Trabajo final de máster

HandsTalk

Daniel Fort Franco

ÍNDICE

Resumen

HandsTalk es un programa de inteligencia artificial capaz de reconocer el abecedario del lenguaje de signos inglés y transcribe y recita las palabras.

Elegí este proyecto como trabajo de fin de máster ya que abarca varias áreas del mundo de la ciencia de datos. Además, el desarrollo de **HandsTalk** prometía ser desafiante, permitiéndome consolidar y perfeccionar mis habilidades técnicas y metodológicas en los ámbitos del Deep Learning, análisis y tratamiento de datos y desarrollo con Python.

Introducción

El desarrollo de HandTalk requiere la combinación de diferentes funcionalidades que cooperan entre sí para lograr el objetivo de la aplicación.

Antes de hablar sobre la estructura del proyecto es fundamental hablar de la parte más importante de un proyecto de ciencia de datos: los datos. Para el entrenamiento del modelo se han utilizado **66.376 imágenes** distribuidas en tres conjuntos: 49.769 imágenes para entrenamiento (Train), 9.942 imágenes para validación (Validation) y 6.665 imágenes para pruebas (Test). La mayoría obtenidas del siguiente dataset, y complementadas por imágenes capturadas de mis propias manos.

https://www.kaggle.com/datasets/debashishsau/aslamerican-sign-language-aplhabet-dataset

La estructura básica del programa se divide en 3 capas principales:

1. Captura y tratamiento de la imagen.
2. Predicción del modelo y tratamiento del texto.
3. Actualización de la interfaz de usuario (UI) con el texto generado

Inicialmente, la aplicación obtiene la información capturada por la cámara del dispositivo utilizando OpenCV (cv2) y lo almacena en un array de NumPy. Posteriormente, utiliza el modelo **HandDetector** del paquete **cvzone**, basado en OpenCV y MediaPipe, para detectar la mano dentro de la imagen y extraer la bounding box (punto central, ancho y alto).

Una vez se obtiene una imagen aislada de la mano, aplica los mismos filtros aplicados en los datos en la fase de entrenamiento para proceder con la predicción.

El carácter predicho se incorpora al texto escrito y, finalmente, la interfaz de HandsTalk se actualiza reflejando el nuevo contenido. Este proceso se repite continuamente para capturar y transcribir nuevas letras en tiempo real.

El desarrollo de HandsTalk se ha llevado a cabo principalmente en Visual Studio Code, mientras que los entrenamientos del modelo han sido realizados en Google Colab Pro aprovechando la potencia de sus tarjetas gráficas.

La aplicación está programada en **Python** y utiliza las siguientes librerías:

- **TensorFlow** y **Keras**: para la creación, entrenamiento y despliegue del modelo de Deep Learning.
- **Pandas** y **NumPy**: para el manejo y procesamiento de datos.

- **OpenCV**: para la captura, tratamiento y análisis de imágenes de entrenamiento y en tiempo real y para actualización de la interfaz de usuario.
- **Cvzone**: Para el tratamiento y análisis de las imágenes de entrenamiento y en tiempo real.
- **Scikit-learn**: para evaluación de modelos.
- **gTTS**: para convertir texto a voz.
- **Pygame**: para la gestión de audio.
- **PIL** (Python Imaging Library): para el manejo de imágenes.
- **os**, **shutil** y **time**: para operaciones auxiliares de sistema de archivos y control de flujo.

Metodología

En este apartado se describe la metodología y los pasos seguidos para la construcción del modelo y aplicación.

Construcción modelo

A lo largo del proyecto he repetido en 3 ocasiones las fases de EDA, entrenamiento, evaluación y ajuste, tratando de mejorar cada vez los resultados anteriores.

Fase 1

Para comenzar, descargué un dataset (**ASL_AlphabetDataset**) compuesto por aproximadamente **223.000 imágenes** de **28 clases**, e inicié una primera exploración. Las imágenes estaban organizadas en carpetas correspondientes a cada clase. Las clases incluían todas las letras del abecedario, un símbolo para el espacio, uno para borrar, y otra clase llamada "Nothing" que no utilicé.

El proceso de elección y limpieza de datos comenzó con un script llamado *FilterDataImages.py*, que recorría las imágenes del dataset. Por cada imagen, el script utilizaba el modelo **HandDetector** para localizar la **bounding box** de la mano. A continuación, se aplicaba un cv2.resize() para convertir la imagen a un tamaño de **(300, 300, 3)**, que coincide con las dimensiones de entrada del modelo. Estos nuevos datos se almacenaban en una carpeta llamada **dataset**, guardando un número fijo de imágenes por clase.

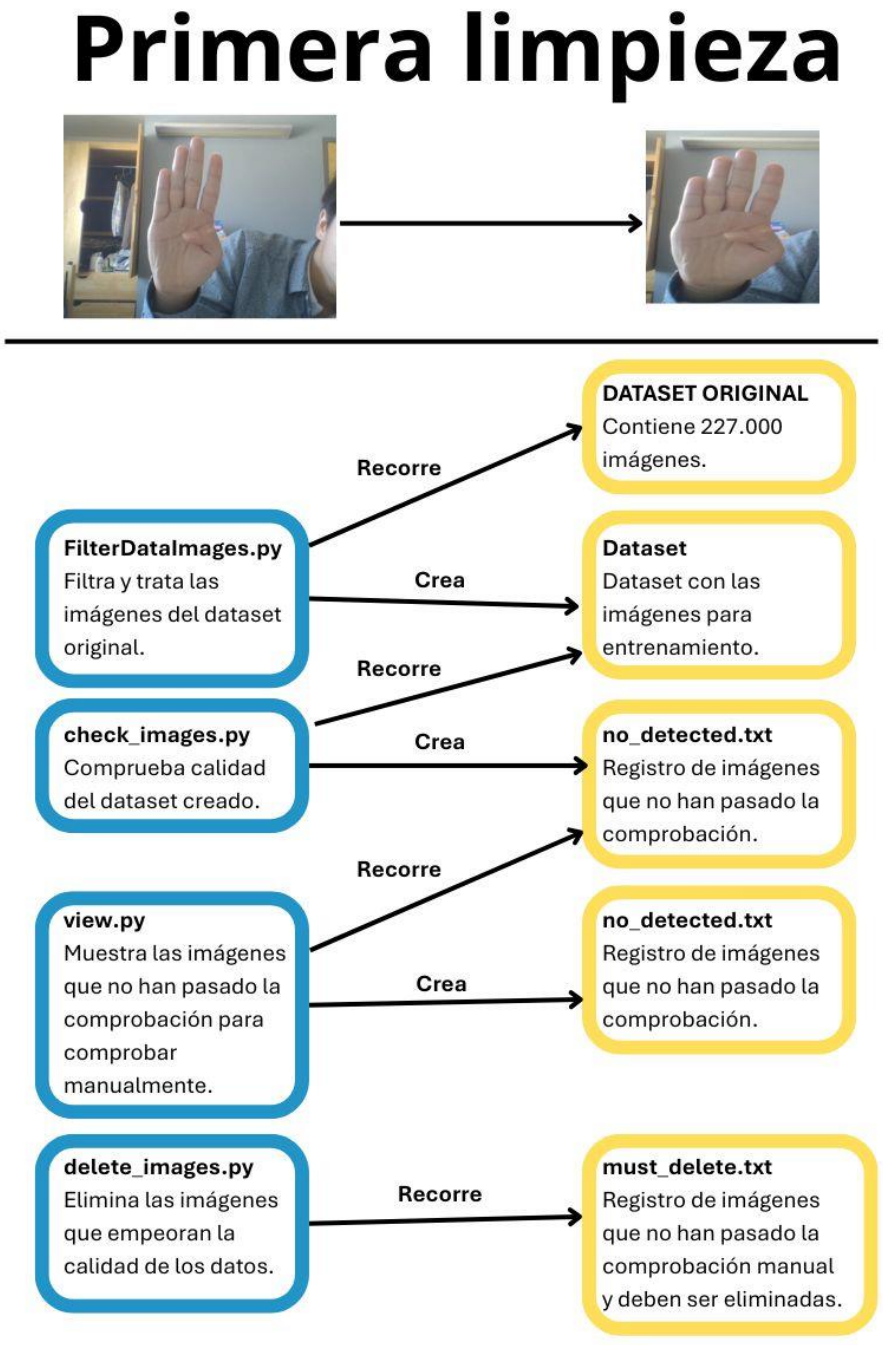
Para comprobar la calidad de los datos, creé el script *check_images.py*, que recorre las imágenes del nuevo dataset e intenta detectar manos. En caso de no detectar ninguna mano en una imagen, la ruta de la imagen se guarda en el archivo *no_detected.txt*.
Durante esta comprobación, observé que había muchas imágenes en las que el **HandDetector** no detectaba correctamente la mano. Por ello, modifiqué *FilterDataImages.py* para añadir un filtro de tres niveles:

1. Primero, analiza y obtiene la imagen aislada de la mano.
2. Luego, intenta detectar manos en la imagen dos veces adicionales.
3. Si supera los tres niveles de verificación, la imagen se guarda en el nuevo dataset; en caso contrario, se descarta y se pasa a la siguiente.

Tras aplicar esta mejora, volví a ejecutar *check_images.py*, obteniendo resultados considerablemente mejores.

Para seguir depurando los datos, creé el script *view.py*, que mostraba las imágenes listadas en *no_detected.txt*. Analicé manualmente estas imágenes: si no se veía claramente una mano o estaba muy borrosa, añadía su ruta a un nuevo archivo

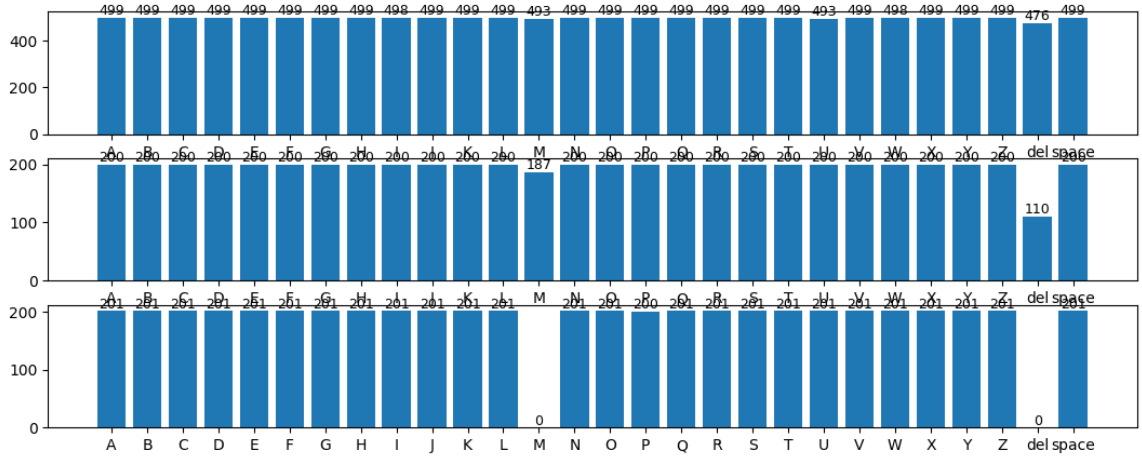
llamado *must_delete.txt*.
Una vez terminada la revisión, utilicé el script *delete_images.py* para eliminar automáticamente todas las imágenes listadas en *must_delete.txt*.



Con los datos listos para entrenamiento, empecé a crear y evaluar los primeros modelos en Google Colab.

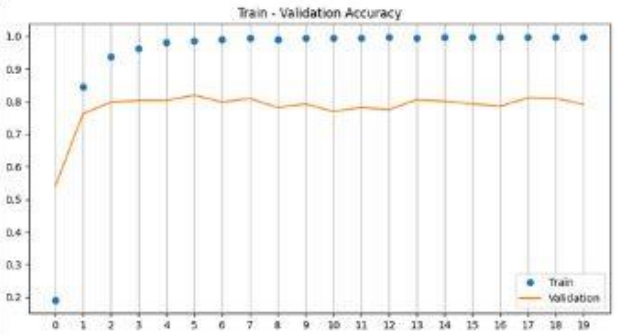
Este primer conjunto de dataset tenía 13.935 imágenes para entrenamiento, **5.497 imágenes** para validación y **5.225 imágenes**. El entrenamiento y las pruebas se realizaron en el siguiente Notebook:

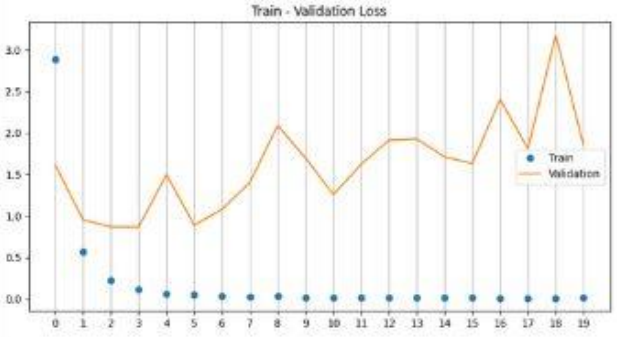
https://colab.research.google.com/drive/1RycCwxJ_3ZH9xKwu4rQ8vvYZfTdaurYw?usp=sharing .



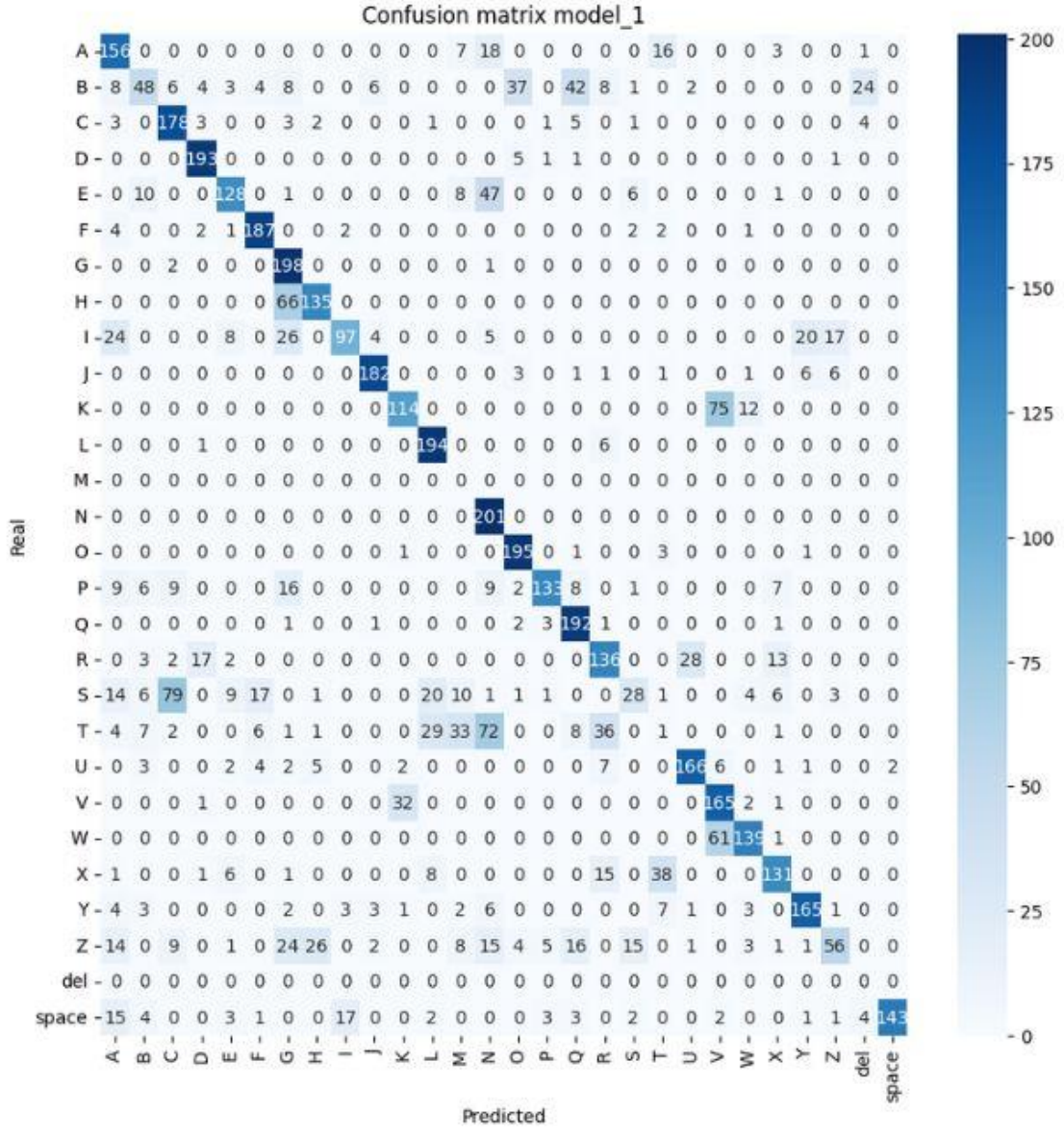
Los modelos eran modelos secuenciales, formados por varias capas convolucionales y una capa clasificadora final. Tras varias pruebas, los resultados aún presentaban muchas áreas de mejora: había numerosos fallos en los datos de test y la predicción en vivo fallaba con frecuencia.
Por ello, decidí **aumentar el dataset** para mejorar el rendimiento del modelo y aplicar técnicas para reducir el **overfitting** en las siguientes pruebas.

Layer (type)	Output Shape	Param #
conv2d_11 (Conv2D)	(None, 298, 298, 16)	448
conv2d_12 (Conv2D)	(None, 296, 296, 16)	2,320
max_pooling2d_5 (MaxPooling2D)	(None, 148, 148, 16)	0
conv2d_13 (Conv2D)	(None, 146, 146, 32)	4,640
conv2d_14 (Conv2D)	(None, 144, 144, 32)	9,248
max_pooling2d_6 (MaxPooling2D)	(None, 72, 72, 32)	0
conv2d_15 (Conv2D)	(None, 70, 70, 64)	18,496
conv2d_16 (Conv2D)	(None, 68, 68, 64)	36,928
max_pooling2d_7 (MaxPooling2D)	(None, 34, 34, 64)	0
conv2d_17 (Conv2D)	(None, 32, 32, 128)	73,856
conv2d_18 (Conv2D)	(None, 30, 30, 128)	147,584
max_pooling2d_8 (MaxPooling2D)	(None, 15, 15, 128)	0
conv2d_19 (Conv2D)	(None, 13, 13, 256)	295,168
conv2d_20 (Conv2D)	(None, 11, 11, 256)	590,080
conv2d_21 (Conv2D)	(None, 9, 9, 256)	590,080
max_pooling2d_9 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten_1 (Flatten)	(None, 4096)	0
dense_2 (Dense)	(None, 512)	2,097,664
dense_3 (Dense)	(None, 28)	14,364

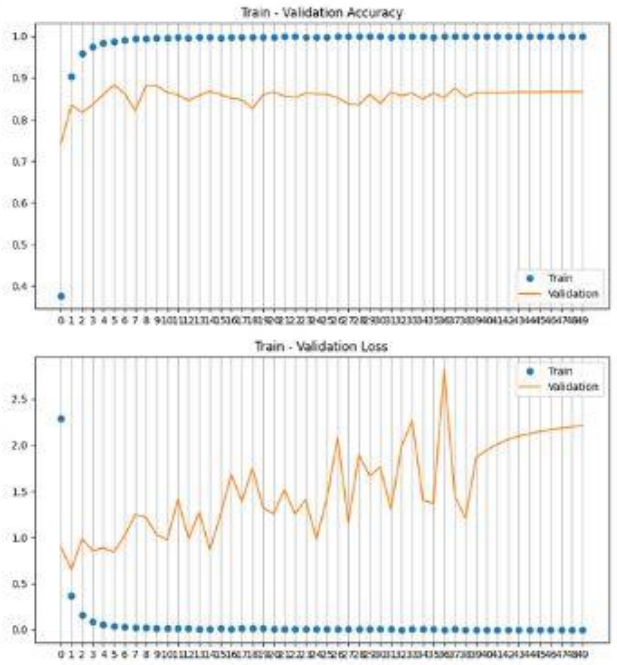




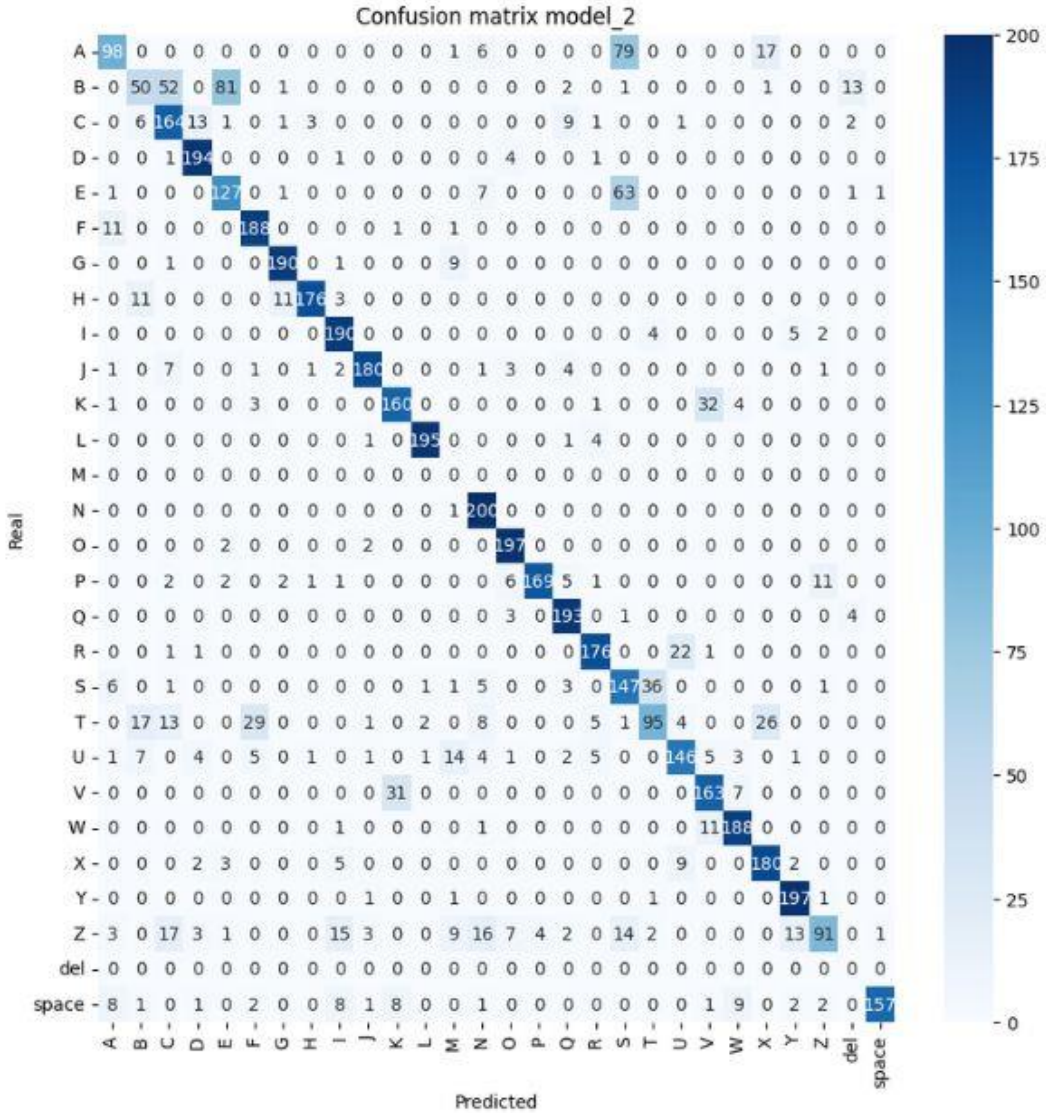

```
Test Accuracy: 0.7007
Max val Accuracy: 0.8195
Max train Accuracy: 0.9978
```



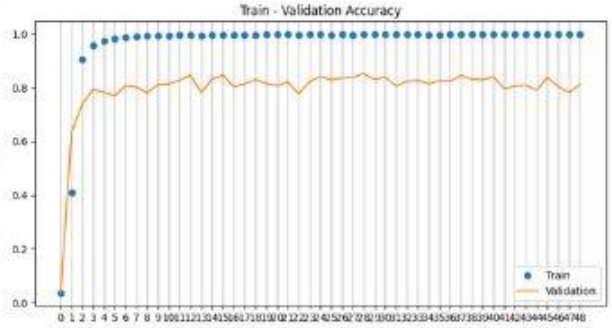
Layer (type)	Output Shape	Param #
conv2d_11 (Conv2D)	(None, 298, 298, 16)	448
conv2d_12 (Conv2D)	(None, 296, 296, 16)	2,320
max_pooling2d_5 (MaxPooling2D)	(None, 148, 148, 16)	0
conv2d_13 (Conv2D)	(None, 146, 146, 32)	4,640
conv2d_14 (Conv2D)	(None, 144, 144, 32)	9,248
max_pooling2d_6 (MaxPooling2D)	(None, 72, 72, 32)	0
conv2d_15 (Conv2D)	(None, 70, 70, 64)	18,496
conv2d_16 (Conv2D)	(None, 68, 68, 64)	36,928
max_pooling2d_7 (MaxPooling2D)	(None, 34, 34, 64)	0
conv2d_17 (Conv2D)	(None, 32, 32, 128)	73,856
conv2d_18 (Conv2D)	(None, 30, 30, 128)	147,584
max_pooling2d_8 (MaxPooling2D)	(None, 15, 15, 128)	0
conv2d_19 (Conv2D)	(None, 13, 13, 256)	295,168
conv2d_20 (Conv2D)	(None, 11, 11, 256)	590,080
conv2d_21 (Conv2D)	(None, 9, 9, 256)	590,080
max_pooling2d_9 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten_1 (Flatten)	(None, 4096)	0
dense_2 (Dense)	(None, 512)	2,097,664
dense_3 (Dense)	(None, 28)	14,364



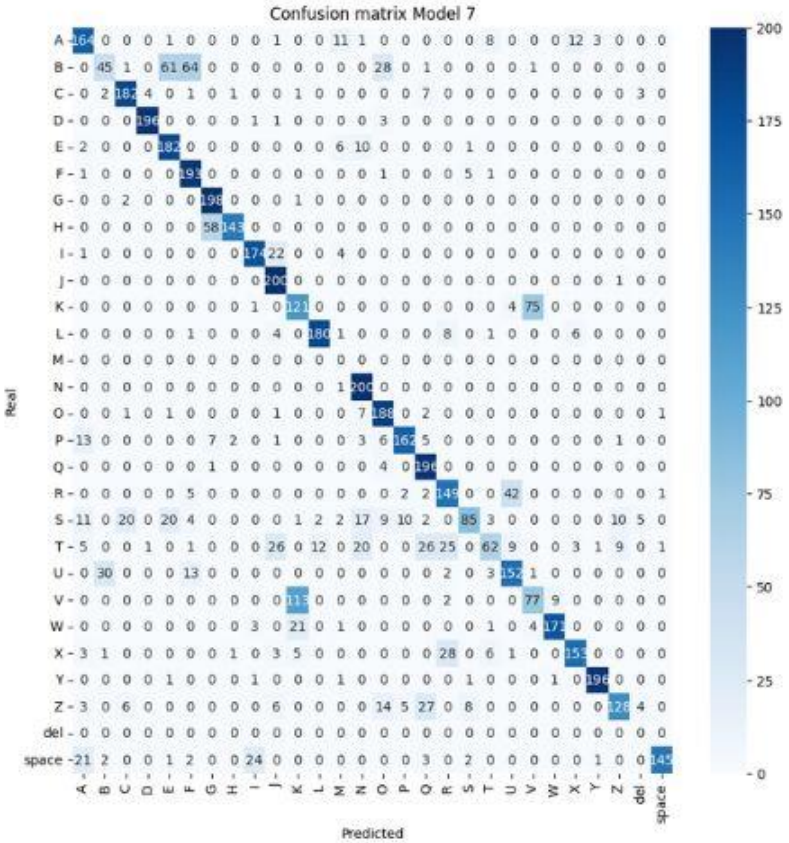

```
Test Accuracy: 0.8059
Max val Accuracy: 0.8830
Max train Accuracy: 1.0000
```



Layer (type)	Output Shape	Param #
conv2d_289 (Conv2D)	(None, 300, 300, 8)	224
max_pooling2d_132 (MaxPooling2D)	(None, 150, 150, 8)	0
conv2d_290 (Conv2D)	(None, 150, 150, 16)	1,168
conv2d_291 (Conv2D)	(None, 150, 150, 16)	2,320
max_pooling2d_133 (MaxPooling2D)	(None, 75, 75, 16)	0
conv2d_292 (Conv2D)	(None, 75, 75, 32)	4,640
conv2d_293 (Conv2D)	(None, 75, 75, 32)	9,248
max_pooling2d_134 (MaxPooling2D)	(None, 37, 37, 32)	0
conv2d_294 (Conv2D)	(None, 37, 37, 64)	18,496
conv2d_295 (Conv2D)	(None, 37, 37, 64)	36,928
conv2d_296 (Conv2D)	(None, 37, 37, 64)	36,928
max_pooling2d_135 (MaxPooling2D)	(None, 18, 18, 64)	0
dropout_9 (Dropout)	(None, 18, 18, 64)	0
conv2d_297 (Conv2D)	(None, 18, 18, 128)	73,856
conv2d_298 (Conv2D)	(None, 18, 18, 128)	147,584
conv2d_299 (Conv2D)	(None, 18, 18, 128)	147,584
max_pooling2d_136 (MaxPooling2D)	(None, 9, 9, 128)	0
dropout_10 (Dropout)	(None, 9, 9, 128)	0
conv2d_300 (Conv2D)	(None, 9, 9, 256)	295,168
max_pooling2d_137 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten_18 (Flatten)	(None, 4096)	0
dense_36 (Dense)	(None, 512)	2,097,664
dense_37 (Dense)	(None, 28)	14,364




```
Test Accuracy: 0.7736
Max val Accuracy: 0.8523
Max train Accuracy: 0.9991
```



Fase 2

En la siguiente fase, empecé repitiendo el proceso de la fase anterior, con una pequeña modificación: el script *FilterDataImages.py* ya no separaba los datos en Train, Validation y Test, sino que simplemente organizaba las imágenes por clases en un nuevo dataset llamado **AumentedData**. Además, aumenté el número de imágenes por clase.

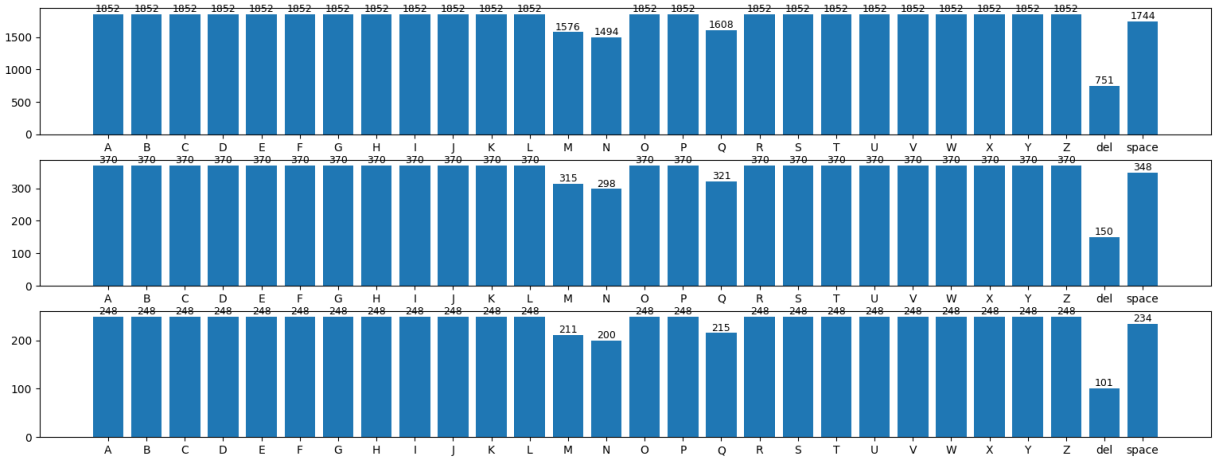
También creé el script *take_hand_pictures.py*. El objetivo de este script era añadir imágenes de mis propias manos, realizando los signos, al dataset. El script capturaba imágenes mediante la webcam, les aplicaba los tres filtros de calidad, y añadía las imágenes validadas a un nuevo dataset llamado **AugmentedData**. En la mayoría de los casos, añadí unas 470 imágenes por clase. Algunas clases, tras ejecutar *delete_images.py*, habían perdido muchas imágenes, así que añadí más imágenes propias para compensar.

Finalmente, antes de volver a Google Colab a reentrenar el modelo, creé el script *split_data.py*, que separaba el nuevo dataset en conjuntos de Train, Validation y Test.

Con el nuevo dataset preparado, procedí a entrenar nuevos modelos en el siguiente Notebook:

https://colab.research.google.com/drive/1EsJZwYyBRLBIW-vfMQ9oyt_5MMLTMpaU?usp=sharing

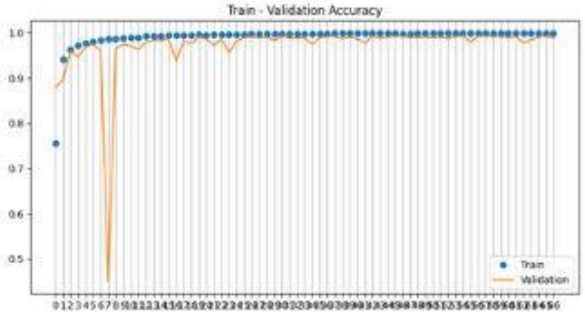
Estos entrenamientos se realizaron utilizando **49.769 imágenes** para entrenamiento, **9.942 imágenes** para validación y **6.665 imágenes** para pruebas.

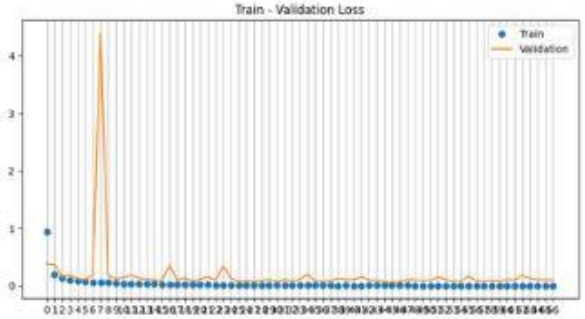


Con los nuevos datos y aplicando técnicas para reducir el overfitting, los resultados sobre el conjunto de prueba mejoraron notablemente. Para controlar el overfitting, añadí varias capas **Dropout** y utilicé **EarlyStopping**.

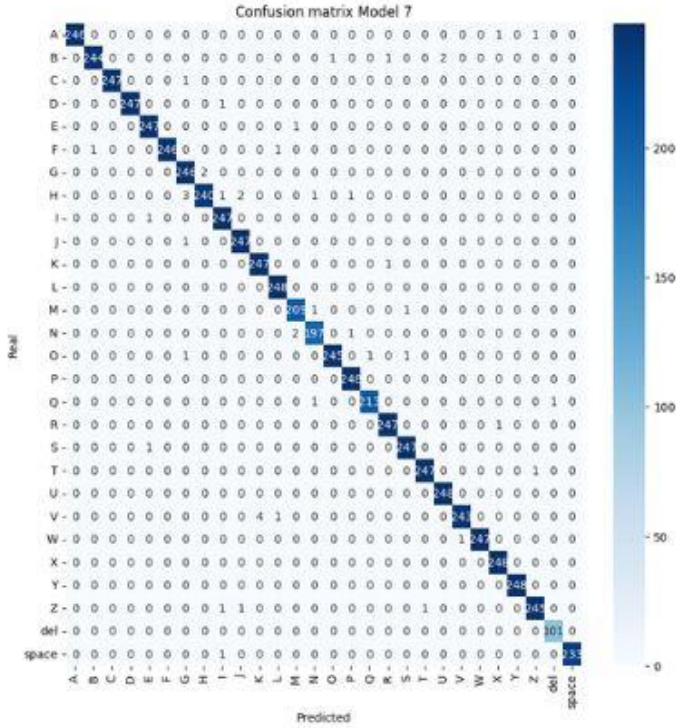
Aun así, al probar el modelo en la aplicación en vivo, seguía cometiendo bastantes errores, por lo que fue necesario volver a ajustar y mejorar los datos.

Layer (type)	Output Shape	Param #
conv2d_14 (Conv2D)	(None, 300, 300, 8)	224
batch_normalization (BatchNormalization)	(None, 300, 300, 8)	32
max_pooling2d_6 (MaxPooling2D)	(None, 150, 150, 8)	0
conv2d_15 (Conv2D)	(None, 150, 150, 16)	1,168
conv2d_16 (Conv2D)	(None, 150, 150, 16)	2,320
batch_normalization_1 (BatchNormalization)	(None, 150, 150, 16)	64
max_pooling2d_7 (MaxPooling2D)	(None, 75, 75, 16)	0
conv2d_17 (Conv2D)	(None, 75, 75, 32)	4,640
conv2d_18 (Conv2D)	(None, 75, 75, 32)	9,248
batch_normalization_2 (BatchNormalization)	(None, 75, 75, 32)	128
max_pooling2d_8 (MaxPooling2D)	(None, 37, 37, 32)	0
conv2d_19 (Conv2D)	(None, 37, 37, 64)	18,496
conv2d_20 (Conv2D)	(None, 37, 37, 64)	36,928
conv2d_21 (Conv2D)	(None, 37, 37, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 37, 37, 64)	256
max_pooling2d_9 (MaxPooling2D)	(None, 18, 18, 64)	0
dropout_2 (Dropout)	(None, 18, 18, 64)	0
conv2d_22 (Conv2D)	(None, 18, 18, 128)	73,856
conv2d_23 (Conv2D)	(None, 18, 18, 128)	147,584
conv2d_24 (Conv2D)	(None, 18, 18, 128)	147,584
batch_normalization_4 (BatchNormalization)	(None, 18, 18, 128)	512
max_pooling2d_10 (MaxPooling2D)	(None, 9, 9, 128)	0
dropout_3 (Dropout)	(None, 9, 9, 128)	0
conv2d_25 (Conv2D)	(None, 9, 9, 256)	295,168
max_pooling2d_11 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten_1 (Flatten)	(None, 4096)	0
dense_2 (Dense)	(None, 512)	2,097,664
dense_3 (Dense)	(None, 28)	14,364

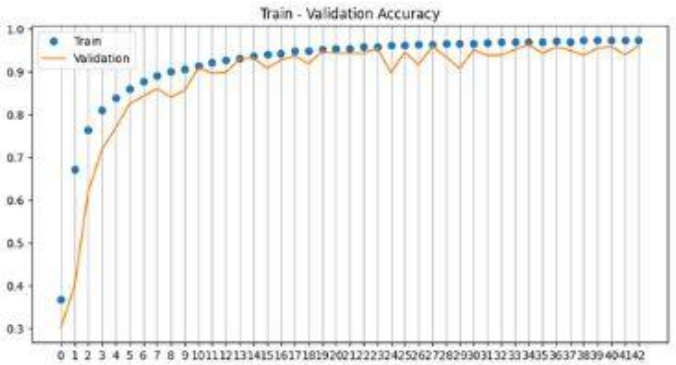


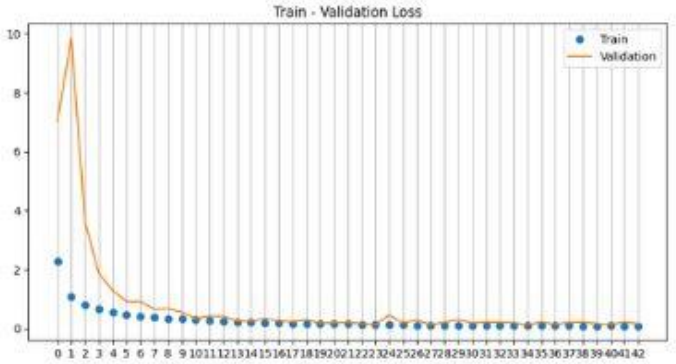



```
Test Accuracy: 0.9935
Max val Accuracy: 0.9934
Max train Accuracy: 0.9996
```

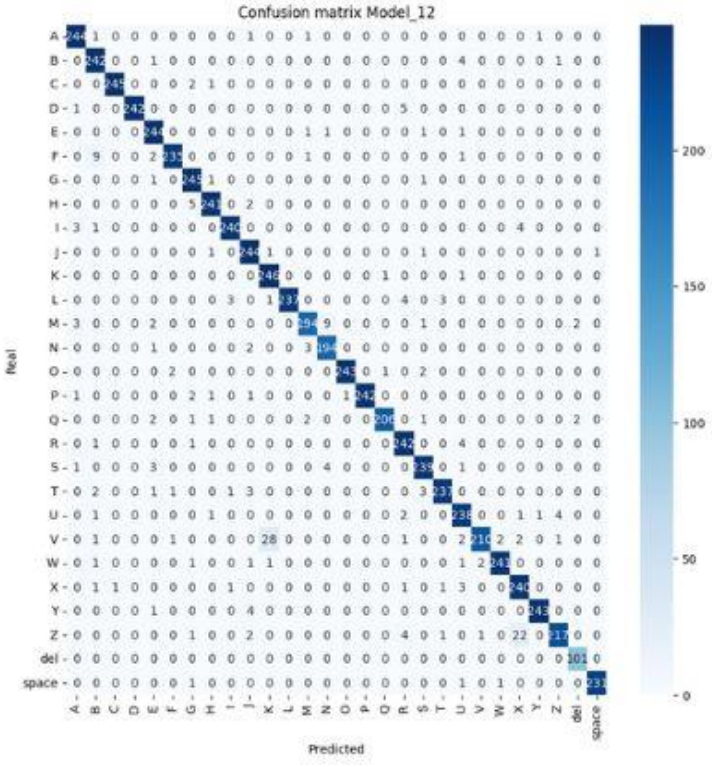


Layer (type)	Output Shape	Param #
separable_conv2d_20 (SeparableConv2D)	(None, 300, 300, 16)	91
batch_normalization_25 (BatchNormalization)	(None, 300, 300, 16)	64
max_pooling2d_26 (MaxPooling2D)	(None, 150, 150, 16)	0
dropout_28 (Dropout)	(None, 150, 150, 16)	0
separable_conv2d_21 (SeparableConv2D)	(None, 150, 150, 16)	416
batch_normalization_26 (BatchNormalization)	(None, 150, 150, 16)	64
max_pooling2d_27 (MaxPooling2D)	(None, 75, 75, 16)	0
dropout_29 (Dropout)	(None, 75, 75, 16)	0
separable_conv2d_22 (SeparableConv2D)	(None, 75, 75, 32)	688
batch_normalization_27 (BatchNormalization)	(None, 75, 75, 32)	128
max_pooling2d_28 (MaxPooling2D)	(None, 37, 37, 32)	0
dropout_30 (Dropout)	(None, 37, 37, 32)	0
separable_conv2d_23 (SeparableConv2D)	(None, 37, 37, 64)	2,400
batch_normalization_28 (BatchNormalization)	(None, 37, 37, 64)	256
max_pooling2d_29 (MaxPooling2D)	(None, 18, 18, 64)	0
dropout_31 (Dropout)	(None, 18, 18, 64)	0
separable_conv2d_24 (SeparableConv2D)	(None, 18, 18, 128)	8,896
batch_normalization_29 (BatchNormalization)	(None, 18, 18, 128)	512
max_pooling2d_30 (MaxPooling2D)	(None, 9, 9, 128)	0
dropout_32 (Dropout)	(None, 9, 9, 128)	0
separable_conv2d_25 (SeparableConv2D)	(None, 9, 9, 256)	34,176
batch_normalization_30 (BatchNormalization)	(None, 9, 9, 256)	1,024
max_pooling2d_31 (MaxPooling2D)	(None, 4, 4, 256)	0
dropout_33 (Dropout)	(None, 4, 4, 256)	0
flatten_5 (Flatten)	(None, 4096)	0
dense_10 (Dense)	(None, 512)	2,097,664
dropout_34 (Dropout)	(None, 512)	0
dense_11 (Dense)	(None, 28)	14,364






```
Test Accuracy: 0.9631
Max val Accuracy: 0.9638
Max train Accuracy: 0.9740
```



Fase 3

Aunque los resultados sobre el conjunto de prueba eran buenos, la aplicación seguía sin clasificar correctamente en algunos casos. Por ello, decidí aplicar un cambio importante a los datos.

Creé un tercer dataset, basado en el anterior, llamado **ProcessedData**, al que apliqué varios filtros de OpenCV a todas las imágenes. El objetivo era que el modelo se centrara más en los rasgos relevantes de la imagen, eliminando la mayor cantidad posible de ruido. Los filtros aplicados fueron los siguientes:

- **cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)**: Convertía la imagen a escala de grises.
- **cv2.medianBlur(img, 5)**: Eliminaba el ruido tipo "sal y pimienta", manteniendo los bordes bien definidos.
- **cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)**: Detectaba las formas principales y eliminaba el resto de la información, dejando en la imagen solo los contornos.

Este proceso lo realicé en el siguiente Notebook: https://colab.research.google.com/drive/1XEPfICGA-bs8K8L5hDX2Ujo4U7PlUeg4?usp=sharing

Una vez listo el nuevo dataset, volví a Google Colab para reentrenar los modelos, en el siguiente Notebook:

https://colab.research.google.com/drive/1SmjwYDlR6ALQCP2X7bJ0y91O2BkyvEIq?usp=sharing

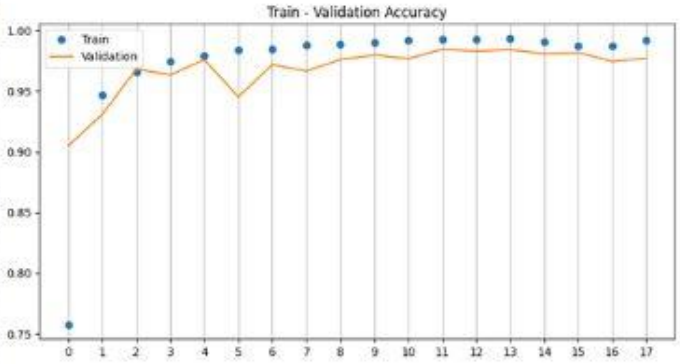
Esta vez, los resultados sobre los datos de prueba fueron igualmente prometedores, e incluso ligeramente superiores a los anteriores. Además, el rendimiento en la aplicación en vivo mejoró considerablemente: el modelo clasificaba correctamente y de forma mucho más robusta.

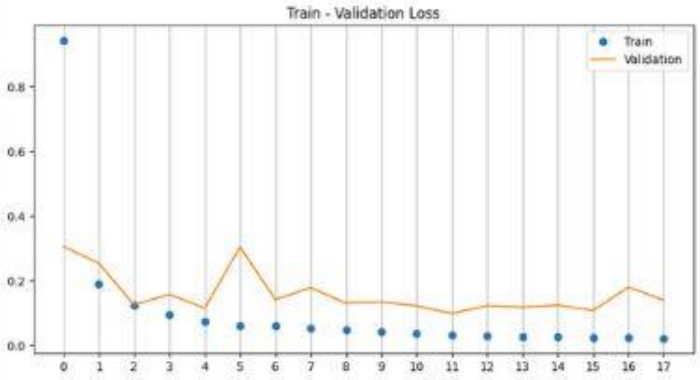
El modelo definitivo utilizado en la aplicación es el siguiente:

https://drive.google.com/file/d/1yCBwKlpz1v-WH8VRKBvWPo_9_So17u7R/view?usp=sharing

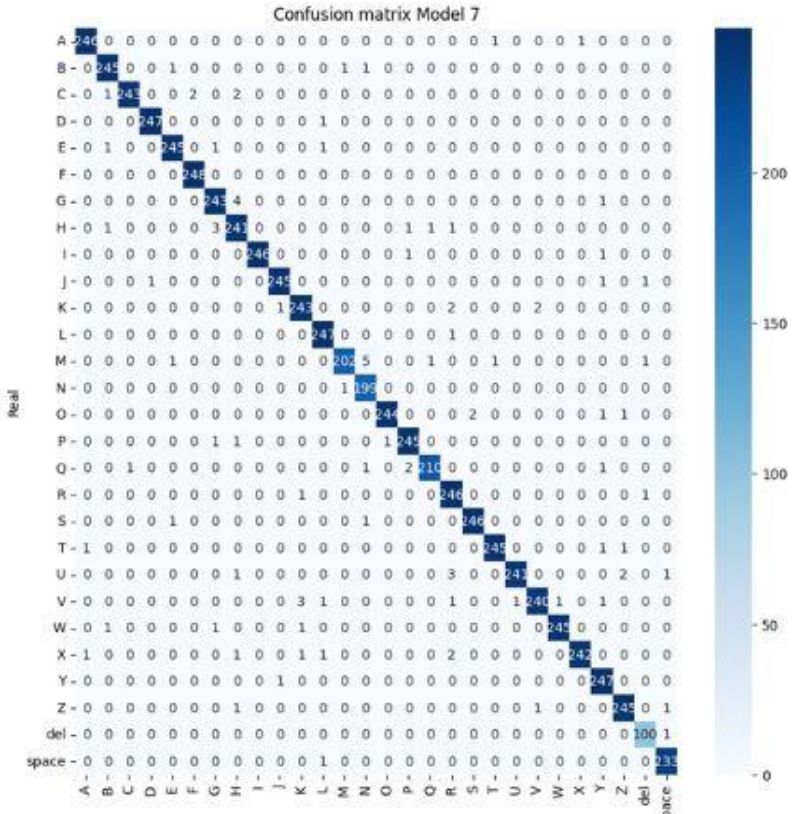
El modelo estaba listo y al hacer las pruebas en script *LiveHandDetector.py* los resultados eran buenos me puse a desarrollar el programa en Python para agregar las funcionalizades necesarias para seguir consiguiendo mi objetivo.

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 300, 300, 8)	80
batch_normalization_11 (BatchNormalization)	(None, 300, 300, 8)	32
max_pooling2d_12 (MaxPooling2D)	(None, 150, 150, 8)	0
conv2d_13 (Conv2D)	(None, 150, 150, 16)	1,168
conv2d_14 (Conv2D)	(None, 150, 150, 16)	2,320
batch_normalization_12 (BatchNormalization)	(None, 150, 150, 16)	64
max_pooling2d_13 (MaxPooling2D)	(None, 75, 75, 16)	0
conv2d_15 (Conv2D)	(None, 75, 75, 32)	4,640
conv2d_16 (Conv2D)	(None, 75, 75, 32)	9,248
batch_normalization_13 (BatchNormalization)	(None, 75, 75, 32)	128
max_pooling2d_14 (MaxPooling2D)	(None, 37, 37, 32)	0
conv2d_17 (Conv2D)	(None, 37, 37, 64)	18,496
conv2d_18 (Conv2D)	(None, 37, 37, 64)	36,928
conv2d_19 (Conv2D)	(None, 37, 37, 64)	36,928
batch_normalization_14 (BatchNormalization)	(None, 37, 37, 64)	256
max_pooling2d_15 (MaxPooling2D)	(None, 18, 18, 64)	0
dropout_9 (Dropout)	(None, 18, 18, 64)	0
conv2d_20 (Conv2D)	(None, 18, 18, 128)	73,856
conv2d_21 (Conv2D)	(None, 18, 18, 128)	147,584
conv2d_22 (Conv2D)	(None, 18, 18, 128)	147,584
batch_normalization_15 (BatchNormalization)	(None, 18, 18, 128)	512
max_pooling2d_16 (MaxPooling2D)	(None, 9, 9, 128)	0
dropout_10 (Dropout)	(None, 9, 9, 128)	0
conv2d_23 (Conv2D)	(None, 9, 9, 256)	295,168
max_pooling2d_17 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten_2 (Flatten)	(None, 4096)	0
dense_4 (Dense)	(None, 512)	2,097,664
dense_5 (Dense)	(None, 28)	14,364






```
Test Accuracy: 0.9854
Max val Accuracy: 0.9845
Max train Accuracy: 0.9936
```



Desarrollo aplicación

La aplicación está desarrollada en **Python** y tiene varios scripts con diferentes funcionalidades.

Para hacer el flujo principal me inspiré en cómo funciona el motor de videojuegos **Unity**. Primero ejecuta una función **start()** al crear los objetos. A continuación, el programa entra en un bucle infinito donde, en cada iteración, se llama a la función update() de cada objeto para actualizar sus datos, y finalmente se llama a la función draw() para renderizar todos los elementos en la imagen final.

La organización de carpetas y archivos de la aplicación es la siguiente:

- AAP
    - Files
        - Models: Contiene el modelo utilizado para la aplicación.
        - Sounds: Contiene los archivos *.wav*.
    - Ui
        - Layout.py: Se define la clase base de Layout.
        - Elements: Contiene elementos básicos.
            - Circle.py: Se define la clase base que se renderiza como círculo.
            - Rectangle.py: Se define la clase base que se renderiza como rectángulo.
            - Text_field.py: Se define la clase base que se renderiza como texto.
        - Custom_elements: Se definen clases que heredan de *Elements* y añaden funcionalidades específicas.
            - Animated_circle.py: Se define la clase encargada de mostrar un círculo animado que marca los tiempos en los que el programa permite escribir. Además, controla cuando el modelo puede predecir y cuando no.
            - Running_layout.py: Se define la clase que hereda de Layout. Es la encargada de almacenar todos los elementos que ser verán en la interfaz de usuario y de actualizar sus estados en orden específico.
            - Text_box.py: Se define la clase que hereda de Rectangle y contiene los textos que se van actualizando cada frame.
            - Writed_text_field.py: Se define la clase que se encarga de seleccionar el texto que se mostrará por pantalla, además de gestionar el cursor.

- Utils
  - App_manager.py: gestiona las modificaciones del texto y emite los sonidos correspondientes. Almacena el texto que se ha escrito y el fragmento que debe mostrarse por pantalla.
  - Image_flow.py: gestiona las modificaciones aplicadas a la imagen que se muestra al final de cada frame.
  - Sound_manager.py: contiene las funciones que controlan la reproducción de sonidos.
  - Time_control.py: Define la clase que calcula calculas los FPS establecidos. También calcula el tiempo transcurrido entre frames (deltatime) que necesitan algunas clases.
- Visión
  - Clasiffier.py: Se define la clase clasificadora y todas sus funciones necesarias.
  - Web_cam_reader.py: Se define la clase que almacena la imagen que captura la webcam y llama a clasificar al modelo en el momento adecuado.
- Main.py: Ejecuta el bucle principal del proyecto.

La aplicación tiene varias funcionalidades importantes en las que forman parte varias clases. A continuación, hablaré sobre todas ellas en detalle, antes hablaremos sobre el flujo principal de la aplicación.

Flujo aplicación

En *main.py* se crea un bucle infinito en el que se gestiona todo el flujo. Antes de ello crean tres instancias necesarias.

- FrameTimer() →**time_controller**
- WebCamReader() →**wc**
- RunningLayout() →**lout**

Cada iteración del bucle realiza las siguientes acciones:

1. **Control de tiempo**: Se comprueba si ha pasado el tiempo suficiente entre frames para ejecutar de nuevo el código. Esto mantiene un ritmo constante, independientemente del dispositivo que ejecute el programa.
2. **Captura y procesamiento de imagen**: Se llama a update() de **wc**, que almacena la imagen capturada por la webcam y realiza predicciones si es necesario.
3. **Actualización de la UI**: Se llama a update() de **lout**, que a su vez llama a la función update() de todos sus componentes.

4. **Renderizado**: Se llama a draw() de **lout**, que modifica la imagen final añadiendo los componentes visuales.
5. **Visualización**: Se muestra la imagen final y se reinicia el bucle.

Para lograr esta estructura, RunningLayout contiene una lista de elementos y, en su función update(), recorre esta lista llamando a la función update() de cada elemento. Lo mismo ocurre con las funciones start() y draw().

Funcionalidades

Muestra de círculo que marca los tiempos

Para mostrar el círculo por pantalla creé la clase *AnimatedCircle()* y la añadí una instancia a la lista de elementos de *RunningLayout()*.

En cada frame, esta clase suma a un contador el deltatime multiplicado por una velocidad de animación. Pasando el valor del contador a una función **Sin(x)** marcando unos límites máximos y mínimos, modifiqué el radio del círculo para obtener el efecto de vaivén llevando al círculo de grande a pequeño.

Para mejorar la sensación de movimiento modifiqué la variable **thickness** en base al mismo contador, pero, esta vez usé la función **Cos(x)**. Esto consigue un efecto de grosor desfasado respecto al radio.

Además, cuando el círculo está completamente cerrado modifica el estado de *app_manager.py* y permite o bloquea la escritura, además de cambiar el color del círculo para mejorar la comprensibilidad.

Control de la escritura

El control de la escritura lo maneja la clase *WebCamReader()*. Esta clase contiene una instancia del clasificador y del detector de manos.

Cada frame, esta clase almacena la imagen capturada por la cámara. A continuación, intenta detectar manos, si detecta alguna mano aplica todas las transformaciones aplicadas con los datos de entrenamiento:

1. Obtener la bounding box de la imagen.
2. Redimensionar imagen.
3. Convertir imagen a escala de grises
4. Aplicar desenfoque
5. Aplicar limites adaptativos
6. Normalizar imagen
7. Aplicar np.expand_dims() para que el modelo la pueda leer.

Una vez tiene la imagen lista se manda al clasificador para que devuelva el carácter detectado. Una vez tiene el resultado muestra en una esquina de la imagen la imagen detectada y un texto con la predicción.

Mientras el círculo sea rojo (el modelo puede predecir) el clasificador guardará en una lista todas las predicciones que ha realizado en ese periodo de tiempo.

Con los dos booleanos *app_manager.get_is_able_to_write()* y el booleano de *WebCamReader* **self.iswritting** la clase sabe cuándo es el último frame que puede escribir, en ese momento se llama la función **finish_prediction()** del clasificador, quien devolverá el valor más repetido en las últimas predicciones y vaciará el registro. Cuando *WebCamReader* recibe esta última predicción, llama a la función **add_update_to_writed_text** de *app_manager.py* pasando la predicción como valor.

En esta función pueden pasar 3 cosas:

- Recibe un **carácter**: Se añade a la cadena de texto, se reproduce un sonido de adición y, si la última palabra supera 20 caracteres, se añade un espacio automático.
- Recibe un **espacio**: Añade un espacio a la cadena de texto, reproduce sonido del espacio y reproduce por voz la última palabra.
- Recibe **borrar**: Se elimina el último carácter y se reproduce un sonido de borrado.

Muestra de texto escrito

Esta función la gestiona una instancia de la clase *WritedTextField* creada dentro de *TextBlock*.

En *writed_text_field.py* se define una clase *PartialText()*. Esta hereda de *TextField()* y representa cada línea que se vaya a escribir a la pantalla. Además, contiene un número máximo de caracteres que se puede almacenar y una función **add_word** que devuelve el texto actualizado o **None** en caso de que la palabra no quepa.

WritedTextField() contiene dos instancias de esta clase (**t1,t2**).

Cada vez que se llama a update de *WritedTextField()* se llama a su función **split_text().** Inicialmente, divide el texto a escribir y lo convierte en una lista de palabras haciendo un **str.split(' ')** por espacios.

Recorre esta lista y en cada interacción llama a la función **add_word()** de t1 para ir rellenando su texto. Si en algún momento le devuelve **None** significa que t1 está lleno, entonces, continúa rellenando t2.

Si en algún momento t2 está lleno, el programa modifica el texto a mostrar por pantalla y le quita todo lo que tenía t1, dejando solo la parte final y vuelve a ejecutar

split_text() recursivamente. Como resultado, t1 almacena el contenido que almacenaba t2 y t2 contiene la palabra que no sobrepasaba el límite, manteniendo así siempre las dos últimas líneas escritas por el usuario.

En su función draw llama a cv2.putText y añade a la imagen el contenido de t1 y de t2, el segundo unos píxeles más abajo que se definen en el constructor.

Cursor

WritedTextField() controla la visibilidad del cursor.

Cada update suma a un contador el tiempo transcurrido entre frames, **deltatime**, y comprueba si es mayor a un tiempo preestablecido para hacer el parpadeo. En caso de que lo supere invierte la visibilidad del cursor y reiniciará el contador. Antes de dibujar el texto, decide en qué línea (t1 o t2) debe añadirse el cursor, según si hay texto en ambas.

Reproducción de sonido

App_manager.py contiene una instancia de *SoundPlayer()* y en cada cambio de texto ejecuta los sonidos.

SoundPlayer() tiene 3 funciones:

- play_char_added_sound(): Emite sonido de carácter añadido mediante la librería pygame.
- play_char_deleted_sound(): Emite sonido de carácter borrado mediante la librería pygame.
- play_space_added_sound(): Emite sonido de espacio añadido mediante la librería pygame.
- play_text_sound(): Recibe por parámetros un string que convertirá a un archivo .wav mediante la librería gtts y lo reproducirá mediante pygame.

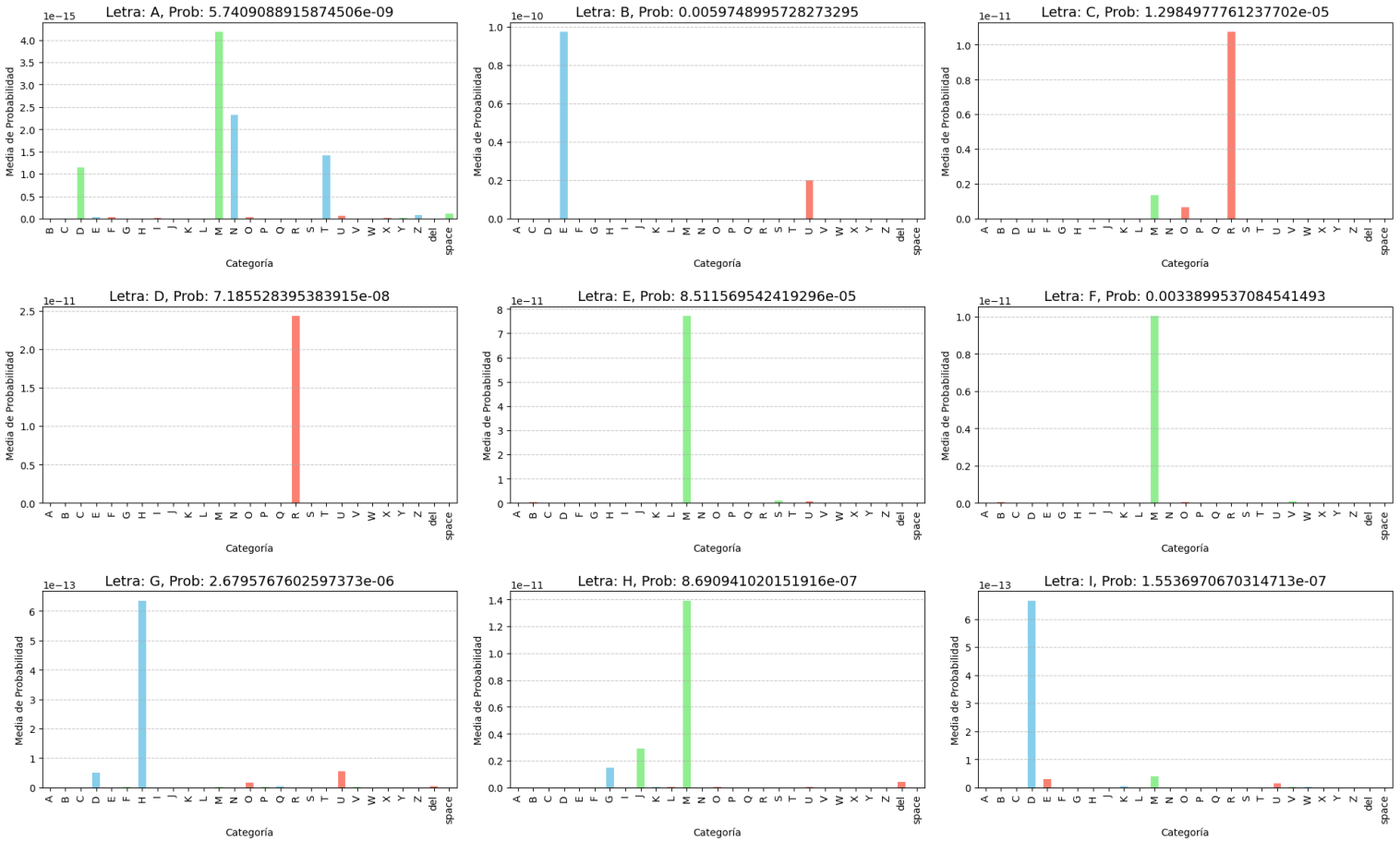
En conclusión, el flujo basado en las funciones start(), update() y draw() permite mantener una ejecución estable en cuanto a velocidad de frames, además de favorecer una arquitectura clara y modular. Cada clase cumple una responsabilidad específica, siguiendo principios sólidos de programación orientada a objetos, lo que facilita tanto el mantenimiento como la escalabilidad del proyecto.

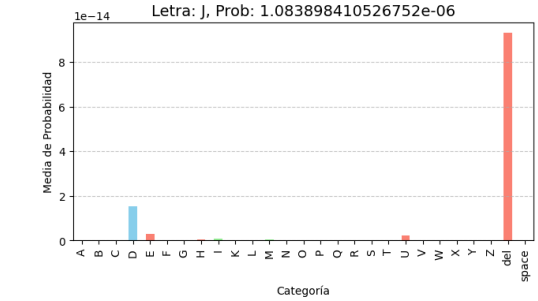
Resultados finales

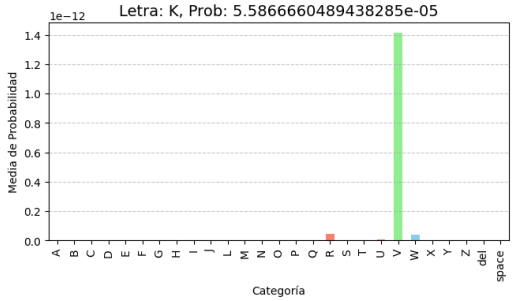
El resultado del proyecto es un programa funcional que permite al usuario escribir mediante el uso de signos, lo que podría ser útil en diversas situaciones.

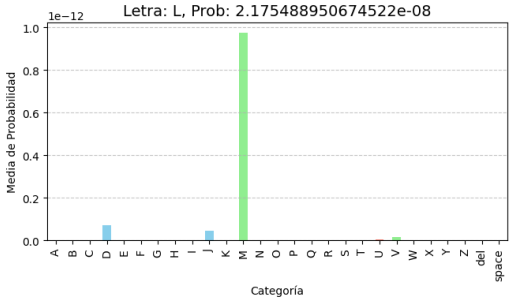
Lo hace posible la combinación de una aplicación estructurada y un modelo clasificador entrenado con buena precisión.

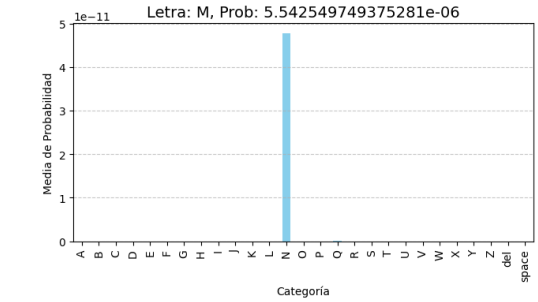
Hice un estudio para comprobar la probabilidad de confusión entre clases y estos son los resultados:

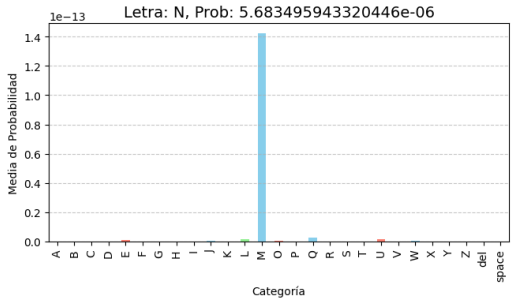


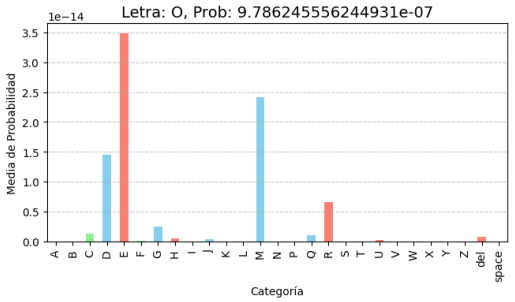

Letra: J, Prob: 1.083898410526752e-06
1e−14
Media de Probabilidad
Categoría

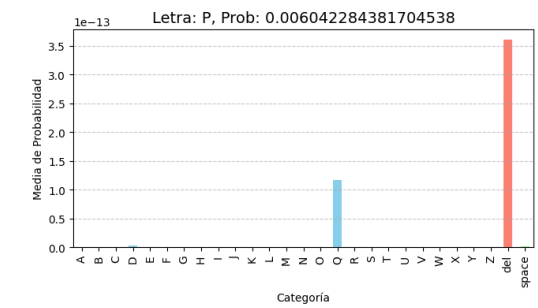

Letra: K, Prob: 5.5866660489438285e-05
1e−12
Media de Probabilidad
Categoría

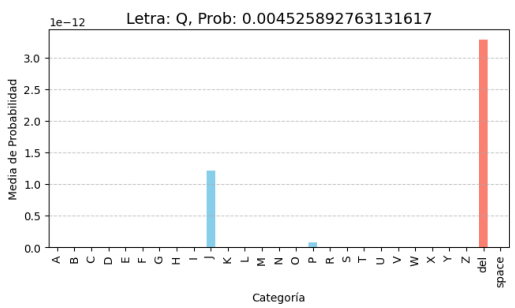

Letra: L, Prob: 2.175488950674522e-08
1e−12
Media de Probabilidad
Categoría

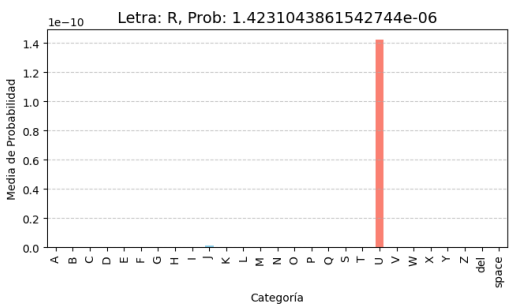

Letra: M, Prob: 5.542549749375281e-06
1e−11
Media de Probabilidad
Categoría

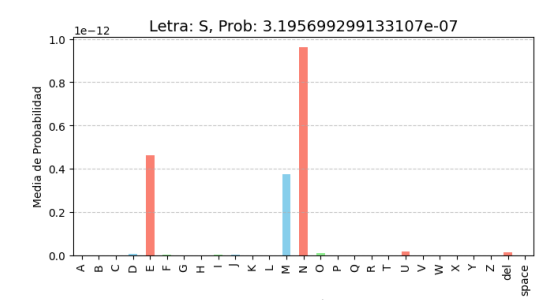

Letra: N, Prob: 5.683495943320446e-06
1e−13
Media de Probabilidad
Categoría

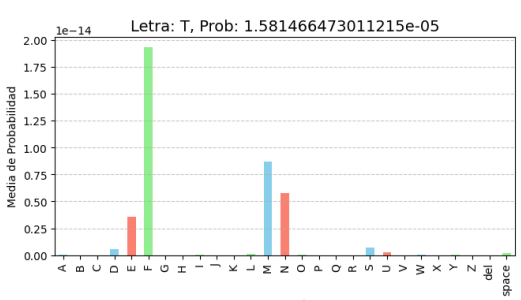

Letra: O, Prob: 9.786245556244931e-07
1e−14
Media de Probabilidad
Categoría

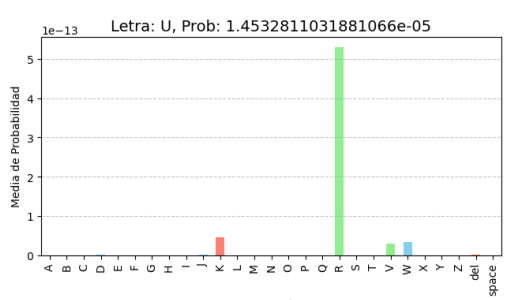

Letra: P, Prob: 0.006042284381704538
1e−13
Media de Probabilidad
Categoría

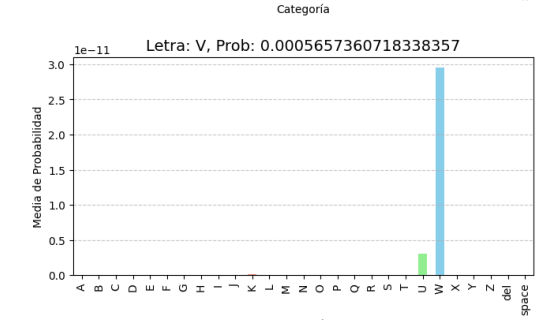

Letra: Q, Prob: 0.004525892763131617
1e−12
Media de Probabilidad
Categoría

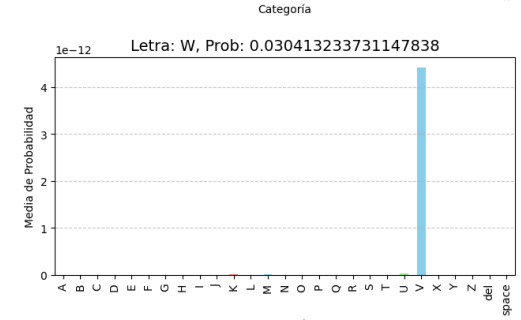

Letra: R, Prob: 1.4231043861542744e-06
1e−10
Media de Probabilidad
Categoría

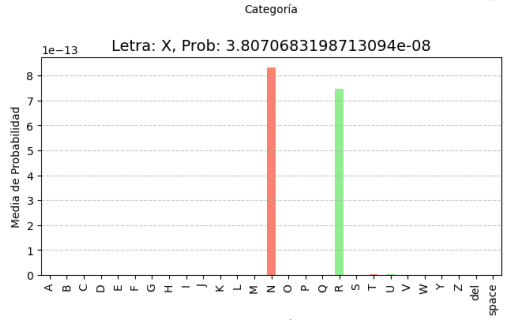

Letra: S, Prob: 3.195699299133107e-07
1e−12
Media de Probabilidad
Categoría

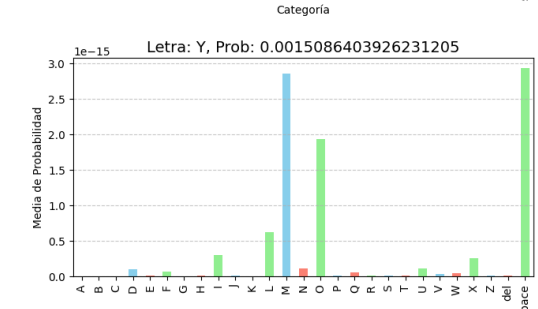

Letra: T, Prob: 1.581466473011215e-05
1e−14
Media de Probabilidad
Categoría

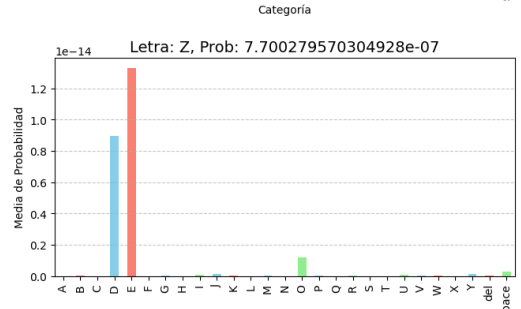

Letra: U, Prob: 1.4532811031881066e-05
1e−13
Media de Probabilidad
Categoría

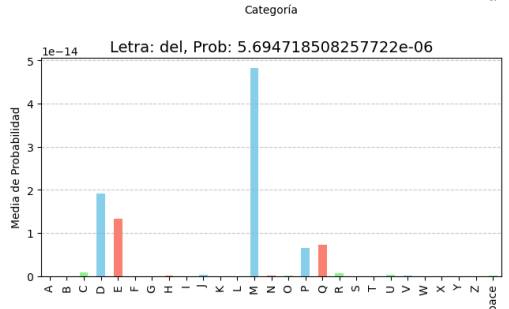

Letra: V, Prob: 0.0005657360718338357
1e−11
Media de Probabilidad
Categoría

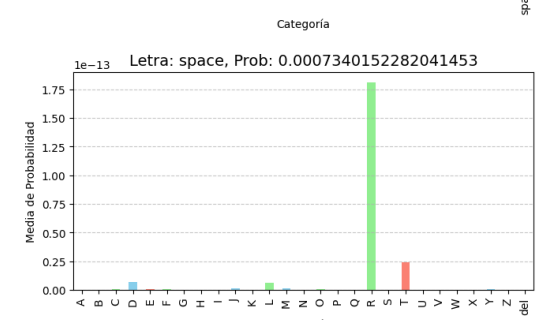

Letra: W, Prob: 0.030413233731147838
1e−12
Media de Probabilidad
Categoría


Letra: X, Prob: 3.8070683198713094e-08
1e−13
Media de Probabilidad
Categoría


Letra: Y, Prob: 0.0015086403926231205
1e−15
Media de Probabilidad
Categoría


Letra: Z, Prob: 7.700279570304928e-07
1e−14
Media de Probabilidad
Categoría


Letra: del, Prob: 5.694718508257722e-06
1e−14
Media de Probabilidad
Categoría


Letra: space, Prob: 0.0007340152282041453
1e−13
Media de Probabilidad
Categoría

Conclusiones

El desarrollo de este proyecto ha supuesto un desafío personal que me ha permitido mejorar mis habilidades en Python, redes neuronales convolucionales, diseño de aplicaciones estructuradas y manejo de librerías como OpenCV. Aunque el programa es funcional, el objetivo principal no era únicamente su utilidad inmediata, sino afrontar y superar retos técnicos importantes.

Durante el desarrollo, me encontré con dificultades como la imposibilidad de crear un detector de manos propio debido a la falta de conocimientos en ese momento y a problemas de compatibilidad, lo que me llevó a utilizar el detector ofrecido por la librería cvzone. Otro reto considerable fue optimizar el rendimiento del modelo, ya que a pesar de obtener buenos resultados en los test, su funcionamiento en la aplicación real no era suficiente.

A lo largo del proceso he adquirido experiencia práctica en la estructura modular de aplicaciones y en el control del flujo de trabajo.

De cara al futuro, me gustaría mejorar el proyecto desarrollando un detector de manos propio y optimizando la clasificación de ciertas letras como "G", "D", "J" y "Z", que presentan más dificultad. Además, me gustaría implementar un sistema de autocompletado basado en procesamiento de lenguaje natural (NLP) para facilitar la escritura, similar al sistema de predicción de texto de los teclados móviles. También considero el despliegue de la aplicación en plataformas móviles como un posible siguiente paso.

En resumen, este proyecto ha sido una oportunidad para superar retos técnicos relevantes, ampliar mis conocimientos en varias áreas de la programación y sentar una base sólida para futuras mejoras y expansiones.

Bibliografía

Dataset original: https://www.kaggle.com/datasets/debashishsau/aslamerican-sign-language-aplhabet-dataset

Modelo detección manos: https://www.youtube.com/watch?v=CKmAZss-T5Y&ab_channel=Murtaza%27sWorkshop-RoboticsandAI

Documentación Tensorflow: https://www.tensorflow.org/api_docs

Documentación OpenCV: https://opencv.org/

ChatGPT: https://chatgpt.com/

Enlaces de interés

Dataset fase 1: https://drive.google.com/drive/folders/1C4q9f8B7NgFUH1FCW-d8Qvv6CNVbtRza?usp=sharing

Dataset fase 2:

https://drive.google.com/drive/folders/1IOnNDU81rsty1VCpDjTTTqBgnSlmHKF7?usp=sharing

Dataset fase 3: https://drive.google.com/drive/folders/1XySivipdBRmM0ed-MdQ8HhUT6vqKrXAs?usp=sharing

Entrenamiento fase 1: https://colab.research.google.com/drive/1RycCwxJ_3ZH9xKwu4rQ8vvYZfTdaurYw?usp=sharing

Entrenamiento fase 2: https://colab.research.google.com/drive/1EsJZwYyBRLBIW-vfMQ9oyt_5MMLTMpaU?usp=drive_link

Entrenamiento fase 3: https://colab.research.google.com/drive/1SmjwYDlR6ALQCP2X7bJ0y91O2BkyvEIq?usp=drive_link

Tratamiento datos fase 2-3: https://colab.research.google.com/drive/1XEPfICGA-bs8K8L5hDX2Ujo4U7PlUeg4?usp=sharing

Modelo entrenado: https://drive.google.com/file/d/1yCBwKlpz1v-WH8VRKBvWPo_9_So17u7R/view?usp=drive_link

Git hub Proyecto: https://github.com/DaniFort/Nodd3r_TFM