GRADO EN CIENCIA DE DATOS

ESCUELA DE ARQUITECTURA Y POLITÉCNICA

TRABAJO FIN DE GRADO

Traducción automática de Alfabeto Dactilológico a Alfabeto Convencional

Autor: Jesús Llorens Planelles

Tutor: Jesús Friginal López

Fecha Convocatoria: 27/06/2023

Resumen

Este TFG propone una solución innovadora para facilitar la comunicación de las personas sordas con su entorno, mediante la traducción en tiempo real del alfabeto dactilológico al alfabeto convencional, el alfabeto dactilológico es de la Lengua de Signos Española (LSE).

El TFG plantea el diseño de un modelo óptimo que permite reconocer imágenes del alfabeto dactilológico y etiquetarlos, a partir de una red neuronal convolucional. Para la prueba de concepto se crea una aplicación que utiliza tecnologías de visión computacional permitiendo al usuario sordo formar frases a partir del procesamiento del alfabeto dactilológico automáticamente.

Palabras clave: Alfabeto Dactilológico, Lengua de Signos Español, Clasificador de Imágenes, Visión Computacional, Red Neuronal Convolucional, TensorFlow, OpenCV

Abstract

This TFG proposes an innovative solution to facilitate the communication of deaf people with their environment, through the real-time translation of the dactylological alphabet to the conventional alphabet, the dactylological alphabet is from the Spanish Sign Language (LSE).

The TFG proposes the design of an optimal model that allows recognizing images of the alphabet and labeling them, based on a convolutional neural network. For the proof of concept, an application is created that uses computer vision technologies allowing the deaf user to form sentences from the processing of the alphabet automatically.

Keywords: Dactylological Alphabet, Spanish Sign Language, Image Classifier, Computer Vision, Convolutional Neural Network, TensorFlow, OpenCV

Capítulo 1:

Introducción

El presente trabajo de fin de grado, se enfoca principalmente en desarrollar un programa que en tiempo real permite traducir el alfabeto dactilológico al alfabeto convencional de Lengua de Signos Española (LSE). Mediante conceptos de visión por computadora junto con aprendizaje automático podremos reconocer las letras del alfabeto dactilológico, que son los gestos que se hacen con la mano, con una simple webcam.

1.1. Objetivos

De manera esquematizada, estos son los objetivos más importantes del trabajo:

- **Documentación y estudio:**
  El estudio y comprensión de los fundamentos de la inteligencia artificial y conceptos relacionados, como la visión por computación. Además, el estudio de las herramientas y requisitos necesarios para llevar a cabo un problema como este.

- **Creación del clasificador de imágenes:**
  Crear un modelo con Tensor Flow, que cumpla con lo esperado y con el clasificador de imágenes, para su aplicación final. Que tengamos una precisión mínima para que la traducción en tiempo real sea correcta y fluida.

- **Creación de la aplicación:**
  Crear exitosamente la aplicación en Python que acceda a la webcam y mediante la librería OpenCV, pueda reconocer la mano del usuario y se pueda cargar el modelo de clasificación de imágenes en el programa.

- **Experimentos y resultados acordes a lo esperado:**
  Formar una frase con el programa en marcha, a partir de la signatura de la frase “Hola Mundo” en LSE.

1.2. Motivación

En la sociedad actual, la comunicación efectiva es fundamental para el desarrollo y la inclusión de todas las personas. Sin embargo, existe una barrera significativa que afecta a las personas sordas, quienes encuentran dificultades para comunicarse con aquellos que no conocen LSE. Esta limitación impide su plena participación en diversas interacciones sociales y puede generar aislamiento y exclusión.

Conscientes de esta necesidad social, surge la motivación de este proyecto: desarrollar un programa innovador que aborde este desafío y permita a las personas sordas comunicarse fluidamente con aquellos que no comprenden LSE. La idea es proporcionar una

solución tecnológica que facilite la comprensión en tiempo real de los gestos o signos utilizados por las personas sordas, acelerando así la interacción y fomentando una mayor inclusión social.

Este enfoque no solo busca mejorar la vida de las personas sordas al brindarles una herramienta para comunicarse con más facilidad, sino que también busca promover la integración de estas personas discapacitadas en la sociedad. Al derribar las barreras comunicativas, se les brinda la oportunidad de participar activamente en diversas actividades, relacionarse con personas que no conocen LSE y ampliar sus horizontes.

Además, la utilización de herramientas de visión computacional y aprendizaje automático en este proyecto supone un reto emocionante. Entrenar un modelo de clasificador de imágenes y mejorar los conocimientos en visión por computadora, permite la adquisición de habilidades valiosas en inteligencia artificial y se contribuye al desarrollo de soluciones innovadoras.

1.3. Organización del TFG

En este punto vamos a explicar los puntos en los que se ha dividido el TFG, explicando de una forma concisa el contenido y composición de cada capítulo.

En el primer capítulo se ha realizado una breve introducción al trabajo de fin de grado, haciendo una enumeración de los objetivos de dicho trabajo y el problema que se intenta resolver con este proyecto.

En el siguiente capítulo, el marco teórico, se trata de dar un contexto y hacer una investigación para aprender todos los conceptos relevantes y relacionados con el trabajo. Los campos tratados en la elaboración del TFG son conceptos de la inteligencia artificial y sus ramas.

En el tercer capítulo, se otorga información menos teórica sobre conceptos y más bien se proporciona información técnica relacionada con el trabajo. Se enseñan las herramientas utilizadas, las librerías y lo que es el dataset.

En el cuarto capítulo, se habla de todo el proceso del desarrollo de la aplicación, se entra en profundidad en todas las etapas y fases del desarrollo que comprenden la aplicación final.

En el quinto capítulo, tratamos de ver y analizar todos los resultados obtenidos. Vemos qué resultados nos ha dado el programa, comparamos los resultados y tratamos de saber si el programa ha cumplido las expectativas.

Acabando con el último capítulo, se realiza una reflexión de conclusiones tras haber terminado el proyecto. Además, vemos las limitaciones y problemas que hemos obtenido durante el desarrollo del trabajo. También, vemos futuras características que podríamos implementar el proyecto, para así, mejorar el trabajo en el caso que se siga desarrollando en el futuro.

Por último, se enumeran las referencias que se han utilizado para desarrollar este trabajo.

Capítulo 2:

Marco teórico

En este capítulo se hará el estudio de distintas aplicaciones desarrolladas con relación al de este proyecto. Además, se hace una investigación sobre los conceptos relevantes de la Inteligencia Artificial, la visión por computadora y la clasificación de imágenes.

2.1 Estado del arte:

La traducción en tiempo real del alfabeto dactilológico es un campo de investigación y desarrollo que ha experimentado avances significativos en los últimos años. A continuación, se brindarán algunos ejemplos relevantes de tecnologías y proyectos relacionados con la traducción del alfabeto dactilológico:

Proyecto SignAloud[1]: Desarrollado por un grupo de estudiantes de la Universidad de Washington, Sing Aloud es un guante equipado con sensores que capturan los movimientos de las manos y los traducen en palabras habladas. El sistema busca facilitar la comunicación entre personas sordas y oyentes.

Guante gestual de investigación de la Universidad de California: Investigadores de la Universidad de California han desarrollado un guante que utiliza sensores para detectar los movimientos de la mano y reconocer los gestos del alfabeto dactilológico. El guante se conecta a un dispositivo móvil y puede traducir los gestos en tiempo real.

Proyecto SignSpeak[2]: SignSpeak es un proyecto que combina la tecnología de sensores y el aprendizaje automático para reconocer y traducir los gestos de la lengua de signos. Utiliza un guante con sensores y una cámara para capturar los movimientos de las manos y proporcionar una traducción visual en tiempo real.

Estos son solo algunos ejemplos destacados de los avances en la traducción en tiempo real de la lengua de signos y del alfabeto dactilológico. Como se puede apreciar, la combinación de tecnologías como sensores, cámaras y algoritmos de aprendizaje automático ha abierto nuevas posibilidades para mejorar la comunicación y la inclusión de las personas sordas en diversos entornos.

2.2. Inteligencia Artificial y CNNs

La inteligencia artificial (IA)[3] es una disciplina que ha cobrado un papel fundamental en el desarrollo de la tecnología y ha generado un impacto significativo en diversos campos de aplicación. Se refiere a la capacidad de las máquinas para realizar tareas que requieren de habilidades y capacidades propias de la inteligencia humana, como el razonamiento, el aprendizaje, la percepción y la toma de decisiones. La IA busca emular y

superar las capacidades cognitivas humanas, permitiendo a las máquinas realizar actividades complejas de manera automatizada.

Dentro del campo de la inteligencia artificial (IA)[4], se han desarrollado diferentes enfoques para abordar el desafío de replicar la inteligencia humana en las máquinas. Estos enfoques han evolucionado a lo largo del tiempo y han sido fundamentales para el avance de la IA en diversas aplicaciones.

Estos son los cuatro tipos principales de IA: la IA basada en reglas, el aprendizaje automático, las redes neuronales y la IA basada en conocimiento.

La IA basada en reglas, también conocida como IA simbólica, se fundamenta en el uso de reglas y conocimientos específicos proporcionados por expertos. Estas reglas permiten tomar decisiones y realizar tareas específicas. Aunque este enfoque fue popular en las primeras etapas de la IA, su principal limitación radica en la dificultad de capturar todo el conocimiento humano en reglas explícitas.

Por otro lado, el aprendizaje automático se centra en capacitar a las máquinas para aprender de los datos sin una programación explícita. Los algoritmos de aprendizaje automático analizan los datos disponibles y extraen patrones y relaciones para realizar predicciones o tomar decisiones. Dentro del aprendizaje automático, se distinguen dos enfoques principales: el aprendizaje supervisado, donde se utilizan datos etiquetados para entrenar modelos predictivos, y el aprendizaje no supervisado, donde se buscan patrones y estructuras ocultas en los datos sin información de referencia previa.

Las redes neuronales artificiales, inspiradas en el funcionamiento del cerebro humano, son algoritmos de IA que utilizan interconexiones de nodos para procesar información. Estas redes son capaces de aprender y reconocer patrones complejos en los datos, lo cual ha llevado a importantes avances en áreas como el reconocimiento de imágenes y el procesamiento del lenguaje natural. Su capacidad para modelar relaciones no lineales y su adaptabilidad las convierten en herramientas poderosas en la IA contemporánea.

En nuestro TFG el uso de la inteligencia artificial es clave para el desarrollo del proyecto. Es clave porque lo que buscamos es, que el programa, reconozca mediante la visión computacional lo que un humano podría hacer con sus ojos. Como se ha visto con anterioridad, el tipo de inteligencia artificial que vamos a utilizar en este trabajo son las redes neuronales artificiales junto con aprendizaje profundo, que posteriormente vamos a hablar más en detalle.

Las redes neuronales artificiales[6], también conocidas como modelos de aprendizaje profundo o deep learning, han surgido como una de las tecnologías más prometedoras en el campo de la inteligencia artificial y el procesamiento de datos. Estas redes están inspiradas en el sistema nervioso biológico y su estructura y funcionamiento se asemejan a las redes neuronales presentes en el cerebro humano.

La historia de las redes neuronales se remonta a la década de 1940, cuando los primeros investigadores comenzaron a explorar la idea de imitar el comportamiento de las neuronas biológicas utilizando sistemas electrónicos. Sin embargo, fue en la década de 1980

cuando las redes neuronales experimentaron un renacimiento y se desarrollaron técnicas más sofisticadas que permitieron su aplicación práctica.

Las redes neuronales están compuestas por un conjunto de neuronas interconectadas, donde cada neurona procesa la información que recibe y la transmite a otras neuronas a través de conexiones ponderadas. Estas conexiones, también conocidas como pesos, determinan la influencia que una neurona tiene sobre otra. Durante el proceso de entrenamiento, los pesos se ajustan mediante algoritmos de optimización para que la red aprenda a reconocer patrones y realizar tareas específicas. Una de las principales características de las redes neuronales es su capacidad para aprender de manera automática a partir de ejemplos o datos de entrenamiento. Esto se logra mediante algoritmos de aprendizaje que ajustan los pesos de las conexiones de la red para minimizar la diferencia entre las respuestas obtenidas y las respuestas deseadas.

Las redes neuronales han demostrado un gran potencial en una amplia gama de aplicaciones. En el campo de la visión por computadora, las redes neuronales convolucionales han logrado avances significativos en el reconocimiento de objetos, la detección de rostros y la segmentación de imágenes. En el procesamiento del lenguaje natural, las redes neuronales recurrentes han mejorado la traducción automática, la generación de texto y el análisis de sentimientos[7].

Además, las redes neuronales se utilizan en aplicaciones como el reconocimiento de voz, la recomendación de productos, el análisis de datos, la predicción de comportamientos y muchas otras áreas. Su capacidad para manejar grandes volúmenes de datos y extraer características relevantes ha revolucionado la forma en que se abordan problemas complejos en campos como la medicina, la investigación científica, la industria y la tecnología.

Las redes neuronales artificiales representan un enfoque poderoso y versátil para abordar problemas complejos de aprendizaje automático y procesamiento de datos. Su capacidad para aprender de manera automática a partir de ejemplos y extraer características relevantes de grandes conjuntos de datos ha impulsado avances significativos en una variedad de campos. A medida que se continúa investigando y refinando las técnicas de aprendizaje profundo, se espera que las redes neuronales desempeñen un papel cada vez más importante en la transformación de la sociedad y la industria.

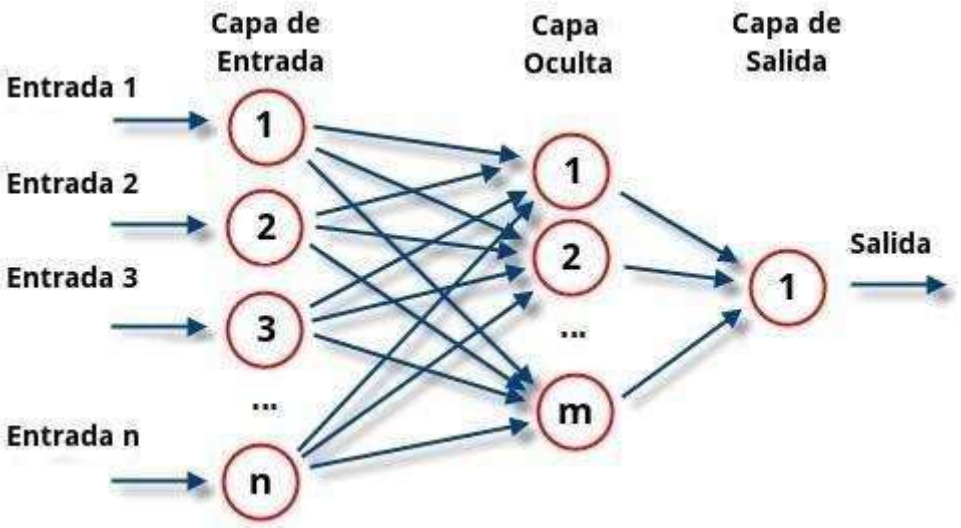

Fig.1 - Ejemplo de red neuronal.

De acuerdo con la estructura neuronal del cerebro, estas se agrupan por capas que se van conectando las salidas de una a las entradas de la siguiente formando la red neuronal. Las redes neuronales se organizan en tres tipos de capas:

- **Capa de entrada**:
  Este conjunto inicial de neuronas recibe la información original suministrada al sistema para realizar predicciones.

- **Capas ocultas**:
  Son conjuntos de neuronas interconectadas que se encargan de inferir el modelo y proporcionar un resultado para ser procesado por la capa de salida. El número de capas ocultas define la profundidad de la red neuronal.

- **Capa de salida**:
  Este último conjunto de neuronas combina todos los resultados obtenidos anteriormente y genera un resultado basado en las etiquetas del modelo.

2.3. Visión por computadora

La visión por computadora es una rama de la inteligencia artificial y la informática que se ocupa de permitir a las máquinas ver, comprender y procesar imágenes, de una

manera parecida a la que lo hacemos los seres humanos. Se podría decir que busca dotar a las máquinas de la capacidad de interpretar y comprender el contenido visual de las imágenes o vídeos, de la misma manera en que los seres humanos lo hacen. Este estudio se desarrolla mediante algoritmos y sistemas que permiten a las computadoras obtener información visual a partir de imágenes. Permitiendo a las máquinas realizar tareas como reconocimiento de objetos, detección de rostros, seguimiento de objetos en movimiento, entre otras aplicaciones[8].

La visión por computadora requiere el desarrollo de varios componentes fundamentales. Estos incluyen la adquisición de imágenes, que implica el uso de cámaras y otros dispositivos para capturar imágenes digitales. A continuación, el preprocesamiento se encarga de realizar operaciones básicas en las imágenes, como el filtrado y la eliminación de ruido. Seguidamente, se extraen características visuales, como bordes, texturas o formas, que permiten identificar objetos o patrones en la imagen. Posteriormente, se emplean técnicas de reconocimiento para clasificar y reconocer objetos o personas en las imágenes. Por último, la toma de decisiones permite que las máquinas interpreten la información visual y actúen en consecuencia.

La visión por computadora se basa en conceptos y técnicas provenientes de otras disciplinas, como la inteligencia artificial, el aprendizaje automático y la robótica. La inteligencia artificial proporciona el marco teórico y conceptual para comprender cómo las máquinas pueden adquirir conocimiento a partir de la información visual. El aprendizaje automático permite que las máquinas mejoren su desempeño en tareas de visión por computadora mediante algoritmos y modelos que aprenden de los datos. Por su parte, la robótica integra la visión por computadora en sistemas autónomos y robots para interactuar con el entorno.

El procesamiento de imágenes comprende diversas operaciones, como el filtrado, donde se aplican filtros espaciales o de frecuencia para resaltar o suavizar características de la imagen. También se realizan mejoras de contraste para ajustar la distribución de intensidades y hacer que los objetos sean más distinguibles.

Además, se lleva a cabo la eliminación de ruido, que puede deberse a imperfecciones en el sensor o durante la transmisión de la imagen. La segmentación consiste en dividir una imagen en regiones o segmentos significativos. Se utilizan algoritmos para separar objetos del fondo o para identificar diferentes partes de un objeto en la imagen. Esto resulta útil para análisis posteriores o para extraer características específicas de las regiones segmentadas. La extracción de características busca identificar y describir patrones visuales relevantes en una imagen, como texturas, bordes, colores o formas. Estas representaciones de características se utilizan posteriormente en algoritmos de reconocimiento y clasificación.

La detección de objetos consiste en localizar y reconocer la presencia de objetos específicos en una imagen o secuencia de imágenes. Por su parte, el seguimiento de objetos implica rastrear el movimiento de un objeto a lo largo del tiempo en un video. Se emplean técnicas como el seguimiento óptico y algoritmos de aprendizaje automático para lograr estas tareas. Asimismo, esta área se ocupa de interpretar los movimientos y gestos humanos capturados en imágenes o videos. Los algoritmos pueden analizar los patrones de

movimiento y las posiciones de las articulaciones para reconocer gestos específicos, como saludos, señales o comandos.

La visión por computadora encuentra aplicaciones en diversos campos. Se utiliza en sistemas de control de calidad y en procesos automatizados para inspeccionar productos, detectar defectos o realizar tareas de ensamblaje. Además, tiene aplicaciones en el campo médico, como el diagnóstico por imágenes, donde se analizan radiografías, tomografías computarizadas o resonancias magnéticas para detectar enfermedades o anomalías.

La visión por computadora puede llevar a cabo una amplia variedad de tareas:

- Detección y reconocimiento de objetos ●
- Seguimiento de objetos.
- Reconocimiento y clasificación de patrones.
- Análisis de movimiento
- Reconstrucción 3D
- Realidad aumentada

2.4. Clasificación de imágenes

La clasificación de imágenes es una tarea fundamental en el campo del aprendizaje automático y la visión por computadora. Se refiere al proceso de asignar etiquetas o categorías a las imágenes según su contenido visual. El objetivo es entrenar un modelo de aprendizaje automático para reconocer y distinguir diferentes objetos, patrones o conceptos presentes en las imágenes. En la clasificación de imágenes, se utiliza un conjunto de imágenes previamente etiquetadas como datos de entrenamiento. Estas imágenes se utilizan para entrenar un modelo, que aprende a extraer características distintivas de las imágenes y asociarlas con las etiquetas correspondientes. El modelo utiliza estas características para realizar predicciones sobre nuevas imágenes, asignándoles las etiquetas más apropiadas en función de lo que ha aprendido durante el entrenamiento[9].

Capítulo 3:

Metodología y Herramientas Empleadas

En este capítulo se ven las diferentes metodologías y herramientas pertinentes con este proyecto. En este capítulo se verán toda la metodología pertinente con la clasificación de imágenes, mostrando con un diagrama la arquitectura del modelo. Además, se procederá a explicar lo que será nuestro dataset y las características que debe seguir.

3.1 Metodología de clasificación de imágenes

El proceso de clasificación de imágenes consta de varias etapas que se seguirán en el Trabajo de Fin de Grado (TFG) con el objetivo de lograr la clasificación en tiempo real de los del alfabeto dactilológico utilizando la cámara. Estas etapas son las siguientes:

En primer lugar, se realiza el procesamiento de datos. Esto implica redimensionar y normalizar las imágenes de entrenamiento, así como aplicar técnicas de mejora de calidad, como el filtrado y la eliminación de ruido. Estas acciones aseguran que las imágenes tengan una entrada adecuada al modelo.

En la segunda etapa, se lleva a cabo la extracción de características. Para esto, se utilizan técnicas como las redes neuronales convolucionales (CNN) que permiten extraer características significativas de las imágenes. Estas características capturan detalles visuales relevantes, como bordes, formas, texturas o colores, que son fundamentales para que el modelo pueda distinguir entre diferentes clases.

La tercera etapa se enfoca en el entrenamiento del modelo. Aquí se utilizan los datos de entrenamiento para ajustar los parámetros del modelo. Durante este proceso, el modelo aprende a asignar pesos adecuados a las características extraídas y a realizar predicciones precisas en función de dichas características.

Posteriormente, se lleva a cabo la cuarta etapa, que implica la evaluación y el ajuste del modelo. Para ello, se utilizan datos de validación o prueba con el fin de evaluar el rendimiento del modelo. En caso necesario, se realizan ajustes en los hiper parámetros o en la arquitectura del modelo para mejorar su desempeño.

Finalmente, en la última etapa, se realiza la inferencia. Una vez que el modelo ha sido entrenado y evaluado, se utiliza para clasificar nuevas imágenes. Las características aprendidas durante el entrenamiento se aplican para asignar las etiquetas adecuadas a las imágenes de prueba.

De esta manera, se seguirán todos estos pasos en el TFG con el propósito de entrenar una red neuronal capaz de realizar la clasificación de imágenes en tiempo real de los signos utilizando la cámara.

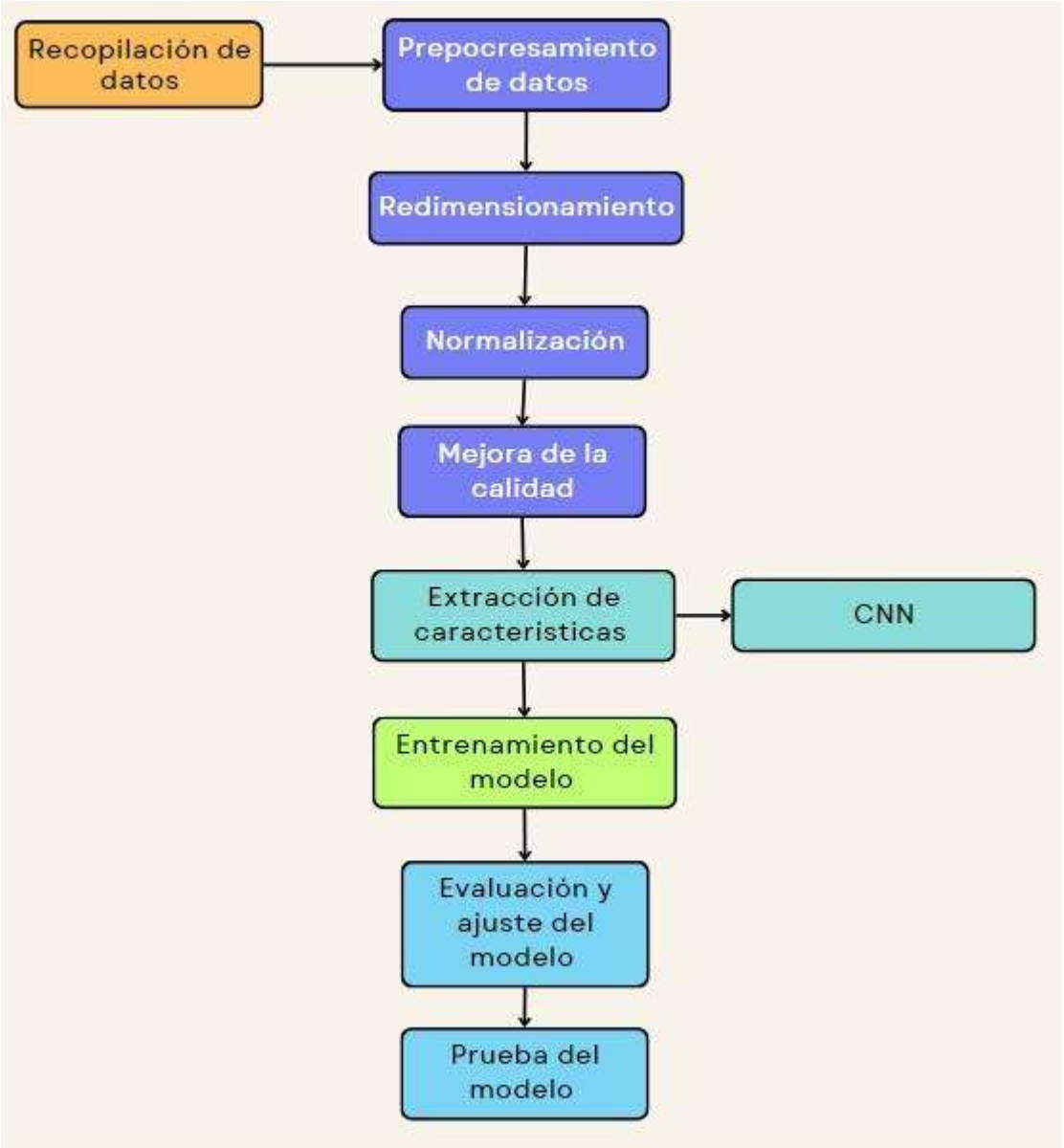

Fig.2 - Arquitectura de nuestro modelo de clasificación de imágenes.

Como se puede ver en el diagrama, este es el proceso que nuestro modelo de inteligencia artificial seguirá, cada color representa una etapa. En el siguiente capítulo se verá cómo se crea el modelo siguiendo estas pautas.

3.2. Dataset

Para entrenar a un modelo de inteligencia artificial es necesario alimentarlo con los datos adecuados. Si el modelo de entrenamiento no recibe unos datos mínimos, o de calidad el modelo de inteligencia artificial no será útil para nuestro objetivo.

La correcta recolección de estos elementos es fundamental en el proceso de entrenamiento, ya que trabajar con una muestra insuficiente impedirá que el modelo pueda llegar a conclusiones válidas al no ajustarse adecuadamente el algoritmo de predicción. Por otro lado, utilizar un conjunto de datos de entrada centrado únicamente en una o varias etiquetas tampoco arrojará resultados óptimos, ya que se generalizaron incorrectamente los patrones asociados a esas etiquetas más frecuentes en comparación con el resto del conjunto de datos, lo que conducirá a predicciones erróneas o, en muchos casos, a un sobreajuste.

En el presente TFG se maneja un dataset comprendido por imágenes como tipo de dato de entrada, esto se debe a la complejidad del problema propuesto, un clasificador de imágenes. Dichos datos tendrán una etiqueta, que significa lo que representan, así el modelo sabrá identificar dichas imágenes. Por tanto, será necesario obtener un conjunto de imágenes de todos los tipos de signos que se comprendan el alfabeto dactilológico del lenguaje de signos español. Para enriquecer los datos, se deberá hacer fotos del mismo signo con distintos ángulos y distintas perspectivas, esto lo que nos dará es una mejor perspectiva posible para los signos y por ende aumentará las probabilidades de mejores precisiones a la hora de predecir los resultados.

Una estrategia adicional en este tipo de conjuntos de datos de imágenes para ampliar la colección de imágenes recopiladas es la técnica de enriquecimiento de datos (Data Augmentation). Este enfoque implica duplicar los datos del conjunto original y aplicarles filtros o modificaciones, sin comprometer las características fundamentales de los datos. Esta metodología se puede aplicar sin dificultad en este caso debido a la amplia gama de filtros y acciones disponibles para alterar una imagen sin perder sus características distintivas.

Cabe destacar que hemos definido las fotos con un tamaño que va del 0 a 255 píxeles. El uso de un rango de 0 a 255 para representar los píxeles facilita el procesamiento y la manipulación de imágenes, ya que se ajusta bien al formato de datos binarios y a las operaciones aritméticas comunes. Además, muchos algoritmos y técnicas de procesamiento de imágenes están diseñados para trabajar con este rango específico de valores de píxeles.

Es importante coger nuestro dataset y dividirlo en dos grupos, “train” y “test”.
Al tener el conjunto de datos dividido lo que podremos hacer es entrenar el modelo con un conjunto y luego validamos ese modelo, con las fotos del otro grupo. Dividir nuestro conjunto de datos en conjuntos de entrenamiento (train) y prueba (test) es una práctica fundamental en el aprendizaje automático (machine learning). Esta división nos permite evaluar la capacidad predictiva de nuestros modelos de aprendizaje automático de manera más confiable y realista. Esto es muy importante ya que podremos evitar problemas como el sobreajuste de datos.

Lo que tenemos que hacer es definir ciertos aspectos técnicos. En este apartado lo que hago, además de tener un conjunto de fotos medianamente grande, aplicó clases de Keras como “ImageDataGenerator” que nos aumenta el conjunto de datos de imágenes de entrada. Lo primero es la función “rescale”, normaliza los valores de píxeles de las imágenes dividiéndolos por 255, lo que los coloca en el rango de 0 a 1. Esto es comúnmente hecho para facilitar el entrenamiento del modelo. Luego, “Shear Range” aplica una transformación

de cizallamiento aleatoria a las imágenes. El cizallamiento cambia la forma de la imagen al deslizar una parte de la imagen en una dirección determinada. El Zoom Range, aplica una transformación de zoom aleatoria a las imágenes. El zoom modifica la escala de la imagen, acercándose o alejándose. Por último, Horizontal Flip voltea horizontalmente las imágenes de manera aleatoria. Estas dos funciones, son muy interesantes porque como hemos hablado en puntos anteriores, a la hora de entrenar el modelo brinda de forma óptima, diferentes puntos de vista o ángulos de las fotos, con las que nos permite enriquecer el conjunto de datos.

Capítulo 4:

Desarrollo tecnológico

En este capítulo se explica todo el desarrollo tecnológico para el proyecto, sus librerías, sus etapas de desarrollo y la implementación de dicho código. Librerías como Tensor Flow, Tensor, Keras y OpenCV proporcionan herramientas y funcionalidades clave que facilitan la implementación y el desarrollo de modelos de aprendizaje automático, especialmente en tareas de clasificación de imágenes.

4.1. Librería Tensor Flow

Tensor Flow es una biblioteca de software de código abierto utilizada para desarrollar y entrenar modelos de aprendizaje automático. En su núcleo, Tensor Flow se basa en la idea de representar los cálculos matemáticos como un grafo de flujo de datos, donde los nodos representan operaciones matemáticas y los bordes del grafo representan los datos en forma de tensores, que son arreglos multidimensionales. Esta estructura de grafo permite realizar cálculos eficientemente en hardware especializado como GPU y TPU.

Tensor Flow proporciona una interfaz de programación sencilla y flexible que permite a los desarrolladores construir modelos de aprendizaje profundo de manera eficiente y escalable. Además, Tensor Flow ofrece una variedad de herramientas y recursos para el desarrollo de modelos, como capas predefinidas, algoritmos de optimización y visualización de datos, lo que facilita la implementación y experimentación en proyectos de aprendizaje automático[10].

El Tensor Flow es ampliamente utilizado para la clasificación de imágenes debido a varias razones clave.

En primer lugar, Tensor Flow proporciona una amplia gama de modelos pre entrenados y capas de redes neuronales convolucionales (CNN) altamente optimizadas. Estas capas especializadas en el procesamiento de imágenes permiten una extracción de características eficiente y una representación efectiva de los datos visuales.

En segundo lugar, Tensor Flow ofrece una gran flexibilidad para construir y personalizar modelos de clasificación de imágenes. Los desarrolladores pueden utilizar la API de alto nivel de Tensor Flow (Keras) para construir rápidamente modelos con pocas

líneas de código o pueden trabajar directamente con la API de bajo nivel de Tensor Flow para tener un control más preciso sobre cada componente del modelo.

Además, Tensor Flow cuenta con una amplia comunidad de usuarios y una documentación exhaustiva. Esto facilita el acceso a tutoriales, ejemplos de código y recursos educativos que ayudan a los desarrolladores a comprender y utilizar eficientemente las capacidades de Tensor Flow para la clasificación de imágenes.

Por último, Tensor Flow es compatible con el procesamiento en GPU y TPU, lo que acelera significativamente el entrenamiento y la inferencia de modelos de clasificación de imágenes. Esto permite manejar conjuntos de datos grandes y complejos, y acelerar el tiempo de respuesta en aplicaciones en tiempo real.

4.2 Tensor

Un tensor es una estructura de datos fundamental en el campo de la matemática y el aprendizaje automático. En términos simples, un tensor es una generalización multidimensional de un vector o una matriz. Puede contener elementos de datos organizados en forma de una o más dimensiones. En el contexto del aprendizaje automático, los tensores son fundamentales para representar y manipular datos numéricos en modelos de aprendizaje automático, como redes neuronales. Los frameworks de aprendizaje automático, como Tensor Flow y PyTorch, utilizan tensores como estructura de datos principal para realizar cálculos y entrenar modelos. Es una estructura de datos multidimensional utilizada para representar y manipular datos numéricos en el aprendizaje automático. Puede tener diferentes grados de dimensionalidad y es fundamental en el procesamiento y análisis de datos en modelos de aprendizaje automático.

Los tensores permiten representar y almacenar datos numéricos en diferentes dimensiones. En el aprendizaje profundo, los datos de entrada, como imágenes, texto o secuencias de tiempo, se representan como tensores para que puedan ser procesados por los modelos de aprendizaje profundo. Los tensores capturan la estructura y las relaciones entre los datos, lo que es esencial para el procesamiento y análisis de datos complejos.

En el entrenamiento de los modelos de aprendizaje profundo, los tensores se utilizan para representar los datos de entrada, los pesos de las conexiones entre las unidades y los gradientes de las funciones de pérdida. Durante el proceso de entrenamiento, se realizan cálculos en los tensores para ajustar los pesos del modelo y optimizar su rendimiento. Los tensores permiten calcular gradientes y aplicar algoritmos de optimización, como el descenso de gradiente, para mejorar la capacidad del modelo para realizar predicciones precisas.

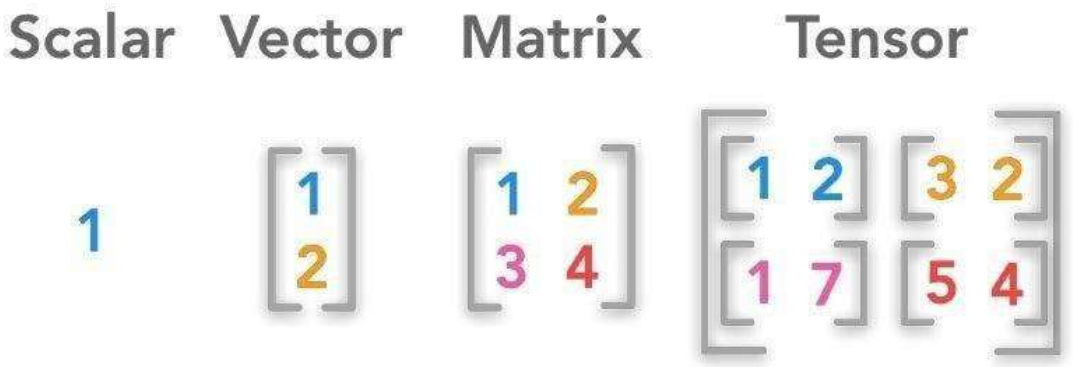

Fig.3 -Representación visual de un tensor.

4.3. Librería Keras

Keras es una biblioteca de alto nivel escrita en Python que se utiliza para el desarrollo de redes neuronales y el aprendizaje profundo (deep learning). Proporciona una interfaz sencilla y fácil de usar para crear, entrenar y evaluar modelos de aprendizaje automático, especialmente en el campo de la visión por computadora, el procesamiento del lenguaje natural y otras tareas relacionadas con datos secuenciales[11].

Keras fue desarrollado originalmente por François Chollet como parte de su proyecto de investigación en el año 2015. Su objetivo principal era proporcionar una biblioteca de aprendizaje profundo que fuera simple, modular y fácil de extender. En sus primeras versiones, Keras se ejecutaba sobre el framework Theano. Posteriormente, se amplió para ser compatible con otros frameworks populares como Tensor Flow y Microsoft Cognitive Toolkit (CNTK). Desde entonces, Keras se ha convertido en una de las bibliotecas más utilizadas y populares para el desarrollo de redes neuronales en Python.

Keras se utiliza ampliamente en el campo del aprendizaje profundo debido a su enfoque en la facilidad de uso y la flexibilidad. Keras permite la construcción de modelos de manera modular, donde las capas se pueden apilar y conectar fácilmente para formar una arquitectura compleja. Esto facilita la creación de redes neuronales con múltiples capas y conexiones personalizadas. Keras es compatible con varios frameworks populares de aprendizaje automático, como Tensor Flow. Esto proporciona flexibilidad y opciones para utilizar el backend preferido según las necesidades y preferencias del usuario.

Keras es ampliamente utilizado para desarrollar modelos de aprendizaje profundo en tareas de reconocimiento y clasificación de imágenes. Proporciona una amplia gama de funcionalidades y herramientas específicas para abordar este tipo de problemas.

Keras nos va a ser sumamente útil en este trabajo ya que nos va a proporcionar muchas funciones y características que utilizaremos:

- **Capas especializadas**
  Ofrece una variedad de capas especializadas para el procesamiento de imágenes, como capas de convolución, capas de pooling y capas de normalización. Estas capas son fundamentales para extraer características relevantes de las imágenes y construir modelos de clasificación efectivos.

- **Modelos pre-entrenados**
  Estos modelos pre-entrenados han sido entrenados en grandes conjuntos de datos y se pueden utilizar como punto de partida para tareas de clasificación de imágenes, permitiendo una rápida implementación y un buen rendimiento inicial.

- **Generadores de flujo de imágenes**
  Keras cuenta con generadores de flujo de imágenes, como el "ImageDataGenerator", que facilitan la carga y la transformación de grandes conjuntos de imágenes durante el entrenamiento.

- **Métricas de evaluación**
  Keras ofrece una variedad de métricas de evaluación específicas para la clasificación de imágenes, como precisión (accuracy), pérdida (loss), precisión por clase (class accuracy) y matriz de confusión (confusion matrix). Estas métricas permiten evaluar y comparar el rendimiento de los modelos en la clasificación de imágenes.

4.4. Librería OpenCV

OpenCV es una biblioteca de visión artificial originalmente desarrollada por Intel en 1999 y se ha convertido en un estándar de facto en la comunidad de visión por computadora debido a su amplio soporte, documentación extensa y su capacidad para trabajar con múltiples lenguajes de programación.. El significado del nombre es Open Computer Vision[12].

Su popularidad se debe a que es libre, por lo que nos permite usarla libremente para propósitos comerciales tanto como de investigación. Además, es multiplataforma, podemos usar esta librería en cualquier sistema operativo, tanto en Linux y sus distribuciones, como Mac OS y Windows. Podemos ejecutar esta librería en cualquier ordenador.

OpenCV está desarrollado en C + +, orientado a objetos y con una alta eficiencia computacional, pero incluye compatibilidad con otros lenguajes de programación, por lo que la hace una librería muy dinámica. Los otros lenguajes son, Python, Java, Matlab, Octave y JavaScript, como podemos ver al tener acceso a tantos lenguajes diferentes la librería brinda una libertad a los usuarios finales que otras APIS o librerías de computer vision no tienen.

El objetivo principal de OpenCV es proporcionar un conjunto de herramientas y algoritmos para el procesamiento de imágenes y videos.Una de las fortalezas de OpenCV es su capacidad para trabajar con imágenes en tiempo real, lo que lo hace adecuado para aplicaciones que requieren un procesamiento rápido y en tiempo real, como el seguimiento

de objetos, la detección de caras, el reconocimiento de objetos, entre otros. OpenCV también proporciona algoritmos y métodos para la detección de características en imágenes, como esquinas, bordes y líneas, así como técnicas más avanzadas como la detección de contornos y regiones de interés. Estas características pueden ser utilizadas para aplicaciones como la detección de objetos y la extracción de información relevante de las imágenes.

Además, OpenCV se integra con otras bibliotecas y herramientas populares utilizadas en el campo de la ciencia de datos y el aprendizaje automático, como NumPy, SciPy y Tensor Flow.

Esto permite combinar las capacidades de procesamiento de imágenes de OpenCV con técnicas de aprendizaje automático para tareas más avanzadas, como el reconocimiento de patrones y la clasificación de imágenes.

4.5. Entorno de desarrollo

Para este trabajo de fin de grado, se ha utilizado un ordenador con características que no son ni las más modernas, ni tampoco las mejores prestaciones en cuanto a hardware. Por lo que la velocidad de ejecución, el rendimiento y la calidad del entorno de desarrollo podría mejorar con un hardware más potente.

Como características principales del ordenador son:

- Procesador: Intel i7-6700K CPU 4.00GHz
- RAM: 16.00 GB
- Tipo de sistema: Sistema operativo de 64 bits (x64)
- Sistema Operativo: Windows 10
- Lenguaje de programación: Python 3.8.10
- Librería Numpy == 1.24.3
- Librería Mediapipe == 0.9.3.0
- Librería Opencv == 4.7.0.72
- Librería Cvzone ==1.5.6
- Software de Desarrollo = Visual Studio Code
- Formato de las imagenes = .jpg

Estas son las características que he tenido a la hora del desarrollo del trabajo de fin de grado. Esas versiones de las librerías de Python, tanto del propio Python son las que he necesitado para que el código funcionase y no hubiese problemas de compatibilidad. Las características del hardware son óptimas para un desarrollo de visión computacional y aprendizaje automático ya que la RAM como el procesador son recomendados para este tipo de trabajos.

Para ejecutar el código de el entrenamiento del modelo tanto como el código que ejecuta el programa se harán en visual studio, ambos. Para el código de ejecución del

programa se necesitará de una webcam integrada en el ordenador, o como es mi caso una que se puede conectar mediante usb.

4.6. Diseño de la aplicación

El diseño de este proyecto se segmenta en dos partes, la primera sería la creación del clasificador de imágenes y la segunda es la aplicación, el programa que orquesta el modelo entrenado, junto a la cámara del ordenador que mediante ella detecta nuestra mano. Por lo que necesitaremos hacer un primer archivo python, que entrene el modelo y otro a parte que tenga el acceso a la cámara, donde se le ha importado el modelo entrenado en el otro. Para ambos archivos python los ejecutamos en el entorno de desarrollo Visual Studio. Por lo que tendremos una estructura de archivos en nuestro Visual Studio, tal que así:

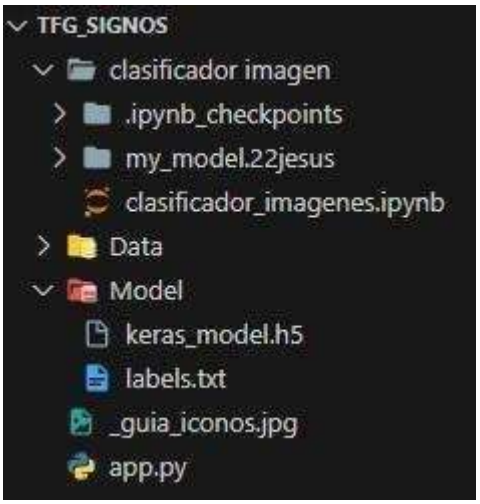

Fig.4 - Arquitectura de nuestro proyecto.

Como hemos hablado anteriormente, nuestro proyecto se compone por un “app.py” en el cual se ejecuta la aplicación como tal. El fichero “Data” en el que tendremos las carpetas de cada letra del abecedario que contienen más de 150 fotos, representando la letra. Seguido tenemos la carpeta “Model”, donde tenemos el modelo exportado, el cual hemos entrenado en el otro archivo. Por último, tenemos un fichero que contiene un archivo tipo notebook de python[13] donde está el modelo que hemos entrenado, por eso la terminación del archivo es distinta. He preferido usar un notebook, porque para hacer una red neuronal, tendremos que seguir muchos pasos, los cuales quedan más ordenados en formato celdas, lo que nos permitirá ejecutar el código por pasos.

Cabe recalcar que lo que se ejecutará es “app.py”, el modelo de clasificación se dejará de utilizar una vez hayamos obtenido el modelo deseado.

4.7. Obtención de los datos

Para la obtención de nuestros datos lo que he decidido hacer, es crear mi dataset a mano. Tras una búsqueda por internet no he podido encontrar un dataset de libre acceso con fotos de calidad, del lenguaje de signos español. Como el Español es el segundo idioma del mundo y no he encontrado un dataset libre, hemos decidido hacer nuestro propio dataset. Para ello me he basado en una guía de los gestos que hay que hacer para cada signo, la cual he seguido con cada letra del abecedario. Para ello, primero probaremos a hacer las fotos con el teléfono móvil, pero esto implica varios problemas, como que se hace un proceso muy lento y no es lo más cómodo. Además, de que luego hay que pasarlos al ordenador en lo cual se puede perder calidad de las fotos en el proceso.

Para un clasificador de imágenes necesitamos un dataset necesita por lo general una media de mil fotos por categoría, para entrenar el modelo de una forma decente. Como se ha hablado en puntos anteriores (3.6. Dataset) se usará la función de data augmentation para crear más fotos de forma artificial.Las imágenes se tomaron con la librería Opencv, que permite hacer capturas desde la webcam. Lo bueno de esta librería es que ya tiene integrada funciones que permiten reconocer las manos, y segmentar las manos con puntos, para que sea más fácil entrenar el modelo. Así quedaría nuestro dataset, después explicaremos una función para aumentar el dataset en el código.

Cabe recalcar que para nuestro modelo seguimos la regla del 70/30, el primer grupo para los datos de entrenamiento y el segundo para los de prueba. Aquí una representación gráfica de cómo están distribuidos nuestros datos.

FOTOS	NºIMÁGENES	Reserva Training:	Reserva Testing:
A	150	105	45
B	150	105	45
C	150	105	45
D	150	105	45
E	150	105	45
F	150	105	45
G	150	105	45
H	150	105	45
I	150	105	45
J	150	105	45
K	150	105	45
L	150	105	45
M	150	105	45
N	150	105	45
Ñ	150	105	45
O	150	105	45
P	150	105	45
Q	150	105	45
R	150	105	45
S	150	105	45
T	150	105	45
U	150	105	45
V	150	105	45
W	150	105	45
X	150	105	45
Y	150	105	45
Z	150	105	45
TOTAL fotos:	**4050**	**2835**	**1215**

Fig.5 - Número y tipo de clases para sus respectivas proporciones para el modelo.

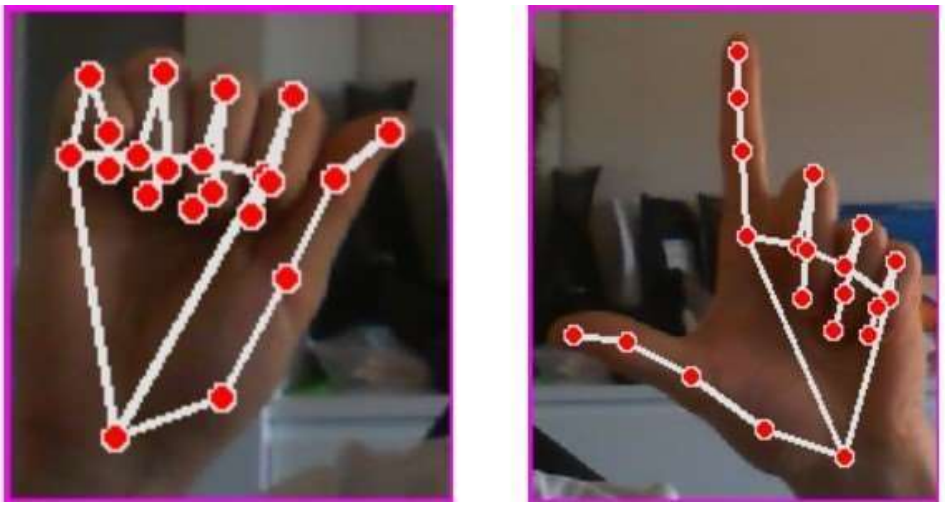

Fig.6 - Ejemplo de dato tomado con Opencv, izquierda A, derecha L.

Como se puede apreciar en la foto, se ven los puntos que segmentan mi mano lo que facilita al ordenador a entender los gestos y diferenciarlos. Al final el ordenador se ayudará de estos puntos, ya que podrá seguir patrones bien predefinidos. Para brindar una mayor calidad a las fotos he decidido centrar todas las fotos, para dejar siempre el mismo tamaño de en los lados de blanco, así todas los signos están en el centro. En este caso no se decidió pasar las fotos a blanco y negro, que en ocasiones es útil, pero en nuestro caso no hemos querido que el color de la mano es un valor que lo queremos conservar a la hora que el modelo pueda detectar las manos luego. Se ha decidido entrenar el modelo sin una uniformidad del fondo, por el motivo de que queremos a la hora de probarlo en la aplicación el fondo no será siempre el mismo, porque se usará con la webcam, no se quiere restringir el modelo así. No asegurar la uniformidad del fondo puede ayudar al modelo a adaptarse y clasificar imágenes en diferentes contextos y condiciones.

Como se puede ver en está foto, importamos librerías básicas, como las primeras que se ven, que nos permiten acceder a rutas en el ordenador, o exportar archivos en el ordenador. Librerías como “numpy” que nos permitirá re-escalar el tamaño de imágenes y “matplotlib” que nos va permitir graficar las imágenes dentro del código python. Las otras librerías que se pueden ver, son todas librerías relacionadas o que parten de Tensor Flow, que veremos más adelante.

La función “train_generator” es un generador de flujo (flow) de imágenes para el conjunto de entrenamiento. Se utiliza el método “flow_from_directory()” para cargar las imágenes desde un directorio y generar lotes (batches) de imágenes de tamaño 32. Además, se especifica el tamaño objetivo de las imágenes (224x224 píxeles en este caso). El conjunto de entrenamiento se espera que esté ubicado en el directorio “./datasets/train”.

La función “test_datagen” y “test_generator”, estos objetos se crean de manera similar a los anteriores, pero se utilizan para el conjunto de prueba. También se aplican las mismas transformaciones de datos, como “rescale”, cizallamiento, zoom y volteo horizontal. El conjunto de prueba se espera que esté ubicado en el directorio “./datasets/test”.

Se podría resumir como que este código utiliza “ImageDataGenerator” de Keras para generar conjuntos de datos de imágenes de entrenamiento y prueba con transformaciones y aumentos de datos aplicados.

```
train_datagen = ImageDataGenerator(rescale=1./255, # Rescale the pixel values between 0 and 1
                                   shear_range=0.2, # Apply random shear transformation
                                   zoom_range=0.2, # Apply random zoom transformation
                                   horizontal_flip=True) # Flip the image horizontally

train_generator = train_datagen.flow_from_directory(
        './datasets/train',
        target_size=(224, 224), # Resize the images to 224x224 pixels
        batch_size=32,)
        #class_mode='categorical') # Set the class mode to categorical

test_datagen = ImageDataGenerator(rescale=1./255,
                                  shear_range=0.2,
                                  zoom_range=0.2,
                                  horizontal_flip=True)

test_generator = test_datagen.flow_from_directory(
        './datasets/test',
        target_size=(224, 224), # Resize the images to 224x224 pixels
        batch_size=32,)
```

Fig.7- Uso de las funciones Data Generator.

4.8. Planificación del modelo:

Con el dataset finalizado podemos proceder con el entrenamiento del modelo, ahora procedemos a decidir qué arquitectura de modelo vamos a elegir para está aplicación. Gracias a la documentación de la librería Tensor Flow, de la cual hemos hablado en el marco teórico, utilizaremos esta librería para hacer nuestro código del clasificador de imágenes.

En primer lugar vamos a importar todas las librerías que requerimos para ejecutar el código.

Cuando hablamos de generar lotes (o batches) de imágenes de tamaño 32 significa agrupar varias imágenes juntas en conjuntos de 32. En el contexto de entrenamiento de modelos de aprendizaje automático, el conjunto de datos puede ser demasiado grande para procesarlo de una vez, por lo que se divide en lotes más pequeños para realizar el entrenamiento de manera más eficiente.El uso de lotes tiene varias ventajas, como la utilización eficiente de los recursos computacionales y la capacidad de aprovechar la paralelización en el procesamiento de los lotes en sistemas con múltiples núcleos o GPUs. Además, dividir el conjunto de datos en lotes también puede ayudar a evitar problemas de falta de memoria al procesar grandes cantidades de datos. En nuestro caso es interesante aplicarlas ya que estamos usando un volumen de datos que no es pequeño, que son 4050 fotos, además de que con “dataAugmentation”, estamos usando más valores para el modelo. Al dividir el conjunto de datos en lotes más pequeños, se pueden aprovechar mejor los recursos computacionales disponibles, como la capacidad de paralelización en sistemas con múltiples núcleos o unidades de procesamiento gráfico (GPUs). Esto permite un procesamiento más eficiente y rápido de las imágenes durante el entrenamiento.

Acto seguido, usamos las funciones creadas anteriormente, para tener los conjuntos de datos de train y test.

```
(x_train, y_train) = next(train_generator)
(x_test, y_test) = next(test_generator)
```

Fig.8 - Inicialización de las funciones con el conjunto de los datos.

Las líneas de código "print(x_train.shape)" y "print(x_test.shape)" muestran la forma (shape) de los conjuntos de datos de entrenamiento (x_train) y prueba (x_test) respectivamente. El resultado impreso "(32, 224, 224, 3)" indica la forma de los tensores, que se compone de cuatro dimensiones. Se podría decir que cada conjunto contiene 32 imágenes en color, cada una con una resolución de 224x224 píxeles.

En este caso, la forma del tensor x_train es (32, 224, 224, 3), lo que significa lo siguiente:

- El primer valor (32) representa el tamaño del lote (batch size), es decir, la cantidad de ejemplos de entrenamiento incluidos en cada lote de procesamiento. En este caso, hay 32 imágenes en cada lote.

- Los siguientes dos valores (224, 224) indican el tamaño de la imagen. Cada imagen tiene una resolución de 224 píxeles de alto y 224 píxeles de ancho.

- El último valor (3) representa los canales de color de la imagen. En este caso, hay 3 canales de color, que corresponden a las componentes rojo, verde y azul (RGB). Esto significa que las imágenes son imágenes en color.

4.9. Diseño de la Red Neuronal Convolucional:

En este tramo de código se define y compila un modelo de redes neuronales convolucionales (Convolutional Neural Network, CNN) utilizando la biblioteca Keras con el backend de Tensor Flow.

Primero hacemos la importación de la capa “MaxPooling2D”. Esta capa se utiliza para realizar operaciones de max pooling en el modelo, que es una técnica de reducción de dimensionalidad comúnmente utilizada en CNN para extraer características importantes de las imágenes.

Luego hacemos la definición del modelo, aquí se utiliza la función Sequential() para crear una instancia del modelo secuencial de Keras, que es una pila lineal de capas. Las capas se agregan secuencialmente al modelo.

En las capas convolucionales y de max pooling se definen varias capas “Conv2D” (convolucionales) y “MaxPooling2D” (de max pooling) para extraer características de las imágenes. Cada capa “Conv2D” especifica el número de filtros (32, 64, 128) y el tamaño del filtro (3x3). La función de activación 'relu' se utiliza para introducir no linealidad en el modelo. Después de cada capa “Conv2D”, se agrega una capa “MaxPooling2D” con un tamaño de ventana de (2, 2) para realizar el max pooling y reducir la dimensionalidad de las características.

En las capas Flatten y Dense, después de las capas de convolución y de max pooling, se agrega una capa Flatten para aplanar las características en un vector unidimensional. Luego, se agrega una capa Dense con 512 neuronas y función de activación 'relu', seguida de otra capa Dense con 21 neuronas (correspondientes a las clases de salida) y función de activación 'softmax'. La capa Dense realiza la clasificación final de las características extraídas.

Para la compilación del modelo se utiliza el método compile() para compilar el modelo. Aquí se especifica el optimizador 'adam', que es un algoritmo popular para la optimización de redes neuronales. La función de pérdida se establece en 'categorical_crossentropy', que es comúnmente utilizada en problemas de clasificación de varias clases. Se especifica también la métrica 'accuracy' para evaluar el rendimiento del modelo durante el entrenamiento y la evaluación.

El modelo consta de varias capas convolucionales y de max pooling, seguidas de capas Dense para la clasificación final. El modelo se compila con un optimizador, una función de pérdida y una métrica de evaluación especificados.

```
from tensorflow.keras.layers import MaxPooling2D

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(512, activation='relu'),
    Dense(21, activation='softmax')
])

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Fig.9 - Definición y compilación de un modelo de red neuronal convolucional.

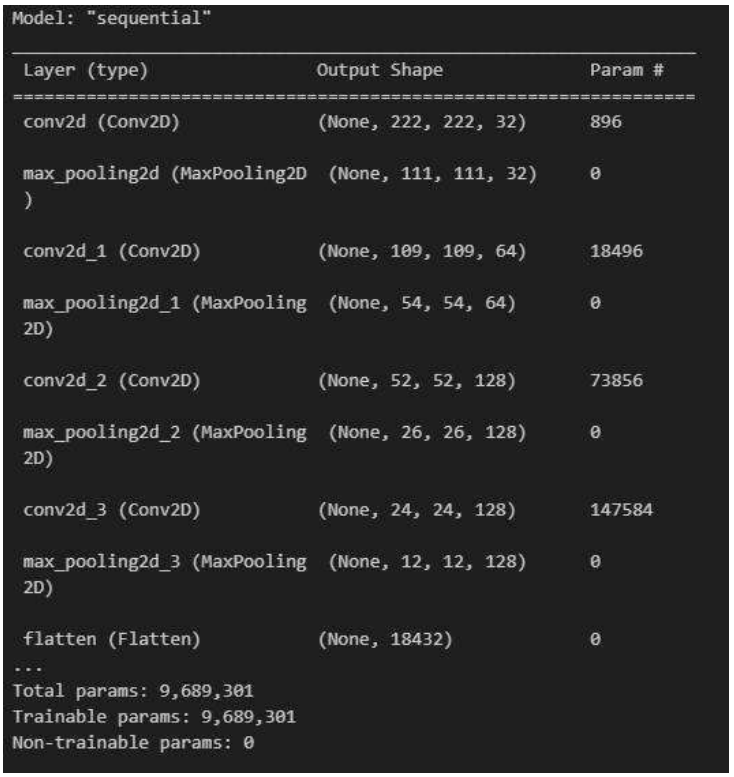

Fig.10 - Resumen de cómo está compuesto el modelo.

4.10. Entrenamiento del modelo:

En el siguiente apartado , el código muestra una parte del proceso de entrenamiento de un modelo de aprendizaje automático utilizando la función fit() de Keras.

La primera línea inicia el entrenamiento del modelo. “x_train” representa los datos de entrenamiento y “y_train” representa las etiquetas correspondientes a esos datos. El argumento epochs=21 especifica el número de veces que el modelo iterará sobre todo el conjunto de entrenamiento. Podemos ver cómo, en cada fila se pasa cada lote, en este caso son 21 capas. Tenemos varios valores por cada etapa, el tiempo que ha tomado por cada iteración, el valor de la función pérdida obtenida durante la iteración, “loss” y “accuracy” que nos indica la precisión alcanzada por el modelo en esa iteración. El objetivo del entrenamiento es ajustar los pesos del modelo de manera que la pérdida se minimice y la precisión se maximice en los datos de entrenamiento.

```
model.fit(x_train, y_train, epochs=21)
```

```
Epoch 1/21
1/1 [==============================] - 1s 928ms/step - loss: 1.4701 - accuracy: 0.6562
Epoch 2/21
1/1 [==============================] - 1s 920ms/step - loss: 1.3749 - accuracy: 0.4688
Epoch 3/21
1/1 [==============================] - 1s 904ms/step - loss: 1.0001 - accuracy: 0.7500
Epoch 4/21
1/1 [==============================] - 1s 914ms/step - loss: 0.8768 - accuracy: 0.6562
Epoch 5/21
1/1 [==============================] - 1s 869ms/step - loss: 0.5461 - accuracy: 0.9375
Epoch 6/21
```

Fig.11 - Definición y compilación de un modelo de red neuronal convolucional.

4.11. Validación del modelo:

Siguiendo nuestro código hemos hecho dos gráficas para validar que el modelo esté bien entrenado. Estas gráficas nos indican que nivel de “accuracy” y “validation loss” tenemos al final de todo.

```
results = model.evaluate(y_test, verbose=0)

print("    Test Loss: {:.5f}".format(results[0]))
print("Test Accuracy: {:.2f}%".format(results[1] * 100))
```

```
    Test Loss: 0.52093
Test Accuracy: 87.27%
```

Fig.12 - Gráficas de validación de la exactitud y pérdida del modelo.

Como se puede ver en este código, se obtiene una accuracy del 87%. Sin tener precedentes para la validación de este dato, nos parece suficiente para continuar con el entrenamiento. Será la experiencia la que nos dirá si este porcentaje es válido o hay que reconfigurar el modelo.

Como se puede ver en la siguiente gráfica tenemos, la línea de “validation” y de training, al final lo que podemos ver es el eje X que corresponde al número de veces que se iterará el modelo, los “epoch” que en este caso tenemos 21. El eje Y que corresponde al valor que se ve representado por “loss” o “validation”. Según las gráficas podemos ver que la línea azul corresponde al modelo con los datos de train y la roja con los datos de validación. Donde vemos un alto valor de “accuracy” qué significa exactitud, por lo que el modelo es capaz de reconocer las letras a partir de las imágenes de los gestos, con buena seguridad. En la segunda gráfica podemos ver las líneas invertidas, lo cual es bueno porque

lo que queremos es el menor valor de “loss” posible. Un valor bajo de pérdida significa que el modelo está haciendo predicciones precisas y que hay poca discrepancia entre las salidas predichas y las salidas reales. Esto es deseable, ya que indica que el modelo ha aprendido patrones importantes en los datos y puede generalizar bien para hacer predicciones en nuevos datos.

Gracias a estas gráficas podemos pensar que el modelo está correctamente entrenado por lo que procedemos a desarrollar la aplicación y comprobar que funciona correctamente, ya que nuestro objetivo es probarlo con una webcam.

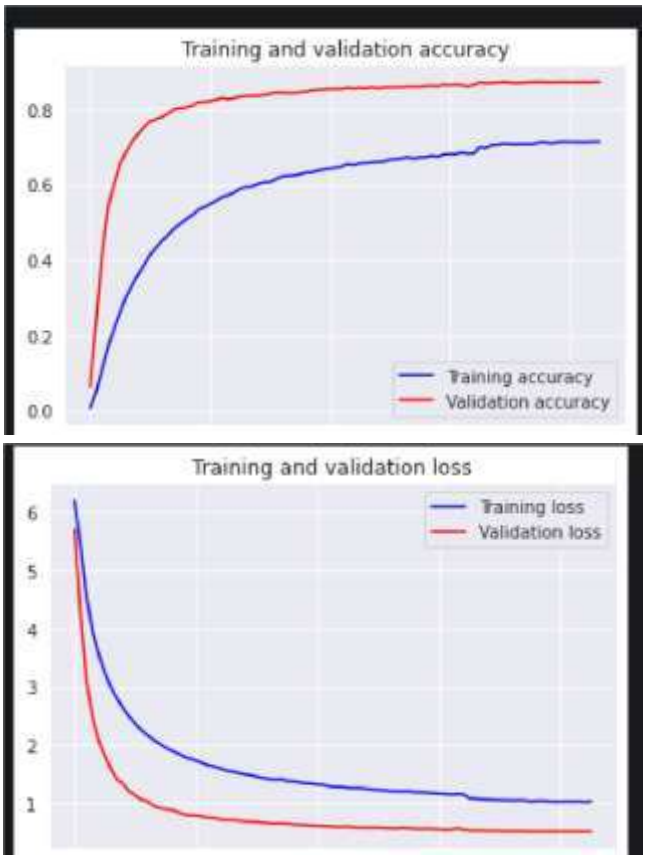

Fig.13 - Gráficas de validación de la exactitud y pérdida del modelo.

Por último en este programa, solo nos queda guardar el modelo para así poder utilizarlo posteriormente en la aplicación.

```
model.save('keras_model.h5')
```

Fig.14 - Función para guardar y exportar el modelo creado.

4.12. Desarrollo de la aplicación, estructura de la aplicación:

Para el desarrollo de la aplicación, se ha utilizado la librería OpenCV, como habíamos visto anteriormente. Para este programa se ha hecho un código que primero accede a la cámara del ordenador y detecta la mano del usuario, gracias a la librería anteriormente dicha. Después, compara el signo del usuario con el modelo de Keras que se ha creado en el punto anterior.

Primero, procedemos a importar las bibliotecas necesarias, que en este caso corresponden a OpenCV("cv2"), con sus respectivas funciones de "HandDetector" y "Classifier". Hand Detector es una clase proporcionada por la biblioteca cvzone que se basa en OpenCV y ofrece funcionalidades para detectar y rastrear manos en imágenes o videos. Utiliza un modelo de aprendizaje automático para reconocer y localizar las manos en una imagen o video. En el código, se utiliza para detectar las manos en los fotogramas de vídeo capturados y proporcionar información sobre la ubicación y el seguimiento de las manos detectadas. Classifier es otra clase proporcionada por la biblioteca cvzone que se utiliza para clasificar objetos o gestos en imágenes utilizando modelos de aprendizaje automático preentrenados. En este código, se utiliza un clasificador pre-entrenado para reconocer y clasificar los gestos realizados con la mano en letras. El clasificador toma como entrada una imagen recortada de la mano y devuelve la etiqueta de clase y el índice correspondiente a la letra reconocida.

Se configura la cámara mediante la creación de un objeto VideoCapture para acceder al video en tiempo real de la cámara predeterminada. Se inicializan los detectores, HandDetector y Classifier. A HandDetector le pasamos el parámetro de una sola mano, para que no detecte las dos cámaras al mismo tiempo, aunque se podría hacer y no daría problemas. Se definen variables importantes, como el desplazamiento (offset), el tamaño de la imagen (imgSize), un contador y una lista de frases. La variable "offset" se refiere a un valor que se utiliza para agregar un margen alrededor de la región de interés de la mano. Este margen asegura que la mano esté completamente visible en la imagen recortada, permitiendo así un mejor procesamiento y detección de los gestos realizados. La variable "imgSize" determina el tamaño al que se redimensionarán las imágenes recortadas de la mano. Establecer un tamaño fijo garantiza la consistencia en el procesamiento de las imágenes y ayuda a mantener la relación de aspecto original de la mano durante el redimensionamiento. La variable "phrases" es una lista que se utiliza para almacenar las frases generadas por los gestos de la mano. Cada vez que se completa una frase, se agrega a esta lista. Esta variable permite mantener un registro de las frases generadas a medida que se reconocen los gestos y se captura el texto correspondiente. Las frases almacenadas en esta lista se pueden mostrar posteriormente en la imagen de salida o utilizar para otros fines en la implementación del sistema.

```
cap = cv2.VideoCapture(0) # ID de la cámara
detector = HandDetector(maxHands=1)
classifier = Classifier(r"C:\Users\jesus\Desktop\tfg_signos\Model\tres_keras_model.h5",
                        r"C:\Users\jesus\Desktop\tfg_signos\Model\tres_keras_model.txt")
offset = 20
imgSize = 300
counter = 0
phrases = []
current_phrase = ""
```

Fig.15 - Definición de los parámetros de la App.

Además, se puede ver como se han cargado el modelo de Keras, entrenado para la clasificación de imágenes y los labels del modelo. Los label, son el valor de cada clase, en nuestro caso, las letras del abecedario, vemos como está en tipo“txt”, un archivo de texto.

Se inicia un bucle infinito para capturar y procesar continuamente los fotogramas de vídeo. En cada iteración del bucle, se captura un fotograma del video utilizando la función cap.read(). El resultado se almacena en la variable img. En este caso si la variable de imagen está vacía se rompe el bucle. La línea de código "imgOutput = img.copy()" crea una copia de la imagen capturada o procesada en el momento actual y la almacena en la variable "imgOutput". Al realizar una copia de la imagen, se crea una nueva instancia de la imagen que es independiente de la imagen original. Esto significa que cualquier modificación realizada en la variable "imgOutput" no afectará a la imagen original "img" y viceversa.

```
while True:
    success, img = cap.read()
    if img is None:
        break
    imgOutput = img.copy()
```

Fig.16 - Inicio del bucle de la app.

```
hands, img = detector.findHands(img)
if hands:
    hand = hands[0]
    x, y, w, h = hand['bbox']

    imgWhite = np.ones((imgSize, imgSize, 3), np.uint8) * 255
    imgCrop = img[y - offset:y + h + offset, x - offset:x + w + offset]

    imgCropShape = imgCrop.shape

    aspectRatio = h / w

    if aspectRatio > 1:
        k = imgSize / h
        wCal = math.ceil(k * w)
        imgResize = cv2.resize(imgCrop, (wCal, imgSize))
        imgResizeShape = imgResize.shape
        wGap = math.ceil((imgSize - wCal) / 2)
        imgWhite[:, wGap:wCal + wGap] = imgResize
        prediction, index = classifier.getPrediction(imgWhite, draw=False)
        print(prediction, index)

    else:
        k = imgSize / w
        hCal = math.ceil(k * h)
        imgResize = cv2.resize(imgCrop, (imgSize, hCal))
        imgResizeShape = imgResize.shape
        hGap = math.ceil((imgSize - hCal) / 2)
        imgWhite[hGap:hCal + hGap, :] = imgResize
        prediction, index = classifier.getPrediction(imgWhite, draw=False)
```

Fig.17 - Código de procesamiento de las manos.

Este fragmento de código se encarga de procesar la imagen de la mano detectada y realizar la clasificación de los gestos realizados por la mano.

La función “findHands()” del objeto detector se utiliza para detectar y rastrear las manos en la imagen img. Retorna una lista de diccionarios que representan las manos detectadas, y también actualiza la imagen img con las anotaciones visuales de las manos detectadas. La lista de manos detectadas se asigna a la variable hands.

El fragmento “if hands:” verifica si se han detectado manos. Si la lista hands no está vacía (es decir, si se han detectado manos), se procede a procesar la primera mano detectada.

En “hand = hands[0]:”, se obtiene la primera mano detectada de la lista hands y se almacena en la variable hand. En este código, sólo se procesa una mano a la vez, por lo que se toma la primera mano de la lista.

Seguido, “x, y, w, h = hand['bbox']:” es donde se extraen las coordenadas del cuadro delimitador (bounding box) de la mano detectada. Estas coordenadas representan la posición (x, y) y el tamaño (ancho, alto) del cuadro que rodea la mano.

El trozo código de “imgWhite = np.ones((imgSize, imgSize, 3), np.uint8) * 255:” se usa para crear una imagen en blanco “imgWhite” utilizando la biblioteca numpy. Esta imagen se utiliza como lienzo para procesar la región de interés de la mano y se establece en color blanco.

La línea de “imgCrop = img[y - offset:y + h + offset, x - offset:x + w + offset]:” es para recortar la región de interés de la mano de la imagen original img utilizando las coordenadas del cuadro delimitador. Se agrega un margen (offset) alrededor del cuadro para asegurarse de que se capture toda la mano. La región recortada se asigna a la variable “imgCrop”.

La línea de “aspectRatio = h / w:” es para calcular la relación de aspecto de la mano dividiendo la altura (h) entre el ancho (w). Esto ayuda a determinar si la mano está en posición vertical u horizontal.

La condición “if aspectRatio > 1:”, si la relación de aspecto es mayor que 1, significa que la mano está en posición vertical.

El código de “k = imgSize / h:” es para calcular el factor de escala k dividiendo el tamaño de la imagen (imgSize) entre la altura de la región de interés (h). Este factor de escala se utiliza para redimensionar la imagen de la mano proporcionalmente y mantener su relación de aspecto original.

El cálculo de “wCal = math.ceil(k * w):” se utiliza para calcular el ancho redimensionado (wCal) multiplicando el factor de escala k por el ancho original (w) y redondeando hacia arriba utilizando la función math.ceil(). Esto garantiza que el ancho redimensionado sea un número entero.

En “imgResize = cv2.resize(imgCrop, (wCal, imgSize)):” se redimensiona la imagen recortada “imgCrop” al nuevo tamaño utilizando la función resize() de OpenCV. El ancho redimensionado (wCal) y el tamaño de la imagen (imgSize) se pasan como parámetros.

Con “wGap = math.ceil((imgSize - wCal) / 2):” se calcula el espacio adicional (wGap) que se agregará a los lados izquierdo y derecho de la imagen redimensionada para que tenga el tamaño completo (imgSize). Esto se calcula restando el ancho redimensionado del tamaño de la imagen y dividiendo el resultado entre 2, y luego redondeando hacia arriba.

Con “imgWhite[:, wGap:wCal + wGap] = imgResize:” Se copia la imagen redimensionada imgResize en la imagen en blanco “imgWhite”, en la posición correspondiente para que se ajuste al ancho completo de la imagen. Los píxeles se copian en la columna desde wGap hasta wCal + wGap.

Al final en “prediction, index = classifier.getPrediction(imgWhite, draw=False):” se utiliza el clasificador “classifier” para predecir el gesto realizado por la mano representada en la imagen “imgWhite”. Se pasa la imagen redimensionada como entrada a la función “getPrediction()”, que devuelve la etiqueta de clase predicha (prediction) y el índice de la etiqueta (index).

Y por último, “print(prediction, index):”, imprime en la consola la etiqueta de clase predicha y el índice correspondiente.

El código también contiene un bloque else que se ejecuta si la relación de aspecto es menor o igual a 1, lo cual indica que la mano está en posición horizontal. En ese caso, se realiza un proceso similar de redimensionamiento y se obtiene la predicción del gesto utilizando el ancho redimensionado y el alto original de la imagen de la mano. Este fragmento de código recorta y redimensiona la región de interés de la mano detectada, y luego utiliza un clasificador para predecir el gesto realizado por esa mano.

4.13. Implementación de las frases e impresión en pantalla:

```
        cv2.rectangle(imgOutput, (x - offset, y - offset - 50),
                      (x - offset + 90, y - offset - 50 + 50), (255, 0, 255), cv2.FILLED)
        cv2.putText(imgOutput, labels[index], (x, y - 26), cv2.FONT_HERSHEY_COMPLEX, 1.7, (255, 255, 255), 2)
        cv2.rectangle(imgOutput, (x - offset, y - offset),
                      (x + w + offset, y + h + offset), (255, 0, 255), 4)

        if cv2.waitKey(1) & 0xFF == ord('v'):  # Espera a que se presione la tecla "v"
            current_phrase += labels[index]

        if cv2.waitKey(1) & 0xFF == ord(' '):  # Espera a que se presione la tecla "espacio"
            current_phrase += " "  # Agrega un espacio a la frase actual

        if cv2.waitKey(1) & 0xFF == ord('s'):  # Espera a que se presione la tecla "s" para guardar la frase actual
            phrases.append(current_phrase)
            current_phrase = ""

    # Dibujar frases en la imagen
    y_offset = 100
    for phrase in phrases:
        cv2.putText(imgOutput, phrase, (50, y_offset), cv2.FONT_HERSHEY_COMPLEX, 1.5, (0, 0, 255), 2)
        y_offset += 50

    cv2.putText(imgOutput, current_phrase, (50, 50), cv2.FONT_HERSHEY_COMPLEX, 2, (0, 0, 255), 3)
    cv2.imshow("Image", imgOutput)
    cv2.waitKey(1)
```

Fig.18 - Código que se encarga de formar las frases y mostrar la imagen por pantalla.

Este fragmento de código se encarga de visualizar los resultados y realizar acciones en función de las teclas presionadas. A continuación se explica cada parte:

La parte que se encarga de visualizar los resultados:

En la línea de “cv2.rectangle(imgOutput, (x - offset, y - offset - 50), (x - offset + 90, y - offset - 50 + 50), (255, 0, 255), cv2.FILLED):” se dibuja un rectángulo relleno en la parte superior izquierda de la imagen de salida (imgOutput). Este rectángulo se utiliza para mostrar la etiqueta de clase predicha en el gesto de la mano.

En “cv2.putText(imgOutput, labels[index], (x, y - 26), cv2.FONT_HERSHEY_COMPLEX, 1.7, (255, 255, 255), 2):” se muestra la etiqueta de clase predicha en la imagen de salida. La etiqueta se coloca cerca del cuadro delimitador de la mano, utilizando las coordenadas (x, y - 26). Se especifica el tamaño y el estilo de la fuente.

En “cv2.rectangle(imgOutput, (x - offset, y - offset), (x + w + offset, y + h + offset), (255, 0, 255), 4):” se dibuja un rectángulo alrededor del cuadro delimitador de la mano en la imagen de salida. Este rectángulo resalta la región de la mano detectada.

En esta parte el código se encarga de escribir las frases:

En la primera condición “if cv2.waitKey(1) & 0xFF == ord('v'):” se verifica si la tecla "v" ha sido presionada. Si es así, se agrega la etiqueta de clase actual al final de la frase actual (current_phrase).

En la segunda condición, “if cv2.waitKey(1) & 0xFF == ord(' '):” se verifica si la tecla "espacio" ha sido presionada. Si es así, se agrega un espacio en blanco a la frase actual.

En la última condición, “if cv2.waitKey(1) & 0xFF == ord('s'):” se verifica si la tecla "s" ha sido presionada. Si es así, se guarda la frase actual en la lista de frases (phrases) y se reinicia la frase actual (current_phrase) para comenzar una nueva.

Al final, “y_offset = 100:” inicializa un desplazamiento vertical para mostrar las frases en la imagen de salida.

El bucle “for phrase in phrases:” itera sobre cada frase almacenada en la lista de frases (phrases).

Para mostrar las frases se usa “cv2.putText(imgOutput, phrase, (50, y_offset), cv2.FONT_HERSHEY_COMPLEX, 1.5, (0, 0, 255), 2):” que muestra cada frase en la imagen de salida en una posición específica (50, y_offset). Las frases se muestran con un tamaño y estilo de fuente definidos.

Con “y_offset += 50:” se incrementa el desplazamiento vertical para la siguiente frase a mostrar.

Aquí es para mostrar multiples frases, “cv2.putText(imgOutput, current_phrase, (50, 50), cv2.FONT_HERSHEY_COMPLEX, 2, (0, 0, 255), 3):” que muestra la frase actual en la esquina superior izquierda de la imagen de salida. La frase se muestra con un tamaño y estilo de fuente más grandes.

Al final, “cv2.imshow("Image", imgOutput):” muestra la imagen de salida en una ventana titulada "Image".

Para finalizar, “cv2.waitKey(1):” espera un breve tiempo para detectar las pulsaciones de teclas. El valor 1 indica que se espera 1 milisegundo.

Con esto el código ya funciona correctamente, el código permite detectar las manos, encuadrarlas, mostrar a qué letra corresponde el signo e imprimir en pantalla las mismas, así pues se podrán formar las frases. Ahora quedará ver los resultados y experimentos, para comprobar que se han podido alcanzar los objetivos propuestos.

Capítulo 5:

Experimentos y resultados

En el capítulo 5 se hace el análisis de los resultados obtenidos, comprobando si el código ha funcionado y si se ha obtenido los resultados esperados.

5.1. Pruebas y análisis de los resultados obtenidos

Como último paso, se realizó una prueba experimental para verificar el resultado del proceso descrito en el Trabajo de Fin de Grado (TFG).

Cabe recalcar, que para obtener este resultado, se han hecho múltiples pruebas en las que no se ha obtenido conseguir la frase, para solucionar esto se ha tenido que reentrenar el modelo de clasificación de imágenes y comprobar la calidad de las fotos de las clases. En algunas ocasiones la letra “N”, se detectaba como “M”, esto se debía que a la hora de obtener las fotos, se habían detectado mal, por lo que se tuvo que limpiar la calidad de esos datos y eliminar esos datos. Al final, cuando los datos de entrenamiento no son de buena calidad o no representan adecuadamente el problema que se desea resolver, se puede decir que los datos son deficientes o insuficientes para entrenar el modelo de manera efectiva. Esto puede resultar en un mal rendimiento del modelo durante el entrenamiento y, en última instancia, en un modelo con una baja precisión.

El experimento ha sido exitoso, ya que se logró imprimir la frase "Hola Mundo". Esto demuestra que el sistema de reconocimiento de gestos de la mano implementado en el Trabajo de Fin de Grado (TFG) es capaz de capturar y procesar correctamente los gestos realizados por la mano para generar texto, y valida el porcentaje de precisión de 87% conseguido. La detección y clasificación de los gestos se realizó de manera precisa, permitiendo la formación de la frase deseada. Este resultado valida la eficacia y funcionalidad del sistema propuesto, lo que respalda la viabilidad de su implementación en aplicaciones prácticas de entrada de texto basadas en la signatura.

Fig.19 - Función para guardar y exportar el modelo creado.

Capítulo 6:

Conclusiones

6.1. Conclusiones Finales

Este proyecto es un modesto intento de contribuir a la sociedad y marcar una pequeña pero significativa diferencia en la vida de las personas sordas. A través de la creación de un programa que facilite la comunicación entre personas sordas y aquellas que no conocen el LSE, esperamos fomentar la inclusión y promover la igualdad de oportunidades.

Al concluir el TFG, es apropiado realizar una breve reflexión sobre los objetivos planteados al inicio del proyecto y considerar posibles aplicaciones para la solución desarrollada.

A través de la implementación del clasificador de imágenes, se logró comprender el funcionamiento de la inteligencia artificial, en particular las redes convolucionales, así como su implementación mediante el uso de diversas herramientas y bibliotecas enfocadas en la ciencia de datos y la inteligencia artificial. Estas herramientas y bibliotecas son accesibles y fáciles de utilizar para cualquier usuario con conocimientos básicos en el tema.

La integración del modelo en una aplicación de Python demuestra la versatilidad que ofrece esta tecnología, al permitir a los usuarios utilizar la aplicación con solo disponer de una cámara sencilla en su ordenador. Esto resalta la capacidad de adaptación de la inteligencia artificial y su aplicación práctica en diferentes escenarios, proporcionando una experiencia accesible y conveniente para los usuarios.

Se procederá a comparar los objetivos previstos al principio directamente:

- **Documentación y estudio:**
  Se ha cumplido con el estudio y documentación durante el desarrollo del TFG.

- **Creación del clasificador de imágenes:**
  Se ha creado con éxito un clasificador de imágenes gracias a Tensorflow y Keras. Además, hemos alcanzado una precisión adecuada, de 87%, para la comunicación fluida.

- **Creación de la aplicación:**
  Se ha podido crear una aplicación conceptual, que nos permite validar el modelo. Utiliza simplemente una webcam para reconocer la mano y los signos.

- **Experimentos y resultados acordes a lo esperado:**
  Se ha conseguido el resultado, predefinido en los objetivos, crear una frase como “Hola Mundo”.

Por lo que podemos concluir, este trabajo ha cumplido con los objetivos previstos, se ha conseguido desarrollar la aplicación para que detecte en tiempo real los gestos y así pues

sean traducidos, permitiendo formar frases en tiempo real. Un programa fácil de usar, al alcance de cualquiera que disponga de un ordenador.

Capítulo 7:

Bibliografía

[1]Ayala, S. R. (2016, 12 mayo). SignAloud: guantes que traducen lenguaje de señas a audio - Tecnius. Tecnius.

https://www.tecnius.cl/index.php/2016/05/12/signaloud-guantes-que-traducen-lenguajede-senas-a-audio/

[2]Sign Speak. (s. f.). Sign Speak. https://www.sign-speak.com/

[3]Diego de la Torre, "¿Cómo nació la Inteligencia Artificial? (2018), ThinkBig https://blogthinkbig.com/historia-como-nacio-inteligencia-artificial

[4]Rockwell Anyoha, "The History of Artificial Intelligence" (28 de agosto de 2017), SITN Harvard University https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/

[5]Brett Grossfeld, "Aprendizaje profundo y aprendizaje automático: una forma sencilla de entender la diferencia" (23 de enero de 2020), Blog de Zendesk https://www.zendesk.es/blog/machine-learning-and-deep-learning/

[6]Izaurieta, F., & Saavedra, C. (2000). Redes neuronales artificiales. Departamento de Física, Universidad de Concepción Chile.

[7]Caparrini, F. S. (s. f.). Redes Neuronales: una visión superficial - Fernando Sanch Caparrini. http://www.cs.us.es/~fsancho/?e=72

[8]What is Computer Vision? | IBM. (s. f.). https://www.ibm.com/topics/computer-vision

[9]Sanghvi, K. (2023, 21 abril). Image Classification Techniques - Analytics Vidhya - Medium. https://medium.com/analytics-vidhya/image-classification-techniques-83fd87011cac

[10]TensorFlow. (s. f.). TensorFlow.

https://www.tensorflow.org/

[11]Team, K. (s. f.). Keras documentation: Keras API reference. https://keras.io/api/

[12]OpenCV documentation index. (s. f.). https://docs.opencv.org/

[13]Project Jupyter Documentation — Jupyter Documentation 4.1.1 alpha documentation. (s. f.).

https://docs.jupyter.org/en/latest/

Código del TFG: jesusllorens79/TFG-signos (github.com)