

UNIVERSIDAD: RAMON LLULL

LENGUAJE DE SIGNOS A TEXTO

Trabajo Final de Grado

Tutoras:

Núria Valls Canudas y Elisabet Golobardes Ribé

Autora:

Mar Galiana Fernández

Índice

1. Resumen	2
2. Introducción	3
3. Estado del arte	4
3.1. El lenguaje de signos	4
3.2. Aproximaciones actuales	6
3.2.1. Datos	7
3.2.2. Técnicas	12
3.2.3. Resultados	13
4. Fundamentos	16
5. Hardware	17
6. Experimentación	18
6.1. Experimentación 1: Búsqueda del dataset	18
6.2. Experimentación 2: Uso de redes neuronales	20
6.3. Experimentación 3: Uso de árboles de decisión	28
6.4. Comparativa de las experimentaciones 2 y 3	32
6.5. Experimentación 4: Incremento dataset	34
6.6. Experimentación 5: Mejorar rendimiento de la CNN	40
6.6.1. Tamaño del <i>Batch</i> y número de <i>Epochs</i>	41
6.6.2. Algoritmos de optimización	43
6.6.3. Índice de aprendizaje e impulso	44
6.6.4. Inicialización del peso de la red	46
6.6.5. Función de activación neuronal	48
6.6.6. Regulación del <i>Dropout</i>	49
6.6.7. Número de neuronas	52
6.6.8. Resultados de la optimización	53
6.7. Comparativa de las experimentaciones 4 y 5	58
6.8. Experimentación 6: Uso de clasificadores binarios	60
7. Propuesta	66
8. Costes del proyecto	67
9. Conclusiones y líneas de futuro	68
9.1. Conclusiones	68
9.2. Líneas de futuro	68
A. Anexo	69
A.1. Proyecto Github	69

1. Resumen

2. Introducción

3. Estado del arte

El análisis del estado del arte, que se realizará en este proyecto, hace referencia a las tecnologías y ayudas que disponen las personas sordomudas a la hora de comunicarse con una persona que no sepa el lenguaje de signos.

Esta comunicación se dificulta, no solo por el echo de que la mayoría de la población no tiene los conocimientos necesarios para poder comunicarse en un lenguaje no verbal; sino por el echo de que existen unos 140 lenguajes de signos, con sus respectivos alfabetos y gesto, según la organización Ethnologue [1]. A demás, la mayoría de las personas que nacieron sin la capacidad de oír no saben leer.

3.1. El lenguaje de signos

It is estimated that over 5 % (466 million people) of the global population are deaf. [2].

<https://arxiv.org/pdf/1803.10435.pdf> – types of gesture

introducción, es oro, – file:///Users/margaliana/Downloads/appsci-11-05594.pdf

The general SLR problem includes the following three different tasks: 1. Static or continuous letter/number gesture recognition (classification problem); 2. Static or continuous single word recognition (classification problem); 3. Sentence-level sign language recognition (NLP problem).

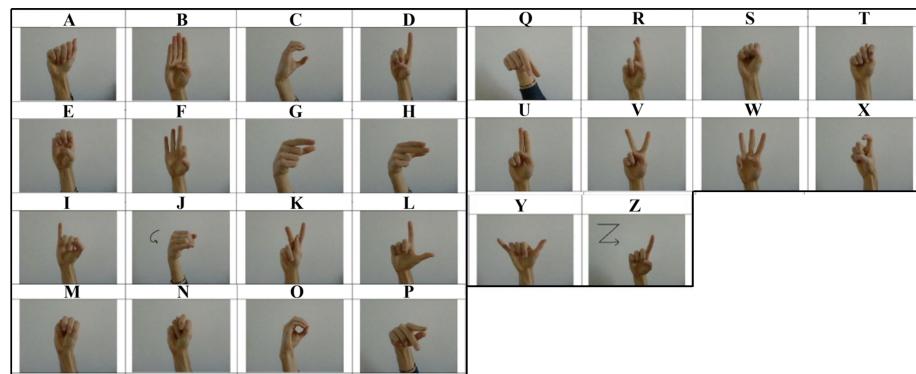


Figura 1: Alfabeto del lenguaje de signos americano (ASL). Todos los signos son estaticos, excepto la 'J' y la 'Z'.

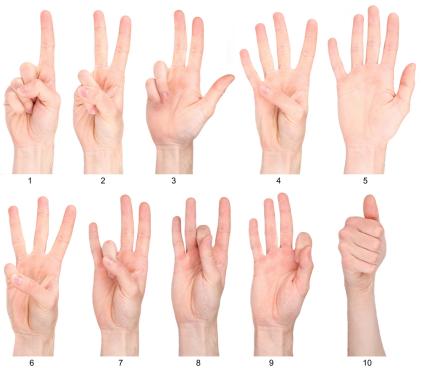


Figura 2: Números del lenguaje de signos americano (ASL),

3.2. Aproximaciones actuales

Durante las últimas décadas, se han realizado diversas investigaciones en el campo del reconocimiento del lenguaje de signos (SLR). La mayoría se pueden dividir en dos enfoques:

- Los que hacen un reconocimiento de los signos estáticos, como es el abecedario, los números, algunas expresiones... Estos signos no tienen un movimiento, son capaces de ser representados a partir de un único gesto.
- Los que hacen un reconocimiento donde se incluyen los signos dinámicos, los cuales suceden en un espacio temporal.

Hay otra categorización en las investigaciones realizadas, la cual depende del modo de obtención de los datos. Podemos encontrar dos enfoques:

- Un enfoque basado en la visión, donde se hace uso de cámaras o detectores de imágenes. En estos se encuentran limitaciones a la hora de realizar y analizar las imágenes, se debe tener en cuenta la iluminación, el color y el fondo utilizado. Es importante poder diferenciar la parte del cuerpo que realiza el signo de la parte de la imagen que no aporta información.
- Un enfoque no basado en la visión. En este se hace uso de guantes, sensores de movimiento, rastreadores... Estos también presentan limitaciones, ya que se requiere de un coste adicional de *Hardware* y restringe los movimientos permitidos por los interpretes.

Durante la realización de este apartado se han investigado diversos artículos y libros de proyectos realizados sobre el reconocimiento del lenguaje de signos. Finalmente se ha dado más importancia a diez proyectos realizados con diferentes características, las cuales son: el tipo de datos empleados, las técnicas utilizadas y los resultados obtenidos.

3.2.1. Datos

En los artículos investigados se diferencian tres tipos de datos utilizados para el entrenamiento del modelo predictivo implementado. Estos son: imágenes, vídeos o coordenadas.

Las **imágenes** se suelen utilizar en los reconocimientos estáticos, ya que no requieren de ningún espacio temporal.

En el artículo *Sign language recognition using convolutional neural networks* [3], se adquieren las imágenes a partir de una cámara web. Una vez adquiridas son procesadas de tres formas:

- En el primer paso se aplica el algoritmo *Hue, Saturation, Value* (HSV), donde los fondos son detectados y eliminados. Este modelo divide el color de una imagen en 3 partes: la tonalidad, la saturación y el valor.
- El segundo paso de este procesamiento consiste en la segmentación, transformando las imágenes a escala de grises. Citando el artículo: "Por mucho que este proceso resulte en la pérdida de color en la región del gesto de la piel, también mejorará la robustez de nuestro sistema a los cambios de iluminación" [3].

Una vez realizada la transformación a una escala de grises, se modifica el tamaño a un marco de 64 por 64 píxeles.

- Por último, se realiza la extracción de características. "En nuestro caso, las características que se consideran cruciales son los píxeles binarios de las imágenes. Escalar las imágenes a 64 píxeles nos ha llevado a obtener características suficientes para clasificar de manera efectiva los gestos del lenguaje de señas estadounidense. En total, tenemos 4096 características, obtenidas después de multiplicar 64 por 64 píxeles." [3].

El dataset utilizado dispone de 10 signos del alfabeto del lenguaje de signos americano (ASL). Las letras son: 'A', 'B', 'C', 'D', 'K', 'N', 'O', 'T' e 'Y'. "Nuestro conjunto de datos tiene 2000 imágenes de gestos de signos estadounidenses, de las cuales 1600 son para entrenamiento y el resto 400 son para propósitos de prueba" [3].

En el artículo *Deep Learning-based sign language translation system* [4], se implementa un sistema capaz de detectar la posición de las manos a partir de imágenes obtenidas por cámara. Se utiliza un dataset que contiene 24 gestos

del lenguaje de signos americano (ASL).

Los artículos analizados también hacen uso de los **vídeos** como tipos de muestra, estas son los más comunes entre los SLR. Se suelen utilizar en el reconocimiento de gestos dinámicos, ya que requieren de una secuencia de imágenes para detectar la totalidad del gesto.

Un ejemplo del uso de este tipo de muestra es el artículo *Large-scale Video Classification with Convolutional Neural Networks* [5]. En este se explica el porque es más difícil el procesado de este tipo de datos, “a diferencia de las imágenes que se pueden recortar y volver a escalar a un tamaño fijo, los vídeos varían mucho en extensión temporal y no se pueden procesar fácilmente con una arquitectura de tamaño fijo.” [5]

Los datos son tratados como un conjunto de secciones de un mismo vídeo de tamaño fijo. “Dado que cada sección contiene varios fotogramas contiguos en el tiempo, podemos ampliar la conectividad de la red en la dimensión temporal para conocer las características espacio-temporales.” [5]

En el articulo *An Improved Sign Language Translation Model with Explainable Adaptations for Processing Long Sign Sentences* [6] se utiliza un dataset del lenguaje de signos alemán. Este contiene “4.839 terminologías, 8.257 videoclips, 947.756 fotogramas y 113.717 palabras en total.” [6]. Estos valores se muestran en la imagen 3.

	Train	Dev	Test
Vocab.	2,887	951	1,001
Clips	7,096	519	642
Frames	827,354	55,775	64,627
Tot. words	99,081	6,820	7,816

Figura 3: Estadísticas del dataset alemán utilizado en el artículo *An Improved Sign Language Translation Model with Explainable Adaptations for Processing Long Sign Sentences* [6].

El *Fingerspelling recognition with semi-Markov conditional random fields* [7] es otro articulo en el que se implementa un SLR a partir de vídeos. En este se estudia el abecedario americano (ASL), con la intención de poder deletrear, analizando las estadísticas de sus secuencias.

“Los experimentos utilizan grabaciones de vídeo de cuatro intérpretes nativos de ASL. Los datos se registraron a 60 fotogramas por segundo en un entorno de estudio” [7]. Con tal de poder guardar las muestras se creó el escenario mostrado en la imagen 4, con la intención de facilitar la obtención de muestras.



Figura 4: Escenario para la recopilación de datos en el artículo *Fingerspelling recognition with semi-Markov conditional random fields* [7]

“Cada intérprete representó una lista de 300 palabras a medida que aparecían en la pantalla de una computadora. Cada palabra se deletreaba dos veces, lo que arrojaba 600 instancias de palabras por cada intérprete. Las listas contenían palabras en inglés y extranjeras, incluidos nombres propios y sustantivos comunes en inglés” [7].

Una vez obtenidas todas las muestras y creado el dataset, se dividió en una proporción 80:10:10. El 80 % de los datos se utiliza como un conjunto de entrenamiento, el 10 % como un conjunto de desarrollo para ajustar los hiperparámetros y el 10 % restante como un conjunto para la prueba final.

Otros estudios, como es el caso de *Real-Time American Sign Language Recognition Using Desk and Wearable Computer Based Video* [8] han querido analizar en qué posición de la cámara se adquirían mejores resultados. La imagen 5 muestra las dos posiciones, la primera desde una visión frontal y la segunda desde una visión satélite, con la cámara situada en el sombrero del intérprete.



Figura 5: Posición de la cámara en dos obtenciones de muestras. La primera con una posición frontal y la segunda con una visión satélite con la cámara situada en el sombrero. Obtenido del estudio *Real-Time American Sign Language Recognition Using Desk and Wearable Computer Based Video* [8].

Las muestras contienen una visión global del interprete, estudios como *Deep Learning-Based Approach for Sign Language Gesture Recognition With Efficient Hand Gesture Representation* [9] utilizan frameworks como OpenPose para la detección de la posición de las manos de los interpretes.

Los últimos datos utilizados como muestras de un modelo predictivo son las **coordenadas**. Se obtienen a partir de componentes, tales como: sensores de movimiento, guantes, rastreadores... Su utilización es menos frecuente, ya que es incómodo para el interprete, su uso no es recomendable cuando llueve y es difícil de transportar ya que requieren de una computadora [3].

Otro tipo de componente que se puede utilizar para la obtención de este tipo de datos es el enfoques basados en modelos 3D, como se utiliza en el estudio *Exploiting Recurrent Neural Networks and Leap Motion Controller for Sign Language and Semaphoric Gesture Recognition* [10].

“El método propuesto utiliza un modelado basado en el esqueleto, donde una representación virtual de manos esqueléticas (u otras partes del cuerpo) se asigna a segmentos específicos” [10].

Cada muestra se identifica a partir del ángulo de la articulación junto con las longitudes del segmento. Luego, “mide las variaciones en el tiempo de las articulaciones del esqueleto cuyas coordenadas espaciales son adquiridas por un *Leap Motion Controller* (LMC)” [10].

Se comprobó que para adquirir una mayor precisión también se debía considerar “la información de los ángulos intradedo y la información espacial de la palma de la mano” [10].

Por último, otro componente a utilizar es el sistema de posicionamiento magnético, utilizado en el estudio *Gesture Recognition of Sign Language Alpha-*

bet Using a Magnetic Positioning System [11].

Este trabajo propone el uso de un sistema de posicionamiento magnético para reconocer los gestos estáticos asociados al alfabeto del lenguaje de signos. “Mide la posición en 3D y la orientación de los dedos dentro de un volumen operativo de aproximadamente 30 x 30 x 30 cm, donde los nodos receptores se colocan en posiciones conocidas” [11]. Esta estructura se muestra en la imagen 6.

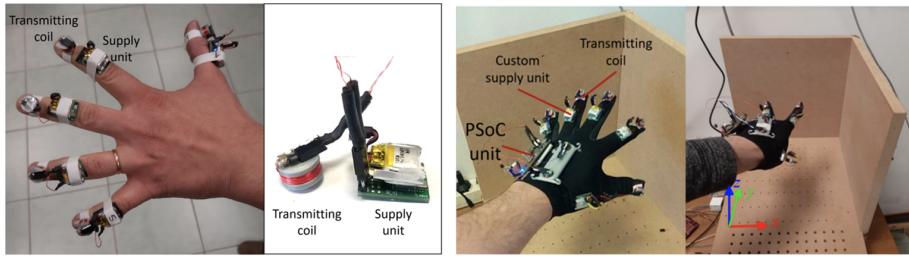


Figura 6: Ejemplo del uso de un sistema de posicionamiento magnético, obtenido del estudio *Gesture Recognition of Sign Language Alphabet Using a Magnetic Positioning System* [11]

Una ventaja de este componente, en comparación con los otros mencionados, es que “mide la posición absoluta de los dedos. Permite una alta precisión y un seguimiento sin desviaciones. Además, funciona también en presencia de obstrucciones causadas por objetos o partes del cuerpo” [11].

Una vez se obtienen todas las muestras, se divide en dataset en una proporción 90:10, el 90% para el entrenamiento y el 10% restante para las pruebas finales.

3.2.2. Técnicas

Entre los artículos investigados se diferencian ocho tipos de técnicas para entrenar el modelo predictivo.

El primero utilizado es el modelo de Márkov,

3.2.3. Resultados

Una vez se ha analizado toda la información de los destinos artículos y libros que se han consultado en el apartado , se ha creado la tabla 1. Esta es un resumen de los proyectos más significativos que se han encontrado, donde se esquematizan los resultados obtenidos dependiendo de la técnica y los datos utilizados.

Es difícil hacer una comparación entre ellos, ya que cada autor ha mostrado la precisión del modelo predictivo a partir de diferentes parámetros. Estos han sido: la *accuracy*, el *letter error rate* (LER), el error cuadrático medio (RMSE), el coste de memoria y el coste de tiempo.

Aún así, nos podemos hacer una idea de cuales pueden ser las técnicas mas recomendadas dado los datos utilizados.

TÉCNICAS / DATOS	Imágenes	Vídeos	Coordenadas
Markov	—	Entre un 92 % y un 97 % de accuracy [8]	—
Semi-Markov con Neural Network	—	Mejora en el LER de un 16,3 % a un 11,6 % [7]	—
Support Vector Machines (SVM)	—	—	Entre 77,19 % y 96,79 % de accuracy [11]
Convolutional Neural Networks (CNN)	Accuracy superior al 90 % [3]	Entre 55,3 % y 63,9 % de accuracy [5]	—
CNN y SVM	Accuracy del 99,9 % RMSE de 0,0126 [4]	—	—
Recurrent Neural Network (RNN)	—	71,9 % de accuracy [2]	Entre 91,43 % y 97,62 % de accuracy [10]
Gated recurrent unit (GRU) y el algoritmo heurístico FSDC	—	Reducción de un 9,28 % de los datos de entrenamiento. Reducción de un 10 % del tiempo de procesamiento. [6]	—
3DCNN	—	87,69 % de accuracy [9]	—

Tabla 1: Comparativa de los resultados obtenidos en diez proyectos dependiendo de las técnicas y los tipos de datos utilizados.

Siguiendo la tabla 1, si se utiliza un dataset estático, sería recomendable utilizar un modelo predictivo basado en redes neuronales convolucionales con el algoritmo de aprendizaje supervisado SVM como clasificador.

Si se está utilizando un dataset dinámico, donde se utilizan datos en formato de vídeo, una muy buena opción sería hacer uso de las redes neuronales convolucionales en 3D.

Por último, si los datos se están obtenido en formato de coordenadas, ya sea por el uso de guantes, modelos basados en 3D o un sistema con posicionamiento magnético, una red neuronal recurrente podría dar una precisión muy elevada.

Aún así, se debe tener en cuenta que los resultados no solo dependen del tipo de datos utilizado, pero la tabla 1 es una buena forma de esquematizar los resultados obtenidos para facilitar su evaluación.

4. Fundamentos

EXPLICAR:

- Redes neuronales: los diferentes tipos que hay (sobretodo la convolucional), sus diferencias, ventajas y desventajas. ¿Cuáles serían beneficiosas para este problema?
- Árboles de decisión: tipos (sobretodo el Gradient Boosting) y comparativa, ventajas y desventajas. ¿Cuáles serían beneficiosas para este problema?
- algoritmos de optimización utilizados en el experimento 5:
 - **SGD:**
 - **RMSprop:**
 - **Adagrad:**
 - **Adadelta:**
 - **Adam:**
 - **Adamax:**
 - **Nadam:**

Link útil: <https://ruder.io/optimizing-gradient-descent/>

- Network Weight Initialization: uniform, lecun uniform, normal, zero, glorot normal, glorot uniform, he normal y he uniform.
Link que pot ser útil: <https://keras.io/api/layers/initializers/>
- Funciones de activación: relu, softmax, softplus, softsign, tanh, sigmoid, hard_sigmoid y linear. Link: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>
<https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/>
<https://www.v7labs.com/blog/neural-networks-activation-functions>
<https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>

5. Hardware

A explicar:

- Mac
- Google collab (git)
- Jupyter notebook

Explicar que es cada cosa, ventajas y desventajas

6. Experimentación

El objetivo de este proyecto ha sido poder interpretar el lenguaje de signos y mostrar su significado en formato texto, a partir de imágenes. Para poder alcanzar este objetivo se han hecho diversas pruebas y experimentos.

6.1. Experimentación 1: Búsqueda del dataset

En la primera versión se ha utilizado un dataset de la comunidad de datos Kaggle. [12] Consiste en 37 gestos diferentes que representan el abecedario inglés, con las letras de la 'A' a la 'Z', los números del 0 al 9 y el carácter espacio. Cada gesto dispone de 1500 imágenes de 50x50 píxeles. Cada imagen está repetida, ya que hay dos versiones de la misma, una ya preprocessada con la intención de mejorar el entrenamiento y las pruebas de las redes neuronales, y otra versión con las imágenes originales en color. Se han utilizado las dos versiones para poder comparar sus resultados.

En la figura 7 se muestran diez imágenes que hacen referencia a diez gestos obtenidos del dataset de la segunda versión mencionada.



Figura 7: Imágenes obtenidas del dataset *Sign Language Gesture Images Dataset* [13] en color.

Como se puede contemplar en la figura 7, todas las imágenes han sido tomadas con un fondo de un mismo color, centrándose únicamente en la mano que realiza el gesto. La tonalidad de las imágenes cambia dependiendo del gesto, pero todas las imágenes que hacen referencia al mismo, contienen una tonalidad similar.

A continuación, en la figura 8, se muestran otras diez imágenes que hacen referencia a los mismos gestos que en la figura 7, pero estas han sido obtenidas de la primera versión mencionada.



Figura 8: Imágenes obtenidas del dataset *Sign Language Gesture Images Dataset* [13] preprocesadas con la intención de mejorar el entrenamiento de las redes neuronales.

En total, este dataset dispone de unas 55.500 imágenes, englobando las dos versiones.

6.2. Experimentación 2: Uso de redes neuronales

Una vez obtenido el dataset, se decidió empezar utilizando redes neuronales, ya que, como se ha podido apreciar en el estado del arte, es una de las soluciones más recomendadas y utilizadas para el reconocimiento de imágenes.

Se analizaron diferentes tipos de redes neuronales, como se especifica en los fundamentos, en el apartado 4,. Finalmente se decidió empezar por la implementación de una básica y de una convolucional, con el objetivo de poder comparar la precisión de cada una de ellas.

La red neuronal básica se implementó con una estructura de dos capas. Los datos de entrada, que son las imágenes pertenecientes al dataset ya especificado, son previamente procesados, para que sean de una dimensión de 150*150 píxeles. La última capa tienen una dimensión de 38 elementos.

FALTA EXPLICAR BASIC NN

En la figura que hay a continuación, se muestra el resultado del entrenamiento cuando se utiliza el dataset mencionado con las imágenes a color, pero antes de proceder con el entrenamiento son procesadas en blanco y negro, para disminuir el coste de memoria.

```

Model: "sequential"
-----
Layer (type)      Output Shape       Param #
=====
dense (Dense)     (None, 100)        2250100
=====
dense_1 (Dense)   (None, 38)         3838
=====
Total params: 2,253,938
Trainable params: 2,253,938
Non-trainable params: 0
-----
Epoch 1/10
304/304 [=====] - 28s 20ms/step - loss: 3.2980 - accuracy: 0.1572 - val_loss: 1.9880 - val_accuracy: 0.6661
Epoch 2/10
304/304 [=====] - 5s 18ms/step - loss: 1.7224 - accuracy: 0.6878 - val_loss: 1.1555 - val_accuracy: 0.8127
Epoch 3/10
304/304 [=====] - 5s 18ms/step - loss: 1.0269 - accuracy: 0.8228 - val_loss: 0.7742 - val_accuracy: 0.8527
Epoch 4/10
304/304 [=====] - 6s 18ms/step - loss: 0.6952 - accuracy: 0.8701 - val_loss: 0.5717 - val_accuracy: 0.8846
Epoch 5/10
304/304 [=====] - 5s 18ms/step - loss: 0.5093 - accuracy: 0.9025 - val_loss: 0.4581 - val_accuracy: 0.8943
Epoch 6/10
304/304 [=====] - 6s 18ms/step - loss: 0.4034 - accuracy: 0.9202 - val_loss: 0.3776 - val_accuracy: 0.9111
Epoch 7/10
304/304 [=====] - 5s 18ms/step - loss: 0.3270 - accuracy: 0.9351 - val_loss: 0.3146 - val_accuracy: 0.9299
Epoch 8/10
304/304 [=====] - 5s 18ms/step - loss: 0.2750 - accuracy: 0.9462 - val_loss: 0.2726 - val_accuracy: 0.9417
Epoch 9/10
304/304 [=====] - 5s 18ms/step - loss: 0.2344 - accuracy: 0.9549 - val_loss: 0.2399 - val_accuracy: 0.9505
Epoch 10/10
304/304 [=====] - 5s 18ms/step - loss: 0.2035 - accuracy: 0.9635 - val_loss: 0.2083 - val_accuracy: 0.9571

```

Figura 9: Resultado de la ejecución de la función *summary* con la red neuronal básica.

La red neuronal convolucional fue implementada a partir de cinco capas. La primera tiene dos dimensiones, los inputs son introducidos es una matriz de 150x150 píxeles. La última capa es de una única dimensión de 38 elementos.

FALTA EXPLICAR CNN

La figura que se muestra a continuación contiene el resultado del entrenamiento de la red neuronal convolucional que se acaba de especificar. El dataset utilizado es el que contiene las imágenes a color, pero igual que se ha hecho con la red neuronal básica, las imágenes han sido procesadas en blanco y negro, para disminuir el coste de memoria.

```

Model: "sequential"
-----
Layer (type)      Output Shape       Param #
-----
conv2d (Conv2D)    (None, 148, 148, 25)   250
-----
max_pooling2d (MaxPooling2D) (None, 148, 148, 25)   0
-----
flatten (Flatten) (None, 547600)        0
-----
dense (Dense)     (None, 100)          54760100
-----
dense_1 (Dense)   (None, 38)           3838
-----
Total params: 54,764,188
Trainable params: 54,764,188
Non-trainable params: 0
-----
Epoch 1/10
304/304 [=====] - 319s 1s/step - loss: 2.5943 - accuracy: 0.3136 - val_loss: 0.5930 - val_accuracy: 0.8491
Epoch 2/10
304/304 [=====] - 319s 1s/step - loss: 0.4330 - accuracy: 0.8983 - val_loss: 0.2760 - val_accuracy: 0.9010
Epoch 3/10
304/304 [=====] - 325s 1s/step - loss: 0.1851 - accuracy: 0.9567 - val_loss: 0.1389 - val_accuracy: 0.9611
Epoch 4/10
304/304 [=====] - 426s 1s/step - loss: 0.1037 - accuracy: 0.9755 - val_loss: 0.0895 - val_accuracy: 0.9888
Epoch 5/10
304/304 [=====] - 399s 1s/step - loss: 0.0639 - accuracy: 0.9863 - val_loss: 0.0563 - val_accuracy: 0.9880
Epoch 6/10
304/304 [=====] - 369s 1s/step - loss: 0.0375 - accuracy: 0.9934 - val_loss: 0.0423 - val_accuracy: 0.9895
Epoch 7/10
304/304 [=====] - 369s 1s/step - loss: 0.0266 - accuracy: 0.9961 - val_loss: 0.0450 - val_accuracy: 0.9864
Epoch 8/10
304/304 [=====] - 370s 1s/step - loss: 0.0192 - accuracy: 0.9967 - val_loss: 0.0307 - val_accuracy: 0.9940
Epoch 9/10
304/304 [=====] - 363s 1s/step - loss: 0.0157 - accuracy: 0.9972 - val_loss: 0.0171 - val_accuracy: 0.9974
Epoch 10/10
304/304 [=====] - 349s 1s/step - loss: 0.0090 - accuracy: 0.9998 - val_loss: 0.0143 - val_accuracy: 0.9978

```

Figura 10: Resultado de la ejecución de la función *summary* en la red neuronal convolucional.

Se han implementado tres nuevas estrategias para poder entrenar las redes neuronales especificadas. Estas estrategias permiten guardar los modelos que se utilizan, facilitando la ejecución con diferentes datasets y modelos de redes neuronales, no teniendo que ejecutar de nuevo el procesamiento de las imágenes y el entrenamiento en cada cambio que se realice en el código.

El inconveniente de tener el proyecto separado en estrategias es que se genera un aumento en el coste de memoria, ya que se necesita guardar los cambios realizados en archivos para poder utilizarlos en la siguiente ejecución.

Las estrategias implementadas son:

- **SaveDatabase:** tiene como objetivo procesar el dataset, transformándolo al formato adecuado: cambiando el tamaño y el color de cada imagen. Una vez se han hecho las modificaciones pertinentes, se guarda el modelo en un archivo denominado pickle, con extensión *.pkl*.

De cada dataset se generarán dos pickels, uno con las imágenes de training y otro con las de testing. Con el dataset con el que estamos trabajando actualmente, en la versión de las imágenes a color, el pickle de testing

tiene un tamaño de 3 GB y el de training de 6,99 GB, 9,99 GB en total.

En la tabla que se muestra a continuación se puede apreciar la comparación de los tamaños de las dos versiones del dataset cuando se guardan en un directorio, tal y como se descargaron de Kaggle [13], respecto al tamaño del archivo pickel.

	Tamaño directorio	Tamaño pickel
Dataset en color	106,7 MB	9,99 GB
Dataset preprocesado	81,6 MB	9.99 GB

Tabla 2: Comparativa del tamaño del dataset cuando se guardan las imágenes en un directorio, o bien cuando se utiliza un pickel.

Como se puede apreciar en la tabla 2, el archivo pickel tiene un tamaño mayor que el propio dataset, pero aún así es beneficioso ya que su lectura es mucho más rápida y nos permite no tener que procesar los datos en cada ejecución. En futuros experimentos se intentará reducir este tamaño con la intención de poder disminuir el coste de memoria.

En la tabla 3, se muestra el coste temporal de la ejecución de esta estrategia. Se puede apreciar como es un tiempo reducido, no llega a los cinco minutos, y es independiente del tipo de procesado. Su diferencia es del orden de segundos, lo que puede variar por cada ejecución.

	Tiempo ejecución
Dataset en color	00:03:01 horas
Dataset preprocesado	00:03:56 horas

Tabla 3: Comparativa del tiempo de ejecución de la estrategia SaveDatabase utilizando dos modelos con diferente procesado.

- **TrainNeuralNetwork:** esta estrategia tiene como objetivo entrenar la red neuronal y guardar el modelo en un archivo con extensión *h5*. Para poder ejecutarla se necesita especificar el tipo de red neuronal que se quiere utilizar. Actualmente hay dos posibilidades: la red neuronal básica y la convolucional. Otro parámetro que también se debe especificar es el pickel con el que se entrenará la red neuronal.

El modelo se guardará en el archivo *h5* con la intención de poder utilizarlo en otras estrategias, como la que se explicará a continuación, sin tener que volver a entrenarla para poder trabajar con este modelo.

El tamaño de este archivo depende del tipo de red neuronal utilizada, en este caso el modelo de la básica tiene un tamaño de 27,1 MB, y la convolucional de 657,2 MB, independientemente del pickle utilizado. Esta información se muestra de una forma más gráfica en la tabla 4:

	Tamaño modelo red neuronal básica	Tamaño modelo red neuronal convolucional
Dataset en color	27,1 MB	657,2 MB
Dataset preprocesado	27,1 MB	657,2 MB

Tabla 4: Tamaño del archivo *h5* por los dos tipos de redes neuronales con los dos datasets que se están utilizando actualmente.

El tiempo de ejecución de esta estrategia varía mucho dependiendo del tipo de red neuronal que se esté entrenando y del pickle seleccionado. La red neuronal convolucional puede llegar a tardar alrededor de una hora más que la básica, como se muestra en la tabla que hay a continuación:

	Tiempo entreno red neuronal básica	Tiempo entreno red neuronal convolucional
Dataset en color	00:05:54 horas	00:55:45 horas
Dataset preprocesado	00:03:29 horas	01:00:16 hora

Tabla 5: Tiempo de ejecución de la estrategia de *TrainingNeuralNetwork* dependiendo del dataset y del tipo de red neuronal.

- **AccuracyNeuralNetwork:** Una vez entrenada la red neuronal, se habrá creado un archivo *h5* donde se habrá guardado el modelo entrenado. Esta estrategia procesa el modelo guardado e intenta hacer las predicciones con la parte del dataset definida para el testing, una vez tiene el resultado lo analiza para poder comprobar el número de aciertos. El resultado se mostrará por el terminal.

En la siguiente tabla se muestra una comparativa del *accuracy* entre las diferentes versiones del dataset: la original y la preprocesada con las dos redes neuronales implementadas.

	<i>Accuracy</i> red neuronal básica	<i>Accuracy</i> red neuronal convulacional
Dataset en color	72.27 %	34.46 %
Dataset preprocesado	100 %	99.96 %

Tabla 6: Comparativa del *accuracy* de los dos tipos de redes neuronales dado dos modelos con diferente procesado.

Como se aprecia en la tabla 6, la red neuronal básica tiene una *accuracy* mas elevada en los dos Dataset, con lo que se han deducido dos posibilidades:

- No se están introduciendo suficientes datos en la red neuronal convulacional para poder hacer un buen entrenamiento, ya que este tipo de redes necesitan muchos datos para poder aprovechar todo su potencial.
- La configuración con los que se han implementado la red neuronal convulacional no son las más adecuadas para este dataset.

En cuanto al tiempo de ejecución de esta estrategia, si que se encuentra diferencia entre el tipo de red neuronal, pero su duración no llega al minuto, como se muestra la tabla que hay a continuación:

	Tiempo accuracy red neuronal básica	Tiempo accuracy red neuronal convulacional
Dataset en color	11 segundos	48 segundos
Dataset preprocesado	15 segundos	43 segundos

Tabla 7: Tiempo de ejecución de la estrategia de *AccuracyNeuralNetwork* dependiendo del dataset y del tipo de red neuronal.

En futuros experimentos se comprobará si las suposiciones que se han mencionado son acertadas o no, con la intención de mejorar la *accuracy* de las dos redes neuronales.

Una vez analizadas cada una de estas estrategias, podemos comparar el coste global, tanto de memoria como de tiempo. En la siguiente tabla se muestra la suma de los costes de memoria que ha habido en las tres estrategias, dependiendo del dataset y del tipo de red neuronal.

	Coste memoria red neuronal básica	Coste memoria red neuronal convolucional
Dataset en color	10,02 GB	10,65 GB
Dataset preprocesado	10,02 GB	10,65 GB

Tabla 8: Comparativa del coste de memoria en las tres estrategias entre los dos tipos de redes neuronales, dado dos modelos con diferente procesado.

El coste de memoria ha sido calculado a partir de la suma de los archivos que se generan en cada estrategia. En concreto, la suma del tamaño del pickle y del archivo que contiene el modelo entrenado de la red neuronal. Como conclusión, se puede observar como el dataset no supone una diferenciación en el coste de memoria, ya que los pickels tienen el mismo tamaño con los dos procesados. Lo que provoca la diferencia entre los dos tipos de red neural, es el archivo *h5* que se genera una vez ha sido entrenada y que tiene un tamaño mayor en la convolucional, como queda reflejado en la tabla 8.

En la siguiente tabla se muestra el coste de tiempo de las tres estrategias, sumando el tiempo de ejecución de cada una de ellas, dependiendo del dataset y de la red neuronal utilizada.

	Coste tiempo red neuronal básica	Coste tiempo red neuronal convolucional
Dataset en color	00:09:06 horas	00:59:34 horas
Dataset preprocesado	00:07:40 horas	01:04:55 hora

Tabla 9: Comparativa del coste de tiempo en las tres estrategias entre los dos tipos de redes neuronales, dado dos modelos con diferente procesado.

En la tabla 9 se puede observar como hay una diferencia abismal entre el coste del tiempo de ejecución de la red neuronal básica a la convolucional, prácticamente la diferencia es de una hora. Con el coste de tiempo sucede lo mismo que con el coste de memoria, es indiferente el dataset utilizado. La razón de esto, es que al final hay el mismo número de datos, únicamente cambia el tipo de procesado, el dataset que originalmente es en color, ha sido transformado para poder trabajar en tonalidades grises, demostrando así que no hay un coste adicional de memoria utilizado para guardar la información del color.

En cuanto a la conclusión de la implementación de las redes neuronales, se ha observado, en la tabla 6, como la red neuronal básica ha dado mejores resultados que la convolucional, y como el procesado del dataset ha significado un mejora inimaginable en los dos tipos de redes. Esta información se tendrá en cuenta cuando se vaya a utilizar un dataset diferente al actual.

Durante la explicación de cada estrategia se han contemplado diferentes me-

joras que se llevarán a cabo en futuras experimentaciones, con la intención de poder obtener el mejor resultado con el mínimo coste posible de las redes neuronales.

6.3. Experimentación 3: Uso de árboles de decisión

Una vez se han obtenido resultados y conclusiones de las redes neuronales, se ha decidido experimentar con otro tipo de estructuras, como son los árboles de decisión.

Como se muestra en los fundamentos, en el apartado 4, se han analizado diferentes tipos de arboles de decisión. Pero se acabó decidiendo por implementar un Boosted Tree.

EXPLICACIÓN DE LA ELECCIÓN DEL BOOSTED TREE

Se ha utilizado la librería xgboost para poder llevar a cabo la implementación, de donde se ha utilizado la clase XGBClassifier. La imagen que hay a continuación muestra como se ha hecho uso de esta librería para entrenar el árbol de decisión:

```
xgboost_model = XGBClassifier()
xgboost_model.fit(x_train, y_train, verbose=True, eval_set=[(x_test, y_test)])
```

Figura 11: Uso de la librería xgboost en la implementación del boosted decision tree.

Se ha entrenado el árbol de decisión con los dos mismos datasets con los que se han entrenado las redes neuronales, con la intención de poder comparar la *accuracy* final entre las dos estructuras.

Para poder llevar a cabo este entrenamiento se han implementado dos nuevas estrategias, con el mismo objetivo que con las redes neuronales: poder separar la ejecución del entrenamiento de la ejecución de la predicción, con tal de no tener que entrenarlo cada vez que se quiera hacer una predicción. Las estrategias creadas son:

- **TrainDecisionTree:** Esta estrategia tiene el objetivo de entrenar el árbol de decisión dado un pickle que contendrá el dataset con el que se quiere trabajar.

Una vez entrenado, se guardará el modelo en un archivo con extensión *.pickle.dat*, el cual nos permitirá recuperar dicho modelo para utilizarlo en otras estrategias.

En la siguiente tabla, referenciada con el número 10, se muestra el coste de memoria que genera el archivo donde se guarda el modelo entrenado del árbol de decisión. Como se puede apreciar, EXPLICAR RESULTADOS, COMPARATIVA ENTRE AMBOS TAMAÑOS.

Tamaño del modelo del árbol de decisión	
Dataset en color	1,1 MB
Dataset preprocesado	2,2 MB

Tabla 10: Comparativa del coste de memoria en la estrategia *TrainDecisionTree*, dado dos modelos con diferente procesado.

Para poder evaluar la eficiencia de este modelo predictivo también se debe tener en cuenta el coste de tiempo en su entrenamiento. Como podemos ver en la tabla 11, el tiempo de ejecución mínimo de esta estrategia es de prácticamente cuatro horas, tres horas más que con la red neuronal, como se comparará más adelante. También podemos apreciar que no hay mucha diferencia entre el tiempo de entrenamiento de ambos datasets analizados, pero debemos recordar que ambos pickels, estructura donde se guardan los datasets, tienen el mismo tamaño. ASEGURARSE DE QUE EL TIEMPO ES SIMILAR

Tiempo ejecución del entrenamiento del árbol de decisión	
Dataset en color	03:35:45 horas
Dataset preprocesado	02:07:53 horas

Tabla 11: Comparativa del coste de tiempo en la estrategia *TrainDecisionTree*, dado dos modelos con diferente procesado.

- **AccuracyDecisionTree:** La segunda estrategia que se ha implementado para poder utilizar el aprendizaje basado en árboles de decisión tiene como objetivo mostrar el *accuracy* del modelo entrenado. Se especificará el modelo que se quiere utilizar, el cual será el archivo con extensión *.pickle.dat* que esta estrategia procesará. Una vez se haya leído el modelo se hará una predicción con todos los datos de testing del dataset seleccionado, el cual deberá ser el mismo con el que se hizo el entrenamiento. El tanto por ciento de aciertos se mostrarán por el terminal, informando así del *accuracy* de modelo.

En la tabla 12 se muestran los resultados de esta estrategia, comparando la *accuracy* de los dos modelos entrenados con los dos datasets con los que

se han hecho todos los experimentos hasta el momento.

<i>Accuracy</i> del árbol de decisión	
Dataset en color	100 %
Dataset preprocesado	100 %

Tabla 12: Comparativa de la *accuracy* del árbol de decisión, dado dos modelos con diferente procesado.

Como se muestra en la tabla 12, se obtiene una *accuracy* máxima con ambos datasets, lo que muestra que con este tipo de aprendizaje se llegan a obtener resultados muy óptimos para las condiciones del problema actual que se intenta resolver.

En cuanto a los costes de esta estrategia, nos centraremos en el coste de tiempo ya que no se genera ningún archivo adicional que pudiese incrementar el coste de memoria, el único coste significativo que hay es el propio coste de ejecución.

Se ha ejecutado esta estrategia dos veces, una por cada dataset con el que estamos trabajando, la duración de cada ejecución se muestra en la tabla 13:

Tiempo ejecución del <i>accuracy</i> del árbol de decisión	
Dataset en color	00:00:23 horas
Dataset preprocesado	00:00:20 horas

Tabla 13: Comparativa del coste de tiempo en la estrategia *AccuracyDecisionTree*, dado dos modelos con diferente procesado.

Se puede apreciar como el tiempo de ejecución de esta estrategia es mínimo con ambos datasets. Se han obtenido muy buenos resultados con este método de aprendizaje: una *accuracy* del 100 % con un tiempo de predicción de unos 20 segundos.

Una vez analizadas las dos estrategias, podemos comparar el coste global de tiempo, ya que el coste de memoria únicamente se ha mencionado el de la primera estrategia, por lo tanto el coste global de memoria será el mismo que el de la estrategia de *TrainDecisionTree*.

En la siguiente tabla se muestra la suma de los costes de tiempo que ha habido en las dos estrategias, dependiendo del dataset utilizado.

	Coste tiempo del árbol de decisión
Dataset en color	1,1 MB
Dataset preprocesado	2,2 MB

Tabla 14: Comparativa del coste de tiempo en el uso del árbol de decisión como modelo de predicción dado dos modelos con diferente procesado.

El coste de tiempo ha sido calculado a partir de la suma de los tiempos de ejecución de ambas estrategias. El alto valor de este coste es debido a la primera estrategia, que es la causante de este incremento, la segunda estrategia tenía un coste del orden de segundos. Esto demuestra que el entrenamiento de un árbol de decisión es mucho más costoso que su predicción, deberemos comprobar si, para el problema que queremos resolver en este proyecto, nos puede afectar este coste tan elevado o si es despreciable.

6.4. Comparativa de las experimentaciones 2 y 3

Una vez explicada la experimentación de los árboles de decisión, se puede hacer la comparativa entre los resultados de estos y los de las redes neuronales.

Para poder realizar una comparativa adecuada se deberán utilizar los parámetros que se han ido estudiando durante las experimentaciones: la *accuracy* y los costes, tanto de tiempo como de memoria.

En la tabla 15 se puede apreciar la comparación de la *accuracy* entre las tres estructuras implementadas: la red neuronal básica, la red neuronal convolucional y el árbol de decisión. Se han utilizado los dos datasets con los que se ha estado trabajando en ambos experimentos.

	<i>Accuracy</i> red neuronal básica	<i>Accuracy</i> red neuronal convolucional	<i>Accuracy</i> árbol de decisión
Dataset en color	72,27 %	34,46 %	100 %
Dataset preprocesado	100 %	99,96 %	100 %

Tabla 15: Comparativa de la *accuracy* entre las estructuras implementadas dado dos modelos con diferente procesado.

Podemos apreciar cómo al árbol de decisión es el que tiene una mejor *accuracy*, ya que su valor es del 100 % con ambos datasets. Otra conclusión que podemos obtener es que todas las estructuras tienen un precisión muy elevada cuando se utiliza el dataset específicamente procesado para favorecer el entrenamiento.

En cuanto al coste de tiempo, compararemos la duración de la ejecución del entrenamiento y de la predicción por separado. En la tabla 16 se pueden ver la duración del entrenamiento en la red neuronal básica, la convolucional y el árbol de decisión, estos valores se han obtenido de las tablas: 5 y 11.

	Duración red neuronal básica	Duración red neuronal convolucional	Duración árbol de decisión
Dataset en color	00:05:54 horas	00:55:45 horas	03:35:45 horas
Dataset preprocesado	00:03:29 horas	01:00:16 hora	02:07:53 horas

Tabla 16: Comparativa de la duración del entrenamiento entre las estructuras implementadas dado dos modelos con diferente procesado.

En la tabla 17, se muestra una comparativa de la duración de las predicciones entre la red neuronal básica, la convolucional y el árbol de decisión. Estos valores han sido obtenidos de las tablas: 7 y 13.

	Duración red neuronal básica	Duración red neuronal convolucional	Duración árbol de decisión
Dataset en color	00:00:11 horas	00:00:48 horas	00:00:23 horas
Dataset preprocesado	00:00:15 horas	00:00:43 horas	00:00:20 horas

Tabla 17: Comparativa de la duración de la predicción entre las estructuras implementadas dado dos modelos con diferente procesado.

Comparando los valores mostrados en las dos tablas, podemos observar como el árbol de decisión es el que más se deriva, ya que su duración de entrenamiento llega prácticamente a triplicar el de las redes neuronales. En cuanto a la duración de las predicciones, todos tienen una duración del orden de segundos, siendo la red neuronal convolucional la que mas se acerca al minuto.

Podemos deducir entonces, que el árbol de decisión es el que nos proporciona una mejor *accuracy*, pero nos genera un mayor coste. En futuras experimentaciones se intentaran mejorar los valores de la red neuronal convolucional, ya que hay diversos parámetros que pueden incrementar su *accuracy* y que podrían obtenerse con un coste menor al árbol de decisión.

6.5. Experimentación 4: Incremento dataset

El dataset que se ha utilizado durante las anteriores experimentaciones disponía de 27.750 muestras, 55.500 en total con los dos datasets con diferente procesado. Con este número de muestras se puede obtener muy buenos resultados, pero aún serían mejores si se utilizaran dos datasets diferentes, ya que el ángulo, el color y la forma de la imagen abarcarían mas posibilidades. Conseguiríamos así una accuracy mejor para cuando se hagan predicciones con imágenes que no pertenezcan al testing del mismo dataset.

Se han evaluado varios datasets, buscando uno que contenga los mismo gestos que el actual pero con un enfoque diferente en los datos. Los enlaces que se muestran a continuación contienen los datasets que se han estado evaluando. Como se puede apreciar, todos pertenecen a la comunidad de datos Kaggle [12].

- <https://www.kaggle.com/kareemalaa74/dataset-asl-test-and-train>
- <https://www.kaggle.com/grassknotted/asl-alphabet>
- <https://www.kaggle.com/lucasvieirademiranda/aslalphabet>
- <https://www.kaggle.com/ammarnassanalhajali/american-sign-language-letters>

Finalmente se ha optado por escoger el primer enlace mostrado:

<https://www.kaggle.com/kareemalaa74/dataset-asl-test-and-train>

Los motivos de esta elección han sido:

- Dispone de los gestos que hacen referencia a los números. La mayoría de los otros datasets únicamente contenían letras, con lo que se debería de haber utilizado dos dataset, uno para las letras y otro para los números.
- Las imágenes de este dataset han sido tomadas desde diferentes ángulos que las del actual dataset.
- Las imágenes no se han tomado con un fondo de un mismo color, como pasa con el dataset actual, sino que se han realizado con diferentes fondos.

El nuevo dataset tiene como nombre: *Dataset ASL test and train* y se ha obtenido de la comunidad de datos Kaggle [12]. Dispone de los mismos gestos que el dataset *Sign Language Gesture Images Dataset* [13], pero este, adicionalmente, contiene el número 10. Cada gesto dispone de 300 imágenes, 301 con la imagen de test, con unas dimensiones de 1500×1500 píxeles.

A continuación, se muestra una comparativa del dataset que se ha estado utilizando, con el nuevo que se va a añadir al actual. Las imágenes 12 y 13 hacen referencia al dataset actual. Son las mismas imágenes que se mostraron y explicaron en la experimentación de la búsqueda del dataset, en la sección 6.1.



Figura 12: Imágenes obtenidas del dataset *Sign Language Gesture Images Dataset* [13] en color.



Figura 13: Imágenes obtenidas del dataset *Sign Language Gesture Images Dataset* [13] preprocesadas con la intención de mejorar el entrenamiento de las redes neuronales.

La imagen 14 muestra diez gestos del dataset *Dataset ASL test and train* [14]. Hace referencia a los mismos gestos que las imágenes 12 y 13, se han situado en el mismo orden para facilitar su comparación. Se puede apreciar como cumple las características explicadas anteriormente del porque se ha seleccionando este dataset.



Figura 14: Imágenes obtenidas del dataset *ASL test and train* [14].

Se ha realizado dos experimentaciones con este dataset: usándolo individualmente, y añadiendo las muestras del dataset *Sign Language Gesture Images* [13]. La intención de esto es poder saber la accuracy y los costes de ambos datasets por separado, y poder compararlo con los valores que nos dan estas características cuando se utilizan a la par.

Para poder realizar la segunda experimentación, se han tenido que implementar unos cambios en el proyecto, ya que hasta ahora no se podía seleccionar más de un dataset. Se ha pensado la mejor solución para este problema, y se han acabado encontrado dos posibles implementaciones:

- Permitir seleccionar más de un dataset en la creación de un pickel, en la estrategia *SaveDatabase*, con lo que en las otras estrategias no sería necesario realizar ningún cambio, ya que seguiría utilizando únicamente un pickel.
- En un pickel únicamente se puede almacenar un dataset, no teniendo que cambiar esta estrategia. El cambio se debería realizar en las otras estrategias, ya que estas deberían permitir seleccionar tantos pickels como el usuario desee.

Finalmente se optó por implementar la segunda opción. A pesar de tener que realizar más cambios en el proyecto actual, ya que prácticamente se debía cambiar la estructura de todas las estrategias, se pensó que, en cuanto a coste de memoria, sería mejor tener un único pickel por dataset, que no tener diferentes pickels con variaciones del mismo dataset. Como se ha comentado anteriormente en la tabla 2, un pickel tiene un tamaño del orden de gigabytes.

Una vez realizados los cambios convenientes para poder utilizar más de un pickel en una estrategia, se ejecutó el proyecto para poder averiguar la *accuracy*

de este nuevo dataset.

A continuación, se compararan los resultados y el coste de las ejecuciones de las tres siguientes combinaciones de datasets:

- Con el dataset *Sign Language Gesture Images* [13], que hace referencia a la imagen 12.
- Con el nuevo dataset, *ASL test and train* [14], mostrado en la imagen 14.
- Combinando ambos datasets.

Los tres tablas, que hay a continuación, referenciadas con los números: 18, 19 y 19, muestran la comparativa de las tres combinaciones de datasets que se acaban de explicar. La tabla 18 contiene la información de la accuracy de la red neuronal básica, la convolucional y el árbol de decisión, estas tres estructuras serán las que se van a ir comparando con las combinaciones de datasets.

	Dataset Sign Language Gesture Images	Dataset ASL test and train	Combinación de ambos datasets
<i>Accuracy</i> red neuronal básica	72,27 %	33,85 %	2,88 %
<i>Accuracy</i> red neuronal convolucional	34,46 %	98,95 %	99,30 %
<i>Accuracy</i> árbol de decisión	100 %	100 %	—

Tabla 18: Comparativa de la *accuracy* de las tres estructuras implementadas dado dos datasets y una combinación de ambos.

La tabla 19 compara el tiempo de ejecución de los entrenamientos entre las diferentes estructuras. El formato de la tabla es el mismo que el anterior, facilitando así la lectura de la información de ambas tablas y pudiendo compararlas a la vez.

	Dataset Sign Language Gesture Images	Dataset ASL test and train	Combinación de ambos datasets
Tiempo entreno red neuronal básica	00:05:54 horas	00:01:01 horas	00:12:09 horas
Tiempo entreno red neuronal convolucional	00:55:45 horas	00:27:51 horas	01:35:02 horas
Tiempo entreno árbol de decisión	03:35:45 horas	03:38:52 horas	–

Tabla 19: Comparativa del tiempo de entrenamiento de las tres estructuras implementadas dado dos datasets y una combinación de ambos.

Por último, la tabla 20, muestra la comparativa del tiempo de ejecución de las predicciones.

	Dataset Sign Language Gesture Images	Dataset ASL test and train	Combinación de ambos datasets
Tiempo predicción red neuronal básica	00:00:11 horas	00:00:04 horas	00:00:41 horas
Tiempo predicción red neuronal convolucional	00:00:48 horas	00:00:24 horas	00:01:43 horas
Tiempo predicción árbol de decisión	00:00:23 horas	00:00:10 horas	–

Tabla 20: Comparativa del tiempo de predicción de las tres estructuras implementadas dado dos datasets y una combinación de ambos.

Como se ha podido contemplar, en la tabla no se muestran los resultados de las combinaciones de ambos datasets utilizando el árbol de decisión como modelo de predicción. El motivo es que no se pudo acabar de realizar la ejecución de su entrenamiento, después de 24h únicamente se había completado el 30 %. Por lo tanto, si no hubiese ningún problema durante la ejecución, como podría ser un coste de memoria superior al que el dispositivo podría soportar, el tiempo total hubiese sido de 80 horas, 3,33 días. Se paró la ejecución en el 30 % ya que se dedujo que este tiempo de entrenamiento era inviable bajo las características en las que se está realizando este proyecto.

Hasta el momento, el árbol de decisión era una solución que nos estaba aportando un *accuracy* muy elevada, a pesar de tener el tiempo de entrenamiento más alto. Pero viendo que no es capaz de proceder con el entrenamiento de ambos datasets, cuando tenemos otras estructuras implementadas que también dan muy buenos resultados con un tiempo de entrenamiento asumible, se ha decidido descartar este modelo de predicción.

En cuanto a la red neuronal básica, podemos apreciar como esta dispone del tiempo de ejecución más reducido en, prácticamente, todas las ejecuciones. Aún así, en la tabla 18 se muestra como disminuye la *accuracy* hasta el 2,88 % cuando se utilizan ambos datasets. Se realizaron varias ejecuciones para poder asegurar que este valor no variaba, pero la *accuracy* de todas estas ejecuciones no varió, la máxima desviación fue de una unidad.

Con estos resultados podemos verificar que la red neuronal básica no es un modelo fiable en este proyecto, por lo tanto, no se puede utilizar como solución a este problema.

Por último, la red neuronal convolucional dio una *accuracy* del 99,30 %, un valor muy elevado y totalmente valido para la solución. Su tiempo de ejecución en el entrenamiento es mas elevado que el de la red neuronal básica, pero es un tiempo asequible. Se concluye que este modelo es el óptimo de los implementado hasta ahora.

6.6. Experimentación 5: Mejorar rendimiento de la CNN

Como se ha demostrado en la sección 6.5, la estructura implementada con la que se han obtenido los mejores resultados, tanto de rendimiento como de costes, ha sido la red neuronal convolucional.

Todos los experimentos que se hagan a partir de ahora se realizarán con la unión de los datasets y con la red neuronal convolucional.

Aún haber obtenido muy buenos resultados con esta red neuronal, en esta sección se intentará acotar sus parámetros para poder obtener el mayor rendimiento con las mínimas pérdidas. Se ha utilizado la librería Scikit-learn [15] para poder implementar la estrategia que realizará la comparativa de parámetros. La nueva estrategia implementada se denomina: *Hyperparameter Optimization Strategy*

Se ha seguido el artículo de *Grid search hyperparameters deep learning models python keras* [16] para la implementación de este experimento. Se ha utilizado como guía y modelo, pero se ha modificado y adaptado a las necesidades del actual problema.

Se comenzó haciendo las pruebas con todos los datos de ambos datasets, se estaba haciendo una estimación del valor que deberían asignarse al tamaño del *Batch* y el número de *Epochs*. Ejecutando el training de la red neuronal convolucional con estos parámetros variables, se comprobó que no era posible realizar la ejecución con todos los datos. Después de cuatro días ejecutándose, cuando se calculaba que aún faltaba el doble de tiempo, como mínimo, para finalizar su ejecución, el dispositivo se quedó sin espacio de memoria y tuvo que interrumpirla.

Como solución, se implementó una modificación en el proyecto. En lugar de leer todos los datos de los *Pickels*, archivos donde se almacenan las imágenes procesadas, únicamente se obtienen 20 imágenes de cada señal. Consiguiendo así disminuir el tiempo de entrenamiento de la red neuronal. Los cambios en el código han sido notorios, ya que se ha tenido que implementar una nueva funcionalidad donde se pudiesen obtener el mismo número de imágenes por cada señal, la reducción del dataset debía continuar siendo óptima.

La estrategia *Hyperparameter Optimization Strategy* consta de siete atributos a optimizar, el cual se especifica en los parámetros de la ejecución de esta. A continuación se explicará cada atributo y los resultados obtenidos de este.

6.6.1. Tamaño del *Batch* y número de *Epochs*

Para la realización de este entrenamiento, se ha utilizado el mismo modelo que el implementado en la red neuronal convolucional, ya que es el que se busca mejorar. En esta ejecución se optimizan los siguientes atributos:

- El tamaño del *Batch*, el cual define la cantidad de datos con los que trabajar.
- El número de *Epochs* que define el número de veces que el algoritmo de aprendizaje se ejecutará por cada conjunto de datos de entrenamiento.

El parámetro que se debe introducir para poder ejecutar la estrategia optimizando estos atributos es ”*batch_and_epoch*”.

Los parámetros utilizados para el clasificador se muestran en la imagen 15.

```
@staticmethod
def __get_parameters_for_batch_epochs(n_classes, image_size):
    batch_size = [10, 20, 40, 60, 80, 100]
    epochs = [10, 50, 100]

    param_grid = dict(batch_size=batch_size, epochs=epochs, num_classes=[n_classes], image_size=[image_size])
    classifier = KerasClassifier(build_fn=create_model_batch_epochs, verbose=2)

    return param_grid, classifier
```

Figura 15: Parámetros evaluados en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*batch_and_epoch*”

El tamaño del *Batch* se ha evaluado con los valores: 10, 20, 40, 60, 80 y 100. En cambio, el parámetro del número de *Epochs* es menos variado, ya que oscila entre los valores: 10, 50 y 100.

La imagen 16 muestra los resultados de esta ejecución. La mejor combinación de parámetros es de un tamaño del *Batch* y un número de *Epochs* igual a 10.

```

Best: 1.000000 using {'batch_size': 10, 'epochs': 10}
1.000000 (0.000000) with: {'batch_size': 10, 'epochs': 10}
1.000000 (0.000000) with: {'batch_size': 10, 'epochs': 50}
1.000000 (0.000000) with: {'batch_size': 10, 'epochs': 100}
1.000000 (0.000000) with: {'batch_size': 20, 'epochs': 10}
1.000000 (0.000000) with: {'batch_size': 20, 'epochs': 50}
1.000000 (0.000000) with: {'batch_size': 20, 'epochs': 100}
1.000000 (0.000000) with: {'batch_size': 40, 'epochs': 10}
1.000000 (0.000000) with: {'batch_size': 40, 'epochs': 50}
1.000000 (0.000000) with: {'batch_size': 40, 'epochs': 100}
1.000000 (0.000000) with: {'batch_size': 60, 'epochs': 10}
1.000000 (0.000000) with: {'batch_size': 60, 'epochs': 50}
1.000000 (0.000000) with: {'batch_size': 60, 'epochs': 100}
1.000000 (0.000000) with: {'batch_size': 80, 'epochs': 10}
1.000000 (0.000000) with: {'batch_size': 80, 'epochs': 50}
1.000000 (0.000000) with: {'batch_size': 80, 'epochs': 100}
1.000000 (0.000000) with: {'batch_size': 100, 'epochs': 10}
1.000000 (0.000000) with: {'batch_size': 100, 'epochs': 50}
1.000000 (0.000000) with: {'batch_size': 100, 'epochs': 100}
[INFO]: Strategy executed successfully
[INFO]: Execution finished
[INFO]: The duration of the execution has been 02:55:57 [hh:mm:ss]

```

Figura 16: Resultado de la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*batch_and_epoch*”

Podemos apreciar como todos las combinaciones tienen el mismo resultado, un *accuracy* del 100 % sin ninguna pérdida. Se utilizará el resultado que se muestra como el mejor, con un valor de 10 en ambos atributos. Aún así, teniendo como objetivo únicamente la *accuracy*, se podría utilizar cualquiera de las combinaciones probadas y se obtendría el mismo resultado.

Esta ejecución ha tenido un coste de tiempo de 2 horas, 55 minutos y 57 segundos. Es un tiempo muy reducido comparado con el de la primera ejecución, cuando se utilizó todo el dataset. Como se ha comentado anteriormente, este se estuvo ejecutando durante cuatro días cuando se paró su ejecución sin haber llegado a la mitad de esta. Con las modificaciones realizadas en el proyecto y estimando el tiempo total de ejecución con el dataset completo, se ha conseguido reducir un 98,44 % el coste del tiempo de ejecución.

Los valores obtenidos en este experimento se utilizarán en las optimizaciones de los siguientes atributos.

6.6.2. Algoritmos de optimización

Una vez obtenidos los mejores valores para los atributos del tamaño del *Batch* y el número de *Epochs*, se ha querido evaluar cual sería el algoritmo de entrenamientos más óptimo.

Para poder implementar esta funcionalidad se ha utilizado la estrategia, comentada anteriormente, *Hyperparameter Optimization Strategy*. Para poder ejecutar esta optimización se debe introducir el parámetro ”*optimization_algorithms*”.

En la imagen número 17, se muestran los parámetros introducidos en este clasificador. Los algoritmos con los que se han hecho las comparaciones son: *SGD*, *RMSprop*, *Adagrad*, *Adadelta*, *Adam*, *Adamax* y *Nadam*. Las principales características de estos algoritmos están explicados en los fundamentos, en el apartado 4.

```
@staticmethod
def __get_parameters_for_optimization_algorithm(n_classes, image_size):
    optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']

    param_grid = dict(optimizer=optimizer, num_classes=[n_classes], image_size=[image_size])
    classifier = KerasClassifier(build_fn=create_model_optimizer_algorithm, epochs=EPOCHS, batch_size=BATCH_SIZE,
                                 verbose=2)

    return param_grid, classifier
```

Figura 17: Parámetros evaluados en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*optimization_algorithms*”

Como se muestra en la imagen anterior, los valores del tamaño del *Batch* y el número de *Epochs* son constantes, ya que adquieren el valor del mejor resultado obtenido en la ejecución del apartado 6.6.1. Estos resultados se guardan en un archivo con extensión *.py*, donde actualmente se encuentran los valores de *EPOCHS* y de *BATCH_SIZE*.

En la siguiente imagen, se muestran los resultados de la ejecución de esta estrategia con el clasificador y los parámetros mostrados en la figura 17.

```
In [15]: python3 Src/main.py --showOptimizedHyperparameter optimization_algorithms asl_alphabet_gray_150x150px sign_gesture_gray
38/38 - 7s - loss: 2.1799e-04 - accuracy: 1.0000
Epoch 7/10
38/38 - 7s - loss: 1.8106e-04 - accuracy: 1.0000
Epoch 8/10
38/38 - 7s - loss: 1.5470e-04 - accuracy: 1.0000
Epoch 9/10
38/38 - 7s - loss: 1.3471e-04 - accuracy: 1.0000
Epoch 10/10
38/38 - 7s - loss: 1.1921e-04 - accuracy: 1.0000
Best: 1.000000 using {'optimizer': 'SGD'}
1.000000 (0.000000) with: {'optimizer': 'SGD'}
1.000000 (0.000000) with: {'optimizer': 'RMSprop'}
1.000000 (0.000000) with: {'optimizer': 'Adagrad'}
1.000000 (0.000000) with: {'optimizer': 'Adadelta'}
1.000000 (0.000000) with: {'optimizer': 'Adam'}
1.000000 (0.000000) with: {'optimizer': 'Adamax'}
1.000000 (0.000000) with: {'optimizer': 'Nadam'}
[INFO]: Strategy executed successfully
[INFO]: Execution finished
[INFO]: The duration of the execution has been 00:33:55 [hh:mm:ss]
```

Figura 18: Resultados obtenidos en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*optimization_algorithms*”

El algoritmo que ha obtenido el mejor resultado es el *SGD*, *Stochastic gradient descent*, con un *accuracy* del 100 %. Aún así, si observamos los resultados obtenidos con los otros algoritmos veremos que su resultado no varía, con obtenemos la máxima *accuracy*.

La duración de esta ejecución ha sido de 33 minutos y 55 segundos, dos horas y media menos que la anterior ejecución. El motivo de la reducción de este coste de tiempos es que únicamente se debían probar siete combinaciones, una por cada algoritmo.

En las siguientes ejecuciones utilizaremos el algoritmo que se muestra como el mejor, pero en caso de querer repetir este experimento sería interesante poder variar el algoritmo para comprobar otros parámetros de su ejecución, como el coste de tiempo, por ejemplo.

6.6.3. Índice de aprendizaje e impulso

En el apartado anterior hemos deducido que el algoritmo de entrenamiento que se utilizará será el *Stochastic gradient descent*, el cual tiene dos parámetros a configurar:

- El índice de aprendizaje (*learn rate*): define la velocidad a la que el modelo se adapta al problema. Cuanto menor sea su valor, mayor deberá ser el tamaño del *Batch* de entrenamiento. Su valor oscila entre los rangos de 0.0 y 1.0

- El impulso (*momentum*): permite acelerar al gradiente en la dirección correcta. En lugar de utilizar solo el gradiente del paso actual, el *momentum* acumula el gradiente de los pasos anteriores para determinar la dirección a seguir.

El parámetro que se debe introducir para poder ejecutar la estrategia optimizando estos atributos es ”*learn_rate_and_momentum*”.

En la imagen que hay a continuación se muestran los parámetros introducidos en el clasificador. El índice de aprendizaje es evaluado a partir de los valores: 0.001, 0.01, 0.1, 0.2 y 0.3. En cambio, el *momentum* puede adquirir uno de los siguientes valores: 0.0, 0.2, 0.4, 0.6, 0.8 o 0.9.

```
@staticmethod
def __get_parameters_for_learn_rate_and_momentum(n_classes, image_size):
    learn_rate = [0.001, 0.01, 0.1, 0.2, 0.3]
    momentum = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]

    param_grid = dict(learn_rate=learn_rate, momentum=momentum, num_classes=[n_classes], image_size=[image_size])
    classifier = KerasClassifier(build_fn=create_model_learn_rate_and_momentum, epochs=EPOCHS,
                                 batch_size=BATCH_SIZE, verbose=2)

    return param_grid, classifier
```

Figura 19: Parámetros evaluados en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*learn_rate_and_momentum*”

En la figura 20 se muestra el resultado de esta ejecución, donde podemos apreciar como los mejores parámetros son la combinación de un índice de aprendizaje del 0.001 y de un *momentum* de 0.

```
[INFO]: Best: 1.000000 using {'image_size': (150, 150), 'learn_rate': 0.001, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.001, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.001, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.001, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.001, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.001, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.001, 'momentum': 0.9, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.9, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.9, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.01, 'momentum': 0.9, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.9, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.02, 'momentum': 0.9, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.9, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.0, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.2, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.4, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.6, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.8, 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'learn_rate': 0.03, 'momentum': 0.9, 'num_classes': 39}
[INFO]: Strategy executed successfully
[INFO]: Execution finished
[INFO]: The duration of the execution has been 01:29:37 [hh:mm:ss]
```

Figura 20: Resultados de la ejecución de la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*learn_rate_and_momentum*”

Dado los resultados obtenidos, utilizaremos el algoritmo *Stochastic gradient descent* sin *momentum*, ya que es un coste adicional innecesario que no nos aporta un mayor rendimiento en la red neuronal convolucional.

Con todas las otras combinaciones obtenemos el mismo *accuracy*, esto ha estado sucediendo en todas las ejecuciones que se han realizado en la optimización de los hiper-parámetros. Se debe remarcar que, tal y como se muestra en la tabla 18, se había conseguido un rendimiento del 99,30 % en la red neuronal convolucional que se está intentando mejorar con ambos datasets. Su valor inicial es prácticamente perfecto, es de esperar que con una *accuracy* tan elevada se obtengan tales resultados con todas las combinaciones.

Esta ejecución ha tenido un coste temporal de 1 hora, 29 minutos y 37 segundos.

6.6.4. Inicialización del peso de la red

Cada nodo de la red neuronal tiene asignado una ponderación que se utiliza para calcular la suma de los *inputs*. El algoritmo de optimización *Stochastic gradient descent*, el cual se está utilizando, va incrementando los pesos con la intención de minimizar la función de pérdida.

Este algoritmo necesita tener asignados unos valores al inicio de la optimización, y es el inicializador el que definen la forma de establecer los pesos iniciales según la capa de la red neuronal. [17]

El parámetro que se debe introducir para poder ejecutar la estrategia optimizando este atributo es "*network_weight_initialization*". Con este podremos comparar el rendimiento de las distintas heurísticas y acabar asignado la que nos aporte una mayor precisión.

Como se muestra en la figura 21, los inicializadores utilizados son: *uniform*, *lecun uniform*, *normal*, *zero*, *glorot normal*, *glorot uniform*, *he normal* y *he uniform*. Todos ellos están detallados y explicados en los fundamentos, en el apartado 4.

Estos son los principales, pero en el enlace que se encuentra a continuación, se muestran todos los inicializadores que la dispone la librería Keras [18].

<https://keras.io/api/layers/initializers/>

```
@staticmethod
def __get_parameters_network_weight_init(n_classes, image_size):
    init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal', 'glorot_uniform', 'he_normal',
                'he_uniform']

    param_grid = dict(init_mode=init_mode, num_classes=[n_classes], image_size=[image_size])
    classifier = KerasClassifier(build_fn=create_model_network_weight_init, epochs=EPOCHS, batch_size=BATCH_SIZE,
                                 verbose=2)

    return param_grid, classifier
```

Figura 21: Parámetros evaluados en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro "*network_weight_initialization*"

En la imagen 22 se muestran los resultados de esta ejecución, donde podemos apreciar que el inicializador *uniform* es el clasificado como el más óptimo. Aún así, podemos ver, como nos ha estados sucediendo en todas las optimizaciones hasta ahora, que los otros inicializadores nos aportan la misma eficiencia.

```
In [16]: !python3 Src/main.py --showOptimizedHyperparameter network_weight_initialization asl_alphabet_gray_150x150px sign_gestu
Epoch 7/10
38/38 - 8s - loss: 0.0026 - accuracy: 1.0000
Epoch 8/10
38/38 - 8s - loss: 0.0022 - accuracy: 1.0000
Epoch 9/10
38/38 - 8s - loss: 0.0019 - accuracy: 1.0000
Epoch 10/10
38/38 - 8s - loss: 0.0016 - accuracy: 1.0000
[INFO]: Best: 1.000000 using {'image_size': (150, 150), 'init_mode': 'uniform', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'uniform', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'lecun_uniform', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'normal', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'zero', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'glorot_normal', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'glorot_uniform', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'he_normal', 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'image_size': (150, 150), 'init_mode': 'he_uniform', 'num_classes': 39}
[INFO]: Strategy executed successfully
[INFO]: Execution finished
[INFO]: The duration of the execution has been 00:22:42 [hh:mm:ss]
```

Figura 22: Resultados de la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*network_weight_initialization*”

Al tener únicamente ocho combinaciones a probar, la duración de esta ejecución ha sido menor a todas las realizadas hasta ahora, con un tiempo de 22 minutos y 42 segundos.

6.6.5. Función de activación neuronal

Otro parámetro a optimizar en la red neuronal convolucional es la función de activación. Permite utilizar la información importante, descartando los datos irrelevantes.

La función de activación comprobará si una neurona debe activarse o no utilizando operaciones matemáticas. Transformará la suma de ponderaciones del nodo en un valor de salida que se enviará a la siguiente capa, este valor servirá para identificar si la neurona está activa [19].

Las funciones de activación probadas en esta ejecución, tal y como se muestra en la imagen 23, son: *relu*, *softmax*, *softplus*, *softsign*, *tanh*, *sigmoid*, *hard_sigmoid* y *linear*. Todas estas funciones están explicadas en los fundamentos, en el apartado 4.

Se han utilizado las funciones más comunes, pero en el siguiente enlace se muestran todas las funciones de activación disponibles en la librería *Keras* [18].

<https://keras.io/api/layers/activations/>

El parámetro que se debe introducir para poder ejecutar la estrategia optimizando este atributo es ”*neuron_activation_function*”.

```

@staticmethod
def __get_parameters_neuron_activation_function(n_classes, image_size):
    activation = ['relu', 'softmax', 'softplus', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']

    param_grid = dict(activation=activation, num_classes=[n_classes], image_size=[image_size])
    classifier = KerasClassifier(build_fn=create_model_neuron_activation_function, epochs=EPOCHS,
                                 batch_size=BATCH_SIZE, verbose=2)

    return param_grid, classifier

```

Figura 23: Parámetros evaluados en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”neuron_activation_function”

En la imagen 25 se muestran los resultados de esta ejecución. Podemos apreciar como se obtienen diferentes valores dependiendo de la función de activación, tres de los cuales tiene una *accuracy* máxima: *relu*, *softplus* y *linear*.

```

In [18]: a1n.py --showOptimizedHyperparameter neuron_activation_function asl_alphabet_gray_150x150px sign_gesture_gray_150x150px
Epoch 7/10
38/38 - 7s - loss: 0.1879 - accuracy: 0.6027
Epoch 8/10
38/38 - 7s - loss: 0.1398 - accuracy: 0.9120
Epoch 9/10
38/38 - 7s - loss: 0.1044 - accuracy: 0.9840
Epoch 10/10
38/38 - 7s - loss: 0.0792 - accuracy: 0.9973
[INFO]: Best: 1.000000 using {'activation': 'relu', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'activation': 'relu', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 0.400000 (0.000000) with: {'activation': 'softplus', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'activation': 'softplus', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 0.877333 (0.167852) with: {'activation': 'softplus', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 1.333333 (0.471405) with: {'activation': 'tanh', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 0.000000 (0.000000) with: {'activation': 'sigmoid', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 0.000000 (0.000000) with: {'activation': 'hard_sigmoid', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: 1.000000 (0.000000) with: {'activation': 'linear', 'image_size': (150, 150), 'num_classes': 39}
[INFO]: Strategy executed successfully
[INFO]: Execution finished
[INFO]: The duration of the execution has been 00:30:51 [hh:mm:ss]

```

Figura 24: Resultado de la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”neuron_activation_function”

Se utilizará la función de activación *ReLU*, la cual es la más utilizada en las redes neuronales convolucionales, ya que acelera su entrenamiento al tener un cálculo tan simple. Cualquier elemento negativo se iguala a cero, no utiliza operaciones costosas tales como exponentiales, multiplicaciones o divisiones.

La ejecución de la optimización de este parámetro ha tenido un coste temporal de 30 minutos y 51 segundos.

6.6.6. Regulación del *Dropout*

El *Dropout* [20] es una técnica que ayuda a regular el *overfitting* en todo tipo de redes neuronales con un coste computacional leve. Este método descarta o ignora, aleatoriamente, una cierta cantidad de neuronas y es muy común

en redes neuronales convolucionales bidimensionales con una gran cantidad de nodos en la red. Esta técnica se puede añadir después de una o varias capas.

En la configuración de esta técnica, utilizando la librería *Keras* [18], se debe introducir un atributo denominado ”*dropout rate*”. Este parámetro tiene un rango de valores decimales del cero al uno. Por ejemplo, si se configura con un *dropout rate* del 0,2, el 20 % de los nodos serán descartados aleatoriamente.

Otro atributo a configurar es el ”*kernel constraint*”, el cual restringe los pesos de las capas ocultas garantizando que la norma máxima de los pesos no excede el valor definido.

Estos dos atributos son los que se van a intentar optimizar en la ejecución de esta estrategia introduciendo el parámetro ”*dropout_regularization*”.

En la imagen 25 se muestra el clasificador de este modelo, con los posibles rangos de valores para el *dropout rate* y el *kernel constraint*.

```
@staticmethod
def __get_parameters_dropout_regularization(n_classes, image_size):
    weight_constraint = [1, 2, 3, 4, 5]
    dropout_rate = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

    param_grid = dict(dropout_rate=dropout_rate, weight_constraint=weight_constraint, num_classes=[n_classes],
                      image_size=[image_size])
    classifier = KerasClassifier(build_fn=create_model_dropout_regularization, epochs=EPOCHS, batch_size=BATCH_SIZE,
                                 verbose=2)

    return param_grid, classifier
```

Figura 25: Parámetros evaluados en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*dropout_regularization*”

En la imagen 26 se muestran los resultados más significativos de esta ejecución, ya que el *output* ocupaba demasiado como para poder incluirlo en una única imagen.

```

[INFO]: Best: 1.000000 using {'dropout_rate': 0.0, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 3}
[INFO]: 0.805333 (0.242153) with: {'dropout_rate': 0.0, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 1}
[INFO]: 0.640000 (0.376652) with: {'dropout_rate': 0.0, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 2}
[INFO]: 1.000000 (0.000000) with: {'dropout_rate': 0.0, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 3}
[INFO]: 0.664000 (0.469530) with: {'dropout_rate': 0.0, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 4}
[INFO]: 0.978667 (0.016438) with: {'dropout_rate': 0.0, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 5}
[INFO]: 1.000000 (0.000000) with: {'dropout_rate': 0.1, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 1}
[INFO]: 0.677333 (0.456320) with: {'dropout_rate': 0.1, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 2}
[INFO]: 0.976000 (0.033941) with: {'dropout_rate': 0.1, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 3}
[INFO]: 0.920000 (0.107530) with: {'dropout_rate': 0.1, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 4}
[INFO]: 0.997333 (0.003771) with: {'dropout_rate': 0.1, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 5}
[INFO]: 0.666667 (0.471405) with: {'dropout_rate': 0.2, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 1}
[INFO]: 1.000000 (0.000000) with: {'dropout_rate': 0.2, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 2}
[INFO]: 1.000000 (0.000000) with: {'dropout_rate': 0.2, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 3}
[INFO]: 0.336000 (0.469530) with: {'dropout_rate': 0.2, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 4}
[INFO]: 1.000000 (0.000000) with: {'dropout_rate': 0.2, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 5}
[INFO]: 0.773333 (0.314915) with: {'dropout_rate': 0.3, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 1}
[INFO]: 0.624000 (0.439102) with: {'dropout_rate': 0.3, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 2}
[INFO]: 0.706667 (0.235272) with: {'dropout_rate': 0.3, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 3}
[INFO]: 1.000000 (0.000000) with: {'dropout_rate': 0.3, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 4}
[INFO]: 0.792000 (0.294156) with: {'dropout_rate': 0.3, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 5}
[INFO]: 0.464000 (0.288666) with: {'dropout_rate': 0.4, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 1}
[INFO]: 0.450667 (0.414167) with: {'dropout_rate': 0.4, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 2}
[INFO]: 0.309333 (0.398522) with: {'dropout_rate': 0.4, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 3}
[INFO]: 0.746667 (0.335894) with: {'dropout_rate': 0.4, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 4}
[INFO]: 0.664000 (0.469530) with: {'dropout_rate': 0.4, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 5}
[INFO]: 0.594667 (0.411065) with: {'dropout_rate': 0.5, 'image_size': (150, 150), 'num_classes': 39, 'weight_constraint': 1}

```

Figura 26: Resultados de la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*dropout_regularization*”

Podemos ver como la opción que el optimizador muestra como la ideal tiene un *dropout rate* de 0, lo que significa que ningún nodo será descartado. En cuanto al *dropout_regularization*, que en la imagen anterior se muestra como el *weight_constraint*, tiene la máxima *accuracy* con un valor de 3, combinado con un *dropout rate* nulo.

Aún así, cuando se vaya a entrenar la red neuronal con los parámetros optimizados se harán dos comprobaciones, uno sin *Dropout* y otro con la combinación de un *dropout rate* con un valor igual a 0,2 y un *kernel constraint* igual a 3. Esta dos combinación se muestran en la imagen 26 con una *accuracy* del 100 %.

La ejecución de esta estrategia ha tenido un coste de tiempo igual a 2 horas, 29 minutos y 33 segundos.

6.6.7. Número de neuronas

El último atributo a optimizar es el número de neuronas en las capas de la red neuronal convolucional. El parámetro que se debe introducir para poder ejecutar la estrategia optimizando este atributo es ”*number_neurons*”.

En la imagen 27 se muestran los rangos de valores que pueden adquirir las dos capas con un número de nodos configurables.

```
@staticmethod
def __get_parameters_number_neurons(n_classes, image_size):
    neurons_conv_layer = [1, 5, 10, 15, 20, 25, 30, 40, 50, 60, 80, 100, 120, 140]
    neurons_dense_layer = [10, 20, 25, 30, 40, 50, 60, 80, 100, 120, 140]

    param_grid = dict(neurons_conv_layer=neurons_conv_layer, neurons_dense_layer=neurons_dense_layer,
                      num_classes=[n_classes], image_size=[image_size])
    classifier = KerasClassifier(build_fn=create_model_number_neurons, epochs=EPOCHS, batch_size=BATCH_SIZE,
                                 verbose=2)

    return param_grid, classifier
```

Figura 27: Parámetros evaluados en la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*number_neurons*”

En la siguiente imagen aparece el resultado de la ejecución de esta estrategia.

```
[INFO]: Best: 1.000000 using {'image_size': (150, 150), 'neurons_conv_layer': 15, 'neurons_dense_layer': 100, 'num_classes': 39}
[INFO]: 0.666667 (0.471405) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 10, 'num_classes': 39}
[INFO]: 0.773333 (0.209261) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 20, 'num_classes': 39}
[INFO]: 0.666667 (0.471405) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 25, 'num_classes': 39}
[INFO]: 0.666667 (0.471405) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 30, 'num_classes': 39}
[INFO]: 0.666667 (0.471405) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 40, 'num_classes': 39}
[INFO]: 0.800000 (0.282843) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 50, 'num_classes': 39}
[INFO]: 0.960000 (0.051016) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 60, 'num_classes': 39}
[INFO]: 0.624000 (0.303789) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 80, 'num_classes': 39}
[INFO]: 0.466667 (0.410961) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 100, 'num_classes': 39}
[INFO]: 0.994667 (0.007542) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 120, 'num_classes': 39}
[INFO]: 0.952000 (0.067882) with: {'image_size': (150, 150), 'neurons_conv_layer': 1, 'neurons_dense_layer': 140, 'num_classes': 39}
[INFO]: 0.221333 (0.160577) with: {'image_size': (150, 150), 'neurons_conv_layer': 5, 'neurons_dense_layer': 10, 'num_classes': 39}
[INFO]: 0.202667 (0.286614) with: {'image_size': (150, 150), 'neurons_conv_layer': 5, 'neurons_dense_layer': 20, 'num_classes': 39}
[INFO]: 0.666667 (0.471405) with: {'image_size': (150, 150), 'neurons_conv_layer': 5, 'neurons_dense_layer': 25, 'num_classes': 39}
[INFO]: 0.266667 (0.316941) with: {'image_size': (150, 150), 'neurons_conv_layer': 5, 'neurons_dense_layer': 30, 'num_classes': 39}
[INFO]: 0.000000 (0.000000) with: {'image_size': (150, 150), 'neurons_conv_layer': 5, 'neurons_dense_layer': 40, 'num_classes': 39}
```

Figura 28: Resultados de la ejecución de la estrategia *Hyperparameter Optimization Strategy* con el parámetro ”*number_neurons*”

Según el resultado obtenido, la combinación más óptima para el número de neuronas son 15 en la capa convolucional y 100 en la densa oculta.

La ejecución de esta estrategia tiene un coste de tiempo de 10 horas, 56 minutos y 9 segundos. Este parámetro es el que ha tenido la ejecución más larga en la estrategia *Hyperparameter Optimization Strategy*.

6.6.8. Resultados de la optimización

Una vez realizadas todas las optimizaciones se creó un nuevo modelo de red neuronal convolucional mejorado. Este nuevo modelo se muestra en la imagen 29.

```
def __get_improved_sequential_model(n_classes, shape):
    model = Sequential()
    model.add(Conv2D(NEURONS_CONV_LAYER, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation=ACTIVATION,
                    input_shape=(shape[1], shape[2], 1), kernel_initializer=INIT_MODE,
                    kernel_constraint=max_norm(WIGHT_CONSTRAINT)))
    model.add(Dropout(DROPOUT_RATE))
    model.add(MaxPool2D(pool_size=(1, 1)))
    model.add(Flatten())
    model.add(Dense(NEURONS_DENSE_LAYER, kernel_initializer=INIT_MODE, activation=ACTIVATION))
    model.add(Dropout(DROPOUT_RATE))
    model.add(Dense(n_classes, activation='softmax'))

    return model
```

Figura 29: Modelo de la red neuronal convolucional obtenida a partir de los resultados de las ejecuciones de la estrategia *Hyperparameter Optimization Strategy*.

Se han realizado diversos cambios comparado con la red neuronal convolucional presentada en el apartado 6.2. Se han modificado el valor de varios atributos, tales como:

- El número de nodos en la capa convolucional y en la capa densa oculta. En la anterior versión de este modelo se utilizaron 25 nodos en la convolucional y 100 en la densa. Actualmente, se han reducido los 25 nodos a 15.
- Anteriormente no se definía ningún inicializador, pero como se muestra en la imagen 30, que hace referencia al constructor de la capa convolucional, el inicializador por defecto es el *glorot_uniform*. En el nuevo modelo se define con el inicializador *uniform*.
- Por último, se han añadido dos nuevas capas de *Dropout* y un *kernel constraint* en la capa convolucional. Este parámetro tiene un valor nulo cuando

no se define, como se muestra en la imagen 30, por lo tanto, en el modelo anterior no se restringía los pesos de las capas ocultas.

Actualmente se tienen dos conjuntos de valores a probar, uno con un *dropout rate* de 0.2 y un *kernel constraint* de 2, o un 0 de *dropout rate* y un 3 de *kernel constraint*.

```
def __init__(self,
            filters,
            kernel_size,
            strides=(1, 1),
            padding='valid',
            data_format=None,
            dilation_rate=(1, 1),
            groups=1,
            activation=None,
            use_bias=True,
            kernel_initializer='glorot_uniform',
            bias_initializer='zeros',
            kernel_regularizer=None,
            bias_regularizer=None,
            activity_regularizer=None,
            kernel_constraint=None,
            bias_constraint=None,
            **kwargs):
    super(Conv2D, self).__init__(
        rank=2,
        filters=filters,
        kernel_size=kernel_size,
```

Figura 30: Parámetros requeridos en el constructor de la clase *Conv2D* de la librería *keras.layers*.

Se ha entrenado la red neuronal convolucional mejorada con los dos conjuntos de valores del *dropout*, en la siguiente tabla se comparan los resultados obtenidos y los costes temporales y de memoria de cada uno de ellos.

	<i>Sin Dropout rate = 0.0 Kernel Constraint = 3</i>	<i>Dropout rate = 0.2 Kernel Constraint = 2</i>
Accuracy	99,72 %	97,26 %
Coste temporal entrenamiento	01:06:50 hora	01:29:03 hora
Coste temporal predicción	00:01:13 horas	00:01:05 horas
Coste memoria	394,4 MB	394,4 MB

Tabla 21: Comparativa de la red neuronal convolucional con los parámetros optimizados con y sin *Dropout*

Dados los resultados obtenidos en la tabla 21, podemos concluir que para el problema que estamos intentado solucionar, con los dos datasets combinados y con las características definidas de la red neuronal convolucional, obtenemos un resultado más óptimo sin *dropout*.

En cuanto al coste de ambas ejecuciones, tanto la suma del temporal como el de memoria es menor sin el *dropout*, lo que nos confirma cual es la mejor solución.

Aún así, una *accuracy* del 97,26 % es un resultado excelente, pero al tener un valor más elevado con un coste menor, se utilizará la primera opción.

Todos los resultados obtenidos a partir de las ejecuciones de la estrategia *Hyperparameter Optimization Strategy* se han guardado en un archivo denominado *hyperparameters.py*. La imagen 31 muestra el contenido de este archivo con los valores finales.

```
"""
After executing the --showOptimizedHyperparameter strategy, these are the best results obtained for the convolutional
neural network.
"""

EPOCHS = 10
BATCH_SIZE = 10
OPTIMIZER_ALGORITHM = 'SGD'
LEARN_RATE = 0.001
MOMENTUM = 0.0
INIT_MODE = 'uniform'
ACTIVATION = 'relu'
DROPOUT_RATE = 0.0
WEIGHT_CONSTRAINT = 3
NEURONS_CONV_LAYER = 15
NEURONS_DENSE_LAYER = 100
```

Figura 31: Archivo del proyecto que contiene los valores de los atributos optimizados obtenidos a partir de las ejecuciones de la estrategia *Hyperparameter Optimization Strategy*

Los valores de la tabla 21 han sido obtenidos a partir de un mismo modelo

modificando los dos atributos diferenciados. Esto significa que en el caso del *dropout rate* igual a cero, se ha entrenado la red neuronal con una capa adicional que no era necesaria. Utilizando la lógica diríamos que una capa de *Dropout* con un índice nulo únicamente estaría incrementando el coste de memoria, ya que en sí no tiene ninguna utilidad.

En la tabla 22 se comparan los mismos atributos que en la tabla 21, pero en este caso las dos redes neuronales convolucionales a comparar se diferencian en si tienen las capas de *Dropout* con un índice nulo, o si no disponen de estas capas.

Esta comparación se lleva a cabo para evaluar si el resultado que se espera utilizando la lógica es el mismo que el resultado real.

	<i>Dropout rate igual a cero</i>	<i>Sin las capas Dropout</i>
Accuracy	99,72 % - 99,75 %	99.46 % - 99.57 %
Coste temporal entrenamiento	01:06:50 hora	01:14:57 hora
Coste temporal predicción	00:01:13 horas	00:01:17 horas
Coste memoria	394,4 MB	394,4 MB

Tabla 22: Comparativa de la red neuronal convolucional con los parámetros optimizados con y sin *Dropout*

Se han ejecutado las estrategias de entrenamiento y de predicción diversas veces con la intención de poder comparar si realmente tienen características diferentes. En la tabla 22 se muestran los rangos de valores más significativos en esta comparación.

Se aprecia como hay una diferencia significativa en la *accuracy* entre ambas redes. En todos los entrenamientos que se han realizado, en ningún momento el modelo sin *Dropout* ha llegado a igualar el peor valor del modelo con el *dropout rate* nulo.

Al no comprender el porque de esta diferenciación en la *accuracy* entre ambas redes neuronales, se ha hecho una búsqueda exhaustiva por internet. La única información que se ha conseguido obtener ha sido una publicación en *Stack Overflow*¹, sin ninguna respuesta, y un *bug* abierto en *Git Hub*² que hacia referencia exactamente a este error. El *bug* tenía como título:

Keras Dropout layer changes results with dropout=0.0

¹<https://es.stackoverflow.com/>

²<https://github.com/>

Este se puede encontrar en el siguiente enlace:

<https://github.com/tensorflow/tensorflow/issues/10845>

El problema está en que este *bug* se abrió en 2017, y la solución que aparece en los comentarios no hace referencia al estado actual de la librería *tensorflow*. Con lo que no se puede entender exactamente el porque de esta diferenciación.

En este proyecto se utilizará el modelo sin la capa de *Dropout*, ya que no la otra opción no es una solución fiable si contiene un error. Aunque, aparentemente, pueda parecer que nos aporta una *accuracy* más elevada con un coste menor, no entendemos cual es la forma en la que está implementado. Al no poder dar una explicación a este problema, no se puede dar como valido el modelo con un *dropout rate* igual a cero.

En futuras investigación sería recomendable investigar cual es el modelo matemático que hay detrás de esta capa, de la librería *tensorflow*, para poder concluir si este *bug*, no solucionado, supondría un problema en futuros entrenamientos.

6.7. Comparativa de las experimentaciones 4 y 5

En la experimentación del apartado 6.6, se han evaluado unas ciertas características del modelo utilizado en la red neuronal convolucional. Se han obtenido unos resultados donde se mostraba cuales eran los valores óptimos para estos atributos, buscando una mayor *accuracy* en la red neuronal.

Después de probar las combinaciones especificadas, se ha obtenido el modelo que mejor se adapta al problema que se intenta solucionar. Se ha deducido que la mejor solución era utilizar una red neuronal con una capa convolucional de 25 nodos y una densa de 100, con el inicializador *glorot_uniform* y sin ninguna capa de *Dropout*.

Este ha sido el resultado obtenido dada las configuraciones de la optimización, pero lo que no se ha evaluado es la comparación de este resultado con el obtenido en el apartado 6.5. Se podrían dar tres posibilidades en cuanto a la comparación de la *accuracy*:

- No haber ninguna modificación. Aún haber evaluado los valores más óptimos de los atributos a configurar, al tener ya de por si una *accuracy* muy elevada, prácticamente perfecta, estas modificaciones podrían no implicar una mejora.
- Mejorar la eficiencia de la red neuronal. Al haber obtenido los mejores valores centrados exclusivamente en el problema que se intenta solucionar con el dataset con el que se hace el entrenamiento, podríamos haber encontrado una configuración más específica que nos incremente su eficiencia.
- Empeorar la eficiencia de la red neuronal. Las comprobaciones de las optimizaciones no se han podido realizar con todos los datos del dataset por un problema de coste de memoria y tiempo. Al haber hecho las pruebas con un conjunto de datos muy pequeño, comparado con el que se hace el entrenamiento real, se podrían haber obtenido resultados que no acabasen de ser específicos para nuestro problema.

Teniendo presente estas tres posibilidades se realizó el entrenamiento y las predicciones con la red neuronal, supuestamente, optimizada. A continuación, se muestra una tabla con los valores de la comparación.

	CNN no optimizada	CNN optimizada
Accuracy	99,30 %	99,52 %
Coste temporal entrenamiento	01:35:02 hora	01:06:50 hora
Coste temporal predicción	00:01:43 horas	00:01:13 horas
Coste memoria	657,2 MB	394,4 MB

Tabla 23: Comparativa de los resultados de las redes neuronales convolucionales de los apartados 6.5 y 6.6.

Evaluado los valores de la tabla 23, podemos apreciar como la red neuronal optimizada incrementa la *accuracy* 22 décimas. Este incremento puede parecer muy leve, pero al ser un valor de eficiencia tan elevado, el poder incrementar unas décimas significa una gran mejora en el modelo.

Además, podemos apreciar como el coste temporal de entrenamiento ha sido reducido media hora, mejorándose un 30,52 %. y en cuanto al coste de memoria, se han disminuido 262,8 MB, un 39,99 %.

Dados estos resultados, podemos concluir que la experimentación que se ha centrado en la optimización de la red neuronal convolucional ha sido exitosa, y esta es la mejor solución que se tiene hasta el momento.

6.8. Experimentación 6: Uso de clasificadores binarios

En el apartado 6.7, se muestra una posible solución a este proyecto, donde se entrena una red neuronal convolucional. Hasta el momento todas las implementaciones que se han realizado han estado centradas en analizar los datos de un modo categórico.

Este método nos ha aportado un rendimiento prácticamente perfecto, si nos centramos en evaluar únicamente su *accuracy*. La solución que se propone tiene un rendimiento del 99,52% con un coste de tiempo de entrenamiento y predicción de una hora y con un coste de memoria de 394,4 MB.

Aún haber obtenido un resultado excelente, se ha querido evaluar otra técnica. El objetivo es ampliar el conocimiento de las diferentes técnicas de clasificación y poder comprobar si esta también daría unos resultados tan satisfactorios.

Como se explica en los fundamentos, [en el apartado 4](#), hay diferentes tipos de clasificadores. Hasta ahora, se ha utilizado el clasificador multi-clase, ya que es el que mejor se adapta a las características de los datos con los que se está trabajando. Pero hay otra técnica que se puede utilizar y que no se ha implementado en las experimentaciones anteriores. Esta consiste en adaptar un clasificador binario para que sea capaz de clasificar más de una clase. La explicación de esta técnica se encuentra [en el apartado 4](#).

Se utilizará el modelo que mejores resultados nos ha proporcionado hasta el momento, en este caso es la red neuronal convolucional presentada en el apartado 6.7. Se modificarán los parámetros necesarios para que se adapte a las características del tipo de clasificador que se implementará.

Para la realización de este experimento se han implementado dos estrategias, siguiendo la misma estructura que con el árbol de decisión y las redes neuronales. Estas estrategias se denominan:

- **Train Binary Neural Network:** Para ejecutarla se deben definir los *Pickels* (archivos donde se almacenan los datos) a utilizar. Una vez definidos, se procesaran los datos con tal de poder crear tantos clasificadores como tipos de clases haya, en este caso, hay una clase por cada letra y número del lenguaje de signos, 39 en total.

Los datos dependerán del clasificador, cada clasificador se centra en identificar si las imágenes son una señal en concreto o no. Es por esto que los datos serán transformados en binarios previamente al entrenamiento de cada modelo.

Cuando se han creado todos los clasificadores, se construyen los modelos de las redes neuronales convolucionales y se entrena basados en los datos de *training*.

Antes de finalizar la ejecución, se almacenan todos los modelos en archivos con extensión *.h5*, comprimidos en un *zip* para minimizar el coste de memoria.

La imagen 32 muestra el diagrama de flujo de esta estrategia.

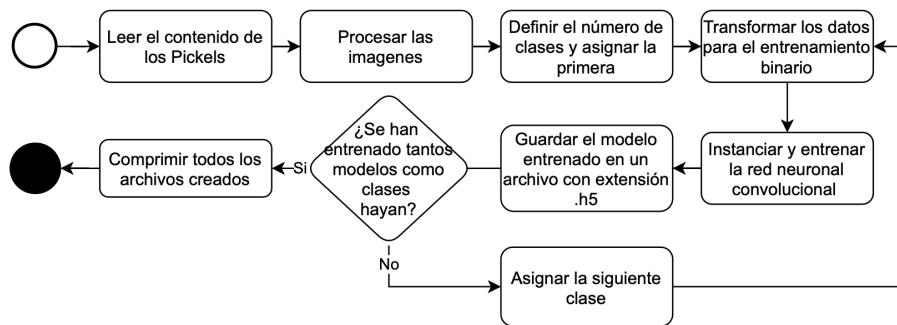


Figura 32: Diagrama de flujo de la estrategia *Train Binary Neural Network*

- **Accuracy Binary Neural Network:** Esta estrategia recuperará los modelos creados en la anterior ejecución. Por cada modelo, introducirá los datos de *testing* y evaluará la predicción de cada una de las redes neuronales binarias, mostrando la *accuracy* de cada una de ellas.

Una vez obtenidas todas las predicciones, se calculará, por cada dato introducido del *testing*, qué modelo ha proporcionado un valor más elevado, pudiendo calcular así cual es la clase resultante. A continuación, se mostrará la *accuracy* global de esta técnica.

La imagen 33 hace referencia al diagrama de flujo de la estrategia *Accuracy Binary Neural Network*.

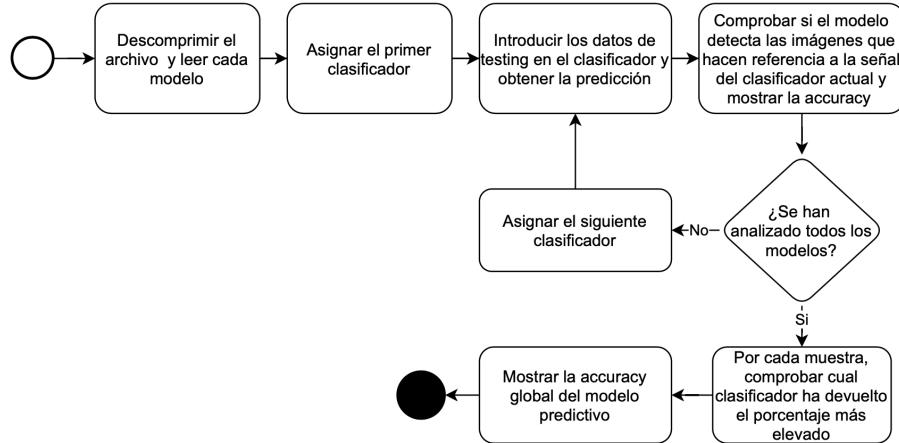


Figura 33: Diagrama de flujo de la estrategia *Accuracy Binary Neural Network*

Estas estrategias poseen un coste muy elevado, tanto de memoria como de tiempo. Se intentó entrenar el modelo predictivo con todas las muestras de los datasets, pero después de ocho horas, no había acabado de entrenar ni 1 red neuronal, y se debían entrenar 39.

Se decidió probar este modelo con menos datos, utilizando únicamente tres clases:

- La señal que hace referencia a la letra 'A'.
- La señal que hace referencia a la letra 'B'.
- La señal que hace referencia a la letra 'C'.

Se han restringido los valores de las muestras, con la intención de únicamente aceptar estos tres valores y reducir el tamaño del dataset. Una vez eliminados las muestras que no nos interesan, se han ejecutado las dos estrategias mencionadas. En la imagen 34 se muestra un diagrama de del entrenamiento con las tres clases mencionadas.

En el primer paso se obtienen las muestras buscadas, y se procesan para obtener las etiquetas (X) y el valor de los píxeles (Y) en dos estructuras diferentes. En el tercer paso, el cual se ejecuta secuencialmente, no en paralelo, se analizan los valores. La estructura, que almacena las etiquetas, modifica el valor de sus elementos, con un valor de 0, si no pertenece a la clase actual, y con uno si pertenece a la clase.

El cuarto paso se entrena todos los modelo, pero, como se muestra el diagrama de flujos de la imagen 32, el tercer y el cuarto paso se llevan a cabo secuencialmente. Una vez se ha finalizado el entrenamiento, se vuelve al paso tres con la siguiente clase.

El paso cinco hace referencia al *output* que proporcionarán estos modelos cuando se realiza la predicción.

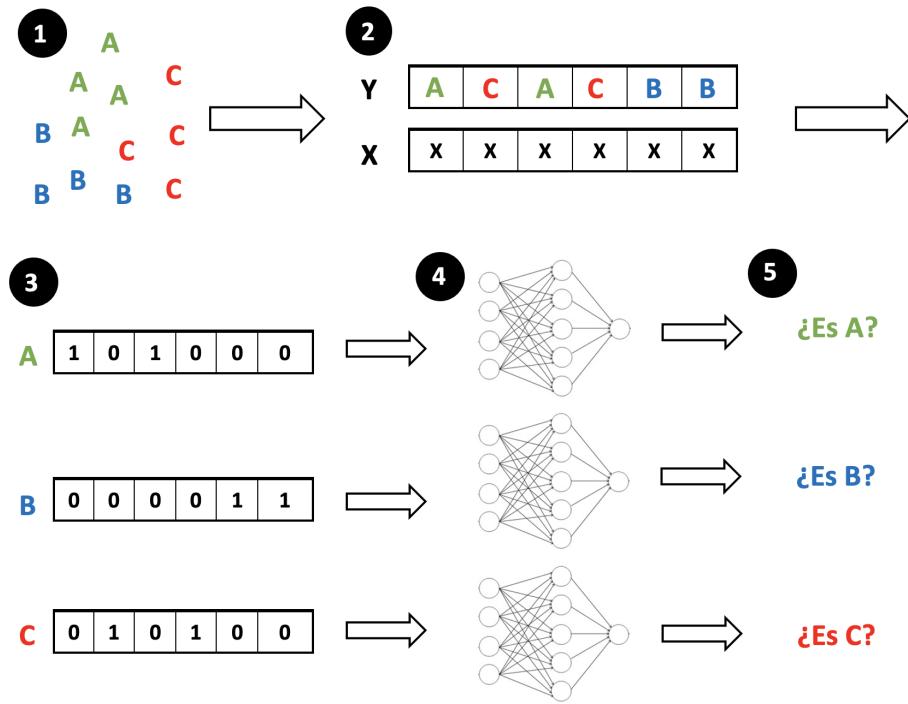


Figura 34: Diagrama de flujo de la estrategia *Accuracy Binary Neural Network*.

En la imagen 35, se muestra la accuracy obtenida con este modelo con las tres clases definidas. Se puede apreciar la accuracy de cada uno de los clasificadores, el cual ha sido:

- Clasificador del signo 'A': 100 %.
- Clasificador del signo 'B': 99,82 %.
- Clasificador del signo 'C': 99,95 %.

La accuracy global se calcula analizando el clasificador que ha devuelto la predicción más elevada, puede pasar que más de uno devuelva el máximo valor, pero como se muestra en la imagen 35 no se ha dado el caso en esta ejecución.

La accuracy global ha sido del 100 %, lo que muestra que esta técnica puede ser muy prometedora.

```
!python3 Src/main.py --accuracyBinaryNeuralNetwork abc abc_asl_alphabet_gray_150x150px-sign_gesture_gray_150x150px_mode
[INFO]: Strategy selected: --accuracyBinaryNeuralNetwork
[INFO]: Arguments entered:
        * Signs to train: abc
        * Neural Network model file: abc, abc_asl_alphabet_gray_150x150px-sign_gesture_gray_150x150px_models.zip
[INFO]: The accuracy of the binary neural network of the sign 'C' is: 100.00%
[INFO]: The accuracy of the binary neural network of the sign 'B' is: 99.82%
[INFO]: The accuracy of the binary neural network of the sign 'A' is: 99.95%
[INFO]: The global accuracy is: 100.00%
[INFO]: 0.00% of the correct samples had more than one solution.
[INFO]: Strategy executed successfully
[INFO]: Execution finished
[INFO]: The duration of the execution has been 00:00:41 [hh:mm:ss]
```

Figura 35: *Output* de la ejecución de la estrategia *Accuracy Binary Neural Network* con las señales: 'A', 'B' y 'C'.

La tabla 24, muestra las características de esta ejecución. Se puede apreciar como comprimir los archivos genera una reducción en el coste de memoria de la ejecución de ambas estrategias.

El tiempo de entrenamiento y de predicción puede parecer leve, comparado con los resultados de otros modelos predictivos implementados en anteriores estrategias. Pero debemos recordar que esta ha sido entrenada únicamente con tres clases, si se hubiese entrenado con 39, como es el dataset real, estos valores se dispararían.

Modelo predictivo con redes neuronales binarias de los signos 'A', 'B' y 'C'	
Accuracy	100 %
Tiempo ejecución del entrenamiento	00:47:38 horas
Tiempo ejecución de la predicción	00:00:41 horas
Coste de memoria de los archivos comprimidos	975,9 MB
Coste de memoria de los archivos sin comprimidos	1,18 GB

Tabla 24: Características de la ejecución del modelo predictivo con redes neuronales binarias con las muestras de los signos 'A', 'B' y 'C'.

Una vez obtenidos este rendimiento tan prometedor, se volvieron a ejecutar

las mismas estrategias, pero esta vez admitiendo las clases numéricas, las señales del cero al diez.

En este caso, los resultados no fueron tan favorables. Como se muestra en la imagen 36, la máxima *accuracy* ha sido del 100 %, pero el clasificador de la señal '4' nos proporciona una *accuracy* del 24,80 %, un valor muy pobre.

Al tener clasificadores con un rendimiento tan leve, afectan a la *accuracy* global del modelo predictivo, ya que proporcionan valores, en sus predicciones, más elevados que el clasificador correcto.

Figura 36: *Output* de la ejecución de la estrategia *Accuracy Binary Neural Network* con las señales numéricas.

Estos resultados pueden ser debidos a que el modelo de red neuronal, que se está utilizando, ha sido diseñado para el modelo predictivo multi-clase, no para ser entrenado de forma binaria. Se debería repetir la experimentación del apartado 6.6 con los datos de *training* que se utilizan en este apartado.

Aún haber obtenido una *accuracy* muy reducida en algunos clasificadores, más de la mayoría tienen un valor superior al 90 %. Con los modelos de red neuronal convolucional optimizados puede llegar a ser una solución muy buena para el problema que se está intentando resolver, siempre y cuando se sea consciente del elevado coste computacional que presenta.

7. Propuesta

Explicación de las diferentes estrategias.

Para la estrategia AccuracyDecisionTree hace falta instalar graphviz para poder mostrar (plot) el modelo de ábol de decisión. En el caso de un mac se tiene que hacer con: brew install graphviz

conda install graphviz python-graphviz

<https://www.mikulskibartosz.name/how-to-plot-the-decision-trees-from-xgboost-classifier/>

<https://drive.google.com/file/d/0B0c0MbnP6Nn-eUNRRkVOOGpkbFk/view?resourcekey=0-nVw3WhovKW5FPvPM5GPHfg>

8. Costes del proyecto

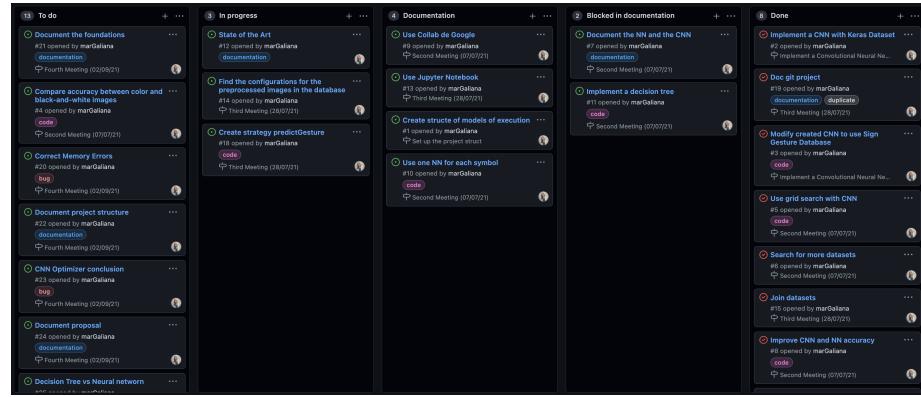
9. Conclusiones y líneas de futuro

9.1. Conclusiones

9.2. Líneas de futuro

A. Anexo

A.1. Proyecto Github



Referencias

- [1] Alexey Karpov, Irina Kipyatkova, and Milos Zelezny. Automatic technologies for processing spoken sign languages. *Procedia Computer Science*, 81:201–207, 2016.
- [2] Mark Borg and Kenneth Camilleri. Phonologically-meaningful subunits for deep learning-based sign language recognition. In *ECCV Workshops*, pages 199–217, 01 2020.
- [3] Lionel Pigou, Sander Dieleman, Pieter-Jan Kindermans, and Benjamin Schrauwen. Sign language recognition using convolutional neural networks. In Lourdes Agapito, Michael M. Bronstein, and Carsten Rother, editors, *Computer Vision - ECCV 2014 Workshops*, pages 572–578, Cham, 2015. Springer International Publishing.
- [4] John Bush Idoko. Deep learning based sign language translation system. *NICOSIA*, 1:1–109, 2016.
- [5] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.
- [6] Jiangbin Zheng, Zheng Zhao, Min Chen, Jing Chen, Chong Wu, Yidong Chen, Xiaodong Shi, and Yiqi Tong. An improved sign language translation model with explainable adaptations for processing long sign sentences. In *Computational Intelligence and Neuroscience*, volume 2020, page 11, 2020.
- [7] Taehwan Kim, Gregory Shakhnarovich, and Karen Livescu. Fingerspelling recognition with semi-markov conditional random fields. *2013 IEEE International Conference on Computer Vision*, pages 1521–1528, 2013.
- [8] Thad Starner, Joshua Weaver, and Alex Pentland. Real-time american sign language recognition using desk and wearable computer based video. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20:1371 – 1375, 01 1999.
- [9] Munee Al-Hammadi, Ghulam Muhammad, Wadood Abdul, Mansour Al-sulaiman, Mohammed A. Bencherif, Tareq S. Alrayes, Hassan Mathkour, and Mohamed Amine Mekhtiche. Deep learning-based approach for sign language gesture recognition with efficient hand gesture representation. *IEEE Access*, 8:192527–192542, 2020.
- [10] Danilo Avola, Marco Bernardi, Luigi Cinque, Gian Luca Foresti, and Cristiano Massaroni. Exploiting recurrent neural networks and leap motion controller for the recognition of sign language and semaphoric hand gestures. *IEEE Transactions on Multimedia*, 21(1):234–245, Jan 2019.

- [11] Matteo Rinalduzzi, Alessio Angelis, Francesco Santoni, Emanuele Buchicchio, A. Moschitta, Paolo Carbone, Paolo Bellitti, and Mauro Serpelloni. Gesture recognition of sign language alphabet using a magnetic positioning system. *Applied Sciences*, 11:5594, 06 2021.
- [12] Anthony Goldbloom (Co-founder and CEO). Kaggle. <https://www.kaggle.com>.
- [13] Ahmed Khan. Sign language gesture images dataset. <https://www.kaggle.com/ahmedkhanak1995/sign-language-gesture-images-dataset>, 2019.
- [14] Kareem. Dataset asl test and train. <https://www.kaggle.com/kareemalaa74/dataset-asl-test-and-train>, 2021.
- [15] David Cournapeau. scikit-learn. machine learning in python. <https://scikit-learn.org/stable/>.
- [16] Jason Brownlee. Grid search hyperparameters deep learning model python keras. <https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/>, August 2016.
- [17] Jason Brownlee. Weight initialization for deep learning neural networks. <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/>, February 2021.
- [18] Aakash Nain, Sayak Paul, and Margaret Maynard-Reid. Keras. <https://keras.io/>.
- [19] Pragati Baheti. 12 types of neural networks activation functions: How to choose? <https://www.v7labs.com/blog/neural-networks-activation-functions>.
- [20] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.