Reconocimiento de lenguaje de señas mediante redes neuronales LSTM

Didac Piferrer Iglesias

Resumen– En este trabajo de final de grado, se recoge un estudio realizado sobre el reconocimiento de lenguaje de señas a través de la estructura de una red neuronal recurrente (RNN), concretamente mediante una LSTM (*Long-Short Term Memory*). Se parte de un software inicial donde la recolecta de datos se realiza manualmente, y simplemente se trabaja con 3 clases. El artículo se centra en explicar como se ha dinzamizado este software, haciéndolo más escalable y óptimo. Esto ha sido posible gracias al diseño e implementación de un técnica de diezmado específica. Por último, se probaran diferentes sintonizaciones y arquitecturas y se seleccionará la que proporcione mejores prestaciones.
Palabras clave– Redes neuronales, RNN, Deep Learning, LSTM, Lenguaje de señas, Inteligencia Artificial

Abstract– In this final degree project, a study carried out on the recognition of sign language through the structure of a recurrent neural network (RNN), specifically through an LSTM (*Long-Short Term Memory*), is collected. It starts from an initial software where the data collection is done manually, and you simply work with 3 classes. The article focuses on explaining how this software has been streamlined, making it more scalable and optimal. This has been possible thanks to the design and implementation of a specific decimation technique. Finally, different tunings and architectures will be tested and the one that provides the best performance will be selected.

Keywords– Neural networks, RNN, Deep Learning, LSTM, Sign Language, Artificial Intelligence

1 Introducción

Este documento recoge una posible solución a un problema que afecta a una minoria de la población mundial. Según la Federación Mundial de Sordos [1], existen aproximadamente 70 millones de personas sordas en todo el mundo. Más del 80 por ciento vive en países en desarrollo y como colectivo, utilizan más de 300 diferentes lenguas de señas.

Si se focaliza el problema a nivel nacional, en España hay más de un millón de personas sordas [2]. De ellas, unas 70.000 utilizan el lenguaje de signos. Motivo suficiente para tratar de buscar una herramienta que facilite la comunicación de esta minoria con el resto de personas.

El primer problema real resuelto por la aplicación de una red neuronal fue implementado en 1960 por Bernard Widrow y Marcial Hoff. Desarrollaron el modelo Adaline (*ADAptative LINear Elements*) (filtros adaptativos para eliminar ecos en las líneas telefónicas), que se ha utilizado comercialmente durante varias décadas. Actualmente, las redes neuronales son un campo inmensamente amplio de investigación donde se han realizado avances brutales. Debido a estos grandes avances tecnológicos, hoy se puede plantear una posible solución a un prblema complejo, como es la comunicación entre una persona sorda/muda y una persona que no lo es, gracias a las redes neuronales.

En este artículo se planteará como un problema de clasificación, donde cada seña representa una palabra (puediendo esta ser un nombre, verbo, adverbio, etc.). Para ello, se usará una variante de la RNN (*Recurrent Neuronal Network*), la LSTM (*Long-Short Term Memory*). Se parte de la base de un proyecto realizado en Jupyter-Notebook [https://jupyter.org/] de una red neuronal capaz de reconocer 3 clases (en este caso, 3 señas), (*"Hello", "Thanks", "I Love You"*) [3].

• E-mail de contacto: didacpiferrer@hotmail.com
• Trabajo tutorizado por: Jose Lopez Vicario (Departamento Telecomunicaciones y Ingeniería de Sistemas)
• Curs 2021/22

La captación de los datos se realiza en el instante, es decir, el propio software abre la cámara del ordenador y proporciona una serie de indicaciones para que el usuario empiece a crear por cuenta propia las señas que alimentaran la red neuronal.

Este software no es escalable, puesto que si se quisiera aumentar el número de señas a detectar, no se podría poner a una persona a realizar todas las señas deseadas. Además, este solo alimenta a la red neuronal con las señas recolectadas en ese preciso instante, o con señas que se hayan recolectado con el mismo software previamente. Por esta razón, se ha decidido modificar el software y permitir el procesado de vídeos grabados previamente, con cualquier dispositivo y desde cualquier parte del mundo.

2 Estado del arte

En breves palabras, la Inteligencia Artificial corresponde a los softwares que tienen la capacidad de pensar y razonar como lo haría un humano (ver en la figura 1). Dento de este campo, se encuentra el *Machine Learning* el cual mediante algoritmos permite que el software aprenda (se dice que el software aprende cuando su desempeño mejora con la experiencia y mediante el uso de datos).Y por último, dentro del campo del *Machine Learning*, se encuentra el *Deep Learning* que es un conjunto de algoritmos de aprendizaje automático que intenta modelar abstracciones de alto nivel en datos usando arquitecturas computacionales que admiten transformaciones no lineales múltiples e iterativas de datos expresados en forma matricial o tensorial.

Este artículo se centra en trabajar sobre un modelo *Deep Learning*. Para ello se presentaran los conocimientos básicos para poder entender con detalle los procedimientos y estudios realizados.

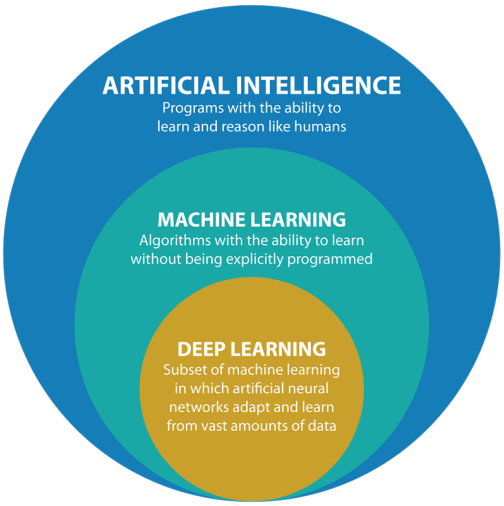

Fig. 1: Estructura de los diferentes subconjuntos dentro de la Inteligencia Artificial

2.1. Aprendizaje supervisado

Se encuentran 4 tipos de aprendizaje [16]: supervisado, no supervisado, semisupervisado y por refuerzo. Sin embargo, el problema que se quiere solucionar requiere un algoritmo de clasificación, y este forma parte del aprendizaje supervisado.

Los algoritmos de aprendizaje supervisado basan su aprendizaje en un conjunto de datos de entrenamiento previamente etiquetados. Se entiende por etiquetado cuando para cada ocurrencia del conjutno de datos de entrenamiento se conoce el valor de su atributo objetivo. Esto permite al algoritmo aprender una función capaz de predecir el atributo objetivo para un conjunto nuevo de datos.

- **Algoritmos de clasificación**: los algoritmos de clasificación se usan cuando el resultado es una etiqueta discreta. Esto quiere decir que se utilizan cuando la respuesta se fundamenta en un conjunto finito de resultados.
- **Algoritmos de regresión**: El análisis de regresión es un subcampo del aprendizaje automático supervisado cuyo objetivo es establecer un método para la relación entre un cierto número de características y una variable objetivo continua.

2.2. La neurona

La etimología de la red neuronal viene inspirada por las redes neuronales biológicas que forman parte del cerebro humano y el sistema nervioso. Entonces, la neurona es una célula (unidad anatómica fundamental de todos los organismos vivos) que sirve para procesar y transmitir información a través de señales químicas y eléctricas.

Análogamente, una neurona es la unidad básica de procesamiento que se puede encontrar dentro una red neuronal. Similiar a las neuronas biológicas, estas neuronas tienen conexiones de entrada a través de los que reciben estímulos externos, los valores de entrada. Con estos valores las neuronas realizaran un cálculo interno y generara un valor de salida. Básicamente, una neurona no deja de referirse con otro nombre a una función matemática [4][5].

Internamente, la neurona utiliza todos los valores de entrada para realizar una suma ponderada de estos. La ponderación de cada una de las entradas viene dada por el peso que se le asigna a cada una de las conexiones de entrada. Estos pesos, son los parámetros del modelo, y son los parámetros que se van ajustando para que la red neuronal aprenda. Además, hay un parámetro adicional, conocido como *bias* para que la función no esté condicionada a pasar por el origen, permitiendo de esta manera un mayor ajuste a la realidad.

2.2.1. Funciones de activación

A continuación, se presentarán las funciones de activación más usadas, así como una pequeña descripción de cada una de ellas con su representación gráfica en la figura 2.

Función lineal (Linear) : Se trata de la función identidad. Esta función hace que la salida sea igual a la entrada. Si la función de activación es esta, entonces esa neurona se comporta exactamente igual que una regresión lineal.

Función escalón : Útil cuando la salida es categórica y se pretende clasificar. Sin embargo, en la práctica no es muy utilizada debido a que el escalón hace que sea difícil trabajar con su derivada.

Función sigmoide (*Sigmoid*) : Resulta muy interesante porque gracias a ella los valores muy grandes convergen a 1 y los valores muy pequeños a -1, por lo que sirve para representar probabilidades. Tiene un inconveniente y es que su rango no está centrado en el origen. No obstante, este último problema se puede solventar usando la función tangente hiperbólica, que resulta muy similar.

Función tangente hiperbólica (Tanh) : La función de activación de Tangente Hiperbólica, tal y como se acaba de comentar, tiene la salida ajustada en cero debido a que su rango está entre -1 y 1. La optimización que provee es más fácil de usar en la práctica que la función sigmoide o lineal.

Función unidad rectificada lineal (ReLu) : Se trata de una de las funciones de activación más utilizadas. Se comporta como una función constante para valores negativos y como una función lineal para valores positivos.

Función *Softmax* : La función softmax es una función que convierte un vector de K valores reales en un vector de K valores reales que suman 1 y es usada para la clasificación multiclase. Los valores de entrada pueden ser positivos, negativos, cero o mayores que uno, pero softmax los transforma en valores entre 0 y 1, para que puedan interpretarse como probabilidades. A medida que el valor es mayor, mayor será su probabilidad asociada. De echo, la función *softmax* y la función *sigmoid* son similares, la primera opera sobre un vector mientras que la segunda toma un escalar.

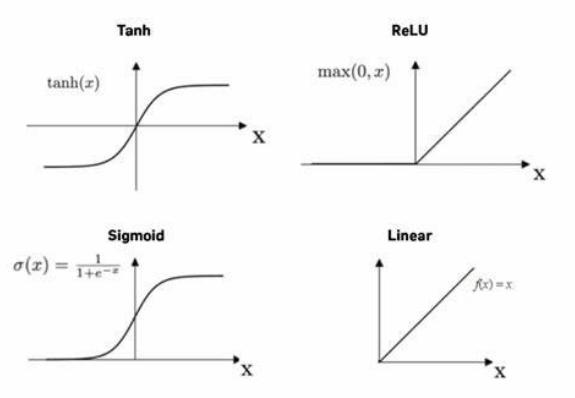

Fig. 2: De las funciones de activación nombradas, en este proyecto se han usado la Tanh, ReLU , Sigmoid y Softmax

2.3. Las redes neuronales

Una neurona por si sola tiene tiene grandes limitaciones y no realiza grandes operaciones. Es por esta razón que surge la necesidad de agrupar estas neuronas para llevar a cabo procesos más complejos. Estas neuronas comentadas previamente se organizan en capas para formar así la red neuronal. La cual contiene una capa de entrada, una o varias capas ocultas y una capa de salida.

Desde este punto hasta el final de esta subsección, se explicará el proceso de aprendizaje en las redes neuronales (ver esquema en la figura 3) y los diferentes factores involucrados.

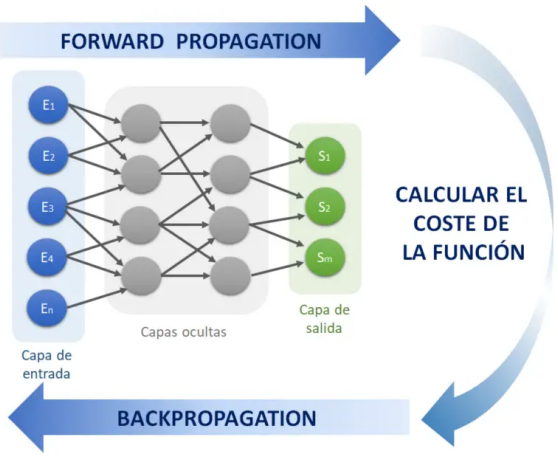

Fig. 3: Proceso de aprendizaje de una red neuronal

Tal y como se ha comentado anteriormente, una neurona puede verse como una función matemática que recibe unos valores de entrada y arroja un valor de salida.

Esos valores que reciben se ven afectados por unos parámetros o pesos. Estos parámetros determinan la influencia que tiene un valor de entrada en la función de activación. De esta manera, dichos pesos tienen una importancia directa en el error que pueda tener la red neuronal en sus predicciones. Este proceso que afecta a todas y cada una de las neuronas, desde que se recibe la entrada en la primera capa hasta que se obtiene el valor de salida, es conocido como *forward propagation*.

2.3.1. Loss function

Una vez la red neuronal realiza las predicciones de los *inputs*, la función de pérdida o *loss function* es la encargada de cuantificar el error entre la predicción arrojada y el valor real. Al diliar en este estudio con un problema de clasificación con más de dos clases , la *loss function* utilizada es la *categorical cross-entropy* (figura 4).

La función del optimizador es ir optimizando los valores de los parámetros obtenidos para reducir el error cometido por la red, es decir, minimizar la función de pérdida. El proceso mediante el cual se realiza esto se conoce como *backpropagation*[7] (veáse el esquema resumen A.3) [8].

$$\text{Loss} = -\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

Fig. 4: Función de pérdida *Categorical cross-entropy*. Contra más cercano a 0 sea el resultado, más preciso será el modelo que se esté entrenando.

2.3.2. Optimizadores

- **Descenso de gradiente estocástico** (SGD): Es el método básico de optimización de redes neuronales. Selecciona aleatoriamente una muestra de un *batch* del conjunto de datos para actualizar el parámetro de peso en función del valor de degradado en el punto actual, es decir, aplicar el descenso de gradiente [6]. La fórmula viene dada por la figura 6 y gráficamente se podría representar como en la figura 5.

- ***Adaptive Gradient Algorithm*** (AdaGrad): El algoritmo AdaGrad [8] introduce una variación muy interesante en el concepto de factor de entrenamiento (*Learning rate*[1]): en vez de considerar un valor uniforme para todos los pesos, se mantiene un factor de entrenamiento específico para cada uno de ellos. Sería inviable calcular este valor de forma específica así que, partiendo del factor de entrenamiento inicial, AdaGrad lo escala y adapta para cada dimensión con respecto al gradiente acumulado en cada iteración.

- ***Root Mean Square Propagation*** (RMSprop): RMSProp es un algoritmo similar. También mantiene un factor de entrenamiento diferente para cada dimensión, pero en este caso el escalado del factor de entrenamiento se realiza dividiéndolo por la media del declive exponencial del cuadrado de los gradientes

- ***Adaptive moment estimation*** (Adam): El algoritmo Adam combina las bondades de AdaGrad y RMSProp. Se mantiene un factor de entrenamiento por parámetro y además de calcular RMSProp, cada factor de entrenamiento también se ve afectado por la media del momentum del gradiente. Los algoritmos mas recientes como este, están construidos en base a sus predecesores, por tanto se puede esperar que su rendimiento sea superior. Es por eso que el modelo original se decanta por este.

2.3.3. Overfitting y Underfitting

El *underfitting* ocurre cuando un modelo estadístico o un algoritmo de aprendizaje automático no puede capturar la tendencia subyacente de los datos, esto ocurre cuando el modelo es demasiado simple. Unas posibles maneras de reducirlo sería aumentado los datos, entrenar con más datos, o por ejemplo, reducir la complejidad de los datos.

[1]En el aprendizaje automático y las estadísticas, la tasa de aprendizaje es un parámetro de ajuste en un algoritmo de optimización que determina el tamaño del paso en cada iteración mientras se mueve hacia un mínimo de una función de pérdida.

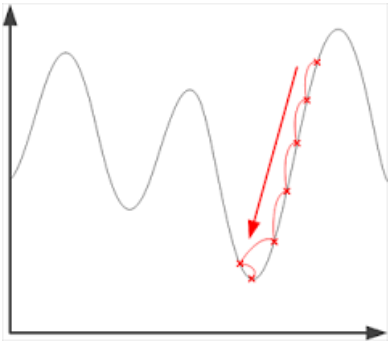

Fig. 5: Representación gráfica del algoritmo del descenso del gradiente

$$w_{k+1} = w_k - \eta \nabla f_{w_k}(x^i)$$

Fig. 6: Función del descenso del gradiente: resta entre los parámetros actuales y la derivada de la función de costes multiplicada por el *Learning Rate*

El *overfitting* es un error de modelado que se produce cuando una función se ajusta demasiado a un conjunto limitado de puntos de datos. Se podría decir que el modelo "memoriza" los datos de entrenamiento y no predice bien datos nunca vistos. Dos posibles maneras para reducirlo son el *dropout* y el *Early Stopping*.

En la figura 7 se puede ver como un modelo que no alcanza una pérdida cercana a cero no es óptimo, y no va a generalizar correctamente los datos, sin embargo, si el modelo entrena demasiado, puede llegar a memorizar características específicas y no generalizar correctamente. Es entonces cuando la pérdida de los datos de validación empieza a aumentar y la de los datos de entrenamiento a disminuir, puesto que los datos de entrenamiento los predice todos correctamente.

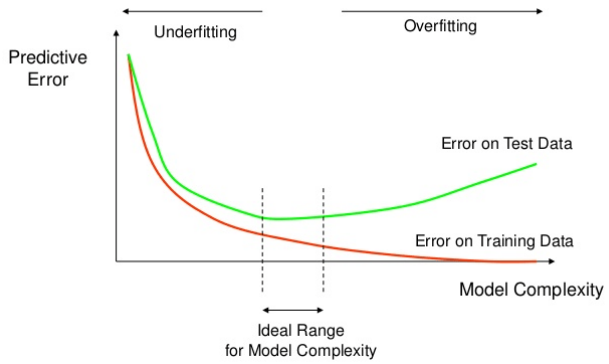

Fig. 7: Gráfica donde se distingue entre *Underfitting* y *Overfitting*

2.3.4. Dropout y Early Stopping

La técnica de regularización de *dropout* [20] desactiva de forma aleatoria un porcentaje de neuronas de la capa de la red neuronal. En cada iteración de la red neuronal, el *dropout* desactivará diferentes neuronas, tal y como se muestra

en la figura 8. Las neuronas desactivadas no se toman en cuenta para el *forwardpropagation* ni para el backwardpropagation lo que obliga a las neuronas cercanas a no depender tanto de las neuronas desactivadas. Este método ayuda a reducir el overfitting ya que las neuronas cercanas suelen aprender patrones que se relacionan y estas relaciones pueden llegar a formar un patrón muy especifico con los datos de entrenamiento, con *dropout* esta dependencia entre neuronas es menor en toda la red neuronal, de esta manera la neuronas necesitan trabajar mejor de forma solitaria y no depender tanto de las relaciones con las neuronas vecinas.

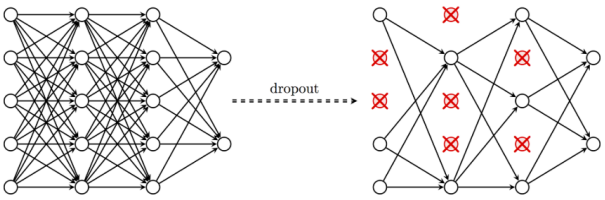

Fig. 8: Aplicación de la técnica *dropout* en una red neuronal de ejemplo

Por otro lado, el ***early stopping*** es de útil aplicación cuando se entrena un modelo que llega a alcanzar valores de pérdida bastante buenos hasta que esta pérdida empieza a aumentar en el caso de los datos de validación. Por ejemplo, en la figura 7 sería una buena técnica aplicar el *early stopping* dentro del rango ideal y detener el proceso de entrenamiento.

2.4. Recurrent Neural Networks

Una red neuronal recurrente (RNN) [9] es un tipo de red neuronal artificial que utiliza datos secuenciales o datos de series temporales. Estos algoritmos de aprendizaje profundo se usan comúnmente para problemas ordinales o temporales, como traducción de idiomas, procesamiento de lenguaje natural (nlp), reconocimiento de voz y subtítulos de imágenes; se incorporan a aplicaciones populares como Siri, búsqueda por voz y Google Translate. Al igual que las redes neuronales convolucionales y de avance (CNN), las redes neuronales recurrentes utilizan datos de entrenamiento para aprender. Se distinguen por su ”memoria”, ya que toman información de entradas anteriores para influir en la entrada y salida actual. Mientras que las redes neuronales profundas tradicionales asumen que las entradas y salidas son independientes entre sí, la salida de las redes neuronales recurrentes depende de los elementos previos dentro de la secuencia. Si bien los eventos futuros también serían útiles para determinar el resultado de una secuencia determinada, las redes neuronales recurrentes unidireccionales no pueden tener en cuenta estos eventos en sus predicciones.

2.4.1. Recurrent Neural Network vs. Feedforward Neural Network

Para entender mejor este concepto vamos a poner el ejemplo de una frase coerente como por ejemplo “no me encuentro bien”. Para que esta frase tengo sentido y se entienda como tal, tiene que ser expresada en ese correcto orden. Como resultado, las redes recurrentes deben tener en cuenta la posición de cada palabra en el idioma y usan esa información para predecir la siguiente palabra en la secuencia. En la figura 9 se puede ver una representación gráfica de la comparativa.

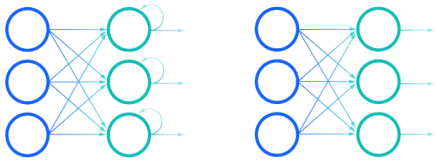

Fig. 9: Comparación de redes neuronales recurrentes (a la izquierda) y redes neuronales *feedforward* (a la derecha)

La imgen 10 “enrollada”de la RNN representa toda la red neuronal, o más bien la frase predicha completa, como “no me encuentro bien”. El visual “desenrollado” representa las capas individuales, o pasos de tiempo, de la red neuronal. Cada capa se asigna a una sola palabra en esa frase, como “bien”. Las entradas previas, como “no” y “me”, se representarían como un estado oculto en el tercer paso de tiempo para predecir la salida en la secuencia, “encuentro”.

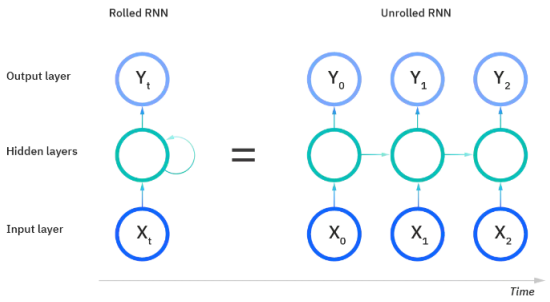

Fig. 10: Arquitectura simplificada de la RNN

Otra característica distintiva de las redes recurrentes es que comparten parámetros en cada capa de la red. Mientras que las redes *feedforward* tienen diferentes pesos en cada nodo, las redes neuronales recurrentes comparten el mismo parámetro de peso dentro de cada capa de la red. Dicho esto, estos pesos aún se ajustan a través de los procesos de *backpropagation* y descenso de gradiente para facilitar el aprendizaje por refuerzo.

Las redes neuronales recurrentes aprovechan el algoritmo de *backpropagation through time* (BPTT) [11] para determinar los gradientes, que es ligeramente diferente de la *backpropagation* tradicional, ya que es específico de los datos de secuencia. Los principios de BPTT son los mismos que los de la *backpropagation* tradicional, donde el modelo se entrena a sí mismo calculando errores desde su capa de salida hasta su capa de entrada. Estos cálculos permiten ajustar los parámetros del modelo adecuadamente. BPTT difiere del enfoque tradicional en que BPTT suma los errores en cada paso de tiempo, mientras que las redes feedforward no necesitan sumar errores ya que no comparten parámetros en cada capa.

A través de este proceso, las RNN tienden a encontrarse con dos problemas, conocidos como gradientes de explosión y gradientes de desaparición [12]. Estos problemas se

definen por el tamaño del gradiente, que es la pendiente de la función de pérdida a lo largo de la curva de error. Cuando el gradiente es demasiado pequeño, continúa haciéndose más pequeño, actualizando los parámetros de peso hasta que se vuelven insignificantes, es decir, 0. Cuando eso ocurre, el algoritmo ya no está aprendiendo. Los gradientes explosivos se producen cuando el gradiente es demasiado grande, lo que crea un modelo inestable. En este caso, los pesos del modelo crecerán demasiado y finalmente se representarán como NaN (Not a Number). Una solución a estos problemas es reducir la cantidad de capas ocultas dentro de la red neuronal, eliminando parte de la complejidad en el modelo RNN.

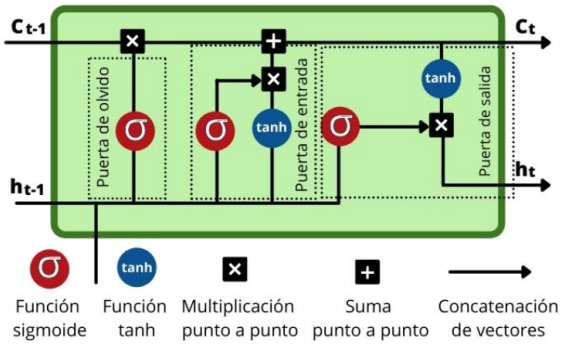

Fig. 11: Arquitectura de una celda LSTM

2.4.2. Variantes de RNN

***Bidirectional recurrent neural networks* (BRNN)** : se trata de una arquitectura de red variante de las RNN. Si bien los RNN unidireccionales solo pueden extraerse de entradas anteriores para hacer predicciones sobre el estado actual, los RNN bidireccionales extraen datos futuros para mejorar su precisión. Si se regresa al ejemplo de ”no me encuentro bien.ªnterior en este artículo, el modelo puede predecir mejor que la segunda palabra de esa frase es ”me”si sabe que la última palabra de la secuencia es ”bien”.

***Long short-term memory*(LSTM)** : esta es una arquitectura RNN popular y es la que va a ser usada en este estudio. Fue presentada por Sepp Hochreiter y Juergen Schmidhuber como una solución al problema del gradiente de fuga. Si el estado anterior que influye en la predicción actual no pertenece al pasado reciente, es posible que el modelo RNN no pueda predecir con precisión el estado actual.

Como ejemplo, se quieren predecir las siguientes palabras en cursiva: “Alicia es alérgica a las nueces. Ella no puede comer mantequilla de maní”. El contexto de una alergia a los frutos secos puede ayudar a anticipar que el alimento que no se puede comer contiene frutos secos. Sin embargo, si ese contexto fuera unas pocas oraciones antes, entonces sería difícil, o incluso imposible, que la RNN conectara la información.

Para remediar esto, los LSTM tienen “celulas” en las capas ocultas de la red neuronal, que tienen tres puertas [13], :una puerta de entrada, una puerta de salida y una puerta de olvido. Estas puertas controlan el flujo de información que se necesita para predecir la salida en la red. Por ejemplo, si los pronombres de género, como “ella”, se repitieron varias veces en oraciones anteriores, puede excluirlos del estado de la celda. Así pues, las celdas LSTM (11) contienen, a diferencia de las celdas de una RNN, una entrada extra, la de la “célula”, que servirá para poder mantener la información relevante. Gracias a las puertas se consiguen mantener dos estados en cada unidad LSTM, el estado de la “célula” y el estado oculto.

- **Puerta del olvido**: Esta puerta decide que información permanece y cual se olvida. Esto se consigue con la función sigmoide la cual tiene un dominio de 0 a 1. Cuanto más cerca del 0 menos importante es y cuanto más cerca del 1 más importante es.
- **Puerta de entrada**: El papel de esta puerta consiste en actualizar el estado oculto (*hidden state*) de la célula. Para esta tarea, el nuevo input es añadido al estado oculto de un tiempo anterior. La cantidad de información que se conserva se controla con la función sigmoide transformando los valores entre 0 y 1, y realizando una multiplicación con la función tanh. Cero significa que es poco importante y por tanto puede eliminarse. La unidad significa que es importante y que ese conocimiento debe permanecer.
- **Puerta de salida**: Esta puerta se encarga de decidir cual será el estado oculto de la célula en el siguiente *timestep*. Para ello hace uso de la función sigmoide y tanh.

***Gated recurrent units* (GRUs)** : Esta variante de RNN es similar a los LSTM, ya que también funciona para abordar el problema de la memoria a corto plazo de los modelos RNN. En lugar de utilizar una información de regulación de “estado celular”, utiliza estados ocultos y, en lugar de tres puertas, tiene dos: una puerta de reinicio y una puerta de actualización. Al igual que las puertas dentro de los LSTM, las puertas de reinicio y actualización controlan la cantidad y la información que se debe retener.

2.5. Estudios relacionados

Recientemente, destacamos dos estudios realizados con la detección de gestos y el reconomiento de acciones.

El primero [14], propone una red LSTM profunda totalmente conectada de extremo a extremo para el reconocimiento de acciones basado en esqueleto. Inspirados por la observación de que las co-ocurrencias de las articulaciones caracterizan intrínsecamente las acciones humanas, toman el esqueleto como entrada en cada intervalo de tiempo e introducen un

nuevo esquema de regularización para aprender las características de co-ocurrencia de las articulaciones del esqueleto. Para entrenar la red LSTM profunda de manera efectiva, proponen un nuevo algoritmo de *dropout* que opera simultáneamente en las puertas, células y respuestas de salida de las neuronas LSTM. Los resultados experimentales en tres conjuntos de datos de reconocimiento de acciones humanas demuestran consistentemente la efectividad del modelo que proponen.

El segundo [15], se presenta un método basado en el aprendizaje profundo para el reconocimiento de lenguaje de señas (SLR). El enfoque que muestran representa información multimodal (RGB-D) a través de imágenes dinámicas para describir la ubicación y el movimiento de la mano. Además, extraen un marco representativo que describe la forma de la mano, que es la entrada a dos modelos de CNN de transmisión múltiple. Por último, aplican una etapa de fusión posterior para la clasificación final. Los resultados experimentales también demuestran la viabilidad del enfoque propuesto.

En este estudio se quiere tener estos dos artículos como referencia, pero no se va a pretender mejorar las *accuracy*es obtenidas, debido a que se este se centra en aplicar la técnica de diezmado específico y ver como esta se comporta frente a arquitecturas no tan complejas. Un posible frente abierto sería probar la técnica del diezmado específico, que se describirá a continuación, en estos dos anteriores ejemplos de redes, puesto que esta permite una flexibilidad bastante grande a la hora de predecir los vídeos, explicado con detalle en la sección 6.

3 Objetivos

Dentro del mundo del reconocimiento de lenguaje de señas se pueden encontrar diferentes estudios que trabajan con las señas como imagenes estáticas. Es menos común ,debido a su complejidad, ver redes neuronales que procesen vídeos de señas. Además, para la comunicación fluida en el lenguaje de señas, no basta con expresar con imágenes estáticas letras. Es el seguido de sequencias de gestos co-ocurrentes de las manos, los brazos y expresiones faciales las que llevan a comunicar de forma completa lo que la persona quiere expresar.

Es por esto que con este estudio se quiere contribuir con la ayuda de de la minoria de sordo/mudos y aprender a trabajar con problemas que involucran las redes neuronales han sido pues las dos grandes motivaciones para realizar este artículo.

- El **principal objetivo** de la tesis, es transformar el software original para que se pueda entrenar la red neuronal con vídeos previamente grabados desde cualquier lugar, haciendo esta más dinámica, escalable y óptima.
- El **objetivo secundario** residirá en tratar de conseguir un modelo para el nuevo conjunto de datos que solucione el problema de clasificación.

Para cumplir los objetivos, se han desglosado estos en varias tareas:

1. Entender y replicar el programa original en local.
2. Diseñar un algoritmo específico de diezmado.
3. Sintonización de la red neuronal.
4. Evaluación de los resultados

4 Herramientas usadas

Como ya se ha comentado anteriormente, se parte de un programa realizado con Python v3.7 [https://docs.python.org/3/index.html] que capta los datos a través de la cámara del dispoistivo usando la libreria opencv-python [https://docs.opencv.org/4.x/], en el apéndice A.4 se puede apreciar el flujo de trabajo y como se ha eliminado la herramienta de la cámara en la versión final, haciendo esta escalable y mucho más óptima.

Para el procesado de los vídeos se ha usado el paquete de Mediapipe [https://google.github.io/mediapipe/], ya que una de las soluciones que proporciona este paquete es la *MediaPipe Holistic* que permite la percepción en vivo de poses humanas simultáneas, puntos de referencia faciales y seguimiento de manos en tiempo real. También se ha usado la libreria Keras ya que proporciona un acceso de alto nivel mediante una API a las funcionalidades de la librería tensorflow, usada para la creación a bajo nivel de modelos de *Deep learning* [https://www.tensorflow.org/api_docs/python/tf/keras/].

Se ha trabajado con la herramienta Anaconda [https://www.anaconda.com/] desplegando el aplicativo Jupyter-Notebook, y cuando las capacidades hardware se han visto comprometidas, se ha contratado el servicio de Google Colab Pro+ [https://colab.research.google.com/].

5 Software Original

El autor divide el cuaderno de Jupyter-Notebook [3] en 11 partes, que se resumen en los siguientes apartados:

1. Instalación de dependencias, definición de las funciones y creación del directorio de trabajo.
2. Recolecta de los vídeos a través de la cámara del computador.
3. Procesamiento de los datos y creación de los vectores de características
4. Construcción y entrenamiento la red neuronal LSTM
5. Evaluación de los resultados
6. Testeo en tiempo real

```
#Original_relu
model = Sequential()
model.add(LSTM(64, return_sequences=True, activation='relu',
               input_shape=(no_frames,1662)))
model.add(LSTM(128, return_sequences=True, activation='relu'))
model.add(LSTM(64, return_sequences=False, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(actions.shape[0], activation='softmax'))
```

Fig. 12: Arquitectura de la red neuronal LSTM propuesta por el autor

La configuración de la red neuronal propuesta por el autor en la figura 12 plantea un modelo de 3 capas LSTM y 3 capas densas. Las 5 primeras capas mantienen la misma función de activación, ReLu a excepción de la última capa densa de salida que trabaja con la función de activación *softmax* para proporcionar las diferentes probabilidades de cada clase.

Una vez replicado y puesto a punto en la máquina local, se decidió duplicar el conjunto de datos, es decir, recolectar 3 clases más, con 30 vídeos cada una.

Se observaron los resultados obtenidos del modelo original duplicando el número de clases y este parecía sufrir *overfitting* [18] (figura 13), problema ocasionado cuando el modelo no generaliza bien a partir de los datos de entrenamiento a datos no vistos, ya que se ve como la línea azul correspondiente a los datos de validación, en el caso de la precisión se estanca en 0.7, y la función de pérdida o *loss*, se observa como en el caso de los datos de validación a partir de la epoch 100 empieza a aumentar. Sin embargo, con el conjuto de datos de entramiento la función de perdida disminuye a prácticamente 0, indicando así que la red se ha memorizado el conjunto de datos de entrenamiento y que para nuevos inputs no vistos no es del todo precisa. Para disminuir el efecto se decidió aplicar un *dropout* factor 0.4, este anula el 40 % de las neuronas de la capa.

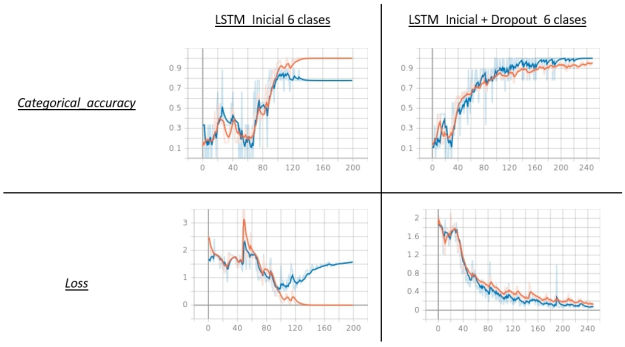

Fig. 13: Comparativa de la arquitectura inicial con 6 clases sin *dropout* y con *overfitting*, y con *dropout* y sin *overfitting*

6 Algoritmo de diezmado específico

Para poder adaptar la red neuronal para que entrene con vídeos pregrabados, lo primero es entender como esta funciona y como son generados los vectores de características que se introducirán en la red neuronal.

Las soluciones *MediaPipe Holistic* proporcionan diferentes *landmarks* para el cuerpo, cara, mano izquierda y mano derecha. Tanto cara y manos, representan cada *landmark* con un tupla de 3 posiciones, representando así las 3 dimensiones, alto y ancho respecto las dimensiones de la imagen, y profundidad. Es en el caso de la posición del cuerpo que a parte de estos 3 ya comentados se le añade un elemento más a esta tupla, la visibilidad, que indica la probabilidad de que el punto de referencia sea visible (presente y no ocluido) en la imagen.

Para la cara hay un total de 468*landmarks*, para cada mano 21 más y para la pose del cuerpo 33 más. Para poder introducir esta información en forma de vector, se convierte cada matriz en una unidimensional. Realizando los cálculos ((21+21+468) x 3 + 33 x 4), se obtiene un vector de características de 1662 posiciones.

La tupla que deseamos obten para introducir a la red neuronal tiene la forma de (x,y,z). Donde “x” hace referencia al número de vídeos, “y” al número de *frames* por vídeo y ’z’ hace referencia al número de características obtenidas en cada frame (1662).

El modelo LSTM va recibiendo como entradas los diferentes vídeos en formato de tupla (30,1662), donde 30 corresponde al número de *frames* de cada vídeo. En el programa original controlar el número de *frames* es sencillo ya que se realiza la captación del vídeo a través del codigo donde ya esta puesta la limitación de 30 *frames*. Pero, si se quiere entrenar la red con vídeos previamente grabados, donde no ha habido un control de los *frames*, necesitamos “diezmar” este vector de *frames* previamente para poderle entregar a la red neuronal para todos los vídeos el mismo número de *frames*.

6.1. Diseño del algoritmo

Para que el “diezmado” a 30 *frames* no represente una gran pérdida de información, se hará la suposición de que los vídeos con los que se trabajan no duran más de 5 segundos.

En este caso, las distintas señas de los diferentes lenguajes de señas no suelen durar más de ese período de tiempo. El número de *frames* es fijado a 30, todo y que gracias a las funciones que se mostraran a continuación se podría ajustar otro valor cualquiera.

6.1.1. Primera versión

Suponiendo un vídeo con 90 *frames*, la solución para rebajarlo a 30 es sencilla, se aplica un diezmado con factor 3 y ya obtendríamos nuestro nuevo vídeo con 30 *frames*. Sin embargo, no siempre se tendrá esta suerte, y hay mucha más probabilidad de que el número de *frames* que se tenga no sea múltiple de nuestro número de *frames* deseado (30).

Suponiendo el caso de un vídeo con 100 *frames*, este primer algoritmo (figura 14) eliminaría los 5 primeros *frames* y los 5 últimos para obtener un número múltiple de 30 y poder hacer ahora sí, el diezmado con factor 3. Por consiguiente, en este ejemplo, eliminar 5 *frames* seguidos por alante y otros 5 seguidos por detrás no suponen una pérdida excesiva de la información ya que se puede hacer la suposición de que cuando se graba un vídeo siempre hay un tiempo inicial hasta que se realiza la seña y un tiempo final después de finalizar la seña que no aportan mucha información más que los momentos previos y finales del gesto.

```
def roadTo30(arr, div):
    if div > 0:
        while div != 1:
            arr = np.delete(arr, np.arange(1,len(arr),div))
            div = div - 1
    return arr

def synthesis(arr, p, f):
    principio = False
    final = False
    if (len(arr)%30) != 0:

        if (f == True): #eliminado por el final
            new = np.delete(arr, 0) #eliminamos por el principio
            principio = True

        elif (p == True):  #eliminado por el principio
            new = np.delete(arr, len(arr) - 1) #eliminamos por el final
            final = True

        new = synthesis(new, principio, final)

    else:
        new = roadTo30(arr, int(len(arr)/30))

    return new
```

Fig. 14: Primera versión de la función recursiva de diezmado, llamada *synthesis()*

Sin embargo, haciendo la suposición de un escenario más extremo pero totalmente posible, se supone que ahora el vídeo es de 119 *frames*. En este caso, el múltiple más cercano es 120, pero según el algoritmo propuesto, no hay la posibilidad de doblar algún *frame*. Entonces, se tienen que eliminar 29 *frames* para alcanzar el anterior múltiple (90). Pero, eliminar 15 por delante y 14 por detrás si que se podría considerar una pérdida relevante de la información.

Todo indica que la solución es decidir cuando duplicar frames y cuando eliminar por los extremos, y el sentido común indica que el umbral debería de ser la mitad del valor al que se quiere reducir el vídeo (15), pero nunca se puede hacer la suposición de que ese gesto no empiece en el instante 0, o acabe en el instante t-1. Otra posible solución sería directamente siempre duplicar uniformemente los *frames* que faltan y después hacer el diezmado correspondiente, pero está no se ha considerado debido a la excesiva corrupción que se le realiza al vídeo.

6.1.2. Segunda versión y definitiva

Esta versión llamada *diezmadoRec()* (ver en la figura 16), esta basada en un algoritmo recurisvo que busca en cada iteración el número de *frames* óptimo a quitar, y si le quedan por eliminar, los elimina en la siguiente, y así hasta que se llega al número deseado.

El principio de funcionamiento viene descrito por el diagrama de flujo en el apéndice A.1 y en la figura 15 se puede ver en un formato más visual, además también se puede encontrar en el apéndice A.2 un fragmento de código donde estan definidas las funciones usadas en esta proceso de *data cleaning* en el cual se ve involucrada la función *diezmadoRec()* .

De forma resumida, la longitud del vector original de *frames* menos el número deseado de *frames* da como resultado una diferencia. De forma iterada y con el divisor empezando en 1, se realiza la división de la longitud del vector original entre el divisor. Si el cociente de esta división es mayor que la diferencia, previamente calculada, se aumenta el divisor. Cuando el cociente sea menor que la diferencia, guardamos los valores divisor y cociente. El cociente indicará cuantos *frames* se podran quitar del vídeo, y el divisor indica cada cuantos *frames* eliminamos uno.

Dicho de otra manera, se realiza una especie de diezmado. Con el ejemplo de la foto 15 la primera llamada a la función *diezmadoRec()* se detiene con cociente 17 y divisor 3. En la siguiente llamada a la función, el vector ya no es de 52 posiciones, ahora es de 52 - 17 = 35, y por lo tanto la diferencia entre la longitud del vector y el valor de *frames* deseado ahora pasa a ser 5. Se procede de la misma manera, y se llega a un valor de cociente 5 y divisor 7. Por lo tanto, en la segunda iteración cada 7 elementos, se eliminará 1.

7 LSA64 DATASET

A la hora de ampliar el conjunto de datos, se ha usado el LSA64: *an Argentinian sign language dataset* [21].

Se ha descargado la *raw version* que ocupa 1.9 GB. Esta carpeta contiene 64 clases diferentes y 50 vídeos por cada clase. Los vídeos no duran más de 4 segundos y han sido grabados con una Sony HDR-CX240 con la resolución de vídeo 1920x1080 a 60fps. El trípode estaba situado a 2 metros de distancia y a una altura de 1.5 metros, datos importantes puesto que las características de los *frames* viene dada en referencia al encuadre de los vídeos. Por otro, lado los sujetos utilizan guantes de colores llamativos para destacar las manos, pero como la red neuronal LSTM no procesa imágenes, este es un factor que no afecta.

Antes de trabajar con estos datos, estos se tuvieron que modelar, puesto que en la carpeta descargada estan todos los vídeos en bruto sin organizar por carpetas. Así pues, se tuvo que montar un diccionario a mano para que después se pueda acceder a las clases indexándolas (A.5). Más, montar el directorio nuevo donde se guardaran los vídeos y llenarlo con estos (A.6).

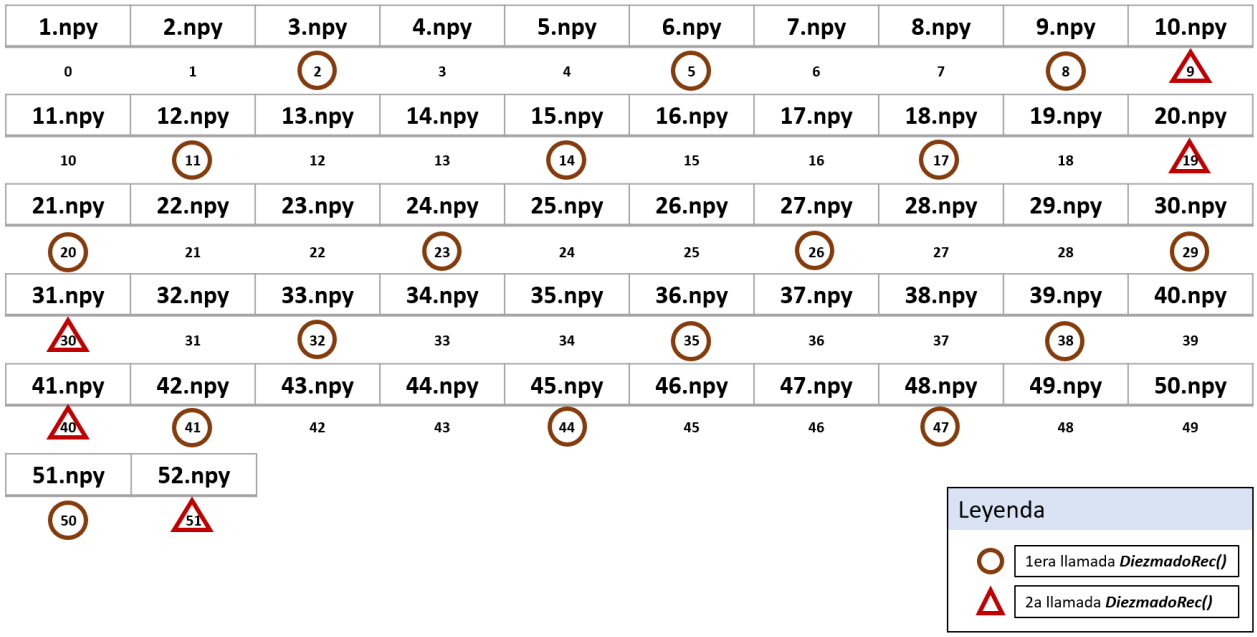

Fig. 15: Visualización gráfica del funcionamiento de la función *diezmadoRec()* cogiendo de ejemplo un vídeo de 52 *frames* convertido a un vídeo de 30.

```
def diezmadoRec(arr, valor):
    dividendo = len(arr)
    limite = dividendo - valor
    divisor = 1
    cociente = int(dividendo/divisor)
    while cociente > limite:
        divisor = divisor + 1
        cociente = int(dividendo/divisor)
    resta = limite - cociente
    arr = np.delete(arr, np.arange(divisor - 1,
                                   dividendo, divisor))
    if resta != 0:
        arr = diezmadoRec(arr, valor)
    return arr
```

```
#Testeo de la función diezmadoRec(arr,valor)
valor = 30
n_frames = []
for i in range(500):
  if i >= 30:
    arr = np.arange(0,i,1)
    res = diezmadoRec(arr,valor)
    n_frames.append(len(res))
n_frames = np.array(n_frames)
print(np.unique(n_frames))
```

```
[30]
```

Fig. 16: Versión final de la función de diezmado específico y un fragmento de código para su testeo.

Usando este conjunto de datos no es necesaria la verificación del número de *frames* de cada vídeo , sin embargo, para que el programa sea más general y la red más escalable, se realizaron una serie de funciones que permetian verificar el número de *frames* de los vídeos (A.6,A.7). De esta manera, si en otra situación se desea investigar con otras dimensiones de entrada en la red neuronal LSTM, se pueda verificar con anterioridad que todos los vídeos del conjunto de datos cumplen con el mínimo número de *frames*.

8 Sintonización de la red neuronal

La sintonización de una red neuronal consiste en encontrar los mejores parámetros de funcionamiento de esta. No obstante, previamente a la sintonización de la red neuronal, se ha tenido que modificar cierta parte del código y de la estructura para que la obtención de los datos, y la sintonización de esta sea lo más óptima posible.

8.1. Flujo de trabajo del progama final

8.1.1. Obtención de los *frames*

En el programa inicial, se recolectan los vídeos en directo y el procesado en ocasiones es más lento debido a que se dibujan los *landmarks* del sujeto para hacer el aplicativo más interactivo, sin embargo, estás funciones se podían evitar, haciendo el bucle mucho más óptimo. Adicionalmente, se tuvieron que realizar pequeñas modificaciones de código ya que ahora los *frames* venian límitados por el tamaño del vídeo. A medida que se iban leyendo los frames se iban guardando en el directorio correspondiente.

8.1.2. Diezmado específico de los vídeos

Una vez se encuentran todos los *frames* guardados , cada uno expresado como un archivo ".npy" tal y como se visualiza en la figura 15, se procede a llamar a la función *new_data_cleaning(RUTA,no_frames)* (A.2).

8.1.3. Creación y guardado de las tuplas

Después de realizar el "diezmado" previamente explicado, de igual manera a como se realizaba en el programa original, se cargan los archivos ".npy" en dos listas para formar las tuplas de entrada a la red.

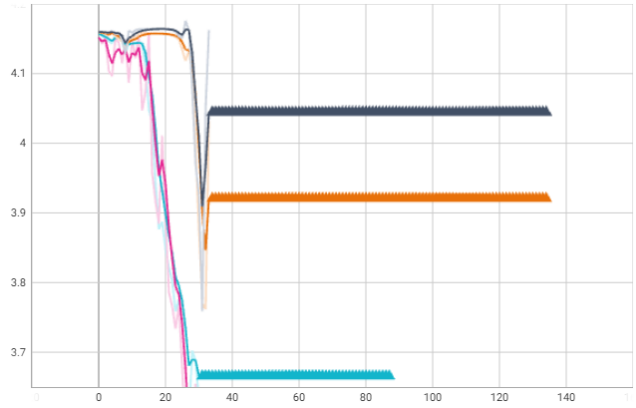

Fig. 17: Grafica de pérdida en 3 entrenamientos diferentes del modelo original (con la función de activación 'relu').

- En la primera lista se guarda por cada vídeo, el número de *frames* (número fijo previamente definido por el "diezmado"), y por cada *frame* sus características guardadas en el archivo ".npy". Por ejemplo, usando el conjunto de datos LSA64 y un número de *frames* por vídeo de 30, el array tendría la dimensión (3200,30,1662).
- En la segunda se guarda la codificación de las correspondiente clases. Siguiendo el ejemplo anterior, se creará un *array* de dimensión (3200,), donde cada el elemento será un valor comprendido entre 0 y 63.

Para no tener que repetir este proceso de creación de los *arrays* (dependiendo del entorno de ejecución puede llegar a tardar 8 horas), estos se guardan en el directorio de trabjo para que después, cuando el usuario se vuelva a conectar al cuaderno, simplemente tenga que cargar estos datos.

8.1.4. Construcción y entrenamiento del modelo

Con los *arrays* descargados, se separa un porcentaje para la validación, normalmente entre el 5 % y el 20 %. Se realiza la construción de la arquitectura del modelo de la red neuronal, y por último, se entrena el modelo.

8.2. Modelos

8.2.1. Original

Partiendo de la base del modelo propuesto se ha decidido probar una sintonización para el valor de 200 *epochs*. La principal idea era probar todas las combinaciones posibles variando entre un *dropout* de 0.4 o 0 (en las capas LSTM), función de activación 'ReLu' o 'Tanh' y optimizador, SGD o Adam.

Sin embargo, para la función de activación 'Relu' el modelo no convergia y se estancaba en un valor de pérdida de 4.1 (veáse la figura 17). Esto, seguramente es debido a que el conjunto de datos es demasiado grande y el modelo muy complejo para converger con la función de activación "relu". Este efecto, ya se ha podido notar con el código original cuando se han añadido simplemente 3 clases más (veáse la figura 13).

Por otro lado, con la función de activación 'tanh', todas las pruebas han convergido, dando resultados bastante coherentes (ver en la tabla 1). Además, se quiere dar incapié en la diferencia entre usar un optimizador u otro, por lo general 'Adam' debido a que mantiene el *learning rate* por parámetro y usa el momentum del gradiente hace que entrene más rápido que el optimizador 'SGD' ya que este tiene más ruido que el 'Adam' (veáse en la figura 18).

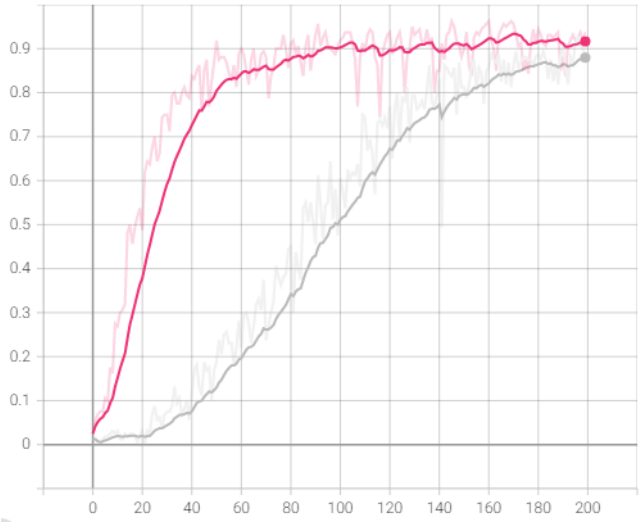

Fig. 18: Comparativa del entrenamiento a 200 *epochs* sin dropout del optimizador 'sgd' (gris) y 'adam' (rosa).

Los resultados en la tabla, son bastante robustos. Debido a que se trabaja con un conjunto de datos muy grande, el *dropout* no proporciona una mejora en el modelo puesto que no hay *overfitting* en el modelo. Así pues, para 200 *epochs* mientras que para el optmizador 'Adam' ya es suficiente para alcanzar un valor estable para el optimizador 'sgd' no es suficiente con 200 debido al lento aprendizaje. Tampoco necesita muchas más, con 50 epochs más se llegan a conseguir resultados muy parecidos. Por ejemplo, para un entreno de 250 *epochs* y haciendo un *stop early* en la 236 el resultado obtenido con el optimizador 'SGD' y sin *dropout* es de 91.87 % (veáse la figura 19). Aun así, se puede verificar que el "Adam" sigue siendo mejor optimizador en este tipo de redes neuronales.

Realizando la comparativa entre la resolución usada para entrenar los modelos, no se aprecia una una diferencia notable. Esto significa que los modelos aún pueden distinguir las características distintivas entre gestos, como pueden ser los dedos, los labios de la boca, etc. Esta resolución se podría ir disminuyendo para ver hasta donde es capaz la red neuronal de distinguir las características del conjunto de datos. Hay que tener en cuenta que esto también dependería del *dataset* con el que se trabajase, es por eso que sería interesante investigarlo con varios *datasets*.

8.2.2. Modelo simple (1 capa oculta)

Este modelo entrena sorprendentemente bien cuando se plantea con la función de activación "tanh", al contrario que con la función "relu" ya que el modelo es tan simple

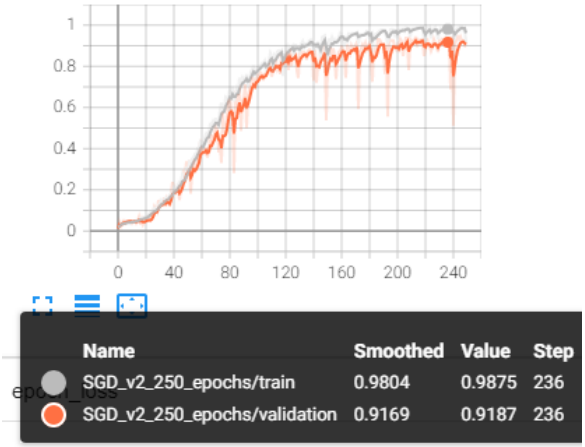

Fig. 19: Precisión del modelo original para 250 *epochs* con optimizador “SGD” y función de activación “tanh”.

que causa *underfitting* para el caso de hasta 300 epochs (ver la figura 20 para la gráfica y la figura 20 para la implementación).

```
model = Sequential()
model.add(LSTM(30,return_sequences=False,
               activation='relu', input_shape=(no_frames,1662)))
model.add(Dense(actions.shape[0],activation='softmax'))
```

Fig. 20: Modelo simple 1 capa. Las lineas azul y turquesa clarito son las correspondientes al conjuto de datos de validación.

Mientras que el modelo con la función de activación “tanh” con 250 *epochs* llega a un valor bastante correcto, alrededor del 90 % de *accuracy*, el mismo modelo con la función de activación “relu” padece de *underfitting* puesto que no es capaz de generalizar las características principales del conjunto de datos, adquiriendo como máximo en la *epoch* 110 aprox. del 58 %. La tendencia indica que por muchas más *epochs* que se añadan al entrenamiento el modelo nunca llegará a generalizar de forma correcta.

Por lo tanto, si se tuviera que elegir una función de activación para esta arquitectura, sin duda alguna se escogería la que usa la función de activación “tanh”, ya que obtiene mejores resultados.

8.2.3. Modelo complejo (3 capas ocultas)

Se plantea una arquitectrua con 3 capas ocultas, dos LSTM y una Densa, todas con un factor 0.4 de *dropout* en la segunda capa del LSTM, puesto que al estar en *return_sequences=False* esta devuelve el estado oculto de la última iteración, devolviendo un tensor de 2 dimensiones (ver en la figura 23). Con este *dropout* lo que se pretende es que la segunda capa LSTM no memorice las características generales del modelo para cuando se las tenga que pasar a la siguiente capa (código del modelo en la figura 22).

Fig. 21: Entrenamiento del modelo simple. En tonos fríos azulados, las gráficas del conjunto de datos de validación. En tonos cálidos, rojo y naranja, las gráficas del conjunto de datos de entrenamiento.

```
model = Sequential()
model.add(LSTM(30, return_sequences=True,
               activation='tanh', input_shape=(no_frames,1662)))
model.add(LSTM(30, return_sequences=False,
               dropout = 0.4, activation='tanh'))
model.add(Dense(30, activation='tanh'))
model.add(Dense(actions.shape[0], activation='softmax'))
```

Fig. 22: Modelo con 3 capas ocultas y 30 neuronas por cada capa oculta

Se realizará una primera prueba con 30 neuronas en cada capa, y después una segunda triplicando el número de neuronas. Se usará la función de activación “tanh” y el optimizador “Adam” puesto que son los que mejor resultado han dado en el modelo original.

La diferencia en cuanto a entrenamiento no es muy destacable. Lo destacable es que para el modelo de 30 neuronas/capa y usando la función de activación “tanh” en todas las capas (la Densa también, puesto que en todos los otros entrenamientos solo se cambiaban las funciones de activación de las capas LSTM. Ver en la figura 22), si se aplica la técnica de *Early stopping* comentada anteriormente se puede conseguir una precisión en el test de validación del 97.50 %.

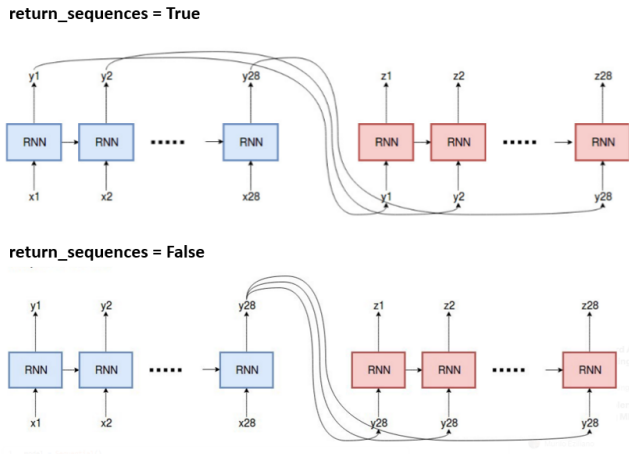

Fig. 23: Comparativa del parámetro encargado de devolver (return_sequences = True) o no (return_sequences = False) el estado oculto de cada *bacth*. Devolviendo o un tensor de 3 dimensiones o de 2, respectivamente [19].

TABLA 1: SINTONIZACIÓN DEL MODELO ORIGINAL

	Dropout	Optimizador	Accuracy(%)
Original	Sí	SGD	89.37
	No	SGD	90
	Sí	Adam	90.63
	No	Adam	**93.13**
Redim.	Sí	SGD	85.63
	No	SGD	88.75
	Sí	Adam	93.75
	No	Adam	**95**

9 RESULTADOS OBTENIDOS

Después de analizar con detalle los resultados de las sintonizaciones se destacan ciertas observaciones.

En primer lugar, el mejor optimizador, tal y como ya se suponía es el mismo que el autor propone en el modelo original, el optimizador "Adam". Este ha dado mejores resultados en todas las ejecuciones realizadas.

Por otro lado, si hay que destacar una función de activación, esta es la "tanh", puesto que modela y generaliza correctamente modelos que la "relu" no puedo.

El resultado más destacable se extrae mirando la tabla 1, donde se puede apreciar la poca diferencia que hay entre entrenar con el conjunto de datos original y el conjunto de datos redimensionado. Ya que usando correctamente la técnica de *early stopping* se llega a a alcanzar la misma *accuracy*.

Por último, se procederá a explicar una técnica que ha sido usada durante todas las ejecuciones y que a la hora de analizar con detalle los modelos, es muy útil para encontrar cuales son las clases en las que el modelo tiene más problemas para predecir.

9.1. Matriz de confusión

La matriz de confusión permite saber si el modelo predice de forma correcta de una forma muy visual. Además, permite detectar para que clases el modelo tiene más dificultades a detectar.

Esta matriz de dimensión (numero_clases,numero_clases) contiene en el eje "X" la predicción de la clase, y en el eje "Y" la verdadera clase.

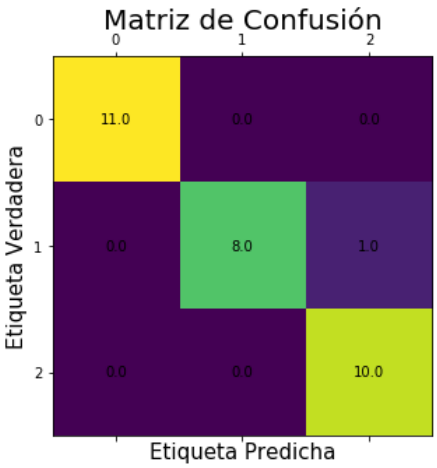

Fig. 24: Ejemplo de matriz de confusión para 3 clases

En la práctica, a la hora de analizar la matriz, conviene tener todos los valores en la diagonal identidad de la matríz. A continuación, se evaluará la matriz de confusión para los dos caso del modelo simple.

Usando la función de activación "tanh" se obtenie la figura 25 , donde se observa que todos los valores estan en la diagonal identidad. Para obtener el 100 % de *accuracy* la suma de los valores de la diagonal identidad debería de ser igual al número de muestras que se han predicho.

```
accuracy_score(ytrue, yhat)
```

```
0.91875
```

```
confusion_matrix(ytrue, yhat)
```

```
array([[1, 0, 0, ..., 0, 0, 0],
       [0, 4, 0, ..., 0, 0, 0],
       [0, 0, 2, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 2, 0, 0],
       [0, 0, 0, ..., 0, 2, 0],
       [0, 0, 0, ..., 0, 0, 4]])
```

Fig. 25: Matriz de confusión de las 5 primeras clases y *accuracy score* para el modelo simple entrenado durante 300 *epochs* para la función de activación "tanh"

No ocurre lo mismo con la función de activación "relu", donde no se puede ver bien exactamente donde estan algunos valores, debido a la dimensión de la matriz, pero

es suficiente para observar que se encuentran muy pocos valores en la diagonal identidad de la matriz. De hecho, se puede observar que el modelo confunde bastantes clases con la que se encuentra en la penúltima columna.

En una situación hipotética no tan drástica donde el *accuracy* se encontrase en el 80 %, se podría usar la matriz de confusión para analizar cuales son las clases que peor generaliza el modelo, puediendo actuar justo en el foco del problema.

```
accuracy_score(ytrue, yhat)
```

```
0.01875
```

```
confusion_matrix(ytrue, yhat)
```

```
array([[0, 0, 0, ..., 0, 2, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 3, 0],
       [0, 0, 0, ..., 0, 3, 0]])
```

Fig. 26: Matriz de confusión de las 5 primeras clases y *accuracy score* para el modelo simple entrenado durante 300 *epochs* para la función de activación “relu”

9.2. Modelo final

La mejor arquitectura encontrada durante la realizaciñon del proyecto es la reflectada en la figura 29, y es la comentada en la sección anterior (modelo complejo con 3 capas ocultas), exactamente la que semuestra en la figura 22.

Comparando ambas arquitecturas, la original tiene 2.8 veces más parámetros totales (veáse la figura 28). Esto implica que la arquitectura final tiene menos pesos que actualizar y por lo tanto una mayor velocidad de entrenamiento.

Como no tenía sentido comparar con el modelo original, puesto que este usaba la función de activación “relu” y se ha visto que no convergía, se ha cambiado la función de activación por la “tanh” para realizar la comparativa.

Los dos modelos han sido entrenados durante 200 *epochs* y usando el optimizado “Adam”. En la tabla 2 se encuentra la comparativa. Se consigue un *speed-up* del x1.94 en el proceso de entrenamiento y se consigue aumentar ligeramente la *accuracy*. En la figura 27 se aprecia como el modelo final entrena de forma más estable y como el conjunto de datos de validación consigue disminuir ligeramente más respecto el original. La ligera inestablidad del modelo inicial es debido al ruido producido por las excesivas neuronas por cada capa. Esto, podría solucionarse o añadiendo *dropout* al modelo o, tal y como se ha hecho en el modelo indicial, reducir el número de neuronas por capa reduciendo la complejidad del modelo.

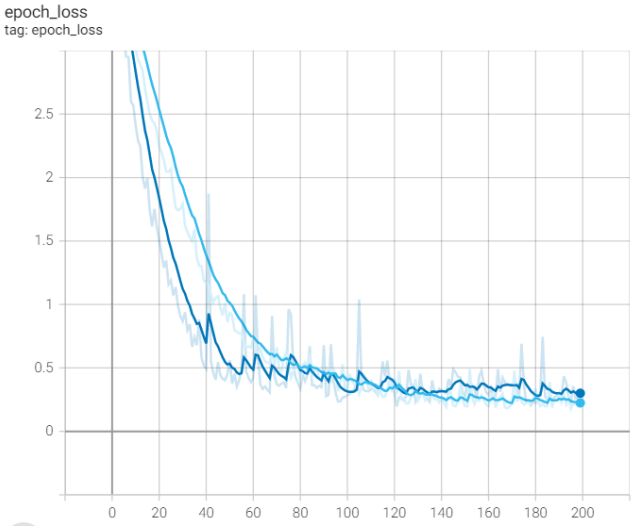

Fig. 27: Comparativa de la perdida entre el modelo original y el modelo final. La gráfica azul clarito corresponde al modelo final, y la azul más fuerte al modelo original

TABLA 2: COMPARATIVA DE LOS MODELOS ORIGINAL Y FINAL

	Tiempo (s)	**Accuracy (%)**
ORIGINAL	414	94.38
FINAL	**213**	**95.63**

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_11 (LSTM)              (None, 30, 64)            442112

 lstm_12 (LSTM)              (None, 30, 128)           98816

 lstm_13 (LSTM)              (None, 64)                49408

 dense_11 (Dense)            (None, 64)                4160

 dense_12 (Dense)            (None, 32)                2080

 dense_13 (Dense)            (None, 64)                2112

=================================================================
Total params: 598,688
Trainable params: 598,688
Non-trainable params: 0
_________________________________________________________________
```

Fig. 28: Arquitectura del modelo original

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_13 (LSTM)              (None, 30, 30)            203160

 lstm_14 (LSTM)              (None, 30)                7320

 dense_13 (Dense)            (None, 30)                930

 dense_14 (Dense)            (None, 64)                1984

=================================================================
Total params: 213,394
Trainable params: 213,394
Non-trainable params: 0
_________________________________________________________________
```

Fig. 29: Arquitectura del modelo final

10 AMPLIACIONES

Habiendo realizado este estudio sobre el procesamiento de vídeo para el reconocimiento de lenguaje de señas, se podría continuar con la búsqueda de un mejor modelo probando con diferentes *textit*, para así encontrar el modelo más robusto.

Por otro lado, el traspaso de esta herramienta a una applicación, podría resultar interesante. Obviamente, no se buscaría tener un modelo que prediga el lenguaje de signos entero. Pero si un modelo que sea capaz de predecir y traducir al instante al usuario, que desconoce el lenguaje, de las señas más usadas, básicas y fundamentales para poder comunicarse en alguna situación urgente.

Por último, sería interesante realizar un estudio sobre como afecta el distintio número de *frames* en la entrada de la red neuronal y probar de trabajar con vídeos más largos, quizás sin necesidad de estar enfocados en el lenguaje de señas, pero si para explorar , probar y mejorar esta técnica de diezmado específico.

11 CONCLUSIONES

Trabajar con redes neuronales es todo un reto. Muchas veces la forma correcta de encontrar la solución no viene dada por funciones matemáticas, sino que son a base de prueba y error. Así es como después de realizar muchas pruebas, se han podido coger unos cuantos ejemplos para explicar el funcionamiento de las distintas arquitecturas.

Se ha conseguido implementar un SW escalable que permita el trabajo con redes neuronales del tipo LSTM. Además, gracias a la técnica aplicada del diezmado específico sobre los vídeos originales, se necesita mucha menos memoria para trabajar.

Gracias al trabajo conseguido, se ha tenido la oportunidad de trabajar con un problema real que afecta a una cierta minoria de la población, para el cual se han usado conocimientos adquiridos durante la carrera para llevar a cabo desarrollo. Un aspecto motivador que ha ayudado mantener las ganas y la energia en este proyecto.

AGRADECIMIENTOS

Primero de todo, quiero agradecer el apoyo otorgado por mi familia y amigos que en todo momento y a su manera, me han brindado y facilitado la ayuda que he necesitado.

Por otro lado, quiero dedicar esta tesis a todas las personas que forman parte de la minoria de sordo/mudos. Sois todos/todas unos luchadores/luchadoras.

También, agradecer a mi tutor Jose Lopez Vicario, por confiar en mi desde el inicio y darme tanta libertad a la hora de realizar este trabajo, ya que todo y los problemas que hubo que en la primera propuesta de trabajo se ha seguido adelante hasta conseguirlo.

Por último, quiero dedicar en especial este trabajo a mi tío, Justo Piferrer Rodríguez, el cual se encuentra luchando día y noche contra la ELA (Esclerosis Lateral amiotrófica). Recuerdo cuando me dijiste que la inteligencia artifical no existía, refiriéndose a que tal y como las películas la muestran esta nunca la vamos a presenciar. Pues no soy yo quien para afirmar o negar tal hecho, pero aquí te dedico una tesis sobre un tema relacionado con el Deep Learning, que este si que existe :'). Siempre has sido un referente para mí, muchas gracias de todo corazón.

REFERENCIAS

[1] "Día Internacional de las Lenguas de Señas — Naciones Unidas". United Nations. https://www.un.org/es/observances/sign-languages-day (accedido el 1 de agosto de 2022).

[2] "14 de junio: Día Nacional Lengua de Signos en España". ACCEDES. https://accedes.es/70-000-personas-usan-la-lengua-de-signos-en-espana/ (accedido el 1 de agosto de 2022).

[3] N. Renotte. "GitHub nicknochnack/ActionDetectionforSignLanguage: A practical implementation of sign language estimation using an LSTM NN built on TF Keras". GitHub. https://github.com/nicknochnack/ ActionDetectionforSignLanguage (accedido el 7 de junio de 2022).

[4] F. Alonso. Redes Neuronales y Deep Learning. Capítulo 2: La neurona". Future Space S.A. https://www.futurespace.es/redes-neuronales-y-deep-learning-capitulo-2-la-neurona/ (accedido el 1 de agosto de 2022).

[5] Dot CSV. ¿Qué es una Red Neuronal? Parte 1 : La Neurona — DotCSV. (19 de marzo de 2018). Accedido el 29 de agosto de 2022. [vídeo en línea]. Disponible: https://www.youtube.com/watch?v=MRIv2IwFTPg

[6] F. Alonso. "Redes Neuronales y Deep Learning — Descenso por gradiente". Future Space S.A. https://www.futurespace.es/redes-neuronales-y-deep-learning-descenso-por-gradiente/ (accedido el 1 de agosto de 2022).

[7] F. Alonso. "Redes Neuronales y Deep Learning — Backpropagation". Future Space S.A. https://www.futurespace.es/redes-neuronales-y-deep-learning-brackpropagation/ (accedido el 1 de agosto de 2022).

[8] L. Velasco. .ºptimizadores en redes neuronales profundas: un enfoque práctico". Medium. https://velascoluis.medium.com/optimizadores-en-redes-neuronales-profundas-un-enfoque-práctico-819b39a3eb5 (accedido el 3 de agosto de 2022).

[9] "What are Recurrent Neural Networks?ÏBM - Deutschland — IBM. https://www.ibm.com/cloud/learn/recurrent-neural-networks (accedido el 1 de agosto de 2022).

[10] "Keras LSTM Layer Explained for Beginners with Example - MLK - Machine Learning Knowledge". MLK - Machine Learning Knowledge. https://machinelearningknowledge.ai/keras-lstm-layer-explained-for-beginners-with-example/ (accedido el 1 de agosto de 2022).

[11] "Coding Ninjas". Coding Ninjas. https://www.codingninjas.com/codestudio/ library/backpropagation-through-time-rnn(accedido el 2 de agosto de 2022).

[12] R. Grosse, "Lecture 15: Exploding and Vanishing Gradients". Toronto.

[13] R. Canadas. "Redes neuronales recurrentes — Qué son las RNN". abdatum. https://abdatum.com/tecnologia/redes-neuronales-recurrentes (accedido el 2 de agosto de 2022).

[14] W. Zhu, C. Lan, J. Xing, Y. Li, L. Shen y X. Xie, "Co-occurrence Feature Learning for Skeleton based Action Recognition using Regularized Deep LSTM Networks".

[15] L. Ramirez, "Dynamic Sign Language Recognition combining Dynamic Images and Convolutional Neural Networks", 22 de septiembre de 2020, Trujillo, Perú.

[16] Redacción España. "Qué son regresión y clasificación en Machine Learning". Agencia B12 Tech4Business: Big Data + IA + Ventas. https://agenciab12.com/noticia/que-son-regresion-clasificacion-machine-learning (accedido el 2 de agosto de 2022).

[17] T. Wood. "Softmax Function". DeepAI. https://deepai.org/machine-learning-glossary-and-terms/softmax-layer (accedido el 2 de agosto de 2022).

[18] "Overfitting in Machine Learning: What It Is and How to Prevent It". EliteDataScience. https://elitedatascience.com/overfitting-in-machine-learning (accedido el 3 de agosto de 2022).

[19] R. Shinde. Ünderstanding sequential/TimeSeries data for LSTM…". Medium. https://medium.com/@raman.shinde15/understanding-sequential-timeseries-data-for-lstm-4da78021ecd7 (accedido el 18 de agosto de 2022).

[20] "Aplicación del dropout a la cuantificación de la incertidumbre en redes neuronales - Archivo Digital UPM". Archivo Digital UPM - Archivo Digital UPM. https://oa.upm.es/57875/ (accedido el 3 de agosto de 2022).

[21] F. Ronchetti, F. Quiroga, C. A. Estrebou, L. C. Lanzarini y A. Rosete, "LSA64: an Argentinian sign language dataset", La Plata, Argentina, 1 de enero de 2016.

[22] J. I. Barrios. "La matriz de confusión y sus métricas – Inteligencia Artificial –". https://www.juanbarrios.com/la-matriz-de-confusion-y-sus-metricas/ (accedido el 23 de agosto de 2022).

APÉNDICE

A.1. Diagrama de flujo - 1

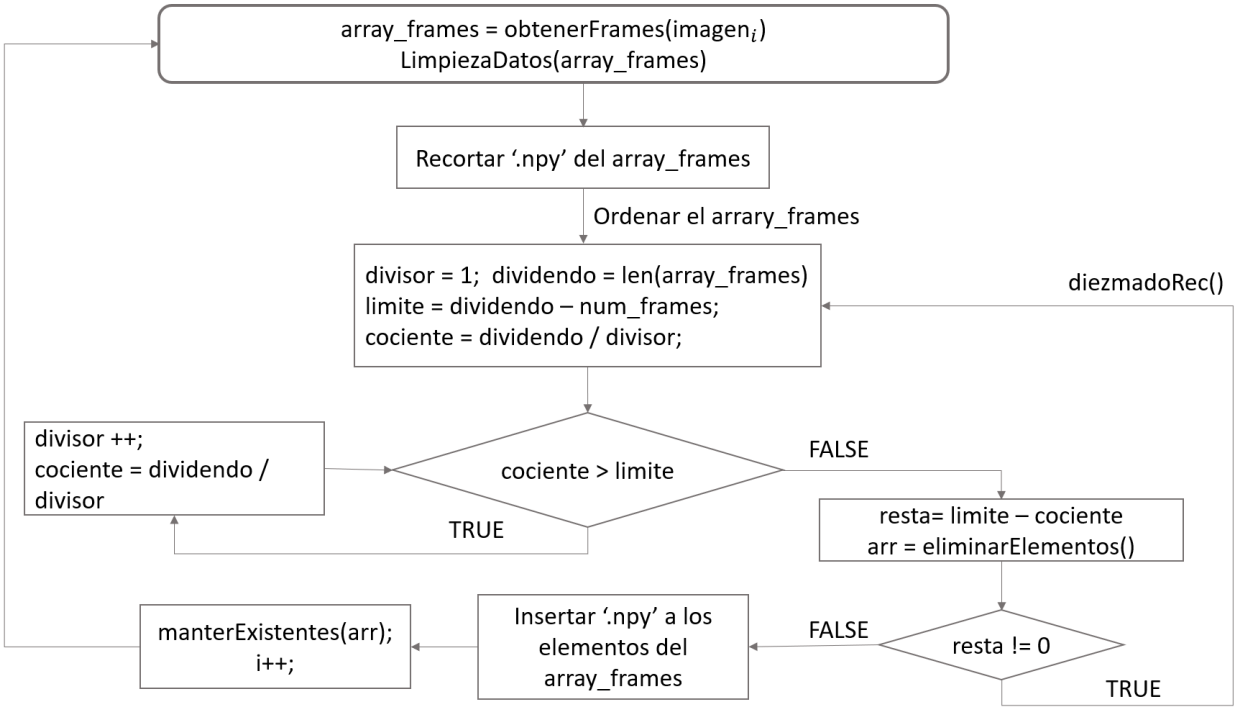

Fig. 30: Diagrama donde se aprecia el flujo del conjunto de llamadas necesarias incluida la de la función *diezmadoRec()* para conseguir el número de *frames* deseado

A.2. Fragmento de código - 1

```python
def recortarNPY(arr):
    new_arr = []
    for el in arr:
        cadena = el[:-4]
        new_arr.append(int(cadena))
    return new_arr

def insertNPY(arr):
    new_arr = []
    for el in arr:
        cadena = str(el)
        cadena = cadena + ".npy"
        new_arr.append(cadena)
    return new_arr

def preserve_existing(arr, path):
    old_files = os.listdir(path)
    for el in old_files:
        if el not in arr:
            os.remove(path + "/" + el)
```

```python
def diezmadoRec(arr, valor):
    dividendo = len(arr)
    limite = dividendo - valor
    divisor = 1
    cociente = int(dividendo/divisor)
    while cociente > limite:
        divisor = divisor + 1
        cociente = int(dividendo/divisor)
    resta = limite - cociente
    arr = np.delete(arr, np.arange(divisor - 1, dividendo, divisor))
    if resta != 0: #si la resta
        arr = diezmadoRec(arr, valor)
    return arr

def new_data_cleaning(DATA_PATH, no_sequences):
    #numero de frames por video que queremos tener
    valor = no_sequences
    labels = os.listdir(DATA_PATH)
    for label in labels:
        image_list = os.listdir(DATA_PATH + "/" + label)
        for image in image_list:
            files_list = os.listdir(DATA_PATH + "/" + label + "/" + image)
            if len(files_list) != 0:
                files_list = recortarNPY(files_list)
                files_list.sort()
                files_list_synthesis = diezmadoRec(files_list, valor)
                files_list_synthesis = insertNPY(files_list_synthesis)
                preserve_existing(files_list_synthesis,
                                  DATA_PATH + "/" + label + "/" + image)
```

Fig. 31: Definición de las funciones que permiten el diezmado específico

A.3. Esquema - 2

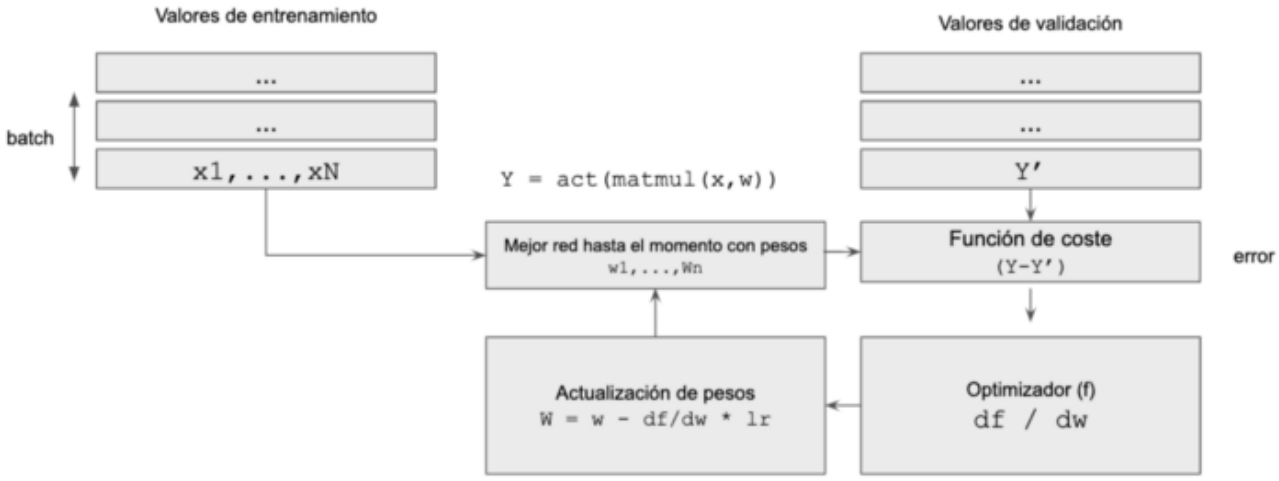

Fig. 32: Algoritmo de *backpropagation* para entrenamiento de redes neuronales

A.4. Diagrama de flujo - 2

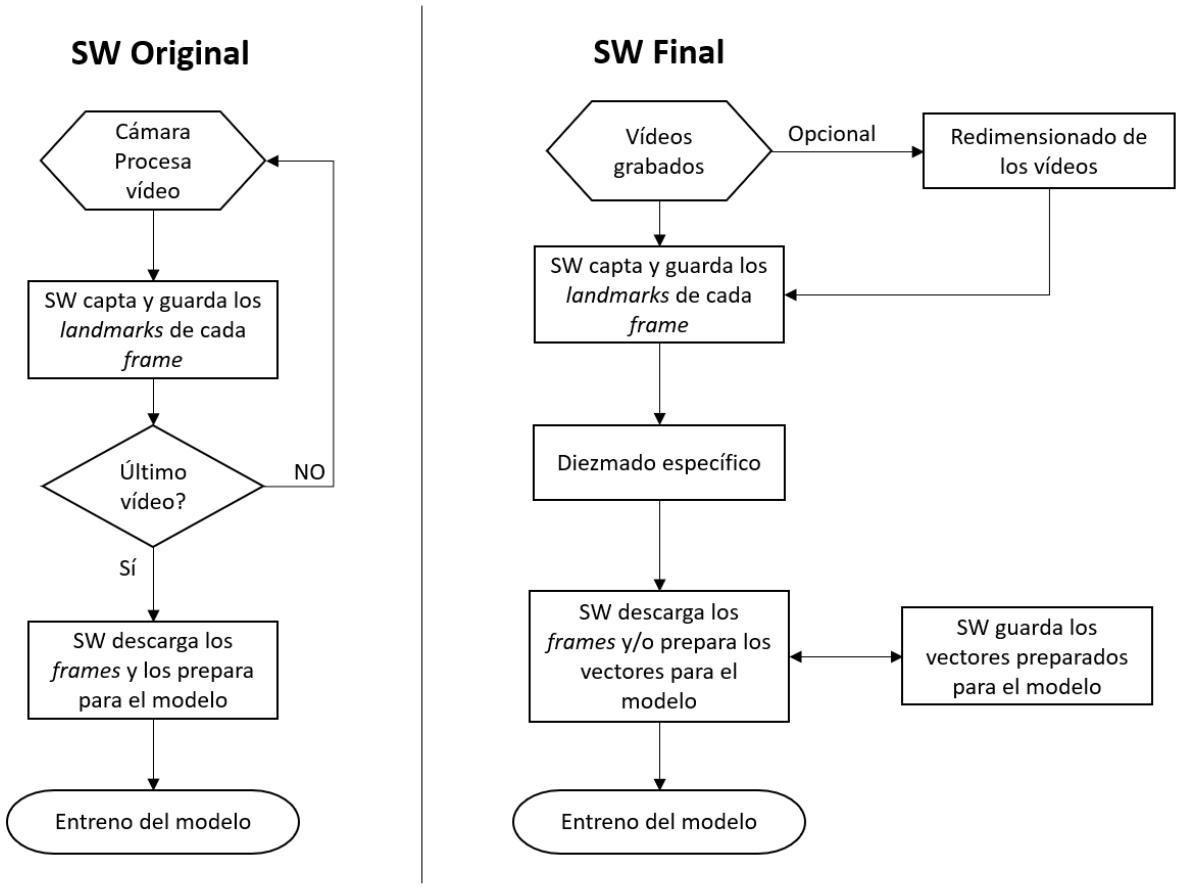

Fig. 33: Comparativa del diagrama de flujo del software inicial con el diagrama de flujo del software final

A.5. Fragmento de código - 2

```
diccionario = dict( Opaque=1, Red=2, Green=3, Yellow=4, Bright=5, Lightblue=6, Colors=7, Pink=8, Women=9, Enemy=10, Son=11,
                    Man=12, Away=13, Drawer=14, Born=15, Learn=16, Call=17, Skimmer = 18, Bitter=19, Sweetmilk=20, Milk=21,
                    Water=22, Food=23, Argentina=24, Uruguay=25, Country=26, Lastname=27, Where=28, Mock=29, Birthday=30,
                    Breakfast=31, Photo=32, Hungry=33, Map= 34, Coin=35, Music=36, Ship=37, Noone=38, Name=39, Patience=40,
                    Perfume=41, Deaf=42, Trap=43, Rice=44, Barbecue=45, Candy=46, Chewinggum=47, Spaghetti=48,
                    Yogurt=49, Accept=50, Thanks=51, Shutdown= 52, Appear=53, Toland=54, Catch=55, Help=56, Dance=57, Bathe=58,
                    Buy=59, Copy=60, Run=61, Realize=62, Give=63, Find=64)
newlist = []
for key in diccionario.keys():
    newlist.append(key)
id = 1
newlist[id - 1] #accediendo al valor 0
```

```
'Opaque'
```

Fig. 34: Montar diccionario para poder indexar las clases

A.6. Fragmento de código - 3

1

```
lsa64Dataset = "ls64_DATASET"
for key in diccionario.keys():
    try:
        os.makedirs(os.path.join(lsa64Dataset, key))
    except:
        pass
```

2

```
PATH_ORIGEN = "lsa64_raw/all"
PATH_DESTINO = "ls64_DATASET"

video_list = os.listdir(PATH_ORIGEN)
print("Copiando dataset...")
for video in video_list:
    vId = int(video.split(sep='_')[0])
    label = newlist[vId - 1]
    shutil.copy(os.path.join(PATH_ORIGEN,video),
                os.path.join(PATH_DESTINO,label))
print("Dataset copiado")
```

3

```
min_num_frames = 30
PATH_ORIGEN = "ls64_DATASET"
PATH_DESTINO = "ls64_DATASET_REDIM"
vTD = getVideosToDel(PATH_DESTINO, min_num_frames)
videosToDel = vTD[0]
actions = vTD[1]
label_map = vTD[2]
```

```
checkVideosToDel(videosToDel, actions, label_map)
```

```
delVideos(videosToDel, actions, label_map)
```

Fig. 35: 1: montar el directorio, 2: llenar el directorio, 3: verificar los vídeos del directorio

A.7. Fragmento de código - 4

```python
def frameTest(path):
    videos = []
    for video in os.listdir(path):
        frames = []
        cap = cv2.VideoCapture(os.path.join(path,video))
        while(cap.isOpened()):
            ret, frame = cap.read()
            if ret == False:
                break
            frames.append(frame)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
        cap.release()
        videos.append([video,frames])
    cv2.destroyAllWindows()
    return videos
```

```python
def getNumFrames(arr):
    arr = np.array(arr)
    lengths = []
    for i in arr:
        lengths.append(len(i[1]))
    return np.array(lengths)


def getIndex(arr, min_val):
    indexs = []
    for i,j in enumerate(arr):
        if j < min_val:
            indexs.append(i)
    return indexs

def getVideos(arr, path, min_val):
    videos = []
    for i in arr:
        if len(i[1]) < min_val:
            videos.append(os.path.join(path,i[0]))
    return videos
```

```python
def getVideosToDel(VIDEO_PATH, min_num_frames):
    actions = os.listdir(VIDEO_PATH)
    label_map = {label:num for num, label in enumerate(actions)}
    videosToDel = []
    for i,j in enumerate(actions):
        fullPath = os.path.join(VIDEO_PATH, j)
        print("Procesando la clase:" + j)
        arr = frameTest(fullPath)
        videosToDel.append(getVideos(arr, fullPath, min_num_frames))
    return [videosToDel, actions, label_map]

def checkVideosToDel(videosToDel, actions, label_map):
    for action in actions:
        if len(videosToDel[label_map[action]]) == 0:
            print("NO HAY VIDEOS PARA ELIMINAR DE LA CLASE: " + action)
        else:
            print("De la clase "+ action + ": hay que eliminar " +
                  str(len(videosToDel[label_map[action]])) +
                  " video(s) y esta es la lista:")
            print("     ")
        for video in videosToDel[label_map[action]]:
            print(video)
        print("------------------------------------------------------------")
    print("PARA ELIMINAR LOS VIDEOS LLAMAR A LA FUNCION: delVideos(videosToDel)")

def delVideos(videosToDel, actions, label_map):
    for action in actions:
        for video in videosToDel[label_map[action]]:
            os.remove(video)
            print("El video:" + video + " ha sido eliminado correctamente.")
```

Fig. 36: Definiciones que permiten realizar la validación de los datos