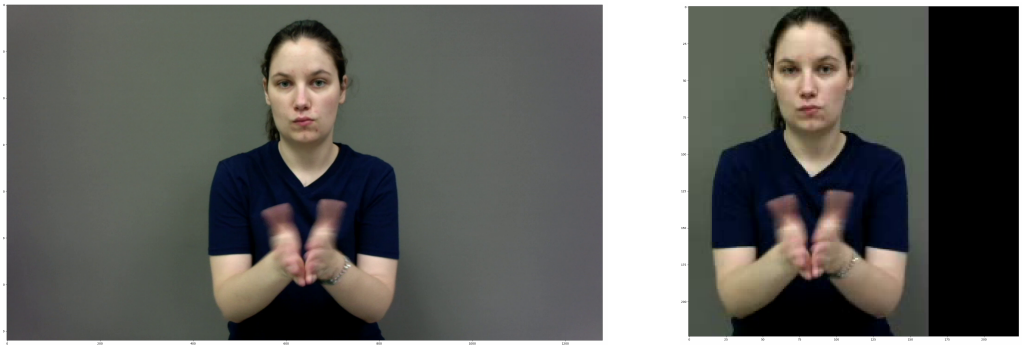


Figura 1.1: Rimozione sfondo

1.2.5 Risultati ottenuti

La pre-elaborazione risultante dal ridimensionamento dei video ha portato al vantaggio della riduzione di dimensione dell'intero dataset, il quale è passato a pesare da 5 GB a meno di 1,5 GB. Il processo ha impiegato 3 ore di esecuzione, elaborando 50 video in parallelo; in questo modo verrà risparmiato tempo nelle esecuzioni successive. Di seguito, l'esempio mostra un frame prima e dopo l'elaborazione.



(a) Frame originale (b) Frame pre-processato

Figura 1.2: Differenza tra i frames

Capitolo 2

Hand-crafted features

Dopo aver analizzato e pre-processato i dati, è stata effettuata l'estrazione delle features. Le feature hand-crafted rappresentano un insieme di caratteristiche estratte manualmente dai frame appartenenti ai video, mirando a catturare aspetti specifici e distintivi che possono facilitare la successiva fase di classificazione. Tra le principali categorie di feature estratte vi sono quelle legate alla **forma**, alla **tessitura**, al **colore** e al **movimento**. Sebbene ogni categoria possa offrire un contributo variabile in termini di efficacia, la loro combinazione consente di migliorare significativamente le prestazioni dei sistemi di riconoscimento.

2.1 Feature di forma

Per ottenere i descrittori di forma, è stato condotto uno studio dei contorni delle persone nei video.

2.1.1 Edge extractor

Per la ricerca dei bordi, sono state effettuate una serie di trasformazioni a ciascun frame. Si è reso necessario:

- convertire l'immagine in scala di grigi:

```
gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

- applicare un filtro di sfocatura gaussiana, per migliorare la robustezza al rumore:

```
blurred_frame = cv2.GaussianBlur(gray_frame, (5, 5), 0)
```

Il parametro (5, 5) rappresenta le dimensioni del kernel della finestra gaussiana e 0 indica la deviazione standard lungo l'asse X.

- applicare un equalizzatore di istogrammi, usato per migliorare il contrasto dell'immagine, specialmente quando l'istogramma è concentrato in determinati intervalli di intensità:

```
equalize_hist = cv2.equalizeHist(blurred_frame)
```

- l'uso dell'operatore Sobel per calcolare il gradiente dell'immagine equalizzata. L'operatore Sobel è utilizzato per rilevare i bordi in un'immagine calcolando le derivate di primo ordine:

```
gx = cv2.Sobel(equalize_hist, cv2.CV_64F, 1, 0, ksize=3)
gy = cv2.Sobel(equalize_hist, cv2.CV_64F, 0, 1, ksize=3)
sobel_image = cv2.convertScaleAbs(np.sqrt((gx ** 2) + (gy ** 2)))
```

  Le prime due operazioni calcolano la derivata rispettivamente sull'asse X e Y utilizzando l'operatore Sobel sull'immagine equalizzata (`equalize_hist`). Il risultato (`gx`) è un'immagine con le intensità dei gradienti lungo l'asse X, mentre il risultato (`gy`) è un'immagine con le intensità dei gradienti lungo l'asse Y. L'ultima operazione combina le immagini dei gradienti `gx` e `gy` per ottenere l'immagine del gradiente totale (`sobel_image`). La formula utilizzata calcola la magnitudo del vettore gradiente.

- l'uso di un metodo per binarizzare un'immagine. La funzione *threshold* rende i pixel sopra una certa soglia bianchi e quelli al di sotto neri.

```
_, binary_frame = cv2.threshold(sobel_image, 0, 255,
                  cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

  I parametri sono impostati come segue:

  - 0: Soglia minima. I pixel con intensità al di sotto di questa soglia saranno impostati a 0.
  - 255: Soglia massima. I pixel con intensità al di sopra di questa soglia saranno impostati a 255.
  - `cv2.THRESH_BINARY + cv2.THRESH_OTSU`: indica che la soglia viene automaticamente calcolata utilizzando l'algoritmo di Otsu. L'algoritmo di Otsu trova la soglia ottimale in modo da separare le classi (in questo caso, i pixel di sfondo e i pixel della persona in oggetto).

- l'uso di un'operazione di chiusura morfologica. L'operazione è composta da una dilatazione seguita da un'erosione. Questo processo è utile per chiudere i piccoli buchi e connettere alcune regioni unite debolmente.

```
kernel = np.ones((3, 3), np.uint8)
result_frame = cv2.morphologyEx(binary_frame, cv2.MORPH_CLOSE, kernel)
```

Il kernel è formato da una matrice quadrata 3x3 composta interamente da elementi unitari ed è stata usata l'operazione di chiusura morfologica `cv2.MORPH_CLOSE`.

Il risultato che si ottiene (`result_frame`) è l'immagine modificata in cui sono stati evidenziati i contorni. Di seguito viene mostrata l'immagine di un frame a cui sono state applicate le trasformazioni appena descritte.

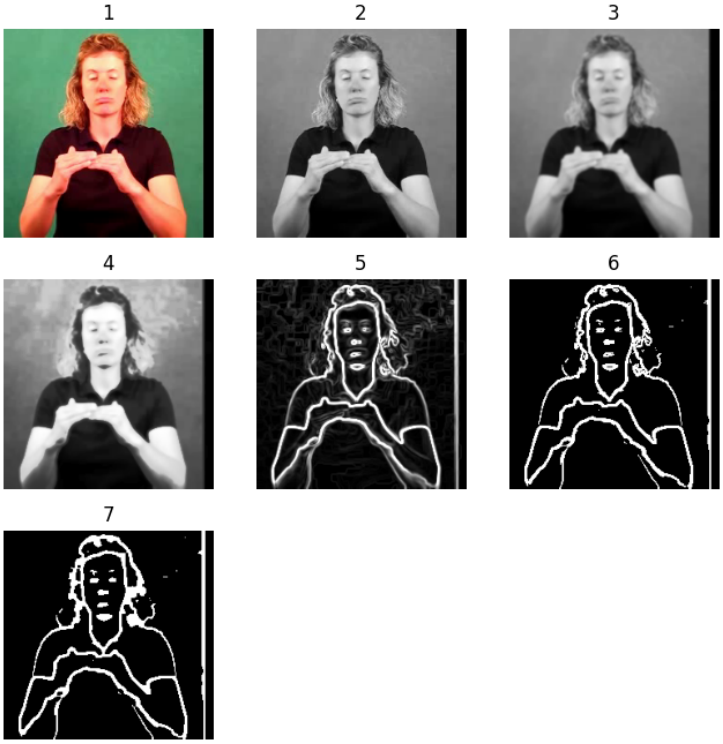


Figura 2.1: Edge detection: trasformazioni applicate ad un frame

E' stato valutato un ulteriore metodo di estrazione dei contorni, tramite le funzioni già implementate di OpenCV: `cv2.findContours` e `cv2.drawContours`. I risultati si sono dimostrati peggiori visivamente e quindi si è scelto il metodo precedente.

2.1.2 HoG extractor

Histogram of Oriented Gradients (HoG) è una tecnica utilizzata per l'estrazione di caratteristiche dalle immagini, spesso impiegata nel rilevamento

di oggetti. In questo caso si vogliono catturare le informazioni sulle transizioni di gradiente nell'immagine, fornendo una rappresentazione compatta e discriminante delle caratteristiche della persona presa in esame. Le caratteristiche HoG sono efficaci nel catturare informazioni sulla struttura e sull'aspetto generale, ma possono non essere ottimali per riconoscere gesti specifici delle mani o delle dita, come quelli coinvolti nel linguaggio dei segni. In seguito sono stati considerati approcci più specifici che tengono conto delle caratteristiche del volto, delle mani e dei gesti particolari coinvolti nel linguaggio dei segni. Le features HoG sono state ottenute tramite la libreria `skimage.feature` che ha fornito una funzione `hog()` che accetta come input un singolo canale colore o un'immagine in scala di grigi, oltre a vari parametri.

```
hog_params = {
    "orientations": 9,
    "pixels_per_cell": (8, 8),
    "cells_per_block": (2, 2),
    "block_norm": "L2-Hys",
    "visualize": True,
    "transform_sqrt": True
}
hog_features, hog_image = hog(gray_frame, **hog_params)
```

I parametri utili da passare alla funzione sono contenuti in una variabile chiamata `hog_params`, al cui interno è specificato:

- il numero di `orientations`, un numero intero che rappresenta il numero di bin dell'istogramma di orientamento;

- il parametro `pixels_per_cell` indica in pixel (tupla di due elementi x e y) la dimensione della cella su cui viene calcolato ciascun istogramma gradiente;

- il parametro `cells_per_block` viene anch'esso passato come tupla di due elementi e specifica l'area locale su cui verrà normalizzato il conteggio dell'istogramma in una determinata cella;

- `block_norm` è il tipo di normalizzazione da applicare ai blocchi. "`L2-Hys`" è una combinazione di normalizzazione L2 seguita da un processo di *thresholding* (istogramma con soglia). Questo tipo di normalizzazione è spesso utilizzato per migliorare la robustezza dell'algoritmo HoG a variazioni di luminosità e contrasto nell'immagine. La normalizzazione L2 è particolarmente utile per normalizzare i vettori delle caratteristiche, mentre il *thresholding* aiuta a concentrarsi su caratteristiche più rilevanti;

- impostare `visualize` a `True` significa che oltre a restituire le caratteristiche HoG estratte, ritorna anche l'immagine dell'istogramma di orientazione risultante;
- se il valore *booleano* di `transform_sqrt` è pari a `True`, allora applica una normalizzazione che può aiutare a ridurre gli effetti delle ombre e migliorare la robustezza nei confronti delle variazioni di luminosità. In pratica, viene calcolata la radice quadrata dei valori dell'immagine.

In figura 2.2, si può vedere un esempio dell'applicazione della funzione HoG.

Figura 2.2: HoG extraction

2.2 Feature di texture

I descrittori di tessitura sono utilizzati per rappresentare le caratteristiche strutturali di una superficie o di un *pattern* in un'immagine. Essi permettono di identificare proprietà ripetitive e statistiche della distribuzione spaziale delle variazioni di intensità. Nel nostro studio, abbiamo utilizzato i seguenti metodi: **Haar-like features** e **Local Binary Patterns** (LBP).

2.2.1 Haar-like features

L'analisi delle caratteristiche di tessitura è stata condotta tramite l'estrazione di caratteristiche basate sul metodo di Haar, con particolare enfasi sull'utilizzo del localizzatore di *Viola e Jones* per individuare la posizione del viso del soggetto. Inizialmente, si è tentato di estendere l'applicazione di questo localizzatore per identificare anche la posizione delle mani; tuttavia, questa strategia è stata abbandonata a causa dei risultati insoddisfacenti. La principale ragione di tali risultati risiede nella notevole variabilità che un arto in movimento può presentare. Mentre il volto, nel dataset, è per lo più non oscurato e orientato frontalmente rispetto alla camera, le mani sono soggette a continui movimenti e adottano posizioni altamente variabili, rendendo difficile la loro

identificazione attraverso metodi tradizionali. Il primo step è la creazione del classificatore; si è utilizzato `CascadeClassifier` di OpenCV, un localizzatore basato sul metodo di Haar e l'algoritmo di *Viola e Jones*, andando a caricare un localizzatore già addestrato per la localizzazione di volti.

```
classifier = cv2.CascadeClassifier()
classifier.load(cv2.samples.findFile("app/haar/haarcascades/face.xml"))
```

In seguito vengono fatte alcune analisi frame per frame del video:

- viene equalizzato l'istogramma del frame in scala di grigi, aumentandone il contrasto:

  ```
  gray_hist = cv2.equalizeHist(frame.gray)
  ```

- si utilizza il classificatore precedentemente caricato per estrarre le posizioni del volto:

  ```
  detections = self.classifier.detectMultiScale(gray_hist)
  ```

- siccome `CascadeClassifier` può restituire multiple posizioni è stato necessario selezionare quella ritenuta migliore. Dall'analisi del dataset 1.1 si è osservato come il volto tende a non cambiare posizione durante il video. La posizione migliore viene quindi scelta sulla base delle posizioni precedentemente identificate, sfruttando la distanza euclidea da esse.

  ```
  dist = np.sqrt(
      (detection[0] - avg_pos[0]) ** 2 + (detection[1] - avg_pos[1]) ** 2
  )
  ```

  Le prime posizioni potrebbero però risultare errate sfruttando questa tecnica, in quanto il primo volto che viene scelto dipende dalla confidenza che il classificatore assegna alla posizione. Nei primi dieci frame le posizioni vengono quindi ricalcolate utilizzando la posizione media aggiornata.

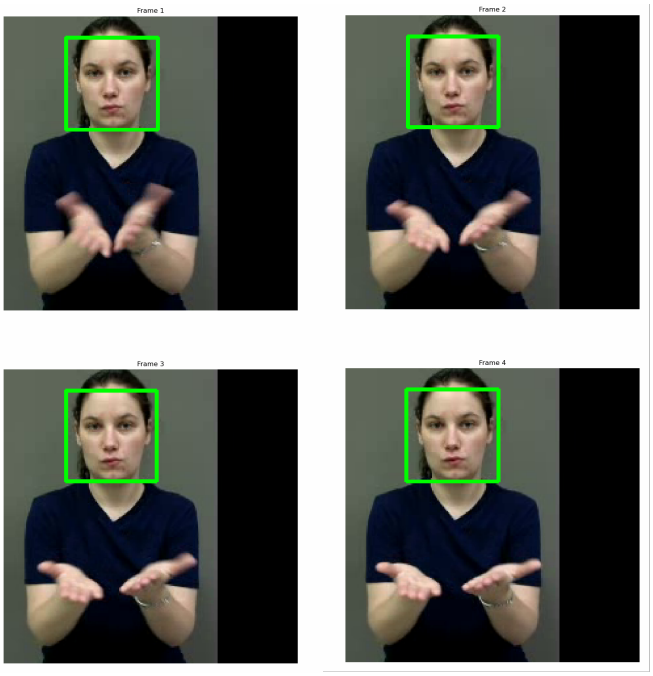


Figura 2.3: Haar detection su una serie di frames

La posizione del volto risulta utile, ma gli arti sono la componente più importante per interpretare il gesto in esame. L'estrazione degli arti si basa sulla posizione estratta dal localizzatore sopra citato. Partendo dalla posizione estratta utilizzando *Viola e Jones*:

- siccome la posizione individuata dal localizzatore risulta troppo larga ed alta si vanno a modificare questi parametri:

```
new_w = int(w * 0.6)
new_h = int(h * 0.6)
x = int(x + (w - new_w) / 2)
y = int(y + (h - new_h) / 2)
w = new_w
h = new_h
head_region = frame[y : y + h, x : x + w]
```

- successivamente si trasforma lo spazio colore in HSV in quanto permette maggiore precisione nel riconoscimento di aree di colore simile e si estrae il valore medio di tonalità, saturazione e luminosità:

```
hsv_head = cv2.cvtColor(head_region, cv2.COLOR_BGR2HSV)
mean_color = np.mean(hsv_head, axis=(0, 1))
```

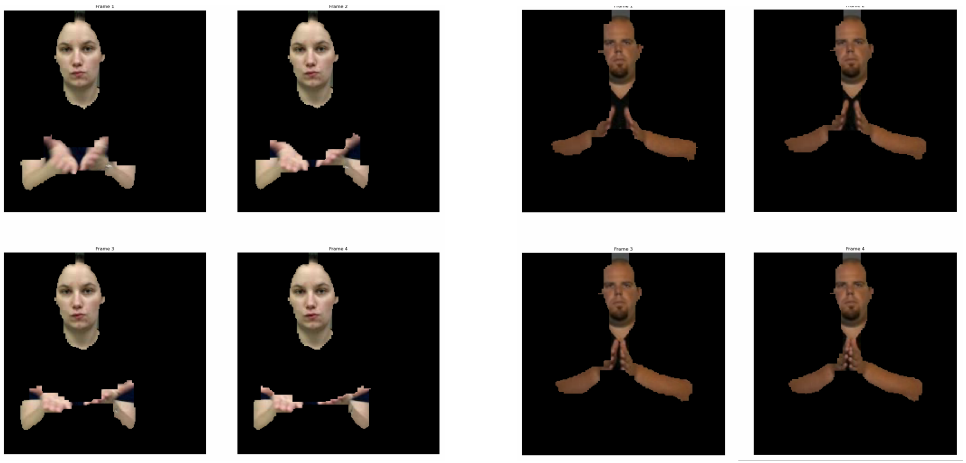
- si utilizza un intorno dei valori trovati per estrarre tutte le porzioni dell'immagine nelle quali è presente la pelle del soggetto, ovvero arti e viso:

```
mask = cv2.inRange(hsv, lower_skin, upper_skin)
```

- si applica poi un'operazione di chiusura morfologica, permettendo di chiudere alcuni buchi che si verificano nella maschera:

```
kernel = np.ones((35, 35), np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
```

La combinazione della posizione estratta sfruttando il localizzatore Haar Cascade in combinazione con l'elaborazione del colore ha permesso di ottenere risultati simili anche in presenza di differenti livelli di illuminazione, colore, texture e movimenti del soggetto in esame.



(a) Soggetto 1 (b) Soggetto 2

Figura 2.4: Segmentazione di volto ed arti

2.2.2 Local Binary Pattern

Il LBP è un metodo per descrivere i pattern locali in un'immagine. L'idea è confrontare il valore del pixel centrale con i pixel circostanti, creando un numero binario che rappresenta la relazione tra il pixel centrale e i suoi vicini. Il calcolo di LBP si può rappresentare con la seguente formula:

$$LBP_{P,R}(g_c) = \sum_{i=0}^{P-1} s(g_i - g_c) \cdot 2^i \tag{2.1}$$

Dove:

- P: è il numero di vicini che si vuole considerare;
- R: è il raggio della circonferenza su cui si trovano i pixel vicini;
- g_c: è il valore di intensità del pixel centrale;
- g_i: è il valore di intensità dei pixel vicini;
- *s(x)*: è una funzione soglia che restituisce 1 se x è maggiore o uguale a 0, e 0 se x è minore a 0.

La funzione soglia è definita come segue ed è usata per confrontare il valore di ciascun vicino col valore del pixel centrale.

$$s(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ 0 & \text{se } x < 0 \end{cases} \tag{2.2}$$

Passi per calcolare il LBP:

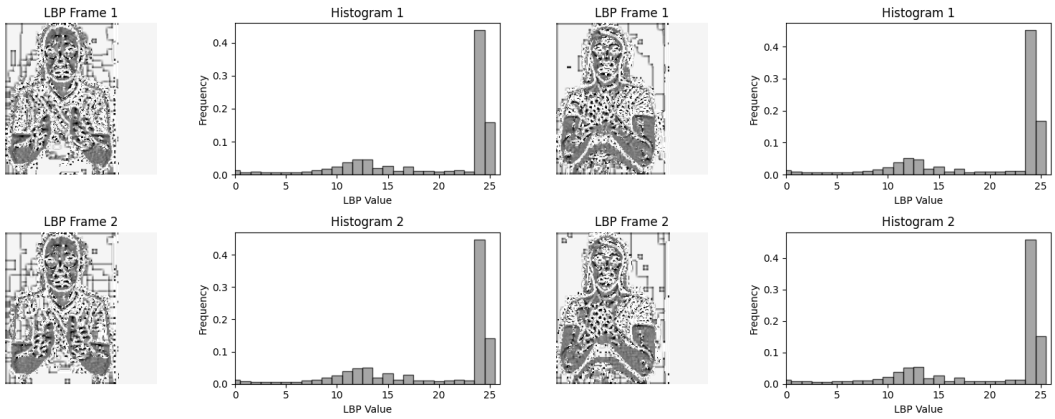
1. Selezionare il pixel centrale g_c;
2. Esaminare i vicini g_i: scegliere P vicini disposti attorno al pixel centrale, sono distribuiti su una circonferenza con raggio R;
3. Per ciascun pixel vicino g_i calcolare la differenza con il valore del pixel centrale;
4. Applicare la funzione soglia *s(x)*;
5. Combinare i risultati in una stringa binarie: dopo aver applicato la soglia a ciascun vicino, si otterrà una sequenza di valori binari. Questi valori vengono combinati per formare una stringa binaria, dove ogni valore è associato a un peso di potenza di 2;
6. Sommare i valori binari creando un unico valore numerico che rappresenta il pattern locale.

Nel progetto si è fatto uso della funzione `local_binary_pattern()` della libreria `skimage.feature`. Importante notare che i frame, prima di essere processati, vengono convertiti in scala di grigi e l'istogramma viene normalizzato. Di seguito un estratto del codice della classe `LBPExtractor`:

```
lbp_frame = local_binary_pattern(
    frame, self._n_points, self._radius, method="uniform"
)
```

```
hist, bins = np.histogram(
    lbp_frame.ravel(),
    bins=np.arange(0, self._n_bins + 1),
    range=(0, self._n_bins),
)
hist = hist.astype(np.float32) / hist.sum()
```

Gli istogrammi rappresentano le feature estratte, mentre i frame e i bins vengono utilizzati meramente per scopi di visualizzazione. Sotto è possibile vedere un esempio di applicazione del LBP sui frame e a fianco i relativi istogrammi, prendendo due video di riferimento.



(a) Soggetto 1 (b) Soggetto 2

Figura 2.5: Visualizzazione LBP sui frame e relativi istogrammi

2.3 Feature colore

Le feature di colore catturano la distribuzione cromatica presente in un'immagine, fornendo informazioni che possono essere sfruttate per la classificazione e il riconoscimento dei soggetti. Queste caratteristiche risultano particolarmente utili in contesti dove le informazioni basate sulla forma o sulla texture siano insufficienti.

2.3.1 Color histogram

Il color histogram è una tecnica che consente di rappresentare la distribuzione dei colori in un'immagine, suddividendo lo spazio dei colori in un numero definito di bins e contabilizzando la frequenza di ciascuna tonalità. Grazie alla normalizzazione, questo metodo permette un confronto efficace tra immagini

di dimensioni diverse, risultando robusto rispetto a variazioni di illuminazione e piccole traslazioni. La scelta dello spazio colore e del numero di bins rappresenta un aspetto critico per garantire una rappresentazione accurata e discriminante.

Per il calcolo dell'istogramma è stata sfruttata la funzione `calcHist` della libreria OpenCv. La scelta dello spazio colore è fondamentale per andare a mostrare diverse caratteristiche dell'immagine, il codice per l'estrazione dell'istogramma rimane però lo stesso:

```
def _extract_separate(frame: np.ndarray, normalize: bool):
    hists = []
    for i in range(3):
        hist = cv2.calcHist([frame], [i], None, [256], [0, 256])
        if normalize:
            hist = cv2.normalize(hist, hist, 0, 1, cv2.NORM_MINMAX)
        hists.append(hist)
    return hists
```

In questo modo viene estratto, per ogni componente dello spazio colore, un istogramma da 256 bin, che può, in caso, essere opportunamente normalizzato. Il passo di normalizzazione rende la feature più robusta rispetto a cambiamenti di scala e traslazioni.

Di seguito vengono mostrati 4 frame di due diversi video nello spazio *HSV* e i relativi istogrammi colore normalizzati:

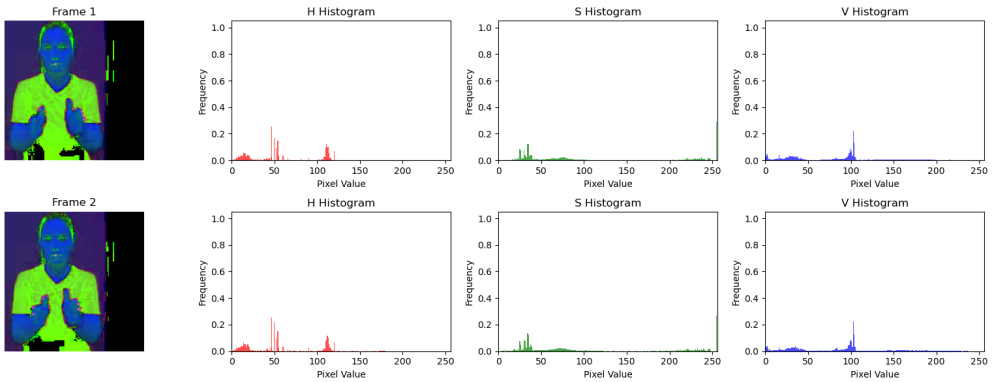


Figura 2.6: Soggetto 1

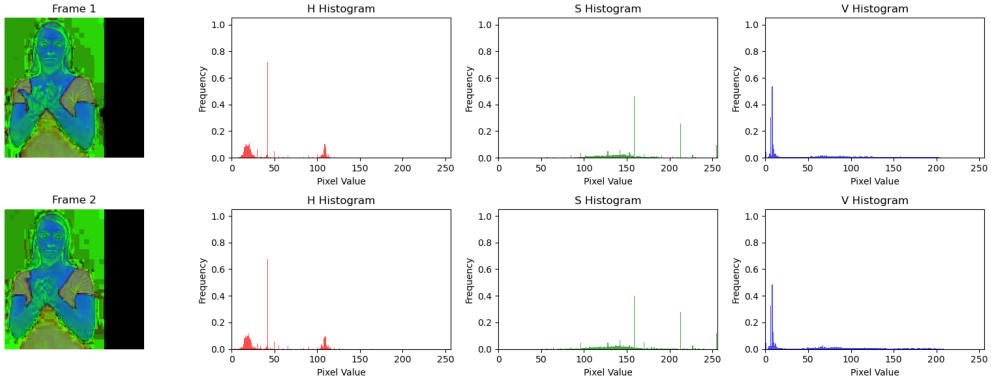


Figura 2.7: Soggetto 2

Figura 2.8: Visualizzazione frame nello spazio HSV e relativi istogrammi

2.4 Feature di movimento

Le features di movimento descrivono le variazioni spazio-temporali in una sequenza di frame, andando ad esaltare il movimento degli oggetti nel tempo.

2.4.1 Optical flow

Per la ricerca delle features di movimento sono state utilizzate due diverse implementazioni di algoritmi forniti dalla libreria OpenCV. La prima implementazione utilizzava l'algoritmo di *Lucas-Kanade* andando a mostrare il tracciamento di alcuni keypoint specifici. I risultati di questa tecnica sono scarsi per il caso in esame, in quanto la variabilità di movimento e rotazione delle mani rendeva inefficiente il tracciamento dei vari keypoint identificati. Si è quindi implementata la versione che sfrutta l'algoritmo di *Gunnar Farnebäck*:

- si calcola il risultato dell'algoritmo andando a confrontare 2 frame successivi, e passando i parametri necessari ed ottimizzati per il problema in oggetto:

```
self.params = {
    "pyr_scale": 0.3,
    "levels": 5,
    "winsize": 10,
    "iterations": 6,
    "poly_n": 5,
    "poly_sigma": 1.5,
    "flags": 0,
}
```

```
flow = cv2.calcOpticalFlowFarneback(
    prev_frame, next_frame, None, **self.params
)
```

I parametri utili da passare alla funzione sono contenuti in una variabile chiamata `params`, al cui interno è specificato:

- `pyr_scale`: indica quanto ridurre l'immagine a ogni livello della piramide;
- `levels`: numero di livelli della piramide;
- `winsize`: dimensione della finestra per le medie ponderate;
- `iterations`: numero di iterazioni per ogni livello della piramide;
- `poly_n`: dimensione della finestra per il filtro polinomiale;
- `poly_sigma`: sigma per il filtro polinomiale Gaussiano.

- l'algoritmo ritorna due vettori che mostrano il movimento dei pixel nell'immagine di destinazione. I valori in particolare sono chiamati **angle** e **magnitude** estratti nel modo seguente:

```
magnitude, angle = cv2.cartToPolar(flow[..., 0], flow[..., 1])
```

  - **angle** rappresenta la direzione del movimento del pixel;
  - **magnitude** rappresenta l'intensità di movimento. In particolare, un valore elevato di questo parametro va ad indicare un movimento più veloce del pixel in questione.

- i valori estratti vengono quindi normalizzati, andando a trasformare *angle* in un valore tra 0 e 180 e *magnitude* tra 0 e 255. Si assegnano quindi al layer di tonalità e luminosità di un nuovo frame rispettivamente, permettendo quindi la stampa a video:

```
hsv[..., 0] = angle * 180 / np.pi / 2
hsv[..., 2] = cv2.normalize(magnitude, None, 0, 255, cv2.NORM_MINMAX)
```

Segue un esempio di calcolo dell'optical flow su un video.

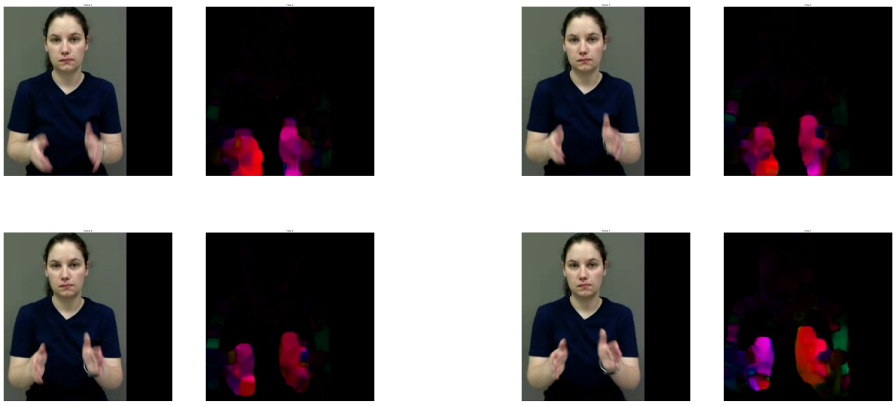


Figura 2.9: Risultati del calcolo dell'optical flow su 4 frames consecutivi.

2.5 Estrazione keypoint con MediaPipe

Tra le varie soluzioni presenti su Kaggle, quella più completa estrae le features utilizzando MediaPipe[1], un framework open source per l'elaborazione di dati multimediali. Data la complessità del problema, si è pensato di utilizzarlo nel progetto per confrontarne poi i risultati. In particolare, è stato utilizzato il modello `holistic`, che permette di estrarre i keypoints relativi al volto, alla posa e alle mani dei soggetti. Le features che vengono estratte da MediaPipe sono estremamente semplici e di facile interpretazione. Un esempio è possibile vederlo in Figura 2.10.

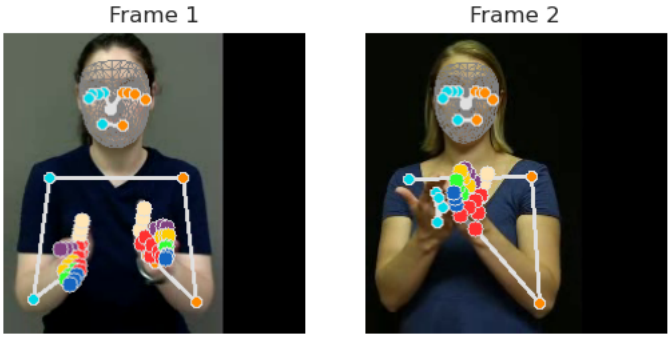
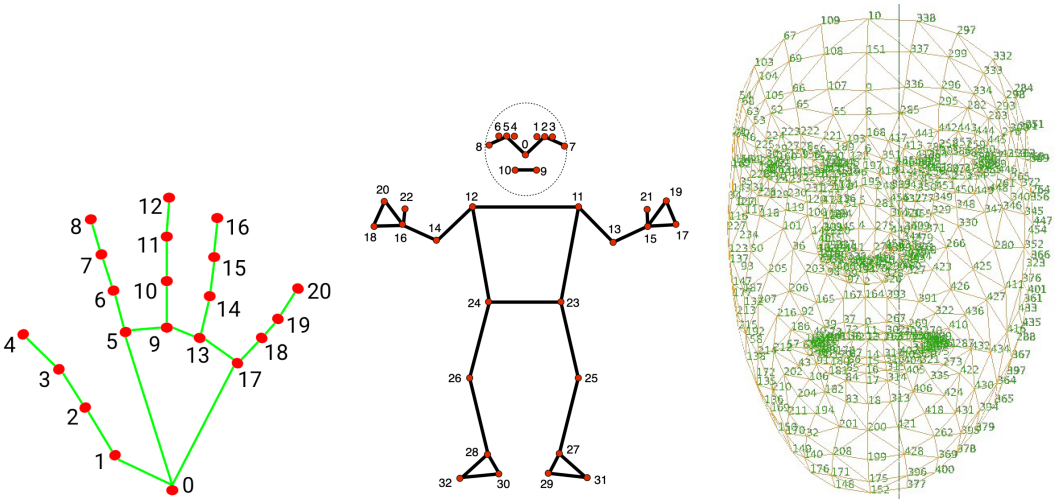


Figura 2.10: Visualizzazione dei keypoint estratti tramite MediaPipe.

L'ordine dei keypoint non è casuale ma segue regole ben precise, nel caso della mano, il punto 0 è sempre il palmo, mentre gli altri indicano le falan-

[1] https://ai.google.dev/edge/mediapipe/solutions/guide

gi partendo dal pollice e crescendo di numero verso il mignolo. Posa e volto seguono regole simili. Nel nostro caso risulta interessante il fatto che in questo modo è possibile andare a confrontare i keypoint direttamente con quelli estratti in altri frame, o da altri soggetti. Come vengono assegnati i keypoint:



(a) Hand (b) Pose (c) Face

Figura 2.11: Keypoints: mani, posa e volto

Per ottenere un miglior risultato durante il processo successivo di data augmentation, si è deciso di utilizzare il dataset wlasl-resized[2], in quanto i video non hanno padding solo a destra del soggetto, ma mantengono il soggetto al centro del frame. I dati estratti sono stati salvati in un file `.pickle`, questo per permettere di utilizzarli in un secondo momento senza dover rielaborare i frame. A seguito di questo passo, la dimensione del dataset è diventata 18GB, un incremento significativo, ma necessario per evitare di dover effettuare nuovamente l'estrazione dei keypoints ad ogni esecuzione, il che comportava un grande *overhead* computazionale.

I keypoint estratti da MediaPipe seguono il seguente formato:

```
landmark {
  x: 0.5392837524414062
  y: 0.7506672739982605
  z: -0.06435433775186539
}
```

[2] https://www.kaggle.com/datasets/sttaseen/wlasl2000-resized

Capitolo 3

Classificazione video

In questo capitolo si entrerà nel merito della classificazione per quanto riguarda il problema del riconoscimento dei segni. Innanzitutto si è proceduto a effettuare la classificazione estraendo le feature hand-crafted tramite le tecniche descritte precedentemente e classificate utilizzando `Dynamic Time Warping`. In seguito verranno poi utilizzate le feature estratte con MediaPipe e classificate sempre con `Dynamic Time Warping`. Infine verranno messi a confronto i risultati ottenuti.

3.1 Dynamic Time Warping (DTW)

L'algoritmo `Dynamic Time Warping (DTW)`[2] è stato implementato per la classificazione dei video, rivelandosi particolarmente efficace. La scelta di DTW è stata dettata dalla sua peculiarità di gestire e confrontare serie temporali con velocità e durate variabili, un aspetto cruciale nell'analisi di sequenze video. La distanza euclidea, invece, non riesce a gestire il disallineamento delle sequenze temporali. Si può vedere nell'esempio di seguito:

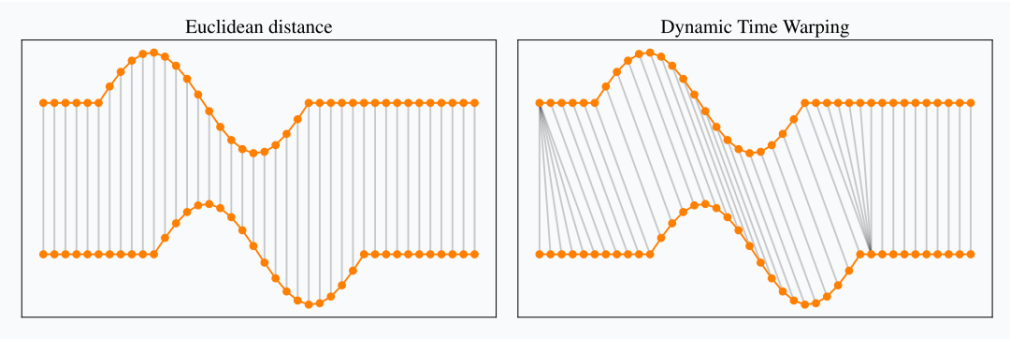


Figura 3.1: Distanza Euclidea VS DTW

La metodologia adottata prevede il calcolo della distanza DTW tra le feature estratte da ogni video del *test set* e ogni video del *training set*. Questi valori di distanza, misurano la similarità tra i video, sono stati poi utilizzati come base per l'assegnazione della classe (parola del linguaggio dei segni) ai video di test.

In particolare si è utilizzata la libreria `fastdtw`[1], in quanto permette di ridurre il costo dell'algoritmo da $O(N^2)$ a $O(N)$.

3.2 Classificazione tradizionale

Per la classificazione in maniera tradizionale si è proceduto innanzitutto a creare una classe `DTWClassifier` che prende in ingresso il `Dataset` e le label `glosses`.

Suddivisione training e test set

La suddivisione dei dati di *training* e *test* segue la struttura definita nel file `WLASL_v0.3.json`. Siccome i video per ogni *gloss* non sono molti, per non lasciare inutilizzato il *validation* è stato aggiunto al *training set*.

Features hand-crafted

Le feature estratte per la classificazione tradizionale sono quelle descritte nel Capitolo 2. Tuttavia, tra tutte le caratteristiche considerate, sono state utilizzate soltanto le `HOG` e le `LBP`, in quanto, sulla base di diversi test empirici, si sono dimostrate le più efficaci. Si ipotizza che tale risultato sia dovuto alla loro semplice interpretabilità e al fatto che risultano essere le più discriminanti.

Classificazione

A questo punto si è proceduto a richiamare il metodo `compute_dtw_similarity_matrix` che ritorna la matrice di similarità per il test set, le label associate al training set e le vere label del test set.

La matrice di similarità viene calcolata andando a confrontare ogni video del *test set* con tutti i video del *training set*. Viene quindi effettuato un confronto per ogni coppia di video ottenendo le features per ciascuna e, dopo averle concatenate, vengono passate alla funzione `dtw_kernel`, che calcola la distanza DTW con `fastdtw` tra le sequenze. Questo valore di distanza viene poi normalizzato, come si può vedere dalla funzione mostrata di seguito (così

[1] https://github.com/rmaestre/FastDTW

da mantenere un valore compreso tra 0 e 1) e trasformato in similarità che risulta più facilmente interpretabile.

$$\mathrm{ND} = \frac{\mathrm{D}}{1 + \mathrm{D}} \tag{3.1}$$

$$\mathrm{s} = e^{-\mathrm{ND}} \tag{3.2}$$

Viene poi, richiamato il metodo `dtw_predict` sulla matrice `X_test` e `y_train` per classificare i video di *test* associandoli alla parola del gloss che ha similarità più alta.

3.3 Classificazione con MediaPipe

Per la classificazione delle parole del linguaggio dei segni, utilizzando i *keypoints* estratti con `MediaPipe`, si è deciso di utilizzare solamente quelli relativi alle mani dei soggetti, in quanto più significativi.

Siccome la classificazione sarà effettuata con l'algoritmo `DTW`, si possono osservare alcuni potenziali problemi con il formato dei dati che MediaPipe fornisce, come mostrato nella Sezione 2.5, in particolare nella rappresentazione del formato dei landmark. Trattandosi di coordinate relative, queste sono dipendenti dalla posizione e dalla scala all'interno dell'immagine. Per ovviare a questo problema, sempre per quanto riguarda le mani, durante il caricamento dei video si va a effettuare il calcolo dell'angolo in radianti:

$$\theta = \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}\right) \tag{3.3}$$

Dove,

- **a** è il vettore in tre dimensioni che rappresenta il primo keypoint;
- **b** è il vettore in tre dimensioni che rappresenta il secondo keypoint.

I keypoint sono selezionati tra i landmark connessi tra loro, seguendo le connessioni come mostrato in Figura 2.11a. Per questo calcolo, abbiamo preso come riferimento il progetto **Sign Language Recognition MediaPipe DTW**[2].

Nel notebook `4_wlasl_mediapipe.ipynb` è possibile trovare il codice relativo alla classificazione con MediaPipe. Innanzitutto, è stata effettuata la separazione dei dati di training e di test; allo scopo sono state sfruttate le funzioni `get_train_videos` e `get_test_videos` della classe di utilità `Launcher`.

[2] https://github.com/gabguerin/Sign-Language-Recognition--MediaPipe-DTW/tree/master

Suddivisione training e test set

La suddivisione dei dati di *training* e *test* segue la struttura definita nel file `WLASL_v0.3.json`; al training set vengono aggiunti anche i video di validation, siccome il classificatore DTW non necessita di un validation set.

I video di *test* sono una semplice lista, mentre i video di *training* sono organizzati in un dizionario, in cui ogni parola è associata a un elenco di video corrispondenti. Questa differenza nasce dalla natura dei dati: i video di test vengono pre-processati in modo da includere direttamente le *feature* estratte con `MediaPipe`, in particolare l'angolo tra i *landmark* delle mani (i punti tra loro connessi). Se lo stesso approccio venisse applicato anche ai video di *training*, il consumo di memoria diventerebbe rapidamente insostenibile, soprattutto con grandi volumi di dati. Per questo motivo, i due insiemi di video sono strutturati in modo diverso.

Data augmentation

Una volta ottenuti i dati, questi vengono passati alla funzione `classify` in cui si può decidere se fare o meno *augmentation*. Nel caso in cui si sia scelto di fare *augmentation*, bisogna semplicemente indicare il numero di trasformazioni che si vogliono applicare. Per ciascun video che si sta considerando, vengono effettuate una combinazione randomica di rotazioni delle mani sugli assi `x`, `y` e `z`. Queste trasformazioni sono basate su quelle descritte nella Sezione 3.4.

Classificazione

A questo punto si confronta ogni video di *test* con tutti i video relativi alla parola che si sta analizzando di *training* con in aggiunta i video aumentati.

Si procede poi, confrontando ciascun video di *test* con tutti i video di *training* relativi a una determinata parola. In particolare, per ogni coppia viene calcolata la distanza DTW e il risultato più piccolo viene aggiunto ai risultati (distanza minima indica maggiore similarità). Il calcolo della distanza DTW segue un algoritmo particolare, in quanto non tutti i segni utilizzano entrambe le mani; per questo motivo, se in entrambi i video non corrispondono, alla distanza viene associato un valore infinito. Terminato questo step, ad ogni video di test sarà associato un insieme di distanze, una per ogni parola. Il risultato della classificazione si ottiene restituendo la parola associata alla distanza minima all'interno dell'insieme.

3.4 Classificazione con rete neurale e MediaPipe

In questa sezione verrà descritta la soluzione proposta su Kaggle: **MuteMotion: WLASL Translation Model**[3].

Preparazione dei dati

La suddivisione dei dati di *training*, *validation* e *test* segue la struttura definita nel file `WLASL_v0.3.json`. L'estrazione dei *keypoints* con MediaPipe è stata effettuata utilizzando `solution.hands`, `solution.pose` e `solution.face_mesh`. Non tutti i *keypoints* estratti vengono utilizzati, ma quelli di posa e faccia vengono filtrati. In particolare, saranno usati tutti e 42 quelli riferiti alle mani, 6 della posa che fanno riferimento alla parte superiore del corpo e 132 per il volto, concentrandosi su occhi, sopracciglia, bocca e contorno della faccia.

Filtraggio delle etichette

È stata utilizzata la libreria `fasttext` e in particolare il modello `cc.en.300.bin`; esso fornisce un vettore di dimensione 300 per ogni parola. I vettori vengono quindi utilizzati per calcolare la similarità coseno tra diverse parole. Sono quindi state scelte 351 etichette simili tra loro (entro una certa soglia) che verranno usate per il processo di classificazione.

Data augmentation

Le 351 etichette selezionate appartengono a un totale di `2770` video di *training*, `748` video di *validation* e `560` di *test*. Per aumentare il numero di video di *training* ad ogni video viene applicata una trasformazione, portando il totale a `5540`. Le trasformazioni che è possibile applicare vengono scelte tra le seguenti:

- **Rotazione sugli assi x, y e z**: viene scelto un angolo tra $\pm[10, 30]^\circ$ e sulla base dell'asse di rotazione viene creata una matrice di rotazione. Tramite la funzione di utilità `rotate` viene applicata la rotazione a tutti i *landmark* dei frame del video. La funzione `rotate` assume come centro dell'immagine il punto $[0.5, 0.5, 0]$, in quanto i *landmark* estratti da MediaPipe utilizzano coordinate relative. È inoltre fondamentale che il

[3] https://www.kaggle.com/code/abd0kamel/mutemotion-wlasl-translation-model/notebook

soggetto si trovi al centro dei frame per garantire un risultato migliore; infatti, sono stati utilizzati i video[4], descritti nella Sezione 2.5.

- **Zoom**: i *landmark* vengono scalati verso l'interno (zoom-in) o verso l'esterno (zoom-out). Viene scelto casualmente un fattore di scala nel range $[0.8, 1.2]$ e viene quindi applicato lo zoom a tutti i *landmark*.
- **Shift**: trasla i *landmark* orizzontalmente e verticalmente. Viene scelto un offset per gli assi `x` e `y` nel range $[-0.2, 0.2]$ e viene applicata la traslazione.
- **Mask**: rimuove un sottoinsieme dei *landmark*. Vengono mascherati circa il 30% dei *landmark* delle mani e il 60% relativi agli altri.
- **Flip orizzontale**: il video viene specchiato orizzontalmente.
- **Speedup**: la sequenza temporale viene velocizzata andando a rimuovere ogni secondo frame, quindi effettivamente raddoppiandone la velocità.

Padding dei video

Siccome le sequenze non hanno tutte la stessa lunghezza, viene applicato un padding a tutti i video; in particolare, i video vengono troncati a 200 frames se hanno lunghezza maggiore, se invece sono più brevi, viene applicato un padding per portarli fino a questa dimensione, andando a creare nuovi *landmark* con valori `x=-100`, `y=-100` e `z=-100`.

Definizione del modello

Il modello realizzato si compone di:

- **Masking** e **Flattening layers** utilizzati per mascherare i valori di padding e poi linearizzare i dati;
- seguono 4 blocchi composti di **Dense**, **BatchNormalization** e **Dropout**, i *layer* densi con 512 neuroni e attivazione `relu`, e i primi due di dropout con 0.3 di rate, mentre gli ultimi due con 0.5;
- infine un *layer* **Dense** con 300 neuroni e attivazione `linear` che fornisce l'output e va a stimare i valori estratti con `fasttext`.

Viene utilizzata la similarità coseno come *loss*, `Adam` come *optimizer*, e viene monitorata la metrica di accuratezza sul validation set durante il training, con *callback* di *early stopping* con pazienza di 50 epoche.

[4] https://www.kaggle.com/datasets/sttaseen/wlasl2000-resized

Risultati

Da un problema di classificazione, è stato riportato il problema a uno di regressione, dato che sono state estratte 300 feature riguardanti ogni parola. La fase di addestramento del modello termina dopo 83 epoche; di conseguenza, il risultato migliore è stato ottenuto alla 33-esima. La valutazione del modello sul test set ottiene un'accuratezza pari al $44,64\%$, un risultato che a prima vista sembra ottimo data la complessità del problema, ma ci sono alcune problematiche con la soluzione proposta che di seguito verranno analizzate.

Problematiche riscontrate

Durante la fase di "filtraggio delle etichette" 3.4 a partire dalle *label* dei video sono stati estratti 300 valori, andando quindi a trasformare il problema da uno di classificazione ad uno di regressione; la rete neurale dovrà stimare i 300 valori invece di assegnare la label corretta ai video.

Durante la "definizione del modello" 3.4 la valutazione dei risultati viene però effettuata utilizzando come metrica l'accuratezza. La metrica di accuratezza è definita come:

```
correct_class = [np.argmax(y_pred) == np.argmax(y) for y in y_true]
correct_percentage = (sum(correct_class) / len(correct_class)) * 100
```

Si tratta di una metrica che punta a valutare i risultati in problemi di classificazione, dove l'output (*y_pred*) contiene una lista di valori decimali, la cui somma è pari a 1, che mostra la probabilità che un dato appartenga a ognuna delle classi. Nel modello descritto precedentemente, l'ultimo *layer* ha come attivazione `linear`, una tipologia di attivazione tipicamente usata per problemi di regressione; la rete neurale viene quindi addestrata per stimare i 300 valori estratti con `fasttext` invece che predire la classe associata al video. Questa discrepanza implica che un risultato sarà considerato corretto se l'indice con valore più alto stimato è lo stesso rispetto a quello predetto, indipendentemente dai valori restituiti in tutti gli altri elementi della lista. Siccome le *label* sono state precedentemente filtrate sulla base della loro similarità, non è improbabile che molte di queste abbiano valori simili.

Supponendo di effettuare una classificazione nella quale l'unico risultato predetto è quello dei 300 valori associati alla *label "give"*, e valutando l'accuratezza della predizione dove i valori corretti sono invece quelli delle rimanenti 350 parole, si ottiene un'accuratezza pari al 42%, un valore molto vicino al $44,64\%$ registrato dal modello. Ciò indica che il modello potrebbe non aver effettivamente appreso una generalizzazione efficace, bensì si starebbe adattando a una distribuzione di dati sbilanciata, in cui numerose parole presentano *embedding* simili.

Valutazione reali performance del modello

Si è deciso di valutare le performance del modello sia in un problema di classificazione sia in uno di regressione, mantenendo inalterato quanto più possibile la soluzione proposta su Kaggle nella Sezione 3.4.

- **Classificazione**: per la classificazione è stato necessario andare a modificare l'ultimo *layer* cambiando il numero di neuroni da 300 a 351, ovvero il numero di classi del problema. Inoltre, è stata cambiata anche la funzione di attivazione da `linear` a `softmax`. La *loss* del modello da `cosine_similarity` a `categorical_crossentropy`. Oltre a questo è stato effettuato *one hot encoding* delle label.

  Il modello sul *test set* ha raggiunto un'`accuracy` pari al 2.61%, in figura viene mostrato l'andamento dell'addestramento:

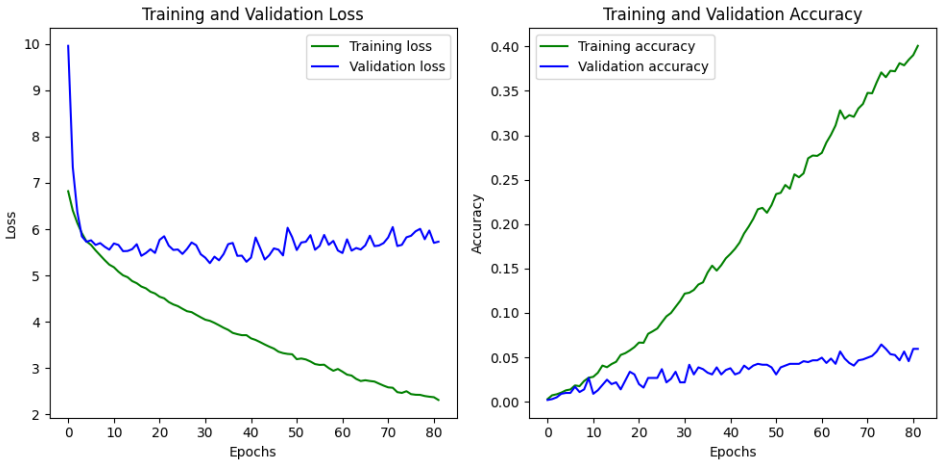


Figura 3.2: Loss e accuratezza nel task di classificazione

- **Regressione**: per il problema di regressione è stato sufficiente modificare la metrica di valutazione del modello da `accuracy` a `mean squared error`.

  Il modello sul *test set* ha raggiunto un `MSE` pari a 75.2, in figura viene mostrato l'andamento dell'addestramento:

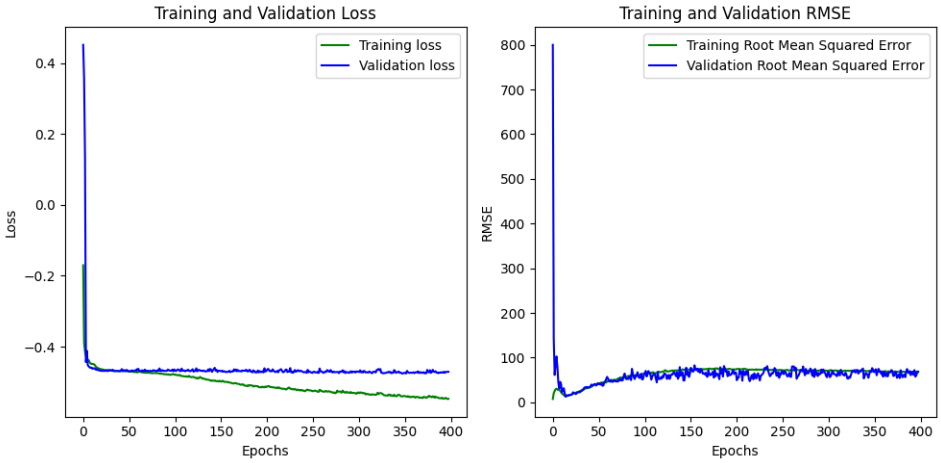


Figura 3.3: Loss e MSE nel task di regressione

In entrambi i casi si può osservare come il modello non sia più migliorato a partire dalla 3ª epoca, rimarcando l'impossibilità di generalizzazione mostrata nel caso di studio.

3.5 Risultati a confronto

In questa sezione verranno confrontati i risultati ottenuti nelle diverse classificazioni.

3.5.1 Metodo tradizionale VS soluzione con MediaPipe

Per cercare di rendere più confrontabili possibili i risultati, i test sono stati svolti con lo stesso numero di parole. Purtroppo sono state usate parole diverse poiché per la classificazione tradizionale è stato usato il dataset con 11982 video mentre la classificazione con MediaPipe è stata basata sul dataset avente 21086 video, con circa 9 mila video in più (video con il padding su entrambi i lati).

Sono stati effettuati diversi test su: 10, 25, 50 e 100 parole. Per MediaPipe sono stati effettuati i test sia senza che con trasformazioni, aggiungendo da uno a tre video in più per ogni video. I risultati sono riportati nella tabella sotto:

Numero Parole	**Accuratezza (%)**		
	Tradizionale	**MediaPipe no aug**	**MediaPipe + aug**
10	32.4%	68.6%	68.6%
25	20.3%	47.5%	48.8%
50	15.7%	39.9%	41.3%
100	10%	33.3%	33.7%

Tabella 3.1: Confronto tra i risultati di classificazione tradizionale e utilizzando MediaPipe senza e con *agumentation*.

Si può osservare come i risultati ottenuti utilizzando MediaPipe abbiano un'accuratezza maggiore rispetto ai risultati ottenuti con metodi tradizionali. Questo è principalmente dovuto al fatto che MediaPipe permette di estrarre features estremamente semplici e discriminanti per il problema in questione. Vi è pochissima differenza rispetto ai risultati con e senza trasformazioni.

Si è voluto, inoltre, per quanto riguarda la classificazione di MediaPipe, estrarre i risultati in forma di Top 5 parole classificate per video di test. Nella seguente tabella sono riportati i risultati:

Numero parole	**Accuratezza (%)**	
	MediaPipe no aug	**MediaPipe + aug**
10	100%	100%
25	88.8%	90%
50	72%	73.4%
100	64.7%	63.6%

Tabella 3.2: Confronto tra i risultati di Top 5 utilizzando MediaPipe senza e con *agumentation*.

3.5.2 Soluzione con MediaPipe VS soluzione proposta su Kaggle

Durante la fase di filtraggio delle *label* (3.4), esse sono state selezionate per la loro similarità; si è osservato nella Sezione 3.4 come questa scelta sia stata fatta per ottenere risultati all'apparenza migliori. Ovviamente sia DTW che una rete neurale riscontrano difficoltà maggiori a distinguere dati simili tra loro; per dimostrare questa teoria vengono mostrati nella tabella sotto i risultati a confronto tra le classificazioni utilizzando DTW e MediaPipe su parole scelte casualmente e parole filtrate.

Numero parole	**Accuratezza (%)**	
	MediaPipe	**MediaPipe + filtered labels**
10	68.6%	43.8%
25	47.5%	41.3%
50	39.9%	35.7%
100	33.3%	31.9%
351	-	26.3%

Tabella 3.3: Confronto tra i risultati utilizzando MediaPipe senza e con *filtered labels*.

Come teorizzato si osserva come le prestazioni diminuiscano analizzando le *label* filtrate, e come all'aumentare del numero di parole esaminate, le prestazioni tendano ad avvicinarsi.

Considerazioni finali sui risultati ottenuti con MediaPipe

Infine, si vogliono effettuare delle considerazioni finali sui risultati ottenuti con MediaPipe.

Alcune delle soluzioni non sono tra loro confrontabili, in particolare, la soluzione in cui è stata effettuata la regressione (3.4) in quanto il risultato ottenuto è misurato tramite *Mean Squared Error*.

Inoltre, il risultato della soluzione proposta su Kaggle non risulta essere significativo per la classificazione, date le problematiche mostrate nella Sezione 3.4.

Andando a confrontare la classificazione fatta col modello descritto precedentemente con la classificazione che sfrutta DTW mostrata nella Sezione 3.3, si può notare che la soluzione proposta in questo progetto ha prestazioni nettamente superiori:

- Top 1: 26.3% rispetto a 2.6%;
- Top 5: 49.5% rispetto a 10.9%.

Questa differenza di prestazioni è dovuta al fatto che il dataset contiene circa 10 video per ogni parola (21 mila video in totale riferiti a 2 mila parole). Questo rende difficile per la rete neurale generalizzare, in quanto i dati non sono sufficienti.

Siccome DTW è una tecnica tradizionale di classificazione (non necessita di addestramento), permette di ottenere risultati migliori per questo dataset. In caso ci fosse stata una quantità superiore di dati, probabilmente le prestazioni sarebbero state più simili.

Capitolo 4

Classificazione Signer

In questo capitolo si entrerà in merito alla classificazione dei volti, in particolare dei *signer*. Trattandosi di un problema diverso rispetto a quello descritto nel Capitolo 3 si è proceduto innanzitutto a ri-analizzare il dataset, in modo da valutare le criticità e gli aspetti che necessitano di ri-elaborazione. In seguito si è effettuato un passo di data augmentation, per poi andare ad effettuare la classificazione sia mediante un classificatore tradizionale con feature hand-crafted, sia tramite una rete neurale convoluzionale.

4.1 Analisi del dataset e pre-elaborazione dei dati

Il primo passo effettuato è l'analisi del dataset e la rielaborazione dei dati.

Caratteristiche dei video

Sono state osservate le seguenti caratteristiche all'interno dei video:

- i *signer* hanno lo stesso sfondo in ogni video in cui appaiono;
- i video sono stati registrati in momenti diversi, infatti i *signer* hanno capigliature e vestiti diversi nei video in cui appaiono.

Per questi motivi si è deciso di rimuovere lo sfondo da tutti i video, sfruttando la libreria `rembg`[1]. Si è inoltre scelto di suddividere i dati in *training*, *validation* e *test* set a livello di video; i video disponibili sono stati assegnati quando possibile (*signer* che appare in almeno 3 video) esclusivamente a uno dei tre insiemi; successivamente sono stati estratti casualmente alcuni frame

[1] https://github.com/danielgatis/rembg

da ognuno dei video assegnati agli insiemi, per comporre i *training*, *validation* e *test set* definitivi.

Sbilanciamento del dataset

Il dataset risulta molto sbilanciato, infatti il *signer* con id `12` ad esempio, appare in un totale di `117374` frames, mentre quello con id `23` solamente in `49`. Per ovviare a questa problematica si è deciso di applicare la *data augmentation* solamente ai frames relativi ai *signer* che appaiono in maniera più sporadica nel dataset; se un signer appare in un numero insufficiente di frames, allora ad esso verrà applicato un numero di *augmentation* per ogni frame. Questo risulta sufficiente per equilibrare il training set.

4.2 Data augmentation

Il processo di data augmentation va ad applicare diverse trasformazioni alle immagini del *training set*; si è deciso di mantenere sempre anche l'immagine originale, in quanto ai frame di alcuni *signer* non sarà effettuato il processo di *data augmentation*. Le trasformazioni sono state realizzate tramite la libreria `imgaug`[1] nel caso della classificazione tradizionale, mentre si è utilizzato un modello sequenziale di `keras`[2] nel modello di classificazione che utilizza una rete neurale. Le trasformazioni effettuate alle immagini sono le seguenti:

- Rotazione nel range $[-\frac{\pi}{4}, \frac{\pi}{4}]$;
- Traslazione orizzontale e verticale;
- Modifica della luminosità;
- Flip orizzontale;
- Applicazione di un filtro di sfocatura gaussiana;
- Addizione di rumore gaussiano;
- Trasformazione elastica;
- Zoom in/out.

Le trasformazioni applicate tramite `keras` sono solamente un sottoinsieme di queste, in quanto non tutte sono disponibili nella libreria.

[2] https://www.tensorflow.org/api_docs/python/tf/keras/Sequential

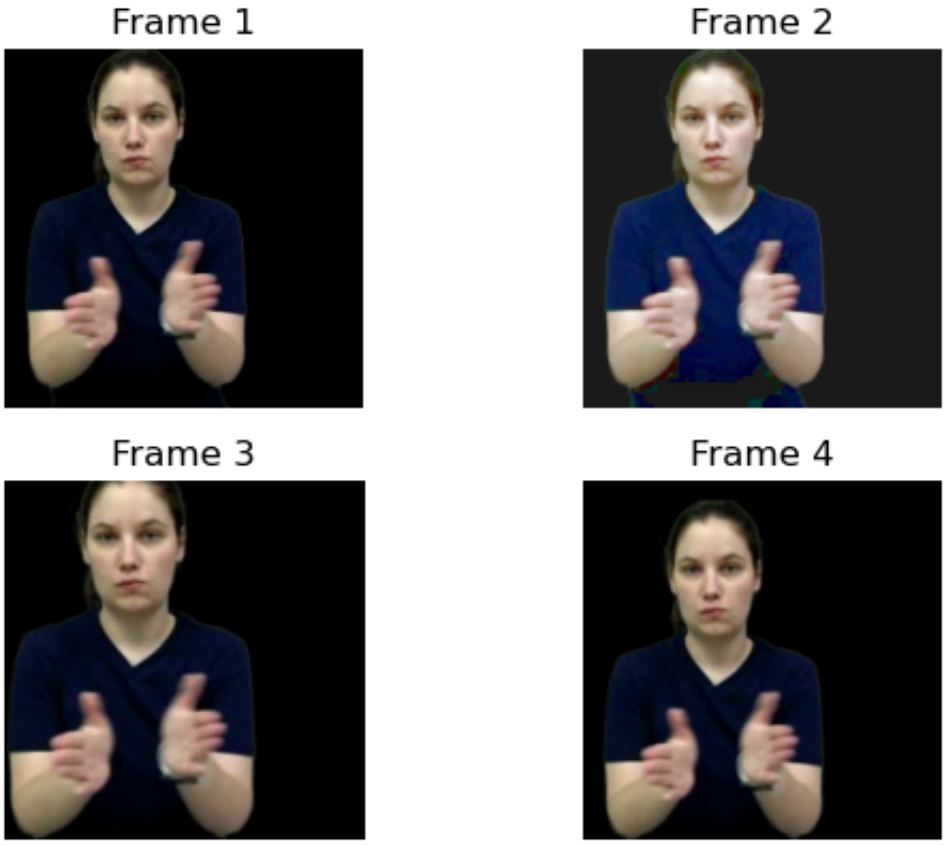


Figura 4.1: Il frame 1 è quello originale, mentre agli altri sono stati applicate le trasformazioni.

4.3 Classificazione tradizionale

Per la classificazione tradizionale si è proceduto utilizzando *XGBoost*[3]. Si è scelto questo classificatore rispetto a *SVC* o *random forest* in quanto *XGBoost* permette di effettuare addestramento incrementale, un punto fondamentale in quanto caricare in memoria RAM l'intero dataset non è possibile.

Suddivisione training e test set

Per la suddivisione nei due insiemi è stata realizzata una classe `FramesSplitter` che riceve in input l'intero *dataset*, la percentuale di video che si vuole utilizzare per il *test* e il numero di frame da estrarre e ritorna sei liste:

[3] https://xgboost.readthedocs.io/en/release_3.0.0/

- Le prime tre relative al *training set*:
  1. La lista dei frame di training, all'interno di una classe ad-hoc che contiene solamente la posizione del frame all'interno della memoria fisica e permette di caricare in memoria il frame tramite il metodo `load_frame`.
  2. La lista contenente, per ogni frame, il numero di passi di augmentation da applicare per bilanciare il *dataset*.
  3. La lista di label contenenti, per ogni frame, l'identificatore del *signer* ripreso nel video.
- Le ultime tre relative al *test set*:
  1. La lista dei frame di test, all'interno della stessa classe.
  2. Una lista contenente tutti 0, in quanto non verrà effettuato alcun passo di augmentation al test set ed utile in quanto facilita il riutilizzo di codice dei dataset.
  3. La lista di label contenenti, per ogni frame, l'identificatore del *signer* ripreso nel video.

Creazione dei dataset di training e test

Sono stati quindi creati due dataset tramite il metodo `create_custom_dataset` della classe `DatasetCreator` che va a creare una lista di `MiniBatch`.

I `MiniBatch` vengono utilizzati in fase di *training* per andare a caricare solamente qualche frame per volta, tramite la funzione `load`. Oltre a caricare i frame da disco, la funzione va ad applicare tutti i passi di *augmentation* richiesti, per infine estrarre le feature *HOG, LBP e color histogram* (come descritto nel Capitolo 2).

Addestramento e valutazione del modello

Si è creato il modello passando i seguenti parametri:

```
xgb_params = {
    "objective":"multi:softmax",
    "num_class":num_classes,
    "eval_metric":"mlogloss",
    "max_depth":6,
    "learning_rate":0.1,
    "n_estimators":100,
    "subsample":0.8,
    "colsample_bytree":0.8,
```

```
    "gamma":0,
    "min_child_weight":1,
    "reg_lambda":1,
    "reg_alpha":0,
    "tree_method":"hist",
}
```

Per poi procedere all'addestramento incrementale tramite:

```
xgb_model = None
for train_batch in tqdm(train_dataset):
    X_batch, y_batch = train_batch.load(shuffle=True)
    dtrain = xgb.DMatrix(X_batch, label=y_batch)
    xgb_model = xgb.train(
        params=xgb_params,
        dtrain=dtrain,
        num_boost_round=10,
        xgb_model=xgb_model
    )
```

Per la valutazione del modello il processo è lo stesso dell'addestramento, senza effettuare lo *shuffle* dei dati all'interno dei `MiniBatch`, in quanto questi sono poi da ricostruire per confrontare i risultati attesi con quelli ottenuti.

4.4 Classificazione con rete neurale

Per la classificazione tramite rete neurale si è proceduto andando a realizzare un semplice modello di rete neurale convoluzionale:

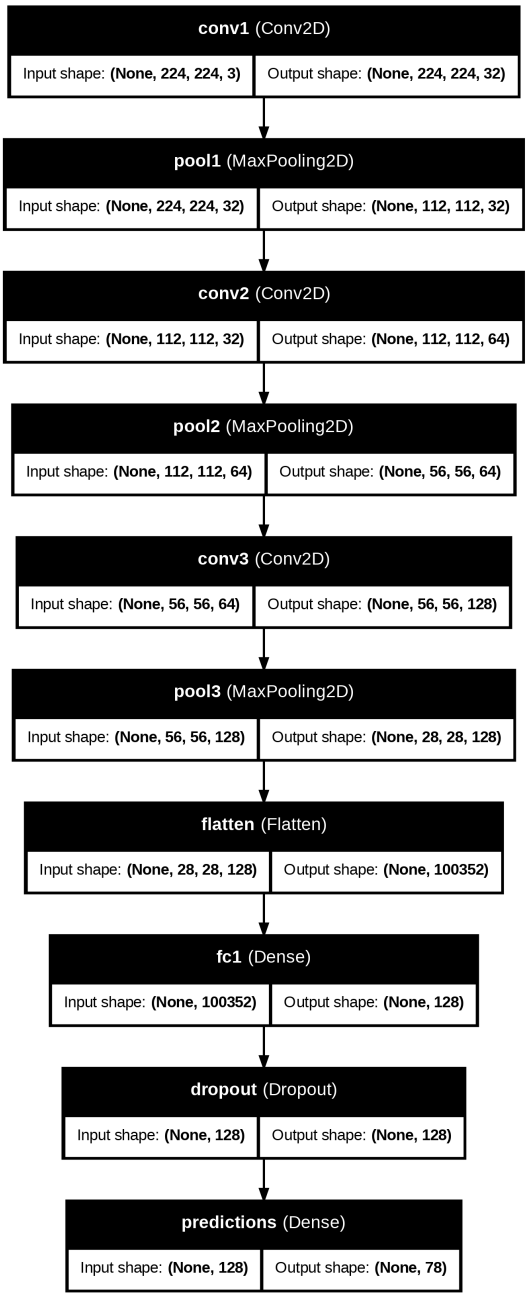


Figura 4.2: Struttura del modello.

La struttura del modello è la seguente:

1. **Input layer** `(input)`: accetta immagini di dimensione $224 \times 224 \times 3$.

2. **Primo blocco convoluzionale**

- `Conv2D (conv1)`: 32 filtri, kernel 3×3, padding `same`, attivazione `ReLU`.
- `MaxPooling2D (pool1)`: finestra di pooling 2×2, stride 2.

3. **Secondo blocco convoluzionale**
   - `Conv2D (conv2)`: 64 filtri, kernel 3×3, padding `same`, attivazione `ReLU`.
   - `MaxPooling2D (pool2)`: finestra di pooling 2×2, stride 2.
4. **Terzo blocco convoluzionale**
   - `Conv2D (conv3)`: 128 filtri, kernel 3×3, padding `same`, attivazione `ReLU`.
   - `MaxPooling2D (pool3)`: finestra di pooling 2×2, stride 2.
5. **Layer di classificazione**
   - `Flatten (flatten)`: appiattisce l'output dei layer precedenti.
   - `Dense (fc1)`: 128 unità, attivazione `ReLU`.
   - `Dropout (dropout)`: tasso di dropout pari a 0.5.
   - `Dense (predictions)`: `num_classes` unità, attivazione `softmax`.

Creazione dei dataset

Per la suddivisione dei dati si è utilizzato lo stesso processo descritto in Sezione 4.3, specificando anche la dimensione del *validation set*, al quale non vengono applicati passaggi di augmentation.

La creazione dei dataset viene delegata al metodo `create_dataset` della classe `DatasetCreator`, che va a realizzare un *dataset* di *TensorFlow*[4]. `DatasetCreator` specifica inoltre come *TensorFlow* dovrà caricare le immagini ed applicare i passi di *augmentation*.

Addestramento e valutazione del modello

Si è quindi proceduto con l'addestramento del modello utilizzando `Adam` come ottimizzatore, 10 steps di *early stopping* ed effettuando 100 epoche, anche se l'addestramento tipicamente termina entro le prime 20.

[4] https://www.tensorflow.org/api_docs/python/tf/data/Dataset

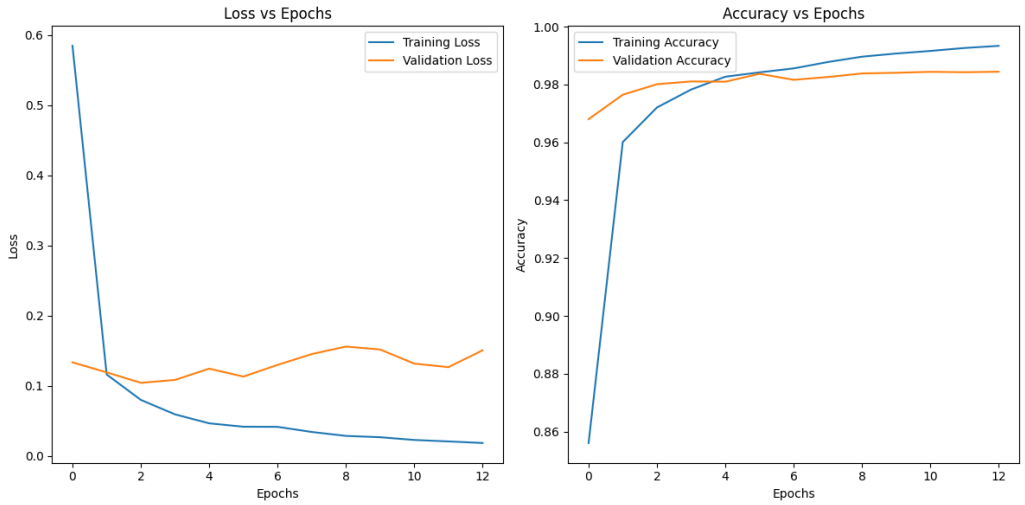


Figura 4.3: Loss e MSE nel task di classificazione.

Si può notare come la *loss* cominci a crescere dalla terza epoca, questo è dovuto al fatto che il modello è riuscito a convergere molto in fretta e ad ogni passo successivo sta andando in *overfitting*.

4.5 Risultati a confronto

Gli addestramenti dei classificatori sono stati effettuati sull'intero dataset, l'unica differenza tra i due è il numero di frames che sono stati estratti per ogni video. Per la classificazione tradizionale sono stati estratti 4 frames per ogni video, mentre per quella con rete neurale 50. La motivazione dietro questa differenza riguarda il tempo di esecuzione, difatti utilizzando *TensorFlow* il caricamento dei dati e l'esecuzione dei passi di *augmentation* vengono svolti sulla *GPU* ed eseguiti mentre la rete sta eseguendo un'epoca, così che al passo successivo i dati siano già disponibili, portando a tempi di esecuzione ridotti. Nonostante la rete neurale sia stata addestrata su un dataset di dimensione maggiore, i risultati sono i seguenti:

Soluzione	**Numero immagini di training**	**Accuratezza (%)**
Tradizionale	33292	94.67%
CNN	185635	97.75%

Tabella 4.1: Confronto tra i risultati utilizzando metodi tradizionali e rete neurale.

Si può osservare come, nonostante ci sia un numero di immagini molto inferiore, per il problema preso in considerazione anche i metodi tradizionali portino a ottimi risultati.

Conclusioni

Questo progetto ci ha permesso di esplorare un ambito di nostro interesse, ma di cui non avevamo conoscenza. Lavorare con i video si è rivelato molto complesso; nonostante ciò, i risultati ottenuti sono stati migliori rispetto alle nostre aspettative.

In futuro, si potrebbero migliorare le performance del classificatore realizzato, in modo che possa essere utilizzato per tradurre istantaneamente il linguaggio dei segni, migliorando l'accessibilità e permettendo una maggiore integrazione anche per le persone mute o con difficoltà di comunicazione. Inoltre, si potrebbero esplorare tecniche più avanzate di estrazione delle caratteristiche e modelli di apprendimento automatico più sofisticati, ampliando il dataset per includere una maggiore varietà di segni, stili di esecuzione e linguaggi diversi.

Bibliografia

[1] Alexander B. Jung, Kentaro Wada, Jon Crall, Satoshi Tanaka, Jake Graving, Christoph Reinders, Sarthak Yadav, Joy Banerjee, Gábor Vecsei, Adam Kraft, Zheng Rui, Jirka Borovec, Christian Vallentin, Semen Zhydenko, Kilian Pfeiffer, Ben Cook, Ismael Fernández, François-Michel De Rainville, Chi-Hung Weng, Abner Ayala-Acevedo, Raphael Meudec, Matias Laporte, et al. imgaug. https://github.com/aleju/imgaug, 2020. Online; accessed 01-Feb-2020.

[2] Romain Tavenard. An introduction to dynamic time warping. https://rtavenar.github.io/blog/dtw.html, 2021.