**INSTITUTO TECNOLÓGICO DE CHIHUAHUA II**

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

# TRADUCTOR DE LENGUAJE DE SEÑAS MEXICANO EN TIEMPO REAL

TESIS

PARA OBTENER EL GRADO DE

**MAESTRO EN SISTEMAS COMPUTACIONALES**

PRESENTA

JESÚS MARTÍN CARRASCO JÁQUEZ

DIRECTOR DE TESIS

M.I.S.C. JESÚS ARTURO ALVARADO GRANADINO

CODIRECTOR DE TESIS

M.C. ARTURO LEGARDA SÁENZ

CHIHUAHUA, CHIH., DICIEMBRE 2023

# Dictamen

Chihuahua, Chihuahua,26 de enero 2024

**M.C. MARIA ELENA MARTINEZ CASTELLANOS**

**COORDINADORA DE POSGRADO E INVESTIGACION.**

**PRESENTE**

Por medio de este conducto el comité tutorial revisor de la tesis para obtención de grado de Maestro en Sistemas Computacionales, que lleva el nombre de: ***"TRADUCTOR DE LENGUAJE DE SEÑAS MEXICANO EN TIEMPO REAL"***, que presenta el C. Jesús Martin Carrasco Jaquéz, hace de su conocimiento que después de ser revisado ha dictaminado la APROBACIÓN de la misma.

Sin otro particular de momento queda de usted.

Atentamente

La Comisión de Revisión de Tesis.

A. Alvarado G.

M.I.S.C. JESUS ARTURO ALVARADO GRANADINO

Director de tesis

M.C. ARTURO LEGARDA SÁENZ

Co –Director

DR. HERNAN DE LA GARZA GUTIERREZ

Revisor

Leonardo Nevárez Chávez

M.C. LEONARDO NEVÁREZ CHÁVEZ

Revisor

# AGRADECIMIENTOS

Quisiera empezar por agradecer a mis padres, los cuales siempre me han brindado todo el apoyo en mi educación y motivarme cada día para superarme ante las dificultades, siendo ambos mi modelo a seguir de dedicación y perseverancia. A mi padre, Martín Carrasco, que ha sido mi soporte de fortaleza, dándome la confianza de enfrentar desafíos con valentía. A mi madre, Guadalupe Jáquez, que es mi fuente de sabiduría, que ha lidiado con mis momentos de debilidad, dando sus mejores consejos.

A mi hermana, Elizabeth Hermosillo, celebro este logro contigo, porque sé que tu amor y apoyo han sido una parte fundamental de mi éxito. Agradezco tus palabras alentadoras, tus abrazos reconfortantes y tu presencia constante.

A mi pareja, Georgina Sánchez, que me ha apoyado con el aprendizaje de la lengua de señas mexicana, quiero expresar mi profundo agradecimiento por tu amor y apoyo incondicional a lo largo de esta travesía, por estar conmigo en mis desvelos a la hora de escribir esta tesis, entrar a las clases en línea y a la lectura y corrección de este documento. Tu presencia ha sido mi mayor fuente de inspiración y consuelo.

También quiero agradecer a mi asesor de tesis, Jesús Arturo Alvarado, su experiencia y orientación han sido fundamentales para alcanzar este logro académico. Aprecio enormemente el tiempo que ha dedicado a revisar mi trabajo, ofrecer valiosas sugerencias y brindarme una perspectiva experta.

Por último, a mi casa de estudios, el Instituto Tecnológico de Chihuahua II, que a parte de ser donde he realizado este posgrado, es también donde he realizado la licenciatura, estoy agradecido por el acceso a recursos, la inspiración proporcionada por el currículo y, sobre todo, por el respaldo continuo que he recibido a lo largo de mi trayectoria. Me siento afortunado de haber sido parte de esta comunidad educativa excepcional.

## RESUMEN

En el presente documento, se exponen los resultados obtenidos en el proyecto para detectar señas del abecedario del Lenguaje de Señas Mexicana (LSM), esto a través de imágenes en formato RGB. Se emplearon algoritmos para la eliminación de fondo, y la extracción de características con el método de Histograma de Gradientes Orientados (HOG por sus siglas en inglés) como datos de entrada para la creación del modelo. Se usó el clasificador de aprendizaje k-Vecinos Más Cercanos (k-NN) como modelo, teniendo como resultado un 95% de precisión.



## ABSTRACT

In this document, the results obtained in the development of a technology fordetecting signs of the Mexican Sign Language (LSM) alphabet are presented. This is done through RGB format images. Algorithms were used for background removal and feature extraction using the Histogram of Oriented Gradients (HOG) method as input data for modelcreation. The k-Nearest Neighbors (k-NN) learning classifier was employed as the model, resulting in a 95% accuracy.

# ÍNDICE

# TABLA DE FIGURAS

# TABLA DE FRAGMENTOS DE CÓDIGO

# 1 INTRODUCCIÓN

Según datos de la INEGI del 2018, la población total en México era de 125.2 millones de personas. De ese total, 29.3 millones eran niños y adolescentes de 5 a 17 años, que corresponde al 23.4% del total, y de los cuales el 2% (580,289) entran dentro del espectro de personas con discapacidad. El 11.3% (65,592) de esos niños y adolescentes presentaban alguna discapacidad auditiva. (INEGI, 2020). El presente proyecto va encaminado a apoyar ese sector de la población.

La comunidad de personas con discapacidad auditiva se apoya para su inclusión en la sociedad, en la utilización de la Lengua de Señas Mexicana (LSM). El Gobierno Federal de México define a la LSM, como "la lengua que utilizan las personas sordas en México, que posee su propia sintaxis, gramática y léxico." (Gobierno de México, 2016).

La LSM, consiste en los siguientes elementos de lenguaje: la dactilología y los ideogramas. La dactilología, consiste en la representación secuencial de las letras de una palabra mediante señas, lo equivalente al deletreo de las palabras que se tiene en el lenguaje hablado. Por otra parte, los ideogramas "representan una palabra con una o varias configuraciones de mano". (Serafín & Gónzález, 2011).

En el ámbito digital, se identificó la necesidad de desarrollar herramientas para facilitar la inclusión de aquellas personas que manejan la LSM, y tener una sinergia con aquellas que no tienen este conocimiento. En el presente documento se exponen los resultados obtenidos en un proyecto de desarrollo tecnológico, realizado en el Instituto Tecnológico Nacional de México, Campus Chihuahua II. En este proyecto, se contrastan los resultados entre el uso de Redes Neuronales Convolucionales (en inglés Convolutional Neuronal Network, abreviado CNN) y el clasificador de aprendizaje k-Vecinos más próximos (abreviado k-NN, del inglés k-nearest neighbors), para realizar un modelado de la dactilología de la LSM. En las pruebas se obtuvo un porcentaje del 75%, para el primero y, un 95% de exactitud para el segundo.

Actualmente existen propuestas para traducir en tiempo real el lenguaje a señas; difiere principalmente en que cada una emplea el lenguaje a señas correspondientes al país en donde se desarrolló, contrastando con México, donde se van inciando con propuestas. En

este proyecto se muestran los resultados obtenidos con el clasificador de aprendizaje k-Vecinos más cercanos (abreviado k-NN, del inglés k-nearest neighbors), aplicados para modelar la dactilología de la LSM. En las pruebas se obtuvo una exactitud del 95%.

Las imágenes de las señas en el modelado anteriormente mencionado fueron procesadas aplicando una técnica de visión computacional de extracción de características de imágenes llamada HOG (del inglés Histogram of Oriented Gradients). Esta técnica se utiliza para extraer características clave de las imágenes y reconocer patrones dentro del conjunto de datos.

## 1.1 Planteamiento del problema

Dada a las circunstancias generadas a la contingencia sanitaria por el COVID 19 a nivel mundial, las conferencias a través de medios digitales han ido en aumento, desde clases en línea, congresos, juntas de trabajo, por mencionar algunas; sin embargo, hay cierta cantidad de población con discapacidad auditiva, siendo su principal comunicación el Lenguaje de Señas Mexicana. Actualmente existen propuestas para traducir el lenguaje a señas, difiriendo principalmente en que cada una emplea el lenguaje a señas correspondiente al país de desarrollo. Contrastando con México, no se tiene un sistema similar para el Lenguaje a Señas Mexicana.

## 1.2 Objetivo general

Desarrollar un software, el cual tendrá como visión la transcripción a texto del Lenguaje de Señas Mexicano. A través de este sistema se busca proporcionar herramientas digitales para la comunicación de las personas con discapacidad auditiva en el ámbito del internet.

## 1.3 Objetivos específicos

- Arquitectura del Software. Con base en la información de la INEGI, estructurar las principales clases y métodos potenciales a utilizar.
- Diseño de interfaz. Plantear las posibles ventanas finales, con las que el usuario final interactuará.
- Desarrollo. Programar el código fuente y la codificación del front-end planteado.
- Implementación. Proceder con la puesta en funcionamiento del proyecto para que los usuarios finales interactúen con él.
- Entrega de resultados. Registrar los resultados de la implementación.

# 2 MARCO TEÓRICO

Dentro de esta sección, se exponen a detalle las definiciones relevantes y antecedentes para el desarrollo de este proyecto.

## 2.1 Inteligencia Artificial

### 2.1.1 Conceptos

El aprendizaje profundo se define como un subcampo del aprendizaje automático que se centra en la extracción de características fundamentales de los datos de entrada, especialmente en situaciones en las que los datos presentan una complejidad notable (Gollapudi, 2019).

Las redes neuronales son un modelo computacional que comparten ciertas similitudes con el funcionamiento del cerebro animal. En este enfoque, múltiples unidades simples operan en paralelo sin depender de una unidad de control centralizada. (Patterson & Gibson, Deep Learning, 2017).

Están compuestas normalmente por

- Número de neuronas.
- Número de capas.
- Conexión entre capas.

La función Sigmoide (Figura 2.1) puede describirse como una versión suavizada de una función escalón y, por lo tanto, es diferenciable. Esta ecuación es beneficiosa al transformar cualquier valor en probabilidades y puede ser empleada en casos de clasificación binaria. La función mapea la entrada a un valor que se encuentra entre 0 y 1. (Shanmugamani, 2018).

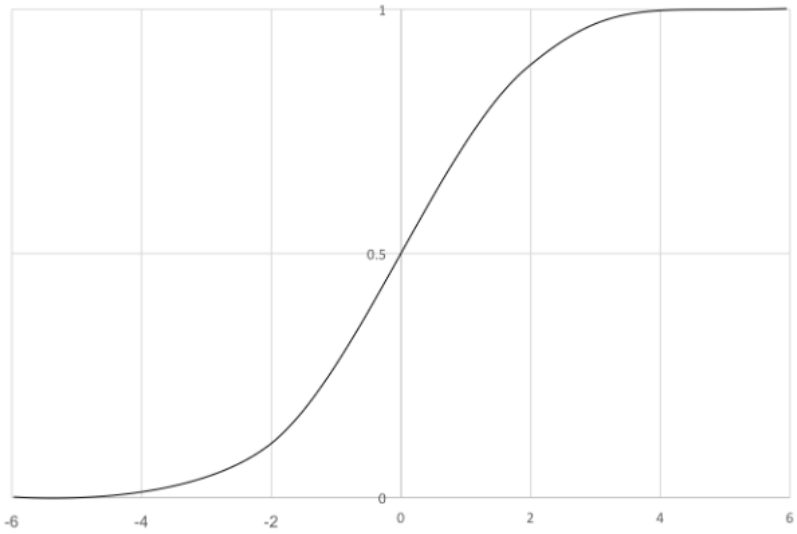


*Figura 2.1 Gráfica de la función Sigmoide. Fuente: (Shanmugamani, 2018).*

En cambio, la función lineal rectificada (ReLU) trata de una transformación intrigante que activa un nodo únicamente cuando la entrada supera un valor determinado. Mientras la entrada sea inferior a cero, la salida se mantiene en cero. Sin embargo, cuando la entrada rebasa un umbral específico, se establece una relación lineal con la variable dependiente f(x) = max(0, x). (Patterson & Gibson, Deep Learning, 2017).

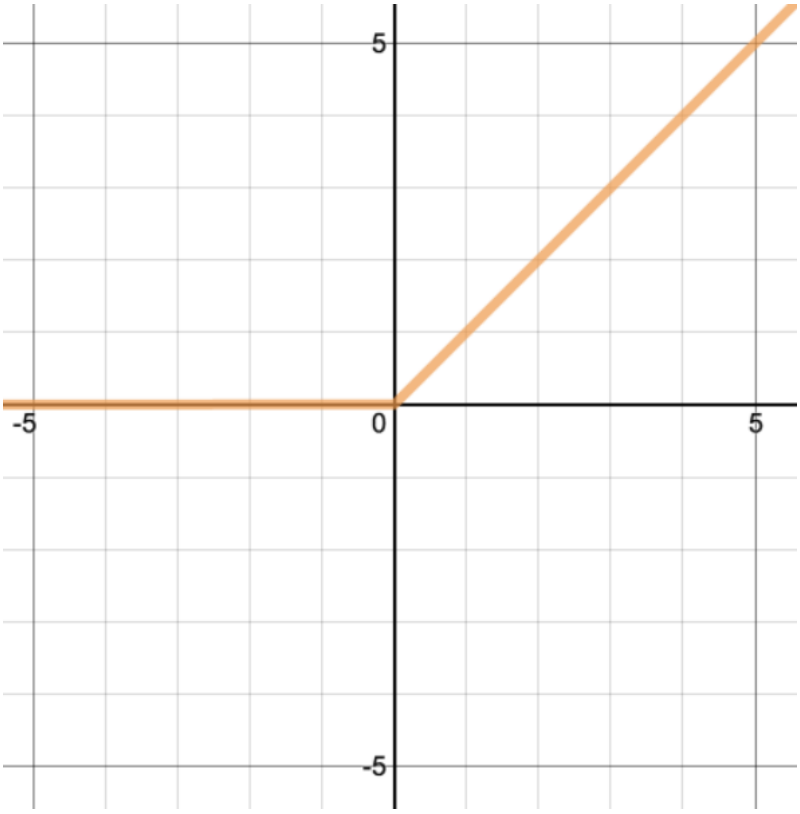


*Figura 2.2 Gráfica de función ReLU. Fuente: (Patterson & Gibson, 2017).*

La función de pérdida (del inglés “Loss Function”) es el método que evalúa el nivel de desempeño que el aprendizaje de la red está teniendo durante el proceso de entrenamiento, esto a través del contraste del valor predicho con el valor real.

La entropía cruzada es una función de pérdida ampliamente usada en Inteligencia Artificial; el término “cruzado” se utiliza para describir la comparación entre dos distribuciones, mientras que “entropía” proviene de la teoría de la información y hace referencia a la incertidumbre. Por ejemplo, una curva normal con una amplia dispersión, o varianza, implica también mayor incertidumbre acerca de la ubicación de los puntos de datos. Esta incertidumbre se denomina entropía (Patterson & Gibson, Deep Learning, 2017). La entropía cruzada realiza una comparación entre la separación de valores obtenidos por el modelo y la distribución real de las etiquetas de los datos. La entropía cruzada se utiliza como una función de pérdida que debe ser reducida para minimizar el error. (Shanmugamani, 2018).

Las Redes Neuronales Convolucionales (CNN por sus siglas en inglés) son una variante de las redes neuronales. La intención “es aprender características de orden superior en los datos mediante convoluciones. Son muy adecuadas para el reconocimiento de objetos en imágenes y consistentemente ocupan los primeros lugares en competiciones de clasificación de imágenes.” (Patterson & Gibson, Deep Learning, 2017).

### 2.1.2 Herramientas

*TensorFlow* es una biblioteca de código abierto diseñada para facilitar el desarrollo y entrenamiento de modelos de aprendizaje automático (*machine learning*) y redes neuronales. Desarrollado por el equipo de *Google Brain*, *TensorFlow* se ha convertido en uno de los *frameworks* más populares y ampliamente utilizados en el campo del aprendizaje automático y la inteligencia artificial. (Google Brain, 2024).

*Keras* es una interfaz de alto nivel por parte de la plataforma de *TensorFlow*, para construir y entrenar modelos de aprendizaje profundo. A partir de *TensorFlow 2.0*, *Keras* se ha integrado en *TensorFlow* como su interfaz predeterminada. (Google Brain, 2024).

En resumen, *TensorFlow* y *Keras* son un marco flexible que proporcionan herramientas esenciales para la construcción, entrenamiento y despliegue de modelos de aprendizaje automático en una variedad de aplicaciones. Su capacidad para trabajar con grandes conjuntos de datos y su flexibilidad en términos de arquitectura de modelos lo convierten en una elección popular entre los desarrolladores e investigadores en el campo del aprendizaje automático.

## 2.2 Visión computacional

Según la Automated Imaging Association (AIA), la visión artificial abarca todas las aplicaciones industriales y no industriales en las que una combinación de hardware y software brinda un guiado operativo a los dispositivos en la ejecución de sus funciones de acuerdo con la captación y procesamiento de imágenes. A continuación, se presentan diferentes técnicas empleadas dentro de este rubro.

### 2.2.1 Imagen

Se puede definir la imagen como la “representación digital de un objeto/escena o de un documento escaneado” (Ansari, 2020). Esta definición implica la conversión de dicha representación en una serie de números que se almacenan en un sistema informático.

Además de lo anterior, es necesario agregar que el proceso de representación digital de imágenes no se limita únicamente a la conversión de objetos, escenas o documentos escaneados en datos numéricos. Como (Gollapudi, 2019) señala, va más allá al objetivo de habilitar a las computadoras, dispositivos o máquinas en general para que sean capaces de ver, entender, interpretar y, en última instancia, 'manipular' la información visual que perciben. Esto implica que, mediante el análisis de imágenes y el procesamiento de datos visuales, estas entidades pueden realizar tareas tan diversas como la detección de objetos, el reconocimiento de patrones, la identificación de rostros y la navegación autónoma, entre muchas otras aplicaciones. En resumen, la representación digital de imágenes es un paso esencial para dotar a la tecnología de la capacidad de interactuar de manera más avanzada y eficiente con el entorno visual que la rodea.

La unidad mínima de una imagen es el pixel. Estos se encuentran acomodados de forma ordenada en filas y columnas llamado arreglos. Por ejemplo, si tenemos una imagen de 100 x 80 píxeles, esto significa que se compone de 100 filas y 80 columnas de píxeles.

Las imágenes se encuentran frecuentemente denotadas por el sistema RGB (por sus siglas en inglés Red, Green, Blue), el cual consiste en tres arreglos del mismo tamaño, cada uno correspondiente a un color primario. Cada uno de estos valores en el arreglo tiene un valor que varía en un rango de 0 a 255, lo que define la intensidad que tiene. Al combinar estos tres arreglos, se genera el espectro completo de colores, definiendo así el color de ese punto en la imagen. Otra manera de representar una imagen es a partir de la escala de grises, que consta de una única matriz de pixeles.

### 2.2.2 Eliminación de fondo

Comúnmente en el proceso del análisis de imágenes se tiene la necesidad de remover elementos que no se encuentran representando al objeto en estudio. Principalmente se utilizan dos algoritmos: por umbralización y segmentación de color.

La umbralización es el proceso por el cual se busca un punto de inflexión para hacer una distinción entre los elementos de una imagen y el fondo, con el fin de la creación de una imagen binaria. Existen umbrales estáticos y dinámicos. El primero, como el nombre lo indica, se fija un valor, y dependiendo de la lógica que se emplee (inverso o normal) los pixeles por debajo de este número serán tomados como cero, por el contrario, uno. Para los umbrales dinámicos, se emplean distintas técnicas para obtener el punto de inflexión. Por ejemplo, el caso de un umbral de OTSU, es una técnica que se basa en el uso de Histogramas. El umbral es determinado por la minimización de la varianza interclase, que se define como la sumatoria de varianzas dentro de dos clases ponderadas. (Dey, 2020).

El YCbCr es un espacio de color utilizado para la segmentación del color de una imagen. Está técnica al convertir una imagen en color RGB a un formato en color YCbCr, se obtiene una imagen resultante que consta de componentes de luminancia (Y) y componentes de crominancia (Cb y Cr).

El uso del espacio de color RGB no es la opción preferida para la detección y análisis basados en el color, debido a la combinación de información de color (crominancia) e intensidad (luminancia) y a sus características no uniformes.

En del artículo “Comparative Study of Skin Color Detection and Segmentation in HSV and YCbCr Color Space” los autores utilizaron los siguientes rangos de valores para los distintos componentes de la técnica YCbCr, para la detección de piel: Cr > 150 && Cr < 200 && Cb > 100 && Cb < 150. (Shaik, P, Kalist, Sathish, & Jenitha, 2015).

### 2.2.3 Detección de bordes

La detección de bordes en visión computacional se refiere al proceso de identificar y resaltar las transiciones abruptas en la intensidad de los píxeles en una imagen. Los bordes representan los límites entre diferentes regiones de una imagen y contienen información importante sobre los cambios en la estructura o la textura de los objetos presentes. Se utiliza

para extraer características relevantes de la imagen que pueden ser utilizadas posteriormente en tareas como segmentación, reconocimiento de objetos, seguimiento de objetos en movimiento, entre otras. El resultado de estas técnicas comúnmente es de naturaleza binaria, en donde los bordes se denotan de color blanco (pixeles con valor 255) y el resto de la imagen de un color negro (pixeles con valor 0). A continuación, se definen algunos de los algoritmos de detección bordes.

El algoritmo de Canny es un método ampliamente utilizado para la detección de bordes. Fue desarrollado por John F. Canny en 1986 (OpenCV, 2023). Realiza un suavizado de la imagen con un filtro Gaussiano de 5 x 5. Maneja un gradiente, en donde se tiene un valor central, que denota el borde, y los valores que se alejen de dicho valor no serán considerados bordes.

El algoritmo de Prewitt es otro método comúnmente utilizado para la detección de bordes en imágenes. Fue propuesto por Judith Prewitt en 1970 y se basa en el cálculo de gradientes en las direcciones horizontal y vertical para identificar cambios abruptos en la intensidad de los píxeles.

Por otra parte, el algoritmo de Sobel fue propuesto por Irwin Sobel y Gary Feldman en 1968 y se basa en el cálculo de gradientes utilizando máscaras de convolución y de igual manera que se hace en el método de Prewitt, se identifica cambios abruptos en la intensidad de los píxeles en las direcciones horizontal y vertical.

Los algoritmos de Prewitt y Sobel tienden a ser simples y computacionalmente eficientes, aunque pueden producir bordes más gruesos o bordes falsos, a comparación del algoritmo Canny. (Raviteja, Gayathri, & Thiyaneswaran, 2022).

### 2.2.4 Descriptor de características

Un descriptor de características es una manera de representar una imagen de manera más sencilla al capturar información relevante sobre sus rasgos fundamentales, como su forma, color, textura o movimiento. Por lo general, un descriptor de características transforma una imagen en un conjunto de características en forma de vector o matriz de longitud n. En el contexto de la creación de modelos, comúnmente se usa el descriptor de características HOG en vez de utilizar los pixeles sin procesar (Fernandez Villan, 2019).

Los descriptores HOG se generan a partir de una imagen mediante el cálculo de las imágenes de gradiente horizontal y vertical, seguido de la creación de histogramas de gradientes y la normalización en bloques. Posteriormente, estos descriptores normalizados se combinan en un vector de características. Estos descriptores normalizados por bloques son conocidos como descriptores HOG. (Dey, 2020).

# 3 ESTADO DEL ARTE

En el año 2019 se desarrolló un software donde contrastaron usando Redes Neuronales Multicapa y Support Vector Machine (SVM), para clasificar las imágenes. Y, para la extracción de características, utilizaron Momentos Hu, Momentos de Zernike y el descriptor de características HOG. Demostrando mejores resultados los momentos Zernike en combinación con SVM, logrando una tasa de reconocimiento del 98.7% para las imágenes estáticas. (Mancilla Morales, Vázquez Aparicio, Arguijo, Meléndez Armenta, & Vázquez López, 2019).

Por otro lado, en otro desarrollo (Pichucho, Constante, Gordón, & Mendoza, 2019), se programó un software que pudiera identificar el alfabeto y números en Lenguaje de Señas Ecuatorianas (LSEC). Se creó un software con una interfaz gráfica intuitiva que permite la interacción a través de una cámara de vídeo. Esto permite al algoritmo identificar tanto las letras y los números del LSEC de manera efectiva. El sistema fue desarrollado en el lenguaje de programación C++ y con el uso de la librería OpenCV. Los clasificadores creados a partir del entrenamiento de las redes neuronales artificiales (RNA) han demostrado una confiabilidad de 92.22% para los números y de 94.93% de precisión para las letras, esto para imágenes estáticas. Como se ve en la Figura 2.1, lo que realiza el algoritmo es por medio de algoritmos de visión computacional. Se puede observar cómo el software hace la detección de los bordes de las manos, para así poder clasificar y poder definir qué seña se está utilizando.

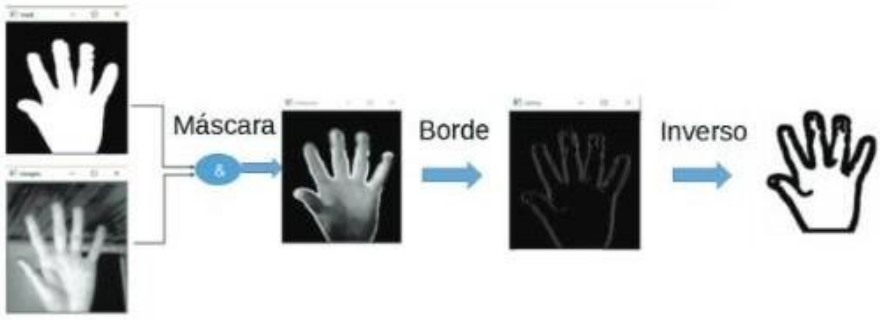


*Figura 3.1 Detección de bordes de una mano.*

Para realizar la clasificación se utilizaron dos redes neuronales. La primera red neuronal se utilizó para clasificar los números, la cual utilizó 256 neuronas en la capa de

entrada, 16 neuronas para la capa oculta y 9 neuronas para la capa de salida. La segunda red neuronal tiene 256 neuronas para la capa de entrada, 34 neuronas para la capa oculta y 25 neuronas para la capa de salida. Para comprobar que el entrenamiento de la máquina fue el adecuado, los autores definieron que las redes neuronales debían de tener un porcentaje de reconocimiento satisfactorio mayor al 86%.

Se ha propuesto una aplicación para reconocer el lenguaje de señas de la India, utilizando Redes Neuronales Convolucional (Kishore, Rao, Kumar, Kumar, & Kumar, 2018), El conjunto de datos que se utilizó es de 30,000 fotogramas de videos, conformado por 200 señas en 5 diferentes ángulos; cada signo era representado por 60 fotogramas. En el artículo se explica que las redes neuronales convolucionales tradicionales requieren una gran cantidad de datos para el entrenamiento; así que, desarrollaron su propia red neuronal para disminuir la cantidad de imágenes para el entrenamiento. Se obtuvo una máxima de exactitud del 95.54%, con un tiempo de entrenamiento de 296 horas. En el artículo se explica que las redes neuronales convolucionales convencionales requieren una gran cantidad de datos para el entrenamiento de ésta, por lo tanto, desarrollaron su propia red neuronal para disminuir la cantidad de imágenes para el entrenamiento. Para la capa de ingreso de datos de la red, utiliza imágenes de un vídeo de tamaño 640x480, de ahí se procede cambiar el tamaño de las imágenes a 128x128x3. Por último, se utilizó la función Softmax para realizar la clasificación de los vídeos.

En el artículo propuesto por Kenneth y Córdova para el reconocimiento de la LSM (Kenneth & Córdova, 2023), aplicaron las redes neuronales recurrentes (RNN por sus siglas en inglés), a imágenes obtenidas por cámaras de profundidad OAK-D, obteniendo los puntos claves usando la librería MediaPipe. Durante la experimentación, se emplearon 27 categorías que representaban el abecedario dactilológico de la LSM, a partir de vídeos realizando la seña. Concluyeron con una exactitud del 80.74%.

En el artículo titulado “Sign Language Recognition and Translation with Kinect” (Xiujuan Chai), los autores crearon un prototipo de traductor de lenguajes de señas del chino utilizando la tecnología de Microsoft Kinect para procesar imágenes. Los autores comentan que con dicha herramienta pueden obtener imágenes con mayor calidad de información, evitando así problemas referentes a la iluminación o de ruido en general en la imagen. El

tamaño del dataset que se utilizó fueron 239 palabras en el lenguaje de señas chino grabado cada uno cinco veces. Para lograr la verificación de las palabras requirieron de trayectorias de la sonda y de vectores y a través de distancias eucladianas generar un resultado de reconocimiento.

Los autores Islam y Raj (Islam & Raj, 2017) realizaron un sistema computacional para el reconocimiento de señalamientos de tráfico en Malasya. Lo que realizaba el software era sacar de una fuente de vídeo los cuadros por segundos en un formato RGB, para así después convertirlo a una escala de grises. Después se aplicó un filtro para así eliminar cualquier ruido que tuviera las imágenes en escala de grises. Para el caso particular se utilizó el algoritmo de thershold a un nivel de 0.18, esto significa que el píxel con un valor de ilumicencia igual o mayor a 0.18 se tomaba en cuenta como un píxel blanco, de otra manera sería negro. Esto reflejó un mayor rendimiento en el sistema. También eliminaron cualquier elemento menor a 300 pixeles, y después etiquetaron los componentes usando objetos conectividad 8. Para la clasificación de las imágenes se utilizó la librería Neural Network Pattern Recognition Tool desarrollado para MATLAB. Para la red neuronal se utilizaron dos capas de ingreso de datos, con la función Sigmoide en la capa escondida, y con la función Softmax en la capa de salida.

Por parte de la Universidad Nacional de Colombia, los autores Nope, Loaiza y Caicedo, se enfocaron en evaluar distintas técnicas de reconocimiento estándar en cuatro gestos diferentes, teniendo como el porcentaje de acertamiento entre el 87.88% y 97.14 %. Se propuso el uso de Máquinas de Estado Finito (FSM), teniendo entre sus ventajas su gran eficiencia, pudiendo ser utilizadas en tiempo real, además de que no requiere grandes cantidades de entrenamiento. Sin embargo, los autores comentan que “los gestos son altamente variables de una persona a otra, e incluso de un ejemplo a otro durante la ejecución por una misma persona” (Nope R., Loaiza C., & Caicedo B., 2008), haciendo hincapié en definir propiedades invariantes de la imagen a clasificar.

Los autores definieron 4 fases para la representación de los gestos las cuales son:

1. Segmentación de la mano. Dada a las problemáticas planteadas, los autores definen el uso de un algoritmo para la detención del color de la piel píxel a

píxel, y para eliminar el ruido de la imagen se centran en el área conexa del color de interés previamente mencionado.

2. Representación del movimiento. Se usaron vectores de flujo para la representación de movimiento, primeramente, reduciendo la resolución de la información sin pérdida significativa de información, después procesando los datos con dos filtros gaussianos, uno para el movimiento y el otro para la dirección.
3. Integración temporal. Esta fase es importante para la clasificación de gestos dinámicos, teniendo como principal función reunir la información del movimiento instantáneo generado en el momento anterior y procesarlas a través de imágenes de energía del movimiento e imágenes de la historia del movimiento.
4. Disminución de la dimensionalidad. Se procesó las imágenes para usar 36 vectores de entrenamiento, con lo cual se creó una matriz que almacena los histogramas normalizados de los vídeos de entrenamiento

Probaron la clasificación con los siguientes métodos: Clasificadores paramétricos, (clasificador bayesiano), clasificadores no paramétricos (el vecino más próximo, los k•vecinos y distancia mínima al centroide) y redes neuronales artificiales (redes neuronales probabilísticas y redes neuronales perceptron multicapa). Teniendo como menor porcentaje de clasificación, con un 87.88%, el método de distancia mínima al centroide, y como mayor porcentaje de clasificación, con 97.14%, con el método de redes neuronales perceptrón multicapa (MLP).

Por parte de la Universidad ESAN, los autores Ale y Fabián, crearon un sistema de detención de fatiga, enfocándolo para el reconocimiento de ojos cansados en conductores, utilizando algoritmos de visión computacional para la extracción de datos, y algoritmos de inteligencia artificial para realizar la clasificación. De los algoritmos destacados que utilizaron para la clasificación fueron Support Vector Machine (SVM), Random Forests y redes neuronales. Utilizaron el lenguaje de programación de Python, con el apoyo de las librerías Numpy, Pandas, Scikit-learn y OpenCV. Su muestra para el entrenamiento se basó en la investigación denominada Closed Eyes In the Wild (CEW), la cual contenía 1192

imágenes de personas con los ojos cerrados y 1231 con ojos abiertos, teniendo como un total de 2423 imágenes.

Un concepto que se aplicó para el desarrollo del proyecto fue Histogram Oriented Gradients (HOG), que se define como “descriptor de características ampliamente usado para la detección de objetos a través de sus formas y texturas, como el reconocimiento de logos en vehículos, así como para la detección del rostro o cuerpo humano.” (Ale & Fabián, 2019), el cual explica que el proceso para obtener HOG, consiste en convertir la imagen introducida en un conjunto de celdas determinada por píxeles, aplicando operaciones matemáticas para obtener el ángulo, considerar cambios de iluminación y de contraste, para tener como resultado final un vector. Dada a la combinación de varios factores, como el movimiento del objeto deseado, la calidad del vídeo, inclinación del usuario, falta de iluminación, entre otras cosas, es muy recurrente que el programa genere tanto Falsos positivos como Falsos negativos.

## 3.1 Inteligencia artificial en otros ámbitos

El Dr. Juan Lorenzo y la Dra. Irene Rodríguez publicaron para la revista científica llamada Medicentro, un artículo titulado “Aplicación de técnicas de visión computacional en la prueba de papanicolaou” (APLICACIÓN DE TÉCNICAS DE VISIÓN COMPUTACIONAL EN LA PRUEBA DE PAPANICOLAOU , 2012). Como el nombre indica, utilizan técnicas de procesamiento de imágenes para las muestras tomadas para la prueba del papanicolaou.

Dicha prueba tiene un impacto muy fuerte en la comunidad, puesto que, como dice el artículo, el cáncer cérvico-uterino es el segunda enfermedad que más afecta a la población femenina. Con esta prueba se puede detectar a tiempo ciertas células anómalas, y con ello, prevenir la enfermedad.

Según los autores, existe una gran carga de trabajo en los centros de salud debido a que la aplicación de las pruebas genera una carga de trabajo considerable. Las pruebas son realizadas principalmente por capital humano, y ésto se le pueden sumar ciertas circunstancias que podrían afectar un resultado óptimo de la prueba. Estos errores, como lo plantean los autores, podrían ser principalmente “en su procesamiento y en su lectura e

interpretación". Se puede añadir la necesidad de realizar pruebas de manera masiva, que representa un reto que los autores plantean resolver con el uso de la visión computacional.

En el artículo los autores explican que, para automatizar el proceso de lectura e interpretación, lo primero que se debe de realizar es definir de manera numérica ciertos parámetros a partir de rasgos característicos (en el caso particular, definir diámetro de núcleos, citoplasma, etc.).

Una vez definidas las variables en términos numéricos, el proceso de análisis automatizado de la imagen consta de tres principales etapas. La primera etapa es la obtención de la imagen digital, encontrados sistemas más modernos, son realizados mecanismos robotizados, y en sistemas más rudimentarios, son directamente tomadas del microscopio por medio de un operador debidamente capacitado. La siguiente etapa consistiría en la segmentación de la imagen. Esto se realiza con el fin de procesar la imagen, y por medio de algoritmos, quitar el ruido en la imagen, arreglar la iluminación y el contraste, entre otras cosas. Los principales algoritmos que emplearon los autores fueron las de basadas en color y la extracción jerárquica de regiones. Y, por último, consiste en la clasificación de las células, en el caso particular, en dos principales categorías: normales y sospechosas. Para este proceso se apoyan en otras técnicas de la inteligencia artificial, como lo es las redes neuronales.

En el artículo "Inteligencia artificial para asistir diagnóstico clínico en medicina" (Lugo-Reyes, Maldonado-Colín, & Murata, 2014), realizan una breve descripción de distintas herramientas que ofrece la inteligencia artificial. Entro ellos, definen al aprendizaje automático como "conocimiento obtenido al procesar computacionalmente datos de adiestramiento contenidos en esas bases de datos".

Se describe también que, por medio de la regresión logística multivariante, en una investigación que se llevó a cabo en el 2012 con los registros médicos de 9023 pacientes a quienes se efectuó amigdalectomía en los siete años previos, con la intención de buscar predecir ciertos factores de riesgo como podría ser una hemorragia. Los investigadores pudieron llegaron a la conclusión de definir los factores de riesgos dependiendo de las condiciones del paciente.

Otro tema que aborda el artículo es acerca de las investigaciones de la Universidad de Tampere, en Finlandia. Dichas investigaciones son basadas bajo el paradigma de "Razonamiento basado en casos", como lo define el autor "resolución de problemas en el que un problema nuevo se soluciona atendiendo casos similares del pasado" (Lugo-Reyes, Maldonado-Colín, & Murata, 2014). La intención de los investigadores de la mencionada universidad es la de crear un sistema experto, el cual sirva para reclasificar las inmunodeficiencias en 11 grupos de defectos apoyándose en cinco médicos. Los resultados preliminares de la investigación se han definido por una correcta definición de 469 casos, con un porcentaje de exactitud del 66%.

Una aplicación interesante de las redes neuronales del que se habla en el artículo es, el de diagnosticar tuberculosis pleural, únicamente con datos clínicos y el estado de infección por VIH. Se obtuvieron resultados favorables, alcanzando un 90% de exactitud. La importancia de este método es la nula intervención de agentes invasores, pudiendo ser una posible alternativa para el diagnóstico de dicha enfermedad.

El Instituto Nacional de Astrofísica Óptica y Electrónica, escribió un artículo, para la revista mexicana de física, titulado "Obtención de los parámetros ópticos de la piel usando algoritmos genéticos y MCML" (Cruzado & Montiel, 2011). El escrito muestra los resultados que obtuvieron para calcular distintos parámetros de la piel (coeficiente de absorción, coeficiente de esparcimiento y factor de anisotropía) por medio de valores experimentales (la transmitancia total, transmitancia colimada y reflectancia difusa) de un tejido biológico.

Para ello utilizaron el Método Monte Carlo, que lo definen como "colección de herramientas para estimar valores a través de muestreo y simulación. El método proporciona una solución aproximada a una variedad de problemas matemáticos realizando pruebas estadísticas en una computadora" (Cruzado & Montiel, 2011). Para el caso particular de la investigación, se enfocaron en usar el método Monte Carlo Multi-Layered (MCML) que fue definido por Lihong Wang y Steven Jacques, el cual vendrá a definir poblaciones de datos de los fotones de los datos tomados, y por medio de un algoritmo genético, realizan las clasificaciones de los parámetros ópticos buscados. El fin de la investigación es tan solo mostrar los datos obtenidos por medio de dichos algoritmos

La visión computacional es un conjunto de algoritmos, que el Instituto Nacional de Astrofísica, Óptica y Electrónica define como “reconocer y localizar objetos en el ambiente mediante el procesamiento de las imágenes” (Sucar & Gómez).

Por parte de la Universidad Autónoma de Yucatán, se escribió un artículo con el título de “Estimación de la Distancia a un Objeto con Visión Computacional” (Magaña Z., 2017). Utilizaron la librería OpenCV para la realización de su proyecto, tomando como base el algoritmo SURF que presenta la librería en cuestión.

Por medio de una plataforma que se movía verticalmente y con un objeto situado en un punto especifico del espacio en un entorno controlado, como se ve en la Figura 3.2, pudieron calcular la distancia que la cámara se encuentra del objeto de observación

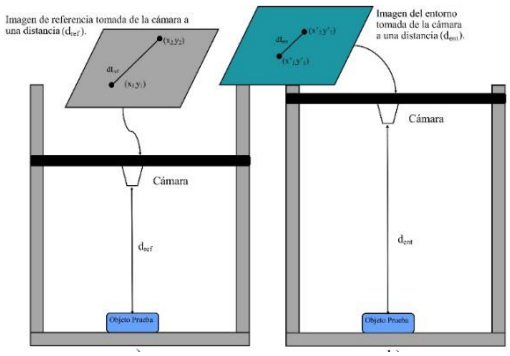


*Figura 3.2 Calcular la distancia de un objeto con visión computacional.*

Esto lo hicieron en base a la fórmula para calcular la distancia calculada entre dos puntos (en este caso serían pixeles):

$$D = ((x_1 - x_2)^2+(y_1 - y_2)^2)^{1/2}$$

Donde D es la distancia, $P(x_1, y_1)$ es el punto uno y $Q(x_2, y_2)$ son las coordenadas del punto dos.

Con base en ello lograron calcular la distancia entre la cámara y el objeto, en un entorno controlado, logrando una estimación del 96.1%, con un promedio de error del 0.99% y una desviación estándar del 0.1% de la distancia.

# 4 DESARROLLO

## 4.1 Creación del conjunto de datos

El conjunto de datos empleado está conformado por dos partes. La primera es un conjunto de datos público por parte del Instituto Tecnológico Superior de Misantla, en su artículo titulado “Traducción del lenguaje de señas usando visión por computadora”, el cual contaba con 21 señas del Lenguaje de Señas Mexicano. (Mancilla Morales, Vázquez Aparicio, Arguijo, Meléndez Armenta, & Vázquez López, 2019).

La estructura de los directorios está pensada en una sola carpeta principal, la que contiene 26 carpetas, correspondientes a cada una de las señas. Así se tendrán divididas y clasificadas cada una de las imágenes. Se utilizaron un total de 18,098 imágenes para el entrenamiento, de las cuales, 6,300 para 21 clases, las cuales fueron tomadas del artículo previamente mencionado.

Para la otra parte del conjunto de imágenes, se creó un script en Python llamado “*create_gesture_data.py*”, como ventana principal. Contaba con un recuadro azul (conocido como ROI), el cual delimita el área donde se debe de colocar la mano. A su vez, en la parte inferior del ROI, hay una leyenda que le debe indicar al usuario a cuál seña se va a etiquetar la imagen. Esta etiqueta puede ser cambiada a través de las teclas A y D. En la Figura 4.1 se visualiza la interfaz gráfica descrita con anterioridad. Para tomar una imagen es necesario presionar la tecla S.

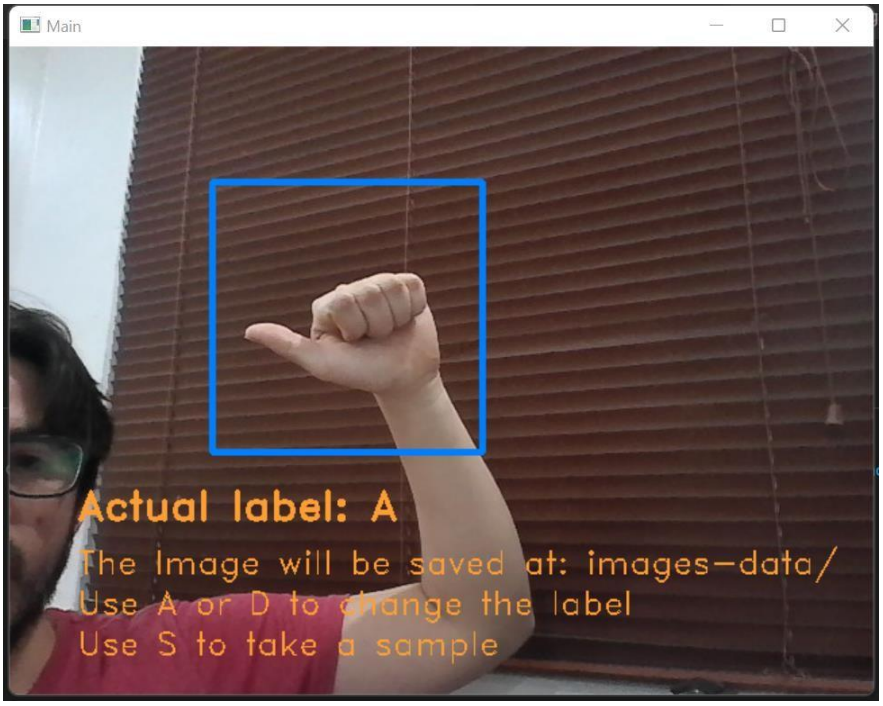


*Figura 4.1 Interfaz gráfica para agregar imágenes al dataset.*

A sí mismo, en el mismo programa, se muestra una ventana adicional, que se observa en la Figura 4.2, en donde la imagen tomada del ROI es procesada, y con base en la salida tener una idea del ruido que pudiera tener la imagen.


*Figura 4.2 ROI procesado para la generación de dataset.*

Para guardar la imagen en el directorio se creó el *Fragmento de código 4.1*, en donde como parámetros recibe la etiqueta, el directorio relativo, y la imagen tomada del programa. Al final de la ejecución se generan tres imágenes dentro de la ruta especificada, en donde a la misma imagen se le aplica un ajuste en la iluminación (refiriéndose a la función "*adjust_gamma*", que será explicada más adelante), para con ello tener más muestras de una misma imagen.

```
def create_img(label, path, my_img):
  darker = adjust_gamma(my_img, gamma=0.5)
  lighter = adjust_gamma(my_img, gamma=1.5)
  class_path = os.path.join(absolute_path, path + label)

  if not os.path.exists(class_path):
    os.mkdir(class_path)

  new_path = class_path + label + str(uuid.uuid1()) + '.jpg'
  new_path_darker = class_path + label + str(uuid.uuid1()) +'-darker.jpg'
  new_path_lighter = class_path + label + str(uuid.uuid1()) +'-
lighter.jpg'

  cv2.imwrite(new_path, my_img)
  cv2.imwrite(new_path_lighter, lighter)
  cv2.imwrite(new_path_darker, darker)
  print('Image added !')
```

*Fragmento de código 4.1 Función para crear imagen.*

Para cambiar la iluminación, se tomó a consideración el *Fragmento de código 4.2*, propuesto por Adrian Rosebrock, en su artículo titulado "*OpenCV Gamma Correction*" (Rosebrock, 2015). Dicho artículo hace una descripción entre las diferencias de cómo se percibe una imagen por parte del ojo humano, en contraste del sensor de una cámara.

```
def adjust_gamma(image, gamma=1.0):
   inv_gamma = 1.0 / gamma
   table = np.array([
      ((i / 255.0) ** inv_gamma) * 255
      for i in np.arange(0, 256)])
   return cv2.LUT(image.astype(np.uint8), table.astype(np.uint8))
```

*Fragmento de código 4.2 Función para cambiar la iluminación de una imagen.*

Cambiando los valores de gamma, se obtienen distintas iluminaciones de la misma imagen. Cuando el valor de gamma es igual a 1, no hay cambio en la imagen original, en cambio cuando el valor de gamma está por debajo de 1, la imagen se vuelve más oscura, y en el caso que sea por encima de 1, la imagen se vuelve más clara. En la Figura 4.3 se muestra los efectos de la función con distintos valores gamma.

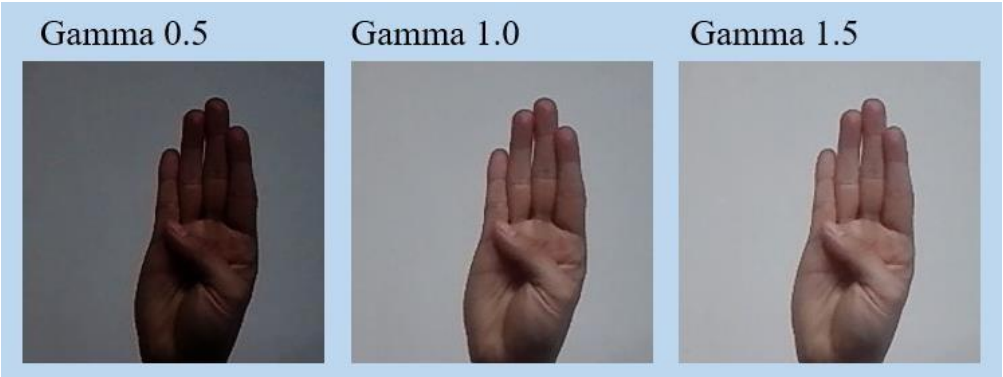


*Figura 4.3 Contraste entre diferentes valores gamma.*

El *Fragmento de código 4.1* es llamada en dos ocasiones, una con la imagen original, y otra dando un giro en el eje vertical de la imagen, intentando emular un espejo. Dado que la función para guardar la imagen crea tres imágenes, al final de la operación se tiene seis imágenes de la misma toma, tres en la posición original y tres giradas al efecto espejo, ambas con tres distintas iluminaciones.

## 4.2 Procesamiento de la imagen

Para el procesamiento de la imagen se creó el script "*process_image.py*". Dicho script tiene las funciones para procesar la imagen. A su vez, se programó para el dado caso de que se quiera aplicar una técnica especifica de visión computacional a una imagen, y visualizarla de manera aislada. También cuenta con la posibilidad de pasarle un directorio (como el generado en la sección de la creación del dataset), para así procesar todas las imágenes de este, creando un nuevo directorio ya con los archivos procesados.

El procesamiento de la imagen se dividió principalmente en tres técnicas de visión computacional: la eliminación de fondo, detección de bordes y la extracción de características. A continuación, se detallará cada una de las técnicas empleadas para cada caso concreto.

### 4.2.1 Eliminación del fondo

Para la eliminación del fondo se utilizaron dos técnicas: por umbralización y por segmentación por color YCbCr. A continuación, se presentarán los resultados obtenidos, aplicando cada método en la Figura 4.4, que presenta el ejemplo de la muestra tomada sin procesamiento.

*Figura 4.4 Imagen sin procesar.*

En un primer aproximamiento, por umbralización, se creó la función "*process_image*" en el archivo "*process_image.py*", que tiene como parámetro el arreglo de imágenes en formato RGB. Como se muestra en el *Fragmento de código 4.3*, se procede a convertir a escala de grises. Luego, se aplica la función threshold (incluída en la librería de OpenCV), haciendo una combinación de la umbralización Inversa y la umbralización de OTSU. Por último, se aplica la máscara a la imagen original en escala de grises, y ya la imagen está lista para aplicarse algún algoritmo de detección de bordes.

```
def process_image(img):
  # Convert the hand region to grayscale
  hand_gray = cv2.cvtColor(hand_region, cv2.COLOR_BGR2GRAY)

  # Apply thresholding to the hand region
  hand_thresh = cv2.threshold(hand_gray, 0, 255, cv2.THRESH_BINARY_INV +
cv2.THRESH_OTSU)[1]

  masked_image = hand_gray & ~hand_thresh

  # Rest of the code for border detection

  return border_image
```

*Fragmento de código 4.3 Obtener máscara con método de umbralización.*

Finalmente, en la Figura 4.5, se puede observar cómo luce el valor obtenido para la variable "*hand_thresh*". Como se aprecia, hay pérdida de la información, en especial cerca de las zonas de los dedos, en donde hay más sombra.

*Figura 4.5 Máscara obtenida con método de umbralización.*

En el archivo "*process_imag.py*", se encuentra la función llamada "*process_image_ybcr*", que detecta bordes a partir de la segmentación por el color YCbCr. Tiene como parámetros los arreglos de bits para la imagen en formato RGB, y el método con el cual se va a detectar los bordes (este último será explicado más adelante en el presente documento).

En primera instancia, se convierte la imagen de RGB a YCbCr, como se ilustra en el *Fragmento de código 4.4*.

```
def process_image_ycbcr(img, border_method):
  # Convert the image from RGB to YCbCr color space
  img_ycrcb = cv2.cvtColor(img, cv2.COLOR_BGR2YCR_CB)

  # Rest of the code for border detection

  return borded_img
```

*Fragmento de código 4.4 Convertir imagen en formato YCbCr.*

La variable llamada "*img_ycrcb*", del *Fragmento de código 4.4*, da como resultado la Figura 4.6.

*Figura 4.6 Imagen en formato YCbCr.*

Una vez con la imagen en formato YCbCr, se define dos arreglos de NumPy ("*lower_skin*" y "*upper_skin*") que representan los valores mínimos y máximos en el espacio de color YCbCr para el tono de piel, mostrado en el *Fragmento de código 4.5*. Adicional, se aplica la función "*cv2.inRange*" para crear una máscara que identifique las regiones de la imagen que están dentro del rango de colores de piel definido anteriormente definidos en las variables "*lower_skin*" y "*upper_skin*".

```
def process_image_ycbcr(img, border_method):
  # Convert the image from RGB to YCbCr color space

  # Define the YCbCr color ranges for skin color
  lower_skin = np.array([0, 135, 85], dtype=np.uint8)
  upper_skin = np.array([255, 180, 135], dtype=np.uint8)

  # Threshold the YCbCr image to obtain the skin color mask
  mask_skin = cv2.inRange(img_ycrcb, lower_skin, upper_skin)

  # Remaining code

  return border_img
```

*Fragmento de código 4.5 Obtener máscara por método YCbCr.*

En la Figura 4.7 se observa el kernel utilizado para las operaciones morfológicas, aplicadas con el fin de reducir posible ruido.

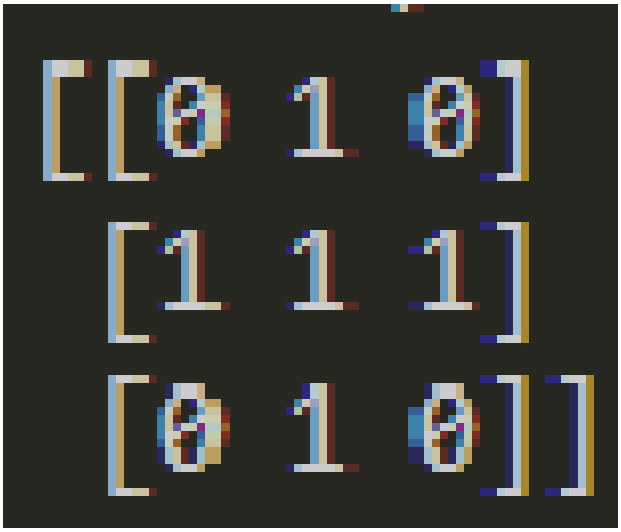


*Figura 4.7 Kernel elíptico de 3 x 3.*

Concretamente, se usaron las operaciones morfológicas de erosión y dilatación. En una primera instancia se elimina el ruido que se podría encontrar en la imagen, y con la dilatación se hace una expansión de los posibles pixeles que pudieran haber visto afectados por la erosión. En el *Fragmento de código 4.6*, con el uso de la librería de OpenCV, se aplican dichas operaciones morfológicas, utilizando el kernel elíptico previamente mencionado.

```
def process_image_ycbcr(img, border_method):
  # Convert the image from RGB to YCbCr color space

  # Apply morphological operations to remove noise and fill gaps in the
mask
  kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
  mask_skin = cv2.erode(mask_skin, kernel, iterations=1)
  mask_skin = cv2.dilate(mask_skin, kernel, iterations=1)

  # Rest of the code for border detection

  return border_img
```

*Fragmento de código 4.6 Operaciones morfologicas para la máscara YCbCr.*

Para este punto, el valor de la variable "*mask_skin*" del *Fragmento de código 4.6* da como resultado la Figura 4.8. Se ha de recalcar cómo es que la máscara detectó la totalidad de la mano.

*Figura 4.8 Máscara obtenida con método YCbCr.*

Llegados a este punto, el paso siguiente sería quitar el fondo de la imagen. En primera instancia, el color de la mano resulta irrelevante para detectar a la misma, entonces se optó por transformar la imagen en una escala de grises, para así contar con una sola matriz numérica, en vez de las tres que podría tener si siguiera manejando RGB. Se realiza una operación lógica "*AND*", entre la imagen en escala de grises y la máscara resultante (como la Figura 4.8), en donde se obtiene una imagen encontrando las similitudes de las dos matrices.

```
def process_image_ycbcr(img, border_method):
  # Convert the image from RGB to YCbCr color space

  # Masked image
  gray_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

  masked_img = gray_image & mask_skin

  # Rest of the code for border detection

  return border_img
```

*Fragmento de código 4.7 Operación para obtener máscara YCbCr.*

Se detectó el contorno resultante de la máscara, y se aplicó sobre la imagen original en un contorno verde, mostrado en la Figura 4.9.

*Figura 4.9 Contorno de mano.*

Ya con el fondo removido, se aplicó un suavizado Gaussiano con un kernel de 3 x 3, ilustrado en el *Fragmento de código 4.8 Aplicación de suavizado Gaussiano*.

```
def process_image_ycbcr(img, border_method):
  # Code to get masked_img

  blur = cv2.GaussianBlur(masked_img, (3, 3), 0)

  # Rest of the code for border detection

  return border_img
```

*Fragmento de código 4.8 Aplicación de suavizado Gaussiano.*

En la Figura 4.10 se observa el gesto de la mano en una escala de grises, ya sin fondo después de la aplicación de la máscara y el suavizado en la función "*process_image_ycbcr*".

*Figura 4.10 Imagen en escala de grises sin fondo.*

### 4.2.2 Detección de bordes

Se compararon el uso de tres algoritmos para la detección de bordes. Para este punto la imagen ya se ha eliminado el fondo con alguna de las técnicas explicadas con anterioridad.

En la función "*process_image_ycbcr*", se recibe el parámetro "*border_method*", que es una cadena de texto, donde el rango de valores es "*prewitt*", "*canny*" y "*sobel*". En el *Fragmento de código 4.9*, explaya la parte de la función en donde se aplica la comparativa para aplicar determinado método.

```
def process_image_ycbcr(img, border_method):
  # Here is the code to remove the background
  blur = …
  # Border detection
  if border_method == 'prewitt':
    kernelx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
    kernely = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
    imgPrewittX = cv2.filter2D(blur, -1, kernelx)
    imgPrewittY = cv2.filter2D(blur, -1, kernely)
    border_image = imgPrewittX + imgPrewittY
  elif border_method == 'canny':
    border_image = cv2.Canny(blur, 100, 200)
    # Expands lines!
    border_image = cv2.dilate(border_image, kernel, iterations = 1)
  else if border_method == 'sobel':
    img_sobel_x = cv2.Sobel(blur, cv2.CV_8U, 1, 0, ksize=5)
    img_sobel_y = cv2.Sobel(blur, cv2.CV_8U, 0, 1, ksize=5)
    border_image = img_sobel_x + img_sobel_y
  return border_image
```

*Fragmento de código 4.9 Condicional para aplicar detección de bordes.*

El primer método empleado es el Prewitt. Este no cuenta con una función directa en la librería de OpenCV, sin embargo, se puede conseguir aplicando un filtro personalizado acorde con el operador Prewitt. Se necesita un kernel X, y un kernel Y, se aplica dichos kerneles en la imagen con la ayuda de la función de OpenCV filter2D. Se obtendrán dos imágenes, que al realizar una sumatoria da como resultado la detección de bordes. La implementación es mostrada en el *Fragmento de código 4.10*.

```
def process_image_ycbcr(img, border_method):
  # Here is the code to remove the background

  blur = …

  # Border detection
  if border_method == 'prewitt':
    kernelx = np.array([[1,0,-1],[1,0,-1],[1,0,-1]])
    kernely = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
    img_prewitt_x = cv2.filter2D(blur, -1, kernelx)
    img_prewitt_y = cv2.filter2D(blur, -1, kernely)
    border_image = img_prewitt_x + img_prewitt_y

  return border_image
```

*Fragmento de código 4.10 Detección de bordes con método Prewitt.*

Para el algoritmo de Canny, se realizó en dos pasos, el primero aplicar el método, y el segundo haciendo una dilatación. Se utilizó la función que viene integrada en la librería de OpenCV llamada Canny (OpenCV, 2023), para hacer la detección de los bordes. Los valores de umbral de 100 a 200, que vienen como parámetros de la función, fueron escogidos empíricamente a través de varias iteraciones. Como paso extra se aplica una dilatación a las líneas, aplicando el mismo kernel que aparece en la Figura 4.7. Todo viene integrado como se ilustra en el *Fragmento de código 4.11*.

```
def process_image_ycbcr(img, border_method):
  # Here is the code to remove the background

  blur = …

  # Border detection
  elif border_method == 'canny':
    border_image = cv2.Canny(blur, 100, 200)
    # Expands lines!
    border_image = cv2.dilate(border_image, kernel, iterations = 1)

  return border_image
```

*Fragmento de código 4.11 Detección de bordes con método Canny.*

Para hacer la detección de bordes con el método de Sobel, se usó la función nativa de OpenCV, donde se detecta los bordes en el eje X y en el eje Y, obteniéndolo con una sumatoria de las imágenes resultantes de las funciones, observado en el *Fragmento de código 4.12*.

```
def process_image_ycbcr(img, border_method):
  # Here is the code to remove the background

  blur = …

  # Here the
  else if border_method == 'sobel':
    img_sobel_x = cv2.Sobel(blur, cv2.CV_8U, 1, 0, ksize=5)
    img_sobel_y = cv2.Sobel(blur, cv2.CV_8U, 0, 1, ksize=5)
    border_image = img_sobel_x + img_sobel_y

  return border_image
```

*Fragmento de código 4.12 Detección de bordes con método Sobel.*

En la Figura 4.11 se aprecia la detección de bordes con los tres algoritmos empleados.

En el caso del método de Canny, el contorno de la mano está bien definido, el contorno interno de los dedos está delimitado, pero en muchos casos no se visualiza los dedos por separado. Llega a ver casos donde detecta falsos bordes.

Por otra parte, el algoritmo de Prewitt no detecta de manera uniforme el contorno de la mano, pero a diferencia de los otros dos algoritmos, el contorno de los dedos se aprecia de una mejor manera.

Para Sobel, el borde de la mano es detectado como una línea gruesa, pero llegado a un punto los bordes se desvanece. Es el método que más falsos bordes presentó.

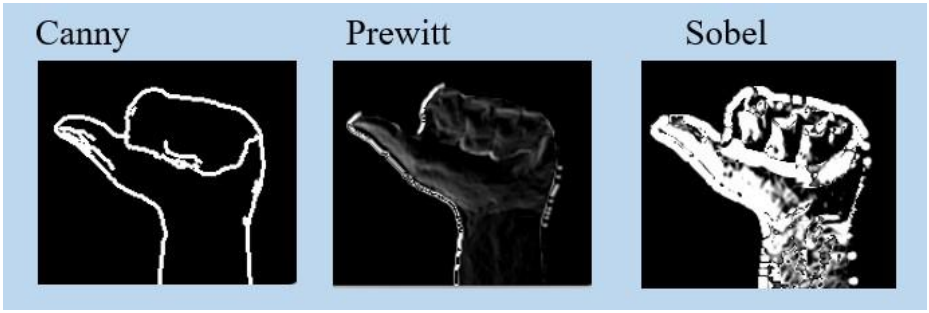


*Figura 4.11 Comparación de distintos algoritmos de detección de bordes.*

### 4.2.3 Extracción de características

Se decidió usar el extractor de características HOG (por sus siglas en inglés, Histogram of Oriented Gradients), para describir la imagen. Las siguientes funciones de Pyhton fueron tomadas del libro "*Mastering OpenCV 4 with Python*" (Fernandez Villan, 2019), fueron incorporadas al archivo "*process_image.py*" y adecuadas según las necesidades que surgieron.

En el *Fragmento de código 4.13*, se incorporó la función "*get_hog*", se apoya en la función HOGDescriptor de la librería OpenCV para la obtención de las características HOG, que retorna el objeto de OpenCV de HOG. Recibe como parámetros los siguientes:

- Tamaño de ventana (Window size): Se refiere al tamaño de la ventana que se desliza por la imagen para calcular los HOG.
- Tamaño de bloque (Block size): Es el tamaño de cada bloque dentro de la ventana.
- Desplazamiento de bloque (Block stride): Representa el desplazamiento o paso que se toma entre bloques adyacentes dentro de la ventana.
- Tamaño de celda (Cell size): Define el tamaño de cada celda dentro de un bloque.
- Número de contenedores (Number of Bins): Hace referencia a la cantidad de contenedores o bins, en los cuales se divide el rango de orientaciones de gradientes.

Se empleó el tamaño de la imagen de 200x200 pixeles como parámetro para el tamaño de ventana. Se utilizó un tamaño de bloque de 8x8, un desplazamiento de bloque de 4x4, y un tamaño de celda de 8x8. Además, se usó un número de bins de 9. Obteniendo un tamaño de arreglo 21,609 como el tamaño del descriptor HOG. Dado que los valores del tamaño de bloque y de la celda deben de estar alineados. El valor del desplazamiento de bloque debe de ser múltiplo del tamaño de la celda.

```
def get_hog(size_image):
  hog = cv2.HOGDescriptor((size_image, size_image), (8, 8), (4, 4), (8,
8), 9, 1, -1, 0, 0.2, 1, 64, True)

  print("hog descriptor size: '{}'".format(hog.getDescriptorSize()))
  return hog
```

*Fragmento de código 4.13 Obtener el tamaño del vector HOG.*

Se creó la función "*get_hog_descriptor*", donde toma dos argumentos: 1) "*hog*", que es la instancia de características HOG proporcionadas (obtenida de la función "*get_hog*"), e 2) "*im*", que es la imagen en cuestión. En el *Fragmento de código 4.14* se ilustra lo anterior descrito.

```
def get_hog_descriptor(hog, im):
  return hog.compute(deskew(im, len(im)))
```

*Fragmento de código 4.14 Obtener el descriptor HOG.*

En el *Fragmento de código 4.15*, se muestra la función "*get_hog_fetures"*, que obtiene el vector de características de una imagen dada. La imagen es proporcionada por medio de un parámetro llamado "*im*". Se le realiza un preprocesamiento, para la eliminación de ruido y convertirlo a escalas de grises. Se hace el uso de las funciones "*get_hog*" y "*get_hog_descriptor*" (*Fragmento de código 4.13* y *Fragmento de código 4.14*, respectivamente) para realizar la extracción de las características. Por último, se realiza operaciones de adecuación del arreglo de datos, para finalmente retornar el vector.

```
def get_hog_features(im):
  proccess_data = process_image_ycbcr(im, 'blur')
  size_image = len(im)
  hog = get_hog(size_image)

  hog_descriptor = get_hog_descriptor(hog, proccess_data)
  hog_descriptor = np.squeeze(hog_descriptor)
  hog_descriptor = hog_descriptor.astype(np.float32)
  hog_descriptor = hog_descriptor.reshape(1, -1)

  return hog_descriptor
```

*Fragmento de código 4.15 Obtener característica HOG.*

## 4.3 Entrenamiento de Red neuronal convolucional (CNN)

El primer modelo que se utilizó fue una red neuronal convolucional (CNN). Esta tenía la siguiente arquitectura denotada en la Figura 4.12:

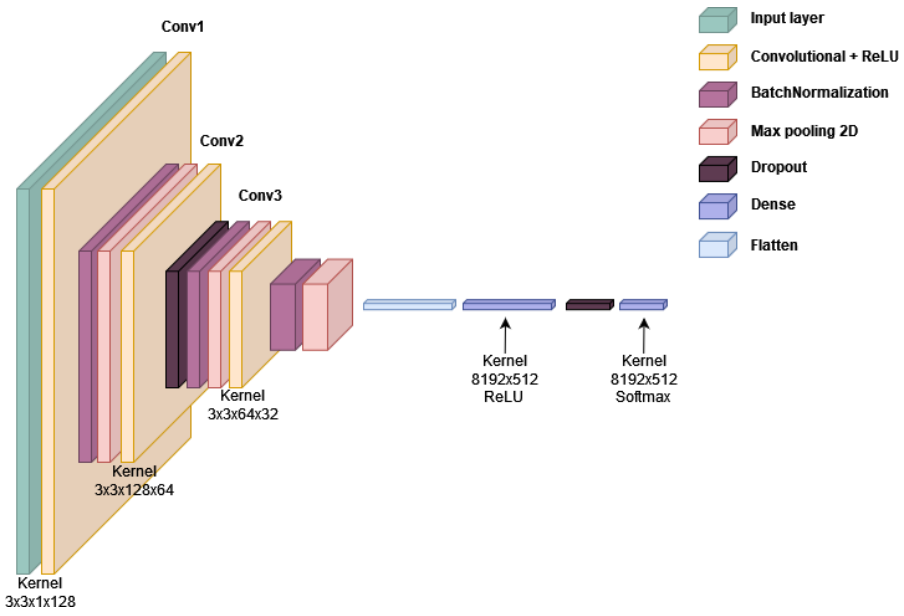


*Figura 4.12 Arquitectura de CNN.*

Se usó la librería de Tensorflow Keras. En el *Fragmento de código 4.16*, se denota el uso de la librería para la creación de las distintas capas de la red neuronal. Todas las capas usan la función de activación "*relu*", a excepción de la última capa, que usa la función "*softmax*". El optimizador que se usó fue el "*Adam*" a una razón de aprendizaje del 0.01.

```
from tensorflow.keras import layers, models

model = models.Sequential()
model.add(layers.Conv2D(75, (3,3), strides=1, padding='same',
activation='relu', input_shape=(img_height, img_width, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPool2D((2,2), strides=2, padding='same'))
model.add(layers.Conv2D(50, (3,3), strides=1, padding='same',
activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.BatchNormalization())
model.add(layers.MaxPool2D((2,2), strides=2, padding='same'))
model.add(layers.Conv2D(25, (3,3), strides=1, padding='same',
activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPool2D((2,2), strides=2, padding='same'))
model.add(layers.Flatten())
model.add(layers.Dense(units=512, activation='relu'))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(units=26, activation='softmax'))

model.compile(optimizer=optimizers.Adam(learning_rate=0.01),
              loss='categorical_crossentropy', metrics=['accuracy'])

model.summary()
```

*Fragmento de código 4.16 Arquitectura de red neuronal convolucional.*

En el archivo "*create_cnn_model.py*", se declararon unas variables globales para la CNN, esto denotado en el *Fragmento de código 4.17*. La primera variable es la de "*imgs_path*", que almacena las imágenes con la detección de bordes, empleando los algoritmos de computación expuestos en el presente documento. Dichas imágenes deben de estar guardada en la misma ruta donde se está ejecutando el archivo. Después, se tiene las variables "*img_height*" e "*img_width*", las cuales almacenan la altura y la anchura de la imagen, respectivamente. Y, por último, se encuentra la variable "*nr_epocs*", que indica el número de iteraciones (o "*epochs*") que realizará la red neuronal.

```
import os

imgs_path = 'images-canny/'
absolute_path = os.path.dirname(__file__)
full_path = os.path.join(absolute_path, imgs_path)

img_height = 200
img_width = 200
batch_size = 32
nr_epochs = 20
```

*Fragmento de código 4.17 Variables para el entrenamiento de la red neuronal.*

La librería de "*Keras*" cuenta con un módulo llamado "*ImageDataGenerator*", el cual ayuda a adecuar las imágenes para el uso dentro de la CNN. En el *Fragmento de código 4.18*, se aprecia la creación de las variables, donde se indican que son "*categorical*" para dividirlos en clases, y se indica que el formato de color "*grayscale*". La razón entre los datos de entrenamiento y de validación ha sido de un 80% y 20%, de las imágenes mencionadas en el apartado "*Creación del conjunto de datos*" del presente documento.

```
from keras.preprocessing.image import ImageDataGenerator
train_generator = train_datagen.flow_from_directory(
    full_path,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    color_mode='grayscale',
    subset='training') # set as training data

batch_size = train_generator.samples

validation_generator = train_datagen.flow_from_directory(
    full_path, # same directory as training data
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    color_mode='grayscale', subset='validation') # set as validation data
```

*Fragmento de código 4.18 Generación de conjunto.*

En la función "*fit*", empleada en la primera línea del *Fragmento de código 4.19*, se inicia con el entrenamiento del modelo. En la segunda sentencia, el modelo es guardado con el nombre "*tf-lsm-model*". Se crea una carpeta en la ruta raíz del archivo con el nombre del modelo.

```
history = model.fit(
    train_generator,
    steps_per_epoch = train_generator.samples // batch_size,
    validation_data = validation_generator,
    validation_steps = validation_generator.samples // batch_size,
    epochs = nr_epochs)

model.save('tf-lsm-model')
```

*Fragmento de código 4.19 Creación del modelo.*

Una primera aproximación que se realizó fue el de utilizar el *Fragmento de código 4.20*, como optimizador de la CNN. Con esto, se obtuvo una variación en el porcentaje de precisión como es mostrado en la Figura 4.13.

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

*Fragmento de código 4.20 Optimizador adam preliminar.*

Para solventar el inconveniente del párrafo anterior, se empleó el *Fragmento de código 4.21*, donde se le pasa como parámetro de razón de aprendizaje llamado "*learning_rate*" un valor equivalente al 0.01. Esto hizo que el aprendizaje tomara más tiempo y recursos computacionales, sin embargo, aumentó el porcentaje de precisión y se eliminó las variaciones de esta.

```
model.compile(optimizer=optimizers.Adam(learning_rate=0.01),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

*Fragmento de código 4.21 Optimizador adam adecuado.*

En los primeros aproximamientos, el aumento entre epochs tenía una forma característica de sierra, fue solucionada adecuando la razón de entrenamiento. Esto surgió dado que la razón de aprendizaje se tenía un número muy alto.

En la Figura 4.13 se ilustra el inconveniente que se comentó previamente del optimizador "*Adam*". Se visualiza el aumento entre epochs, teniendo una forma característica de sierra, donde los valores de la validación subían y bajaban de manera inconsistentemente.

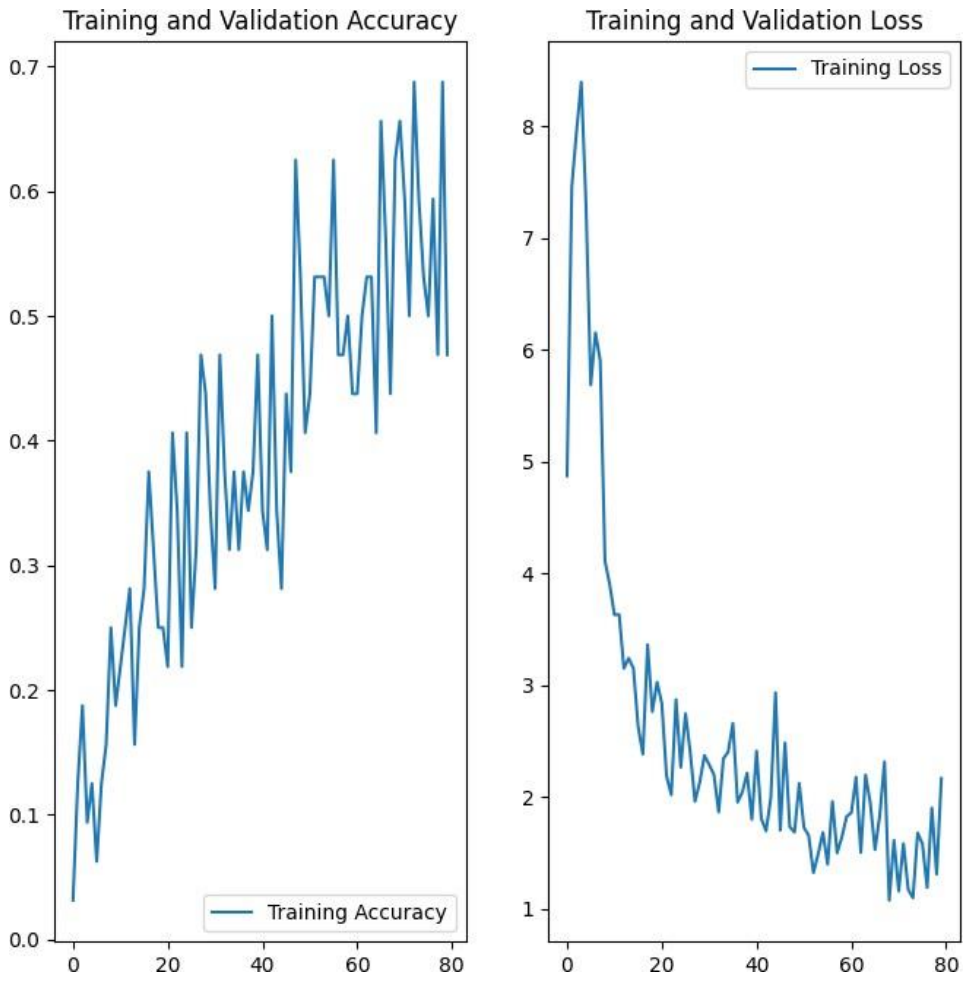


*Figura 4.13 Entrenamiento y pérdida del primer aproximamiento del CNN.*

El entrenamiento se llevó a cabo en 20 epoch. Por cada ciclo se tomaba alrededor de una media de 7 minutos. Para poder contemplar el entrenamiento se tomó aproximadamente 2 horas con 18 minutos. Como se muestra en la Figura 4.14, el entrenamiento empezó en un porcentaje del 5%, alcanzando una máxima del 75%.  Con lo que respecta a la pérdida va de 3 un valor cercano al 0.

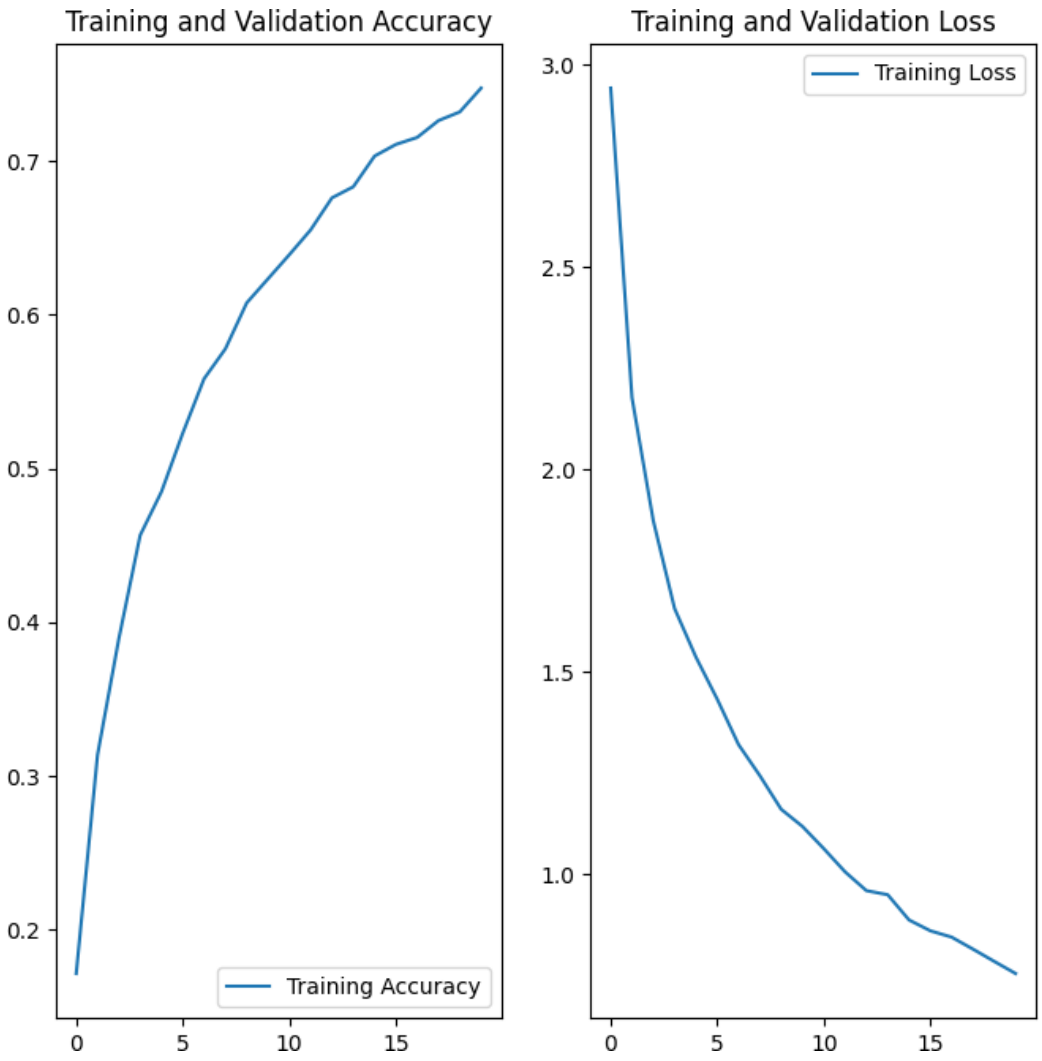


*Figura 4.14 Gráfica de entrenamiento y pérdida.*

La Distancia Euclidiana (DE) es la distancia en línea recta entre dos puntos en un espacio multidimensional. En este caso, cuantifica la disimilitud entre los dos descriptores HOG. Una distancia euclidiana más pequeña indica que los dos descriptores son más similares, mientras que una distancia mayor sugiere una mayor disimilitud.

La Similitud del Coseno (SC) es una medida del coseno del ángulo entre dos vectores. Bajo esa orden de ideas, cuantifica la similitud entre los dos descriptores HOG considerando sus direcciones. Un valor de similitud de coseno de 1 indica que los descriptores son idénticos, mientras que un valor de 0 sugiere que son completamente diferentes.

Se tomó en consideración la seña de la letra A, a la cual se le hizo el cálculo del promedio de estos valores enlistados a continuación:

- Comparado con A → DE: 36.69, SC: 0.29
- Comparado E → DE: 37.24, SC: 0.22
- Comparado O → DE: 37.18, SC: 0.32
- Comparado U → DE: 35.89, SC: 0.20
- Comparado con G → DE: 37.32, SC: 0.22

## 4.4 Entrenamiento de k-Vecinos más próximos (k-NN)

El segundo modelo que se analizó es el de k-Vecinos más próximos (k-NN). Se utilizó el mismo conjunto de datos para el modelo CNN, pero en vez de utilizar la detección de bordes, se utilizó la extracción de características HOG.

El principal componente del modelo k-NN es el uso del parámetro "*k*". Este es un valor que determina la cantidad de vecinos más cercanos que se utilizarán para tomar una decisión o hacer una predicción. Donde "*vecinos*" se refiere a las posibles categorías que se encuentran en el conjunto de datos.

En el modelo tradicional k-NN no existe una fase de entrenamiento, pero dado que se buscó realizar pruebas con distintas distribuciones de datos y valores "*k*", se realizó el siguiente procedimiento.

Se realizaron divisiones de las imágenes en un conjunto de entrenamiento y uno de prueba utilizando un ciclo. Inicialmente, se asignó el 10% de las imágenes para construir el modelo k-NN, mientras que el 90% restante se reservó para las pruebas. Posteriormente, se ajustaron las proporciones en incrementos del 10% hasta llegar al punto en el que se utilizó el 90% de las imágenes para el entrenamiento y el 10% restante para realizar las pruebas.

Se hizo un ciclo anidado, donde se cambiaba el número k, variando en un rango desde 1 hasta 9, incrementando de uno en uno en cada iteración. En cada repetición, se hacía la predicción, se almacenaba, y éste era comprobado con el valor real, dando como resultado el

porcentaje de precisión. El valor k óptimo será analizado en el apartado de resultados. (Fernandez Villan, 2019).

Se realizaron dos iteraciones para analizar el modelo. El primer ciclo, se varió el porcentaje de entrenamiento y de validación. Se inicia con una proporción del 10% de entrenamiento y 90% de validación, aumentando y disminuyendo en 10 puntos porcentuales cada una de las categorías, hasta tener una proporción del 90% de las imágenes para entrenamiento y un 10% para validación.

```
def load_data(test_size=0.2, random_state=42):
  spinner_count = 0
  absolute_path = os.path.dirname(_file_)
  source_img_path = 'dataset/'
  full_path = os.path.join(absolute_path, source_img_path)

  X, y = [], []

  for root, subdirectories, files in os.walk(full_path):
    for file in files:
      print('Processing', '.'*(spinner_count+1), ' '*(2-spinner_count),
end='\r')
      spinner_count = (spinner_count + 1) % 3

      if file.startswith('.') == True:
        continue

      label_image = os.path.basename(root)
      origin_img_path = os.path.join(root, file)

      hand_image = cv2.imread(origin_img_path)

      masked_img = pi.process_image_ycbcr(hand_image,'blur')

      X.append(masked_img)
      y.append(label_image)

   return X, y
```

*Fragmento de código 4.22 Carga de datos.*

El primer paso para obtener el vector HOG, se tiene que definir el descriptor HOG. Está definido por el tamaño de la imagen. Dentro del módulo "*process_image.py*", se programó el *Fragmento de código 4.23* donde se muestra cómo se obtiene el vector a partir del módulo de OpenCV.

```
def  get_hog(size_image):
  """ Get hog descriptor """

  hog = cv2.HOGDescriptor((size_image, size_image), (8, 8), (4, 4), (8,
8), 9, 1, -1, 0, 0.2, 1, 64, True)
  print("hog descriptor size: '{}'".format(hog.getDescriptorSize()))
  return hog
```

*Fragmento de código 4.23 Obtener HOG.*

Igualmente, en el módulo "process_image.py" se definió la función "*get_hog_features*", denotado en el *Fragmento de código 4.24*. En esta función hace la llamada de la función "get_hog" definida en el *Fragmento de código 4.23*.

```
def get_hog_features(im):
  proccess_data = process_image_ycbcr(im, 'blur')
  size_image = len(im)
  hog = get_hog(size_image)

  hog_descriptor = get_hog_descriptor(hog, proccess_data)
  hog_descriptor = np.squeeze(hog_descriptor)
  hog_descriptor = hog_descriptor.astype(np.float32)  # Convert to
np.float32
  hog_descriptor = hog_descriptor.reshape(1, -1)  # Reshape to (1,
num_features)

  return hog_descriptor
```

*Fragmento de código 4.24 Función para obtener el vector HOG.*

Ya dentro del módulo "*knn_with_hog_moments.py*", se hace la carga y el proceso de cada una de las imágenes. Esto se muestra en el *Fragmento de código 4.25*.

```
import process_image as pi
# Load all the signs and the corresponding labels:
X_train, y_train = load_data()

# Compute the descriptors for all the images.
# In this case, the HoG descriptor is calculated
hog_descriptors = []
for img in X_train:
    hog_descriptors.append(pi.get_hog_features(img))
```

*Fragmento de código 4.25 Obtener vector HOG para cada imagen.*

Las imágenes están categorizadas de acuerdo con el carácter que representan. En lugar de categorizar por caracter, es necesario codificar las etiquetas a valores numéricos. Por ejemplo, la letra A se debería de asignar al valor 0. Para esto se usó de la librería "*sklearn*" el módulo de "*preprocessing*" que se encargar de hacer el cambio con la función "*LabelEncoder*", como se muestra el *Fragmento de código 4.26*.

```
import cv2
import numpy as np
from collections import defaultdict
from sklearn import preprocessing

# Create a dictionary to store the accuracy when testing:
results = defaultdict(list)

# Create KNN:
knn = cv2.ml.KNearest_create()

# Label Encoder
le = preprocessing.LabelEncoder()
```

*Fragmento de código 4.26 Encoder label.*

Se realizaron varias iteraciones del modelo k-NN, donde se tenía dos ciclos; en el primero se variaba el porcentaje de datos que se usaban para el entrenamiento, empezando por un 10%, y aumentando de diez en diez, hasta llegar a un 90%. Y, el segundo ciclo, donde se variaban el número de k, iniciando en un valor 1, aumentando de uno en uno hasta llegar a 9.

Para el primer ciclo, se dividen los datos de entrenamiento y de prueba conforme lo indica la variable "*split_values*". En el *Fragmento de código 4.27*, se enseña el algoritmo que obtiene los datos de entrenamiento, almacenada en la variable "*hog_descriptors_train*". Dicha variable es usada como dato de ingreso para la función train de la variable "*knn*", que viene siendo un objeto de OpenCV.

```
# Split data into training/testing:
split_values = np.arange(0.1, 1, 0.1)

for split_value in split_values:
  le.fit(y_train)
  y_train = le.transform(y_train)

  # Split the data into training and testing:
  partition = int(split_value * len(hog_descriptors))
  hog_descriptors_train, hog_descriptors_test = np.split(hog_descriptors,
[partition])
  labels_train, labels_test = np.split(y_train, [partition])

  # Train KNN model
  print('Training KNN model - HOG features')
  knn.train(hog_descriptors_train, cv2.ml.ROW_SAMPLE, labels_train)
```

*Fragmento de código 4.27 Train k-NN model.*

El segundo ciclo de iteración se varió en el valor de k, que denota cuántos vecinos se tomarán en cuenta para la determinación de la clase. El rango de valores va desde 1 a 9, y con los datos que se reservaron para el entrenamiento, se hace predicciones con el modelo

creado, y almacenando el valor de precisión obtenido. Finalmente, en el *Fragmento de código 4.28*, en la última instrucción, se guarda el modelo con el nombre "*knearest_model.xml*".

```
for split_value in split_values:

  # Store the accuracy when testing:
  for k in np.arange(1, 10):
      ret, result, neighbours, dist =
knn.findNearest(hog_descriptors_test, k)
      acc = get_accuracy(result, labels_test)
      print(" {}".format("%.2f" % acc))
      results[int(split_value * 100)].append(acc)

knn.save('knearest_model.xml')
```

*Fragmento de código 4.28 Save k-NN model.*

En la Figura 4.15, se muestra el resultado final del modelado, denotando en la ojiva de color verde olivo, obteniendo un valor del 95%.

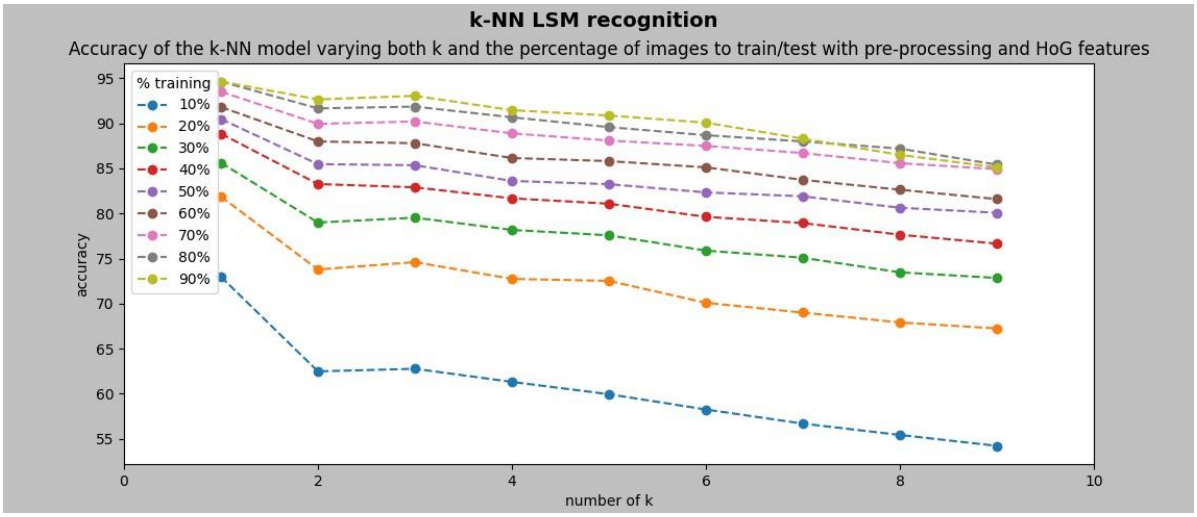


*Figura 4.15 Gráfico comparativo k-NN con características HOG.*

# 5 RESULTADOS

En la Figura 5.1, se explica el entrenamiento para el modelo CNN, el cual empezó en un porcentaje del 5%, alcanzando una máxima del 75%.

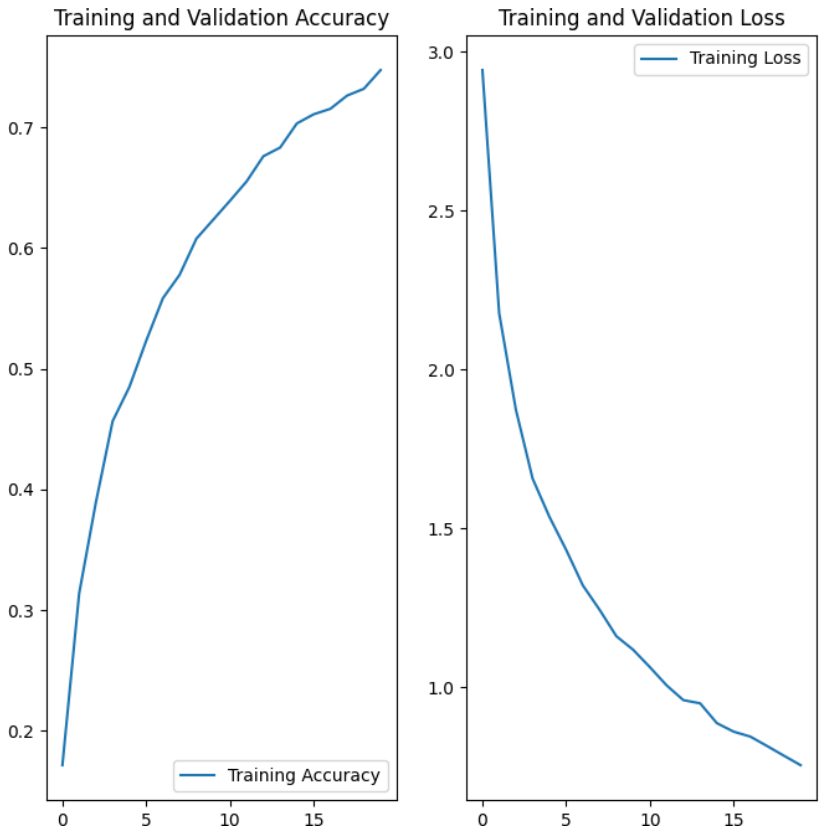


*Figura 5.1 Valores de precisión y pérdida para la CNN.*

En la Figura 5.2 se gráfica los distintos porcentajes de acierto que se obtuvieron con respecto a los distintos valores de k, utilizando el modelo k-NN.

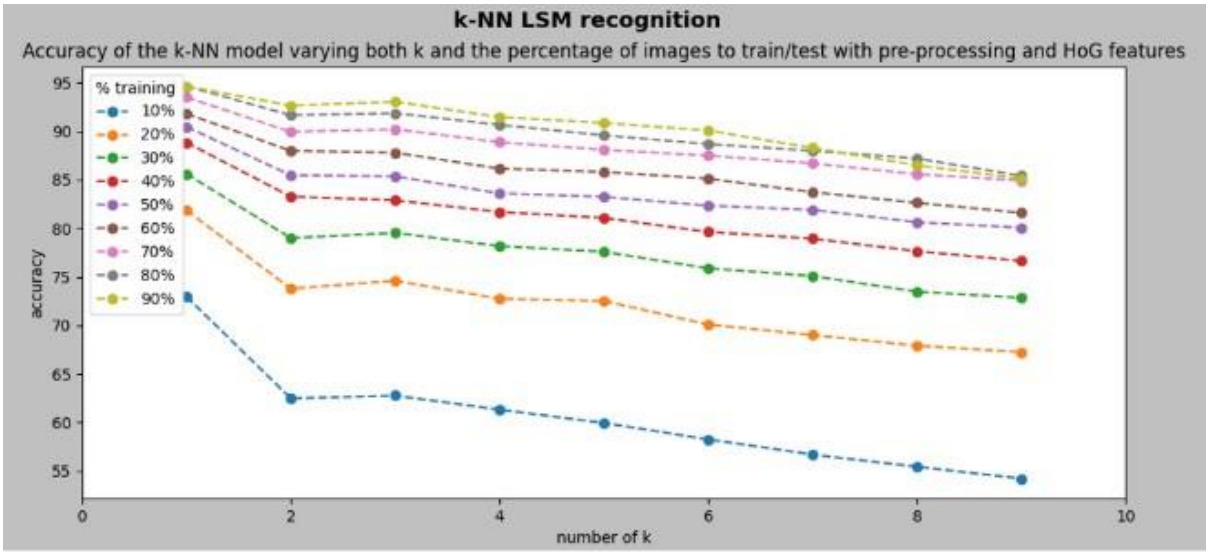


*Figura 5.2 Gráfica de resultados de entrenamiento k-NN.*

Los mejores resultados se vieron cuando se utilizó el 90% de los datos para realizar el entrenamiento, y con k igual a 1.

Como resultado final, se obtuvo hasta un valor del 75% usando CNN, y un valor del 95% para el método de k-NN.

## 5.1 Alcances y limitaciones

El proyecto se centra en el uso de una región de interés (ROI) fija. Esto significa que el usuario selecciona una región específica en la imagen para su procesamiento y clasificación. La aplicación está diseñada para funcionar de manera efectiva dentro de esta limitación y no permite la detección automática de ROI.

El proyecto incluye la capacidad de procesar la imagen dentro de la ROI, aplicando correcciones de exposición y filtros según sea necesario para mejorar la calidad visual de la imagen. El procesamiento de imagen se realizará de manera coherente y controlada para mantener la calidad de la clasificación.

El proyecto se enfoca en clasificar el abecedario de la LSM dentro del ROI, omitiendo la letra Ñ.

Una de las limitaciones clave del proyecto es que los resultados pueden variar significativamente conforme se cambie la iluminación en la imagen. La aplicación no es

capaz de realizar correcciones automáticas adecuadas para todas las condiciones de iluminación y puede tener dificultades en situaciones extremas de alta o baja luminosidad.

El fondo de la imagen, que se encuentra dentro del ROI pero excluye la región de interés (la mano), puede influir en la clasificación. Esto está condicionado especialmente si este se encuentre en un rango de colores parecidos a los del color de la piel, o si se encuentran demasiados elementos que termine detectando algún borde falso.

El programa puede tener una tendencia a clasificar ciertas señas con mayor precisión que con otras. Esto está condicionado a la calidad de los datos de entrenamiento, que a pesar de que se buscó tener el ambiente controlado, esto no fue del todo posible.

## 5.2 Prototipo

La interfaz principal cuenta con dos ventanas. En la primera aparecerá una ventana de visualización en tiempo real que muestra lo que está capturando la cámara del dispositivo donde se está ejecutando. En adición, se muestra un cuadrado azul, el cual es denominado región de interés (ROI por sus siglas en inglés). La imagen procesada se someterá a un proceso de clasificación automática utilizando el método k-NN. Arriba de la región ROI se muestra cuál es la etiqueta de la seña con la información dada, y a su lado un valor numérico que oscila entre el 0.0 y 1.0, donde entre mayor sea el número, mayor certeza se tiene de la predicción de la seña.

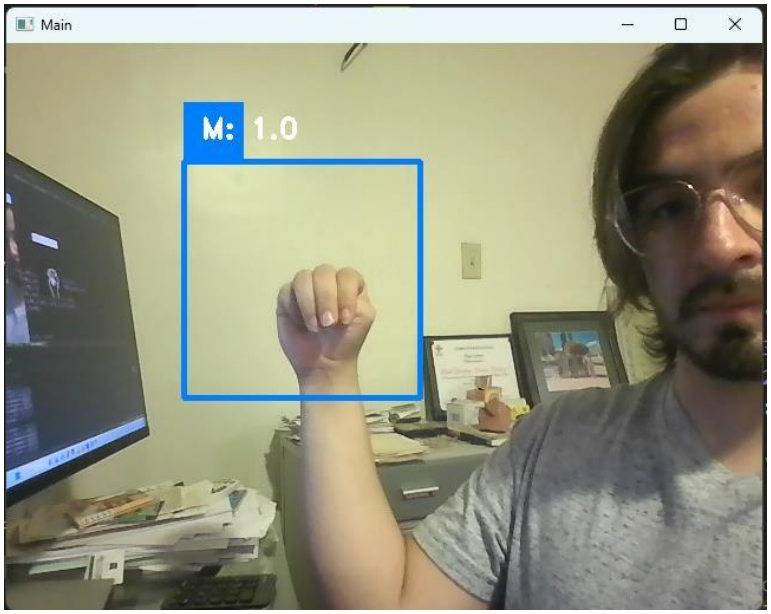


*Figura 5.3 Prototipo: Predicción de letra M.*

Por otra parte, la aplicación procesará automáticamente la imagen capturada desde la ROI. Esto es redactado con mayor profundidad en el capítulo "*Procesamiento de la imagen*" del presente documento. La imagen procesada se mostrará en la pantalla por medio de una nueva ventana.

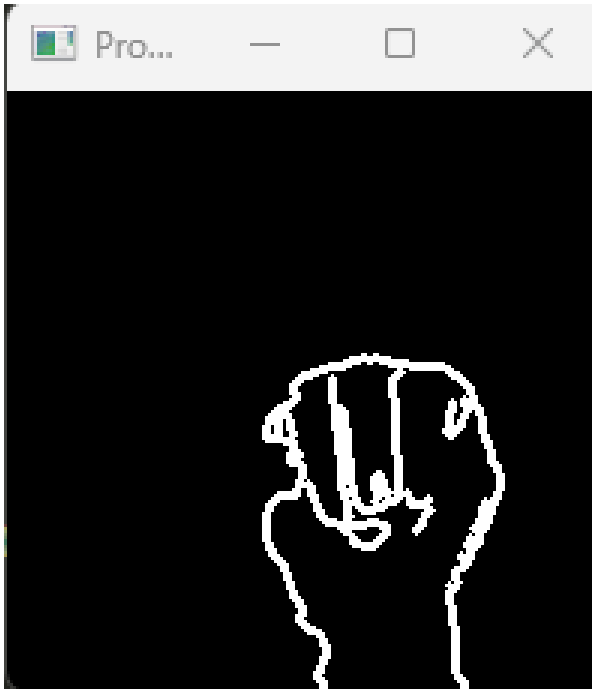


*Figura 5.4 Prototipo: Procesamiento de imagen para la seña de la letra M.*

El caso presentado a continuación, tiene la particularidad de contar con más iluminación, así como variaciones en el fondo.

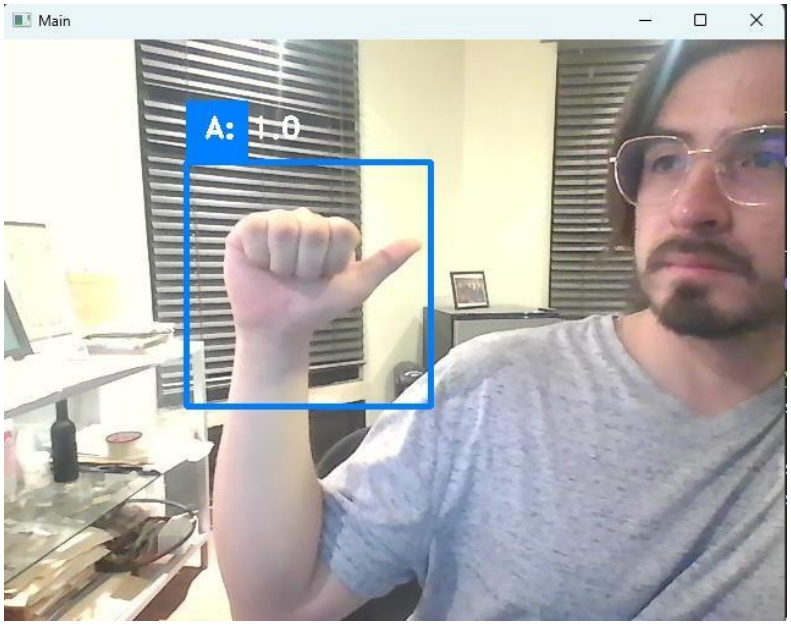


*Figura 5.5 Prototipo: Caso con iluminación alta y fondo no uniforme.*

En la Figura 5.6 se aprecia el resultado del procesamiento de la imagen. Como se observa, el fondo no uniforme generó ruido en la parte de bajo de la mano, pero la información de valor, como podría ser el dedo pulgar, se preservó.

Este ejemplo demuestra cómo las técnicas de procesamiento de imágenes y el modelo de aprendizaje pueden superar desafíos como la iluminación intensa y la presencia de objetos no deseados para lograr una clasificación precisa.


*Figura 5.6 Prototipo: Procesamiento de imagen para la letra A.*

# 6 CONCLUSIONES

En lo que respecta la eliminación del fondo, el método con mejor resultado ha sido la extracción de color YCbCr, con la desventaja que el caso que haya un objeto con la misma tonalidad, este será detectado como parte de la mano.

En cuestión de porcentaje de precisión, k-NN tuvo un valor de precisión del 95%. Adicionalmente, lo que respecta a espacio en disco, el modelo k-NN llegó a pesar 2 GB. En cuestión de tiempo toma alrededor de 1 minuto generar el modelo, y aproximadamente 70 minutos para probar sus distintas variables.

Ya en el aplicativo, las letras con mejor tendencia a ser reconocidas han sido la A, O y B. Las letras que el algoritmo suele tener un resultado erróneo son Q, X y Z.

# 7 RECOMENDACIONES

Se recomienda que, en futuros trabajos de investigación, se exploraren diferentes enfoques y técnicas de reconocimiento de patrones visuales a través de muestras de vídeos, haciendo énfasis en el uso de herramientas accesibles, como lo es una cámara web en vez del uso de una cámara OAK-D, para realizar un modelo para el reconocimiento de ideogramas. Además, considerar la implementación de interfaces de usuario intuitivas y accesibles, que permitan a los usuarios interactuar de manera fluida y eficiente con los vídeos y las representaciones visuales de palabras u oraciones.

## 7.1 EXPERENCIA PERSONAL PROFESIONAL ADQUIRIDA

Mi trabajo de tesis me brindó la invaluable oportunidad de adquirir una experiencia profesional significativa en el campo de la inteligencia artificial y la visión computacional. Durante el desarrollo de mi investigación, tuve la responsabilidad de diseñar, implementar y afinar algoritmos de estas disciplinas. Esta experiencia profesional en el uso de la inteligencia artificial y la visión computacional no sólo enriqueció mi conjunto de habilidades, sino que también me proporcionó una base sólida para futuros proyectos.

Como experiencia personal, el trabajo de tesis no sólo me permitió adquirir una valiosa experiencia profesional en el uso de algoritmos de inteligencia artificial y visión computacional, sino que también me brindó la oportunidad de acercarme a un sector de la

población minoritaria de una manera muy significativa. En particular, mi investigación se centró en el desarrollo de herramientas y sistemas de reconocimiento de lenguaje de señas asistidos por inteligencia artificial, diseñados específicamente para personas sordomudas. Esta inmersión en la problemática de esta comunidad me sensibilizó profundamente ante las barreras de comunicación a las que se enfrentan diariamente. A través de mi trabajo, no sólo buscaba avanzar en la tecnología, sino también contribuir a mejorar la calidad de vida y la inclusión social de las personas sordomudas al proporcionarles herramientas más efectivas y accesibles para la comunicación.

## 7.2 COMPETENCIAS DESARROLLADAS Y/O APLICADAS

En el transcurso de mi trabajo de tesis, me sumergí en un riguroso proceso de investigación y desarrollo de competencias esenciales. La investigación se convirtió en el eje central de mi proyecto, ya que busqué profundizar en el campo de la inteligencia artificial y la visión computacional, tratando en medida de lo posible de contar con un catálogo amplio de fuentes de información. Además, la documentación se convirtió en una destreza crucial, ya que era un paso necesario para llevar registro de los experimentos, ayudando a comunicar de manera clara y efectiva los hallazgos y conclusiones. En última instancia, estas competencias de investigación y documentación fueron cruciales para el éxito de mi proyecto de tesis.

# 8 BIBLIOGRAFÍA


Ale, N. A., & Fabián, J. (2019). *Detección del estado fisiológico de los ojos en Conductores mediante técnicas de visión artificial.* Universidad de Tarapacá.

Ansari, S. (2020). *Building Computer Vision Applications Using Artificial Neural Networks.* Centreville, VA, USA: Apress.

*APLICACIÓN DE TÉCNICAS DE VISIÓN COMPUTACIONAL EN LA PRUEBA DE PAPANICOLAOU* . (16 de Julio de 2012). (Universidad Central Marta Abreu) Recuperado el 13 de Septiembre de 2020, de https://www.medigraphic.com/pdfs/medicentro/cmc-2012/cmc123i.pdf

Cruzado, M., & Montiel, V. y. (8 de Julio de 2011). *Obtención de los parámetros ópticos de la piel usando algoritmos genéticos y MCML*. (Insituo NAciona de Astrofíscia Óptica y Electrónica) Recuperado el 13 de Septiembre de 2020, de http://www.scielo.org.mx/pdf/rmf/v57n4/v57n4a13.pdf

Dey, S. (2020). *Python Image Processing Cookbook.* Birmingham, UK: Packt Publishing Ltd.

Fernandez Villan, A. (2019). *Mastering OpenCV 4 with Python.* Birmingham, UK: Packt Publishing.

Gobierno de México. (10 de Agosto de 2016). *Lengua de Señas Mexicana (LSM)*. (Gobierno de México) Recuperado el 9 de Mayo de 2020, de https://www.gob.mx/conadis/articulos/lengua-de-senas-mexicana-lsm?idiom=es

Gollapudi, S. (2019). *Learn Computer Vision Using OpenCV.* Hyderabad, Telangana, India: Apress.

INEGI. (28 de Abril de 2020). *ESTADÍSTICAS A PROPÓSITO DEL DÍA DEL NIÑO.* Obtenido de INEGI: https://www.inegi.org.mx/contenidos/saladeprensa/aproposito/2020/EAP_Nino.pdf

Islam, K. T., & Raj, R. G. (2017). *Real-Time (Vision-Based) Road Sign Recognition Using an Artificial Neural Network.* MDPI AG.

Kenneth, M., & Córdova, D. (2023). Reconocimiento de la Lengua de Señas Mexicana. *Nthe*, 27-33.

Kishore, P. V., Rao, G. A., Kumar, E. K., Kumar, M. T., & Kumar, D. A. (2018). *Selfie Sign Language Recognition with Convolutional Neural Networks.* Modern Education and Computer Science Press.

Lugo-Reyes, S., Maldonado-Colín, G., & Murata, C. (Abril de 2014). *Inteligencia artificial para asistir el diagnóstico clínico en medicina*. (Revista Alergia México) Recuperado el 13 de Septiembre de 2020, de https://www.redalyc.org/pdf/4867/486755034010.pdf

Magaña Z., J. B. (19 de Septiembre de 2017). *Estimación de la Distancia a un Objeto con Visión Computacional*. (Universidad Autónoma de Yucatán) Recuperado el 13 de Septiembre de 2020, de https://www.redalyc.org/pdf/467/46753192004.pdf

Mancilla Morales, E., Vázquez Aparicio, O., Arguijo, P., Meléndez Armenta, R. Á., & Vázquez López, A. H. (2019). Traducción del lenguaje de señas usando visión por computadora. *Research in Computing Science*, 79-89.

Nope R., S. E., Loaiza C., H., & Caicedo B., E. (2008). Estudio comparativo de técnicas para el reconocimiento de gestos por visión artificial . *Revista Avances en Sistemas e Informática*, 127-134.

OpenCV. (26 de Mayo de 2023). *Canny Edge Detection* . Obtenido de OpenCV: https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html

OpenCV. (19 de Abril de 2023). *OpenCV-Python Tutorials* . Obtenido de OpenCV: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

Patterson, J., & Gibson, A. (2017). *Deep Learning.* Sebastopol, California: O'Reilly Media.

Patterson, J., & Gibson, A. (2017). *Deep Learning.* California: O'Reilly Media, Inc.

Pichucho, J. P., Constante, P. N., Gordón, A. M., & Mendoza, D. J. (2019). Interpretación de lenguaje de señas ecuatoriano empleando visión por computador. *Revista Ibérica de Sistemas e Tecnologias de Informação; Lousada*, 960-971.

Raviteja, C., Gayathri, N., & Thiyaneswaran, N. (2022). Analysing The Skin Wound Texture With Edge Detection Method Using Prewitt And Comparing Healing Rate With Canny Edge Detection. *2022 14th International Conference on Mathematics, Actuarial Science, Computer Science and Statistics (MACS)*, 1-5.

Rosebrock, A. (5 de Octubre de 2015). *OpenCV Gamma Correction*. Obtenido de pyimagesearch: https://pyimagesearch.com/2015/10/05/opencv-gamma-correction/

Serafín, M., & Gónzález, R. (2011). *manos con voz diccionario de lengua de señas mexicana.* México, D.F.: CONAPRED.

Shaik, K. B., P, G., Kalist, V., Sathish, B., & Jenitha, J. M. (2015). Comparative Study of Skin Color Detection and Segmentation in HSV and YCbCr Color Space. *Procedia Computer Science 57*, 41-48.

Shanmugamani, R. (2018). *Deep Learning for Computer Vision.* Birmingham, UK: Packt Publishing.

Sucar, L. E., & Gómez, G. (s.f.). *Visión Computacional.*

Xiujuan Chai, G. L. (s.f.). *Sign Language Recognition and Translation with Kinect.* Beijing, China: Institute of Computing Technology, CAS .