Facultad
de
Ciencias

DEEP LEARNING APLICADO A ESPECTROSCOPIA LÁSER
(Deep learning applied to laser spectroscopy)

Trabajo de Fin de Grado
para acceder al

GRADO EN FÍSICA

Autor: Pedro Corral Saiz

Director: Adolfo Cobo García

Febrero – 2022

Trabajo parcialmente realizado con los medios aportados por el Proyecto PID2019-107270RB-C21, financiado por el MCIN/ AEI /10.13039/501100011033

Agradecicimientos

En primer lugar me gustaría agradecer a D. Adolfo Cobo por la posibilidad
de realizar este trabajo, su dedicación y sobre todo
por su paciencia a lo largo de todo el proceso.

Agradecer a familia, a los que están y los que ya no, y amigos.
A mis padres por darme la oportunidad de realizar este grado en física.

A mi abuelo.

Resumen

La técnica de espectroscopía de ruptura inducida por láser (LIBS) es una potente vía de análisis de composición atómica y molecular de materiales. En este trabajo se han implementado modelos de aprendizaje profundo para el análisis de dichos espectros LIBS. Se busca que estos modelos basados en redes neuronales sean capaces de realizar tareas de clasificación y detección de anomalías.

El trabajo ha sido desarrollado en el entorno Google Colaboratory debido a su facilidad y escasa necesidad de configuración previa y el sistema de ejecución en la nube, basándonos en la librería Keras dentro de TensorFlow para la implementación de los modelos.

Se ha desarrollado un modelo clasificador con una capa de salida con activación softmax para la clasificación de alimentos y un modelo de autoencoder para la detección de anomalías en espectros medidos en conchas de moluscos.

Se realiza un proceso de optimización o 'tuneo' de los hiperparámetros de la red para mejorar su desempeño mediante la herramienta Optuna de libre acceso. Técnicas de reducción de *overfitting* también han sido implementadas. Posteriormente se realiza un análisis de cómo varia el desempeño en función del valor de dichos hiperparámetros.

Finalmente se presentan los resultados en los que el modelo clasificador es capaz de detectar correctamente la procedencia de las patatas en el proceso de validación en un $\sim 92\,\%$ de los casos, mientras que el modelo de autoencoder descubrimos que no es capaz de detectar todos los tipos de anomalías que le presentamos (ver tabla 6.2).

Palabras clave: autocodificacor, clasificador, softmax, aprendizaje profundo, hiperparámetros.

Summary

Laser-induced breakdown spectroscopy (LIBS) technique is a powerful way of analyzing the atomic and molecular composition of materials. In this proyect, deep learning models have been implemented for the analysis of these LIBS spectra. It is intended that these models based on neural networks will be capable of performing classification and anomaly detection tasks.

The proyect has been developed in the Google Colaboratory environment due to its simple and little need for prior configuration and execution system in the cloud, based on Keras library in TensorFlow for the implementation of the models.

A classifier model has been developed with an output layer with softmax activation for the clasification task of food and an autoencoder model for the detection of anomalies in spectra measured in mollusk shells.

An optimization process or hiperparameter tuning of the network is carried out to improve its performance through the free access tool Optuna. Overfitting reduction techniques have also been implemented. Then, an analysis is made of how the performance depends on the value of these hyperparameters.

Finally, the results are shown in which the classifier model is able to correctly detect the origin of the potatoes in the validation process in $\sim 92\,\%$ of the cases, while we found that the autoencoder model is not capable of detecting all types of anomalies that we present to it (see table 6.2).

Key words: autoencoder, classifier, softmax, deep learning, hiperparameter.

Índice general

Capítulo 1

Introducción

"Por miles de años hemos tratado de entender cómo pensamos. El campo de la inteligencia artificial (IA) va más allá, no solo trata de entender si no también crear agentes inteligentes" [1].

1.1. Contexto

La espectroscopía láser es una técnica que permite averiguar la composición química de materiales de forma sencilla. En particular, la espectroscopía láser LIBS (Laser induced breakdown spectroscopy) ofrece información a niveles atómicos debido a la emisión óptica del plasma generado sobre una muestra con un láser pulsado de gran energía y en los últimos años viene colocándose como una técnica en crecimiento en nº de artículos publicados y en patentes registradas [3].

Dicha emisión es captada y mediante un espectrómetro se obtienen las líneas de emisión características de los diferentes elementos presentes en la muestra. Sin embargo, estos espectros no son interpretables de forma inmediata y requieren de un proceso de análisis para obtener la información de presencia o concentración de los diferentes elementos químicos.

Se han propuesto diferentes técnicas quimiométricas como la regresión PLS (partial least square) o LDA (linear discriminant analysis), y otras basadas en inteligencia artificial, que pueden arrojar soluciones interesantes para dicho propósito.

1.2. Objetivos

En este trabajo planteamos como objetivo la implementación de modelos de aprendizaje máquina (Machine Learning) y más en concreto redes de aprendizaje profundo (Deep Learning) para el análisis automático de espectros LIBS con el fin de extraer información de dichos espectros de forma automatizada sin precisar de un análisis complejo.

Por un lado, se realizarán medidas de distintas variedades de patatas (solanum tuberosum) buscando posteriormente clasificarlas en función de su origen.

Posteriormente, a partir de datos ya medidos de moluscos, en concreto lapa común (patela vulgata) tratamos de implementar un modelo de autoencoder que permita detectar las diferentes anomalías que se presentan en función de la zona de medición o presencia de agentes contaminantes no esperados.

Cabe destacar que no buscamos en este trabajo realizar análisis precisos de la composición ni de las patatas ni de los moluscos, nos centraremos en implementar diferentes modelos que sean capaces de discriminar información oculta (al menos a golpe de vista) y relevante en los espectros LIBS.

1.3. Estructura del documento

Tras este capítulo de introducción resumimos el contenido de los capítulos restantes de la memoria:

Capítulo 2 Descripción básica de la técnica de espectróscopía de LIBS, esquema experimental y de los datos recopilados.

Capítulo 3 Introducción al campo del Machine Learning y más en concreto a los algoritmos de Deep Learning.

Capítulo 4 Desarrollo, implementación y optimización del modelo clasificador de las patatas en función de su origen.

Capítulo 5 Análisis de los resultados obtenidos en el Capítulo 4, de las métricas y desempeño del modelo.

Capítulo 6 Implementación y análisis del modelo de aprendizaje no supervisado (autoencoders) para la detección de anomalías en los espectros tomados en las conchas de moluscos.

Capítulo 7 Conclusiones del proyecto y posibles alternativas o mejoras de los modelos implementados.

Capítulo 2

LIBS

El estudio de espectros de emisión atómicos, permite detectar la presencia de determinadas especies en una muestra. Existen numerosos métodos para obtener dichos espectros con diferentes fuentes de energía y colectores de la radiación generada, procesos elásticos, inelásticos...

La metodología LIBS surge dentro de este marco para el análisis químico de materiales mediante la colecta de dichos espectros de emisión. Su acrónimo en inglés (Laser induced breakdown spectroscopy), trata de una técnica que emplea el láser de estado sólido como fuente de energía.

La primera publicación de LIBS de la que hay constancia se reportó a inicios de los años 60 (1962), donde *Bench y Cross* emplean láser para inducir plasma sobre una superficie. En los siguientes años se publican los primeros análisis de muestras con dicha técnica.

No es hasta mediados de los años 90 cuando LIBS gana en popularidad y con el desarrollo y reducción en costes del desarrollo de potentes láseres se ve incrementado exponencialmente tanto artículos como patentes registradas [3].

Técnica revolucionaria, aún en desarrollo, cuenta con numerosas ventajas respecto a otras. Permite el análisis de muestras en cualquier estado (sólidos, líquidos y gas). Es muy versátil, permite el análisis de muestras in-situ ya que el material e instrumentación desarrollado permite su transporte. Por otra parte, no requiere en la mayor parte de los casos de una preparación previa de la muestra lo cuál hace que los resultados no se vean afectados en ese sentido y eso unido a la inmediatez de los resultados hace de esto una de sus principales ventajas.

Por otra parte, podemos destacar que es una técnica aún en desarrollo y no destaca por su precisión ($\sim$10 %) [3], entonces podemos decir que la hace perfecta para áreas de estudio donde queremos realizar un análisis cualitativo, análisis de muestras reales con especies elementales o como nuestro problema de clasificación donde no requerimos de un análisis detallado y lo más exacto posible de su composición química.

2.1. Aspectos generales

LIBS es una técnica de espectroscopia de emisión atómica (AES) donde empleamos como fuente pulsos láser de muy alta energía. Los fotones son absorbidos por los átomos de la muestra haciendo que estos se exciten a estados de energía más altos. Sabemos que los electrones tienden a ocupar los estados de menor energía y transitan desde este estado excitado a la configuración de menor energía emitiendo en la transición fotones [4], que son característicos para cada tipo de elemento.

2.1.1. Formación del plasma

Cuando la energía aplicada sobre los electrones es suficientemente grande, los átomos son ionizados y tenemos los electrones libres y los iones positivos, conformando el plasma.

El láser se focaliza sobre una región minúscula, con pulsos típicamente del orden del nanosegundo, alcanzando temperaturas extremadamente altas ($\sim 10^4 - 10^6$ K dependiendo de la presión ambiente [3])

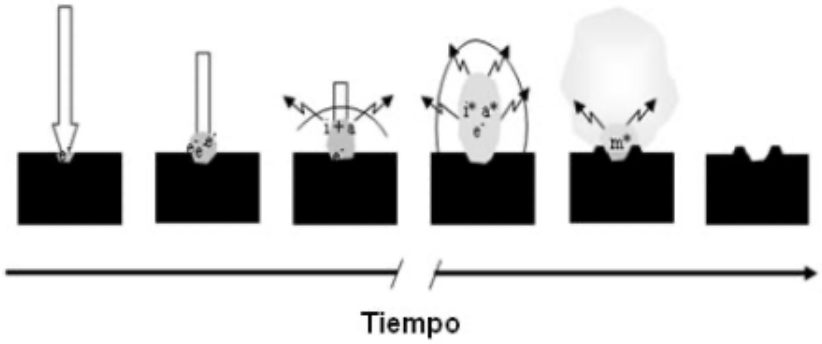


Figura 2.1: Etapas de evolución del plasma. De izquierda a derecha: irradiamos la muestra con el láser, esta comienza a ionizarse liberando electrones y los iones cargados positivamente lo cuál conforma el plasma. Posteriormente este se expande rápidamente. Cuando el pulso acaba, comienza la etapa de desexcitación, se enfría hasta que el plasma desaparece. Finalmente podemos ver la muestra ablacionada. Imagen sacada de [3]

Típicamente en LIBS se tienen plasmas débilmente ionizados [3] donde los electrones suponen menos de un 10 % de las especies presentes en el mismo. Cuando el pulso láser finaliza los electrones pierden energía rápidamente hasta la desaparición del plasma quedando partículas en suspensión producto de la ablación de la muestra.

Junto con las líneas de emisión de los elementos que toman parte tenemos típicamente radiación continua debida a dos procesos [4]. La radiación bremsstrahlung causada por la perdida de energía cinética debido a la deceleración del electrón al acercarse al ión positivo y por otro lado la recombinación radiativa.

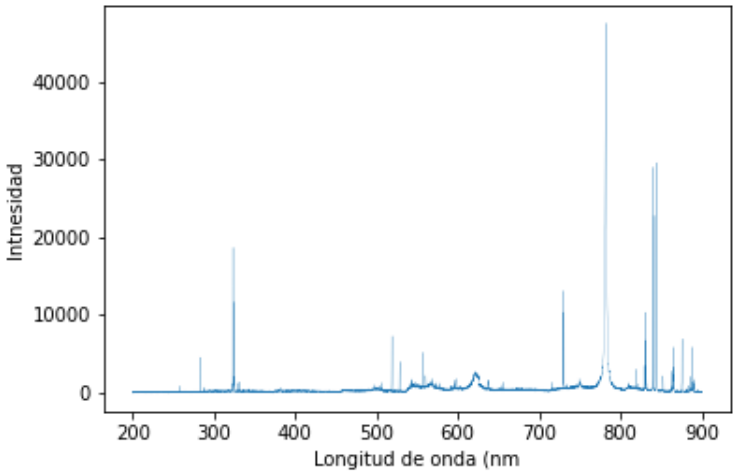


Figura 2.2: Ejemplo de espectro LIBS típico. Aparecen picos característicos a diferentes longitudes de onda. En torno al pico situado a unos 780nm podemos apreciar una "joroba" característica de estos espectros de emisión que se trata de radiación continua principalmente de radiación bremsstrahlung.

2.2. Dispositivo y mediciones

El esquema del dispositivo empleado [1] (estándar de LIBS) se muestra en la figura 2.3

[1] El dispositivo experimental es complejo y no se explica con completo detalle. Ver [3]

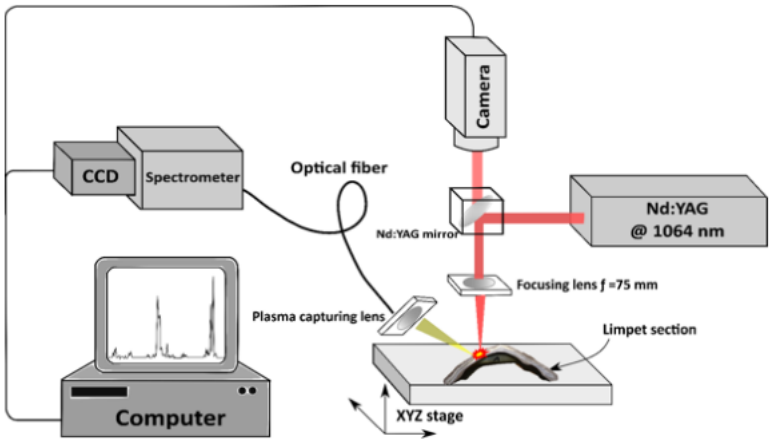


Figura 2.3: Esquema experimental empleado para la obtención de los espectros empleados. El láser de Ng:YAG como fuente de energía, una lente focalizadora, una lente colectora unida con fibra óptica a un espectrómetro que a su vez conecta con una computadora para el procesado de los datos.

Láser de Nd: YAG Lotis LS-2134D de 1064 nm de longitud de onda empleado para la generación del plasma. Este tipo de láser es el más común en LIBS [3] y trata de un cristal de Itrio y Aluminio $Y_3Al_5O_{12}$ dopado con neodimio ($\sim 1\,\%$ en peso). 35mJ de energía de pulso. Doble pulso, trata de una técnica de realizar un segundo pulso para mejorar e inducir una mayor excitación sobre la muestra. Nuevos parámetros han de considerarse como el retardo entre pulsos.

Sistema de lente/espejos empleado para focalizar el haz láser y dirigirlo a la muestra. La distancia focal de la lente focalizadora es de 75nm. La cámara ayuda con la calibración.

Espectrómetro ULS2048-USB2-RM de 8 canales. Se emplea esencialmente para separar la luz colectada en sus diferentes componentes. Ver sección 2.2.1 para la el rango de longitud de onda obtenido. La unidad CCD transforma en una señal eléctrica que puede enviarse a la computadora.

Computadora y software Se tiene conectado una computadora capaz de controlar la acción del láser, los diferentes parámetros del dispositivo mediante MATLAB. Los espectros son guardados y en su caso etiquetados para su posterior análisis.

2.2.1. Datasets obtenidos

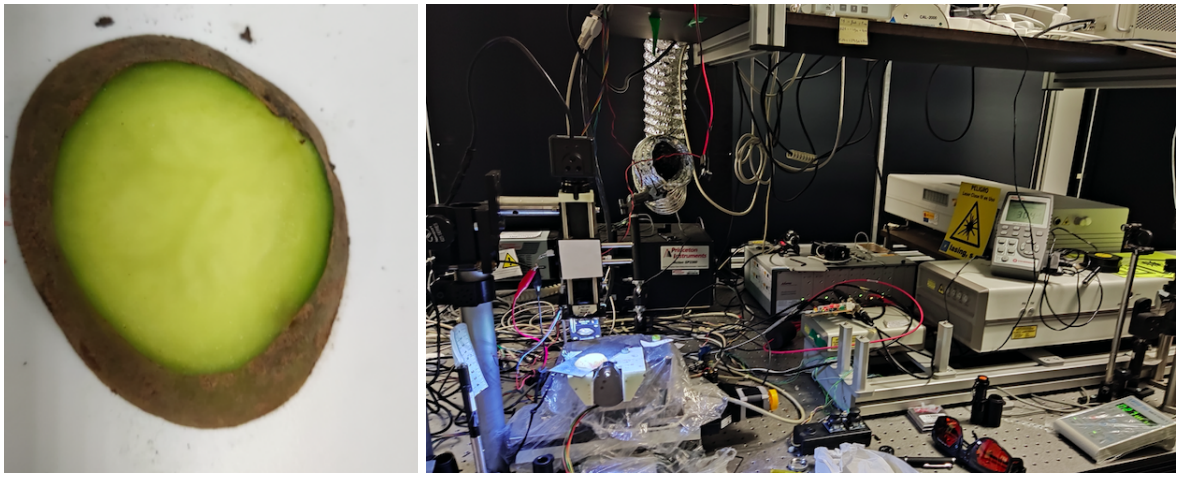
Inicialmente se midieron espectros de una variedad de 10 patatas de distinta procedencia. Las patatas fueron secadas para mejorar la calidad de los mismos y el dispositivo calibrado mediante una primera medición para eliminar ruido de fondo posteriormente de forma computacional.

Los datos son etiquetados según la variedad de patata con números del 1 al 10, esto es realizado de forma automática mediante MATLAB. Los *datasets* son exportados en formato .csv, muy común en este tipo de aplicaciones. El valor en la primera columna hace referencia a la etiqueta de esa muestra y en este caso carece de encabezado. Las longitudes de onda que analiza el espectrómetro se exportaron en otro .csv. Cada espectro cubre 15211 longitudes de onda.

Etiqueta	Variedad de patata	Procedencia	Nº de espectros
#1	Lucinda	-	214
#2	Variedad2	-	210
#3	Variedad3	Andalucía	213
#4	Spunta	Galicia	215
#5	Variedad4	-	213
#6	Monalisa	Envasado Vitoria Udapa	212
#7	Farida	-	214
#8	Colomba	-	216
#9	Variedad9	Huerto Cantabria	214
#10	Variedad10	Huerto Cantabria (verde)	205

Tabla 2.1: Resumen del *dataset* obtenido de los espectros de las patatas con las etiquetas, la variedad de patata y el número de espectros de cada patata.

Los espectros de cada tipo son medidos de la misma patata, configurando el dispositivo experimental para que el láser focalice sobre la superficie y vaya desplazándose a lo largo de la misma. Por otra parte, como destacábamos al principio del capítulo las muestras no requieren apenas de una preparación previa, en este caso un secado y seccionar la patata para poder focalizar el láser directamente en el cuerpo de la misma.



(a) muestra (b) medición

Figura 2.4: (a) Imagen de la patata de la que se obtienen las muestras de la variedad10. (b) imagen del dispositivo experimental durante las mediciones.

Por otro lado se tomaron espectros de una dataset de lapas medidos externamente al trabajo. En este nuevo conjunto de datos tenemos al igual que las patatas, las lapas etiquetadas con un número del 1 al 10 en función del tipo. El formato es similar al caso anterior.

Al tratarse de anomalías, tenemos un dataset desbalanceado dónde encontramos que hay significativamente más datos de tipo "normal".

Etiqueta	Variedad	Nº de espectros
#1	normal	1812
#2	normal	1501
#3	normal	1485
#4	normal	635
#5	resina	35
#6	resina	22
#7	silicio raro	72
#8	ratio bajo	120
#9	normal	1822
#10	ápice	838

Tabla 2.2: Descripción del dataset de espectros de lapas. Tenemos etiquetados los datos normales y por otro lado el resto son catalogados como anomalías, que son las muestras que el modelo sea capaz de detectar de forma automática.

Las etiquetas catalogadas como "normal"(#1, #2, #3, #4 y #9) hacen referencia a espectros donde consideramos correcto el ratio carbonato de calcio y magnesio. Las "resina"(#5 y #6) son espectros contaminados con resina epoxi[2], empleada en los bordes de la lapa para facilitar su corte antes de la medición. Las etiqueta de "silicio raro"(#7) hace referencia a espectros en los que se ha detectado emisión del silicio, que a priori no debería aparecer salvo que la lapa lo haya asimilado. Los espectros "ratio bajo"(#8) son aquellos que presentan una cantidad anómalamente baja de magnesio y no se conoce el por qué. Por último, los espectros "ápice"(#10) son espectros tomados en la punta de la lapa, donde el material es ligeramente diferente y el carbonato de calcio ($CaCO_3$) se forma como cristalización del aragonito y no de calcita.

[2] `https://es.wikipedia.org/wiki/Resina_epoxi`

Capítulo 3

Machine Learning

Dentro del campo de la inteligencia artificial se encuentran el conjunto de técnicas conocidas como machine learning o aprendizaje automático. Consiste en un cambio de paradigma donde tratamos de que la máquina pueda aprender sin ser explícitamente programada entendiendo aprendizaje como la generalización de conocimiento a partir de un conjunto de experiencias. Es un campo propio, altamente amplio que se relaciona con los demás campos dentro de la inteligencia artificial.

Trata de a partir de un conjunto de datos que le proveemos, predecir, clasificar, detectar patrones.... mediante un modelo que en esencia es un algoritmo matemático.

Numerosos autores hablan de capacidad de aprender imitando la capacidad humana, donde la unidad básica es la neurona. El perceptron LTU (linear threshold unit) es la unidad básica sobre la cuál se forman modelos más complejos.

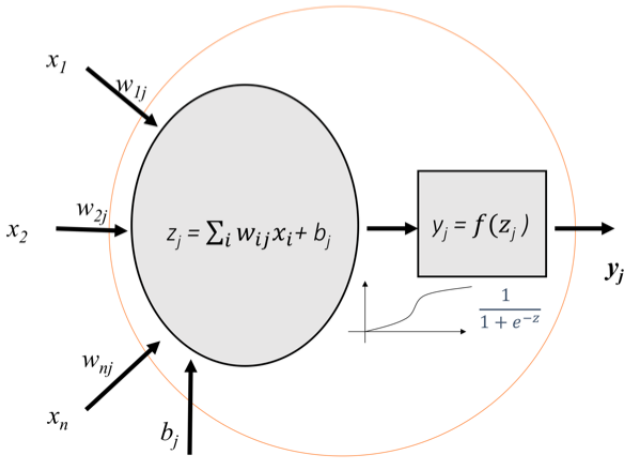


Figura 3.1: Esquema de una neurona artificial, el subídice j hace referencia la las j neuronas de la red que presentaremos más adelante en esta sección. perceptron. Imagen obtenida de: Torres, J. (2018). Neurona artificial [Gráfico]. Jordi TORRES.ai. https://torres.ai/deep-learning-inteligencia-artificial-keras/

Una vez disponemos de la entrada (generalmente es inicializada de forma aleatoria) la salida de la neurona se genera mediante:

$$z = \sum_i w_i x_i + b w_0 \tag{3.1}$$

donde w_i se le denomina pesos y b el sesgo o *bias*, todos ellos son los parámetros que la red debe ajustar durante el entrenamiento.

z tiene una forma lineal, lo cuál es limitante para resolver ciertos problemas. A esta salida se le aplica una cierta función que le distorsiona de alguna forma para poder enfrentar problemas no lineales. A esta función se le conoce como función de activación y en posteriores secciones conoceremos más sobre ella.

Finalmente tenemos y que es la salida de nuestra neurona. Esencialmente la neurona se podría ver y gracias a la facilidad de software que encontramos a la hora de programar como un sistema que le

metemos un input, internamente realiza una serie de operaciones y nos ofrece un output que idealmente será útil para nuestro problema.

A partir de esta neurona básica surgen modelos más complejos donde usamos varias neuronas para resolver problemas más complejos llegando al concepto de redes neuronales donde tenemos las neuronas conectadas entre si, en una o varias capas (conectamos el output de una con el input de la otra). Este concepto permite aprender de forma jerarquizada conceptos cada vez más complejos a medida que profundizamos en la red. La tendencia es que estos modelos cada vez constan de más y más capas, ya que necesitamos y buscamos resolver cada vez mediante estas técnicas problemas más complicados. A estos modelos les llamamos modelos de Deep Learning o aprendizaje profundo.

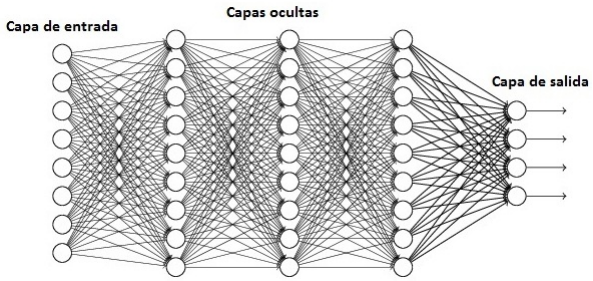


Figura 3.2: Esquema básico de una red neuronal de aprendizaje profundo. El nº de neuronas y el nº de capas ocultas aumenta o disminuye en función de la complejidad de nuestros datos. (Imagen de Adrián Hernández. `https://mlearninglab.com/2017/09/20/por-que-el-deep-learning-funciona-tan-bien/`).

3.1. Proceso de entrenamiento

Hasta ahora hemos introducido la idea de neurona artificial y brevemente el concepto de redes neuronales, donde aparecían los parámetros w_i y b. El proceso de entrenamiento de la red consiste ajustar dichos parámetros.

Tenemos tres tipos de aprendizaje en función de la arquitectura de la red, los datos que tenemos y la forma en que la red es entrenada [5]:

Aprendizaje supervisado busca descubrir la relación que existe entre las variables de entrada y las variables de salida que tenemos (*labels* o etiquetas) y se las mostramos a la red, de forma que ajuste los parámetros para que con los datos etiquetados la salida se aproxime a dichos valores de etiqueta.

Aprendizaje no supervisado consiste en producir a partir exclusivamente de los datos de entrada sin incluir las etiquetas buscando patrones. PCA, clusterización o autoencoders son algunos ejemplos de este tipo de aprendizaje.

Aprendizaje por refuerzo aprende esencialmente a base de prueba y error, de forma que se le premia o penaliza en función de si la evolución es favorable o no[1].

El proceso de ajuste de los parámetros en aprendizaje supervisado se realiza mediante el algoritmo de *backpropagation* basado en la técnica de descenso del gradiente (GD) [6] que consiste en obtener el gradiente de la función de pérdida o *loss* para saber la dirección en la que debemos variar los parámetros para poder disminuir dicha función de pérdida. Se divide en tres etapas:

1. *Forwardpropagation* introducimos los datos en la red y la "cruzan" de tal forma que la red obtiene unas predicciones para esos datos.

2. *Cálculo de la función de pérdida* Una vez obtenidas dicha predicción, podemos estimar el valor de la función de *loss* (ver cap: 3.2), que es una de las métricas que se emplean para la evaluación de la red, y nos da una idea de cuanto se aproximan las predicciones de la red a las etiquetas que tenemos. La idea debe ser la de reducir el valor de la función de *loss* idealmente hasta 0.

[1] Este tipo de aprendizaje no ha sido usado a lo largo del trabajo.

3. *Backward propagation* Es el punto clave de *backpropagation* y antes de existir este algoritmo, los pesos se ajustaban mediante fuerza bruta calculando los gradientes para cada una de las posibles conexiones ya que una variación en un peso en la capa 1 puede afectar a los de las siguientes capas y así sucesivamente. En los modelos de deep learning esto cada vez resultaba más complicado con incrementos enormes en los tiempos de computación. En este momento se analizan los errores en sentido inverso al *forwardpropagation* [6] y es basado en que el error de cada capa depende directamente del error de las capas anteriores. Cada neurona se le asigna una responsabilidad (porcentaje del error) que sirve a la red para saber cuanto ha de variar los parámetros de dicha neurona.

3.2. Hiperparámetros de la red

Una vez presentado un esquema básico de en qué consisten las redes neuronales y de como estas aprenden, vamos a centrarnos en el proceso de aprendizaje en sí, de como le pasamos los datos a la red, velocidad con la que aprende...

3.2.1. Función de activación

Hemos hablado en la sección anterior (3.1) sobre la función de activación, de cómo esta es empleada a la salida de la neurona para introducir deformaciones o no linealidad. Dentro de la librería Keras tenemos implementadas varias funciones de activación. Aquí presentaremos las empleadas en nuestro modelo aunque hay más, con diferentes características y aplicaciones.

ReLU (unidad lineal rectificada) Se trata de una función que al aplicarle entrada por debajo de 0, la salida será nula y si la entrada es mayor que cero la respuesta es lineal de forma:

$$f(x) = \begin{cases} x & si \quad x \geq 0 \\ 0 & si \quad x < 0 \end{cases} \tag{3.2}$$

Sigmoid muy útil para convertir valores en cualquier rango en probabilidad entre 0 y 1 [6]. Para valores altos negativos o altos positivos vemos que la salida de la función se encuentra próxima a 0 y 1 respectivamente.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

Softmax uno de los usos más empleados de esta función es la de colocarla en la capa de salida de un clasificador como veremos en la siguiente sección. Nos retorna en este uso la probabilidad (P) de las diferentes i salidas en base a sus evidencias (x) [5].

$$P(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.4}$$

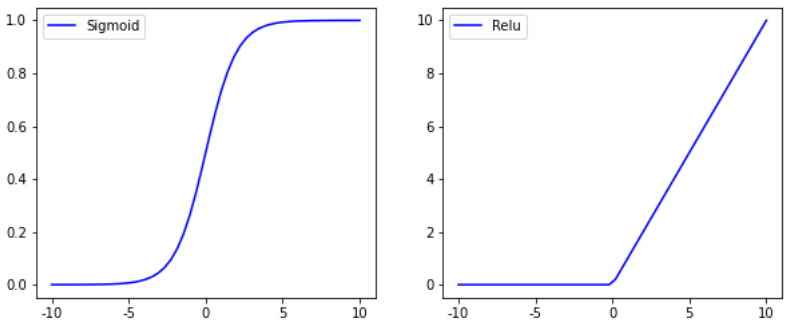


Figura 3.3: Representación gráfica de la funciones ReLU y sigmoide según las ecuaciones 3.2 y 3.3

3.2.2. Función de pérdida

Se trata de una de las métricas por excelencia en las redes neuronales y nos da una idea de cómo funciona nuestra red. En aprendizaje supervisado, tratamos de reducir la función de *loss* o pérdida de forma que la predicción de nuestra red converja con las etiquetas de los ejemplos que disponemos.

Idealmente trataríamos de minimizar la función de pérdida hasta 0, de forma que el modelo se ajuste perfectamente a los datos de entrenamiento pero por otro lado puede resultar que la red se sobreentrene, aparece aquí el concepto de *overfitting*, consiste esencialmente en que la red es capaz de predecir los datos de entrenamiento muy bien pero cuando le introducimos un dato con el que no ha sido entrenado falla o al menos baja notablemente su desempeño.

De nuevo para la función de *loss*, cómo para la función de activación, existen numerosas opciones que emplearemos dependiendo el tipo de datos, la aplicación de la red... Introducimos aquí los empleados durante el trabajo.

Categorical Crossentropy empleada típicamente en problemas de clasificación debido a su gran desempeño.

$$Loss = -\sum_i y_i log(p_i) \tag{3.5}$$

donde y_i es el valor real y p_i la predicción. El signo - aparece de forma que cuando el valor real y el predicho por la red se acercan, la función de *loss* toma valores más pequeños [8].

Debemos presentarle los valores de etiqueta en forma de vector [9], técnica que conocemos con el nombre de *One hot encoding* y consiste en transformar el valor de etiqueta en un vector de ceros salvo un uno en la posición de valor de la etiqueta:

$$Etiqueta\, 2 \rightarrow (0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0) \tag{3.6}$$

Error cuadrático medio (MSE) emplearemos esta función en el modelo de autoencoder, con la premisa de que las diferencias de predicción en valores altos, maximiza su importancia y en valores pequeños los minimiza.

$$MSE = \frac{1}{n}\sum_i^n (x_i - x_{pi})^2 \tag{3.7}$$

donde n será un contador de las diferentes longitudes de onda, x_i la intensidad para cada longitud de onda y x_{pi} las predicciones de la red para cada longitud de onda.

Error absoluto medio (MAE) empleamos esta función en el modelo de autoencoder como alternativa a MSE.

$$MAE = \frac{1}{n}\sum_i^n |x_i - x_{pi}| \tag{3.8}$$

3.2.3. Optimizador y learning rate

El optimizador es uno de los dos argumentos [10], junto con la función de *loss*, que debemos pasarle a la red en Keras para que esta se entrene. Se trata de un algoritmo iterativo que busca ajustar los pesos de forma que disminuya la función de *loss*.

Una de las formas más visuales de entender cómo funcionan es comprender el GD (gradient descent) o descenso del gradiente, que cómo adelantábamos en secciones anteriores se basa en calcular el gradiente (derivadas parciales con respecto a los pesos) de forma que conocemos la dirección en la que variar los pesos para reducir dicha función de *loss* (sentido opuesto al gradiente).

Repetimos este proceso de forma iterativa para acabar en un mínimo [5].

Idealmente, queremos acabar en el mínimo absoluto y no en un mínimo local, aquí entra en juego el seleccionar adecuadamente el optimizador y uno de sus argumentos principales que es el *learning rate* (tasa de aprendizaje).

El *learning rate* es un valor, típicamente entre 0,1 y 0,0001 (dependiendo el optimizador empleado) que multiplicaremos por el valor del gradiente para la actualización de los parámetros, por tanto nos da una idea de cuánto varían dichos parámetros en cada paso. Una idea que resulta interesante es el *learning rate decay* [5] que hace que el valor del *learning rate* disminuya a medida que iteramos de forma que cuando nos encontremos cerca del mínimo, las actualizaciones sean cada vez menores.

Dos de los optimizadores más importantes y usados son ADAM y SGD (ver [10]).

3.2.4. Batch Size

Mini batch consiste en fraccionar el *dataset* en lotes más pequeños que hacemos pasar por la red.

Su uso se justifica en la necesidad de agilizar el proceso de entrenamiento, idealmente haríamos pasar cada variable por la red, el optimizador aplicaría variando los valores de los parámetros y así sucesivamente con las posteriores variables de entrada. Esto no resulta del todo eficiente ya que los tiempos de entrenamiento son demasiado elevados. La alternativa es actualizar los pesos tras el paso de cada *mini batch*.

3.2.5. Número de epochs

Consiste en hacer pasar por la red varias veces los datos. Buscamos con ello continuar bajando el valor de *Loss*, desde donde lo dejamos en el anterior *epoch*.

No podemos incrementar el nº de *epochs* sin control[5] ya que pueden aparecer *overfitting* o quizá las métricas que empleamos para analizar la red empeoren debido a que nos alejemos del mínimo.

3.3. Autoencoders

Un autoencoder es una red neuronal de aprendizaje no supervisado que entrenamos para que trate de reconstruir [7] el dato de entrada que le pasamos.

Emplearemos un modelo de capas densamente conectadas con una entrada de 15211 variables, una para cada longitud de onda de los espectros. Típicamente los autoencoders cuentan con dos partes:

Encoder es una red densamente conectada que codifica el dato de entrada [6] transformándolo en un vector de menos dimensiones al que llamaremos representación latente.

Decoder el decoder conecta con la salida del encoder y trata de reconstruir a partir de esa representación el dato original. Típicamente el decoder es la imagen especular del decoder ya que tanto la salida como la entrada del autoencoder ha de tener las mismas dimensiones.

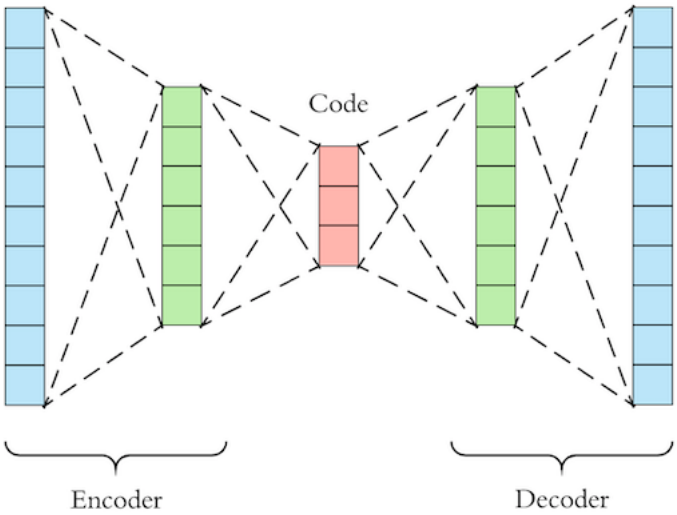


Figura 3.4: Esquema básico de un autoencoder. Imagen de `https://github.com/lightsalsa251/Movie-Recommender-System#readme`

Los autoencoders tienen multitud de aplicaciones [7], la primera e inmediata es la de comprimir información, podemos emplearlo como clasificador a partir de las representaciones latentes, aplicaremos en detección de anomalías y como eliminador de ruido (DAE) [6] lo cuál resulta de que el autoencoder no reconstruye el dato de entrada de forma totalmente perfecta y pierde información.

La métrica principal que empleamos en el desarrollo del modelo será la función de pérdida, buscando minimizar la misma, de forma que la entrada y salida sean lo más parecidas posible.

En este caso el autoencoder se entrena de forma no supervisada ya que no le pasamos durante en proceso de entrenamiento las etiquetas de los datos de entrada mediante el algoritmo de *backpropagation*.

Tratamos de forzar al autoencoder a que la salida tienda al valor de entrada que copie o "aprenda" patrones interesantes de los datos de entrada [7].

3.4. Clasificador: capa softmax

Nuestro modelo se basará en una red neuronal de capas densamente conectadas. Veremos en el siguiente capítulo cómo implementarlo en TensorFlow con módulos de la librería Keras.

Inicialmente necesitamos una entrada de 15211 variables, una para cada longitud de onda de nuestros espectros. Posteriormente tendremos una arquitectura de capas ocultas que detallaremos más adelante y finalmente a la salida le añadimos una capa de 10 neuronas con activación softmax que nos dará las predicciones en forma de vector de probabilidades para los diferentes tipos siendo la mayor probabilidad la predicción concreta que usaremos en las métricas de evaluación de la red.

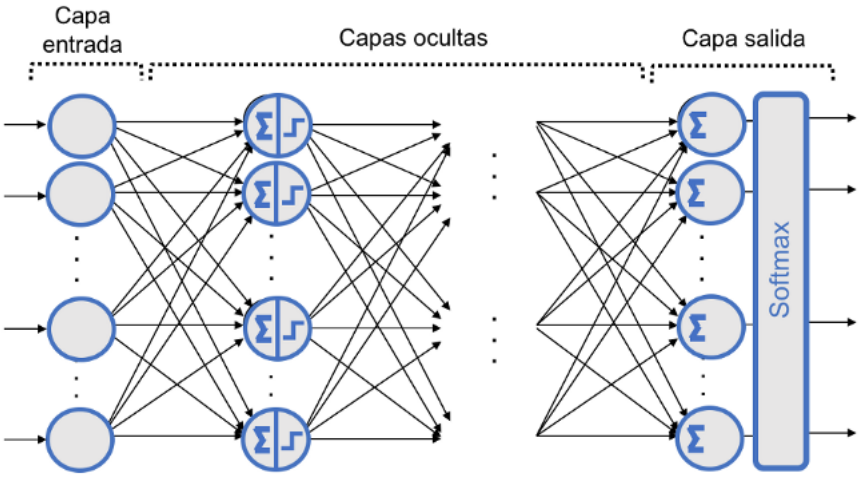


Figura 3.5: Esqéma básico de una red neuronal densamente conectada con activación softmax en la capa de salida. Torres, J. (2021). Casificador softmax [Gráfico]. Jordi TORRES.ai. `https://medium.com/aprendizaje-por-refuerzo/`

Una de las métricas más empleadas en la evaluación de la performance de una red neuronal es la *accuracy*, que consiste en:

$$Acuraccy = \frac{Clasificados\, correctamente}{Total\, de\, casos} \tag{3.9}$$

Nos da la fracción de casos predichos correctamente por la red. En según qué aplicaciones, esta métrica puede resultar ineficaz [5], donde un falso negativo y un falso positivo no tienen la misma importancia.

En este caso, la red se entrena de forma supervisada ya que le pasamos las etiquetas de los datos de entrenamiento que le sirve a la red para comparar con los datos que obtiene y mediante el algoritmo de *backpropagation* ajustar los pesos para acercarse a dichos valores.

Capítulo 4

Desarrollo del modelo

En este capítulo haremos un resumen del proceso de implementación de la red neuronal, partiendo del *dataset* descrito en la la sección 2.2.1, el pre-procesado de los datos, la arquitectura de la red, proceso de entrenamiento, la forma en que se han optimizado los hiperparámetros y el posterior análisis de la performance del modelo.

Todo el código ha sido escrito y ejecutado en Google Colab [11], una herramienta desarrollada por Google que permite crear, editar y ejecutar cuadernos Jupyter (*Jupyter Notebooks*) online. A su vez, permite importar, de forma sencilla, todas las librerías y paquetes que hemos necesitado.

4.1. Pre-procesado de datos

En primer lugar importamos la librería *pandas* que resulta muy útil para tratar con objetos de tipo *dataframe* [12], nos permite leer nuestro archivo .csv de forma sencilla.

```
df = pd.read_csv('/gdrive/MyDrive/EspectrosPatatasSecas/espectros_patatas_secas.csv',
  header=None)
```

Scikit-learn [13] es otra librería muy potente y empleada en trabajos de machine learning y la empleamos para 'splitear' o dividir nuestro conjunto de datos en datos de entrenamiento (*train*) y datos de prueba (*test*). La función *.split* nos premite seleccionar el tamaño del conjunto de datos de *test*, en nuestro caso del 30 %

```
train_X, val_X = train_test_split(df, test_size = 0.3,  random_state=1)
```

El siguiente paso es tomar de cada dato de entrada su etiqueta (la primera columna del *dataframe*) y separarlos de los espectros. Para ello usaremos *numpy* [14], que resulta conveniente por el tipo de dato para pasarlo posteriormente a la red neuronal.

```
  # Etiquetas
cat = df_train[0]
cat_train = cat.values
cat_t = df_test[0]
cat_test = cat_t.values
  # Espectros
trainlist = df_train.to_numpy()
dat_train = np.delete(trainlist, 0, axis=1)
testlist = df_test.to_numpy()
dat_test = np.delete(testlist, 0, axis=1)
```

Utilizamos la clase *StandarScaler* de *scikit-learn* donde desplazamos la media a 0 y escalamos la desviación estándar a 1.

```
scaler = StandardScaler()
scaler.fit(dat_train)
train_scaled = scaler.transform(dat_train)

scaler.fit(dat_test)
test_scaled = scaler.transform(dat_test)
```

Finalmente realizamos el proceso de *One hot* que explicamos en la sección 3.2.2, en el que convertimos las etiquetas de datos numéricos a vectores de 10 componentes (todas son cero salvo la posición de la etiqueta que es 1).

```
# datos de train One hot
cat_train = to_categorical(cat_train, num_classes=11)
cat_correcta = np.delete(cat_train, 0, axis=1)
#datos de test One hot
cat_test = to_categorical(cat_test, num_classes=11)
cat_test_correcta = np.delete(cat_test, 0, axis=1)
```

4.2. Diseño y entrenamiento

Partimos del esquema expuesto en la sección 3.4 donde hemos establecido que la red, densamente conectada, consta de 15211 entradas por cada espectro y al final de la red le añadimos una capa extra con activación softmax que consta de 10 neuronas debido que es la dimensión de los vectores en los que hemos transformado las etiquetas.

A partir de aquí debemos decidir el número de capas, las neuronas de cada capa y su activación y el optimizador.

Implementar la red en Keras resulta razonablemente sencillo, debemos importar paquetes como *Dense*, *Sequential*, *Model* e *Input*

```
#Modelo
np.random.seed(23)
dim_entrada = train_scaled[0].shape
model=Sequential()

model.add(Dense(1000, activation='relu', input_shape=(dim_entrada)))
model.add(Dense(500, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Con la función *Sequential* creamos el modelo y posteriormente con *.add* añadimos las capas a la red especificando que se trata de capas densamente conectadas, el número de neuronas por capa y su función activación.

La función de activación de las capas ocultas es la función ReLU (3.2). Los pesos han sido inicializados de forma aleatoria pero sesgada con la función *random.seed*(23).

El siguiente paso pasarle a la red la función de *loss*, el optimizador y las métricas que emplearemos para el análisis que en nuestro caso es la *accuracy*.

```
model.compile(loss="categorical_crossentropy", optimizer="sgd", metrics = ['accuracy'])
model.fit(train_scaled, cat_correcta, batch_size=15, epochs= 15)
```

La función *fit* realiza el entrenamiento de la red, debemos pasarle la entrada de la red, en nuestro caso es los datos de entrenamiento escalados, las etiquetas en forma de vector, el tamaño de *batch* y el nº de *epochs*.

En este punto, en pantalla se puede ver como la función de *loss* y la *accuracy* se van actualizando con cada paso de los *mini-batch* y cada *epoch*.

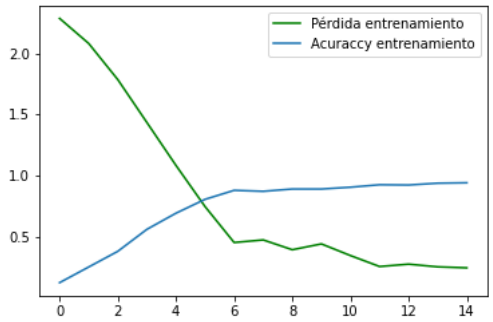


Figura 4.1: Evolución de la función de *loss* y de la *accuracy* durante el entrenamiento. Los datos son los obtenidos al finalizar cada *epoch*.

Por último, podemos evaluar pasando a la red los datos de *test* que también tenemos etiquetados y la función *evaluate* nos devuelve la estimación de *loss* y la *accuracy* para estos nuevos datos.

```
test_loss , test_acc = model.evaluate(test_scaled , cat_test_correcta)
```

La *accuracy* obtenida para para el conjunto de datos de validación es de 0,8119.

Es un resultado razonable, pero si lo comparamos con la *accuracy* de los datos de entrenamiento que es 0,9414, vemos que es bastante menor por lo que la red está trabajando peor con los datos de test. Esto es normal pero queremos tratar de reducir las diferencias evitando así dicho *overfitting*.

En las siguientes secciones 4.3 y 4.4 trataremos de mejorar el desempeño de la red mediante la optimización o 'tuneo' de los hiperparámetros de la red y técnicas de reducción de *overfitting*.

4.3. Optimización de Hiperparámetros

En la sección anterior creamos nuestro modelo de red neuronal y realizamos un primer entrenamiento con parte de los datos y posteriormente se hace un testeo con el resto de datos para ver el desempeño de la red. Atendiendo a los resultados podemos ver que la red funciona clasificando mejor los datos de entrenamiento que los datos de test.

El objetivo en esta sección será el de tunear los diferentes hiperparámetros de la red y modificar su arquitectura de capas y neuronas por capa de tal forma que consigamos mejorar la *accuracy* de los datos de test. Para ello existen diversas herramientas como *Optuna* [16] o *KerasTuner* [17]. En nuestro caso emplearemos el primero.

Optuna es un paquete *Open source* que permite la optimización de los mejores hiperparámetros de forma semiautomática. Permite hacer una búsqueda iterativa de forma sencilla, definir el espacio de búsqueda, el nº de muestreo y realizar la optimización de forma eficiente [18].

```
1  def run(trial):
2      clear_session()
3      np.random.seed(23)
4      dim_entrada = train_scaled[0].shape
5
6      param = trial.suggest_float('n_neurons',0,2)
7      neurons = param * 1000
8      neurons = round(neurons, 0)
9      neurons1 = round(neurons/2, 0)
10     neurons2 = round(neurons1/2, 0)
11
12     model=Sequential()
13
14     model.add(Dense(neurons, activation='relu', input_shape=(dim_entrada)))
15     model.add(Dense(neurons1, activation='relu'))
16     model.add(Dense(neurons2, activation='relu'))
17     model.add(Dense(10, activation='relu'))
18
19     model.add(Dense(10, activation='softmax'))
20
21     model.compile(loss="categorical_crossentropy", optimizer="sgd",
22                   metrics = ['accuracy'])
23     model.fit(train_scaled , cat_correcta , batch_size=15, epochs= 15)
24     test_loss , test_acc = model.evaluate(test_scaled , cat_test_correcta)
25     return test_acc
```

Listado 4.1: Implementación de optuna en el modelo de red de la seción 4.2 para la búsqueda del número de neuronas por capa.

En el listado 4.1 hemos implementado *Optuna* en tres pasos como recomienda en [18].

Definir función objetivo definimos la función objetivo que trataremos de maximizar en nuestro caso. Retorna la *accuracy* de los datos de test. Le incluimos en la línea 2 un comando para que los sucesivos entrenamientos no se solapen y sean independientes.

Establecer el espacio de búsqueda En este punto le "sugerimos" al algoritmo el hiperparámetro que queremos optimizar y el espacio de búsqueda mediante un objeto trial[18].

Modelo Finalmente en el modelo sustituimos el hiperparámetro por el objeto que hemos sugerido previamente.

Ahora creamos un objeto *study* [18], donde le pasamos el número de intentos (*trials*), la función y la dirección que en nuestro caso tratamos de maximizar.

```
study = optuna.create_study(direction='maximize')
study.optimize(run, n_trials=20)
```

Podemos ver que tras los 20 intentos la *accuracy* parece no mejorar sustancialmente, figura 4.2, con respecto a la obtenida tras el primer entrenamiento en la sección 4.2.

Realizamos una optimización del tamaño de los *mini batch*, ya que parece que al variar este hiperparámetro la *accuracy* cambia bastante.

Ahora el parámetro que le sugerimos a la red es el *batch_size* que tras hacer pruebas un tanto aleatorias establecemos entre 5 y 50 espectros. El número de neuronas ahora lo fijamos con el mejor resultado del estudio realizado previamente y el *batch_size* en este caso debemos seleccionar el objeto de muestreo.

```
    batch_size = trial.suggest_int('batch_size', 5, 50)
    neurons = 0.6804258279745674 * 1000    # Sacado de la busqueda con optuna
    model.fit(train_scaled, cat_correcta, batch_size=batch_size   , epochs= 15)
```

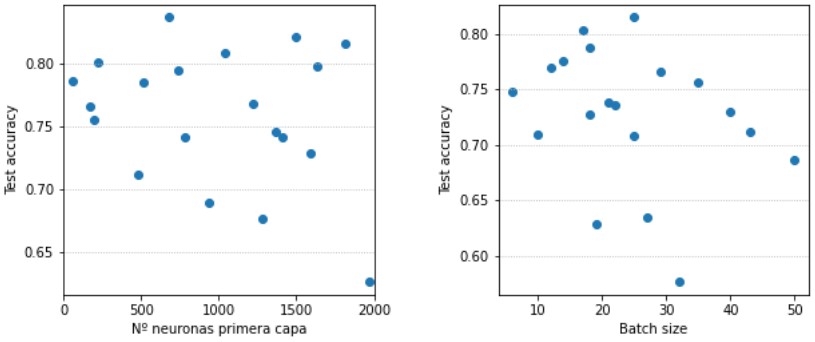


Figura 4.2: En la figura de la izquierda, resultado de la ejecución del estudio con *Optuna* de la búsqueda de nº de neuronas de la primera capa. La figura de la derecha muestra el posterior estudio del tamaño de *batch* hecho con los mismos parámetros salvo el nº de neuronas que fijamos el optimizado.

La ultima capa oculta de 10 neuronas con activación ReLU puede resultar demasiado pequeña y tras eliminarla realizamos un nuevo entrenamiento y se puede ver que mejoran los resultados. Podemos ver que los resultados mejoran notablemente (tabla 4.1).

Repetimos la optimización del *batch_size* pero esta vez sin la capa oculta de 10 neuronas y el tamaño óptimo varía de 25 a 6 (ver tabla 4.1).

El *learning rate* resulta interesante de optimizar, a pesar de que los resultados en este punto son ya razonables en términos de *accuracy*. Para esto debemos introducir algunas líneas de código para establecer el espacio de búsqueda.

```
    lr = trial1.suggest_int('l_r', 90, 109, step=1)
    opt = SGD(learning_rate=lr/10000)
```

Estamos haciendo una búsqueda entre en torno al valor que viene por defecto de en el optimizador SGD que es de 0.01.

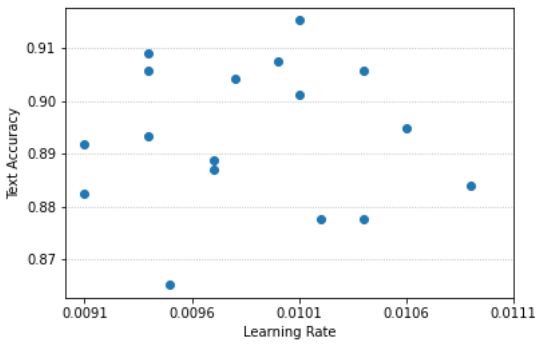


Figura 4.3: Resultados de la optimización del *learning rate* con los valores fijados de los hiperparámetros anteriormente optimizados.

Repetimos el proceso de nuevo con el nº de neuronas por capa, pero esta vez sin la última capa. Los resultados del proceso de la optimización de los hiperparámetros se recogen en la tabla 4.1.

Capas	Batch size	Epochs	Optimizador (lr)	Ejecución	Trials	Test Accuracy
1000 - 500 - 100 - 10	15	15	SGD	Simple	-	0,8119
680 - 340 - 170 - 10	15	15	SGD	Optuna	20	0,8369
680 - 340 - 170 - 10	25	15	SGD	Optuna	19	0,8150
680 - 340 - 170	25	15	SGD	Simple	-	0,8746
680 - 340 - 170	6	30	SGD	Optuna	20	0,9122
680 - 340 - 170	6	30	SGD(0.0101)	Optuna	17	0,9153
878 - 439 - 220	6	30	SGD	Optuna	20	0.9373

Tabla 4.1: Resumen de los resultados durante el proceso de optimización de los hiperparámetros.

4.4. Técnicas reducción de overfitting

Como adelantábamos en el capítulo anterior, un problema típico que puede presentarse en modelos basados en redes neuronales es el problema del *overfitting* o sobreajuste que se da cuando el modelo se ajusta muy bien a los datos de entrenamiento y su desempeño con ellos es muy alto pero cuando le pasamos a la red datos con los que no ha sido entrenado la performance baja notablemente y no es capaz de hacer buenas predicciones o mucho peores que con los datos de entreno.

Existen técnicas para disminuir dicho sobreajuste, como puede ser aumentar el conjunto de datos de entrenamiento para así entrenar a la red con una mayor variedad de ellos. Discutiremos en los siguientes capítulos el por qué no hemos empleado esta vía.

Con el modelo ya optimizado de la sección 4.3 vamos a aplicar técnicas que consisten en añadir capas intermedias a nuestro modelo con distintas funciones.

Dropout Consiste en añadir capas colocadas entre las capas densas que esencialmente desconectan las neuronas [6] o haciendo referencia a su nombre "abandonan".

```
model.add(Dense(neurons, activation='relu', input_shape=(dim_entrada)))
model.add(Dropout(t_a))   # t_a es la tasa de abandono que debe aplicar
model.add(Dense(neurons1, activation='relu'))
model.add(Dropout(t_a))
model.add(Dense(neurons2, activation='relu'))
model.add(Dropout(t_a))
model.add(Dense(10, activation='softmax'))
```

Hacemos un mapeo para distintos valores del *rate* que le pasamos a las capas de *Dropout* y obtenemos su *accuracy* para los datos de test.

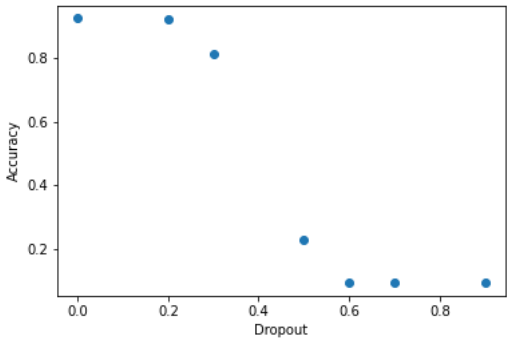


Figura 4.4: Representación de los datos de *accuracy* para el set de test en función de la tasa de abandono aplicada. Ha sido ejecutado con el modelo ya optimizado.

Batch Normalization son capas que normalizan su input [19] y conectan con la siguiente capa. AL igual que con la técnica de *Dropout* hemos removido dichas capas y las sustituimos por estas capas de normalización.

```
model.add(BatchNormalization())
```

La accuracy para los datos de test obtenida aplicando estas capas de *Batch Normalization* es de $0,8950$.

Capítulo 5

Análisis de los resultados

En este capítulo haremos una recopilación del progreso desde el primer entrenamiento realizado hasta tener el modelo optimizado, como de bueno es su desempeño y discutiremos las técnicas empleadas para la reducción del *overfitting*.

5.1. Matriz de confusión

La matriz de confusión es una herramienta que recopila el número de predicciones que hacemos para cada tipo. En las columnas están representados los valores de etiqueta predichos mientras que en las filas los valores reales.

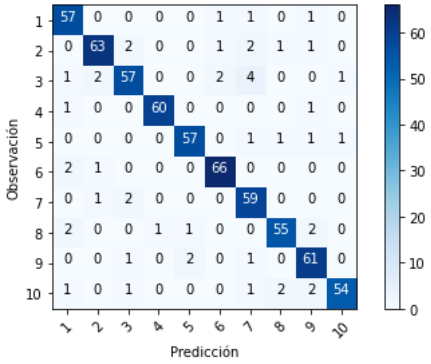


Figura 5.1: Matriz de confusión[1] para el modelo optimizado en la sección 4.3.

A partir de dichas predicciones podemos determinar el valor de la *accuracy* de los datos de test que el programa nos devuelve directamente.

De un vistazo rápido podemos ver que la mayor parte de las predicciones se encuentran en la diagonal principal, es decir, que las predicciones coinciden con las etiquetas originales. Los valores que no se encuentran en dicha diagonal son fallos en la predicción. No destaca ningún valor por encima de los demás.

Existen numerosas métricas para el análisis del desempeño de los modelos, en nuestro caso empleamos *accuracy* ya que en principio, una vez que detectamos un fallo en la predicción, no es relevante la predicción concreta. Este tipo de problema suele ser relevante en áreas de la salud o si detectamos un tipo de patata perjudicial de ingerir, en el que no tiene el mismo peso (en una clasificación binaria) un falso positivo o un falso negativo.

5.2. Resultados optimización

En esta sección analizamos la evolución desde el primer entrenamiento hasta el modelo optimizado.

[1] El código necesario para obtener la matriz de confusión con este aspecto ha sido obtenido expresamente de [20].

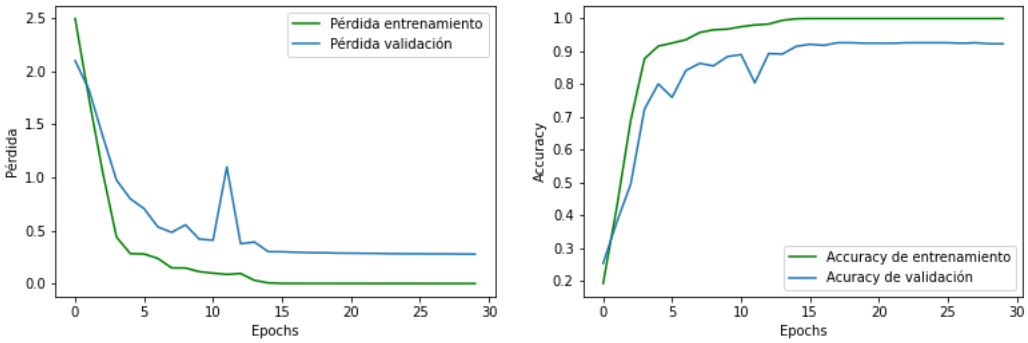


Figura 5.2: Evolución de la estimación de, a la izquierda la función de pérdida y en la gráfica de la derecha la *accuracy* tanto de entrenamiento como de validación con el paso de los *epochs* para el modelo optimizado con los mejores hiperparámetros.

De la figura 5.2 podemos destacar ya a simple vista, que a partir de la época 15 tanto el valor de la función de pérdida como el valor de la *accuracy* permanecen constantes por lo que podemos reducir el entreno a unas 15 épocas.

Podemos observar en torno a la época 10-12 un pico pronunciado tanto en la *accuracy* como en la función de pérdida pero a partir de ahí continua con la tendencia. En el capítulo de conclusiones veremos alguna opciones para tratar de mejorar la estabilidad y tratar de que estos picos se reduzcan lo máximo posible.

El resultado de la *accuracy* para este caso fue de 0,9232 tras ejecutar la red de nuevo con los parámetros ya optimizados. Podemos ver que es ligeramente inferior al valor de 0,9373 obtenido en la última búsqueda con optuna a pesar de que ejecutamos con los mismos hiperparámetros y configuración de capas. Este hecho se ha visto durante todo el proceso. Variaciones del 2-3 % que no hemos sido capaces de evitar.

De la optimización podemos destacar, como se aprecia en la figura 4.3 que el valor de la tasa de aprendizaje (siempre que esté en torno al valor por defecto de 0.01) no es demasiado relevante y no mejora sustancialmente la performance de la red al menos en término de *accuracy*, variaciones del orden de 0.04.

Tanto el tamaño de los *mini batch* y el nº de neuronas de la primera capa (que en nuestro modelo influye en el del resto de capas ocultas) vemos en la figura 4.2 que dependen en buena medida. Son hiperparámetros que no son intuitivos de optimizar y la búsqueda con optuna resulta clave. En ambos casos podemos apreciar variaciones en torno al 20 % en términos de *accuracy*.

Al comienzo del proceso contábamos con una capa oculta extra de 10 neuronas que tras removerla, vimos (ver tabla 4.1) que la performance aumentaba notablemente.

5.3. Análisis de la reducción del overfitting

En esta sección vamos a analizar el proceso de entrenamiento aplicando las capas de regulación de *dropout* y las de *batch normalization* de forma que podemos comparar con el modelo optimizado de la sección anterior.

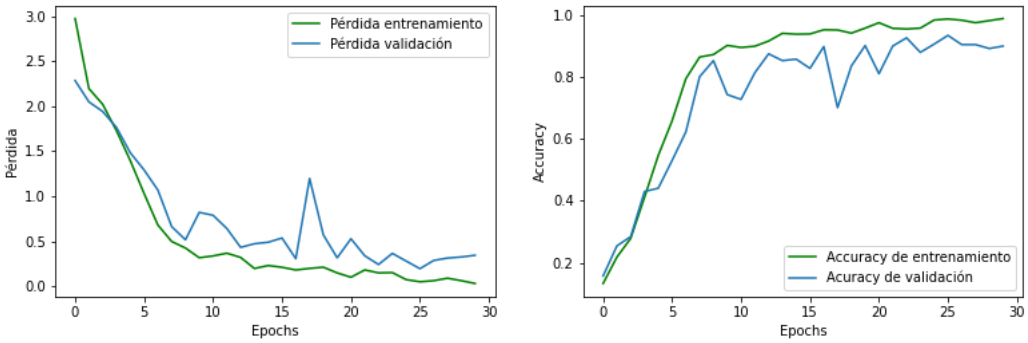


Figura 5.3: Evolución de la estimación de, a la izquierda la función de pérdida y en la gráfica de la derecha la *accuracy* tanto de entrenamiento como de validación con el paso de los *epochs* para el modelo optimizado con los mejores hiperparámetros y añadiendo capas intermedias de *dropout* con una tasa de 0.2.

Un *rate* de *dropout* de 0.2 parece razonable a la vista de la figura 4.4 ya que mantiene el *accuracy*. Si nos fijamos ahora en la figura 5.3 podemos observar tanto en el descenso en la función de pérdida como en la evolución de la *accuracy* un mayor número de picos a medida que se suceden las épocas. Tampoco llegamos a un valor que tienda a ser estable en las últimas épocas.

Analizando los valores de *accuracy* de entrenamiento y de validación en la tabla 5.1, vemos que la diferencia entre ellos no disminuye. Ello unido a la tendencia más errática que observamos hace que no parezca recomendable usar *dropout* en este momento.

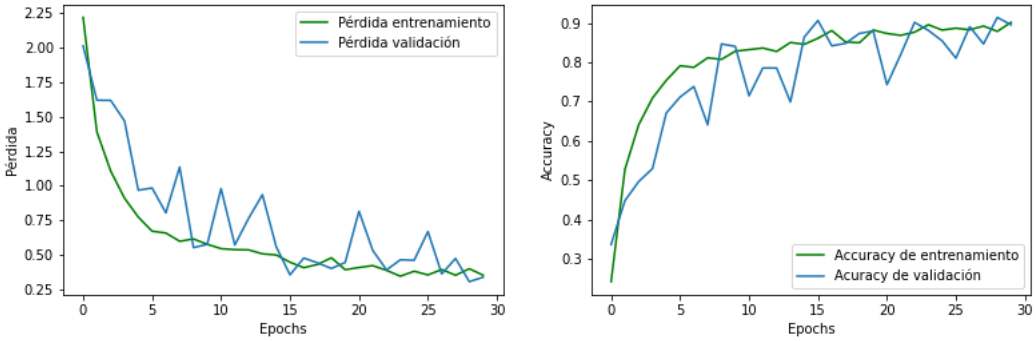


Figura 5.4: Evolución de la estimación de, a la izquierda la función de pérdida y en la gráfica de la derecha la *accuracy* tanto de entrenamiento como de validación con el paso de los *epochs* para el modelo optimizado con los mejores hiperparámetros y añadiendo capas intermedias de *batch normalization*.

A la vista de la figura 5.4 podemos observar al igual que en el caso de aplicar *dropout* anterior, la tendencia tanto de la función de pérdida como la *accuracy* de validación parecen tener un comportamiento más errático aunque la tendencia es la correcta. Los valores pasan de variar un 2-3 % a variar en torno al 10 % en términos de *accuracy*.

De nuevo en este caso la *accuracy* se mantiene en rangos razonables aunque no mejores que sin aplicar estas capas.

En este caso si podemos decir que la diferencia entre la *accuracy* de entreno y de validación se reduce notablemente.

	Pérdida		Accuracy	
Modelo	Entrenamiento	validación	entrenamiento	validación
Sin optimizar	0,2446	0,6667	0,9415	0,8119
Optimizado	3,5e-04	0,2771	1,000	0,9232
Dropout(0.2)	0,0301	0,3440	0.9886	0.8997
Batch normalization	0,343	0,343	0,9012	0,8950

Tabla 5.1: Resumen de estimación de función de pérdida y de *accuracy* para los diferentes modelos.

Capítulo 6

Modelo de autoencoder para detección de anomalías

En este capítulo vamos a tratar de explicar el proceso de implementación del modelo de autoencoder para la detección de las anomalías con el *dataset* provisto de los espectros de las lapas. Posteriormente haremos un análisis del desempeño del mismo.

6.1. Pre-procesado y Modelo

Para este caso, los datos son provistos en diferentes datasets, pero de forma sencilla pueden concatenarse. El proceso es similar al realizado en la sección 4.1 salvo por el hecho de que en este caso no es necesaria la transformación de *One Hot*, ya que en el modelo de autoencoder no trabajaremos con la capa final de activación softmax.

Los datos son tratados de nuevo con la función *StandarScaler()* y están listos para pasarlos al modelo que implementaremos en la librería de Keras.

```
np.random.seed(23)
dim_entrada = dat_train.shape[1]

entrada = Input(shape=(dim_entrada,))
    # Encoder
encoder = Dense(2500, activation='relu')(entrada)
encoder = Dense(1500, activation='relu')(encoder)
encoder = Dense(1000, activation='relu')(encoder)
encoder = Dense(800, activation='relu')(encoder)
    # Decoder
decoder = Dense(1000, activation='relu')(encoder)
decoder = Dense(1500, activation='relu')(decoder)
decoder = Dense(2500, activation='relu')(decoder)
decoder = Dense(dim_entrada, activation='relu')(encoder)
    #Autoencoder = encoder + decoder
autoencoder = Model(inputs=entrada, outputs=decoder)

autoencoder.compile(optimizer=tf.keras.optimizers.Adam, loss='mse')
historia = autoencoder.fit(train_scaled, train_scaled, epochs=20, batch_size=19,
                           validation_data=(test_scaled, test_scaled))
```

Con unas pocas líneas de código podemos crear el modelo y ejecutarlo. La forma en la que añadimos las capas es similar al modelo del clasificador de patatas. En este caso emplearemos como métrica para evaluar el modelo únicamente la función de pérdida (error cuadrático medio).

Realizamos un primer entrenamiento del autoencoder con los parámetros del listado anterior.

El resultado de la función de pérdida estimado para los datos de validación es de 0.6928.

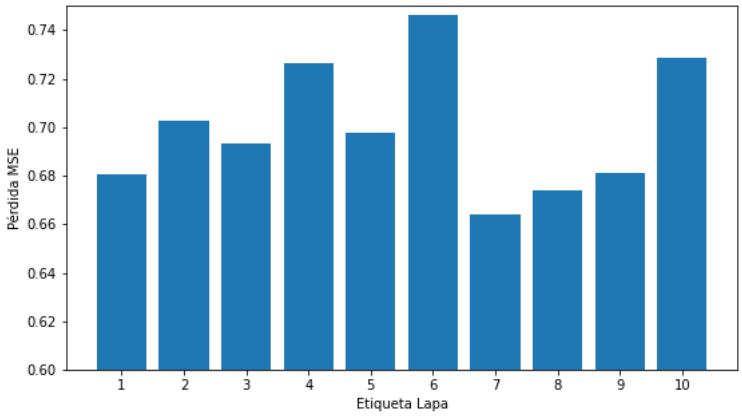


Figura 6.1: Media del error cuadrático medio para cada tipo de lapa de los datos de validación.

6.2. Optimización del modelo

En esta sección buscamos de optimizar los hipérparámetros del modelo para mejorar su performance. Realizaremos una búsqueda del nº de neuronas, tamaño de *mini batch* y de la tasa de aprendizaje.

Se crea una función objetivo a la que le "sugerimos" el espacio de búsqueda de cada hiperparámetro a optimizar y nos retorna la media del error cuadrático medio de los datos de validación al comparar los datos de entrada con las reconstrucciones que es capaz de realizar el autoencoder. Para ello debemos incluir dentro de la función objetivo las siguientes líneas de código para generar dichas reconstrucciones de los datos de entrada de validación.

```
def objetive(trial):
    ...
    ...
    ...
    pred = autoencoder.predict(test_scaled)
    error_normal_validacion = tf.keras.losses.mean_squared_error(pred, test_scaled)
    error_normal = np.mean(error_normal_validacion)
    return error_normal
```

Realizamos una primera búsqueda del número de neuronas de la primera capa, las posteriores el número de neuronas le vamos dividiendo entre 2 sucesivamente hasta el cuello de botella. El decoder será la imagen espejo.

```
n_neurons = trial_neu.suggest_int('n_neurons', 500, 2500)
```

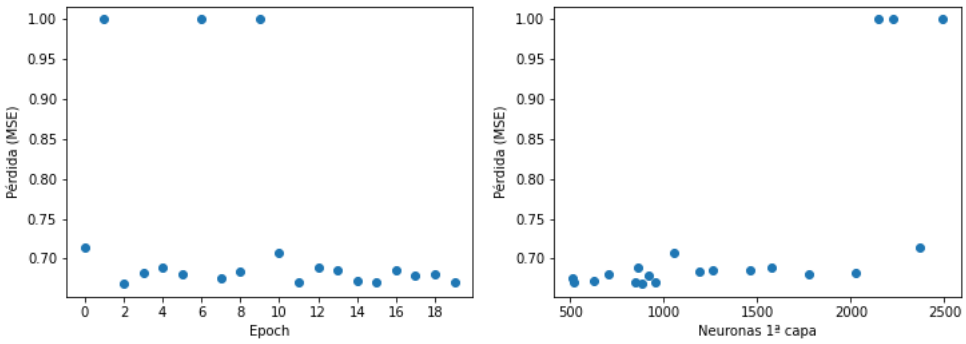


Figura 6.2: Resultado de la búsqueda con *Optuna* del número de neuronas de la primera capa oculta del autoencoder. A la derecha la evolución de la función de pérdida con los *trials* (intentos). A la izquierda representamos la función de pérdida en función del nº de neuronas.

El proceso de optimización es similar a la sección 4.3, realizamos las búsquedas de forma independiente y sucesiva.

Posteriormente realizamos una búsqueda del tamaño de los *mini batches* con la configuración de capas óptima obtenida en la búsqueda anterior.

```
batch_size = trial_batch.suggest_int('batch_size', 5, 100)
```

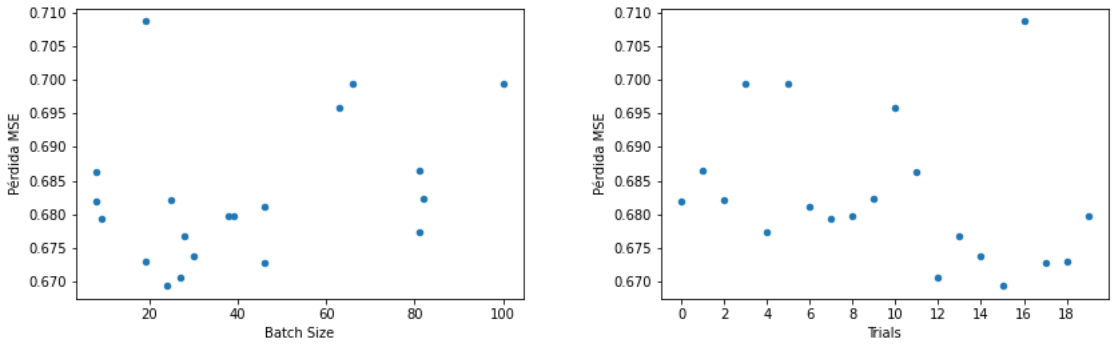


Figura 6.3: A la izquierda representamos la estimación de la función de pérdida en función del tamaño de los *mini batch*. A la derecha representamos la evolución de la función de pérdida con el paso de los entrenamientos (trials).

Por último realizamos una búsqueda de la tasa de aprendizaje para el optimizador Adam.

```
l_r = trial_lr.suggest_float('l_rate', 1e-5, 1e-2)
```

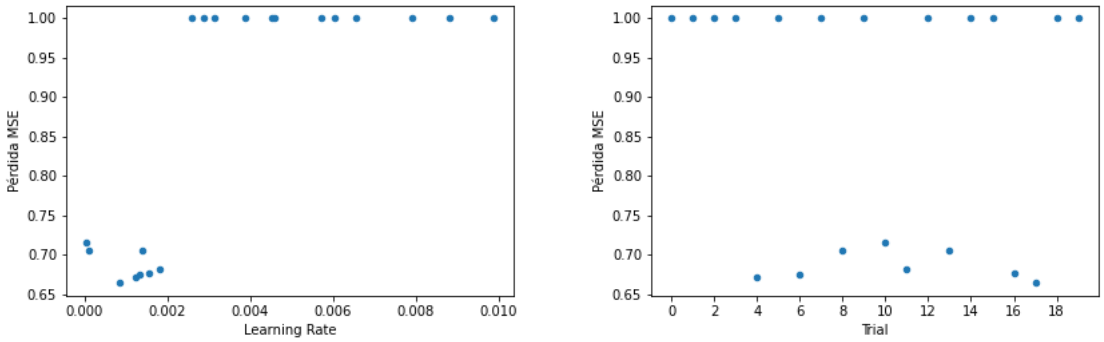


Figura 6.4: A la izquierda representamos la estimación de la función de pérdida en función de la tasa de aprendizaje. A la derecha representamos la evolución de la función de pérdida con el paso de los entrenamientos (trials).

Los resultados del proceso de optimización se recogen en la tabla 6.1

Capas	Batch Size	Epochs	Optimizador	Trials	Error Validación
2500-1500-1000-800-1000-1500-2500	40	15	Adam	-	0.6928
886-442-212-442-886	19	20	Adam	20	0.6803
886-442-212-442-886	24	20	Adam(0.0012)	20	0.6688
886-442-212-442-886	24	20	Adam(0.00085)	20	0.6648

Tabla 6.1: Resultados de la media de error entre los datos de validación y sus correspondientes reconstrucciones del autoencoder durante el proceso de optimización de los hiperparámetros.

6.3. Análisis de los resultados

Realizamos un último entreno con los "mejores"hiperparámetros obtenidos en la sección anterior. Si realizamos una reconstrucción para los valores de validación podemos comparar con la figura 6.1.

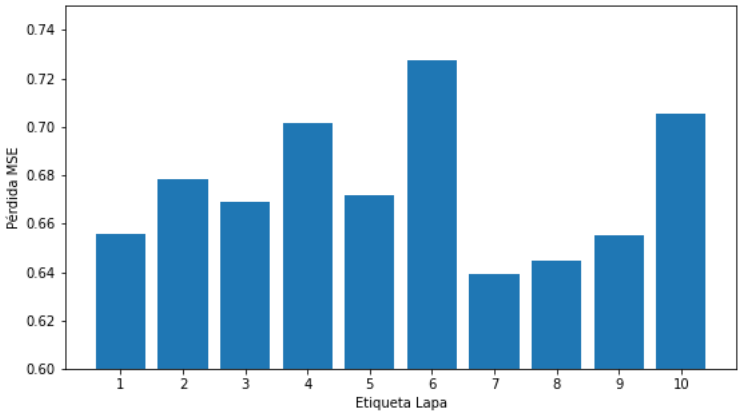


Figura 6.5: Media del error cuadrático medio para cada tipo de lapa de los datos de validación para el modelo ya optimizado.

Podemos observar cómo el objetivo de minimizar las medias de error cuadrático al reconstruir se cumple, las diferencias entre los deferentes tipos permanecen prácticamente constantes. A un primer golpe de vista, sorprende que las medias de los tipos #5, #6 y #7 sean similares o menores al resto de tipos normales.

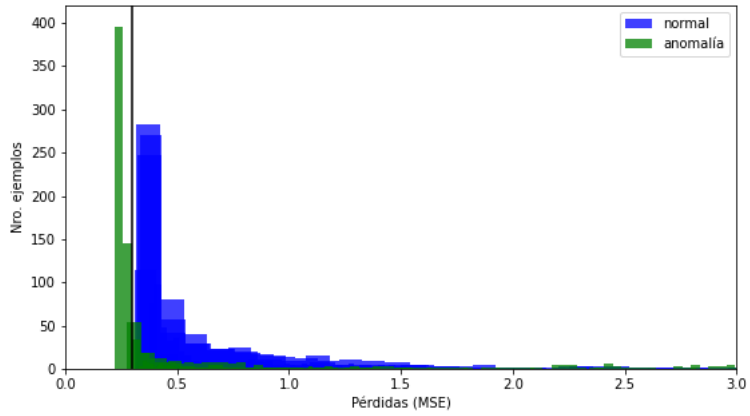


Figura 6.6: Error medio de los datos de validación para cada muestra. Distinguimos en este histograma entre datos normales y anomalías

Se abandona la idea de detectar cada tipo de anomalía de forma independiente. Se pueden ver dos grandes conjuntos de datos.

Trazamos una línea vertical que divide los dos conjuntos, la parte verde a la izquierda de la línea pertenece principalmente a la variedad #10 que será la única distinguible de los datos normales mediante nuestro modelo.

De los datos de validación, contábamos con 838 espectros de tipo #10 y el modelo ha sido capaz de detectar 548 ($\sim 65\,\%$).

Con este criterio, el tipo #8 hemos detectado únicamente el 14 % de las muestras. En el resto de tipos anómalos el modelo no es capaz de detectar y los confunde con espectros normales.

Realizamos un nuevo entrenamiento con el modelo optimizado pero esta vez los datos normales se lo pasamos directamente a la red sin aplicar el *StandarScaler()*. Ahora la métrica que empleamos será la función de pérdida, error absoluto medio (MAE).

Ampliamos el nº de epochs (200) ya que el error en los datos de validación para datos normales disminuye significativamente. Con los datos de Validación y los datos de las anomalías podemos reconstruir las predicciones del autoencoder y obtener las medias de error de cada tipo.

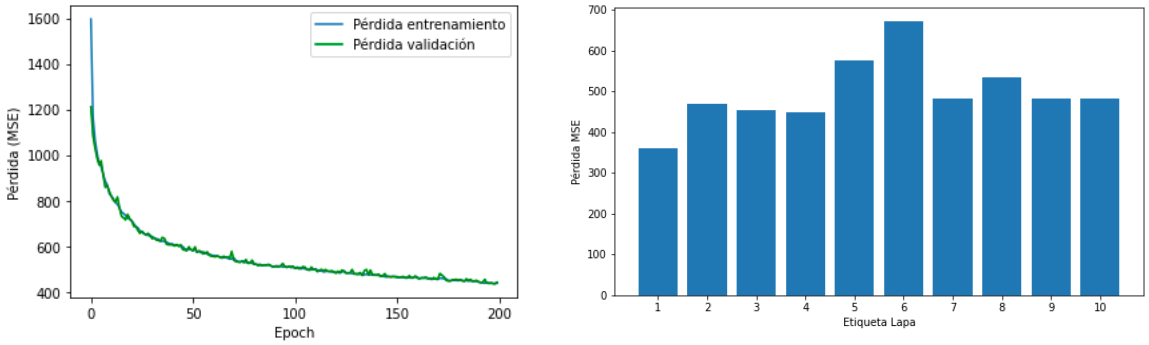


Figura 6.7: A la izquierda, evolución de la función de pérdida (MAE) de los datos de entrenamiento y de los datos de validación. A la derecha vemos representado las medias de los errores de reconstrucción al comparar los datos de cada tipo con las predicciones del autoencoder.

Podemos representar los errores de reconstrucción para cada dato. En este caso, al igual que previamente, se ha optado por no distinguir entre los diferentes tipos normales y los diferentes tipos de anomalías.

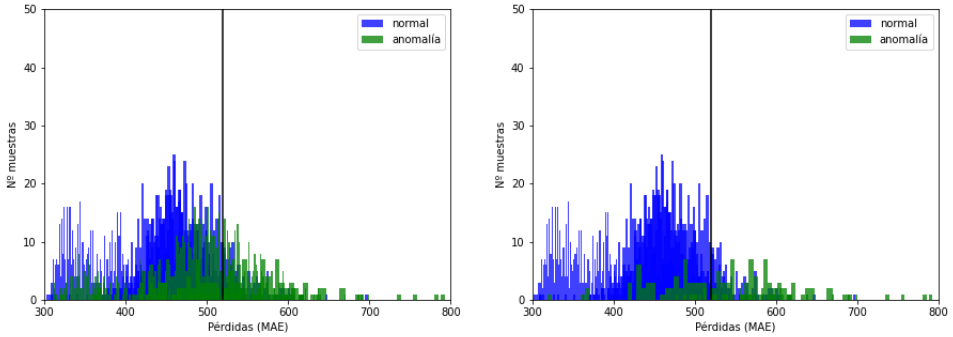


Figura 6.8: Representación de los errores de cada dato con la predicción del autoencoder. A la izquierda están representados todos los datos de validación normales y todas las anomalías. A la derecha suprimimos los datos de tipo #7 y #10 que veremos en la tabla 6.2, el autoencoder no consigue detectar en un porcentaje razonable. Establecemos una barrera que delimita a la izquierda los datos que catalogamos como normales y a la derecha datos anómalos.

Los porcentajes de acierto al catalogar cada dato como normal o anomalía quedan recogidos en la tabla 6.2.

En este caso, a la vista de los datos de la tabla 6.2 y las medias representadas en la figura 6.7, coincide que las anomaías que el autoencoder detecta con mayor porcentaje son las de tipo #5 y #6 que son las que mayor media tienen, algo que no ocurría en el modelo anterior, donde el tipo #6 a pesar de contar con una media mayor no era detectable por el modelo. En cualquier caso, somos capaces de detectarlos como anomalías pero sin concretar de qué tipo.

Etiqueta	Nº aciertos	Porcentaje aciertos (%)
#1	528	100
#2	407	90,6
#3	454	97,0
#4	165	97,6
#5	31	88,5
#6	18	81,8
#7	14	19,4
#8	69	57,5
#9	451	83,0
#10	234	27,9

Tabla 6.2: Datos de aciertos del autoencoder al predecir cada tipo. En verde representados los datos para anomalías.

Capítulo 7

Conclusiones

Las redes de aprendizaje profundo resultan una interesante alternativa para el estudio de espectros de LIBS y en este trabajo hemos implementado con resultados razonables modelos capaces de clasificar o detectar anomalías en dichos espectros.

Destaca la importancia de la optimización de los hiperparámetros de los modelos implementados y como el proceso mejora el rendimiento de la red notablemente.

En el caso del clasificador con capa de activación softmax, observamos cómo una primera capa oculta con demasiadas neuronas no aporta una gran mejora (Figura 4.2). A su vez, una última capa oculta con pocas neuronas empeora notablemente el resultado. Tanto el nº de neuronas como el tamaño de los *mini batch* resulta interesante de optimizar ya que tienen una importante dependencia con el desempeño final de la red. Por su parte, la tasa de aprendizaje parece no influir apreciablemente siempre que nos encontremos en un entorno cercano del valor por defecto implementado en Keras.

Las técnicas de reducción de overfitting no aportan en este caso gran mejora de los resultados lo cuál es razonable ya que el overfitting encontrado con el modelo optimizado es aceptable. Las capas de *batch normalization* lo reduce notablemente aunque en términos de accuracy y de estabilidad de los resultados el modelo empeora ligeramente (Figura 5.4). El modelo con capas intermedias de dropout mantiene una performance cercana al 90 % en accuracy de validación, pero tiene el mismo problema de estabilidad (Figura 5.3).

El modelo optimizado es capaz de clasificar de forma correcta entre los 10 tipos de patatas muestreados con diferente procedencia en el $\sim 92\,\%$ de los casos, lo cuál resulta un resultado aceptable.

En el modelo de autoencoder para la detección de anomalías, realizamos una optimización similar de los hiperparámetros, lo cuál resulta en un descenso de la función de pérdida, pero observamos que disminuir la media de dicha función para los datos normales inicialmente no aporta resultados razonables, por ejemplo, en los datos de tipo #6 este valor es significativamente más grande y aún así el modelo no es capaz de detectar las anomalías de tal tipo, esto puede ser debido a que no disponemos de una cantidad de datos relevante de ciertos tipos entonces los valores de las medias para pocos espectros puede resultar engañoso. En el caso de los datos tipo #10 donde la media es superior, somos capaces de detectar en torno al 65 % con un resultado inesperado ya que (ver Figura 6.6) gran parte de los datos de este tipo son reconstruidos con una media inferior a datos con los que ha sido entrenado el autoencoder.

En el caso de autoencoders resulta interesante emplear diferentes funciones de pérdida como puede ser el error absoluto medio (MAE), vemos que el modelo detecta esta vez los datos de tipo "resina" aunque el desempeño baja para los datos de tipo "ápice".

Para finalizar, es razonable decir que el modelo de autoencoder tiene un gran potencial aunque el desempeño a la hora de detectar ciertos tipos de anomalías no es bueno, pero si así para anomalías de tipo "resina".

7.1. Líneas futuras

Los modelos de aprendizaje profundo brindan un gran número de posibilidades y de variantes a implementar para el estudio de espectros LIBS.

En el caso del clasificador, sería posible repetir el proceso de optimización de forma conjunta, implementando las capas de *batch normalization*. El modelo resulta interesante para el análisis idealmente in-situ y puede entrenarse para clasificar otro tipo de alimentos con el fin de evitar etiquetados de alimentos erróneos.

Las medidas de cada tipo fueron realizadas sobre una única patata por cuestiones de logística para este trabajo, aunque puede resultar interesante obtener datos de misma procedencia pero de patatas diferentes, teniendo en cuenta que las condiciones de medición afectan en los espectros finales.
Otro tipo de muestras podrían ser medidas con el fin de clasificar, fueron planteados combustibles, material bacteriológico... aunque la complejidad de realizar dichas mediciones aumenta.

En el modelo de autoencoder podemos tratar de cubrir diferentes tipos de funciones de pérdida, así como tratar de implementar un búsqueda con *optuna* donde tratamos de maximizar el número de detecciones correctas de anomalías.

Como mencionamos en la sección 3.3, este tipo de modelo tiene distintas utilidades, por ejemplo como eliminador de ruido [21] y eliminar la parte de radiación bremsstrahlung de forma automática para mejorar la precisión.

Bibliografía

[1] Russell, Stuart, J and Norvig P. *Artificial Intelligence: A Modern Approach*, 3rd edition, Prentice Hall, 2009. ISBN 0-13-604259-7

[2] García-Escárzaga, A.; Martínez-Minchero, M.; Cobo, A.; Gutiérrez-Zugasti, I.; Arrizabalaga, A.;P. *Using Mg/Ca ratios fron the limpet Patella depressa Pennant, 1777 measured by Laser Induced Breakdown Spectroscopy (LIBS) to reconstruct paleoclimate.* Appl. Sci. 2021, 11, 2959. `https://doi.org/10.3390/app11072959`

[3] L.Moreira Osorio, L.V. Ponce Cabrera and E. de Posada *Descubriendo la materia mediante luz láser. La espectroscopía de plasma inducido por láser (LIBS) como método para determinar la composición elemental de la materia*, 2012, EAE LAP LAMBERT Academic Publishing GmbH & Co. ISBN: 978-3-8473-6567-9.

[4] F.Anabitarte García, A. Cobo García and J.M. López Higuera *Laser-induced breakdown spectroscopy: fundamentals, applications, and challenges*, 2012, SRN Spectroscopy vol. 2012, article ID 285240, Hindawi Publishing Corporation.

[5] J. Torres, *Deep Learning, Introducción práctica con Keras*, 2018, Watch this space, Primera parte, ISBN: 978-1-983-12981-0

[6] R. Atienza, *Advanced Deep Learning with Keras* (2018). Ed: Packt Publishing Ltd. ISBN: 978-1-78862-941-6

[7] Goodfellow, I, Bengio, Y and Courville, A. 2016. *Deep Learning.* MIT Press. `http://www.deeplearningbook.org` Fecha de consulta: 15 de enero, 2022.

[8] `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy`. Fecha de consulta: 20 de diciembre, 2020.

[9] `https://keras.io/losses/` Fecha de consulta: 8 enero, 2022.

[10] `https://keras.io/optimizers/` Fecha de consulta: 8 enero, 2022.

[11] `https://colab.research.google.com/` Fecha de consulta: 5 de diciembre, 2021.

[12] `https://pandas.pydata.org/` Fecha de consulta: 10 de diciembre, 2021.

[13] Pedregosa, F. Varoquaux, G. Gramfort, A. Michel, V. Thirion, B. Grisel, O. Blondel, M. Prettenhofer, P. Weiss, R. Dubourg, V. Vanderplas, J. Passos, A. Cournapeau, D. Brucher, M. Perrot, M. and Duchesnay, E., *Scikit-learn: Machine Learning in Python* Journal of Machine Learning Research, volume 12, pages 2825–2830, 2011

[14] `https://numpy.org/` Fecha de consulta: 10 diciembre, 2021.

[15] `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html` Fecha de consulta: 20 diciembre 2021

[16] `https://www.optuna.org` Fecha de consulta: 7 diciembre, 2021.

[17] `https://keras.io/keras_tuner/` Fecha de consulta: 3 enero, 2022.

[18] Akiba, Takuya and Sano, Shotaro and Yanase, Toshihiko and Ohta, Takeru and Koyama, Masanori, *Optuna: A Next-generation Hyperparameter Optimization Framework*, Proceedings of the 25rd, ACM, International Conference on Knowledge Discovery and Data Mining, 2019.

[19] `https://keras.io/api/layers/normalization_layers/batch_normalization/` Fecha de consulta: 10 enero, 2022.

[20] `https://scikit-learn.org/stable/index.html` Fecha de consulta: 22 enero, 2022.

[21] S. Ye, Z. Niu, P. Yang and J. Sun, *A sparse autoencoder based denoising the spectrum signal in LIBS* 2018 Chinese Control And Decision Conference (CCDC), 2018, pp.3572-3577, DOI `10.1109/CCDC.2018.8407742`.