Universidad Zaragoza

Facultad de Ciencias Universidad Zaragoza

TRABAJO DE FIN DE GRADO
GRADO EN FÍSICA

Aprendizaje automático y espectroscopia de moléculas magnéticas

Autor:
Iván Montero Collado

Director:
David Zueco Láinez

Departamento de Física de la Materia Condensada

Zaragoza, 28 de junio de 2021

Índice de contenidos

1. Introducción.

En el grupo de investigación QMAD uno de los principales experimentos que se lleva a cabo consiste en medidas de transmisión a lo largo de circuitos superconductores sobre los que se depositan moléculas magnéticas para estudiar el acoplo entre luz y materia. Una de las muestras con las que más se trabaja y cuyos datos se han usado para este trabajo es el compuesto orgánico DPPH [1, 2]. Las moléculas de este compuesto presenta un radical libre, un electrón desapareado, de manera que posee un espín efectivo 1/2. La muestra se prepara en polvo y se coloca en una pastilla encima del circuito superconductor, por el que se hace circular una corriente alterna en el rango de las radiofrecuencias generando un campo magnético. Este campo magnético puede acoplarse a los radicales libres cuando la frecuencia de dicho campo coincide con la frecuencia de resonancia entre los niveles de la molécula, produciendo un decaimiento en la señal transmitida por el circuito. En este caso la molécula absorbe la radiación para producir excitaciones en sus niveles de energía. A través de la comparación de los voltajes medidos en los puertos de entrada y salida del circuito, se puede determinar dicha frecuencia de resonancia, permitiendo realizar espectroscopía en un intervalo de frecuencias entre 0 y 14 GHz [3]. Los picos de absorción que presentan estas medidas de transmisión pueden ajustarse a curvas de tipo Lorentz.

$$f(x) = A\frac{\gamma}{(x - x_0)^2 + \gamma^2} \tag{1}$$

1.1. Objetivos.

Este trabajo tiene como objetivo realizar un ajuste a los datos medidos en el experimento mencionado mediante el uso de técinas de aprendizaje automático (*machine learning*). Para lo que se empleará una red neuronal con arquitectura secuencial entrenada con datos generados de forma artificial (*synthetic data*). Dicha red neuronal será programada en el lenguaje *PYTHON* y se hará uso de la biblioteca *KERAS* [4] que a su vez se combina con la biblioteca matemática desarrollada por Google Brain, *TENSORFLOW* [5].

2. Fundamentos del aprendizaje automático.

Introduciremos desde lo elemental y de una forma compacta los principales elementos necesarios para comprender las técnicas del aprendizaje automático y en concreto, de las técnicas empleadas en nuestro caso. Para comenzar, daremos unas definiciones que normalmente se utilizan de forma indistinta y dan lugar a confusión.

Inteligencia artificial: Se define la inteligencia artificial como una subdisciplina del campo de la informática que busca la creación de máquinas que puedan imitar comportamientos inteligentes. Por ejemplo, un programa creado para que un robot realice un camino es un comportamiento aparentemente inteligente pese a que está programado y no puede ejecutar ningún tipo de tarea que no sea ese camino restringido [6].

Aprendizaje automático (*Machine Learning*): Es una rama dentro de la inteligencia artificial que estudia como dotar a las máquinas de capacidad de aprendizaje, entendido este como

la generalización del conocimiento a partir de un conjunto de experiencias. Es decir, que por ejemplo en el caso anterior el robot hubiera aprendido a desarrollar y seleccionar una ruta de forma independiente. En este documento a menudo nos referiremos a él con su abreviación: ML [6].

Redes neuronales (*Neural networks*): Son un tipo de algoritmos que permiten el aprendizaje de forma jerarquizada, posibilitando una mayor abstracción del conocimiento adquirido para generar aprendizaje. Entre otras cosas, las redes neuronales utilizan unos elementos llamados neuronas que se organizan de formas específicas en lo que llamamos arquitectura de la red, que permiten procesar la información. Es habitual encontrar que a veces se utiliza de forma indistinta el uso de la palabra "modelo" (tenemos un modelo que...) cuando se habla de una red neuronal que realiza una tarea concreta. No existe una diferencia entre ambos términos y en el presente documento se usan de forma indistinta [6].

Aprendizaje profundo (*Deep Learning*): Son las técnicas que emplean el uso jerarquizado de la información obtenida mediante redes neuronales más pequeñas y especializadas para la generación de un aprendizaje más complejo [6].

2.1. Neurona.

La neurona es la unidad básica de procesamiento que nos vamos a encontrar dentro de una red neuronal. Recibe este nombre por su analogía con las neuronas biológicas que todos conocemos ya que emula la elaboración de una respuesta ante un estímulo externo. Sin embargo, a nivel computacional, las neuronas no son otra cosa que funciones matemáticas; reciben uno o varios valores de entrada, *input* (el "estímulo externo"), con los que realizan un cierto cálculo interno y generan un único valor de salida o *output* (la "respuesta"). Para explicar el funcionamiento de las neuronas hablaremos primero del *perceptrón*, que es un modelo muy básico de neurona que engloba las principales características de éstas y es el más similar al comportamiento de una neurona biológica[1]. En el perceptrón tenemos que la neurona recibe entradas únicamente binarias y devuelve una sola salida binaria. Las entradas ($\vec{x}$) presentarán diferente importancia, quedando esta determinada por lo que se conoce como peso o *weight* ($\vec{\omega}$) y se realizará la suma ponderada de dicha entrada mediante $\vec{x} \cdot \vec{\omega}$. El resultado de esta operación se comparará con un umbral, si el resultado se encuentra por encima de dicho umbral la respuesta será 1, en caso contrario será 0. Sin embargo, por convenio es más habitual realizar $\vec{x} \cdot \vec{\omega} + b$ donde b se denomina *bias* o sesgo y no es otra cosa que el umbral, pero con signo opuesto. De esta forma lo que se compara es si el resultado de esta última operación es mayor o menor que cero y se devuelve como salida un 1 si es mayor que cero o

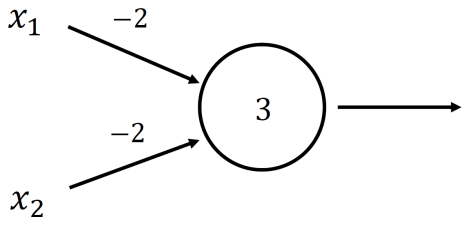

Figura 1: Visualización gráfica del comportamiento de una neurona. Para el caso de un perceptrón: $\vec{x}_1$ y $\vec{x}_2$ formarían parte de la entrada $\vec{x}$, en este caso ambos tienen la misma importancia (peso = -2), el valor de la operación $\vec{x}_1 \cdot (-2) + \vec{x}_2 \cdot (-2)$ será comparado con el sesgo que es 3.

[1] Las neuronas biológicas funcionan "disparándose" cuando reciben un estímulo suficientemente intenso.

un 0 si es menor. A esta operación realizada por la neurona se la denomina *función de activación*. Uno podría pensar que con los perceptrones, que imitan las neuronas biológicas, podemos crear sistemas que funcionen de forma similar a como lo haría un sistema de neuronas biológicas y que todo funcione correctamente. Sin embargo, en la sección 2.2.2 hablaremos de un método que usamos para "generar aprendizaje" y que emplea derivadas de la función de activación. Como se puede intuir, la función de activación de los perceptrones tiene forma de función escalón (o se activa o no se activa), por lo que no es útil cuando aplicamos el método que veremos en 2.2.2 [7, 8].

Vemos entonces que la función de activación elegida va a ser un componente importante en nuestra red. Las funciones de activación además, introducen un componente esencial en las redes neuronales, la no linealidad, esto les otorga la capacidad de generalizar cualquier función. Una prueba muy amigable y visual de este teorema de universalidad la podemos encontrar en el libro de Michael Nielsen [7] y una prueba más rigurosa y matemática en el artículo de George Cybenko [9]. Existen múltiples funciones de activación que van desde una versión suavizada de la función escalón que podemos asociar al perceptrón, como es la *neurona sigmoidea* o la *tanh*, hasta funciones más sofisticadas como *Softmax*, que transforma las salidas a una representación en forma de probabilidades [8]. Más adelante veremos que para nuestro caso necesitamos una función de activación que devuelva valores reales mayores o iguales a cero, es por esto que se ha optado por utilizar la función de activación ReLU, cuya expresión es:

$$f(x)_{RELU} = max(0, x) = \begin{cases} 0 \text{ para } x < 0 \\ x \text{ para } x \geq 0 \end{cases} \tag{2}$$

De esta forma nos aseguramos un valor de salida real y positivo o nulo. Sin embargo, en una red neuronal no vamos a tener una única neurona, tendremos un conjunto de ellas donde la forma en la que las organicemos definirá lo que llamamos arquitectura de la red y el comportamiento de la misma. Existen muchos tipos de arquitecturas, cada una al igual que las neuronas, adaptada para funciones concretas donde su funcionamiento destaca por encima del de otras. En nuestro caso, utilizaremos el modelo de red secuencial, que es un modelo sencillo, pero potente.

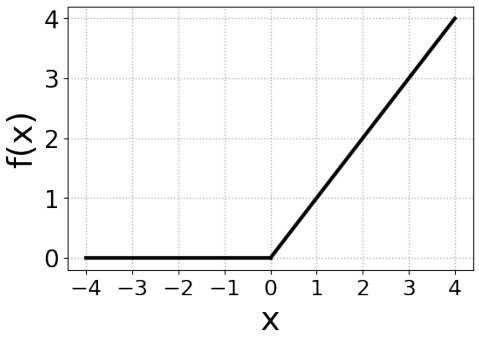

Figura 2: Gráfico de la función ReLU.

2.2. Modelo secuencial.

En este modelo las neuronas se "apilan" en capas y en cada una tendremos un grupo de neuronas que se encargará de procesar la entrada que le llegue. Por convenio se definen 3 tipos principales de capas: la capa de entrada (*input layer*), las capas ocultas (*hidden layers*) y la capa de salida (*output layer*). Con mayor o menor variación tanto en número de capas como en número de neuronas esto sería la arquitectura básica en este tipo de modelos. Podemos ver una representación gráfica de una red secuencial en la figura 3. Generalmente su diseño suele ser bastante sencillo e

intuitivo. Por ejemplo, si procesáramos una imagen de 32x32 bits para determinar qué dígito (del 0 al 9) contiene, la capa de entrada contendría un total de $32 \cdot 32 = 1024$ neuronas y la de salida 10 neuronas en donde cada una representaría un dígito. En cuanto al número de capas ocultas o la cantidad de neuronas de éstas no existen a día de hoy métodos cuantitativos concluyentes que puedan arrojar siempre valores fiables, la comunidad suele mencionar que establecer este tipo de parámetros de forma correcta "es un arte" [7].

Si miramos a la figura 3 podemos ver que los valores de salida de una capa son tomados como entrada de la capa siguiente, a este tipo de redes se las denomina *feedforward*[2]. Se podrían realizar modificaciones para que, por ejemplo, se tome como entrada de una capa una salida posterior, creando bucles donde la entrada depende de la salida, o donde las neuronas se activan solo de forma temporal, pero este tipo de situaciones no nos interesan para nuestra tarea.

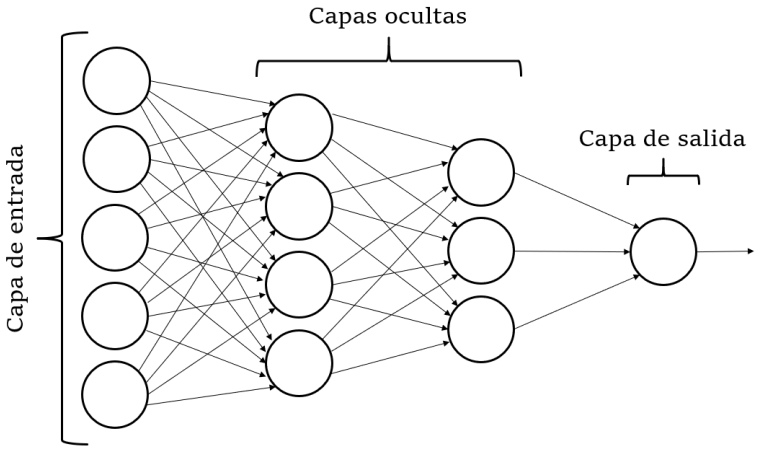

Figura 3: Ejemplo de arquitectura secuencial donde se han indicado las capas de entrada, ocultas y de salida. Aunque veamos que de una neurona salen varias flechas indicando que manda información a varias neuronas, esa información es la misma pues ya habíamos dicho que cada neurona tiene un único output.

2.2.1. Función de pérdidas - función de coste.

Llegados a este punto sería natural preguntarse cómo la unión que planteamos de neuronas, definidas en la sección 2.1, puede generar "aprendizaje". Sin filosofar demasiado sobre qué consideramos aprendizaje diremos que para nuestro caso en particular el aprendizaje se podría resumir en dar una respuesta correcta, es decir, necesitaremos encontrar los pesos y sesgos apropiados para que al introducir unos datos de entrada en nuestra red obtengamos una salida satisfactoria. Para ello introducimos el concepto de función de pérdidas o *loss function*; la función de pérdidas tiene como finalidad determinar el error entre el valor estimado por nuestra red para un dato y el valor real. Definimos además función de coste o *cost function* como un promedio de varios cálculos de la función de pérdidas. A menudo estos términos se usan indistintamente debido a un abuso del lenguaje, en este documento hablaremos principalmente de la función de pérdidas. Existen muchas funciones de este tipo ya definidas e implementadas para su fácil uso en ML que

[2]Este término no tiene una traducción como tal en la lengua española, sería algo parecido a "alimentación directa".

van desde cosas sencillas como el *error absoluto medio* (MAE) hasta cosas más refinadas como la *entropía cruzada binaria* (BCE). Nosotros emplearemos la función error absoluto medio pues es una función sencilla que no conlleva mucho coste computacional y da buenos resultados [7].

$$MAE = \sum_{i=1}^{n} \frac{|\hat{y}_i - y_i|}{n} \tag{3}$$

Aquí $\hat{y}_i$ refiere a los valores reales que querremos que nuestro modelo prediga y que estarían codificados en nuestra entrada, y_i serían las respuestas proporcionadas por nuestra red como función de los pesos y sesgos y n sería el número de muestras.

Recapitulando: tenemos una serie de parámetros (pesos y sesgos) que influyen en el cálculo de una salida la cual será comparada con una respuesta correcta y queremos que la diferencia entre ambas sea la mínima posible. Nos enfrentamos a un problema de minimización.

2.2.2. Optimizadores.

Para afrontar este problema de minimización emplearemos lo que se conoce como *optimizers* u optimizadores. Los optimizadores serán los algoritmos encargados de ir modificando los pesos y sesgos para minimizar la función de pérdidas y parten de la idea del método de *backpropagation* (retropropagación). Este método nos permite comprender y cuantificar cómo un cambio en los pesos y sesgos impacta en el funcionamiento de la red.

Fundamentos de la retropropagación.

Esencialmente su funcionamiento se basa en calcular el gradiente de la función de pérdidas por cada parámetro de la red para modificar dicho parámetro de forma negativa y proporcional al cálculo del gradiente. Esto resulta bastante intuitivo pues el gradiente apunta en la dirección en la que una magnitud varía de forma positiva con respecto a otra, en este caso, la dirección en la que nuestra función de pérdidas aumenta si variamos ω_k (pesos) o b_k (sesgos), luego modificamos en dirección opuesta.

$$\omega_k \ \rightarrow \ \omega_k' = \omega_k - \eta \frac{\partial C}{\partial \omega_k} \tag{4}$$

$$b_k \ \rightarrow \ b_k' = b_k - \eta \frac{\partial C}{\partial b_k} \tag{5}$$

El factor η es un factor que se conoce como *learning rate* (tasa de aprendizaje) y es un modulador en la intensidad de aprendizaje, definiendo como de grandes queremos que sean los saltos que modifiquen nuestros parámetros [7].

Este método además parte de 2 suposiciones; La primera es que la función coste pueda escribirse como un promedio sobre las funciones de pérdidas para un ejemplo de entrenamiento individual:

$$C = \frac{1}{n} \sum_{x=1}^{n} C_x \tag{6}$$

Donde C representa la función de coste y C_x representa la función de pérdidas para el elemento x. La necesidad de realizar esta suposición es que, como veremos unos párrafos más adelante,

el algoritmo de retropropagación tan solo nos deja computar las derivadas parciales para un ejemplo de entrenamiento.

La segunda suposición es que la función de pérdidas se pueda escribir como función de los resultados de la red neuronal. Esto último es fácil de ver si recordamos la expresión 3, en ella se compara el valor real con el predicho por la red y este valor es el resultado de nuestra función de activación en la última capa.

Con todo esto vemos que el problema ahora mismo se basa en calcular la función de pérdidas para cada elemento de nuestro conjunto de datos, calcular las derivadas parciales de esta función para determinar los cambios en los pesos y los sesgos y modificar dichos parámetros para así reducir el valor de la función de pérdidas. El cómo calculamos estas derivadas parciales nos lo dirán las 4 ecuaciones principales del algoritmo de retropropagación. Pero antes de presentarlas explicaremos la notación empleada; Cuando hablemos del peso de una conexión entre la neurona i de la capa $l-1$ y la neurona j de la capa l escribiremos ω_{ij}^l, los subíndices indican neuronas y superíndices capas, luego el sesgo de la neurona i en la capa l será b_i^l. Definimos entonces como *weighted input* o entrada pesada[3] de la neurona j en capa l a:

$$z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l \tag{7}$$

De forma que si llamamos σ a la función de activación de las neuronas, la activación de la neurona i de la capa l será:

$$a_i^l = \sigma(z_i^l) \tag{8}$$

Y vemos que en (7) la entrada pesada depende de la activación en la capa anterior. Definimos el error δ_j^l de la neurona j en la capa l como la variación de C respecto a la entrada pesada:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{9}$$

Para aligerar un poco la notación, de ahora en adelante cuando se prescinda del subíndice se estará indicando el vector que engloba a todos los valores de las neuronas de la capa (para el caso de los pesos será una matriz). Es decir, δ^l será el vector de errores asociados a la capa l.

Dicho todo esto presentamos las ecuaciones del algoritmo de retropropagación:

$$\delta^l = \frac{\partial C}{\partial a^l} \odot \sigma'(z^l) \tag{10}$$

$$\delta_j^l = \frac{\partial C}{\partial b_j^l} \tag{11}$$

$$\frac{\partial C}{\partial \omega_{jk}} = a_k^{l-1}\delta_j^l \tag{12}$$

$$\delta^l = ((\omega^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{13}$$

Estas ecuaciones parten principalmente de la regla de la cadena y nos muestran que podemos relacionar el error con diferentes derivadas parciales. La ecuación (10) se obtiene realizando la

[3]Notar que esta entrada pesada no es otra cosa que la versión generalizada de la operación que mostrábamos en el apartado 2.1.

derivada parcial de la función de pérdidas o función de coste respecto a la entrada pesada, para lo que aplicamos la regla de la cadena donde $\frac{\partial C}{\partial a^l}$ es la derivada parcial respecto a la activación de la capa l y $\sigma'(z^l)$ es la derivada de la función de activación respecto a la entrada pesada. La ecuación (11) es la misma derivada parcial, pero expresada en términos de la derivada parcial respecto al sesgo, aquí se tiene en cuenta que $\frac{\partial b^l}{\partial z^l} = 1$. Nuevamente la ecuación (12) la obtenemos realizando la derivada parcial y aplicando la regla de la cadena, pero esta vez diferenciando respecto a los pesos y despejando el término $\frac{\partial C}{\partial \omega_{jk}}$. Finalmente la ecuación (13) se obtienen reescribiendo la expresión (9) y expresándola en términos del error en la capa siguiente empleando la ecuación (7). Para acabar añadir que en las ecuaciones (10) y (13) el operador $\odot$ representa el producto *Hadamard* que es un producto elemento por elemento de los vectores, sin llegar a sumar que es como lo haríamos en un producto escalar.

Vemos entonces que estas ecuaciones nos permiten calcular cómodamente los cambios en los pesos y sesgos con el siguiente procedimiento: realizamos un *feedforward* completo donde calculamos todas las activaciones y finalmente el error en la última capa, luego por medio de la ecuación (13) propagamos el error hacia atrás (de aquí el nombre) y finalmente modificamos los parámetros según (4) y (5) gracias a (11) y (12) [7, 8].

Descenso de Gradiente Estocástico con mini-lotes.

Una de las formas más conocidas y usadas a la hora de implementar el algoritmo de retropropagación es el método por descenso del gradiente estocástico o SDG (*Stochastic Gradient Descent*). En este método se van tomando datos de forma aleatoria de un conjunto de datos de entrenamiento y se va aplicando el algoritmo de retropropagación tal y como hemos explicado en el párrafo anterior. Sin embargo, nosotros utilizaremos una variante de este conocida como descenso de gradiente estocástico con mini-lotes, de ahora en adelante y por abreviar: lotes (*batches*). En esta variante tomamos los datos de forma aleatoria, pero por lotes y dentro de cada lote aplicamos el algoritmo de retropropagación para cada elemento del mismo. De esta forma el cambio en un parámetro cualquiera vendrá dado por el promedio de los resultados de la retropropagación a cada elemento del lote para ese parámetro:

$$\omega^l \rightarrow \omega^l - \frac{\eta}{m} \sum_x^m \delta^{x,l} (a^{x,l-1})^T \tag{14}$$

$$b^l \rightarrow b^l - \frac{\eta}{m} \sum_x^m \delta^{x,l} \tag{15}$$

Una vez hemos recorrido todos los datos mediante la selección aleatoria de lotes decimos que hemos finalizado una época (*epoch*). Cuantas más épocas entrenemos más veces recorreremos los datos modificando los parámetros y más podremos acercarnos al mínimo de la función de pérdidas [8].

Adam.

Adam es unos cuantos pasos posteriores en cuanto a refinamiento de las técnicas de optimización. Esta técnica se suele presentar como una combinación de otras 2 técnicas: Adaptative Gradient Moment (AdaGrad) y Root Mean Squared Propagation (RMSProp). De las cuales combina

sus características más destacables: tener tasas de aprendizaje independientes por parámetro y que éstas se adapten en función del promedio de magnitudes recientes de los gradientes. Para entender la utilidad de estas características uno puede visualizar gráficamente el problema de minimizar la función de pérdidas como encontrar el mínimo en una superficie donde sabemos que no tendremos las mismas pendientes en todas las direcciones. El hecho de almacenar el histórico de los gradientes de pasos anteriores nos permite tener cierta memoria del recorrido que hemos seguido, impidiendo que nuestro progreso se quede atascado en aquellas situaciones que engloben puntos singulares. Es como si estuviéramos almacenando información acerca de la inercia de una pelota que desciende por la superficie de la que hablamos, pues no tendría sentido que si la pelota alcanzara un valle se parara en seco sin antes balancearse un poco. Por motivos evidentes utilizaremos este, aunque no presentaremos sus procesos internos debido a su complejidad y extensión.

2.3. Paradigmas de aprendizaje.

Una vez introducidos los elementos principales de neurona, arquitectura secuencial y retropropagación debemos pensar de qué formas podemos emplearlos para realizar tareas y es que al final, el objetivo de este tipo de técnicas es transformar la información de la que disponemos en conocimiento. Para esto hablaremos de los 3 principales paradigmas de aprendizaje donde englobamos todas las técnicas de machine learning: aprendizaje supervisado (*supervised learning*), aprendizaje no supervisado (*unsupervised learning*) y aprendizaje reforzado (*reinforced learning*). Nosotros nos centraremos en los dos primeros.

En el **aprendizaje supervisado** la base reside en el hecho de que nosotros suministramos a la red tanto los datos que queremos que clasifique (o de los que queremos que aprenda algo) como el resultado o etiqueta que queremos que asocie con cada dato, lo que se conoce como *dataset*[4]. Por ejemplo, suministramos una serie de fotos en las que solo hay objetos o bien de color rojo o bien azul, cada foto con una etiqueta: "rojo" o "azul". Entonces procedemos a entrenar la red donde le vamos mostrando las fotos junto con su etiqueta. La finalidad de este proceso es conseguir que la red, después del proceso de aprendizaje, sepa distinguir una imagen que contenga objetos rojos o azules sin haberle proporcionado la etiqueta. Como hemos participado de una forma directa en el aprendizaje (le hemos ido "chivando" las respuestas), decimos que es supervisado [10].

En el **aprendizaje no supervisado** el proceso es diferente. Aquí suministramos un dataset, pero no entregamos ninguna etiqueta o respuesta asociada a los datos de entrada. La idea con este paradigma de aprendizaje es que la propia red generalice de alguna forma el conocimiento sin que desde fuera se le diga a qué conclusión tiene que llegar. Este tipo de aprendizaje se ha comprobado que funciona especialmente bien para los problemas que denominamos de "clustering" (en español, clusterización), que se basan en la identificación de propiedades comunes que puedan separar grupos de elementos para que los elementos de un grupo sean similares entre entre ellos y diferentes con los demás grupos [10].

Tanto en los modelos de aprendizaje supervisado como no supervisado debemos conocer además los conceptos de *training data* (datos de entrenamiento) y *validation data* (datos de validación)

[4]De ahora en adelante nos referiremos al conjunto de datos y etiquetas como *dataset*, su traducción al inglés, para abreviar.

que son subgrupos del dataset que empleamos para entrenar la red. Los datos de entrenamiento forman un subgrupo mayor que se emplea propiamente para ir entrenando la red y aplicar los optimizadores que vayan reduciendo la función de pérdidas, mientras que los datos de validación forman un subgrupo menor sobre el cual se va probando el funcionamiento de la red al final de cada época. Cuando se prueba sobre los datos de validación no se modifican los pesos ni los sesgos. Se suele definir un tercer subgrupo llamado *test data* (datos de prueba), tiene una función similar a los de validación, pero este conjunto se probaría una vez ya entrenada la red y es ajeno al dataset original [7].

Lo curioso de los modelos de ML bien sigan un aprendizaje supervisado o no supervisado es que nosotros no sabemos qué aprenden ni cómo lo aprenden. Sabemos que los parámetros de nuestra red se van modificando para alcanzar una mínima discrepancia entre predicción y valor real, pero no sabemos cuánto valen los parámetros ni a qué le está dando más importancia la red para determinar respuestas que nosotros decidamos tomar por correctas. No obstante, esto no quiere decir que no nos interese saber qué sucede con los pesos y sesgos del modelo, tan solo que muchas veces lo único que nos interesa saber es cómo se relaciona el modelo con los datos de entrada y las salidas que devuelve. De hecho, en modelos mucho más complejos que el que aquí se presenta, para trabajar en la comprensión a nivel interno de los modelos, existen algoritmos muy sofisticados que permiten visualizar la activación de las neuronas en base a las entradas introducidas con la esperanza de que esto nos permita comprender más acerca del procesamiento de la información o incluso de los procesos psicológicos humanos [8][5].

3. Nuestro modelo.

3.1. Entrenamiento del modelo.

Típicamente una red (o modelo) ha de ser entrenada con datasets muy grandes (del orden de miles de datos) por lo que utilizar aprendizaje supervisado no siempre es una opción viable, ya que alguien ha tenido que clasificar esos datos previamente al aprendizaje. Es decir, estamos hablando de que necesitamos personas que realicen un trabajo manual generando y/o clasificando datos, que requeriría de un esfuerzo y tiempo considerables. Incluso podríamos decir que no siempre entrenar es posible porque directamente no se dispone de un dataset lo suficientemente grande. Podría ser que el simple hecho de generar datos implicara altos costes económicos no siempre asumibles, esto nos limitaría el uso hasta de los métodos no supervisados. No obstante, existe una alternativa conocida como generación de datos sintéticos (junto con sus etiquetas, para el caso supervisado). La investigación en *synthetic data* [11] (datos sintéticos) es una línea clave en ML ya que para entrenar modelos o probarlos se necesitan datos, a veces protegidos por cuestiones legales[6], y no unos datos cualesquiera, sino unos datos que sean capaces de emular el comportamiento real que se esperaría de unos datos reales. Disponer de un buen dataset es el

[5] Con esto nos referimos a los llamados espacios latentes que permiten visualizar hasta cierto punto el conocimiento aprendido por el modelo.

[6] Por ejemplo, si uno quisiera entrenar una red neuronal que ayudara a clasificar de forma inteligente a pacientes según sus patologías, los datos médicos no se podrían conseguir al estar protegidos por la ley de protección de datos sanitarios, creando así una necesidad de generar datos falsos para entrenar y probar los modelos.

primer "cuello de botella" que enfrenta el entrenamiento de modelos de ML.

En nuestro caso disponemos de tan solo 31 curvas (lorentzianas con ruido) medidas en el laboratorio, que además, se nos han dado remarcando que su obtención ha llevado de considerable tiempo para su correcta visualización. Como es de esperar, 31 datos no nos sirven como dataset de entrenamiento, por no decir que si quisiéramos emplear métodos supervisados deberíamos conocer los parámetros de posición de pico, anchura y amplitud reales de cada curva para poder comparar con algo. Pero si nosotros estudiáramos cada lorentziana para obtener los parámetros que luego suministrar al modelo, realmente ya no necesitaríamos el modelo para nada porque ya tendríamos la información que buscamos. La solución que nosotros proponemos es la generación de un dataset sintético de curvas lorentzianas con ruido que simulen datos obtenidos en el laboratorio. Al haber generado nosotros cada curva podemos almacenar la información relevante y emplearla para el entrenamiento del modelo.

3.1.1. Datos sintéticos.

Así comenzamos enfrentando el primer reto: necesitamos generar un dataset lo suficientemente grande y bueno como para entrenar nuestro modelo. El dataset que nosotros generaremos estará discretizado y normalizado. Normalizaremos los datos de forma que sea más fácil para nuestro modelo generar un factor de normalización al cual nos referiremos como *amplitud* de ahora en adelante. Lo hacemos así pues matemáticamente tendrá el papel de A en la ecuación (1). Y discretizaremos porque, a priori, no nos preocupará la distancia entre los puntos del eje X. Esto se debe a que el modelo va a recibir un vector de datos y "le da igual" si entre cada dato hay 0.3GHz o 7K de diferencia, simplemente almacena cada dato en una posición de memoria que luego procesará y ajustará. Es decir, si consideramos a la expresión (1) la lorentziana en su versión discretizada con x, x_0, $\gamma \in \mathbb{N}$, estaríamos representando $f(x)$ respecto de x y para obtener la curva real haríamos $F(x) = \Delta x f(x)$ donde Δx se obtendría de los datos experimentales. Esto nos permite entrenar la red de forma cómoda generando lorentzianas con tantos puntos como puntos tengan las medidas de laboratorio sin preocuparnos por unidades. Como último comentario mencionar que nosotros trabajaremos con las medidas de *absorción* (= 1 - transmisión) y no de *transmisión* únicamente con fines estéticos a la hora de presentar los resultados.

Como generamos cada lorentziana de igual tamaño a los datos de laboratorio nuestras curvas constarán de 172 puntos. Cada lorentziana estará generada con posición de pico aleatoria dada por una distribución uniforme de probabilidad, donde se tendrá en cuenta que los picos no queden demasiado pegados a los extremos situándolos como mínimo a una distancia $\gamma/2$ de estos. En cuanto a la anchura de pico (γ), generaremos un valor aleatorio uniforme entre un ancho máximo y uno mínimo siendo estos 25 y 1 respectivamente (recordar que hemos discretizado). Luego generamos un ruido que añadiremos punto por punto a la lorentziana. La generación de este ruido vendrá dada por 3 opciones de las cuales se selecciona 1 de forma aleatoria para cada curva: ruido plano y uniforme generado entre -0.5 y 0.5, ruido gaussiano centrado en 0 y la suma de estos 2. Cuando se genera el ruido este se ajusta tomando como escala el ancho de la lorentziana y luego se controla con un factor que se selecciona de forma aleatoria entre un mínimo, que es 0, y un máximo, que es 0.17. Finalmente se normaliza el resultado.

Con este dataset sintético autogenerado tenemos por un lado la posición de pico, anchura y amplitud con las que construímos las lorentzianas, las propias lorentzianas y el ruido que añadimos

a las lorentzianas. Esto nos da cierta libertad para decidir con qué paradigma queremos afrontar el problema. En cuanto al tamaño del dataset esta cuestión se analiza con detalle en la sección 3.2.1.

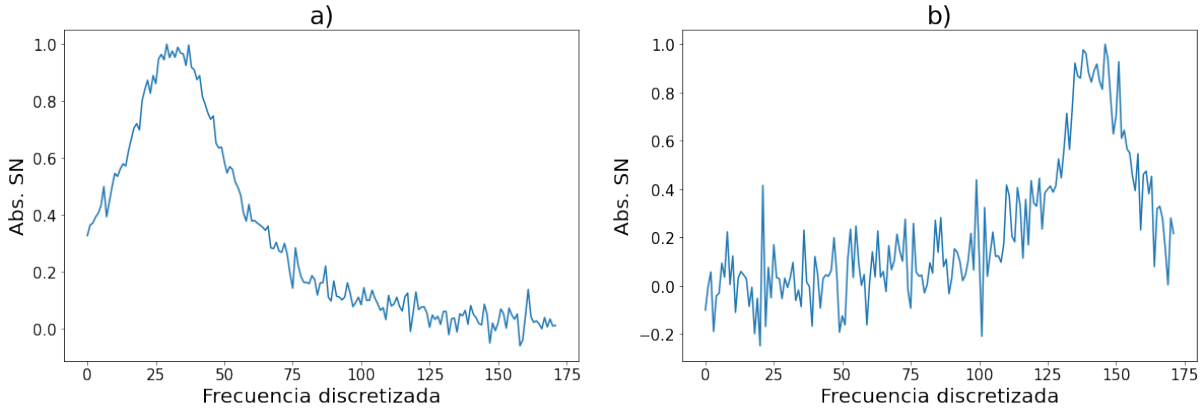

Figura 4: Dos ejemplos del dataset autogenerado. En el eje X estaríamos representando la frecuencia de forma discretizada por lo que no se incluyen unidades. En el eje Y estaríamos representando las medidas de absorción simuladas por nosotros y normalizadas donde Abs. SN quiere decir *Absorción Simulada Normalizada*.

3.1.2. Aprendizaje no supervisado.

Nuestro primer acercamiento al problema fue de forma no supervisada. El primer planteamiento partía de un modelo secuencial con 3 capas: *capa de entrada - capa oculta - capa de salida*. En cuanto al número de neuronas por capas se ha mencionado ya que no existen técnicas concluyentes a día de hoy, luego emplearemos una heurística popular que establece que para el caso de una función no lineal basta con establecer tantas neuronas en la capa de entrada como puntos tenga nuestra función de entrada y en la capa oculta, 2/3 de este valor[7]. Para la capa de salida establecemos 3 neuronas donde esperamos que se extraiga la información de posición de pico, anchura y amplitud. Ahora, si recordamos la expresión (6) vemos que el cálculo de la función de pérdidas se realiza con la salida de nuestra red en su capa de salida (predicción del modelo) y los valores reales ("respuesta correcta"). En nuestro caso, la red devuelve un vector de 3 componentes y la "respuesta correcta", al estar empleando aprendizaje no supervisado, sería la lorentziana completa que recibe como input. Es decir, queremos que la red nos diga la lorentziana que mejor se adapta a los datos de entrada, luego para poder realizar el cálculo de la función de pérdidas necesitamos reconstruir la lorentziana a partir de la predicción de nuestra red y así "comparar" ambas curvas. La idea aquí era programar una función de pérdidas personalizada que antes de realizar el cálculo reconstruyera la lorentziana, pero resultó ser inviable debido a la naturaleza de la clase *loss function* ya definida en *KERAS*. Esta clase, aunque permite programar una función de pérdidas personalizada, impone una restricción que obliga a introducir 2 vectores de igual tamaño como entrada. Al no existir una forma de reconstruir la lorentziana entre salida de la capa de salida y entrada de la función de pérdidas esta vía se dio por fallida.

Esto nos lleva al segundo intento, esta vez con un modelo secuencial de 4 capas: *capa de entrada - capa oculta - capa oculta - capa de salida*. En este modelo la capa de entrada y de salida con-

[7]El número de neuronas de la capa oculta se tratará más adelante en el proceso de optimización.

tendrían el mismo número de neuronas e igual al número de puntos de la lorentziana, la primera capa oculta 2/3 de la capa de entrada y la segunda capa de entrada constaría de 3 neuronas que nuevamente representarían la posición de pico, anchura y amplitud. Este segundo intento se inspiraba en un tipo de algoritmos que se conocen como *autoencoder*[8] [8, 12]. En un autoencoder el tamaño de las capas (entendiendo tamaño de capa como el número de neuronas que tiene) se va reduciendo hasta llegar a un tamaño mínimo donde se busca que la información pase por un "cuello de botella" para luego volver a aumentar el tamaño de las capas hasta igualar el tamaño inicial y esperar que se reconstruya dicha información. Este proceso pretende que la red comprima la información de entrada, en la capa de tamaño mínimo, de forma que se codifique en sus características más importantes y que más adelante le permitirán reconstruirla. De primeras puede resultar un poco redundante, pero este tipo de redes tienen bastantes aplicaciones especialmente en el tratamiento de imágenes, un ejemplo sería el caso de Deep Fakes que permite reconstruir rostros [13]. Este segundo intento pretendía encontrar las características fundamentales de las lorentzianas con la esperanza de que estas características fueran la posición de pico, anchura y amplitud (segunda capa oculta con 3 neuronas). De esta forma se solucionaba el problema de la dimensionalidad reconstruyendo la lorentziana (sin ruido) entre la segunda capa oculta y la capa de salida, así tendríamos 2 vectores de igual tamaño que introducir en la función de pérdidas. Este segundo intento también fracasó porque la reconstrucción de una lorentziana en un paso intermedio en un modelo secuencial tampoco resultó ser viable. Uno podría pensar que tiene sentido pues lo natural en un modelo de ML es precisamente que no sepas qué está sucediendo a nivel interno. No obstante, la realización de una operación concreta entre capas en un modelo secuencial es posible por medio de un tipo de capa especial conocida como *lambda layer* (capa lambda). Sin embargo, el problema residía en un tipo de objetos especiales de *TENSORFLOW* conocidos como *tensor*. Estos son los objetos que se intercambian las capas de la red y son inmutables e imponen muchas restricciones a la hora de operar con ellos. Por ejemplo, multiplicar la salida de nuestra segunda capa oculta por 2 y que lleguen los valores modificados a la última capa no sería un problema porque únicamente modificaríamos los propios valores de salida que seguirán su flujo *feedforward* sin problema (no estamos modificando pesos ni nada). Sin embargo, reconstruir una lorentziana a partir de esos 3 valores y pedirle a la red que almacene en cada neurona de salida un punto de la lorentziana directamente ignora la propia construcción de pesos y sesgos de la red. Por no decir que en redes *feedforward* la salida de cada neurona se manda a todas las neuronas siguientes y nosotros queríamos que a partir de 3 neuronas llegara un único valor a cada neurona de salida. En definitiva, inviable.

Llegamos así al último y tercer intento mediante aprendizaje no supervisado. Este intento se abandonó rápidamente ya que la idea era crear nuestro propio tipo de capa personalizada para satisfacer todas nuestras necesidades técnicas y esta tarea era inviable en el tiempo del trabajo.

Haber empleado un modelo con aprendizaje no supervisado habría sido interesante pues se habrían podido emplear muchas curvas sin necesidad de conocer sus parámetros previamente. Esto nos habría permitido introducir algunos datos de laboratorio en los datos de entrenamiento para aumentar la eficacia. Sin embargo, dado que tenemos un dataset autogenerado y conocemos las posiciones de pico, anchura y amplitud, podemos crear una matriz que contenga todas las etiquetas asociadas a cada lorentziana. Esto nos permite emplear un método supervisado.

[8] Este término tampoco tiene traducción como tal en nuestro idioma, lo más similar sería "autocodificación", pero utilizaremos autoencoder por comodidad y falta de fiabilidad en la traducción.

3.1.3. Aprendizaje supervisado.

Habiendo descartado el aprendizaje no supervisado pasamos a afrontar el problema desde el aprendizaje supervisado. Aquí lo que se plantea es ligeramente diferente pues nos aprovecharemos de que al haber generado los datos de forma sintética disponemos de la información de cada lorentziana (posición de pico, anchura...). La idea ahora reside en que la red, al igual que en el ejemplo de las fotos con objetos rojo o azul, etiquete cada dato con 3 números que se compararán con las etiquetas reales que nosotros conocemos. De esta forma convertimos nuestro problema en algo más similar a un problema de clasificación.

Para este acercamiento comenzamos planteando una arquitectura de red secuencial con 3 capas: *capa de entrada - capa oculta - capa de salida*. El número de neuronas por capa sería equivalente al primer intento no supervisado. La gran diferencia ahora es que vamos a aplicar la función de pérdidas a los vectores de 3 componentes que contienen la información de amplitud, posición de pico y anchura en vez de a la lorentziana en sí. Esto soluciona el problema del tamaño de los vectores que recibe la función pérdidas de keras y tampoco requiere que reconstruyamos la lorentziana. Además, notar que no nos debe preocupar qué neurona de las 3 de salida sea la que refiera a la amplitud, anchura o a la posición de pico, porque esto quedará determinado por el orden en que hayamos guardado las etiquetas al generar el dataset. Cuando comencemos a aplicar los algoritmos de entrenamiento mediante retropropagación todo el comportamiento de la red se irá modificando para minimizar la función de coste (que recordamos que en nuestro caso es la expresión (3) con dos vectores de 3 componentes donde uno es la predicción de la red y el otro las etiquetas generadas), lo que hará que cada neurona vaya aprendiendo a devolver los valores que más se asemejen a la etiqueta asociada. Con este planteamiento la red funcionaba de forma prometedora y ahora el objetivo consistía en construir el modelo y optimizar su funcionamiento.

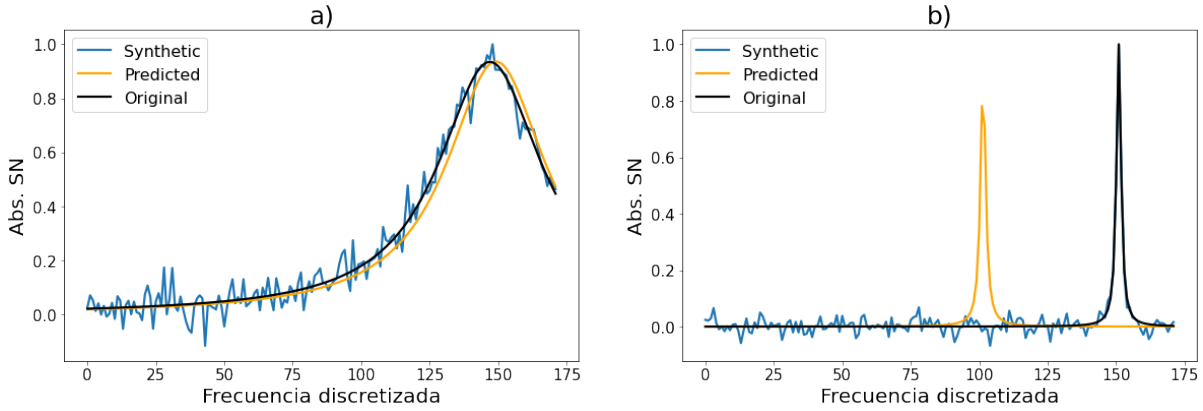

Figura 5: Dos ejemplos del funcionamiento de un primer modelo con $N_{\text{in}} = 172$ neuronas, $N_{\text{hi}} = 115$ neuronas y Lote=300 entrenado con un datset de 10.000 datos a lo largo de 20 épocas. *Azul = Synthetic*, son los datos que nosotros generamos; *Naranja = Predicted*, nos indica la predicción de la red; *Negro = Original*, es la curva original empleada para generar la curva synthetic y la que esperamos que el modelo prediga. Se puede ver en el ejemplo b) que el modelo todavía está lejos de un correcto funcionamiento. El ejemplo a), por el contrario, muestra indicios de la posibilidad de realizar un correcto ajuste.

3.2. Construcción del modelo.

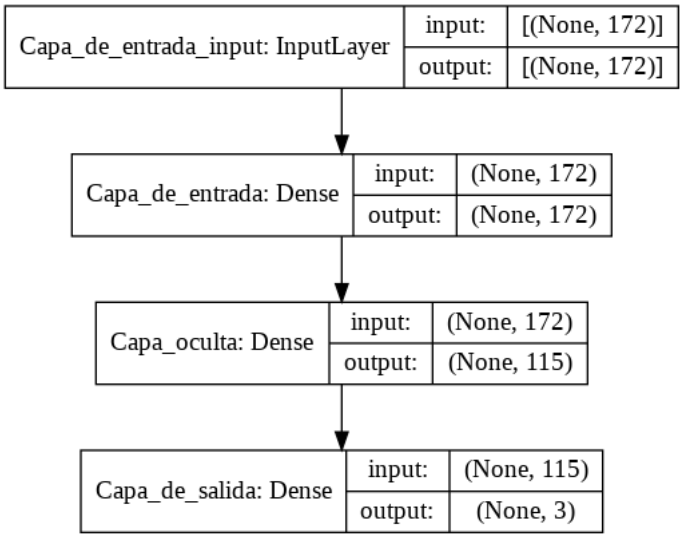

Figura 6: Esta imagen es un diagrama (obtenido por medio de la librería de *KERAS*) que permite visualizar de forma sencilla la arquitectura y el flujo de información entre capas de un modelo. Este flujo se realiza mediante unas variables conocidas como *tensors* que emulan los tensores matemáticos conocidos. El diagrama nos indica con "Capa_de_entrada_input" la forma que tiene el input (datos de entrada) que llega a nuestra verdadera "Capa_de_entrada". Si nos fijamos, identifica a la primera como "InputLayer", esto es debido a que se hace así por convenio y nos indica que esta capa no contiene parámetros entrenables, tan solo la información respecto a la forma del input (en este caso un tensor (None, 172)). Las siguientes capas aparecen como "Dense", esto nos indica que estas capas están dotadas de neuronas y de parámetros entrenables. En estas capas, las cajas que aparecen a su derecha, *input* y *output*, nos indican la forma que tienen los tensores que reciben y la forma que deben tener al salir. Por ejemplo, a nuestra "Capa_oculta:Dense" nos llega un tensor de la forma (None, 172) y se manda como salida un tensor de la forma (None, 115). En los tensores el primer índice refiere al tamaño del lote, por eso hasta que no comencemos a entrenar el modelo aparece como *None* pues el modelo no sabe con qué tamaño de lote se le pedirá entrenar, el segundo índice refiere al número de neuronas o bien de la capa anterior (input) o bien de la capa actual (output).

Partimos de un modelo en el que recordamos que el número de neuronas de la capa de entrada, N_{in}, según la heurística adoptada coincidía con el número de puntos que tienen las lorentzianas de nuestro dataset[9], luego $N_{\text{in}} = 172$. El número de neuronas de la capa oculta cumplía que $N_{\text{hi}} = 2/3 \cdot N_{\text{in}}$, luego $N_{\text{hi}} = 115$. Finalmente para la capa de salida $N_{\text{out}} = 3$ como ya se ha ido mencionando. Podemos observar una representación del modelo en la figura 6.

Nuestro objetivo ahora es determinar los parámetros que hagan que nuestro modelo funcione de la mejor manera posible. Para ello primero partimos de unos valores arbitrarios en base a las recomendaciones populares[10] ya que con los modelos de ML siempre se destaca la individualidad de cada problema. En la figura 5 mostramos dos ejemplos del funcionamiento del modelo, entrenado

[9] Nuestro dataset además estará adaptado para tener la misma forma que los datos que mediremos en el laboratorio.

[10] Y en las múltiples pruebas y errores de este trabajo.

con parámetros y condiciones indicadas en el pie de la misma. Viendo que este primer modelo como mínimo nos devuelve respuestas, estén bien o mal, nos sirve para comenzar a optimizar.

3.2.1. Tamaño del dataset.

Para justificar la elección acerca del tamaño del dataset vamos a crear una programa que entrene redes de las mismas características mencionadas en el apartado anterior para datasets de 1, 10, 100, 1.000, 10.000 y 100.000 lorentzianas. De esta forma podemos ver en la figura 7 las pérdidas[11] al final del entrenamiento en función del tamaño del dataset. Lo que se observa es que para valores inferiores a 1.000 lorentzianas en el dataset obtenemos resultados similares y con pérdidas más altas. Mientras que por encima de 10.000 lorentzianas en el dataset vemos que las pérdidas comienzan a estabilizarse y disminuyen considerablemente. De hecho, el error no varía demasiado entre los tamaños 10.000 y 100.000, cuando recorrer un dataset de 10.000 es bastante más rápido que recorrer uno de 100.000. Teniendo en cuenta que estamos utilizando la función *error absoluto medio* y que las pérdidas representarán la suma del valor absoluto de las diferencias entre valores reales y predichos, alcanzar pérdidas menores que 1, como se ve para datasets mayores, nos indica que nos acercamos considerablemente más a las predicciones correctas. Por esto mismo y mirando por un compromiso entre calidad de resultados y tiempo de computación establecemos un valor intermedio de 60.000 lorentzianas. Una vez determinado el tamaño del dataset a partir de ahora usaremos este tamaño para las siguientes optimizaciones.

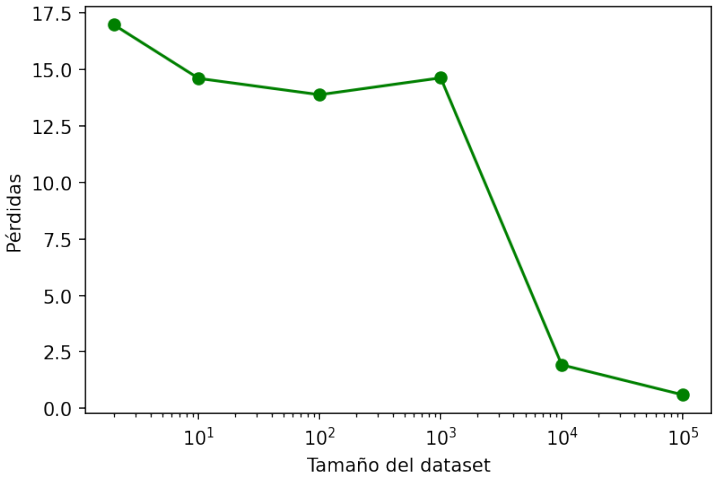

Figura 7: Pérdidas al cabo de 20 épocas de entrenamiento en función del tamaño del dataset con el que se entrena una red de $N_{\text{in}} = 172$ y $N_{\text{hi}} = 115$ neuronas.

3.2.2. Optimización de hiperparámetros.

Los hiperparámetros son aquellos parámetros no entrenables que afectan al comportamiento de la red de una forma más global y menos predecible de lo que por ejemplo lo haría variar los pesos o sesgos. El hecho de variar pesos o sesgos es la base de la retropropagación y sabemos cómo afectan a la función de pérdidas cuando los modificamos. El proceso de optimización de hiperparámetros

[11] La expresión es bastante explicativa, pero por si acaso aclaramos que cuando hablamos de pérdidas nos referimos al resultado de la función de pérdidas, que recordamos, queremos que sea pequeño.

es conocido por su elevado coste computacional ya que es un proceso que tiene como objetivo encontrar el punto de operación óptimo del modelo en cuestión. También es conocido por su trabajo manual pues al no ser unos parámetros que se modifican en el proceso de aprendizaje y dado que se desconocen las sinergias presentes entre ellos de una forma general, requiere de cierto proceso de selección "a posteriori" realizado por una persona [8].

Los hiperparámetros que nos conciernen a nosotros son: el número de neuronas por capa, el número de capas ocultas, el tamaño del lote (*batch*), el número de épocas de entrenamiento (*epochs*) y la tasa de aprendizaje (*learning rate*). Decimos "nos conciernen a nosotros" porque con otro tipo de redes o si empleáramos otro tipo de optimizadores habría otros hiperparámetros a tener en cuenta. En nuestro modelo, de todos los hiperparámetros que nos afectan, nos centraremos en el número de neuronas por capa, tamaño de lote y épocas. Dejamos de lado el número de capas ocultas porque para problemas con simples funciones no lineales, como es nuestro caso, se recomienda el uso de una única capa oculta, ya que añadir más de ellas tan solo aumenta el coste computacional sin observar una mejora justificable en los resultados. Tampoco modificaremos la tasa de aprendizaje pues ADAM utiliza tasas de aprendizaje adaptativas que funcionan de una forma mucho más eficiente de por sí que nuestro planteamiento de ir modificándola y dejándola fija para cada simulación.

Comenzaremos con la selección del número de neuronas para las capas de entrada (N_{in}) y oculta (N_{hi}). La capa de salida debe tener 3 neuronas dada la naturaleza de nuestro modelo de aprendizaje así que esta capa no se modificará. Para llevar a cabo esta selección realizaremos varias simulaciones que recorran diferentes combinaciones de neuronas entre las capas mencionadas. Estas simulaciones deben realizarse para valores constantes del tamaño de lote y durante el mismo número de épocas de entrenamiento, por lo que estableceremos *tamaño de lote* = 300 y *épocas* = 20, simplemente por mantener los parámetros que han funcionado en apartados anteriores.

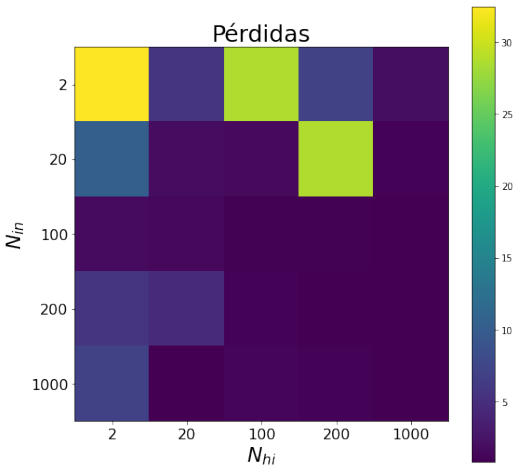

Figura 8: Pérdidas (Losses) calculadas en función de las combinaciones de neuronas N_{in} de la capa de entrada con N_{hi} de la capa oculta. Gráfica obtenida para batch=300 durante 20 épocas de entrenamiento.

El resultado de estas simulaciones se muestran en la figura 8. Lo que se observa es que para pocas neuronas por capa obtenemos pérdidas más altas que con mayor número de neuronas, pero si aumentamos demasiado el número de neuronas por capa tampoco se obtienen mejoras sustanciales en el funcionamiento de la red. Esto último se aprecia a partir de 100 neuronas por capa, donde se ve que los colores se homogeneizan bastante indicando esto mismo. Además, se puede ver que las pérdidas se ven reducidas hasta valores aceptables por primera vez en las primeras combinaciones de 20 y 100 neuronas. Si a esto le sumamos que al aumentar el número de neuronas los parámetros entrenables (pesos y sesgos) se multiplican considerablemente; Para una red con $N_{\text{in}} = N_{\text{hi}} = 20$ neuronas tendríamos un total de 3943 parámetros entrenables, mientras que para una red con $N_{\text{in}} = N_{\text{hi}} = 100$ neuronas tendríamos 27.703. Optando por un compromiso

entre calidad de resultados y tiempo de computación seleccionamos $N_{\text{in}} = N_{\text{hi}} = 50$ neuronas.

Con el número de neuronas establecido pasamos a la elección del tamaño de lote y el número de épocas de entrenamiento. Para esta elección realizaremos nuevas simulaciones que recorran diferentes valores del tamaño de lote (*Tamaño de lote* = [1, 10, 100, 1.000, 10.000]) para un gran número de épocas, 200 por ejemplo. Seguimos este planteamiento porque las librerías empleadas nos permiten almacenar y visualizar el historial de entrenamiento a lo largo de las épocas. Luego entrenando para un elevado número de épocas tenemos también los valores de las pérdidas para épocas anteriores. El resultado de esta simulación se muestra en la figura 9. Aquí podemos ver una tendencia que, en realidad, podríamos haber esperado. A mayor número de épocas el error se ve reducido porque damos tiempo a que la red entrene más y por lo tanto continúe modificando sus parámetros para alcanzar el mínimo. También observamos que para menor tamaño de lote el error es, por lo general, más pequeño. Esto tiene sentido ya que los lotes estaban definidos para acelerar el proceso de entrenamiento pues las modificaciones que haría a mis parámetros serían en base a un promedio por lote y me evitaría tener que hacer cálculos para cada caso del dataset. No obstante, si cojo lotes muy grandes para promediar no voy a obtener una estimación válida. En la figura vemos que para Batch = 10 y 100 obtengo valores prácticamente idénticos que para Batch = 1, si además nos vamos a partir de 100 épocas de entrenamiento tenemos que los valores alcanzados para Batch = 1000 son, una vez más, muy similares. Luego nos sale mucho más rentable entrenar para un lote mayor un número de épocas cercano a 100 pues el tiempo de computación es hasta 1000 veces menor[12]. Concluimos así que estableceremos de ahora en adelante Lote=1000.

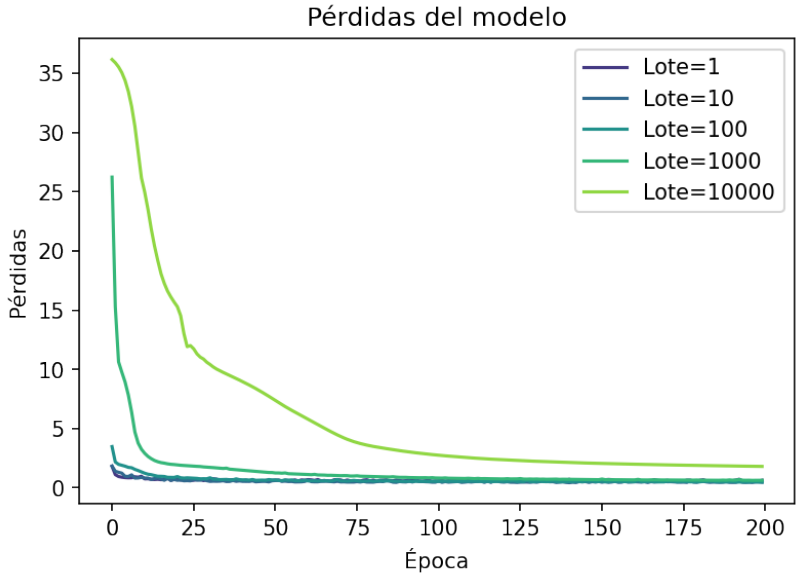

Figura 9: Pérdidas de modelos con $N_{\text{in}} = N_{\text{hi}} = 50$ neuronas para distintos tamaños de lote a lo largo de 200 épocas.

[12]La simulación realizada para obtener la curva de Batch=1 para 200 épocas duró 4h y 14min aproximadamente, mientras que la obtención de la curva para Batch=1000 duró menos de 1 minuto.

3.2.3. Problema de sobreentrenamiento.

Antes de presentar el modelo optimizado hemos de pararnos en una cuestión que hasta ahora no se ha mencionado y es de suma importancia en el ML, el *overfitting* (sobreentrenamiento[13]). El sobreentrenamiento es un problema presente en las técnicas de ML que aparece cuando el modelo se sobreespecializa en ajustar bien los datos de entrenamiento. Esto hace que pierda la capacidad de generalizar, convirtiéndose en un código inservible para su tarea original. Una de las formas de detectar el sobreentrenamiento es graficar las pérdidas de nuestro modelo para los datos de entrenamiento y los de validación, de esta forma si vemos que llegado cierto momento el error para los datos de entrenamiento sigue disminuyendo (o lo hace de forma drástica) mientras que para los datos de validación comienza a aumentar considerablemente tenemos que el modelo sobreentrena.

Entre los factores que afectan al sobreentrenamiento tenemos dos principales: El tamaño de dataset, un tamaño de dataset pequeño es más susceptible a ser memorizado que uno grande. Y la complejidad del modelo, un modelo sencillo consta de pocos parámetros por lo que se pueden modificar más rápido y podrían adaptarse con más facilidad a dar respuestas completamente correctas para los datos de training. Para determinar las condiciones de sobreentrenamiento de nuestro modelo vamos a entrenar uno de las mismas características ($N_{\text{in}} = N_{\text{hi}} = 50$ neuronas), con un dataset mucho más pequeño que el nuestro (1000 datos, por ejemplo), por lo tanto más "memorizable", más épocas que las que vamos a entrenar nosotros (1000, por ejemplo), para recorrer más veces el dataset, y un tamaño de lote menor para realizar promedios menos generales [8].

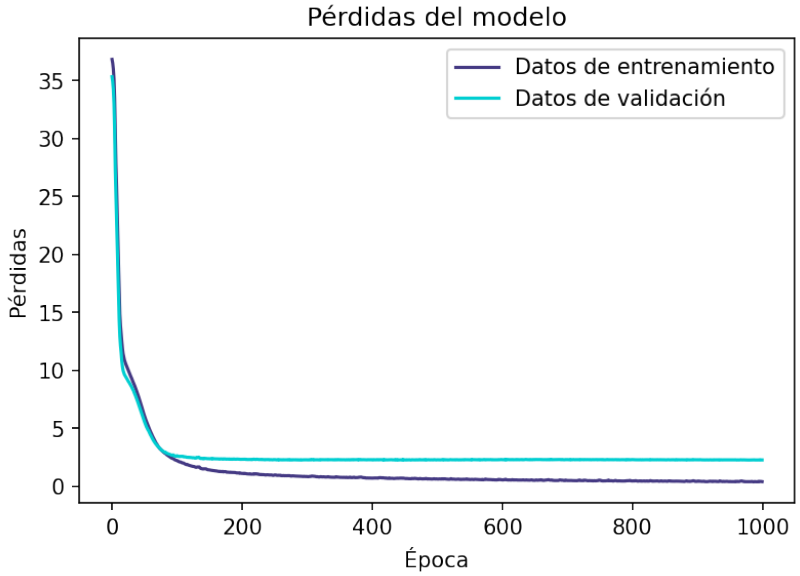

Figura 10: Pérdidas de un modelo con $N_{\text{in}} = N_{\text{hi}} = 50$ neuronas, lote=100 para los datos de entrenamiento y validación a lo largo de 1000 épocas.

En la figura 10 se muestra el historial de entrenamiento a lo largo de 1000 épocas. Podemos apreciar como las pérdidas de los datos de entrenamiento siguen una ligera tendencia decreciente

[13]Sería más literal emplear el término sobreajuste, pero dada la naturaleza de los modelos (los entrenamos) se ha considerado más conveniente utilizar el término sobreentrenamiento.

indicando que cada vez devuelve mejores predicciones para estos, mientras que las pérdidas de los datos de validación a partir de la época 100 se mantienen prácticamente constantes. Esto nos indica que es bastante improbable la memorización del dataset, lo cual no quiere decir que sea imposible. Pero dado que nunca vamos a entrenar nuestro modelo en unas condiciones tan favorables a sobreentrenar, podemos descartar (a efectos prácticos) la idea de observar sobreentrenamiento en nuestro modelo.

Uno podría poner en duda el razonamiento que planteamos pues el problema dispone, entre pesos y sesgos, de muchos grados de libertad, lo que puede llevarnos a pensar que sería fácil ver un sobreentrenamiento (o sobreajuste, en este caso). No obstante, hemos de recordar que esto no es un problema de regresión. En un problema de regresión intentaríamos dar la función que mejor se ajuste a los datos realizando cálculos punto por punto, como por ejemplo en un mínimos cuadrados tradicional, donde el número de grados de libertad proporcionados sí que es relevante. En nuestro caso queremos que la función "mire la foto" y diga si es "roja o azul", es decir, solo queremos que nos etiquete lo que ve con 3 números con los que nosotros construiremos la predicción. En definitiva, nuestro acercamiento por aprendizaje supervisado al problema hace que para que podamos ver sobreentrenamiento el modelo deba "memorizar" el dataset. Es por esto que el haber realizado la simulación anterior en unas condiciones tan lejanas a nuestro "punto de operación" y tan favorables para el sobreentrenamiento nos lleva a concluir que podemos entrenar tranquilamente nuestro modelo sin llegar a sobreentrenar.

3.3. Modelo optimizado.

Después de este proceso de optimización comprobamos el funcionamiento de la red ante datos generados de forma sintética que no han sido introducidos en el dataset original, el resultado de esta prueba se muestra en la figura 11. Podemos decir que estamos satisfechos y que este modelo está preparado para enfrentarse a datos reales. A modo resumen presentamos en la figura 12 el modelo final, el cual se ha entrenado durante 100 épocas con un dataset de 60.000 lorentzianas sintéticas y tamaño de lote = 1000.

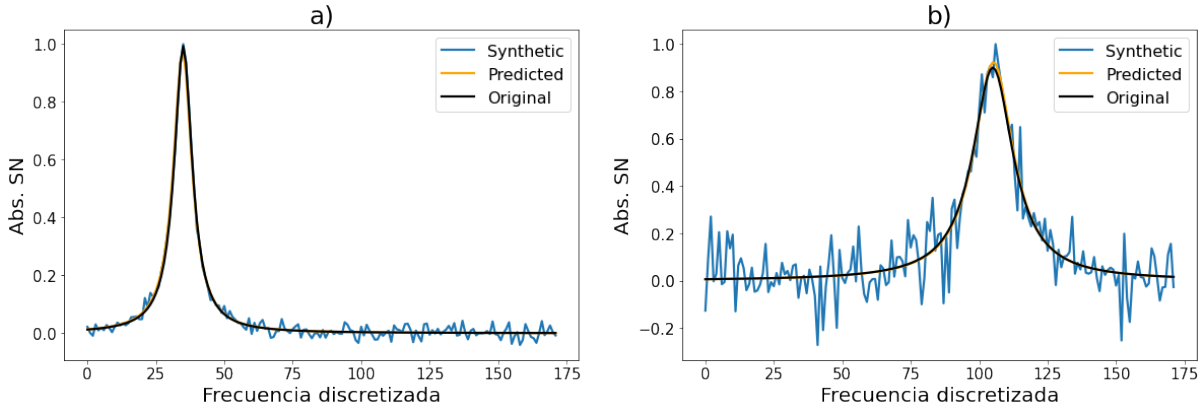

Figura 11: Ejemplos del funcionamiento de nuestro modelo para datos sintéticos desconocidos tras la optimización de los hiperparámetros. Podemos observar una clara mejora en cuanto al funcionamiento con respecto al mostrado en 5.

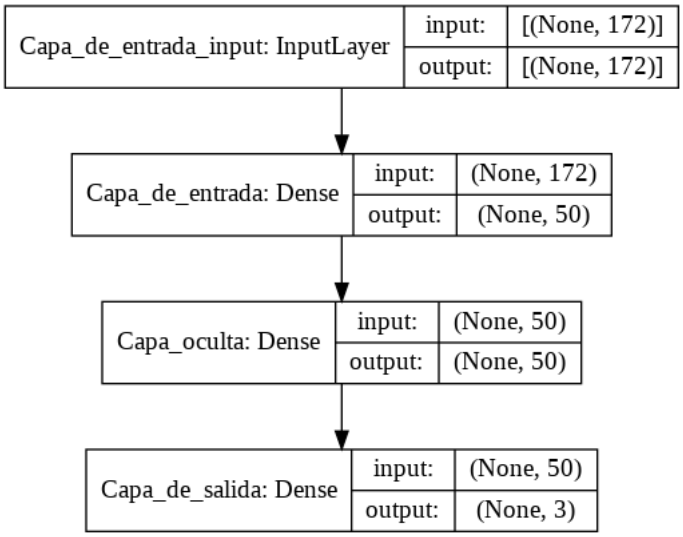

Figura 12: Arquitectura secuencial del modelo tras la optimización de sus hiperparámetros.

4. Prueba del modelo.

Es hora de probar nuestro modelo. En esta prueba suministraremos al modelo los datos del experimento con el DPPH, también normalizados[14], para que realice predicciones sobre las posiciones y anchura del pico de las lorentzianas y las compararemos con un ajuste tradicional por mínimos cuadrados[15]. Como bien mencionábamos en la introducción de este trabajo, el DPPH es un compuesto orgánico cuyas moléculas presentan radicales libres, actuando como un sistema efectivo de dos niveles. Este compuesto, además, por debajo de una cierta temperatura, la temperatura de Néel, presenta orden antiferromagnético con un eje de anisotropía. Cuando nos encontramos por encima de dicha temperatura, en la fase paramagnética, las moléculas interaccionan con el campo magnético generado por el circuito, pero no entre ellas, dando lugar a una única frecuencia de resonancia dada por el efecto Zeeman, $hf_{\text{res}} = g\mu_B B$. Esta frecuencia la podemos calcular ya que conocemos los valores de la expresión siendo $g = 2$, $\mu_B = 9{,}274 \cdot 10^{-24}\ J \cdot T^{-1}$, $h = 6{,}626 \cdot 10^{-34}\ Js$ y $B = 125\ mT$, por lo que $f_{res} = 3{,}499\ GHz$. Sin embargo, por debajo de la temperatura de orden, la interacción de canje genera una distribución de frecuencias de resonancia debido a que, al trabajar con una muestra en polvo, cada molécula presenta su eje de anisotropía en una orientación con respecto al campo magnético distribuida aleatoriamente. Esta distribución se refleja en un ensanchamiento en la medida de transmisión al contrario de la fase paramagnética, donde se observa una línea de absorción a una frecuencia bien definida.

En las imágenes a) y b), de la figura 13, mostramos algunos casos del funcionamiento del modelo frente a casos reales. En las imágenes c) y d) podemos ver la comparativa entre las predicciones de las posiciones de los picos y las anchuras para diferentes temperaturas. Aquí hemos de mencionar que se ha sometido al modelo a predicción de todos los datos suministrados pese a las advertencias de que a bajas temperaturas las medidas no representaban curvas del tipo Lorentz. Este hecho está relacionado con encontrarnos por debajo de la temperatura de Néel del material por lo que de momento centraremos nuestra discusión en aquellos datos cuya $T > 1\text{K}$.

[14] Los normalizamos porque la red está entrenada para estas condiciones y porque la información relevante se encuenctra en la posición y la anchura del pico.

[15] En este caso usaremos funciones ya implementadas en python que realizan ajustes de la forma tradicional.

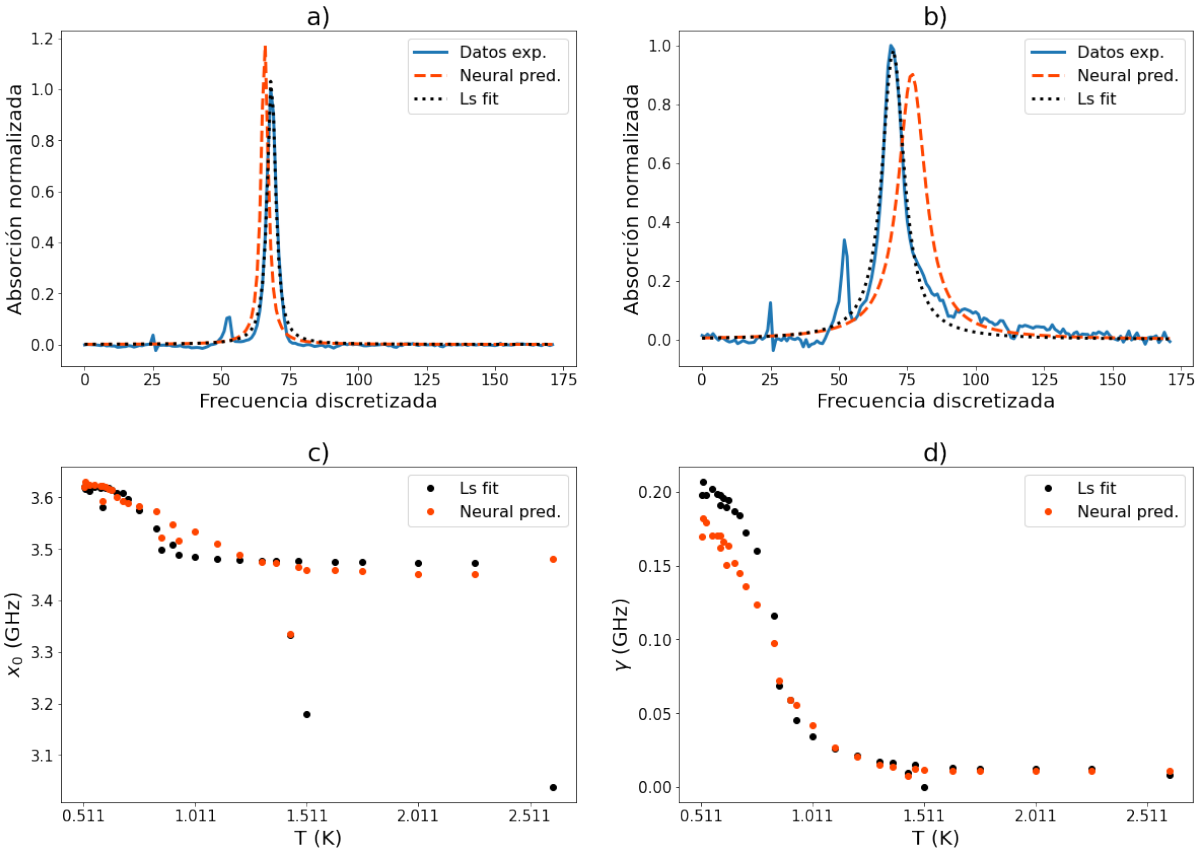

Figura 13: Funcionamiento del modelo optimizado ante datos experimentales reales. El color naranja corresponde a las predicciones de nuestro modelo, el negro se usa para las predicciones realizadas por el ajuste mediante mínimos cuadrados ("Ls fit") y las curvas azules son las curvas experimentales. En a) y b) se muestra en comparación la curva predicha por nuestro modelo y por el ajuste mediante mínimos cuadrados junto con la curva experimental. En c), las predicciones para las posiciones de pico de nuestro modelo en comparación con el ajuste de mínimos cuadrados y en d) lo mismo, pero con las anchuras de los picos.

4.1. Conclusiones.

En las imágenes a) y b) de la figura 13 se puede apreciar un ligero desplazamiento en la posición de los picos también apreciable en la comparación de la imagen c) de la misma figura, por lo que diremos que no acaba de realizar un ajuste correcto. No obstante, parece que la forma del pico es bastante correcta y en la imagen 13.d) vemos que realmente el modelo predice una anchura de pico muy similar a la realizada por ajuste tradicional. Con todo esto, e interpretando que el ajuste tradicional es más correcto, se puede concluir que el modelo es mejorable, por lo que a continuación planteamos dos posibles modificaciones y su funcionamiento.

5. Mejoras del modelo.

En esta última sección planteamos 2 modificaciones a nuestro modelo con la intención de mejorar su funcionamiento. La primera modificación se centra en las características internas del modelo y pretende aumentar la "potencia" de cálculo de la red aumentando el número de neuronas, con lo

que esperamos que pueda realizar un procesamiento de la información más complejo de forma que realice así mejores predicciones. En la segunda pretendemos modificar la información que recibe la red en su entrenamiento modificando el dataset. Si uno se fija atentamente en las gráficas de los datos experimentales (curvas azules dibujadas en las imágenes a) y b) de la figura 13) puede observar unos pequeños picos para los menores valores de frecuencia. Estos picos se deben a modos resonantes electromagnéticos que a veces se forman por imperfecciones en el circuito. Teniendo la ventaja de conocer esta característica de nuestras medidas, podemos modificar la forma de los datos de nuestro dataset introduciéndo pequeños picos de posición aleatoria y un tamaño de como mucho el 20 % del pico grande y ver si esto modifica en algo el funcionamiento de la red. Es decir, emplearemos la misma red que en el apartado anterior no ha dado resultados satisfactorios, pero con un dataset más sofisticado.

A continuación mostramos las predicciones realizadas por los modelos modificados según el orden en el que se han presentado: aumento en el número de neuronas y modificación del dataset. Con el aumento de neuronas (figura 14) podemos apreciar una mayor similitud (llegando incluso al solapamiento en algunos casos) con el ajuste de mínimos cuadrados tanto en la forma de las curvas como en las predicciones para posiciones y anchuras predichas a partir de 1K. Para temperaturas menores que 1K observamos también que las posiciones de pico parecen haber mejorado, entendiendo esta mejora como una mayor similitud al ajuste tradicional, mientras que para las anchuras parece que ahora se predice un valor ligeramente menor que en 13.

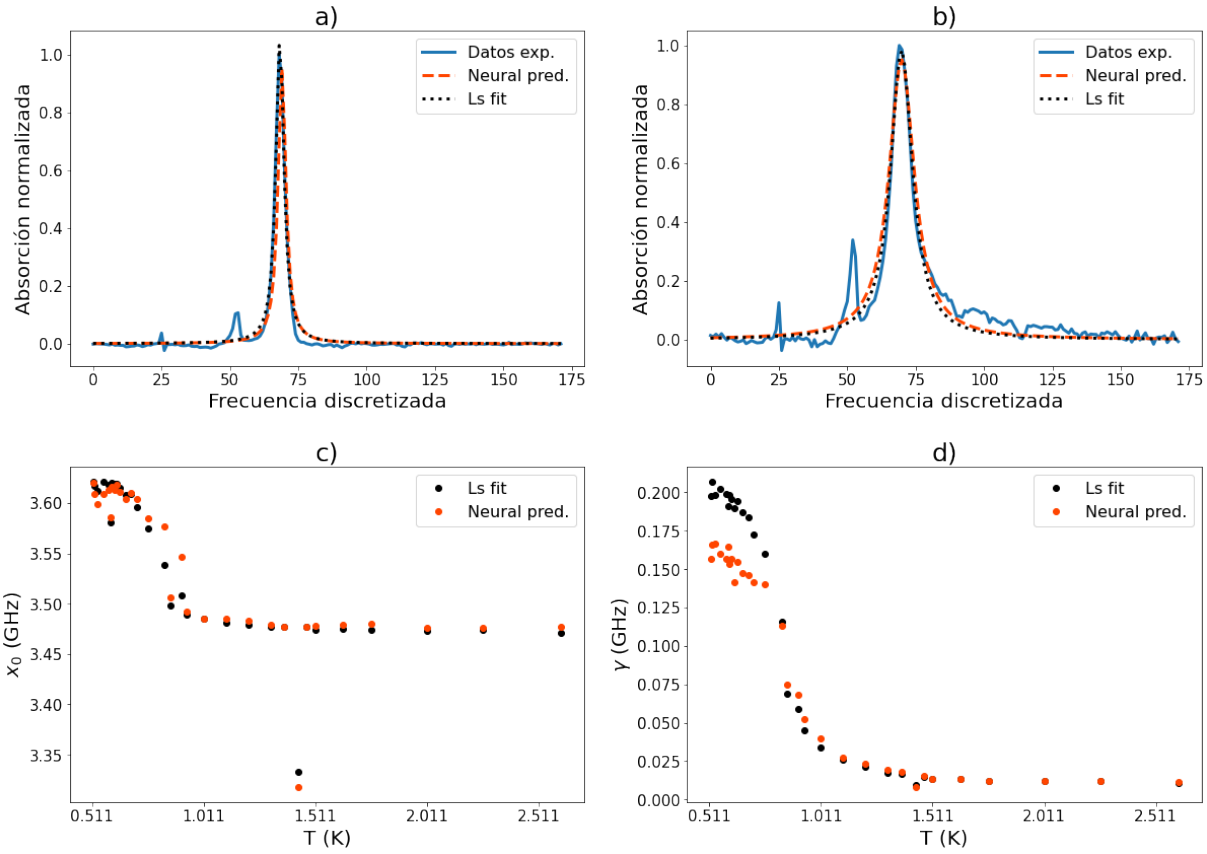

Figura 14: Funcionamiento del modelo con $N_{\text{in}} = N_{\text{hi}} = 100$ neuronas, entrenado con mismo dataset y parámetros que en 13. Las imágenes y la leyenda representan lo mismo, pero para este modelo.

En 15 mostramos las predicciones del modelo entrenado con un dataset modificado. Con esta última modificación también vemos incrementado el acuerdo entre los ajustes realizados por nuestro modelo y tradicional. En cuanto a las curvas predichas, vemos en a) y en b) que no se acaban de ajustar al 100 % a la forma del pico. No obstante, en la imagen c) de la figura 15 se puede observar que sí que obtenemos un cambio destacable en las predicciones de posición de los picos. Vemos también que, sin ser demasiado drástico, parece que este nuevo modelo predice anchuras ligeramente menores para temperaturas mayores a 1K. Se nos ocurre que este hecho puede deberse a que como éste a sido el modelo entrenado con un dataset modificado, la red haya aprendido a ignorar los picos pequeños asociando parte de la información a estos, lo que haría que interprete que menos información se corresponde al pico grande. Concluimos que la modificación del dataset también afecta de una forma positiva en cuanto al acuerdo con el ajuste tradicional.

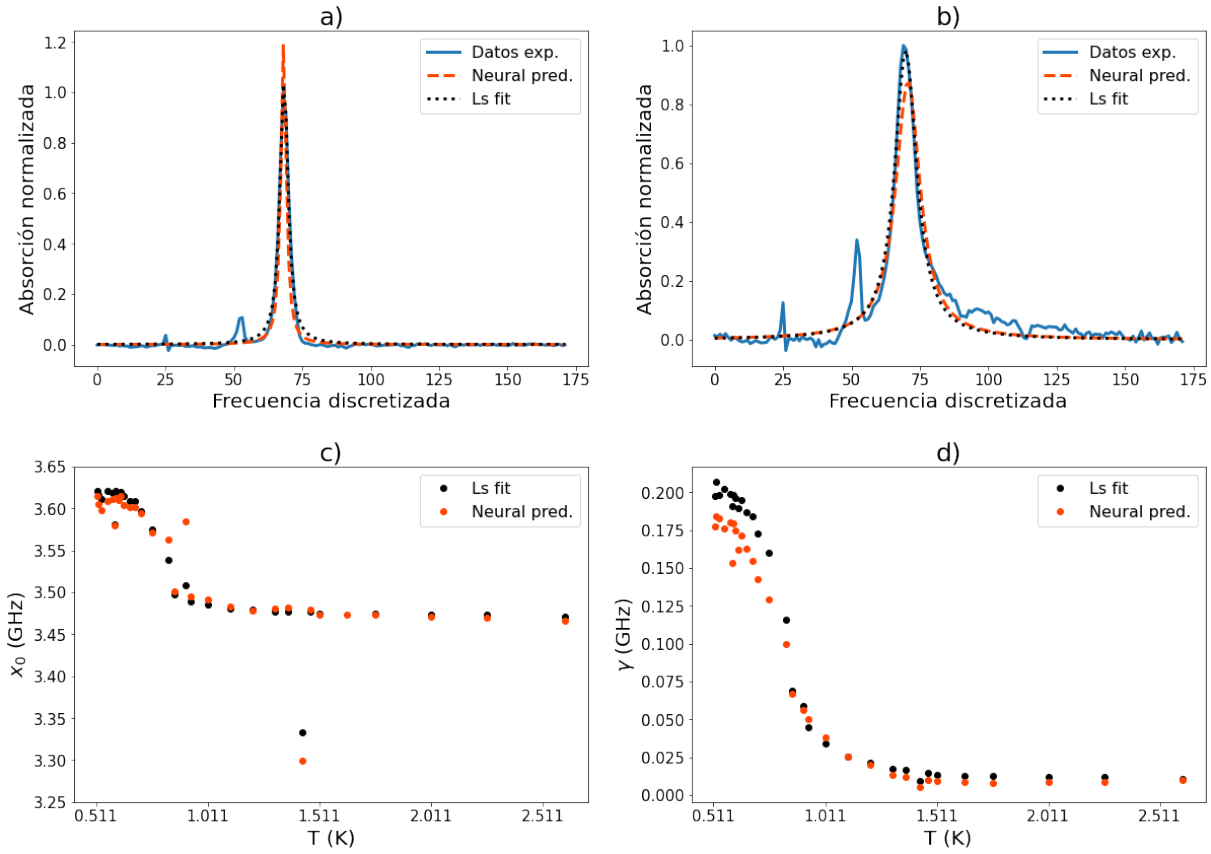

Figura 15: Funcionamiento del modelo con mismos parámetros que en 13, pero entrenado con el dataset modificado. Las imágenes y leyenda siguen, nuevamente, la misma pauta.

5.1. Conclusiones.

Para concluir el trabajo y determinar si las modificaciones realizadas al modelo consiguen una mejora en las predicciones, añadiremos la representación de los valores teóricos. Para el caso de las posiciones de los picos representaremos el valor de la frecuencia de resonancia, f_{res}, dada por el efecto Zeeman, mientras que los valores teóricos para las anchuras han sido suministrados por el grupo QMAD.

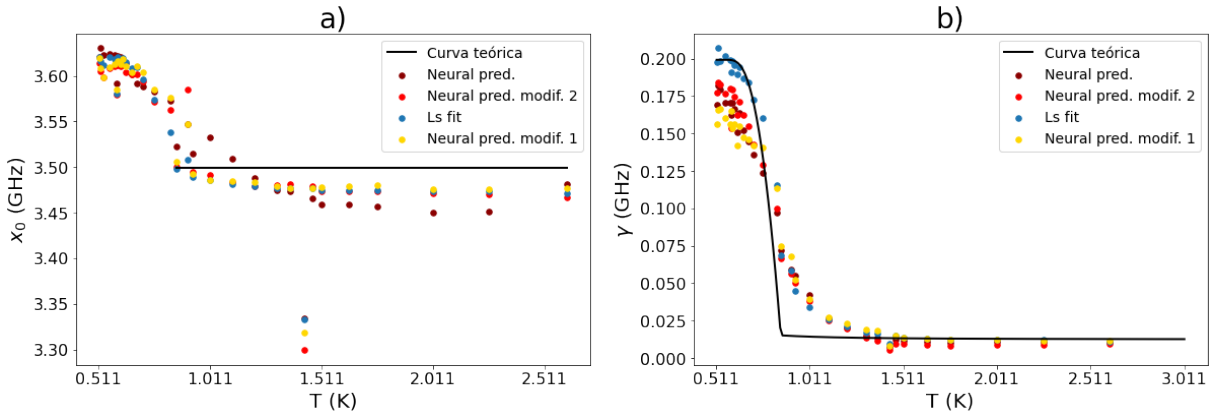

Figura 16: Predicciones de los modelos, ajuste tradicional y valores teóricos. Imagen a), comparación para las posiciones de pico. Imagen b), comparación para las anchuras. *Curva teórica* (color negro) indica los valores teóricos, *Neural pred.* (color rojo oscuro) indica las predicciones realizadas por el modelo tras la optimización de los hiperparámetros sin aún haber realizado ninguna modificación, *Neural pred. modif 1* y *Neural pred. modif 2* indican las predicciones realizadas por los modelos entrenados con más neuronas y con un dataset modificado respectivamente, por último, *Ls fit* indica las predicciones realizadas por el ajuste tradicional.

En la figura 16 vemos que para temperaturas mayores el ajuste en las anchuras es, en general, mejor que el ajuste en las posiciones de los picos, que tanto para nuestros modelos como para el ajuste tradicional, queda por debajo del valor teórico representado. Esta representación conjunta muestra que la modificación que aumenta el número de neuronas (modif. 1) devuelve mejores predicciones para ambos parámetros. En el caso de la posición de pico es la que presenta mayor valor y por tanto la que se acerca más al valor teórico, mientras que para las anchuras se observa un acuerdo bastante general entre todas las predicciones (a mayores temperaturas) y en concreto de esta modificación. Si observamos las temperaturas menores, donde sabemos que aparece un orden antiferromagnético que genera una distribución de frecuencias, vemos que las anchuras quedan claramente subestimadas por todos los modelos en general. Creemos que esto puede deberse a que nosotros hemos entrenado a los modelos con curvas del tipo Lorentz y las medidas tomadas a bajas temperaturas no se ajustan, en principio, a este tipo de curva. Si recordamos además, no estamos realizando un ajuste al uso, lo que nosotros estamos realizando (debido a que conocemos el tipo de curva que esperamos encontrar) es una clasificación de cada "tira de datos" en 3 números que luego nosotros sabemos que son parámetros de una curva.

Con todo esto concluimos el trabajo afirmando que el entrenamiento de un modelo secuencial con datos generados artificialmente presenta resultados satisfactorios. Es cierto que los resultados predichos por el modelo presentan algunas limitaciones, como las observadas a baja temperatura, y es que no debemos olvidar que por mucho que esto sea una red neuronal no deja de ser un programa más sofisticado que lo común para realizar una tarea, en este caso devolver 3 números (que a su vez representan 3 parámetros concretos de un tipo de curva) a partir de unos datos de entrada. Pero resulta muy interesante haber comprobado que unos datos generados artificialmente, algo que puede ser decisivo en algunos experimentos de elevado coste económico o temporal, pueden aportar el conocimiento necesario a una red neuronal para que pueda enfrentarse a datos

reales con este grado de precisión. Además, destacamos el hecho de que una modificación del dataset que consiga simular mejor los datos reales, también consiga mejorar el funcionamiento de la red ante los casos reales. Esto puede resultar muy beneficioso para el caso en que se hayan optimizado los hiperparámetros lo máximo posible, pero aun así estemos moviéndonos en elevados tiempos de computación. Indicándonos que en esta situación igual el camino más óptimo para alcanzar un buen resultado sería diseñar mejores datos. Como futuro trabajo, podría resultar interesante continuar buscando la correcta configuración y las modificaciones adecuadas para mejorar el funcionamiento del modelo y así mejorar su funcionamiento con las temperaturas menores, si es que es posible.

6. Bibliografía.

[1] Dijana Žilić y col. «Single crystals of DPPH grown from diethyl ether and carbon disulfide solutions–crystal structures, IR, EPR and magnetization Studies». En: *Journal of Magnetic Resonance* 207.1 (2010), págs. 34-41.

[2] ND Yordanov. «Is our knowledge about the chemical and physical properties of DPPH enough to consider it as a primary standard for quantitative EPR spectrometry». En: *Applied Magnetic Resonance* 10.1-3 (1996), págs. 339-350.

[3] Ignacio Gimeno y col. «Enhanced molecular spin-photon coupling at superconducting nanoconstrictions». En: *ACS nano* 14.7 (2020), págs. 8707-8715.

[4] Francois Chollet y col. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[5] Martın Abadi y col. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[6] Carlos Santana Vega. *¿Qué es el Machine Learning?¿Y Deep Learning? Un mapa conceptual | DotCSV*. Youtube. 2017. URL: https://www.youtube.com/watch?v=KytW151dpqU&ab_channel=DotCSV.

[7] Michael A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[8] Pankaj Mehta y col. «A high-bias, low-variance introduction to Machine Learning for physicists». En: *Physics Reports* 810 (2019). A high-bias, low-variance introduction to Machine Learning for physicists, págs. 1-124. ISSN: 0370-1573.

[9] G. Cybenko. «Approximation by superpositions of a sigmoidal function». En: *Mathematics of Control, Signals and Systems* 2 (1989), págs. 303-314.

[10] Carlos Santana Vega. *¿Qué es el Aprendizaje Supervisado y No Supervisado? | DotCSV*. Youtube. 2017. URL: %5Curl%7Bhttps://www.youtube.com/watch?v=oT3arRRB2Cw&ab_channel=DotCSV%7D.

[11] Laboratory for Information y Decision Systems. *The real promise of synthetic data*. https://news.mit.edu/2020/real-promise-synthetic-data-1016. 2020.

[12] Fernando Sancho Caparrini. *Variational AutoEncoder*. 2020. URL: http://www.cs.us.es/~fsancho/?e=232.

[13] Ivan Perov y col. *DeepFaceLab: A simple, flexible and extensible face swapping framework*. 2020. arXiv: 2005.05535 [cs.CV].