Escuela
Politécnica
Superior

Reconocimiento de la lengua de signos española mediante Redes Neuronales Profundas

Grado en Ingeniería en Sonido e Imagen en Telecomunicación

Trabajo Fin de Grado

Autor:
Laura Múgica Ramón
Tutor/es:
Jorge Calvo Zaragoza
José Javier Valero Mas

Universitat d'Alacant
Universidad de Alicante

Diciembre 2023

Reconocimiento de la lengua de signos española mediante Redes Neuronales Profundas

Autor

Laura Múgica Ramón

Tutor/es

Jorge Calvo Zaragoza

Departamento de Lenguajes y Sistemas Informáticos

José Javier Valero Mas

Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería en Sonido e Imagen en Telecomunicación

Escuela Politécnica Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Diciembre 2023

Resumen

En el mundo, alrededor de 70 millones de personas de la comunidad sorda utilizan el lenguaje de signos como medio de comunicación principal. Para posibilitar la comunicación con los no usuarios de esta lengua, se requiere de un intermediario: un intérprete. Sin embargo, este servicio esta disponible únicamente en circunstancias especiales como interacciones con la policía, ayuda en trámites administrativos, etc.

Este proyecto expone un estudio del estado de la cuestión en el ámbito del reconocimiento de lengua de signos con fin de conocer las diferentes opciones que existen, actualmente, para dar solución a la escasez de recursos que facilitan la comunicación entre personas sordas y no sordas.

Además, este Trabajo Fin de Grado propone un sistema automático de reconocimiento de signos para la Lengua de Signos Española mediante el uso de Redes Neuronales Recurrentes entrenadas a partir de los datos obtenidos de capturar un conjunto de gestos distintos mediante un sensor Leap Motion.

Abstract

70 million from the deaf world population mainly use sign language to communicate. In order to stablish communication with non-users, a signer is required. Nevertheless, this specific service is only available in certain burocratic processes such as interactions with the police, administrtaive procedures, etc.

This proyect reviews the existing literature about different implementations in terms of sign language recognition with the aim of getting an idea about the current status of the issue.

Furthermore, an automatic sign recognition system for Spanish Sign Language has been developed based on Recurrent Neuronal Networks. The implemented model has been trained with a database consisting on diferent isolated gesture caught by a Leap Motion sensor.

Agradecimientos

En primer lugar, gracias a mi madre por apostar siempre por mi y apoyarme en todo lo que hago. También a mi hermana, por demostrarme que siempre, da igual qué pase, puedo contar con ella. Sin ellas no sería la persona que soy ahora.

En segundo lugar, este proyecto no habría sido posible sin la confianza que mis tutores, Jorge Calvo y José Javier Valero, depositaron en mi desde el momento en el que contacté con ellos. Gracias por guiarme y dar forma a la idea inicial que os presenté, y por enseñarme y guiarme hasta conseguir desarrollar este trabajo.

Por último, gracias a todos los amigos que me llevo de estos cuatro años. Con vosotros he crecido y he disfrutado de esta etapa y, sinceramente, no creo que exista mejor compañía.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1. Introducción

Según datos de la Organización Mundial de la Salud (OMS), más del 5% de la población mundial sufre pérdida de audición discapacitante (OMS, 2021). Este dato supone un total de 430 millones de personas sordas que requieren de algún tipo de rehabilitación. De la población sorda mundial, más de un millón de personas son de nacionalidad española según datos del Instituto Nacional de Estadística (INE).

Las tecnologías auditivas como audífonos o implantes cocleares son las medidas más atractivas para la rehabilitación de esta discapacidad. Sin embargo, solo un 17% de todas las personas que podrían beneficiarse de utilizar audífono lleva uno (OMS, 2021). Por el contrario, las medidas de sustitución sensorial son mucho más populares en términos de comunicación con personas sordas. Entre estas medidas se encuentran: la lectura de los labios, el sistema de «deletrear» palabras en la palma de la mano, el método Tadoma (habla por vibración) o la comunicación por lenguaje de signos, siendo, esta última, una lengua oficial en España, entre otros muchos países europeos, desde octubre de 2007 con la publicación del Boletín Oficial del Estado la Ley 27/2007, de 23 de octubre.

Pese a que el lenguaje de signos es la forma más extendida para comunicarse con personas sordas, siguen existiendo dos barreras de comunicación muy relevantes. Por un lado, se requiere de un intérprete que traduzca de lenguaje de signos a habla o escrito (o viceversa) siempre que una de las partes comunicativas no conozca la lengua de signos. Además, no existe un lenguaje de signos internacional que facilite la comunicación entre miembros de la propia comunidad sorda.

El presente Trabajo Fin de Grado (TFG) parte de la idea de utilizar Deep Learning (aprendizaje profundo) para el reconocimiento automático de la Lengua de Signos Española (LSE) a partir de datos obtenidos mediante un sensor Leap Motion.

1.1. Objetivos

Este proyecto persigue dos objetivos principales relacionados directamente con la interpretación del lenguaje de signos:

- Conocer las soluciones actuales de reconocimiento automático del la lengua de signos.
- El desarrollo de una solución basada en redes neuronales con arquitectura Deep Learning para el reconocimiento automático de la Lengua de Signos Española.

Esta solución pretende servir de idea base para futuro desarrollo de herramientas que faciliten la comunicación entre la comunidad sorda y la comunidad no sorda.

La solución de reconocimiento automático se basará en aprendizaje profundo, concretamente, en redes del tipo Red Neuronal Recurrente (RNN). Para abordar el desarrollo completo de la solución se han establecido diferentes tareas:

- Preparación de los datos previa al desarrollo de la solución.
- Implementación del modelo RNN en lenguaje Python.
- Entrenamiento de la red y evaluación de los resultados del modelo.
- Análisis del rendimiento del modelo.

2. Estado del arte

Este capítulo pretende exponer, de manera resumida, diferentes soluciones que tratan de abordar el mismo problema que este TFG: el reconocimiento automático del lenguaje de signos.

En materia de reconocimiento automático, en este caso de gestos, existen dos partes claramente diferenciadas: la detección y seguimiento de los movimientos de manos (gestos) y la interpretación de los mismos. La primera de estas dos partes puede diferir de una solución a otra; algunas tecnologías utilizadas comúnmente para el desarrollo de estas soluciones son las tratadas en el apartado 2.1 de este capítulo. Por otro lado, una herramienta muy potente para la interpretación de dichos signos son los modelos basados en redes neuronales artificiales. Al igual que la solución implementada en este TFG, estos modelos presentan las características necesarias según los datos con los que trabaja.

En la literatura encontramos diferentes proyectos relacionados con este tema para diferentes idiomas y desde distintas perspectivas. En su mayoría, se trata de soluciones desarrolladas para lenguas de signos extranjeras como por ejemplo la lengua de signos americana (21% de las investigaciones), la lengua de signos india (16%), la lengua de signos árabe (13%), la lengua de signos china (12%) y la lengua de signo persa (4%); el 34% restante se centra en otras lenguas de signos como la brasileña, la alemana, la italiana y la española entre muchas otras (Wadhawan y Kumar, 2019).

2.1. Tecnologías para la captura de gestos

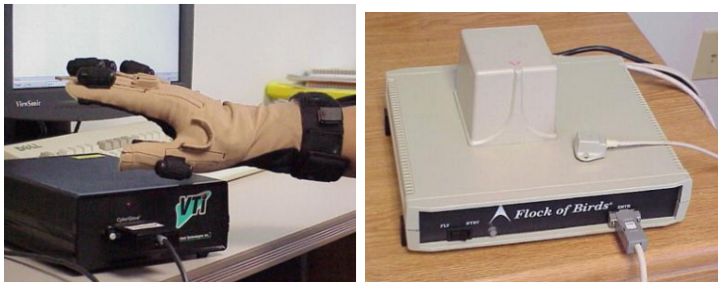
Uno de los principales problemas que nos encontramos a la hora de desarrollar soluciones del tipo reconocimiento automático está directamente relacionado con la obtención y preparación de los datos, sobre todo si estas soluciones se basan en modelos de aprendizaje automático.

A continuación, se exponen diferentes tecnologías que nos permiten lidiar con la parte de la adquisición de los datos, haciendo un análisis más profundo de la tecnología *Leap-Motion Controller* pues la base de datos usada en la implementación que propone este TFG ha sido recogida mediante este tipo de sensor. Además, se describen diversos proyectos que hacen uso de estas tecnologías en su implementación.

2.1.1. Guantes

Una de las opciones que permite la detección y seguimiento del movimiento de las manos son los guantes electrónicos con sensores incorporados capaces de detectar y transmitir información sobre la postura de la mano. La gran ventaja de esta tecnología radica en lo fácil que es extraer información sobre el grado de flexión de los dedos, su posición 3D y la orientación de la mano, lo que supone que se requiera de menos poder computacional a la hora de aplicar esta tecnologías a soluciones en tiempo real. Sin embargo, para obtener buenos resultados se precisan de mucha cantidad de sensores de buena calidad lo que se traduce en un encarecimiento del precio de esta tecnología.

La solución desarrollada en el trabajo **"American Sign Language word recognition with a sensory glove using artificial neural networks"** (Oz y Leu, 2011) utiliza dos dispositivos para llevar a cabo la adquisición de datos: el guante sensorial Cyberglove^{TM} equipado con 18 sensores para la detección de gestos y el rastreador de movimiento Flock of Birds para registrar la posición y orientación de la mano en el entorno tridimensional.



(a) Guante Cyberglove^{TM} con 18 sensores.

(b) Rastreador de movimiento Flock of Birds.

Figura 2.1: Equipos captación de gestos.

El objetivo principal de este proyecto es el reconocimiento continuo en tiempo real de la lengua de signos americana. La solución implementada para lograr este fin consiste en pasar los datos por una red de extracción de características y luego por una red de reconocimiento de palabras. Además, la salida decodificada era enviada a un modelo de síntesis del habla para reproducirla en forma de audio.

La red neuronal usada para la clasificación de los gestos consta de 151 variables de entrada (correspondientes a las características extraídas de cada signo en el anterior paso del proceso), una capa oculta con 100 neuronas y 50 neuronas en la capa de salida pues el proyecto fue entrenado y evaluado para un conjunto de 50 palabras distintas de la lengua de signos americana presentando una precisión cercana al 90% en la tarea de reconocimiento.

2.1.2. Visión artificial (*Computer vision*)

Un subcampo del Machine Learning que puede resultar muy interesante en tareas de reconocimiento de la lengua de signos es el de la Visión Artificial ya que incluye métodos para procesar, analizar y comprender imágenes digitales.

Una técnica basada en esta tecnología es el reconocimiento de objetos. La solución que propone el proyecto **"Dynamic Sign Language Recognition Based on Video Sequence With BLSTM-3D Residual Networks"** (Liao y cols., 2019) usa esta técnica como primer paso en su implementación. La solución que propone este trabajo para el reconocimiento dinámico del lenguaje de señas basado en secuencia de vídeo consta de tres fases:

1. Detección de la posición de la mano mediante el uso del modelo Faster R-CNN basado en redes convolucionales. Una vez entrenada la red con los frames de vídeo, las propuestas se asignan a una última capa que mapea las características del frame, se ajustan las dimensiones por medio de una capa ROI Pooling y un clasificador permite detectar correctamente la posición de las manos. Después de este proceso la información de las manos se puede segmentar del resto de la imagen.

2. Las imágenes generadas en el proceso anterior se introducen en un modelo B3D ResNet que permite extraer el vector de características para cada frame de la secuencia de vídeo.

3. La ultima fase sirve para crear una representación conjunta de estos vectores ya que analiza la dinámica temporal a largo plazo y predice la etiqueta del gesto de la mano.

Todo este proceso se resumen en el siguiente diagrama de bloques disponible en el artículo de dicho proyecto:

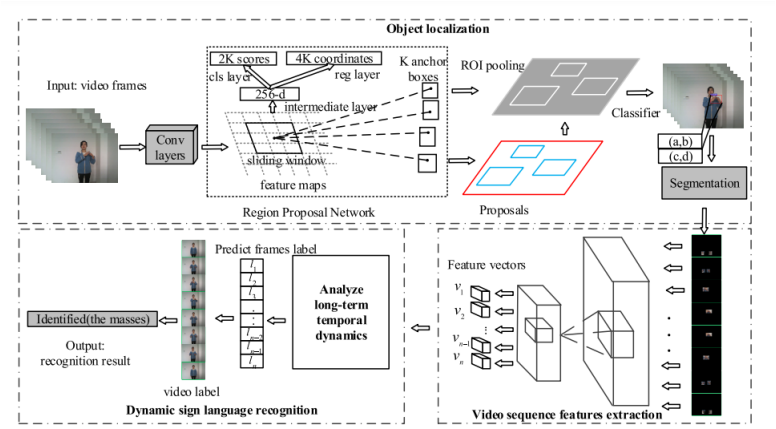


Figura 2.2: Diagrama de bloques de la solución propuesta en el proyecto de Liao y cols. (2019).

Para el proyecto se usaron dos datasets con secuencias de vídeo diferentes:

- El dataset DEVISIGN-D que contiene 500 gestos de la lengua de signo china realizados por 8 intérpretes diferentes resultando una base de datos de 6000 vídeos.

  Para este dataset se obtuvo una precisión del 89,8%.

- El dataset SLR_Dataset contiene 25 mil vídeos etiquetados, con más de 100 horas de vídeo en total en las que participa 50 intérpretes de la lengua de signos china diferentes.

  Para este dataset se obtuvo una precisión del 86,9%.

2.1.3. Sensor Kinect

El sensor Kinect es una cámara de profundidad desarrollada por Microsoft para la indsutria de los videojuegos con el fin de capturar los movimientos de los jugadores pero su aplicación se ha extendido al ámbito de la robótica. Además de en estos ámbitos, se han llevado a cabo diferentes proyectos de reconocimiento de la lengua de signos que usan este sensor para la captación de gestos como los desarrollados por Ren y cols. (2013) y Li (2012).

La solución que propone el trabajo **"Robust Part-Based Hand Gesture Recognition Using Kinect Sensor"** (Ren y cols., 2013) usa esta tecnología para capturar imágenes a color y su correspondiente mapa de profundidad de diferentes gestos. Su implementación consta de dos partes: la detección de la mano y el reconocimiento de gestos.

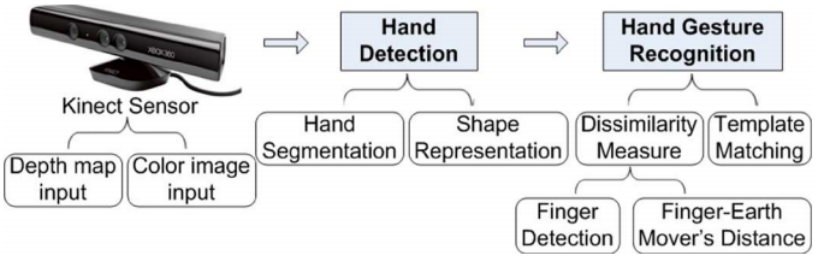


Figura 2.3: Diagrama de bloques de la solución propuesta en el proyecto de Ren y cols. (2013).

Para detectar la mano capturada por el sensor, este proyecto usa el software propio del sensor, también desarrollado por Microsoft para su sistema operativo Windows, Kinect SDK el cual permite segmentar la figura de la mano del fondo y, una vez realizado esto, la forma de la mano se representa como una curva de serie temporal.

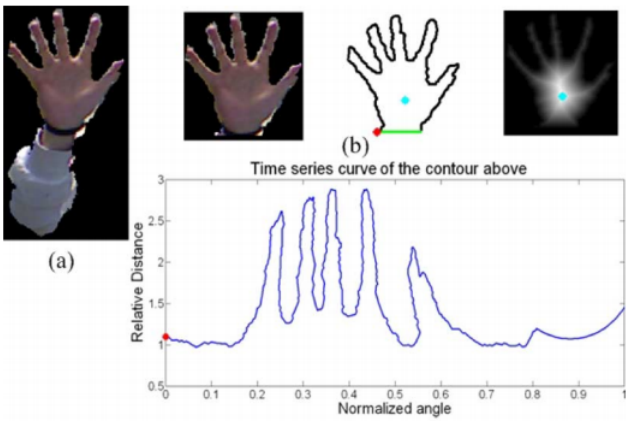


Figura 2.4: Resultado de detección de la mano de la solución de Ren y cols. (2013).

Para el reconocimiento de gestos, este proyecto no hace uso de RNA como los ya comentados, sino que desarrolla y utiliza un algoritmo conocido como *Finger Earth Mover's Distance* (FEMD) que consiste en la conversión directa de un contorno (resultado obtenido en el paso anterior de esta implementación) a un histograma que representa los dedos de la mano.

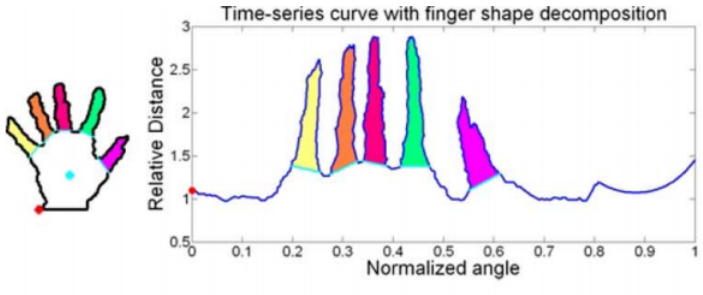


Figura 2.5: Resultado del algoritmo FEMD de Ren y cols..

A partir de este proceso, este trabajo construyó un dataset que consistía de en 10 gestos diferentes, realizados en 10 posturas diferentes por 10 intérpretes distintos lo que supone una base de datos de 1000 imágenes y sus correspondientes mapas de profundidad. La precisión en la tarea de reconocimiento de gestos a partir del algoritmo desarrollado en este trabajo fue de un 93,9%.

Posterior a este proyecto, el algoritmo FEMD que este propone se ha utilizado en otras solucione para abordar la tarea del reconocimiento de la legua de signos por ejemplo en **"A Novel Approach to Extract Hand Gesture Feature in Depth Images"** (Liu y cols., 2017).

2.1.4. Sensor Leap Motion Controller

El sensor Leap Motion Controller, desarrollado por Ultraleap, es un dispositvo de seguimiento óptico que captura el movimiento de manos y dedos para permitir al usuario una interacción natural con diferente contenido digital. Este dispositivo es capaz de detectar elementos dentro de una zona 3D interactiva de hasta 60 centímetros, extendiendo el campo de visión típico de 140x120º. El software que acompaña a este dispositivo distingue 27 elementos de la mano, incluidos huesos y articulaciones. El sensor está formado por:

- Dos cámaras infrarrojas de alta precisión que cuentan con tecnología CMOS sensible a la luz infrarroja y capaz de trabajar a una velocidad de hasta 200 fps.
- Tres LEDs para iluminar la zona de cobertura por inundación infrarroja trabajando en la misma longitud de onda que los sensores de las cámaras, esto es 850 nm.
- Un microcontrolador para recoger la información registrada vía USB al controlador que se esté usando en el ordenador conectado al sensor.

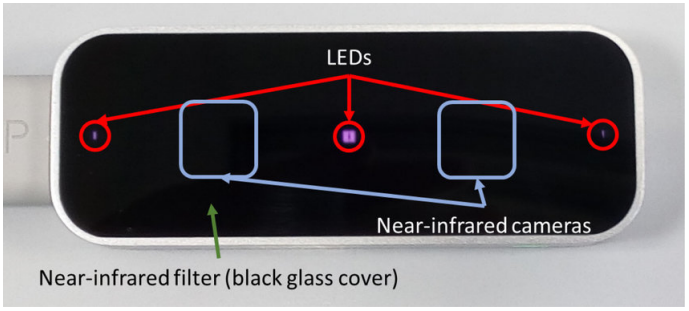


Figura 2.6: Partes del sensor Leap Motion Controller.

El principio de funcionamiento de este dispositivo se basa en que cuando un objeto es iluminado, en este caso con luz infrarroja, se produce una reflexión la cual se recoge mediante las cámaras del sensor. La información captada por los sensores de las cámaras se almacenan en una matriz que se envía vía USB al software especializado ser interpretada. Esta interpretación puede ir de los más simple como es la lectura y almacenamiento de los datos recibidos para su posterior uso hasta aplicaciones de alto nivel como interacciones con entorno digitales o modelos virtuales en 3D del movimiento de las manos.

Al tratarse de una tecnología diseñada exclusivamente para la captación de los movimientos de la mano, su aplicación resulta muy útil en la tarea de adquisición de datos para problemas de reconocimiento de la lengua de signos.

En la literatura podemos encontrar diversos proyectos que usan este sensor en su implementación. En este TFG se comentarán dos soluciones que difieren en la forma en la que se realiza la parte de reconocimiento o clasificación de los gestos.

La solución que propone **"Arabic Sign Language Recognition using Leap Motion Controller"** (Mohandes y cols., 2014) se basa en un dataset, registrado mediante este tipo de sensor, conformado por 28 signos del alfabeto árabe con 10 muestras de 10 matrices de datos para cada uno, suponiendo un total de 2800 matrices de datos recopilados. Estos datos se procesaron en MATLAB extrayendo, al final, 12 características principales de cada matriz. Para la clasificación de los signos se usaron dos métodos diferentes:

- Clasificado de Naïve Bayes o clasificación bayesiana: se trata de un clasificador probabilístico basado en el teorema de Bayes asumiendo características independientes. Con este método, la decisión se toma en función de la mayor probabilidad calculada que minimice el error de clasificación.

  El rendimiento que ofreció este método fue del 98,3%.

- Modelo basado en el perceptrón múltiple: la arquitectura de la red neuronal multicapa se diseñó usando la función sigmoide como función de activación de todas las neuronas que conformaban el modelo.

  El rendimiento que ofreció este método fue del 99,1%.

Otra solución que usa un modelo estadístico para el reconocimiento de gestos es la implementada en el proyecto **"Sign Language Gesture Recognition Using HMM"** (Parcheta y Martínez-Hinarejos, 2017) que, como su título indica, utiliza el modelo oculto de Márkov o HMM del inglés *Hidden Markov Model*, el cual se trata de un modelo estadístico cuyo objetivo es determinar parámetro desconocidos dentro de una secuencia a partir de los datos observables de la misma. Estos parámetros extraídos se pueden analizar, por ejemplo, en aplicaciones de reconocimiento de patrones. En este trabajo, la implementación se lleva a cabo mediante el *Hidden Markov Model Toolkit* (HTK), una librería de código abierto desarrollada en C que contiene módulos que permiten crear, manipular y entrenar modelos HMM. Los resultados de rendimiento en la tarea de reconocimiento de signos, usando el mismo dataset que utiliza este TFG, mediante esta técnica fue del 87,4%.

3. Marco Teórico

3.1. Redes Neuronales Artificiales

Una Red Neuronal Artificial (RNA) pretende imitar el comportamiento de las redes neuronales biológicas mediante un modelo matemático inspirado en las mismas. Una RNA, de forma similar al sistema neuronal humano, aprende, a partir de ejemplos previos, a generalizar nuevos ejemplos y abstraer características comunes, es decir, aprende de la experiencia.

3.1.1. Arquitectura básica

El elemento básico de una RNA es la perceptrón. Estas neuronas son unidades de cálculo que generan una salida como respuesta a un conjunto de datos de entrada. Estas unidades de cálculo se componen de:

- Datos de entrada x_i.
- Pesos sinápticos w_i.
- Umbral w_0.
- Regla de propagación de los datos de entrada combinados con sus correspondientes pesos $\sum_{i=1}^{n} w_i x_i$.
- Función de activación f que proporciona la salida de la neurona.

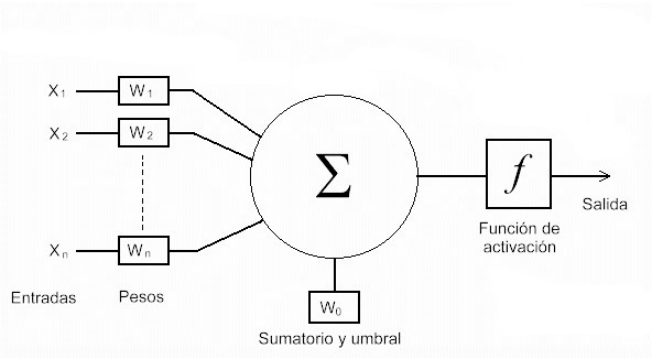


Figura 3.1: Esquema de una neurona artificial.

Para construir la red neuronal, las neuronas se agrupan en capas. Se distinguen tres niveles o tipos de capas:

- **Capa de entrada:** recibe los datos de entrada de la red.
- **Capas ocultas:** encargadas del procesamiento de los datos.
- **Capa de salida:** proporciona la información de salida de la red.

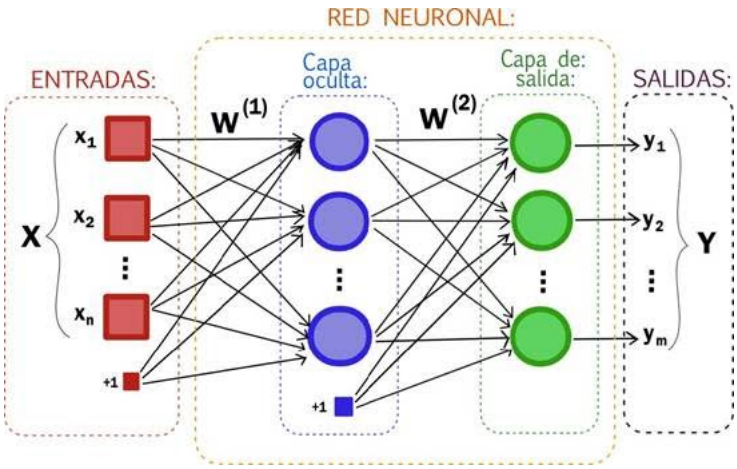


Figura 3.2: Arquitectura básica de una RNA.

Las neuronas que conforman cada capa reciben datos de entrada, los cuales son ponderados con su correspondiente peso sináptico, integrados y transmitidos a las neuronas de la siguiente capa. Cada una de las conexiones entre neuronas tienen asociadas un peso sináptico que guarda la mayor parte del conocimiento de la red.

Las RNA se clasifican según el algoritmo o método de aprendizaje que utilicen para adquirir y ajustar estos valores sinápticos. Siguiendo este criterio, se distinguen los siguientes tipos de redes neuronales:

- **Aprendizaje supervisado**: se dispone de un conjunto de datos etiquetados. Este tipo de arquitecturas se aplican, mayormente, en problemas de regresión y clasificación.
- **Aprendizaje no supervisado**: se dispone de un conjunto de datos no etiquetados. En este caso, la finalidad de la red es identificar la estructura de los datos y detectar relaciones entre sus variables.
- **Aprendizaje híbrido**: mezcla de capas que siguen aprendizaje supervisado y otras que tienen no supervisado.

- **Aprendizaje por refuerzo**: se trata de un método que se basa en determinar una serie de políticas para maximizar un objetivo o recompensa. No existen salidas correctas pero se indica a la red si ha acertado o no.

Una red neuronal puede estar formada por dos (perceptrón simple) o más capas (perceptrón múltiple) y el flujo de datos a través de la red puede ser unidireccional (*feedforward*) o retroalimentado (*feedback*).

Este TFG se centra en redes neuronales retroalimentadas pues es la característica distintiva de cualquier Red Neuronal Recurrente (RNN).

3.1.2. Perceptrón múltiple

Como ya se ha mencionado, un perceptrón múltiple o multicapa es una red neuronal formada por varias capas ocultas en la que el aprendizaje es supervisado. Este tipo de perceptrones siguen la arquitectura de la Figura 3.2. En un perceptrón multicapa, todas las neuronas de una capa están conectadas a todas las neuronas de la siguiente (se suele decir que es una red totalmente conectada). Sin embargo, también es posible encontrarse arquitecturas de red en las que ciertas neuronas no estén conectadas a las siguiente capa (es decir, los pesos sinápticos correspondientes son contantes e iguales a cero) o, por el contrario, redes en las que existan conexiones a capas posteriores o dentro de la misma capa (este es el caso de las RNN, las cuales se tratan en el apartado 3.3).

Mientras que el perceptrón simple (formado únicamente por una capa de entrada y otra de salida) no puede resolver problemas no lineales, el perceptrón múltiple puede aproximar relaciones no lineales entre datos de entrada y salida. Por esta razón, es una de las arquitecturas más utilizadas en problemas reales.

Para definir los parámetros de la red, las redes multicapa hacen uso de un algoritmo que consta de dos fases:

- **Propagación**: se calcula el resultado de la salida de la red desde los valores de entrada hacia delante.

- **Aprendizaje**: los errores obtenidos a la salida de la red se propagan hacia atrás (*Back-propagation*) con el fin de modificar los pesos sinápticos para que su valor estimado se asemeje cada vez más al real; esta aproximación se realiza mediante la función gradiente del error.

Estas dos fases forman un ciclo que se repite a lo largo de la red en cada entrenamiento de la misma. Este ciclo se puede dividir en tres procesos principales:

1. ***Forward propagation***.

   Cada neurona calcula la suma ponderada de todas las entradas de acuerdo a su peso correspondiente, pasa el resultado por su **función de activación** y el resultado pasa a la siguiente capa. Existen diferentes funciones de activación; la más conocidas son la sigmoide, tangente hiperbólica, ReLu y softmax; su uso depende del problema que estemos tratando.

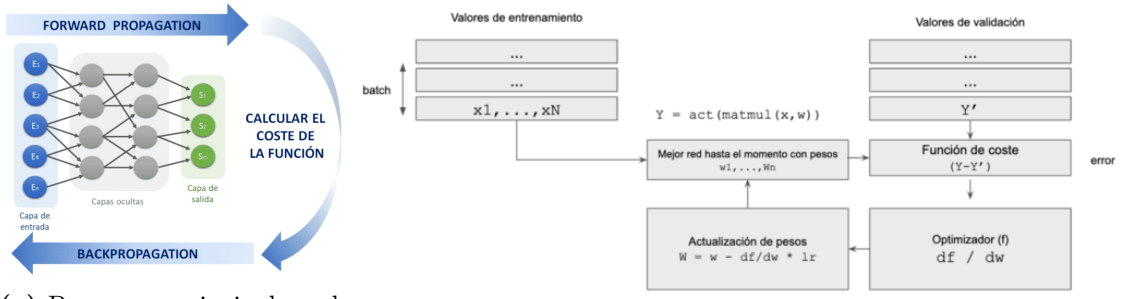
2. **Cálculo del coste de la función**.

   La **función de coste** se encarga de determinar el error entre el valor estimado y el valor real. Al igual que con las funciones de activación, existen diferentes funciones de coste y su uso depende del tipo de problema que se quiera abordar. Entre las más usadas se encuentran la raíz cuadrada media (RMSE), el error absoluto medio (MAE), entropía cruzada categórica (*Categorical Cross-Entropy*) y entropía cruzada binaria (*Binary Cross-Entropy*). Además, se usan **optimizadores** para minimizar los errores obtenidos. Su funcionamiento se basa en calcular el gradiente (derivada parcial) de la función de coste por cada parámetro de la red. Como el objetivo es minimizar el error, cada peso se modificará en la dirección (negativa) del gradiente obtenido. El conjunto de métodos para optimizar o reducir la función de error en búsqueda de un mínimo local son conocidos como métodos basados en el gradiente descendente. Entre los más usados se encuentran: descenso estocástico del gradiente (SGD, del inglés *Stochastic Gradient Descent*), Adam, Adagrad y Adadelta.

3. ***Backpropagation***.

   El algoritmo de *backpropagation* indica cuanto de culpa tiene cada neurona del error global de la red, es decir, pondera el reparto del error para todas las neuronas de la red. Para determinar el error, calcula las derivadas parciales de la función de coste con respecto a cada una de las variables. Este cálculo se realiza primero en la última capa y se propaga hacia atrás para ver cuanta culpa tienen el resto; de ahí el nombre del algoritmo.

La siguiente figura muestra el esquema de funcionamiento de una red neuronal:



(a) Procesos principales de funcionamiento

(b) Algoritmo de backpropagation

Figura 3.3: Esquema de funcionamiento de una RNA.

3.2. Deep Learning

El Deep Learning, o Aprendizaje Profundo, es un subconjunto de algoritmos de Aprendizaje Automático (Machine Learning) que intenta modelar abstracciones de alto nivel usando arquitecturas de red que admiten transformaciones no lineales.

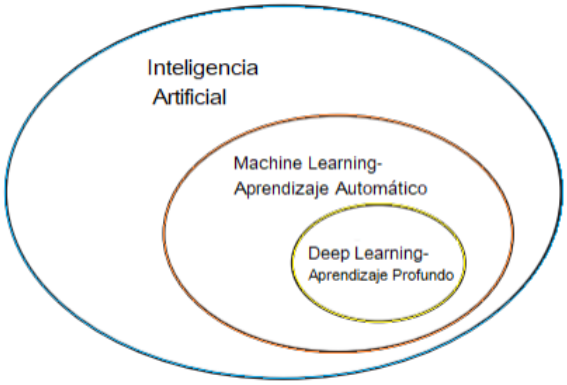


Figura 3.4: Campos de la inteligencia artificial.

Esta técnica pretende que los ordenadores aprendan, mediante el ejemplo, a hacer lo que sería natural para un ser humano. Por este motivo, es una herramienta muy útil para tareas de regresión y clasificación de todo tipo de datos (imágenes, texto, sonido..). Para obtener modelos que ofrezcan buenos resultados es necesario:

- Una **gran cantidad de datos etiquetados** que otorguen un buen rendimiento del entrenamiento de la red.
- Una **gran potencia de cálculo** que permita reducir el tiempo de entrenamiento de estas redes profundas las cuales están formadas por un gran número de capas ocultas en comparación con las RNA clásicas.

Estos modelos logran una gran precisión que, en ocasiones, incluso superan el rendimiento humano. Este factor es crucial para cierto tipo de aplicaciones que puedan poner en riesgo la seguridad del usuario como puede ser el caso de la tecnología de los vehículos sin conductor.

Para conseguir esta precisión se deben tener en cuenta tres factores claves:

- **Arquitectura de red**. Dado el gran número de capas ocultas que conforman la red, la captación de propiedades invariantes se ve notablemente mejorada en comparación con arquitectura convencionales.
- **Regularización**. Pese a que las redes profundas se entrenan con menos datos que números de parámetros, se puede prevenir el sobreajuste de la red haciendo uso de técnicas de regularización.
- **Optimización**. El algoritmo de *Backpropagation* ha evolucionado, para poder aproximar el gradiente sobre los conjuntos de datos enormes con los que trabajan los modelos de Deep Learning, hasta da lugar a los algoritmos de descenso por gradiente.

3.2.1. Tipos de arquitecturas Deep Learning

Aunque las arquitecturas de red existen desde los años 50, las nuevas arquitecturas y procesadores gráficos (GPUs) han potenciado considerablemente las RNA. Este auge de la inteligencia artificial, ha traído consigo, durante las últimas décadas, diversas arquitecturas de aprendizaje profundo capaces de atender mayor cantidad de problemas de diversos tipos.

En la siguiente figura se muestra la línea temporal que marca la aparición de dichas arquitecturas:

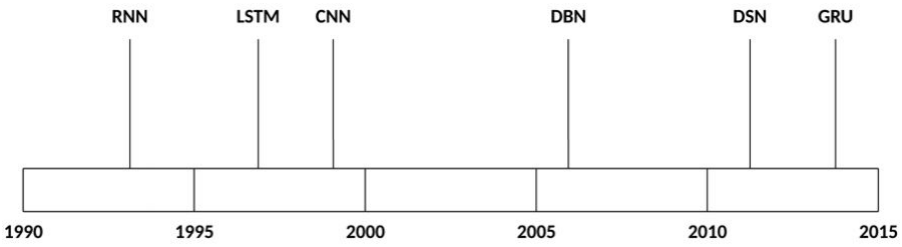


Figura 3.5: Evolución de las redes Deep Learning (Jones, 2017).

La siguiente tabla resumen expone las principales aplicaciones de estas arquitecturas:

Arquitectura	Aplicación
RNN	Reconocimiento de voz, reconocimiento de escritura a mano.
LSTM/GRU	Comprensión de textos de lenguajes naturales, reconocimiento de escritura a mano, reconocimiento de voz, reconocimiento de gestos, captura de imágenes.
CNN	Reconocimiento de imágenes, análisis de vídeos, procesamiento de lenguaje natural.
DBN	Reconocimiento de imágenes, recuperación de imágenes, compresión de lenguajes naturales, predicción de fallas.
DSN	Recuperación de información, reconocimiento continuo de la voz.

Tabla 3.1: Principales aplicaciones redes Deep Learning (Jones, 2017).

Este TFG se basa en RNN, en concreto en las de tipo LSTM. Ambas arquitecturas se desarrollan en el apartado 3.3.

3.2.2. Limitaciones del Deep Learning

Las técnicas de Deep Learning comentadas en el apartado anterior requieren, como cualquier modelo de aprendizaje profundo, de ordenadores lo suficientemente potentes como para trabajar con los grandes conjuntos de datos involucrados en su entrenamiento. Hoy en día, esto no supone ningún problema pues la tecnología actual es, de sobra, capaz de manejar la capacidad de cómputo requerida por estos modelos. Sin embargo, el Deep Learning se ve limitado por sus dos aspectos más problemáticos: el sobreaprendizaje y su carácter de "caja negra".

Se habla de sobreaprendizaje, también llamado *overfitting*, cuando un modelo se ajusta demasiado bien a los datos de entrenamiento y deja de generalizar correctamente cuando se prueba sobre datos de un conjunto de prueba diferente; en otras palabras, el modelo "memoriza" en lugar de "aprender". Este problema surge cuando el número de parámetros ajustables (pesos sinápticos) es muy elevado. Además, cualquier conjunto de datos contiene ruido. Este ruido puede ser de dos tipos: errores en los propios datos debido al proceso de adquisición de los mismos o de su pre-procesamiento, así como, errores de muestreo (por grande que sea el conjunto, no deja de ser una muestra de un conjunto mayor). Cualquier conjunto de entrenamiento contendrá tanto regularidades "reales" como "accidentales" y ambas serán modeladas y, si el modelo se ajusta demasiado a estos rasgos accidentales, perderá capacidad de generalización y caeremos en el sobreaprendizaje.

Para intentar minimizar el *overfitting* es recomendable:

- **Clases variadas y equilibradas** dentro del conjunto de entrenamiento para que este esté balanceado en cantidad de muestras de cada categoría o clase.
- Crear un **conjunto de datos de validación** que nos permita obtener una valoración real, durante el entrenamiento de la red, de la tasa de aciertos.
- Contar con una **cantidad adecuada de *features*** para la cantidad de muestras de entrenamiento de las que disponemos. Tener una cantidad excesiva de variantes sin suficientes muestras entorpece la tarea de generalización.
- **Arquitectura adecuada** ya que un exceso de capas ocultas puede conllevar el sobreprendizaje del modelo. Además, conviene experimentar con la cantidad de iteraciones durante el entrenamiento de la red para dar con un **buen ajuste de los parámetros del modelo**.

Como se ha comentado al principio de este punto, existen técnicas de regularización ideadas precisamente para evitar este sobreajuste de los modelos de aprendizaje profundo. Algunas de estas técnicas son:

- Incorporar un **término adicional en la función de coste** que se quiera minimizar, como puede ser el decaimiento de pesos (*weight decay*).
- Añadir **mecanismos de control** durante el entrenamiento. Un ejemplo de estos mecanismos es el *early stopping* que detiene el entrenamiento del modelo cuando la tasa de error del conjunto de datos de validación aumenta.
- Provocar la **anulación selectiva de partes de la red** para prevenir su coadaptación. Dentro de estas técnicas, una de la más conocida es la *Dropout* la cual consiste en desactivar aleatoriamente un porcentaje de neuronas de la red en cada iteración del entrenamiento.

3.3. Redes Neuronales Recurrentes

Las RNN son uno de los enfoque más antiguos en materia de arquitectura de red, pero también de los más usados hoy en día y las cuales sirven de base para construir otras arquitecturas de Deep Learning.

Al contrario de los perceptrones multicapa clásicos, en las RNN pueden realizarse conexiones a capas anteriores o, incluso, en la misma capa; esta retroalimentación otorga "memoria" a la red. Esta característica fundamental permiten analizar y trabajar con datos de series temporales permitiendo modelar problemas en la dimensión del "tiempo".

3.3.1. Neurona recurrente

Como ya se ha definido en el apartado 3.1.1, una neurona artificial se encarga de calcular la respuesta a unos datos de entrada y trasladar esta información a la siguiente capa. En las RNN, las neuronas incluyen conexiones "hacia atrás" creando retroalimentación entre las neuronas de la capa. Estas conexiones permiten al modelo retener información de pasado de manera que puedan crearse correlaciones entre eventos muy separados en el tiempo.

La siguiente figura muestra la estructura de una neurona recurrente así como la misma desplegada en el tiempo:

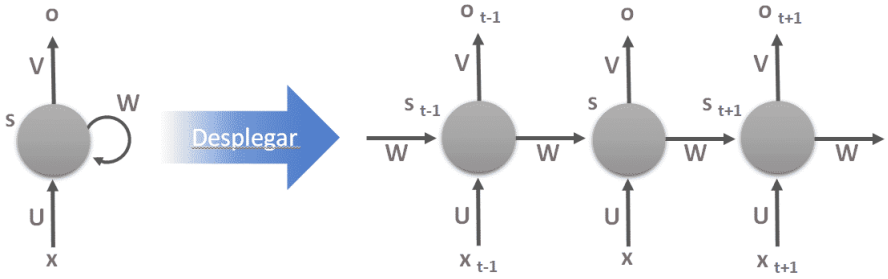


Figura 3.6: Estructura de una neurona recurrente.

Como explica la Figura 3.6, el principio de funcionamiento de una neurona recurrente se basa en que, para cada instante de tiempo, la neurona recibe la entrada de la capa anterior y su propia salida del instante de tiempo anterior para generar la salida del instante actual. Cada neurona recurrente tiene dos conjuntos de parámetros: uno para los datos recibidos de a capa anterior y otro para los datos generados en el instante anterior por la propia neurona.

$$o_t = f(Ux_t + Ws_{t-1} + b) \tag{3.1}$$

Se puede decir, entonces, que la salida de una neurona recurrente es una función de entradas de los instantes de tiempo anteriores. Este factor es el que otorga, en cierta forma, memoria a la red. La parte de la red neuronal que preserva un estado a través del tiempo se conoce como ***memory cell***.

Esta memoria interna de las RNN hace que sean la solución más adecuada para problemas que involucran datos secuenciales. Además, al ser capaces de recordar información relevante de entradas pasadas, son más precisas en las predicciones de lo que vendrá después sin perder información de contexto.

3.3.2. *Backpropagation* a través del tiempo

Como se ha mencionado en el apartado 3.1, durante el entrenamiento de una RNA, los pesos del modelo se ajustan mediante optimizadores del error basados en algoritmos de descenso por gradiente. Este cálculo se realiza desde la última capa hacia atrás (algoritmo de *backpropagation*). La estructura recurrente de este tipo de redes neuronales supone que no exista un punto final en el que la propagación hacia atrás del error pueda detenerse.

En el caso de las RNN, la solución que se aplica a este inconveniente se conoce como *Backpropagation* a través del tiempo (Backpropagation Through Time (BPTT)). Este proceso es complejo ya que la función de coste de un determinado instante de tiempo depende del instante de tiempo anterior. El algoritmo BPTT propaga el error hacia atrás desde el último hasta el primer instante de tiempo mientras se despliegan todos los instantes de tiempo de modo que se puede calcular la función de coste para cada instante y, de esta manera, actualizar los pesos de la red.

3.3.3. Inconvenientes de las RNN

Gracias a la arquitectura particular de este tipo de redes, las RNN permiten procesar entradas de cualquier longitud sin afectar al tamaño del modelo y los pesos del modelo se comparten a lo largo del tiempo.

Pese al potencial que ofrecen los modelos basados en RNN, estas son difíciles de entrenar ya que su computación es lenta lo que produce que acceder a información de estados muy antiguos resulte difícil. Para resolver este problema se diseñaron las redes Long Short-Term Memory (LSTM) (explicadas en profundidad en el apartado 3.3.4).

Además, existen otras dos cuestiones que afectan al rendimiento de una red recurrente:

- ***Vanishing gradient:*** se produce cuando los valores de gradientes son demasiado pequeños y el modelo deja de aprender (o requiere de mucho tiempo para obtener resultados decentes). Este problema se resuelve mediante el concepto *gate units*, este concepto da lugar a nuevas arquitecturas RNN las cuales se explican con más detenimiento en los apartados 3.3.4 y 3.3.5
- ***Exploding gradient:*** se produce cuando el algoritmo asigna una importancia demasiada alta a los pesos sin razón y esto genera un problema en su entrenamiento.

3.3.4. Redes LSTM (*Long-Short Term Memory*)

En términos generales, las redes Long Short-Term Memory (LSTM) son RNN que amplían su memoria para aprender de experiencias que han pasado hace muchos instantes de tiempo solventando así los problemas de memoria a corto plazo y de *vanishing gradient*.

De manera análoga a la memoria de un ordenador, una neurona LSTM puede leer, escribir y borrar la información que contiene en su memoria. Esta memoria se puede ver como una celda que decide si eliminar o almacenar información en función de la importancia (establecida a través de los pesos) asignada a la información que se recibe. En otras palabras, las LSTM aprenden con el tiempo que información es importante y cuál no. Para realizar esta tarea, cada neurona LSTM está formada por tres puertas (*gate units*):

- ***Input gate* (puerta de entrada):** decide si la nueva información que llega a la neurona puede entrar en la memoria.
- ***Forget gate* (puerta de olvidar):** decide si olvidar parte de la información discriminando entre datos importantes y superfluos con el fin de dejar sitio a los nuevos datos.
- ***Output gate* (puerta de salida):** decide si su estado interno pasa a ser el valor de salda o al estado oculto del siguiente instante de tiempo.

La siguiente figura muestra la estructura interna de una neurona LSTM:

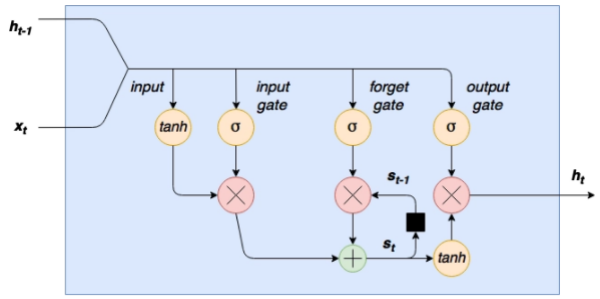


Figura 3.7: Estructura interna neurona LSTM.

Las puertas de una neurona LSTM son análogas a una función sigmoide (σ), es decir toman valores entre 0 y 1. Esto permite, desde el punto de vista matemático, incorporarlas al algoritmo BPTT.

Si analizamos el proceso descrito en la figura 3.7 vemos que a la entrada de la neurona llega la nueva información (x_t) combinada con la salida del instante de tiempo anterior de la propia celda (h_{t-1}). A partir del punto de entrada podemos distinguir los siguientes pasos:

1. La entrada pasa por la función de activación *tanh* para rectificar sus valores en el rango [-1, 1].

$$g = tanh(U^g x_t + W^g h_{t-1} + b^g) \tag{3.2}$$

   Donde U^g y W^g son los pesos de la entra y la salida anterior de la celda y b^g es el sesgo de la entrada (a modo de aclaración, se debe tener en cuenta que los exponentes no son potencias elevadas sino que indican que los pesos y valores de sesgo pertenecen a una puerta en concreto).

2. La entrada rectificada se multiplica, elemento a elemento (operador $\circ$), por la salida de la *input gate* la cual corresponde a un nodo sigmoide activado. En este punto, la salida de la puerta de entrada viene dada por:

$$g \circ i \tag{3.3}$$

3. Se calcula la *forget gate* como:

$$f = \sigma(U^f x_t + W^f h_{t-1} + b^f) \tag{3.4}$$

   De esta forma, la salida de esta etapa de la celda se calcula como el producto por elementos del estado anterior y la puerta de olvido más la salida de la puerta de entrada:

$$s_t = s_{t-1} \circ f + g \circ i \tag{3.5}$$

4. La *output gate* se calcula como:

$$o = \sigma(U^o x_t + W^o h_{t-1} + b^o) \tag{3.6}$$

5. Finalmente, la salida de la celda se calcula como:

$$h_t = tanh(s_t) \circ o \tag{3.7}$$

Todos estos parámetro se aprenden durante el entrenamiento de la red. Por este motivo, los modelos basados en arquitecturas LSTM son muy poderosos pues, partiendo de lo explicado, son capaces de aprender a memorizar dependencias durante mucho tiempo y a olvidar el pasado cuando sea necesario.

3.3.5. Redes GRU (*Gated Recurrent Unit*)

Otra conocida implementación de RNN son las redes Gate Recurrent Unit (GRU). Estas se basan en el mismo principio de *gate units* que las LSTM pero simplificado con el fin de ser mas eficientes computacionalmente. La mayor simplicidad de las GRU con respecto de las LSTM hacen que su entrenamiento sea más rápido y su ejecución más eficiente ya que involucra menos pesos.

Una célula GRU controla la información que recibe y almacena mediante dos puertas:

- ***Update gate* (puerta de actualización):** indica cuánta de la información anterior de la celda hay que mantener y transmitir al siguiente instante de tiempo.
- ***Reset gate* (puerta de reajuste):** define cómo incorporar los nuevos datos con los ya contenidos en la celda decidiendo la información que es irrelevante y, por tanto, debe olvidarse.

A grandes rasgos, una GRU funciona de manera muy similar a una LSTM pero prescindiendo de la *output gate*. Al carecer de una puerta de salida, las redes GRU tienen menos parámetros que ajustar durante su entrenamiento, haciendo de estas un modelo más rápido y eficiente que las LSTM como se ha comentado al principio de este punto. Por esta razón, los modelos basados en arquitecturas GRU resultan muy atractivos para tratar problemas que involucran conjuntos de datos más pequeños.

3.4. Aplicaciones de las RNN

Como se ha remarcado a lo largo del apartado 3.3, las redes recurrentes destacan por su capacidad de retener información y establecer correlaciones entre información separada en el tiempo. Además, existen arquitecturas específicas capaces de decidir sobre el flujo de información que datos son relevante y cuales pueden desecharse. Esta "memoria" hace de los modelos basados en RNN una solución muy adecuada para abarcar problemas basado en secuencias.

En esta parte de la memoria se exponen las diversas aplicaciones que tienen las RNN además de los distintos tipos de arquitecturas que existen según las necesidades del problema al que se apliquen.

3.4.1. Aplicaciones en modelos de secuencias

Según la RAE (Real Academia Española), una **secuencia** es una serie o sucesión de elementos que guardan entre sí cierta relación.

Siguiendo esta definición, un **modelo de secuencias** es una técnica utilizada cuando el orden y la secuencia de un conjunto de datos aportan valor predictivo.

Los modelos de secuencias son claves en el **Procesamiento del Lenguaje Natral**, en inglés Natural Language Processing (NLP), campo del aprendizaje automático que se enfoca en la comprensión del lenguaje humano en el cual la secuencia de palabras es muy importante (en otras palabras, el contexto tiene una carga predictiva muy importante). Dentro de este campo de estudio, las RNN se han aplicado en diversas tareas, entre ellas cabe destacar:

- Análisis y comprensión del lenguaje.
- Reconocimiento del habla.
- Codificación del habla.
- Traducción automática.
- Generación de texto.
- Clasificación de textos.
- Detección de entidades.
- Análisis de sentimientos.
- Descripción de imágenes (*image captioning*).
- Generación de música.

Además de en NLP, las RNN también pueden aplicarse a problemas de clasificación como el que ocupa este TFG donde una secuencia de variables corresponde un gesto (categoría) concreto dentro del Lengua de Signos Española (LSE).

3.4.2. Esquemas RNN para aplicaciones prácticas

Existen diferentes esquemas de red para una arquitectura RNN dependiendo del número de entradas y salidas de la misma. Cada esquema es útil para una aplicación determinada. Los tipo de esquemas que existen en la práctica son:

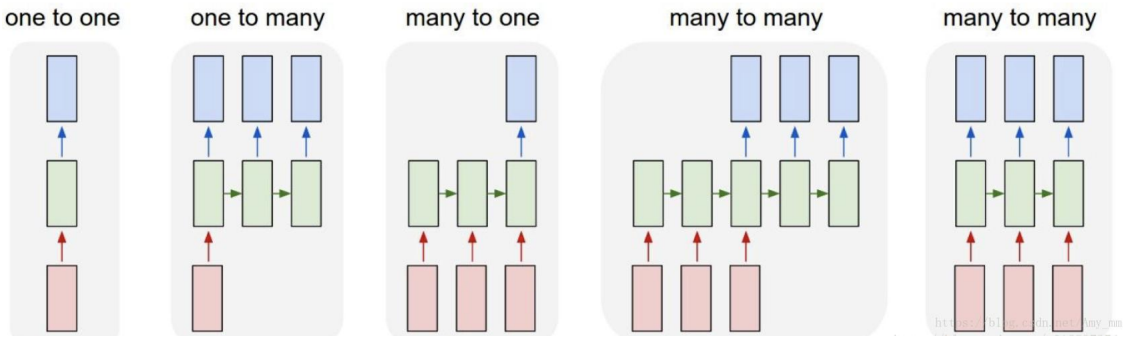


Figura 3.8: Esquemas tipos RNN.

- **One-to-One:** red retroalimentada más simple.
- **One-to-Many:** para una única entrada se devuelven varias salidas. Un ejemplo de aplicación es la conversión de imágenes a texto.
- **Many-to-Many:** se utiliza cuando la longitud de entrada es incierta pero la secuencia de salida respeta las dimensiones de la secuencia de entrada. Un ejemplo común de uso es el reconocimiento de voz.

  Además, también se aplica cuando la longitud de las secuencias de entrada y salida es no tienen porqué coincidir como puede ser el caso de la traducción automática en la que la misma frase en una lengua y en otra puede variar en términos sintácticos (longitud de la secuencia).
- **Many-to-One:** este esquema se utiliza para problemas de categorización como puede ser el análisis de sentimientos en el que una secuencia de palabras (oración) se cataloga dentro de una categoría específica (sentimiento).

Este último caso es el que ocupa el presente TFG aplicado al reconocimiento de gestos aislados de la LSE. Lo que resta de memoria, contiene el desarrollo de la solución implementada.

4. Lenguaje de programación Python

La implementación de la solución propuesta en este proyecto ha sido desarrollada en lenguaje **Python**.

Python es un lenguaje de programación interpretado con licencia de código abierto cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Con el auge de la inteligencia artificial, Python se ha convertido el lenguaje de programación líder para el desarrollo de modelo basados en aprendizaje automático y aprendizaje profundo. Algunas razones por las que han hecho de este lenguaje de programación uno de los favoritos dentro del campo de las redes neuronales son:

- Al tratarse de un **lenguaje de alto nivel**, supone una sintaxis sencilla que ayuda a la legibilidad del código. Este factor es clave en la relación complejidad-rendimiento ya que, por una parte, desarrollar y corregir código en Python resulta más sencillo en comparación con otros lenguajes de bajo nivel con mayor rendimiento como pueden ser C o C++, pero, por otro lado, presenta mayor rendimiento que otros lenguajes de alto nivel como son R o MATLAB®.

- Dispone de una **gran cantidad de librerías** para todo tipo de áreas que cuentan con una gran cantidad de funciones, herramientas y algoritmos ya programados y optimizados listos para usar. Esto permite ahorrar mucho tiempo de programación. También importante destacar la posibilidad de combinar y mejorar estas librerías haciendo de Python una herramienta muy potente. En materia de ciencia de datos y *machine learning*, Python cuenta con numerosas librerías públicas de las que se hablan más adelante en este mismo capítulo.

- Debido a su gran popularidad, en internet se puede encontrar un **amplio soporte por parte de la comunidad** lo que facilita su comprensión y uso.

4.1. TensorFlow y Keras

Keras es una librería de alto nivel capaz de ejecutarse sobre TensorFlow que permite desarrollar de forma muy sencilla arquitecturas basadas en redes neuronales. Lo más atractivo de esta librería es que contiene diversas implementaciones de los bloques constructivos de una red neuronal como son las capas (*layers*), funciones de activación, optimizadores matemático, funciones de coste.. Además de soporte para las redes neuronales convencionales, esta librería también ofrece soporte para Redes Neuronales Convolucionales y para Redes Neuronales Recurrentes.

TensorFlow es una plataforma de extremo a extremo que facilita tanto la compilación como la implementación de modelos de aprendizaje automático (TensorFlow, 2021). Esta librería es una herramienta de bajo nivel que trabaja con grafos de flujo formados por operaciones matemáticas representadas sobre nodos, y cuya entrada y salida es un vector multidimensional (o tensor) de datos.

Para la implementación de la solución propuesta se ha utilizado la API de Keras incluida en TensorFlow por tres motivos principales:

- La API de Keras permite programar de forma sencilla implementaciones de redes neuronales gracias a su sencilla sintaxis.
- La curva de aprendizaje es mucho mayor en la librería de bajo nivel TensorFlow.
- El rendimiento de la librería TensorFlow es muy superior ya que está desarrollada pensando en el cómputo en paralelo y el uso optimizado de la GPU.

4.2. Scikit-learn

Scikit-learn es una librería enfocada al aprendizaje automático que incluye varios paquetes como NumPy, Pandas, Matplotlib, SciPy, etc. Esta librería ofrece, entre muchas otras, herramientas para el preprocesamiento de datos, la selección de características (reducción de la dimensionalidad), la optimización de hiperparámetros y algoritmos de aprendizaje supervisados y no supervisados que permiten crear modelos que den soluciones a problemas de clasificación, regresión y clustering.

En este TFG, la construcción y entrenamiento del modelo se ha realizado a partir de TensorFlow y Keras. La librería Scikit-learn se ha usado para la validación cruzada del modelo ya que esta librería ofrece diversos métodos para verificar la precisión de modelos de clasificación. Los métodos usados para la evaluación del modelo que pertenecen a esta librería se desarrollan en el apartado 6.4.

4.3. Otras librerías

Otras librerías utilizadas para la parte de preparación y preprocesamiento de los datos han sido:

- **NumPy.**

  Esta librería permite trabajar con matrices multidimensionales permitiendo operar matemáticamente sobre ellas con un rendimiento muy elevado.

- **Pandas.**

  Esta librería basada en NumPy, proporciona estructura de datos conocidas como *dataframes* que permiten representar tanto datos tabulares con etiquetas en columnas y filas como series temporales. Además, entre sus funcionalidades encontramos algunas muy útiles para el preprocesmiento de datos como son la lectura y escritura de datos en diferentes formatos (este proyecto en concreto ha trabajado en formato *.csv*), selección y filtrado de los datos organizados en tablas, fusión y unión de datos y transformación de los mismos.

  En este proyecto ha sido de gran utilidad para manejar los datos antes y después de su preprocesamiento pero también para realizar algunas operaciones sobre los mismos.

- **Matplotlib.**

  La principal funcionalidad de esta librería es la generación y personalización de gráficos de diversos tipos a partir de datos contenidos en arrays.

  En este proyecto se ha empleado para generar gráficas con los resultados obtenidos de la evaluación del modelo.

- **os.**

  Este módulo de la librería estándar de Python permite acceder a funcionalidades dependientes del Sistema Operativo que nos permiten manipular la estructura de directorios.

  Su uso principal en este proyecto ha sido la lectura del contenido de los directorios de los datos.

5. Diseño de la solución

Este TFG propone una solución de reconocimiento automático de signos aislados de la LSE mediante uso de redes RNN. Esta tarea se ha planteado como un problema de clasificación del tipo Many-to-One. La base de datos utilizada es la explicada en el apartado 5.1 la cual consta de un total de 3640 muestras de 91 gestos distintos de la LSE, es decir el problema a abordar consta de 91 clases diferentes.

La solución propuesta se compone de cuatro procesos principales:

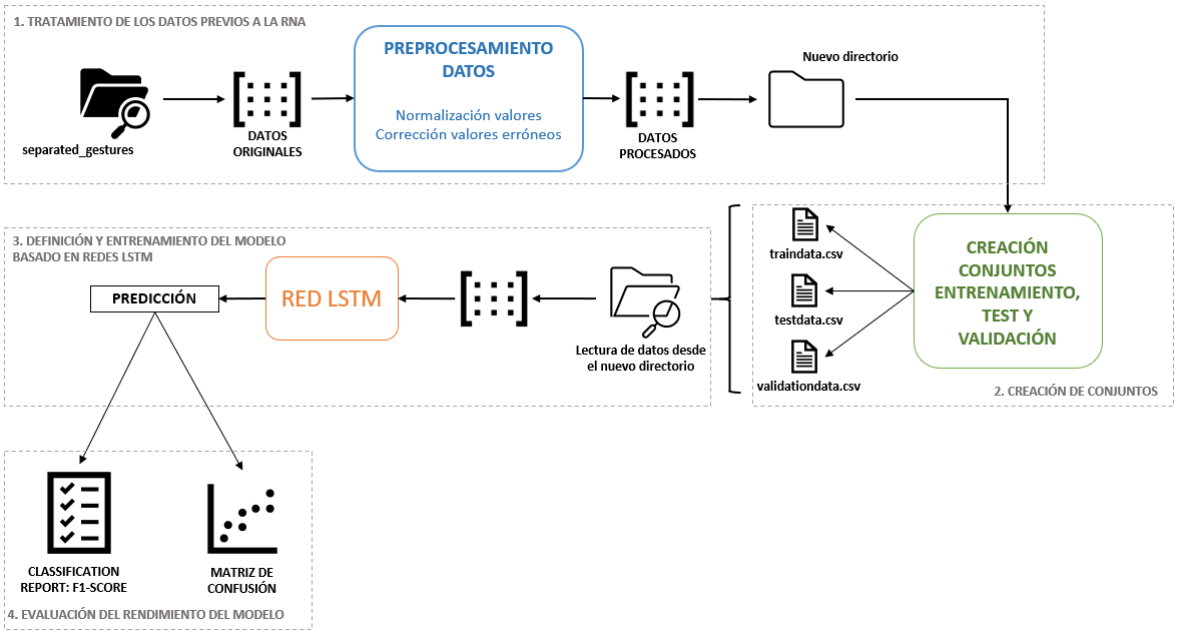


Figura 5.1: Diagrama de flujo solución propuesta.

5.1. Base de datos

La base de datos utilizada para la solución que propone este TFG es la proporcionada por el proyecto de Parcheta y Martínez-Hinarejos (2017) la cual se puede obtener directamente desde su página de GitHub disponible en la bibliografía. Toda la información expuesta en este punto se ha sacado directamente de esa misma página.

Para la solución que propone este proyecto, se ha trabajado con el dataset **separated_gestures** compuesto de 91 gestos del LSE distintos con 40 muestras de cada gesto, lo que supone un total de 3640 archivos. Estos 91 gestos conformarán las 91 clases de nuestro problema de clasificación. Los 91 gestos concretos que forman la base de datos son los que se indican en el Anexo A.

Los datos han sido capturado mediante un sensor Leap-Motion (tecnología explicada en el apartado 2.1.4 de esta memoria). Los archivos resultante contienen 42 columnas correspondientes a 42 variables diferentes con información 3D sobre la posición de la mano (21 variables para cada mano).

A modo de resumen, las variables registradas por el sensor para cada mano son las siguientes:

1. Componente X, Y y Z de la dirección de la posición de la palma desde esta hacia los dedos.
2. Ángulo de la mano.
3. Inclinación de la mano.
4. Dirección de la mano.
5. Dirección del dedo pulgar en los ejes X, Y y Z.
6. Dirección del dedo índice en los ejes X, Y y Z.
7. Dirección del dedo corazón en los ejes X, Y y Z.
8. Dirección del dedo anular en los ejes X, Y y Z.
9. Dirección del dedo meñique en los ejes X, Y y Z.

La frecuencia de muestreo de estas variables es de 30 Hz. Además, si un gesto se realiza son una sola mano, esta información se duplica para rellenar las variables de la otra mano.

Cada fichero CSV de la base de datos, es decir cada muestra, es una matriz cuyo número de filas, equivalente al número de secuencias de datos, es variable en función de la muestra del gesto en particular y cuyo número de columnas es 42 ya que representan las 42 variables que recoge el sensor. Por tanto, cada fichero resulta un array de dimensiones (n secuencias x 42) que sigue la siguiente distribución:

	caraterística 1	*caraterística 2*	*caraterísticas 3 a 40*	*caraterística 41*	*caraterística 42*
sec. 1	-1.202770171	0.19039897	...	-0.932502043	0.277868281
sec. 2	-1.212970352	0.226585982	...	-0.938844164	0.27820857
sec. 3	-1.2234931	0.262852281	...	-0.941933548	0.281369433
sec. 4	-1.236664352	0.305035613	...	-0.942083611	0.288819834
sec. 5	-1.253253006	0.356350804	...	-0.939677899	0.299795086
sec. 6	-1.270299953	0.408030766	...	-0.935969362	0.309178734
sec. 7	-1.282690185	0.443540731	...	-0.932685108	0.309747766
sec. 8	-1.286327055	0.447224759	...	-0.930480187	0.295617755
sec. 9	-1.28128587	0.418933848	...	-0.928860501	0.26529326
sec. 10	-1.270859477	0.363347042	...	-0.92627754	0.223459548
sec. 11	-1.260798522	0.302759564	...	-0.920506346	0.182052017
sec. 12	-1.25893954	0.26378177	...	-0.909633663	0.156317561
sec. 13	-1.266124898	0.265927026	...	-0.893040457	0.156866152
sec. 14	-1.276358464	0.308436544	...	-0.871710769	0.188195381
sec. 15	-1.280993582	0.388957732	...	-0.849788783	0.245530072
sec. 16	-1.274278625	0.485810416	...	-0.832337355	0.315051128
sec. 17	-1.251244126	0.579057412	...	-0.822347561	0.381289625
sec. 18	-1.214700154	0.654014635	...	-0.821204566	0.433436808
sec. 19	-1.172615473	0.707581584	...	-0.829022696	0.464686424
sec. 20	-1.133819567	0.7345561	...	-0.842554555	0.475265023
sec. 21	-1.102656231	0.744164331	...	-0.857381292	0.472140311

Tabla 5.1: Muestra del contenido de un fichero de la base de datos separated_gesture.

5.2. Arquitectura de red

Con el fin de llevar a cabo la tarea de reconocimiento de gestos aislados, la arquitectura de red sigue el esquema propio de un problema de clasificación del tipo Many-to-One. La arquitectura de red que propone el presente TFG se basa en redes LSTM y es la siguiente:

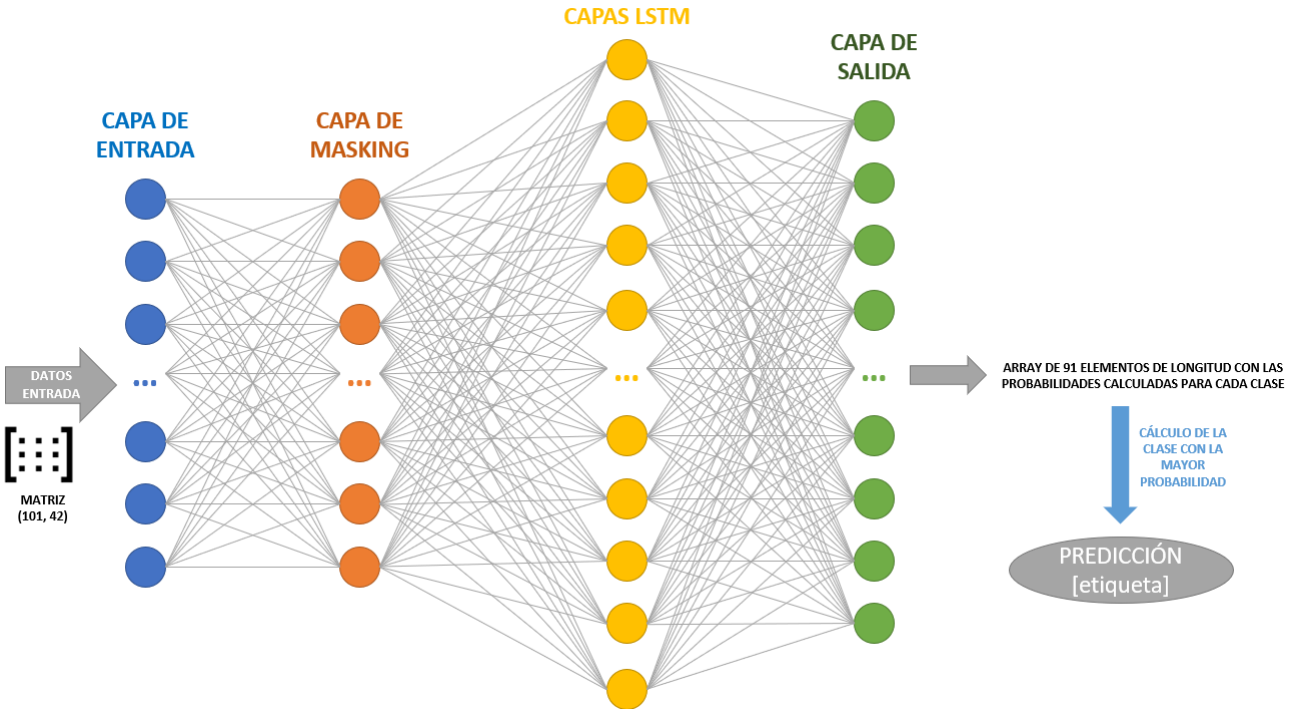


Figura 5.2: Arquitectura de red de la solución propuesta

En este apartado se explican las diferentes capas utilizadas en la arquitectura de red propuesta y el papel que desempeña cada una. Todas pertenecen al módulo *layers* de la librería Keras (ejecutada sobre TensorFlow). Antes de comenzar a enumerar las diferentes capas empleadas en el diseño de la solución, es conveniente indicar que todos los hiperparámetros mencionados en este punto se definen en el apartado 5.3.

La capa ***InputLayer()*** actúa como capa de entrada a la red, es decir, como nodo de entrada de los datos al resto de capas. En ella únicamente se especifica el tamaño de los datos de entrada mediante su hiperparámetro *input_shape*.

Las capas ocultas de la red son de tres tipos:

- ***Masking()*** es la primera capa tras la capa de entrada. Su función es ignorar un cierto valor, especificado mediante su hiperparámetro *mask_value*, si este aparece entre los

datos de entrada por lo que también es necesario especificar el tamaño de los mismos mediante *input_shape*. De esta manera, podemos obviar la información de *padding* de nuestros datos de entrada (explicado con detalle en el apartado 6.3).

- ***LSTM()*** es el tipo de capa más importante de la red ya que permite implementar la red recurrente. Para cada capa LSTM, se especifica el número de neuronas recurrentes a través de su hiperparámetro *units*. Además, otro hiperparámetro importante en esta capa es el booleano *return_sequence* que por defecto tiene valor *False* pero que debe modificarse a *True* siempre que se emplee más de una capa LSTM para construir la red. Su función de activación por defecto es la función tangente hiperbólica (*tanh*).

- ***Dropout()*** es una capa, como se comentó en el apartado 3.2.2, de regularización de la red para desactivar de forma aleatoria parte de las entradas que recibe. El porcentaje de desactivación se indica a través de su hiperparámetro *rate*. La finalidad de esta capa es la prevención o el control del posible *overfitting* obtenido al entrenar la red. Es por eso que esta capa está diseñada para que solo se active durante el entrenamiento de forma que no se descarten valores a la hora de evaluar la red o de usar el modelo para hacer predicciones de nuevos datos.

La capa de salida de la red se trata de una capa tipo ***Dense()***. Al tratarse de un problema de clasificación, la capa de salida debe tener tantas neuronas como categorías tiene problema en cuestión (en nuestro caso concreto serán 91, especificado mediante su hiperparámetro *units*). También se debe indicar la función de activación de la capa a través del hiperparámetro *activation*. En la solución propuesta se ha optado por usar la función de activación *softmax* cuyo funcionamiento, junto con la razón por la que es útil para problemas de clasificación, se indica en el siguiente aparado.

A modo de aclaración cabe destacar que, al trabajar con Keras, basta con indicar las dimensiones de los datos de entrada a la primer capa de la red pues automáticamente se crea una capa del tipo *InputLayer()*. Sin embargo, incluir una capa propia de entrada facilita la comprensión de la arquitectura de red y no supone un aumento en el coste computacional relevante.

5.3. Hiperparámetros de la red

En este apartado se explican todos los hiperparámetros de diseño del modelo así como la justificación de su elección.

5.3.1. Parámetros fijos

Hay parámetros que se han mantenido fijos durante todas las pruebas de la red pues son los que se ajustan al problema de clasificación concreto de este proyecto.

Función de activación. Transformación que aplica cada neurona de la capa (como se explica en el apartado 3.1.2). Se han usado dos funciones de activación distintas:

- *Softmax*: calcula la distribución de probabilidades del evento sobre k eventos diferentes, es decir, calcula las probabilidades de cada categoría sobres todas las categoría posibles. El rango de probabilidades de salida es [0, 1].

$$\phi(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{k} e^{x_j}} \quad i = 0, 1, ..., k \tag{5.1}$$

  Al ser la función de activación de la capa de salida de la red, encontramos que la salida del modelo ante un dato de entrada es una distribución de probabilidad de cada una de las categorías involucradas en el modelo. De esta manera, la clase con la mayor probabilidad será la predicción del modelo ante los datos de entrada. Además, indica cual es la segunda opción de predicción para el mismo dato ya que será la categoría con la segunda probabilidad más alta. Es por esta razón que esta función de activación es muy útil en problema de clasificación.

- Tangente hiperbólica (*tanh*): como se explica en el apartado 3.3.4, la función de activación tangente hiperbólica forma parte del funcionamiento interno de una neurona LSTM. El rango de esta función es [-1, 1] por lo que puede manejar fácilmente valores negativos.

$$tanh\,(x_i) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad \forall\, x \,\epsilon\, \mathbb{R} \tag{5.2}$$

La **función de coste** utilizada para construir el modelo ha sido la entropía cruzada categórica (*Categorical Cross-Entropy*). La fórmula matemática para calcular el error entre la predicción y la categoría real del dato de entrada mediante este método es:

$$L(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} log(p_{ij}) \tag{5.3}$$

Donde y_{ij} es el valor verdadero y p_{ij} es la probabilidad del dato, obtenida mediante la función de activación *softmax*, para la clase i.

Como se explicará en el apartado 6.3, las etiquetas (categorías) del conjunto de datos de nuestro modelo estarán codificadas con valores enteros. Por este motivo, la función de coste disponible en la librería Keras que se ajusta a este formato de los datos es *'sparse_categorical_crossentropy'*.

El **optimizador** de la función de coste escogido ha sido el Adam (*'adam'* en Keras) ya que resuelve el inconveniente principal del descenso estocástico del gradiente en el que la tasa de aprendizaje fija para actualizar los pesos. La tasa de aprendizaje o ratio de aprendizaje (en inglés *learning rate*) controla los cambios que sufren los pesos de la red. Un ratio de aprendizaje demasiado grande, produce cambios grandes en los pesos lo que supone que se dificulte la tarea de encontrar los coeficientes que minimicen la función de coste que es la función principal de los optimizadores. Por el contrario, un ratio de aprendizaje demasiado pequeño se traducirá en un aumento del tiempo necesario para encontrar la solución más óptima. Por este motivo se ha seleccionado el optimizador Adam para construir la red ya que se caracteriza por adaptar el ratio de aprendizaje en función de como están distribuidos los parámetros; por este motivo, si los parámetros están muy dispersos el ratio de aprendizaje aumentará.

Otro parámetro que se mantiene fijo durante las diferentes pruebas del modelo son las **dimensiones de los datos de entrada** de la red (que condicionan los valores del hiperparámetro *input_shape* anteriormente mencionado) que será un array de 101 secuencias de 42 características para cada fichero de datos. Es decir el tamaño de entrada es (101, 42) por cada muestra de entrada. En el apartado 6.2 se explica el porqué de estas dimensiones concretas pues se deben al preprocesamiento que se aplica a los datos antes de usarlos en el modelo.

5.3.2. Parámetros variables

Para dar lugar a diferentes modelos, se ha variado tanto el **número de capas LSTM ocultas** como el tamaño de las mismas, es decir el **número de neuronas (*units*) que conforman cada capa oculta**. Además, se han hecho sin usar *Dropout* y usándolo con diferentes valores de *rate* para ver como afecta este método de regularización al rendimiento de la red (aspecto que se comenta con mayor profundidad en el apartado 7).

Otros parámetros relacionados con el entrenamiento de la red que han tomado diferentes valores durante las pruebas realizadas son:

- **Número de épocas *(epochs)***: indica el número de veces que se entrena la red, es decir, el número de veces que el conjunto de datos de entrenamiento completos se expone a

la red. A mayor número de iteraciones o épocas, mayor rendimiento que se obtiene del modelo. El rendimiento de la red aumenta hasta que se alcanza el estado óptimo de la red. Durante las pruebas realizadas, se ha entrenado el modelo con 10, 15 y 20 épocas para ver como mejora dicho rendimiento.

- **Tamaño del *batch (batch_size)*:** indica el número de muestras que evalúa el modelo entre cada actualización de los pesos. Los dos valores utilizados para probar el rendimiento de la red han sido 64 y 128.

6. Desarrollo

En este apartado se pretende explicar la solución (implementada en lenguaje Python) propuesta en este TFG para el reconocimiento de la LSE.

El proyecto se ha divido en tres ficheros Python diferentes:

- **utils.py** se ha utilizado a modo de librería propia y contiene diversas funciones utilizadas en diferentes partes de la implementación. La finalidad de cada función se explica en los siguientes apartados a medida que vaya siendo necesario.
- **RNN.py** contiene el código que tiene que ver con la lectura de datos, construcción del modelo y entrenamiento del mismo.
- **modelevaluation.py** es un script para cargar un modelo entrenado, hacer una predicción a partir de nuevos datos y evaluar los resultados obtenidos.

Estos códigos se encuentran en el Anexo B, sin embargo, este apartado se centrará en realizar una explicación literaria de los mismos.

La manera en la que se han desarrollado los diferente procesos principales de los que consta la solución implementada (veáse Figura 5.1) se exponen a continuación.

6.1. Creación de los conjuntos de datos

Cuando se pretende desarrollar un modelo de Machine Learning o, como en este proyecto, de Deep Learning, nuestra base de datos se divide en tres conjuntos de datos:

- **Conjunto de entrenamiento:** es el conjunto con mayor cantidad de datos y es con el que el modelo entrena. Son los datos a los que mejor se adapta nuestra red por la que evaluar el modelo a partir de estos datos es una idea errónea y demasiado optimista. Por este motivo, existen otros dos conjuntos menores que sirven para evaluar el modelo y son los que se explican a continuación.
- **Conjunto de validación:** cuenta con una pequeña cantidad de muestras y sirven para evaluar el rendimiento de la red durante el entrenamiento con datos que no haya visto

previamente, es decir, con dato distintos al conjunto de entrenamiento. La mejor *accuracy* que se obtenga para este conjunto de datos (en Keras se denomina *val_accuracy*) será la que indique en qué época el modelo está ofreciendo el modelo.

- **Conjunto de test:** unas cuantas muestras se reservan para evaluar el rendimiento final y real del modelo. En un caso de aplicación práctica, serían nuevos datos de los cuales se quiere obtener una predicción. Este conjunto sirve para obtener resultados de predicción que den una idea realista y cercana a lo que sería el caso de una aplicación práctica.

Para que la implementación de un modelo de clasificación sea correcta, estos tres grupos deben ser balanceados y equilibrados en cuanto a número de muestras por clase se refiere. Esto significa que todas las clases deben estar igualmente representadas. Como se ha comentado en el apartado 3.2.2, este aspecto es importante para minimizar la probabilidad de que la red caiga en *overfitting*. Además, es conveniente que las muestras que se entregan al modelo (sobre todo para su entrenamiento) entren a la red de manera desordenada de forma que el modelo no reciba demasiadas muestras de la misma categoría de forma seguida ya que supondría que el modelo aprendería solo una clase cada vez y olvidaría el resto que había aprendido.

En este proyecto la creación de los conjuntos de entrenamiento, test y validación se ha realizado de manera manual mediante la función **groups()**. Esta función se ha programado buscando cumplir con lo que se acaba de mencionar y asegurando la representación equitativa de datos comentada. Sin entrar en detalles de como se realiza esta tarea, a la hora de llamar a la función se indica el porcentaje de muestras que formará cada grupo. A partir de estos datos, se crean tres ficheros (traindata.csv, testdata.csv y validationdata.csv) que contienen dos tipos de información: el nombre de los ficheros que forman el conjunto y su etiqueta correspondiente. De esta manera logramos obtener los diferentes conjuntos de datos etiquetados necesarios para abordar nuestro problema de clasificación de gestos aislados.

En este proyecto se ha trabajado con los siguientes porcentajes:

- 70% para el conjunto de entrenamiento (28 muestras por clase), lo que supone un total de 2548 muestras para entrenar la red.
- 15% para el conjunto de validación (6 muestras por clase), lo que supone que disponemos de 546 muestras para evaluar el modelo durante su entrenamiento.
- 15% para el conjunto de test (6 muestras por clase), lo que supone que disponemos de otras 546 muestras distintas para realizar predicciones ante nuevos datos y evaluar el rendimiento real de la red.

6.2. Preprocesamiento de los datos

La primera tarea realizar antes de construir un modelo basado en redes neuronales es preparar los datos para el mismo.

La **normalización** de los datos de entrada evita que, cuando existen escalas diferentes entre los mismos, la red otorgue mayor importancia a los valores más altos pues estos serán los que más contribuyan al error del modelo. Además, sirve de ayuda a la convergencia de la red.

Pese a que las magnitudes de los datos empleados en este proyecto no es notable, los datos se han normalizado con el objetivo de minimizar el tiempo necesario de entrenamiento de la red. Los datos se han normalizado en el rango [0, 1] de la siguiente forma:

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{6.1}$$

Este proceso de normalización se ha llevado a cabo para cada columna de datos (correspondientes a cada característica registrada) para todos las muestras de la base de datos mediante la función **mimnax_norm()**.

Además, tras normalizar cada archivo se ha comprobado que no se produjera ningún valor no deseado en el proceso. Esto ha sido necesario porque mientras se trabajaba en el modelo se encontró un error debido a archivos que contenían valores NaN (*Not a Number*) lo que hacía que la red divergiese durante su entrenamiento al toparse con este tipo de datos y no llegase a aprender nunca.

Al implementar la comprobación de valores NaN dentro de los datos normalizados, resultó que cuatro de los archivos de la base datos (gris1.csv, nosotras6.csv, porfavor4.csv y porfavor11.csv) consistían en matrices enteras de NaNs. Estos cuatro archivos han sido descartados de la base de datos. La forma en la que se han descartado se ha adaptado a la forma en la que se crean los conjuntos de entrenamiento, test y validación, ya que en este proceso se tiene en cuenta las 40 muestras de cada gesto (esto se explica en el apartado 6.1 de este mismo capítulo). Por este motivo, la forma de descartar estos ficheros ha sido reemplazando todos los valores NaN por el valor 2 el cual será ignorado por la red como se explica en el apartado 6.4. Además, haciéndolo de esta manera nos aseguramos, en caso de existir algún otro valor NaN oculto entre el resto de muestras, este también sufra dicho reemplazo evitando errores al entrenar el modelo.

Todos los archivos normalizados, comprobados de presencia de valores NaN y corregidos en caso de contener este problema se han guardado en un nuevo directorio que será del que se lean los datos con los que trabajará el modelo.

Todo este procedimiento se realiza dentro de la función **save_norm()**.

6.3. Lectura de los datos

Para la lectura de datos se ha creado la función **read_data()** para obtener, a partir de los archivos que contienen al información sobre los diferentes conjuntos mencionados en el apartado anterior, la siguiente información:

- **Datos X:** obtención de los datos almacenados en los diferentes ficheros de nuestra base de datos. Cada fichero es leído y almacenado en un array multidimensional que contendrá todas las muestras del conjunto que se esté leyendo. Para entrenar una red neuronal, el tamaño de los datos de entrada a la misma debe ser siempre el mismo por lo que cada muestra X tiene el tamaño de 101 secuencias (correspondiente al valor del fichero con más secuencias registradas) con 42 características por cada una (valor fijo que corresponde a las características que registra el sensor utilizado en la captación de gestos). Para conseguir igualar el número de dimensiones, se ha aplicado un *padding* al final de todas ellas de valor 2. Se ha escogido este valor ya que todos nuestros datos están normalizados en el rango [0, 1] y de esta manera podemos indicarle a la red que ignore todos los valores que coincidan con este *padding* haciendo uso de un capa **Masking()** justo después de la capa de entrada al definir la arquitectura de red. Finalmente, los datos X se almacenan en un array de dimensiones (Número_muestras_conjunto x 101 x 42).

- **Datos Y:** consiste en un array con todas las etiquetas codificadas correspondientes a cada muestra de Datos X. Para codificar las etiquetas se ha usado un codificador Label Encoder el cual asigna un valor entero a cada clase. Para usar este codificador de la librería Scikit-learn solo hace falta entrenarlo con el conjunto de gestos (clases) de nuestro problema y almacenarlo en una variable para poder emplearlo a lo largo del programa.

Esta parte del proceso de implementación de la solución puede considerarse, en cierta forma, preprocesamiento de los datos ya que es la última "transformación" que sufren antes de entrar a la red neuronal.

6.4. Definición, entrenamiento y evaluación del modelo

Una vez creados los conjuntos de entrenamiento, test y validación, preparado correctamente los datos para el problema de clasificación que se pretende resolver y habiendo leído los mismos para usarlos en el modelo, el último paso es definir el mismo para entrenarlo y evaluarlo.

Para realizar este último paso, se ha empleado: por una parte, la librería Keras para definir, entrenar el modelo y realizar predicciones con el mismo y, por otra, la librería Scikit-learn para hacer mediciones del rendimiento de dichas predicciones.

El modelo que propone este TFG para abordar el problema de clasificación de gestos es un modelo secuencial, es decir, una agrupación lineal de capas. Las capas que forman la red son las mencionadas en el apartado 5.2. Para dar lugar a los diferentes modelos de prueba se han mantenido fijas la capa de entrada, la capa de enmascaramiento y la capa de salida y se ha jugado con el número y tamaño de capas LSTM, dando lugar a distintos modelos cuyos resultados se exponen en el apartado 7.

Una vez definido el modelo, el siguiente paso es entrenarlo con los datos del conjunto de entrenamiento (haciendo uso del conjunto de validación para tener una valoración real durante el propio entrenamiento). Durante las primeras pruebas, el modelo se entrenaba con pocas épocas buscando el funcionamiento del programa más que la obtención de buenos resultados. Tras varias pruebas aumentando el número de épocas, se comprobó que el rendimiento de la red alcanzaba su punto óptimo entre las 15 y las 20 épocas de entrenamiento. Por esta razón se decidió definir un *checkpoint* que marcase la época para la cual el valor de la *val_accuracy*, esto es la precisión del modelo con los datos de validación, era máximo. De esta forma, los pesos de la red de esa época exacta se guardan y se pueden cargar, al finalizar el entrenamiento, como los pesos finales del modelo. Esto se hace para poder evaluar y hacer predicciones con los pesos de la red que mejor rendimiento ofrecen. Además, el modelo completo ya entrenado se guarda para, posteriormente, poder hacer predicciones de nuevos datos y medir su rendimiento sin necesidad de volver a pasar por el proceso de entrenamiento de nuevo.

Para evaluar los distintos modelos se han usado tres medidas distintas, todas ellas a partir de los datos del conjunto de test.

- El método **evaluate()**, disponible en la librería de Keras, ofrece el valor de la *loss*, es decir, el error (parámetro que indica cuan mala ha sido la predicción) y la *accuracy* obtenida para los datos X e Y introducidos. Esta es una buena forma de evaluar el modelo ya que se trabaja con datos que la red no ha visto durante el entrenamiento.

- El método **predict()**, también de Keras, permite realizar predicciones reales de los datos de entrada. De esta manera obtenemos unos datos Y predichos por la red que podemos comparar con los datos Y originales que corresponden a los de entrada. Una buena forma para evaluar el rendimiento de la red en problemas de clasificación como el que ocupan este proyecto es comparando estos dos conjuntos de datos Y. En la implementación llevada a cabo se ha realizado esta comparación de dos formas distintas:

1. Calculando el ***classification report*** del modelo el cual nos ofrece diferentes medidas, todas en tanto por ciento, para cada clase:

   a) *precision:* mide la capacidad de acierto de la red para cada clase.

   b) *recall:* mide la cantidad de veces que a una predicción se ha asignado a esa clase.

   c) *f1-score:* es una media armónica de las medidas *precision* y *recall* que indica que porcentaje de todas las predicciones de esa clase eran correctas.

   d) *support:* indica la representación que tiene cada clase dentro de la red.

   Además, también indica la *accuracy* del modelo al final de todas las medidas, por lo que se puede prescindir del método **evaluate()** si se quisiera.

2. Calculando la **matriz de confusión** la cual permite visualizar, de manera gráfica y con datos en forma de porcentaje, la relación entre los valores predichos y los reales permitiendo comprobar de manera sencilla si el modelo está confundiendo de forma relevante ciertas clases en concreto. Una matriz de confusión con un modelo muy bien entrenado consistiría básicamente en una diagonal.

7. Pruebas y resultados

Con la finalidad de comprobar el rendimiento que ofrecen diferentes arquitecturas de red, todas basadas en redes LSTM, se han realizado varios experimentos. Este apartado reúne información sobre el rendimiento de los modelos que se han probado y, además, se analizan los resultados obtenidos con el mejor de los modelos.

7.1. Resumen modelos probados

Como se ha comentado en el apartado 5.3, uno de los parámetros que se ha variado para dar lugar a diferentes pruebas de rendimiento, ha sido el número de capas LSTM así como la cantidad de neuronas de estas capas. Además, se han hecho pruebas para comprobar cómo afectan al rendimiento las capas Dropout.

La siguiente tabla resume los modelos construidos y los resultados obtenidos para la *accuracy* en cada uno de ellos.

ID	Arquitectura de red	Parámetros	Accuracy
Modelo1	1 capa LSTM con 150 neuronas	129541	70,70%
Modelo2	2 capas LSTM con 150 neuronas	310141	75,20%
Modelo3	3 capas LSTM con 150 neuronas	490741	74,54%
Modelo4	3 capas LSTM con 150 neuronas		
	+ capas Dropout con un rate del 25% intercaladas	490741	73,99%
Modelo5	1 capa LSTM con 300 neuronas	438991	74,73%
Modelo6	1 capa LSTM con 250 neuronas	315841	72.53%
Modelo7	2 capas LSTM con 250 neuronas	816841	76,01%
Modelo8	2 capas LSTM con 250 neuronas		
	+ capas Dropout con un rate del 25% intercaladas	816841	78,21%
Modelo9	1 capa LSTM con 250 neuronas		
	+ 1 capa Dropout con un rate del 25%		
	+ 1 capa LSTm con 125 neuronas	492466	74%
Modelo10	3 capas LSTM con 250 neuronas		
	+ capas Dropout con un rate del 25% intercaladas	1317841	75,09%

Tabla 7.1: Resumen arquitecturas de red probadas.

Al no disponer de una base de datos muy grande, las variaciones en las arquitecturas de red puestas a prueba no han sido muy relevante. Sin embargo, estas pequeñas modificaciones han sido útiles para comprobar la potencia de las redes LSTM.

7.2. Resultados obtenidos

A lo largo de este apartado se comentarán diferentes aspectos que se han observado al realizar el entrenamiento de los diferentes modelos y su posterior evaluación.

El siguiente gráfico muestra los valores del error de predicción (*loss*) y la tasa de acierto (*accuracy*) obtenidos tras evaluar los diferentes modelos usando el mismo conjunto de datos de test.

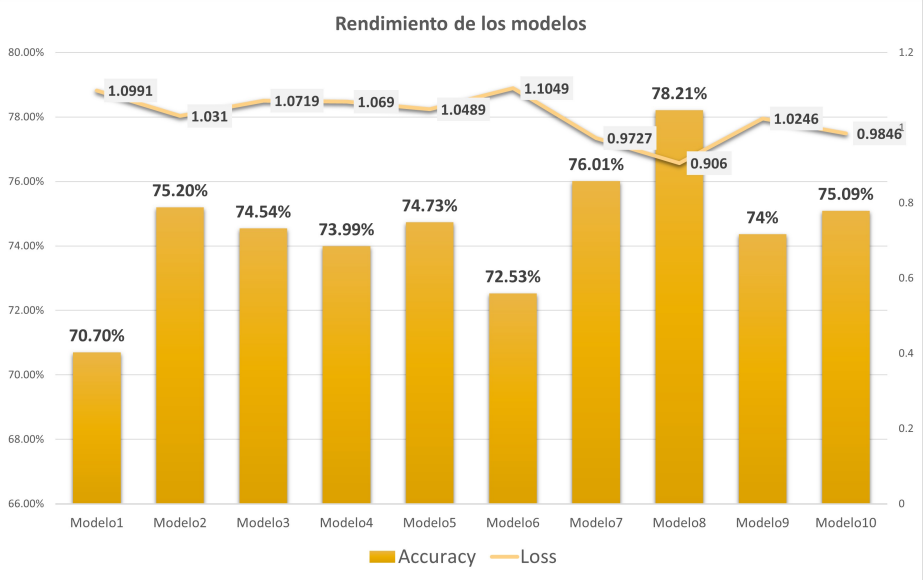


Figura 7.1: Gráfica resumen del rendimiento de los modelos que muestra la *accuracy* (barras) y la *loss* (línea) obtenida con los mismos ante el mismo conjunto de datos de test.

El primer modelo (modelo 1) que se definió, entrenó y evaluó sirvió como punto de partida para analizar el potencial de nuestra red en términos de tasa de acierto. Al contar con una base de datos no muy grande, se decidió empezar con una arquitectura de red en la que la capa LSTM contara con 150 perceptrones. Tras hacer el primer entrenamiento y evaluar el modelo, se obtuvo una *accuracy* del 70,7%, un valor bastante aceptable para tomar como referencia para buscar mejorar el rendimiento de la red.

Si comparamos el modelo 1 con los modelos 2 y 5 podemos ver que duplicar el número de neuronas que conforman la red LSTM, en este caso de 150 a 300 unidades, supone una mejora del rendimiento de la red del 5% aproximadamente. Además, consiguen mejorar el

error de predicción de la red.

Sin embargo, si nos fijamos en los resultados obtenidos para los modelos 3 y 4, lo cuales triplican el tamaño, en términos de cantidad perceptrones LSTM, de nuestro primer modelo, podemos ver que el rendimiento empeora con respecto a los modelos 2 y 5 siendo las arquitecturas de estos últimos más simples.

Pese a que el modelo 4, que contiene regularización de la red mediante las capas Dropout, consigue mejorar un poco el valor del error de la red, la *accuracy* que ofrece el modelo se ve reducida por lo que no resulta una arquitectura de red muy eficiente teniendo en cuenta la cantidad de parámetros que involucra los cuales se traducen en un aumento del tiempo requerido para el entrenamiento de la red.

Después de probar con redes que duplicaban o triplicaban el tamaño de la red de nuestro primer modelo, se buscó una red con un tamaño intermedio entre estos dos puntos que mejorase el rendimiento de la solución.

Tras algunas pruebas, se comprobó que una red LSTM con 250 unidades (modelo 6) mejoraba la *accuracy* de nuestro primer modelo en un 2,5% por lo que se decidió tomar esta red como nuevo punto de partida para buscar un modelo eficiente.

Duplicando el tamaño de la red (modelo 7) se consiguió mejorar la *accuracy* notablemente además de reducir el valor del error obtenido para el modelo. Ante esta mejora, resultó llamativa la idea de probar la misma arquitectura pero añadiendo regularización mediante una capa Dropout intercalada entre las dos capas LSTM con el fin de intentar mejorar aun más el rendimiento del modelo 7.

Esta nueva arquitectura da lugar al modelo 8. Tras entrenar y evaluar el modelo se obtuvo una *accuracy* del 78,21%. Este valor supone una mejora del 7,51% respecto al modelo 1 y del 5,68% respecto al modelo 6. Además, este modelo consigue el valor de *loss* más bajo de todos los modelos probados que sumado a la mencionada mejora de la *accuracy* respecto al resto de modelos lo convierte en la red más potente para nuestro problema de clasificación de gestos aislados.

Para intentar mejorar el rendimiento obtenido con el modelo 8, se probó a reducir el tamaño de la segunda capa LSTM (modelo 9) y también a aumentar el tamaño de la red añadiendo una tercera capa (modelo 10). Al evaluar ambos modelos, se pudo comprobar que el rendimiento de la red empeoraba tanto en términos de *loss* como de *accuracy*.

Esto significa que, para la base de datos disponible, el máximo rendimiento que puede darnos un modelo basado en redes recurrente LSTM es de aproximadamente el 80% de precisión en las predicciones ante nuevos datos.

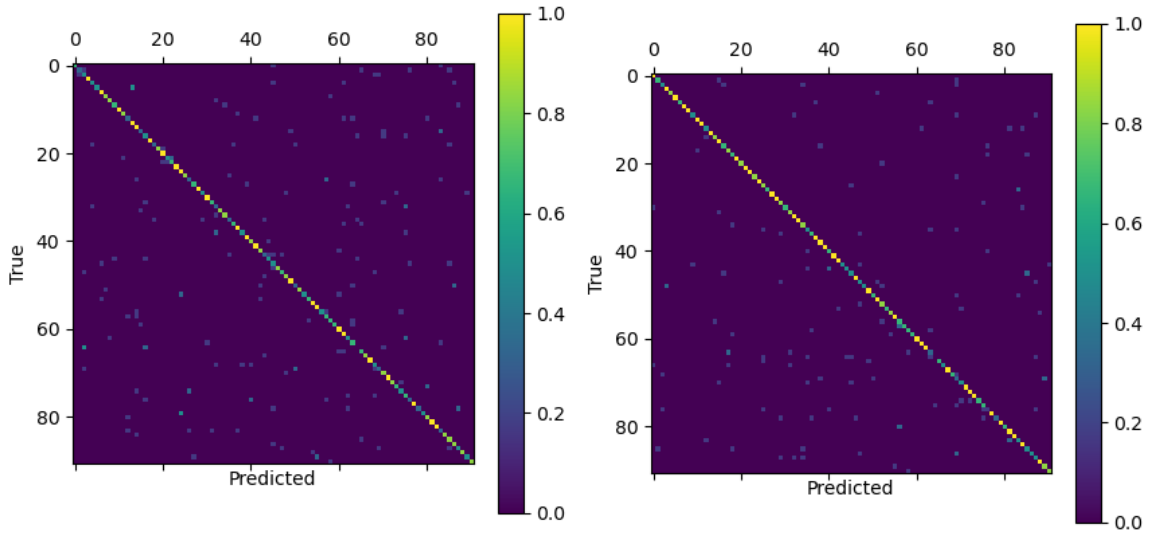
7.3. Comparación modelos

Este apartado pretende hacer una breve comparación entre el modelo de partida, modelo 1, y el modelo con mejor rendimiento, modelo 8.

El anexo C incluye los *classification reports* obtenidos para las predicciones realizadas con ambos modelos.

Si analizamos los resultados de estas tablas se puede apreciar que las medidas de *precision, recall* y *f1-score* (explicadas en el apartado 6.4 de esta memoria) mejoran considerablemente de un modelo a otro. Los valores medios obtenidos para cada media (tanto los macro como los ponderados) también se ve incrementados en, aproximadamente, un 8% de un modelo a otro lo que indica un aumento notorio del rendimiento de la red.

Una forma de visualizarlo es comparando las matrices de confusión obtenidas, ante el mismo conjunto de muestras a predecir, para cada uno de los modelos:



(a) Matriz de confusión modelo 1. **(b)** Matriz de confusión modelo 8.

Figura 7.2: Comparación de las matrices de confusión obtenidas para los modelos 1 y 8 ante el mismo conjunto de muestras (figura ampliada en Anexo D).

Como se puede apreciar, la matriz del modelo 8 muestra una diagonal mucho más limpia y con menos puntos fuera de esta que la matriz del modelo 1. A más cantidad de puntos fuera de la diagonal, mayor será el grado de confusión entre clases, lo conllevará una peor precisión de la red a la hora de realizar predicciones ante nuevos datos.

7.4. Análisis de las predicciones de la solución implementada

Tomando como solución final de este proyecto el modelo 8, se ha realizado un análisis de las clases (gestos aislados) más confundidas por este.

En la siguiente figura se muestran las clases más confundidas por la solución implementada junto con la clase por la que se confunde y el grado en que lo hace en forma de porcentaje:

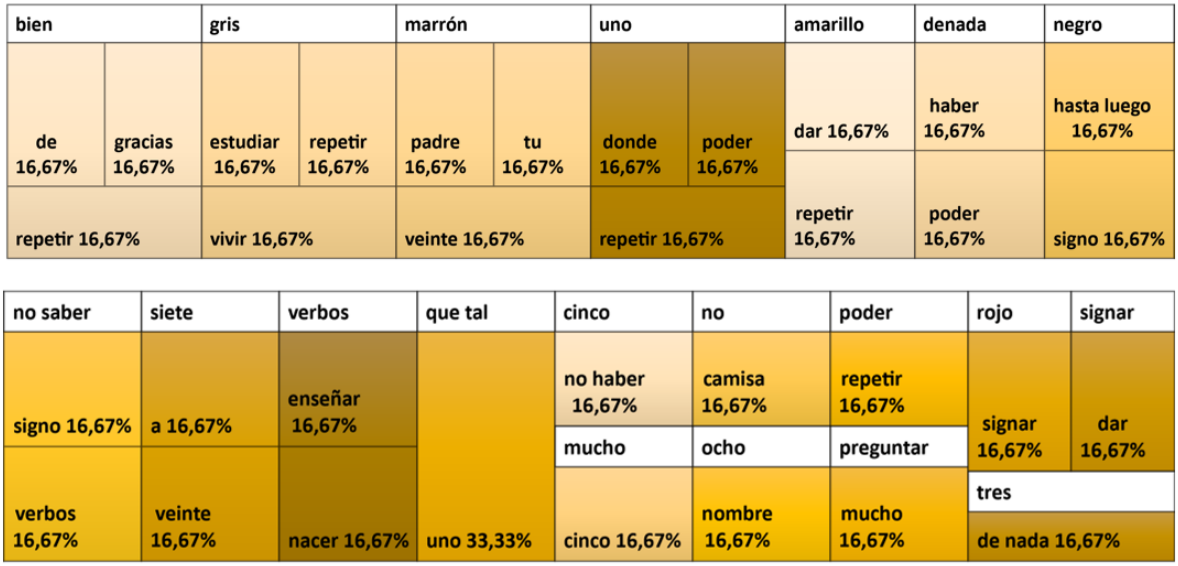


Figura 7.3: Clases más confundidas por el modelo 8.

Para seleccionar las clases más confundidas se han seleccionado aquellas en las que la tasa de acierto es menor al 85%. La mayoría de estas clases "confundidas" por la red cuentan con una tasa de acierto bien del 50% si se confunden con 3 clases o del 66,6% si se confunden con 2 clases (o el caso del gesto "que tal" que se confunde con "uno" en un 33,33%). Sin embargo, otras tienen una tasa de acierto del 83,33% y se confunden con una sola clase en un 16,67%.

7.4.1. Resultados del trabajo de referencia de la base de datos

Como ya se ha comentado, la base de datos utilizada para este proyecto es la misma que la utilizada en el trabajo de Parcheta y Martínez-Hinarejos (2017).

A modo de orientación, el proyecto de referencia, el cual está resumido en el apartado 2.1.4 del estudio del estado de la cuestión que expone este trabajo, consigue una matriz de confusión en el que las clases más confundidas son:

Clase	Confundida con	Grado de confusión	Tasa de acierto final
ella	tu	20%	62,5%
	nosotras	17,5%	
no	uno	25%	47,5%
	tu	15%	
	él	5%	
	once	5%	
	olvidar	2,5%	
que tal	bien	32,5%	62,5%
	gris	5%	
regular	por favor	17,5%	60%
	buenas noches	10%	
	que tal	7,5%	
	nacer	5%	
rojo	España	15%	62,5%
	ojos	10%	
	yo	10%	
	camisa	2,5%	

Tabla 7.2: Clases más confundidas en el proyecto de Parcheta y Martínez-Hinarejos (2017).

Si analizamos estos datos, podemos ver las clases más confundidas por la red presentan un grado de confusión de entre el 35,7% y el 52,5%. Esto supone que, las tasas de acierto obtenidas para estas clases, no difieren mucho de las obtenidas para las clases más confundidas por el modelo que conforma la solución que propone este TFG.

Sin embargo, es importante mencionar que, tanto las clases confundidas como las clases que con las que se confunde, son diferentes para cada uno de los trabajos. Esto puede deberse a la forma en la que cada proyecto trabaja con los datos y realiza la clasificación de los mismo, ya que, el de referencia se basa en un modelo estadístico y, este trabajo lo hace en un modelo de Deep Learning.

7.5. Evaluación del modelo haciendo uso de redes GRU

Como se ha explicado en el marco teórico del proyecto, concretamente en el apartado 3.3, las redes GRU, siendo similares a las LSTM, obtienen mejores resultados con conjuntos de datos pequeños. Por este motivo, se ha replicado el modelo 8 pero utilizando capas GRU en lugar de LSTM para ver si la bondad del sistema de clasificación mejoraba.

Los resultados obtenidos tras entrenar y evaluar el nuevo modelo con los mismos grupos de entrenamiento, test y validación, han sido los siguientes:

Resumen arquitectura	**Parámetros**	**Accuracy**	**Loss**
2 capas GRU con 250 neuronas + 1 capa Dropout con un rate del 25% intercalada	619841	83,52%	0,9024

Tabla 7.3: Evaluación modelo 8 implementado con redes GRU.

La cantidad de parámetros entrenables de esta nueva arquitectura de red se reduce en un 25% respecto al modelo basado en LSTMs. Por otro lado, el modelo basado en GRUs aumenta el rendimiento del antiguo modelo en un 5%. Esto no supone una gran mejora, pero sí que se trata de un modelo más eficaz para la tarea de clasificación de gestos aislados.

La matriz de confusión obtenida para este modelo es la siguiente:

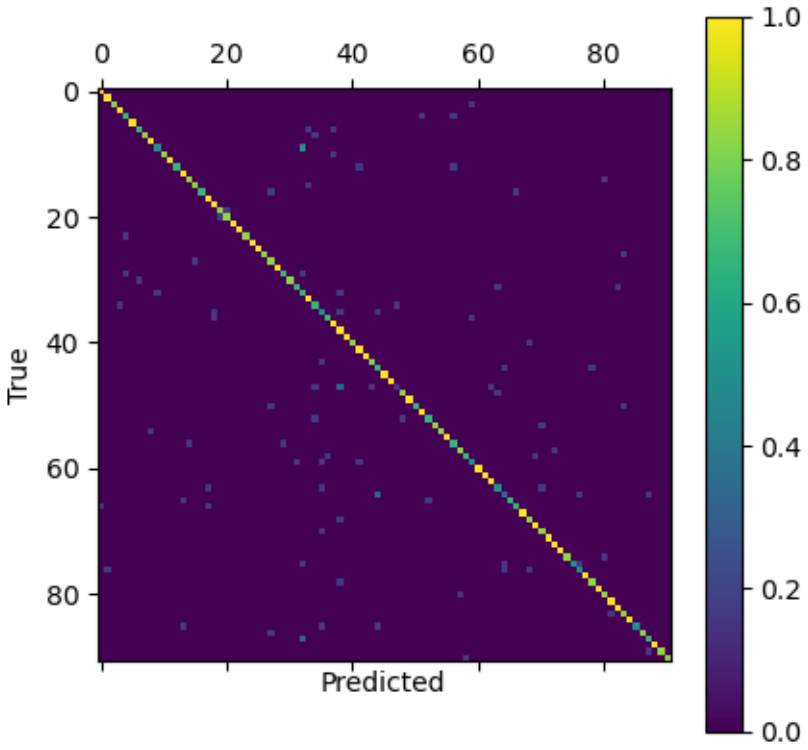


Figura 7.4: Matriz de confusión del modelo basado en redes GRU.

Si la analizamos, se obtiene que las clases más confundidas por la nueva arquitectura de red son:

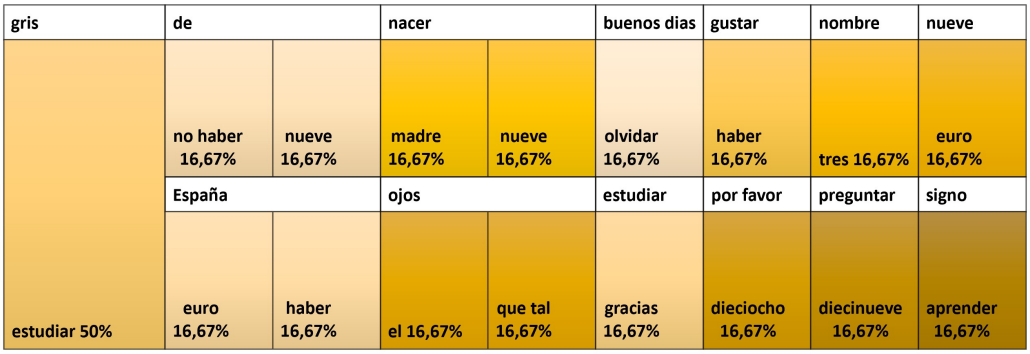


Figura 7.5: Clases más confundidas por el modelo basado en redes GRU.

Al mejorar la tasa de acierto, la cantidad de clases que confunden la red es mucho menor. Sin embargo, las tasas de acierto para estas clases se mantiene en los mismos valores que el modelo basado en LSTMs, es decir, obtenemos valores del 50%, 66,67% y 83,3% para las clases más confundidas.

Esto puede deberse a que el funcionamiento de los perceptrones, tanto LSTM como GRU, se basa en el mismo concepto de *gate unit* para manejar el flujo de información de la red y decidir qué es importante y que se descarta del mismo; por este motivo, es posible que ambas arquitecturas acaben cometiendo los mismo errores de clasificación.

8. Conclusiones

El trabajo de investigación realizado para elaborar el análisis del estado del arte en materia de reconocimiento automático de la lengua de signos ha demostrado lo útiles que resultan los modelos de Deep Learning para abordar este tipo de tareas.

Además, de la implementación de los diferentes modelos basados en RNN, se pueden desatacar varios aspectos a considerar.

Por un lado, a existencia de bibliotecas como TensorFlow o Keras, facilitan el desarrollo de todo tipo modelos basados en RNA. Esto que supone que su aplicación se puede extender, más aún, a cualquier problema que se plantee.

Centrándonos en las redes LSTM en las que se basa el proyecto, es necesario comentar el potencial que estas presentan ya que, ante una base de gestos aislados tan pequeña, han desempeñado un rendimiento más que valido sin necesidad de construir una arquitectura de red que involucrase una cantidad exagerada de parámetros. De hecho, los tiempos de entrenamiento de los diferentes modelos no han superado en ningún caso el tiempo de una hora lo que ha ayudado a poder realizar varias pruebas.

La red recurrente del modelo que representa la solución final ante el problema de reconocimiento de gestos aislados de la LSE que plantea este de este TFG (modelo 8), consta básicamente de dos capas LSTM con 250 perceptrones que, combinadas con la técnica de regularización Dropout al 25%, han conseguido un rendimiento de la red del 78,21%. De estos resultados, se puede, no sólo reafirmar el potencial de las redes LSTM, sino que también se pueden intuir modelos mucho más potentes basados en esta idea.

Algunos aspectos a tratar que podrían resultar en un modelo más eficiente podrían ser:

- Trabajar con una base de datos que contenga mayor cantidad de muestras.
- Indagar sobre las causas que provocan las confusiones entre clases mediante el análisis del estado de la red en diferentes momentos y ante diferentes secuencias de datos.
- Probar con otro tipo de redes que puedan potenciar los resultados de la tarea de clasificación; por ejemplo, redes convolucionales.

8.1. Trabajo futuro

Como línea de trabajo futura sería muy interesante crear una base de datos más completa, esto es, con más gestos de la LSE relacionados con todos los ámbitos y mayor cantidad de muestras para cada uno. Esta tarea no resulta imposible gracias a tecnologías como las mencionadas en el estado de la cuestión que expone este proyecto.

De esta manera, se podrían construir modelos de Deep Learning, cuya arquitectura de red resultaría más compleja que la propuesta en este trabajo, con datos suficientes de entrenamiento para alcanzar un rendimiento casi óptimo en la tarea de clasificación.

Además, a partir de un buen modelo capaz de reconocer casi a la perfección una cantidad importante de gestos aislados, podría implementarse una herramienta que tradujese frases más complejas capturadas directamente de los gestos de un intérprete a texto o incluso voz.

Aunque es una tarea muy compleja, conseguir una mínima implementación de esta significaría un gran avance en materia de inclusión social de las personas sordas debido a que supondría un paso hacia delante para derribar las barreras de comunicación que estas personas tienen que afrontar en su día a día.

Bibliografía

Adaloglou, N., Chatzis, T., Papastratis, I., Stergioulas, A., Papadopoulos, G. T., Zacharopoulou, V., … Daras, P. (2020). A comprehensive study on sign language recognition methods. *ArXiv*, *abs/2007.12530*.

Akmeliawati, R., Ooi, M. P.-L., y Kuang, Y. C. (2007). Real-time malaysian sign language translation using colour segmentation and neural network. En *2007 ieee instrumentation measurement technology conference imtc 2007*. Descargado de `https://ieeexplore.ieee.org/document/4258110`

Bhuvaneshwari, C., y Manjunathan, A. (2020). Advanced gesture recognition system using long-term recurrent convolution network. *Materials Today: Proceedings*, *21*. Descargado de `https://www.sciencedirect.com/science/article/pii/S2214785319322278`

Calvo, D. (2017). *Página web de carácter divulgativo sobre rna.* Descargado de `https://www.diegocalvo.es/definicion-de-red-neuronal/`

Cheok, M. J., Omar, Z., y Jaward, M. (2019). A review of hand gesture and sign language recognition techniques. *International Journal of Machine Learning and Cybernetics*, *10*, 131-153.

CNSE. (2021). *Página web sobre la comunidad sorda española.* Descargado de `http://www.cnse.es/inmigracion/index.php?option=com_content&view=article&id=58&Itemid=241&lang=es`

Jones, M. T. (2017). *artículo sobre el auge de las arquitecturas de aprendizaje profundo en el blog ibm developer.* Descargado de `https://developer.ibm.com/es/technologies/deep-learning/articles/cc-machine-learning-deep-learning-architectures/`

Keras. (2021). *Página web oficial de keras.* Descargado de `https://keras.io/`

Li, Y. (2012). Hand gesture recognition using kinect. En *2012 ieee international conference on computer science and automation engineering.* Descargado de `https://ieeexplore.ieee.org/document/6269439`

Liao, Y., Xiong, P., Min, W., Min, W., y Lu, J. (2019). Dynamic sign language recognition based on video sequence with blstm-3d residual networks. *IEEE Access*, *7*. Descargado de https://ieeexplore.ieee.org/document/8667292

Liu, H., Ju, Z., Ji, X., Chan, C. S., y Khoury, M. (2017). A novel approach to extract hand gesture feature in depth images.. Descargado de http://link.springer.com/article/10.1007%2Fs11042-015-2609-2

Mohandes, M., Aliyu, S., y Deriche, M. (2014). Arabic sign language recognition using the leap motion controller. En *2014 ieee 23rd international symposium on industrial electronics (isie)*. Descargado de https://ieeexplore.ieee.org/document/6864742

OMS. (2021). *Página web sobre sordera y pérdida de audición.* Descargado de https://www.who.int/es/news-room/fact-sheets/detail/deafness-and-hearing-loss

Oudah, M., Al-Naji, A., y Chahl, J. (2020). Hand gesture recognition based on computer vision: A review of techniques. *Journal of Imaging*, *6*. Descargado de https://www.mdpi.com/2313-433X/6/8/73

Oz, C., y Leu, M. C. (2011). American sign language word recognition with a sensory glove using artificial neural networks. *Engineering Applications of Artificial Intelligence*, *24*. Descargado de https://www.sciencedirect.com/science/article/pii/S0952197611001230

Parcheta, Z., y Martínez-Hinarejos, C.-D. (2017). Sign language gesture recognition using hmm. En L. A. Alexandre, J. Salvador Sánchez, y J. M. F. Rodrigues (Eds.), *Pattern recognition and image analysis*. Springer International Publishing. Descargado de https://github.com/zparcheta/spanish-sign-language-db

Ren, Z., Yuan, J., Meng, J., y Zhang, Z. (2013). Robust part-based hand gesture recognition using kinect sensor. *IEEE Transactions on Multimedia*, *15*. Descargado de https://ieeexplore.ieee.org/document/6470686

Saleh, Y., y Issa, G. F. (2020). Arabic sign language recognition through deep neural networks fine-tuning. *Int. J. Online Biomed. Eng.*, *16*, 71-83.

Scikit-learn. (2021). *Página web oficial de scikit-learn.* Descargado de https://scikit-learn.org/stable/index.html

TensorFlow. (2021). *Página web oficial de tensorflow.* Descargado de https://www.tensorflow.org/

Torres, J. (2019a). *Deep learning, introducción práctica con keras (segunda parte).* WHAT THIS SPACE.

Torres, J. (2019b). *Página web de carácter divulgativo sobre rnn con contenido abierto del libro "deep learning, introducción práctica con keras (segunda parte)".* Descargado de `https://torres.ai/redes-neuronales-recurrentes/`

Torres, J. (2020). *Python deep learning, introducción práctica con keras y tensorflow 2.* MARCOMBO.

Ultraleap. (2021). *Página del producto leap motion controller.* Descargado de `https://www.ultraleap.com/product/leap-motion-controller/`

Wadhawan, A., y Kumar, P. (2019). Sign language recognition systems: A decade systematic literature review. *Archives of Computational Methods in Engineering.* Descargado de `https://link.springer.com/article/10.1007%2Fs11831-019-09384-2`

Lista de Acrónimos y Abreviaturas

BPTT	Backpropagation Through Time.
CNSE	Confederación Estatal de Personas Sordas.
GRU	Gate Recurrent Unit.
INE	Instituto Nacional de Estadística.
LSE	Lengua de Signos Española.
LSTM	Long Short-Term Memory.
NLP	Natural Language Processing.
OMS	Organización Mundial de la Salud.
RNA	Red Neuronal Artificial.
RNN	Red Neuronal Recurrente.
TFG	Trabajo Fin de Grado.

A. Gestos que conforman la base de datos

GESTOS				
a	dieciocho	hola	once	tres
adiós	dieciséis	madre	padre	tu
amarillo	diecisiete	mal	perdón	universidad
apellido	diez	marrón	poder	uno
aprender	doce	morado	por favor	veinte
azul	donde	mucho	preguntar	verbos
bien	dos	muchos	que tal	verde
buenas noches	el	nacer	quince	vivir
buenas tardes	ella	negro	regular	vosotras
buenos días	enseñar	niño	repetir	vosotros
camisa	entender	no	rojo	yo
catorce	España	no haber	saber	
cinco	estudiar	nombre	seis	
color	euro	no saber	si	
cuatro	gracias	nosotras	siete	
dar	gris	nosotros	signar	
de	gustar	nueve	signo	
decir	haber	ocho	su	
de nada	hasta luego	ojos	trabajar	
diecinueve	hermano	olvidar	trece	

Tabla A.1: Gestos que conforman la base de datos

B. Códigos del proyecto

Código B.1: Código utils.py

```
import pandas as pd
import numpy as np
import os
import random
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# # A lo largo del código se han incluido diferentes ejecuciones del tipo:
# # print("[INFO])
# # para mostrar un mensaje informativo que permita controlar la ejecución del programa




# # Función para normalizar valores del fichero 'archivo' por columnas
def minmax_norm(archivo):
    return (archivo - archivo.min()) / (archivo.max() - archivo.min())



# # Función que lee todos los ficheros del dataset, los normaliza y los guarda en un nuevo directorio
# # De esta forma, una vez tenemos normalizados todos los ficheros no es necesario volver a realizar esta ↩
    ↪ operación
def save_norm():

    directorio = r"C:\Users\PC\Documents\TFG\datos\separated_gestures"
    contenido = os.scandir(directorio)

    print("[INFO] Normalizando archivos.....")

    # En cada archivo CSV del conjunto original de datos se realizan las mismas tareas:
    for fichero in contenido:

        nombre = fichero.name
        archivo = pd.read_csv(directorio + '/' + nombre, header=None)

        # Normalización de los valores
        archivo_norm = minmax_norm(archivo)

```

```
40          # Comprobación de archivos con valores erróneos (NaNs)
41          check_for_nan = archivo_norm.isnull().values.any()
42          if check_for_nan:
43              print(check_for_nan)
44              archivo_norm = archivo_norm.fillna(2) #En caso de tener algun valor NaN, este se sustituye por el valor 2
45                                                  # de forma que sea ignorado por al red
46
47          # El archivo normalizado se guarda en el nuevo directorio
48          ruta = r"C:\Users\PC\Documents\TFG\v1\separated_gestures_norm" # Este será el directorio de lectura de datos para el modelo
49          archivo_norm.to_csv(ruta + '/' + nombre, header=False, index=False)
50
51      print("[INFO] Archivos normalizados y guardados en el nuevo directorio")
52
53      return
54
55
56
57
58# # Función para crear los grupos de entrenmiento, test y validación
59# # Los nombres de los ficheros que formarán cada grupo se alamacenan en un archivo CSV diferente
60# # A la función hay que indicarle el porcentaje del total de datos (en tanto por uno) que formará cada grupo
61def groups(ntrain, ntest, nval):
62
63      # Creación diccionario clase−palabra
64      archivo_gestos = pd.read_csv(r"C:\Users\PC\Documents\TFG\v1\gestos.csv", sep=";", header=0, names=['clase', 'gesto'])
65      print("[INFO] Creando dicciionario")
66      clases = archivo_gestos['clase']
67      gestos = archivo_gestos['gesto']
68      diccionario = dict(zip(clases, gestos))
69      print("[INFO] Diccionario creado")
70
71
72      # Cantidad de ficheros tomados para conjunto
73      Ntrain = round(ntrain * 40.0)
74      Ntest = round(ntest * 40.0)
75      Nval = round(nval * 40.0)
76
77      # Definición de varibales necesarias para el manejo de datos
78      xtrain = []
79      xtest = []
80      xval = []
81      ytrain = []
82      ytest = []
83      yval = []
84
85
86      for palabra in diccionario.values():
87
```

```
88      # para cada palabra se crea una lista (fnames) con los nombre de los ficheros de las muestras de la ↩
            ↪ gesto en concreto
89      fnames = []
90      for i in range(1, 41):
91          fname = palabra + str(i) + '.csv'
92          fnames.append(fname)
93
94      # la lista resultante se desordena aleatoriamente para crear los grupos
95      # con diferentes muestras de dicha palabra cada vez que se ejecute esta función
96      random.shuffle(fnames)
97
98      # creación del archivo CSV con los nombres de los ficheros del conjunto de entrenamiento
99      for i in range(1, Ntrain):
100         xtrain.append(fnames[i])
101         ytrain.append(palabra)
102
103     y_train = pd.DataFrame(ytrain)
104     x_train = pd.DataFrame(xtrain)
105     data_train = pd.concat([x_train,y_train],axis=1)
106     data_train = data_train.sample(frac=1)
107     data_train.to_csv('traindata.csv', header=['fname', 'gesture'], index=False)
108
109     # El CSV resultante tendrá dos columnas:
110     # fname = nombre del fichero con los datos X
111     # gesture = eitqueta corespondiente a dichos datos
112     # (esto se repite para los otros conjuntos)
113
114
115     # creación del archivo CSV con los nombres de los ficheros del conjunto de test
116     for i in range(Ntrain, (Ntrain + Ntest)):
117         xtest.append(fnames[i])
118         ytest.append(palabra)
119
120     y_test = pd.DataFrame(ytest)
121     x_test = pd.DataFrame(xtest)
122     data_test = pd.concat([x_test, y_test], axis=1)
123     data_test = data_test.sample(frac=1)
124
125     data_test.to_csv('testdata.csv', header=['fname', 'gesture'], index=False)
126
127     # creación del archivo CSV con los nombres de los ficheros del conjunto de validación
128     for i in range((Ntrain+Ntest), Ntrain+Ntest+Nval):
129         xval.append(fnames[i])
130         yval.append(palabra)
131
132     y_val = pd.DataFrame(yval)
133     x_val = pd.DataFrame(xval)
134     data_val = pd.concat([x_val, y_val], axis=1)
135     data_val = data_val.sample(frac=1)
136     data_val.to_csv('validationdata.csv', header=['fname', 'gesture'], index=False)
137
138
```

```
    print("[INFO] Grupos de entrenamiento, test y validación creados")

    return



# # Función para crear y entrenar el Label Encoder con todas las categorías (gestos) que conforman el dataset
# # Label Encoder condifica cada categoría en un valor númerico entero
def label_encoder():
    gestos = pd.read_csv(r"C:\Users\PC\Documents\TFG\v1\gestos.csv", sep=";", header=0, names=['clase',↩
        ↪ 'gesto'])['gesto'].to_list()
    le = LabelEncoder()
    le.fit(gestos)

    return le # La función devuelve el codificador entrenado para poder usarlo en otras partes del código



# # Función para leer los datos X e Y de cada conjunto de datos
# # Se debe especificar que conjunto se quiere leer mediante el argumento type = train / test / validation
# # Se le pasa el codificador ya entrenado para obtener los datos Y
def read_data(type, encoder):

    ruta_datos = 'C:/Users/PC/Documents/TFG/v1/separated_gestures_norm/'

    if type == "train":
        train = pd.read_csv(r"C:\Users\PC\Documents\TFG\v1\traindata.csv", header=0, names=['fname','↩
            ↪ gesture'])

        ficheros_train = train['fname'].to_list()

        # array multidemensional de dimensinone relleno con el valor 2.0 para poder realizar padding a los ↩
            ↪ ficheros leidos
        dataX = np.full([len(ficheros_train), 101, 42], 2.0)

        for i, fichero in enumerate(ficheros_train):
            aux = pd.read_csv(ruta_datos + fichero, header=None, sep=',').to_numpy()
            dataX[i, :aux.shape[0], :aux.shape[1]] = aux # dataX contendrá los datos del fichero (aux) y un ↩
                ↪ padding de 2.0 al final

        y_train = train['gesture'].to_list()
        # se codifican las etiquetas usando el Label Encoder previamente entrenado
        # dataY contendrá la categoría codificada para cada muestra en dataX
        dataY = encoder.transform(y_train)

    if type == "test":
        test = pd.read_csv(r"C:\Users\PC\Documents\TFG\v1\testdata.csv", header=0, names=['fname', '↩
            ↪ gesture'])

        ficheros_test = test['fname'].to_list()
        dataX = np.full([len(ficheros_test), 101, 42], 2.0)
        for i, fichero in enumerate(ficheros_test):
```

```
            aux = pd.read_csv(ruta_datos + fichero, header=None, sep=',').to_numpy()
            dataX[i, :aux.shape[0], :aux.shape[1]] = aux

        y_test = test['gesture'].to_list()
        dataY = encoder.transform(y_test)

    if type == "validation":
        validation = pd.read_csv(r"C:\Users\PC\Documents\TFG\v1\validationdata.csv", header=0, names=['fname','gesture'])

        ficheros_validation = validation['fname'].to_list()
        dataX = np.full([len(ficheros_validation), 101, 42], 2.0)
        for i, fichero in enumerate(ficheros_validation):
            aux = pd.read_csv(ruta_datos + fichero, header=None, sep=',').to_numpy()
            dataX[i, :aux.shape[0], :aux.shape[1]] = aux

        y_val = validation['gesture'].to_list()
        dataY = encoder.transform(y_val)

    return dataX, dataY



# # Función para graficar la evolución de los valores de accuracy y loss de la red durante el entrenmiento
def plot_history(history):

    # Gráfica para la accuracy
    plt.figure('Model Accuracy')
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.savefig(r"C:\Users\PC\Documents\TFG\v1\graficas\Accuracy")

    # Gráfica para la loss
    plt.figure('Model Losses')
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.savefig(r"C:\Users\PC\Documents\TFG\v1\graficas\Losses")

    print("[INFO] Gráficas guardadas")

    return



```

```
237 # # Función para graficar la matriz de confusión obtenida del modelo con los valores normalizados
238 def plot_confusion_matrix(Y_true, Y_pred):
239
240     # Construcción de la matrizde confusión
241     cm = confusion_matrix(Y_true, Y_pred,
242                           normalize='true')
243
244     # Los datos de la matriz de confusión se guardan en un archivo CSV
245     pd.DataFrame(cm).transpose().to_csv('cm.csv', sep = ',')
246
247     # Graficar la matrix de confusión
248     plt.matshow(cm)
249     plt.colorbar()
250     plt.xlabel('Predicted')
251     plt.ylabel('True')
252     plt.plot()
253     plt.savefig(r"C:\Users\PC\Documents\TFG\v1\graficas\ConfussionMatrix")
254
255     return
```

Código B.2: Código RNN.py

```
import utils
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Masking, InputLayer, Dropout
from tensorflow.keras.callbacks import ModelCheckpoint



# utils.groups(0.7, 0.15, 0.15) # SOLO EJECUTAR SI SE QUIEREN RENOVAR LOS GRUPOS DE DATOS

# Crea y entrena el codificador para las etiquetas de las categorías
le = utils.label_encoder()


# Lectura de los datos

print("[INFO] Leyendo datos conjunto de entrenamiento")
X_train, y_train = utils.read_data("train", le)

print("[INFO] Leyendo datos conjunto de test")
X_test, y_test = utils.read_data("test", le)

print("[INFO] Leyendo datos conjunto de validación")
X_val, y_val = utils.read_data("validation", le)


#### CÓDIGO PARA LA COSNTRUCCIÓN DEL MODELO CON ARQUITECTURA RNN #####

features = 42 # 42 características para cada muestra de datos
steps = 101 # Número de secuencias de cada muestra de datos
categories = 91 # Categorías de nuestro problema de clasificación (91 gestos diferentes)
lstms = 150 # Cantidad de nueronas recurrentes por capa LSTM

model = Sequential()

# Capa de entrada
model.add(InputLayer(input_shape=(steps, features)))

# Capas ocultas
model.add(Masking(mask_value=2.0,
                  input_shape=(steps, features)))

model.add(LSTM(lstms, return_sequences=True))
model.add(Dropout(0.25))
model.add(LSTM(lstms))


# Capa de salida
model.add(Dense(categories, activation='softmax'))


```

```
# Compilamos el modelo
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])

# Resumen del modelo
model.summary()

# Creación del checkpoint para quedarnos con los pesos de la red que mejor accuracy ofrecen para el conjunto de validación
checkpoint_path = r"C:\Users\PC\Documents\TFG\v1\pesos\pesosoptimosmodelo"

model_checkpoint_callback = ModelCheckpoint(filepath=checkpoint_path,
                                            save_weights_only=True,
                                            monitor='val_accuracy',
                                            mode='max',
                                            save_best_only=True)



# Entrenamiento de la red
BATCH = 64
EPOCHS = 20

history = model.fit(X_train, y_train,
                    batch_size=BATCH, epochs=EPOCHS,
                    verbose=1, callbacks=[model_checkpoint_callback],
                    validation_data=(X_val, y_val))


# Cargamos los pesos guardados por el checkpoint
model.load_weights(checkpoint_path)

# Guardamos el modelo entrenado
model.save('modelo_1')
print("\n [INFO] Modelo guardado")


# Guardamos en forma de gráfica la evolución de la accuracy y la loss del modelo
utils.plot_history(history)

# Evaluamos el modelo con los datos de test
print("\n [INFO] Evaluación del modelo")
score = model.evaluate(X_test, y_test,
                       batch_size=BATCH, verbose=1)
```

Código B.3: Código modelevaluation.py

```
from tensorflow.keras.models import load_model
import utils
import numpy as np
import pandas as pd
from sklearn.metrics import classification_report

# Cargamos el codificador Label Encoder entrenado
le = utils.label_encoder()

# Leemos los datos de test
print("[INFO] Leyendo datos conjunto de test")
X_test, y_test = utils.read_data("test", le)

# Decodificación de los datos Y para obtener los resultados reales de los datos de entrada
Y_true = le.inverse_transform(y_test)

# Cargamos el modelo
model = load_model(r"C:\Users\PC\Documents\TFG\v1\modelo_1")


# Realizamos la predicción del modelo
print("\n [INFO] Calculando predicciones del modelo")
predictions = model.predict(X_test, verbose=1)

# # La posición del valor máximo de cada predicción = categoría que asigna la red a cada entrada X
# # ya que se ha usado la función de activación SOFTMAX y la red devuelve probabilidades para cada categoria
outs = np.argmax(predictions, axis=1) # lista con las probabilidades máximas de cada muestra de entrada

# # Mediante la transformada inversa del LabelEncoder vemos que gesto predice la red para cada X
Y_pred = le.inverse_transform(outs) # lista con las predicciones del modelo para cada muestra de entrada


# EVALUACIÓN DEL RENDIMIENTO DEL MODELO

# Classification report (incluye medición F1 score)
report = classification_report(Y_true, Y_pred, output_dict=True)
df_report = pd.DataFrame(report).transpose()
df_report.to_csv('report.csv', header = ['precision', 'recall', 'f1-score', 'support'], sep = ',')
print("[INFO] Classification report guardado como archivo CSV")


# Obtención de la matrix de confusión
utils.plot_confusion_matrix(Y_true, Y_pred)
```

C. Classification reports de los modelos 1 y 8

	Modelo 1			Modelo 8		
Clase	precision	recall	f1-score	precision	recall	f1-score
a	66.67%	100.00%	80.00%	75.00%	100.00%	85.71%
adiós	33.33%	16.67%	22.22%	80.00%	66.67%	72.73%
amarillo	50.00%	16.67%	25.00%	60.00%	50.00%	54.55%
apellido	75.00%	100.00%	85.71%	75.00%	100.00%	85.71%
aprender	62.50%	83.33%	71.43%	100.00%	83.33%	90.91%
azul	85.71%	100.00%	92.31%	100.00%	100.00%	100.00%
bien	83.33%	83.33%	83.33%	100.00%	66.67%	80.00%
buenas noches	75.00%	100.00%	85.71%	100.00%	100.00%	100.00%
buenas tardes	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
buenos días	66.67%	33.33%	44.44%	75.00%	50.00%	60.00%
camisa	100.00%	100.00%	100.00%	85.71%	100.00%	92.31%
catorce	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
cinco	66.67%	33.33%	44.44%	60.00%	50.00%	54.55%
color	100.00%	83.33%	90.91%	100.00%	100.00%	100.00%
cuatro	30.00%	50.00%	37.50%	83.33%	83.33%	83.33%
dar	60.00%	100.00%	75.00%	66.67%	100.00%	80.00%
de	60.00%	50.00%	54.55%	66.67%	66.67%	66.67%
decir	83.33%	83.33%	83.33%	62.50%	83.33%	71.43%
de nada	25.00%	16.67%	20.00%	57.14%	66.67%	61.54%
diecinueve	100.00%	83.33%	90.91%	100.00%	100.00%	100.00%
dieciocho	80.00%	66.67%	72.73%	100.00%	83.33%	90.91%
dieciséis	85.71%	100.00%	92.31%	100.00%	100.00%	100.00%
diecisiete	85.71%	100.00%	92.31%	100.00%	100.00%	100.00%
diez	71.43%	83.33%	76.92%	100.00%	83.33%	90.91%
doce	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

Continúa en la siguiente página

	Modelo 1			Modelo 8		
Clase	precision	recall	f1-score	precision	recall	f1-score
donde	80.00%	66.67%	72.73%	55.56%	83.33%	66.67%
dos	71.43%	83.33%	76.92%	100.00%	66.67%	80.00%
el	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
ella	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
enseñar	50.00%	50.00%	50.00%	55.56%	83.33%	66.67%
entender	100.00%	50.00%	66.67%	100.00%	66.67%	80.00%
España	38.46%	83.33%	52.63%	71.43%	83.33%	76.92%
estudiar	50.00%	83.33%	62.50%	75.00%	100.00%	85.71%
euro	85.71%	100.00%	92.31%	100.00%	100.00%	100.00%
gracias	20.00%	16.67%	18.18%	62.50%	83.33%	71.43%
gris	30.00%	50.00%	37.50%	42.86%	50.00%	46.15%
gustar	100.00%	66.67%	80.00%	66.67%	66.67%	66.67%
haber	100.00%	100.00%	100.00%	85.71%	100.00%	92.31%
hasta luego	50.00%	100.00%	66.67%	75.00%	100.00%	85.71%
hermano	85.71%	100.00%	92.31%	100.00%	100.00%	100.00%
hola	40.00%	66.67%	50.00%	57.14%	66.67%	61.54%
madre	66.67%	66.67%	66.67%	100.00%	100.00%	100.00%
mal	85.71%	100.00%	92.31%	85.71%	100.00%	92.31%
marrón	66.67%	33.33%	44.44%	100.00%	50.00%	66.67%
morado	60.00%	50.00%	54.55%	75.00%	50.00%	60.00%
mucho	80.00%	66.67%	72.73%	50.00%	50.00%	50.00%
muchos	83.33%	83.33%	83.33%	85.71%	100.00%	92.31%
nacer	100.00%	33.33%	50.00%	42.86%	50.00%	46.15%
negro	100.00%	33.33%	50.00%	100.00%	50.00%	66.67%
niño	75.00%	100.00%	85.71%	85.71%	100.00%	92.31%
no	40.00%	33.33%	36.36%	80.00%	66.67%	72.73%
no haber	75.00%	100.00%	85.71%	75.00%	100.00%	85.71%
nombre	100.00%	83.33%	90.91%	62.50%	83.33%	71.43%
no saber	40.00%	33.33%	36.36%	100.00%	66.67%	80.00%
nosotras	100.00%	66.67%	80.00%	83.33%	83.33%	83.33%
nosotros	100.00%	100.00%	100.00%	85.71%	100.00%	92.31%
nueve	80.00%	66.67%	72.73%	50.00%	66.67%	57.14%

Continúa en la siguiente página

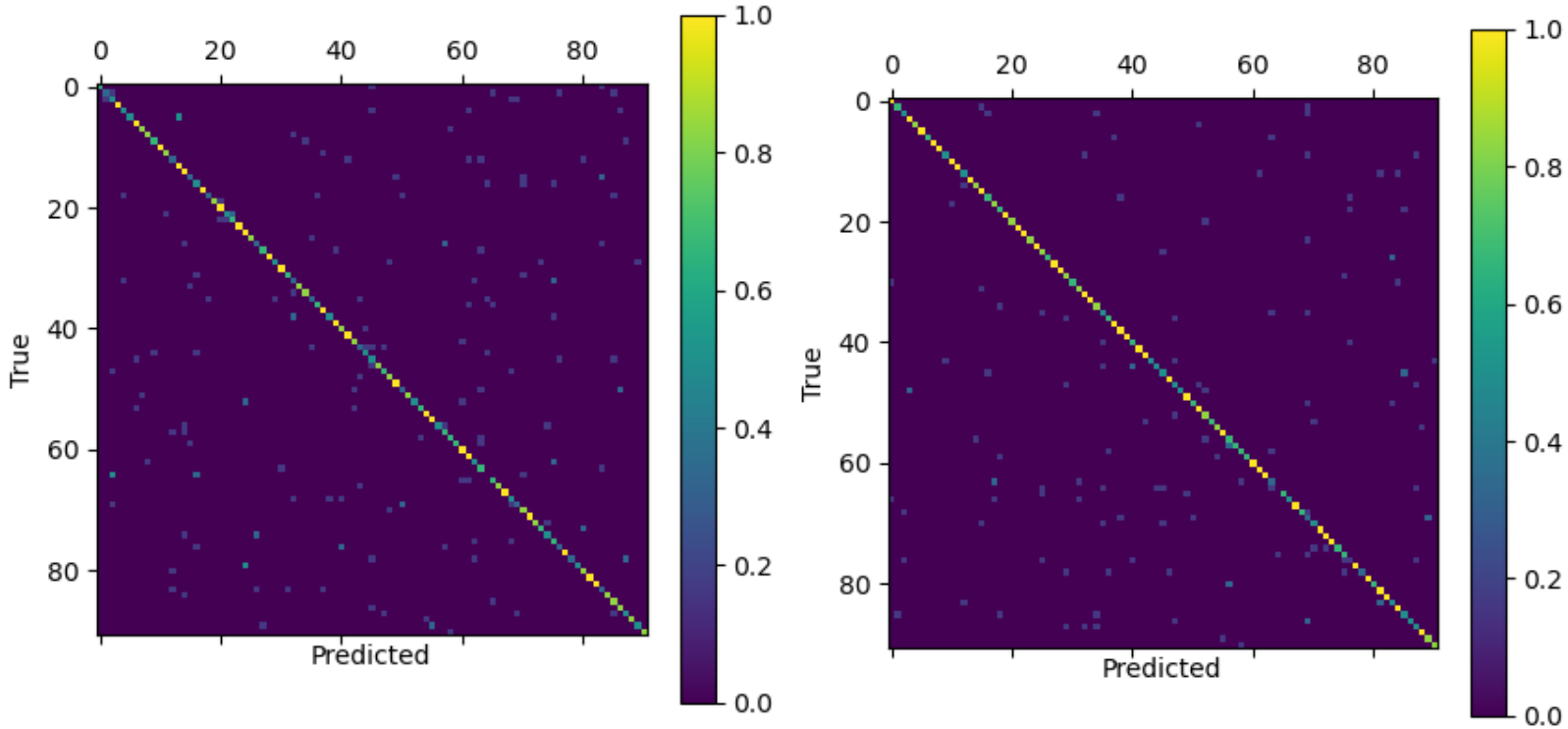
	Modelo 1			Modelo 8		
Clase	precision	recall	f1-score	precision	recall	f1-score
ocho	100.00%	83.33%	90.91%	100.00%	66.67%	80.00%
ojos	80.00%	66.67%	72.73%	66.67%	66.67%	66.67%
olvidar	100.00%	33.33%	50.00%	100.00%	66.67%	80.00%
once	100.00%	83.33%	90.91%	100.00%	100.00%	100.00%
padre	83.33%	83.33%	83.33%	85.71%	100.00%	92.31%
perdón	50.00%	83.33%	62.50%	100.00%	100.00%	100.00%
poder	66.67%	33.33%	44.44%	33.33%	33.33%	33.33%
por favor	25.00%	16.67%	20.00%	0.00%	0.00%	0.00%
preguntar	50.00%	50.00%	50.00%	100.00%	66.67%	80.00%
que tal	80.00%	66.67%	72.73%	100.00%	50.00%	66.67%
quince	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
regular	66.67%	66.67%	66.67%	100.00%	66.67%	80.00%
repetir	16.67%	16.67%	16.67%	16.67%	33.33%	22.22%
rojo	33.33%	16.67%	22.22%	60.00%	50.00%	54.55%
saber	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
seis	62.50%	83.33%	71.43%	85.71%	100.00%	92.31%
si	85.71%	100.00%	92.31%	100.00%	100.00%	100.00%
siete	75.00%	50.00%	60.00%	100.00%	66.67%	80.00%
signar	100.00%	33.33%	50.00%	57.14%	66.67%	61.54%
signo	25.00%	16.67%	20.00%	25.00%	16.67%	20.00%
su	85.71%	100.00%	92.31%	100.00%	100.00%	100.00%
trabajar	57.14%	66.67%	61.54%	100.00%	33.33%	50.00%
trece	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
tres	57.14%	66.67%	61.54%	100.00%	66.67%	80.00%
tu	66.67%	100.00%	80.00%	66.67%	100.00%	80.00%
universidad	100.00%	83.33%	90.91%	100.00%	100.00%	100.00%
uno	50.00%	50.00%	50.00%	50.00%	50.00%	50.00%
veinte	85.71%	100.00%	92.31%	75.00%	100.00%	85.71%
verbos	50.00%	33.33%	40.00%	50.00%	50.00%	50.00%
verde	71.43%	83.33%	76.92%	100.00%	50.00%	66.67%
vivir	33.33%	33.33%	33.33%	42.86%	50.00%	46.15%
vosotras	66.67%	100.00%	80.00%	85.71%	100.00%	92.31%

Continúa en la siguiente página

	Modelo 1			**Modelo 8**		
Clase	precision	recall	f1-score	precision	recall	f1-score
vosotros	83.33%	83.33%	83.33%	71.43%	83.33%	76.92%
yo	83.33%	83.33%	83.33%	83.33%	83.33%	83.33%
accuracy		**70.71%**			**78.21%**	
macro avg	72.42%	70.51%	69.27%	80.34%	78.21%	77.81%
weighted avg	72.42%	70.51%	69.27%	80.34%	78.21%	77.81%

Tabla C.1: Classification reports modelos 1 y 8

D. Figura 7.2 ampliada



(a) Matriz de confusión modelo 1

(b) Matriz de confusión modelo 1

Figura D.1: Ampliación Figura 7.2