Universidad Zaragoza
1542

Trabajo de Fin de Grado

Topología y quiralidad en nanofotónica

Grado en Física

Autor

Óscar Costa Reyes

Directores

Luis Martín Moreno

Fernando Lorén Mastral

FACULTAD DE CIENCIAS

2023

Índice

1. Introducción y objetivos

El concepto de neurona artificial surgió en 1943 cuando Warren McCulloch y Walter Pitts trataron de imitar el comportamiento de las neuronas en el cerebro mediante el uso de computadoras [1]. La neurona artificial consistía en unas entradas y una salida imitando a las dendritas y al axón, respectivamente. A finales de la década de 1950, varios investigadores basándose en este concepto desarrollaron el perceptrón, un dispositivo capaz de realizar tareas de clasificación [2]. Sin embargo, las limitaciones del perceptrón y la falta de poder computacional provocaron un largo "invierno" en el campo.

En la década de 1980 se publicó un artículo que proponía una forma más sencilla de entrenar redes neuronales [3]. Este nuevo método permitía calcular las derivadas de la función de coste de manera más eficiente, lo que condujo a un gran avance en el campo gracias al progreso tecnológico de los procesadores. Esto dio lugar surgimiento de redes neuronales profundas (*deep learning*) capaces de reconocer patrones cada vez más complejos y realizar tareas más avanzadas.

A día de hoy nos encontramos en pleno auge del *Machine Learning*. Disponemos de modelos generadores de texto como GPT-3 o su versión más famosa Chat-GPT, modelos generadores de imágenes como Dall-E 2 o Stable Diffusion 2, siendo esta última de código abierto, inteligencias artificiales generales como GATO o incluso modelos capaces de vencer al ser humano en uno de los juegos de mesa más complejos como es el Go. La inteligencia artificial y las técnicas de aprendizaje automático están revolucionando la ciencia, como demuestra el reciente proyecto de *Deep Mind*, *Alpha Tensor*, que ha desarrollado un algoritmo de multiplicación de matrices más eficiente [4].

En este trabajo vamos a implementar diferentes técnicas del *Machine Learning* en el campo del electromagnetismo, más en concreto, al estudio de resonancias electromagnéticas a través de estructuras nanométricas en una superficie metálica. Estas resonancias, conocidas como plasmones, fueron teorizadas en los años 50 por R. H. Ritchie y se producen cuando la luz incide sobre la superficie metálica, excitando a los electrones [5]. La excitación de los electrones puede tener un impacto significativo en la difracción de la luz, tal y como se observó en estudios realizados por T. Ebbesen en estructuras de metal perforadas con diámetros mucho más pequeños que la longitud de onda de la luz [6]. Estos cambios se deben al acoplamiento entre los fotones incidentes y los plasmones de la superficie.

Primero desarrollaremos las ecuaciones usadas en modelo teórico y las implementaremos en código. Luego, generaremos un conjunto de datos para entrenar una red neuronal y predecir las propiedades de la luz incidiendo sobre una superficie con una estructura periódica determinada. Una vez entrenada la red, analizaremos cómo afecta la variación de los hiperparámetros a su rendimiento. Además, utilizaremos las propiedades de las redes

neuronales para comprender las simetrías del sistema. Finalmente, estudiaremos la aplicación de una estructura de red neuronal llamada *autoencoder* para caracterizar conjuntos de datos no etiquetados.

2. Propagación y difracción de ondas en materiales topológicos

Nuestro sistema está compuesto por una lámina metálica en la que realizamos agujeros rectangulares. Estos agujeros están uniformemente distribuidos por la lámina y cada uno está rotado respecto a los agujeros adyacentes. De esta forma podemos describir la distribución de agujeros como una celda unidad de N agujeros que se repite por toda la lámina en el plano xy. En la Figura 1 observamos cómo es la geometría de la celda unidad. Al tener una distribución periódica podemos describir la luz difractada como una combinación de los modos de la red recíproca. Cada modo se acopla con una amplitud proporcional a R_n^p, donde n es el modo y p la base de polarizaciones escogidas. Como queremos aplicar técnicas de *Machine Learning* necesitamos un número considerable de datos por lo que nuestra mejor opción es simular nuestro sistema y obtener los datos de esta forma. Para ello necesitamos resolver las ecuaciones de Maxwell con las condiciones de contorno de nuestro sistema. Para simplificar los cálculos

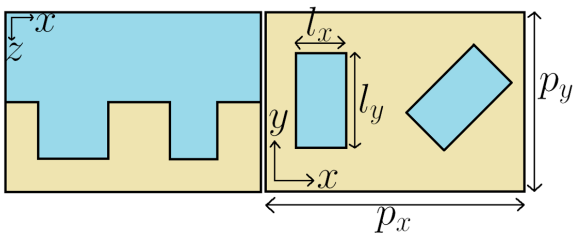


Figura 1: Corte y planta de una celda unidad con 2 agujeros.

vamos a trabajar con una onda incidente de amplitud 1, momento $\vec{k}^0 = \left(k_x^0, k_y^0\right)^T$, polarización σ^0 y longitud de onda λ. En el desarrollo teórico hemos trabajado con polarizaciones lineales, pero nos interesa analizar la respuesta en términos de polarizaciones circulares, lo que implica un simple cambio de base. Podemos escribir el campo electromagnético (E.M.) en el espacio real proyectándolo sobre $\langle \vec{r}_{||} |$:

$$\left\langle \vec{r}_{||} \middle| \vec{k}_{||}, p \right\rangle = \begin{pmatrix} k_x \\ k_y \end{pmatrix} \frac{e^{i\vec{k}_{||}\cdot\vec{r}_{||}}}{k_{||}\sqrt{p_x p_y}} \quad ; \quad \left\langle \vec{r}_{||} \middle| \vec{k}_{||}, s \right\rangle = \begin{pmatrix} -k_y \\ k_x \end{pmatrix} \frac{e^{i\vec{k}_{||}\cdot\vec{r}_{||}}}{k_{||}\sqrt{p_x p_y}} \; ; \tag{1}$$

donde $\vec{k}_{||} = (k_x, k_y)^T$, $\vec{r}_{||} = \left(x_{||}, y_{||}\right)^T$ es la posición en la celda unidad, $k_{||} = \sqrt{{k_x}^2 + {k_y}^2}$ y (p, s) son los elementos de nuestra base de polarizaciones σ. En el caso de $\vec{k}_{||} = \vec{0}$ escogemos:

$$\left\langle \vec{r}_{||} \middle| \vec{0}, p \right\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \frac{1}{\sqrt{p_x p_y}} \quad ; \quad \left\langle \vec{r}_{||} \middle| \vec{0}, s \right\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \frac{1}{\sqrt{p_x p_y}} \; . \tag{2}$$

Como estamos incidiendo con radiación electromagnética sobre una superficie con estructura periódica podemos escribir el momento en el medio I como $\vec{k}_{||}^{I} = \vec{k}^{0} + \vec{G}$, con $\vec{G}$ un vector de la red recíproca.

$$\vec{G} = 2\pi \begin{pmatrix} \frac{m_x}{p_x} \\ \frac{m_y}{p_y} \end{pmatrix} \quad ; \quad \begin{matrix} m_x = \{-M_x, -M_x+1, \cdots, M_x-1, M_x\} \\ m_y = \{-M_y, -M_y+1, \cdots, M_y-1, M_y\} \end{matrix} \text{ , donde } m_x, m_y \in \mathbb{Z} \tag{3}$$

Mientras que en el medio II lo podremos escribir como un modo de guía de ondas:

$$\vec{k}_{||}^{II} = \begin{pmatrix} q_x \\ q_y \end{pmatrix} = \pi \begin{pmatrix} n_x/lx \\ n_y/ly \end{pmatrix} \ . \tag{4}$$

Como en nuestro sistema $l_y > l_x$ solo vamos a tener en cuenta el modo $n_x = 0$ y $n_y = 1$ ya que este es el fundamental. Si proyectamos sobre el espacio real obtenemos:

$$\langle \vec{r}_{||} | n \rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \sqrt{\frac{2}{l_x l_y}} \sin\left(q_y \left(y + \frac{l_y}{2} \right) \right) \ , \tag{5}$$

donde $q_y = \pi / l_y$ es el momento del modo fundamental de guía de ondas. A partir de ahora vamos a usar la siguiente notación, $|0\rangle = \left| \vec{k}^0, \sigma^0 \right\rangle$, $|G\rangle = \left| \vec{k}^0 + \vec{G}, \sigma \right\rangle$, $|n\rangle = \left| n^{th} hole\ mode \right\rangle$ e $Y_n^{II} = k_z^{II}/g$ ya que el modo fundamental tiene polarización transversal eléctrica (TE). Para describir el campo magnético usamos una magnitud equivalente a H:

$$\left| -\hat{u}_z \wedge \vec{H}_i \right\rangle = \pm Y^i_{\vec{k}_{||},\sigma} \left| \vec{E}_i \right\rangle \ , \tag{6}$$

donde i es el medio en el que estamos, el signo $\pm$ depende del sentido de propagación del modo asociado a $\vec{k}_{||}$ e $Y^i_{\vec{k}_{||},\sigma}$ es la admitancia en la región i-ésima para un cierto momento $\vec{k}_{||}$ y una cierta polarización σ, en nuestro caso:

$$Y^I_{\vec{k}_{||},p} = \frac{\epsilon_I g}{k_z^I} \quad ; \quad Y^I_{\vec{k}_{||},s} = \frac{k_z^I}{g} \ , \tag{7}$$

con $g = 2\pi/\lambda$ y $k_z^I = \sqrt{\epsilon_I g^2 - k_{||}^2}$ donde ϵ_I es la constante dieléctrica del medio I. De esta forma podemos escribir los campos en las dos regiones como:

$$I \begin{cases} \left| \vec{E}_I(z) \right\rangle = |0\rangle\, e^{ik_z^I(k^0)z} + \sum_G R_G\, |G\rangle\, e^{-ik_z^I(G)z} \\ \left| -\hat{u}_z \wedge \vec{H}_I(z) \right\rangle = Y_0^I\, |0\rangle\, e^{ik_z^I(k^0)z} - \sum_G Y_G^I R_G\, |G\rangle\, e^{-ik_z^I(G)} \end{cases} \ , \tag{8}$$

$$II \begin{cases} \left| \vec{E}_{II}(z) \right\rangle = \sum_n A_n\, |n\rangle\, e^{ik_z^{II}(n)z} + \sum_n B_n\, |n\rangle\, e^{-ik_z^{II}(n)z} \\ \left| -\hat{u}_z \wedge \vec{H}_{II}(z) \right\rangle = \sum_n Y_n^{II} A_n\, |n\rangle\, e^{ik_z^{II}(n)z} - \sum_n Y_n^{II} B_n\, |n\rangle\, e^{-1k_z^{II}(n)z} \end{cases} \ , \tag{9}$$

donde A_n son las amplitudes de los modos que se propagan en la dirección positiva de z, B_n son las amplitudes de los modos que se propagan en la dirección negativa de z y $k_z^{II} = \sqrt{\epsilon_{II} g^2 - k_{\|}^2}$. Aplicando la continuidad del campo eléctrico en el plano z=0 y proyectando sobre $\langle G|$ obtenemos:

$$\delta_{G0} + R_G = \sum_n (A_n + B_n) S_{Gn} \ , \tag{10}$$

donde $S_{Gn} = \langle G|n\rangle$ son las integrales de solape y se pueden calcular en la aproximación de agujeros pequeños como:

$$S_{Gn} = S \cdot f_{\vec{G},\sigma,n} \cdot e^{-i(k_x x_c + k_y y_c)} \ , \tag{11}$$

donde x_c y y_c son las coordenadas de los centros del agujero n-ésimo, $S = \frac{4}{\pi}\sqrt{\frac{l_x l_y}{2p_x p_y}}$ es una constante que depende de la geometría de la celda unidad y $f_{\vec{G},\sigma,n}$ es:

$$f_{\vec{G},p,n} = \frac{k_x \cos\theta_n + k_y \sin\theta_n}{k_{||}} \quad ; \quad f_{\vec{G},s,n} = \frac{-k_y \cos\theta_n + k_x \sin\theta_n}{k_{||}} \ . \tag{12}$$

Por otro lado, podemos aplicar la continuidad del campo magnético en el plano $z = 0$ sobre las aperturas de los agujeros, ya que no hay metal, y proyectar sobre $\langle n|$, de esta forma obtenemos:

$$Y_0^I S_{0n}^* - \sum_G Y_G^I R_G S_{Gn}^* = Y_n^{II}(A_n - B_n) \ . \tag{13}$$

Además, vamos a suponer que tenemos un conductor eléctrico perfecto (PEC) de tal manera que en el fondo de los agujeros el campo eléctrico ($\vec{E}$) se tiene que anular. Esto implica que:

$$A_n e^{ik_z^{II}(n)h} + B_n e^{-ik_n^{II}(n)h} = 0 \Rightarrow B_n = \underbrace{-e^{2ik_z^{II}(n)h}}_{\phi_n} A_n = \phi_n A_n \ . \tag{14}$$

Con esto podemos sustituirlos en la ecuación 13 y obtener los valores de A_n y B_n

$$A_n = \frac{Y_0^I S_{0n}^* - \sum_G Y_G^I R_G S_{Gn}^*}{Y_n^{II}(1-\phi_n)} \quad ; \quad B_n = \frac{\phi_n}{Y_n^{II}(1-\phi_n)} \left(Y_0^I S_{0n}^* - \sum_G Y_G^I R_G S_{Gn}^* \right) \ . \tag{15}$$

Con esto y definiendo:

$$Z_{GG'} = \frac{1+\phi}{1-\phi} \frac{1}{Y^{II}} \sum_n S_{Gn} S_{G'n}^* \ , \tag{16}$$

podemos escribir la ecuación 10 como:

$$\boxed{R_G = -\delta_{G0} + Y_0^I \cdot Z_{G0} - \sum_{G'} Z_{GG'} Y_{G'}^I R_{G'} \ .} \tag{17}$$

Por lo que hemos llegado a un sistema de ecuaciones cuya solución nos permite calcular los coeficientes de reflexión. El único problema es que, en principio, tenemos infinitos modos con los que se puede acoplar la luz incidente. Para solucionar esto vamos a aprovechar un resultado obtenido en [7] y es que para reproducir unos resultados semejantes solo es necesario resolver estas ecuaciones para $G = (0, \pm 1, \pm 2, \cdots, \pm(N + 2n_w))$ donde n_w es el número de rotaciones de 2π que dan los agujeros a lo largo de la celda unidad. Aunque hemos hecho el desarrollo con la aproximación de conductor perfecto también se pueden resolver las ecuaciones para un metal real cualquiera (aproximación SIBC). En todo este trabajo hemos supuesto que la lámina era de oro y hemos calculado las integrales de solape tal y como se muestra en [7]. Con todo esto podemos escribir un código que resuelva estas ecuaciones en función de las características de los agujeros y de la luz incidente. En la Figura 2 vemos un caso especial,

en el que debido a la excitación de un plasmón logramos cambiar la polaridad de circular a izquierdas (LCP+) a circular a derechas (RCP-) para el modo Bragg $G = +1$ entorno a $\lambda = 540$ nm incidiendo perpendicularmente a la superficie.

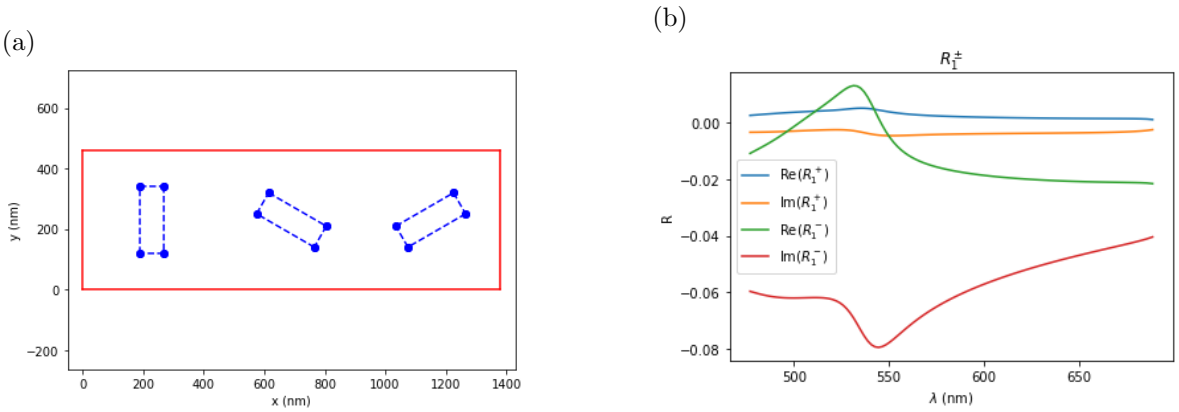


Figura 2: (a)Distribución de $N = 3$ agujeros en la celda unidad. (b)Espectro resultante para el modo Bragg G=+1 cuando la estructura de (a) se ilumina en incidencia normal ($\vec{k}^0 = \vec{0}$) y polarización circular $\sigma^0 = +$ en función de la longitud de onda incidente λ.

3. Acercamiento a las redes neuronales

El aprendizaje automático (*machine learning*) es una rama de la inteligencia artificial que tiene como objetivo desarrollar sistemas que tengan la capacidad de aprender en una o varias tareas sin que se les programe específicamente para ello. Este aprendizaje puede dividirse en dos grupos, el aprendizaje supervisado y el no supervisado. En el supervisado se usan datos previamente etiquetados, en el que se busca que la red aprenda esas etiquetas. En el no supervisado se usan datos sin etiquetar por lo que la red es la que se encarga de encontrar los patrones que caracterizan el conjunto de datos. Además, dentro del *machine learning* existen varias técnicas como los árboles de decisión, modelos de regresión, algoritmos genéticos y muchos más, pero aquí nos vamos a centrar en las redes neuronales.

3.1. La neurona

La neurona es el elemento más básico de una red neuronal. Esta la podemos entender como una función que tiene como entrada un conjunto de números reales y devuelve una suma ponderada de estos más un término independiente que se llama sesgo (o *bias*):

$$f(x_1, \cdots, x_n) = b + \sum_{i=1}^{n} \omega_i x_i \ , \tag{18}$$

donde b es el término de *bias* y ω_i es el peso asociado a la entrada i. Con este modelo tan sencillo podemos encontrar una combinación de parámetros que imite el comportamiento de una puerta lógica AND u OR, pero al igual que en los circuitos lógicos encontramos ciertas

limitaciones al usar solo una. Para resolver este problema podemos conectar las salidas de unas neuronas con las entradas de otras para obtener comportamientos más complejos.

3.2. Redes

A la hora de añadir más neuronas podemos realizarlo de dos formas, la primera forma de hacerlo es que tengan la misma entrada, cuando esto pasa diremos que están en la misma capa. La segunda forma de hacerlo es que la salida de la primera neurona sea la entrada de la siguiente, cuando esto pasa diremos que están en distintas capas. De esta forma podemos obtener un esquema como el de la Figura 3.

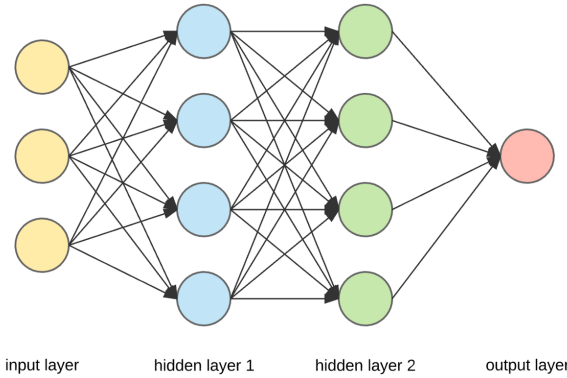


Figura 3: Red Neuronal con tres entradas dos capas ocultas y una salida.

Al conectar las neuronas de esta manera surge un problema y es que como las neuronas solo realizan una suma ponderada se puede ver que el sistema al completo se comporta como si solo tuviese una única neurona con diferentes parámetros. Una forma de solucionar este problema es añadir una no linealidad a la salida de cada neurona. Para ello vamos a transformar la salida de cada neurona aplicándole una función no lineal que se denomina función de activación. Existen muchas funciones de activación pero las más comunes son:

$$\begin{aligned} A_1(x) &= \begin{cases} 1 \text{ si } x \geq 0 \\ 0 \text{ si } x < 0 \end{cases} && \text{Escalón ,} \\ A_2(x) &= \frac{1}{1+e^{-x}} && \text{Sigmoide ,} \\ A_3(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} && \text{Tangente Hiperbólica ,} \\ A_4(x) &= \text{máx}(0, x) && \text{ReLu.} \end{aligned} \tag{19}$$

Cada una de estas funciones tiene sus ventajas e inconvenientes. Por ejemplo, de la función escalón podemos destacar su simplicidad, pero no es continua. Por otro lado la sigmoide y la tangente hiperbólica sí que son continuas y están acotadas, pero la función exponencial es más costosa en términos computacionales. Por último, tenemos la función ReLu que es continua y sencilla de calcular pero no está acotada y su derivada no es continua.

3.3. Descenso del gradiente y back-propagation

Si queremos usar las redes neuronales para resolver un cierto problema debemos ser capaces de encontrar la combinación de parámetros que mejor se ajusta a la tarea que buscamos resolver. En el caso del aprendizaje supervisado necesitamos una serie de datos, que serían la entrada de nuestra red, y el resultado que queremos que prediga la red. Estos datos pueden ser desde imágenes de dígitos escritos a mano, cadenas de palabras, imágenes de objetos o incluso funciones de potencial para ver como afectan a una función de onda. Lo primero que tenemos que parametrizar es si la red que tenemos predice correctamente los datos que le proporcionamos. Para ello definimos una función de coste que medirá cuanto difiere la predicción de la red de lo que queríamos que nos devolviese. Una función de coste puede ser el error cuadrático medio.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{20}$$

donde n es el número de datos de entrenamiento, y_i es el valor real del i-ésimo dato de entrenamiento e $\hat{y}_i$ es el valor predicho por la red neuronal para el mismo dato. Existen más funciones de coste pero esta tiene varias ventajas, es convexa, diferenciable y bastante intuitiva. Una vez tenemos definida nuestra función de coste podríamos usar el algoritmo del descenso del gradiente para movernos en nuestro espacio de parámetros y acabar en un mínimo local de la función de coste. Es decir, calcular las derivadas parciales de la función de coste respecto a cada parámetro que se puede modificar, y después movernos en la dirección opuesta. Esto lo podemos ver en la siguiente ecuación:

$$\theta_j = \theta_j - \eta\frac{\partial}{\partial\theta_j}C(\theta), \tag{21}$$

donde θ_j es el valor del parámetro j-ésimo, $C(\theta)$ es la función de coste, y η es la tasa de aprendizaje (*learning rate*), que controla la velocidad de descenso del gradiente. Cuando nos encontramos en la última capa oculta[2] calcular la derivada parcial respecto a los parámetros es una tarea sencilla. Pero cuanto más nos acercamos a la capa de entrada surge un problema: ¿cómo calculamos cuánto varía la función de coste respecto a un parámetro? Y es que cuando cambiamos un parámetro, este se propaga hacia todas las neuronas de la siguiente capa, y así sucesivamente hasta llegar a la capa final. Es decir, tendríamos que tener en cuenta todos los posibles caminos por los que el cambio de un parámetro se propaga. Esto dificulta mucho el entrenamiento de redes neuronales grandes ya que no escala de forma lineal con el tamaño de la red. Este problema quedó sin ser resuelto hasta 1986, cuando David E. Rumelhart, Geoffrey E. Hinton y Rondld J. Williams propusieron un nuevo algoritmo llamado *back-propagation* mediante el cual se podía calcular las derivadas parciales de la función de coste de forma

[2]Una capa oculta es el conjunto de neuronas que tienen la misma entrada y se encuentra entre la entrada y la salida.

mucho más sencilla [3]. Este algoritmo se basa en calcular el error imputado de las neuronas de la capa l y propagarlo a la capa anterior. Este es un algoritmo iterativo, es decir, se empieza calculando los errores y derivadas de la última capa para calcular los errores y derivadas de la capa anterior. Este algoritmo se puede resumir en las siguientes cuatro ecuaciones.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}\sigma'(z_j^L) \ , \tag{22}$$

$$\delta_j^l = \left(\sum_k w_{jk}^{l+1}\delta_k^{l+1}\right)\sigma'(z_j^l) \ , \tag{23}$$

$$\frac{\partial C}{\partial w_{ij}^l} = a_j^{l-1}\delta_i^l \ , \tag{24}$$

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \ . \tag{25}$$

Estas ecuaciones pueden parecer complicadas, pero vamos a ir explicando de dónde viene cada una. Primero vamos a suponer que tenemos una red neuronal con L capas, así que vamos a empezar a calcular las derivadas respecto a los parámetros de la capa L. En cada neurona tenemos dos tipos de parámetros, los pesos ω y los sesgos b, por lo que vamos a tener que calcular dos tipos de derivadas en las que se puede aplicar fácilmente la regla de la cadena:

$$\frac{\partial C}{\partial \omega^L} = \frac{\partial C}{\partial a^L}\cdot\frac{\partial a^L}{\partial z^L}\cdot\frac{\partial z^L}{\partial \omega^L} \quad , \quad \frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L}\cdot\frac{\partial a^L}{\partial z^L}\cdot\frac{\partial z^L}{\partial b^L} \ .$$

Ya que z^L es el resultado de la suma ponderada y el coste es $C(a^L(z^L))$, con a^L la función de activación de la capa L. Por simplicidad voy a suprimir el uso de subíndices ya que solo indican la posición en la misma capa y hacen la notación engorrosa. Aunque el número de derivadas que tenemos que hacer haya aumentado estas son muy fáciles de calcular y van a depender de la función de coste y de activación que escojamos. En el caso de que escojamos MSE como función de coste y la sigmoide como función de activación las derivadas son

$$\frac{\partial C}{\partial a_j^L} = \left(a_j^L - y_j\right) \ , \qquad \frac{\partial a^L}{\partial z^L} = a^L(z^L)\cdot(1 - a^L(z^L)) \ ,$$

$$\frac{\partial z^L}{\partial \omega^L} = a^{L-1} \ , \qquad \frac{\partial z^L}{\partial b^L} = 1.$$

Ahora podemos definir un nuevo parámetro δ^L que es el error imputado a la neurona y cuantifica como de responsable es la neurona en el error. De esta forma podemos obtener ya tres de las cuatro ecuaciones que veíamos anteriormente

$$\delta^L = \frac{\partial C}{\partial a^L}\cdot\frac{\partial a^L}{\partial z^L} = \frac{\partial C}{\partial z^L} \ ,$$

$$\frac{\partial C}{\partial \omega^L} = \delta^L \cdot a^{L-1} \ ,$$

$$\frac{\partial C}{\partial b^L} = \delta^L \cdot 1 \ ,$$

donde a^{L-1} es la activación de la capa previa. Una vez tenemos las derivadas de la capa L podemos volver a aplicar la regla de la cadena para obtener las derivadas a la siguiente

composición $C(a^L(\omega^L a^{L-1}(\omega^{L-1}a^{L-2}+b^{L-1})+b^L))$.

$$\frac{\partial C}{\partial \omega^{L-1}} = \underbrace{\underbrace{\frac{\partial C}{\partial a^L}}_{\delta^L}\cdot\underbrace{\frac{\partial a^L}{\partial z^L}}_{\omega^L}\cdot\underbrace{\frac{\partial z^L}{\partial a^{L-1}}}_{*}}_{\delta^{L-1}}\cdot\underbrace{\frac{\partial a^{L-1}}{\partial z^{L-1}}}_{a^{L-1}}\cdot\frac{\partial z^{L-1}}{\partial \omega^{L-1}} \ ,$$

$$\frac{\partial C}{\partial b^{L-1}} = \underbrace{\underbrace{\frac{\partial C}{\partial a^L}}_{\delta^L}\cdot\underbrace{\frac{\partial a^L}{\partial z^L}}_{\omega^L}\cdot\underbrace{\frac{\partial z^L}{\partial a^{L-1}}}_{*}}_{\delta^{L-1}}\cdot\underbrace{\frac{\partial a^{L-1}}{\partial z^{L-1}}}_{1}\cdot\frac{\partial z^{L-1}}{\partial b^{L-1}} \ .$$

De esta ecuación ya hemos calculado δ^l, ω^L, a^{L_1} y otras como $\frac{\partial a^{L-1}}{\partial z^{L-1}}$ se puede ver que es la derivada de la función de activación σ'. Por otro lado, ya hemos definido el error imputado de la neurona en la capa $L-1$. Resumiendo, para obtener las derivadas parciales del coste seguimos los siguientes pasos.

1. Computamos el error de la última capa:

$$\delta^L = \frac{\partial C}{\partial a^L}\cdot\frac{\partial a^L}{\partial z^L} \ .$$

2. Retropropagamos el error a la capa anterior:

$$\delta^{l-1} = \omega^L\delta^l\cdot\frac{\partial a^{l-1}}{\partial z^{l-1}} \ .$$

3. Calculamos las derivadas de la capa usando el error:

$$\frac{\partial C}{\partial \omega^{l-1}} = \delta^{l-1}a^{l-2} \quad , \quad \frac{\partial C}{\partial b^{l-1}} = \delta^{l-1} \ .$$

Combinando el algoritmo de *back-propagation* y el del descenso del gradiente podemos hacer que una red neuronal aprenda ya que el de *back-propagation* nos proporciona el gradiente con el que nos moveremos en el espacio de parámetros. Este es un proceso iterativo así que debemos escoger con cuidado nuestro *learning rate* ya que si es muy grande no llegaremos a ningún mínimo y nos moveremos en el espacio de parámetros de forma caótica y si es muy pequeño debemos hacer muchos más pasos.

4. Análisis de hiperparámetros

Una vez hemos entendido qué es una red neuronal y cuál el el proceso de aprendizaje vamos a entrenar nuestra primera red. Para ello necesitamos generar nuestro *data set* de entrenamiento. Vamos a escoger una celda unidad con tres agujeros e incidiremos con luz perpendicular a la lámina y polarización circular a izquierdas (LCP+). Nuestro objetivo va a ser que la red prediga $R_1^{\pm}$, es decir, el coeficiente de reflexión en la base de polarizaciones circulares del modo $G=+1$ cuando le proporcionamos los tres ángulos θ_i para un determinado

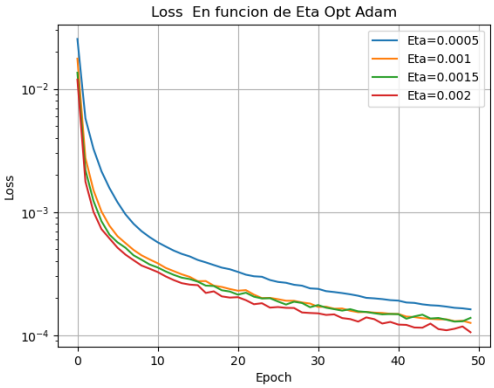
rango de longitudes de onda incidentes $[477, 689]$ nm. Antes de entrenar nuestra red debemos normalizar los datos de entrada ya que, si no lo estuviesen aquellos que fuesen mayores que uno serían los que más peso tendrían en el entrenamiento porque incrementarían mucho la función de coste, mientras que los más pequeños serían ignorados por esta. Por otro lado, para evitar posibles correlaciones realizamos una aleatorización en el orden de nuestros datos. Después los separaremos en dos grupos, el primero contendrá los datos que usaremos para entrenar a la red (datos de entrenamiento), mientras que el segundo lo usaremos para comprobar si nuestra red sufre de *overfitting*[3](datos de validación). El tamaño de estos grupos será aproximadamente de un 80 % del total para el de entrenamiento y de un 20 % para el de validación. Para generar y entrenar las redes neuronales vamos a usar las librerías *TensorFlow-2.10* y *Keras* de *Python* ya que simplifican mucho estas tareas. Una vez tenemos los datos listos debemos escoger adecuadamente los hiperparámetros, estos son una serie de características que se establecen antes del entrenamiento del modelo y van a influir en el resultado de este. Por ejemplo, en una red neuronal, los hiperparámetros incluyen el *learning rate*, el número de neuronas, la función de activación o el *mini batch size*. El *learning rate* determina la velocidad a la que se van a actualizar los parámetros en cada iteración del entrenamiento, además en este caso vamos a usar ADAM, que es un algoritmo que calcula las primeras y segundas derivadas del gradiente para optimizar el valor de η en cada iteración. El *mini batch size* es el número de muestras de entrenamiento utilizadas en cada iteración del algoritmo de entrenamiento. Un tamaño de *mini batch* más grande puede reducir la variabilidad del gradiente y acelerar el entrenamiento, pero también puede aumentar la complejidad computacional y requerir más memoria. Por otro lado tenemos el número de neuronas, o la forma de nuestra red. Si esta es muy pequeña, sufriremos de *underfitting* mientras que si esta es demasiado grande corremos el riesgo de sobreajustar los datos de entrenamiento. Otro problema que surge de tener una red con demasiadas capas ocultas es el del desvanecimiento del gradiente (*vanishing gradient*). Esto produce que las capas iniciales cambien muy poco sus parámetros porque el gradiente se ve reducido cuanto más nos alejamos de la capa final. Se ha observado que usando la función de activación ReLu se obtienen mejores resultados cuando tenemos muchas capas ocultas[8].

Para ver qué hiperparámetros funcionan mejor con los datos que tenemos vamos a escoger como nuestro punto de partida $\eta = 0.001$, MB size=32, ReLu como función de activación en las capas ocultas y una estructura con dos capas ocultas de 200 neuronas cada una. A partir de ahí vamos a modificar cada hiperparámetro por separado y ver qué efecto tiene en un entrenamiento con 50 épocas[4]. Los resultados obtenidos los podemos ver en la Figura 4. En la figura 4a vemos que en el rango estudiado de η no existe una gran diferencia en la

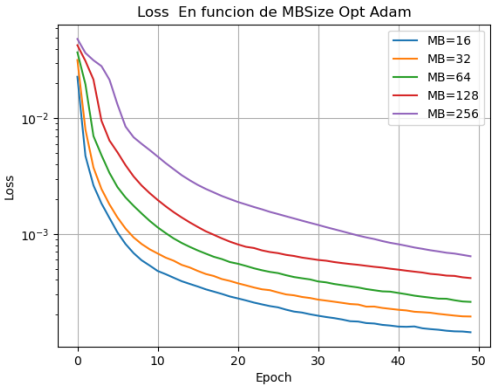
[3]El overfitting ocurre cuando nuestra red minimiza el coste de los datos de entrenamiento mientras que empeora los resultados sobre el grupo de validación.

[4]Una época es una iteración completa a través del conjunto de datos de entrenamiento.

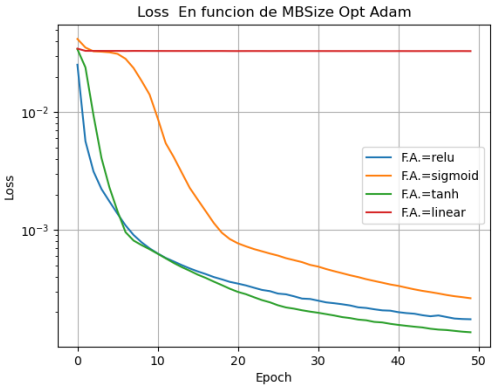
(a) η variable.



(b) Mini batch size variable.



(c) Función de activación variable.



(d) Número de neuronas en las capas ocultas variable.

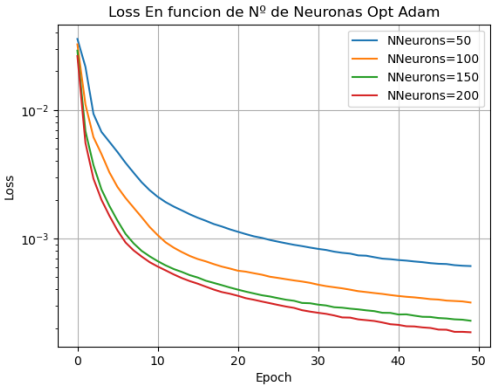


Figura 4: Análisis de hiperparámetros.

evolución de la Loss, esto se puede deber al algoritmo de optimización que hemos escogido. En la Figura 4b obtenemos que cuanto más pequeño es el tamaño del *mini batch* mejor es el ajuste, aunque debemos destacar que también necesitamos más tiempo para entrenar una época. Por otro lado en la Figura 4d tenemos que cuantas más neuronas mejor se ajusta pero también tardamos más en entrenar a nuestro modelo. En la Figura 4c observamos que al usar una función de activación lineal nuestra red no aprende.

Por último, en las Figuras 5 y 6 podemos observar cómo ajusta los datos nuestro modelo para dos casos que se encuentran en el conjunto de validación, es decir, casos que nuestra red no ha visto durante el entrenamiento. Para ello, representamos las parte real e imaginaria de las dos componentes circulares ($\pm$) del modo Bragg $G = +1$ en función de λ, siendo las líneas discontinuas los datos de las simulaciones y las líneas continuas lo predicho por la red.. En la Figura 5 vemos que, a pesar de incidir con luz LCP+, obtenemos coeficiente de reflexión más elevados para el modo RCP-, por lo que nuestra red ha sido capaz de reproducir la interacción de los fotones con los plasmones de la superficie. Por otro lado, en la Figura 6, al ser una distribución sin ningún orden aparente, obtenemos unos coeficiente similares para las dos componentes de polarización.

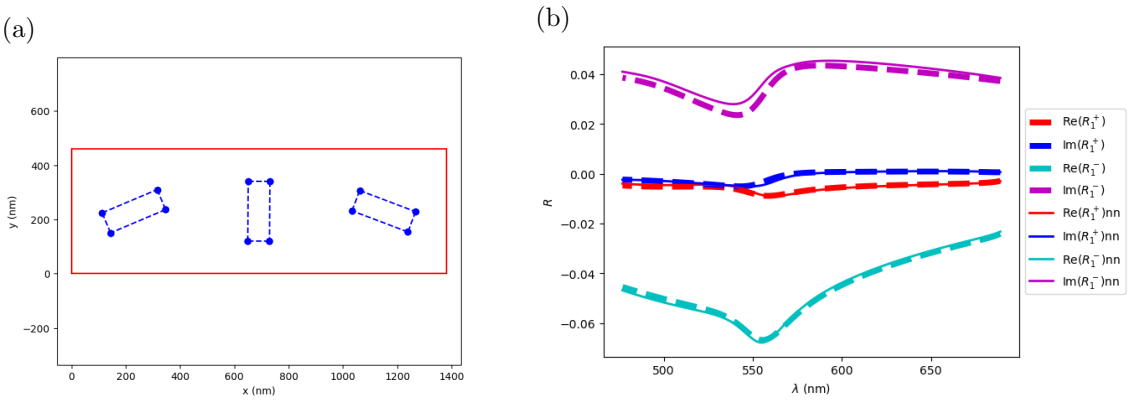


Figura 5: (a) Distribución de agujeros en la celda unidad. (b) Espectros resultante para el modo Bragg $G = +1$ para la estructura de (a) con polarización circular incidente $\sigma = +$ y la predicción de la red en función de λ.

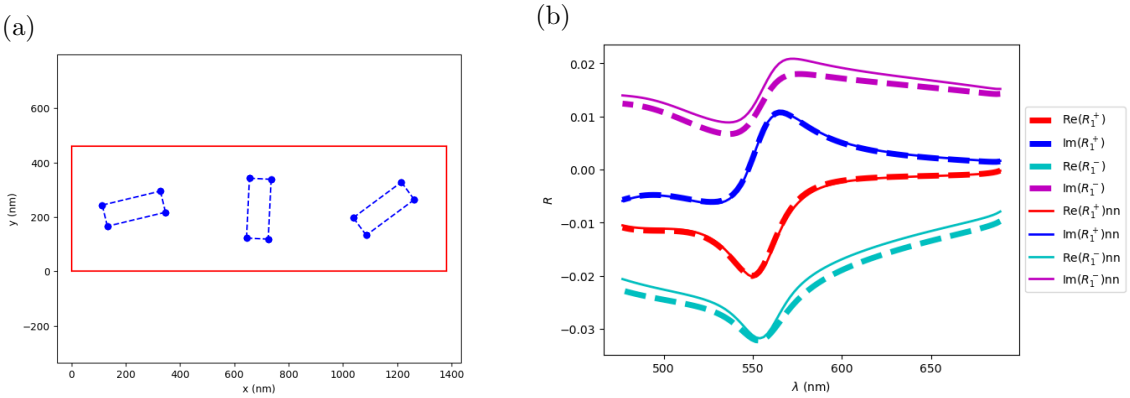


Figura 6: (a) Distribución de agujeros en la celda unidad. (b) Espectros resultante para el modo Bragg $G = +1$ para la estructura de (a) con polarización circular incidente $\sigma = +$ y la predicción de la red en función de λ.

5. Encoder y simetrías

Una ventaja de las redes neuronales es que son muy versátiles ya que pueden tomar formas muy variadas y resolver distintos tipos de problemas. Esto se debe a que las redes neuronales se pueden entender como un función que va de $\mathbb{R}^n$ a $\mathbb{R}^m$, donde n es el número de datos de entrada y m el de salida, y mediante el proceso de entrenamiento se encuentra una función $\vec{F}(\vec{x_n})$ que minimiza el error que hemos definido. Por ejemplo, en el caso de que quisiésemos determinar la configuración de los agujeros dado el espectro de uno de los modos de difracción, podríamos usar los datos que hemos empleado anteriormente para entrenar una nueva red que prediga los ángulos a partir del espectro. Lo único que tenemos que cambiar es que lo que antes era nuestra entrada (θ_i) es ahora nuestra salida y viceversa.

Por otro lado, las redes neuronales tiene una limitación, y es que por el propio proceso de aprendizaje tienen problemas a la hora de imitar funciones no inyectivas. Esto quiere decir que para dos o más elementos de la imagen existe el mismo elemento del dominio. Esto es lo que ocurre cuando queremos obtener los ángulos a partir del espectro.

La primera simetría y más obvia es que al trabajar con ángulos estos tienen una simetría inherente ya que $\theta = \theta + 2\pi$. Esto quiere decir que si rotamos todos los ángulos por una magnitud $\alpha = 2\pi$ el espectro va a ser el mismo. Una forma de solucionarlo es restringir el dominio y trabajar solo en el rango $\theta_i = [0, 2\pi]$. Además, como estamos usando agujeros rectangulares nuestro espectro va a tener otra simetría, ya que si rotamos todos los ángulos por $\alpha = \pi$ la configuración de agujeros es idéntica. Al igual que en el caso anterior podemos solucionar esto restringiendo el rango de los ángulos siendo $[0, \pi]$, de esta forma hemos conseguido que para cada configuración de ángulos θ_i exista un solo espectro, ya que al no incluir el 0 y π en los ángulos no van a existir configuraciones de ángulos con el mismo espectro. En vez de trabajar con la parte real e imaginaria de los coeficientes de reflexión podemos trabajar con el módulo al cuadrado de estos, ya que es una magnitud más sencilla de medir en un laboratorio y que se relaciona con la intensidad de la luz. Para evaluar este comportamiento, generamos distintos conjuntos de entrenamiento con ángulos en $[0, 2\pi]$, $[0, \pi]$ y $[0, \pi/2]$. Usamos una red de dos capas ocultas y 200 neuronas en cada capa. Como lo que buscamos es reducir la información que se necesita para definir un espectro llamamos a esta estructura *encoder*.

Como nuestra función de coste es el error cuadrático medio (MSE) podemos cuantificar cuanto difiere de media la predicción con el esperado como $\delta y = \sqrt{\text{Loss}}$. En la Figura 7 a) observamos el proceso de entrenamiento en los tres rangos de ángulos. En todos los casos, el valor de la función de coste es significativamente alto, lo que indica que las predicciones de la red están lejos de ser precisas, ya que de media $\delta_y = 0.25$ en los datos normalizados. Es decir, la red no está logrando aprender. Lo que está pasando al entrenar con solo el módulo es que los espectros han perdido un grado de libertad, el correspondiente a la fase, y aparece una nueva simetría, en este caso la simetría es que $\theta_i = \theta_i + \alpha$ siendo α un número real, una simetría de fase global. Es decir, los espectros de $|R^{\pm}_{1,2}|^2$ pueden ser generados por varias configuraciones de agujeros cuya única diferencia sea esta fase global, imposibilitando que la red aprenda.

En la figura 7 b) observamos como varía la función Loss en cada uno de los casos cuando entrenamos con la parte real e imaginaria de los coeficientes de reflexión. A primera vista vemos que en el rango $[0, 2\pi]$ obtenemos el peor resultado prediciendo los ángulos obteniendo Loss $\approx 10^{-1}$ esto quiere decir que de media la red se equivoca en cada predicción $\delta y_{[0,2\pi]} = 0.31$. Este es un valor bastante elevado y se debe a que queremos que para un mismo espectro nos dé dos configuraciones de ángulos diferentes, para θ_i y $\theta_i + \pi$. Cuando restringimos los ángulos de $[0, \pi]$ obtenemos $\delta y_{[0,\pi]} \approx 0.08$. Este es un valor más adecuado ya que como estamos trabajando con magnitudes normalizadas quiere decir que la red tiene de media un 8 % de error. Por último, hemos restringido aun más el rango de los ángulos, y hemos obtenido un mejor resultado siendo $\delta y_{[0,\pi/2]} = 0.028$. Aunque, en principio, al restringir los datos de $[0, \pi]$

(a) módulo al cuadrado de $R^{\pm}_{1,2}$.

(b) Parte real e imaginaria de $R^{\pm}_{1}$.

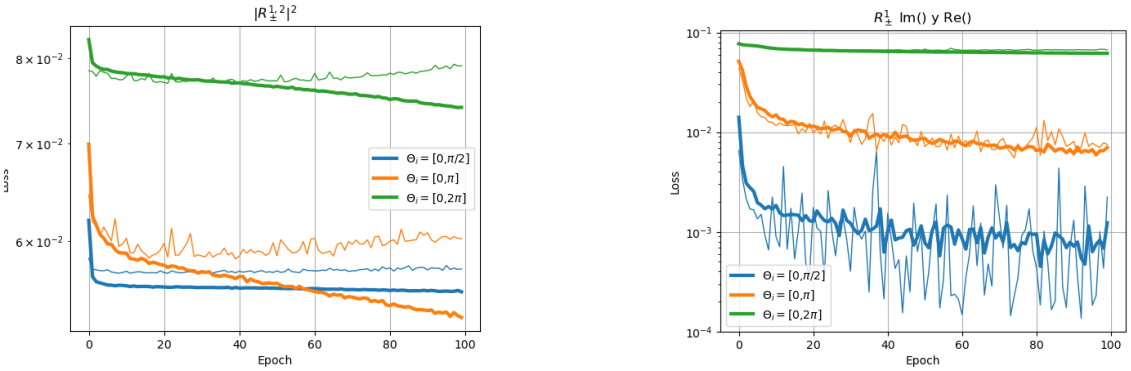


Figura 7: Entrenamiento de un *encoder* dependiendo del rango de ángulos. Incluimos los datos sobre el conjunto de validación para asegurarnos de que aprende correctamente, siendo las líneas gruesas Loss_{Tr} y las finas Loss_{Val}.

hemos conseguido que todas los espectros tengan una única configuración de ángulos parece que la red tiene problemas a la hora de reproducir variables cíclicas. Esto se puede deber a que cuando nos encontramos en el rango de $[0, \pi]$ podemos tener en el conjunto de entrenamiento dos espectros muy parecidos pero con ángulos $(\theta_1, \theta_2, 0)$ y $(\theta_1, \theta_2, \pi)$ o valores muy cercanos a estos. Esto puede producir un aumento en la función Loss ya que aunque físicamente 0 y π sean el mismo ángulo esto la red no lo sabe y asigna un error más grande de lo que debería. Las redes neuronales son eficaces en diseño inverso, encontrando parámetros que dan lugar a un resultado deseado, pero pueden tener problemas con las simetrías del sistema que se estudie.

6. Autoencoder y grados de libertad

Un *autoencoder* es un tipo de estructura de red neuronal en el que, al contrario que en los casos anteriores, se usa un aprendizaje no supervisado. Esto quiere decir que los datos que le suministramos a la red para su aprendizaje no están etiquetados de modo que la red aprende los patrones y correlaciones de nuestro conjunto de entrenamiento. En este tipo de redes neuronales se busca que la salida sea igual que la entrada, pero con una peculiaridad, y es que los *autoencoder* tienen un cuello de botella tal y como podemos ver en la Figura 8. Con este cuello de botella en el medio de la red conseguimos separar nuestra red original en dos subredes, la primera se llama *encoder* y su función va a ser la de reducir al máximo la dimensión de nuestros datos representándolos en un nuevo espacio de parámetros llamado espacio latente. Por otro lado tenemos el *decoder* cuya función es a partir de la representación en el espacio latente obtener los datos originales.

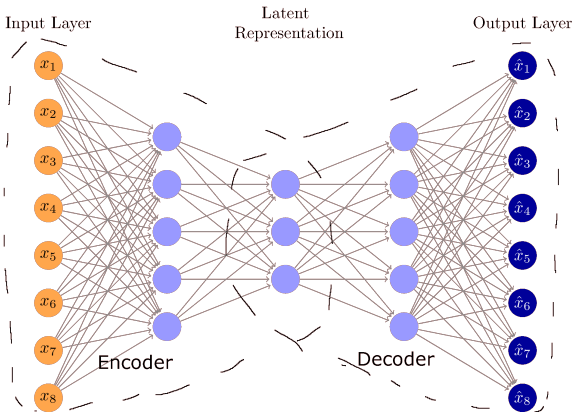


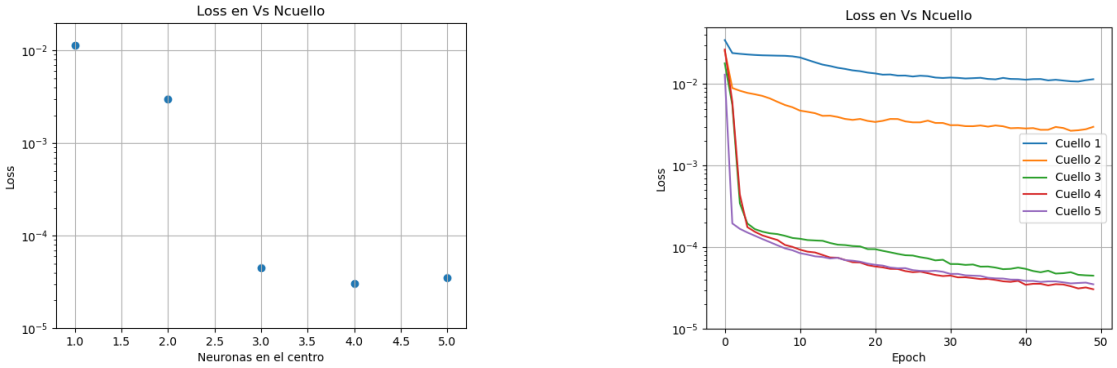
Figura 8: Estructura de un autoencoder.

6.1. Ejemplo de un autoencoder

Para entender un poco como funcionan los *autoencoder* vamos a proponer el siguiente ejemplo. Generamos una red la cual queremos que replique una función $F(t)$ como la de la ecuación 26. Para ello discretizamos t en 50 pasos para que la entrada de la red sea el valor de la función en $t = (0, 2)$. De esta forma nuestra red tendrá 50 neuronas de entrada y 50 de salida.

$$F_{prueba}(k_1, k_2, k_3, t) = k_1 \cos(2\pi t) + k_2 \cos(4\pi t) + k_3 \cos(6\pi t) \tag{26}$$

Generamos nuestros datos de entrenamiento escogiendo valores de $k_i = (0, 3)$ y realizamos distintos entrenamientos varíando la cantidad de neuronas que hay en el centro. Por como es la ecuación 26 vemos que va a tener 3 grados de libertad, ya que en cada caso los k_i se escogen aleatoriamente. Podríamos esperar que la red no logre representar correctamente la función $F(t)$ cuando en número de neuronas en el centro es menor que el número de grados de libertad. Esto es lo que se observa en las gráficas de la Figura 9. Cuando el número de



(a) Loss final en función del número de Neuronas en la constricción del autoencoder.

(b) Loss en función de la época.

Figura 9: Entrenamiento de un autoencoder con una función de prueba.

neuronas es mayor o igual que tres el valor de Loss llega a un mínimo y no parece mejorar con el aumento del número de neuronas. De estas gráficas podemos suponer que en el conjunto

de datos tiene tres grados de libertad. Por otro lado podemos intentar visualizar el espacio latente que ha generado nuestro *autoencoder* cuando tenemos tres neuronas en el centro. Para ello vamos a obtener el valor de las neuronas en el centro para cada dato de entrada. Después con esto entrenamos otra red que tenga como entrada los valores del espacio latente normalizados y como salida el valor de la función $F(t)$. Una vez está entrenada y el valor de Loss es los suficientemente bajo (10^{-4}) podemos ver cuál es la función que más excita la primera neurona cuando usamos como entrada el vector $(1, 0, 0)$ y vemos la salida. También lo podemos hacer con los vectores $(0, 1, 0)$ y $(0, 0, 1)$. Estas funciones las podemos ver en la Figura 10. Uno podría esperar que las funciones base fuesen funciones trigonométricas con una sola frecuencia ya que la función $F(t)$ esta compuesta de tres funciones $\cos(t)$. Pero debido a que las redes neuronales son funciones no lineales las funciones base no tienen por qué ser funciones simples que generen todo el espacio imagen mediante combinaciones lineales.

(a) $f_{lat}(1, 0, 0, t)$. (b) $f_{lat}(0, 1, 0, t)$. (c) $f_{lat}(0, 0, 1, t)$.

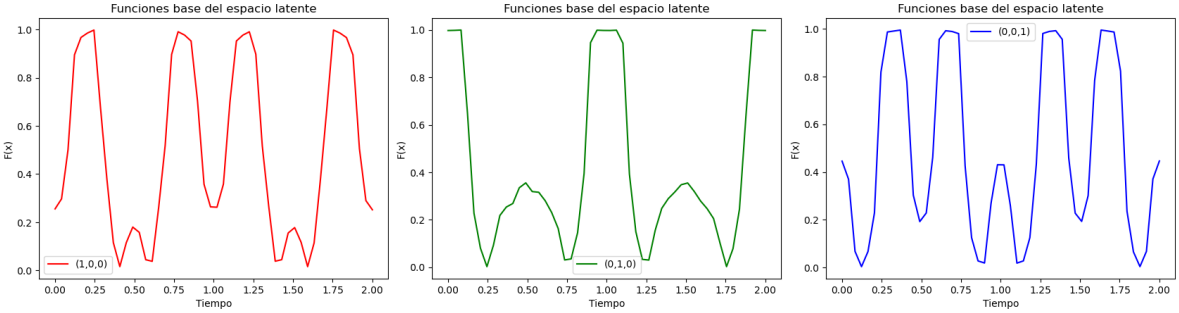


Figura 10: Funciones base del espacio latente en función del tiempo (u.a).

6.2. Autoencoder aplicado a la difracción de ondas

También podemos usar los *autoencoders* para la reflexión de ondas, lo único que tenemos que modificar es la estructura de nuestra red. En nuestro caso vamos a estar trabajando con la parte imaginaria y real de $R_1^{\pm}$ por lo que nuestra red va a tener que analizar 4 espectros con 100 λ' s cada uno. La estructura de capas ocultas que hemos escogido es $400 - 200 - N_c$ para el *encoder* y la estuctura reflejada para el *decoder*, donde N_c es el número de neuronas en el centro. Debido a que tenemos bastantes capas ocultas vamos a usar la función de activación ReLu ya que es la que mejor resultados produce en *Deep Learning*[5]. Además, por el propio método del descenso del gradiente corremos el riesgo de acabar en un mínimo local. Por esto, vamos a realizar varios entrenamientos con distintas configuraciones iniciales y nos quedaremos con el mejor valor de la función de coste para caracterizar cada estructura posible. Por otro lado, también vamos a estudiar qué es lo que pasa cuando restringimos los valores de θ_i, ya que en el apartado anterior hemos visto que las simetrías pueden provocar que el *encoder* no funcione correctamente. Sin embargo, al ser el aprendizaje no supervisado

[5]Aprendizaje de redes neuronales con muchas capas ocultas.

no se le van a suministrar los ángulos de los agujeros y las simetrías angulares del problema no deberían afectar al rendimiento del *autoencoder*. Además también hemos añadido una serie de entrenamientos con un número reducido de datos, ya que al reducir el rango de los ángulos tenemos una muestra más densa del espacio de espectros. De esta forma reduciendo el número de datos de entrenamiento por un factor 2^3 nos aseguramos de que la red se entrena con la misma densidad de datos. En la Figura 11 podemos observar cómo existe una tendencia de la

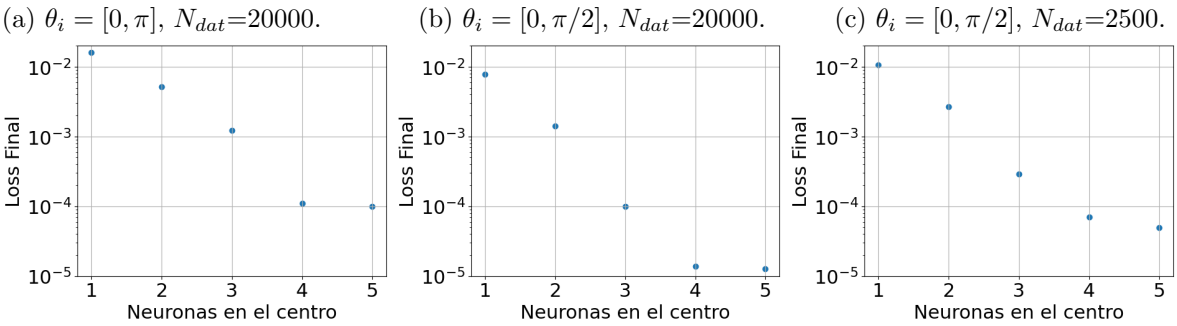


Figura 11: Entrenamiento de $R_1^{\pm}$ con estructura de capas ocultas 400-200-N$_c$.

función Loss que va con $1/N_c$ hasta que llega a $N_c = 4$ donde a partir de ahí se estabiliza en un cierto valor de Loss que depende del rango de θ_i y del número de datos de entrenamiento N_{dat}. Como los resultados que esperábamos era que a partir de $N = 3$ el valor de Loss se estabilizara hemos repetido el procedimiento pero añadiendo una capa oculta a cada lado del cuello de botella con 90 neuronas, de esta forma podríamos saber si estamos limitados por el número de parámetros del modelo. Esto lo podemos encontrar en la Figura 12. Al

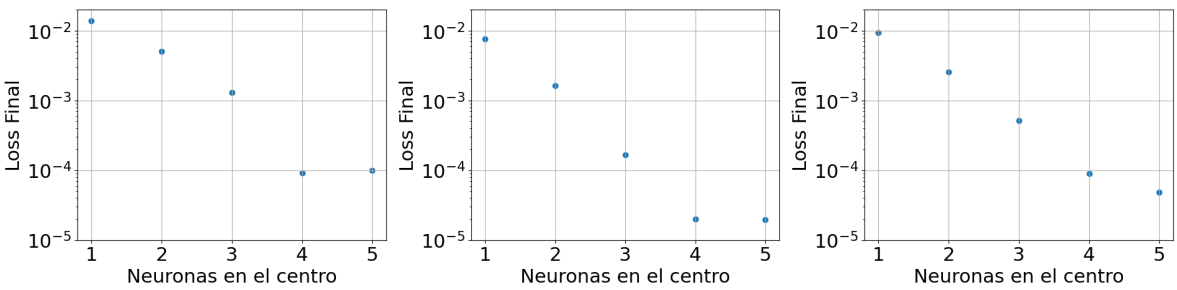


Figura 12: Entrenamiento de $R_1^{\pm}$ con estructura de capas ocultas 400-200-90-N$_c$.

añadir dos capas ocultas lo que encontramos es que el resultado no varía significativamente y seguimos viendo como el valor de Loss se mantiene más o menos estable a partir de $N = 4$. Esto sugiere que tenemos un grado de libertad más de lo que esperábamos, ya que es nuestro sistema tenemos solamente tres agujeros. Por otro lado, tenemos otra magnitud con lo que podemos experimentar, esta es el módulo del coeficiente de reflexión $|R_{1,2}^{\pm}|^2$. Como ya hemos visto en el apartado anterior al realizar el módulo de $R_{1,2}^{\pm}$ estamos perdiendo un grado de libertad por lo que podríamos esperar que a la hora de entrenar un *autoencoder* de la misma manera el valor de Loss se estabilice antes. Al igual que en el caso anterior vamos a estudiar

dos estructuras diferentes. Los resultados obtenidos los podemos encontrar en las Figuras 13 y 14.

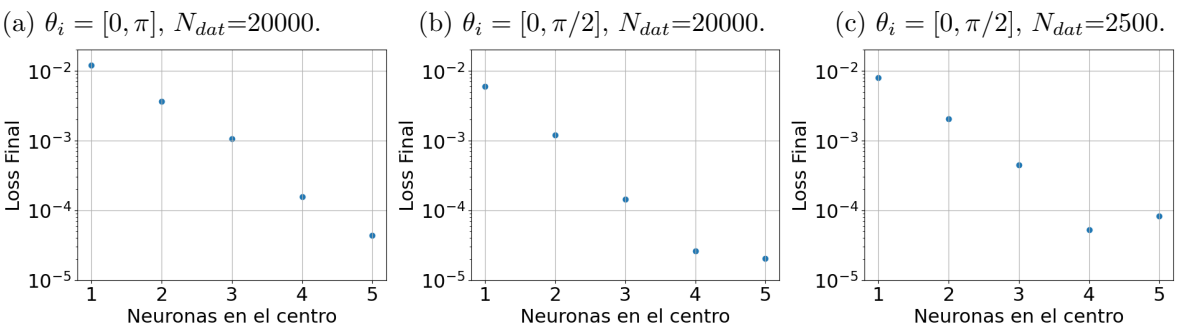


Figura 13: Entrenamiento de $|R^{\pm}_{1,2}|^2$ con estructura de capas ocultas 400-200-N$_c$.

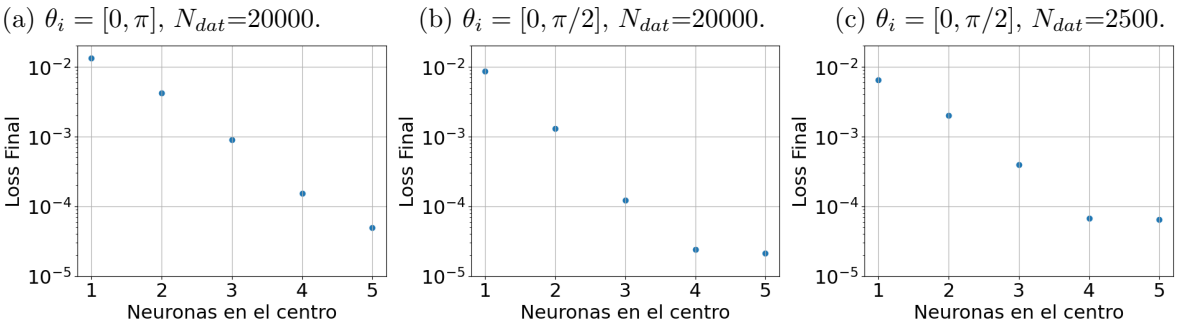


Figura 14: Entrenamiento de $|R^{\pm}_{1,2}|^2$ con estructura de capas ocultas 400-200-90-N$_c$.

A pesar de haber perdido un grado de libertad en los datos de entrenamiento los resultados siguen siendo parecidos para el caso de $\theta_i = [0, \pi/2]$. Sin embargo, en el caso de $\theta_i = [0, \pi]$ obtenemos una completa dependencia del valor de Loss con $1/N$ en el rango estudiado. No hemos sido capaces de explicar por qué ocurre este comportamiento tan extraño del *autoencoder* con el rango de θ_i y de N_c. Una forma de justificar este comportamiento viene de representar el espacio latente con $\theta_i = [0, \pi]$ y $N_c = 3$ del *autoencoder* entrenado con la parte real e imaginaria de $R^{\pm}_1$. Para lograrlo necesitamos asociar cada valor del espacio latente con un ángulo. Esto lo podemos hacer ya que los datos de entrenamiento los hemos generado nosotros y sabemos los ángulos que les corresponde a cada uno. Una vez tenemos los datos correctamente asociados podemos entrenar una red que tenga como entrada los valores del espacio latente y que tenga como salida los ángulos de los agujeros. Una vez entrenada generamos unos vectores de entrada que van a estar uniformemente distribuidos (l_1, l_2, l_3) con $l_i \in (0, 1)$. Después, para cada entrada obtendremos sus ángulos correspondientes. Para visualizar estos datos hemos decidido representar todos los ángulos de salida en un cubo de lado 1. Esto es lo que podemos ver en la Figura 15. Lo que observamos es que hay regiones del espacio de salida que no somos capaces de acceder desde el espacio latente. Esto podría justificar porque necesitamos una dimensión más.

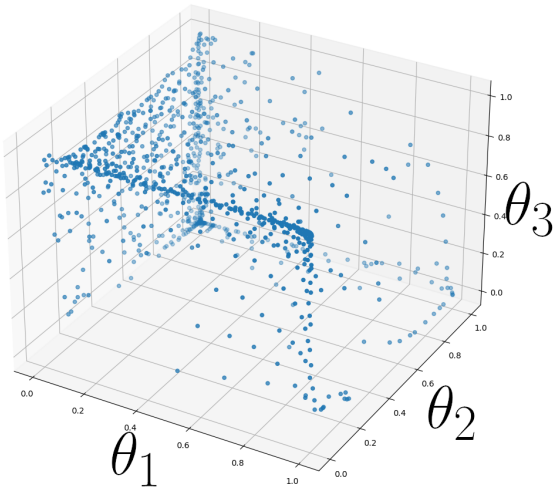


Figura 15: Representación del espacio Latente para $\theta_i = [0, \pi]$ y $N_c = 3$.

7. Aplicaciones de un autoencoder

Como ya hemos visto, mediante el uso de *autoencoders* podemos clasificar un conjunto de datos que no está etiquetado. Además mediante su representación en el espacio latente podemos comprimir la información de nuestro conjunto de entrenamiento mediante un *encoder* y recuperar los datos originales con un *decoder*. Esto significa que podríamos reducir la información de los espectros en un factor 100. Pero esta no es la única utilidad que tienen.

7.1. Detección de anomalías

La detección de anomalías es una aplicación común de los *autoencoders*. El objetivo de la detección de anomalías es identificar patrones en los datos que se desvían significativamente de lo normal. Los *autoencoders* se utilizan en esta tarea utilizando la diferencia entre los datos de entrada y su reconstrucción para detectar datos anómalos. La idea es que los datos anómalos no se ajusten bien a la representación comprimida aprendida por el *autoencoder*, lo que se traduce en una mayor diferencia entre los datos de entrada y su reconstrucción. Por lo tanto, los puntos con una mayor diferencia son considerados como anomalías. Para comprobar esta afirmación vamos a introducir en el conjunto de validación dos espectros completamente arbitrarios. Estos los podemos encontrar en la Figura 16, así como las predicciones del autoencoder. En el primero tenemos una dependencia lineal con la longitud de onda, mientras que la segunda es una función que depende del coseno.

En esta figura podemos observar que, a pesar de que la red tiene como objetivo devolver la misma función que en entrada, no consigue reproducir correctamente estas anomalías. Esto se debe durante el entrenamiento la red solo ha visto espectros producidos por la reflexión de ondas con tres agujeros. Todo lo que no tenga esos patrones no va a poderse replicar al pasarlo

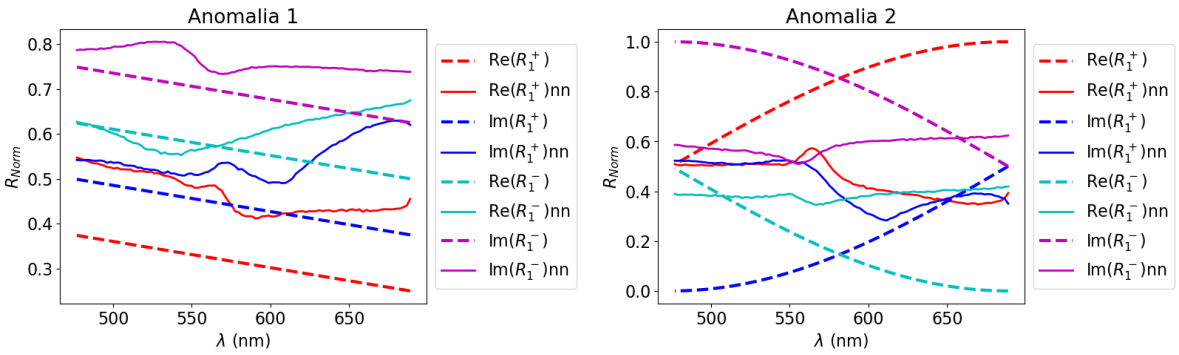


Figura 16: Anomalías introducidas en el conjunto de validación y su predicción.

por el *autoencoder*. Podemos detectar estas anomalías cuando comparamos cuánto dista la predicción de la red con el valor original con los valores del conjunto de validación. Esto es lo que encontramos en la Figura 17. Vemos que existen dos datos que tienen un valor de Loss varios órdenes de magnitud por encima de la media. Estos son los correspondientes a las dos anomalías, confirmando así que los *autoencoders* pueden usarse para detectar anomalías en espectros experimentales.

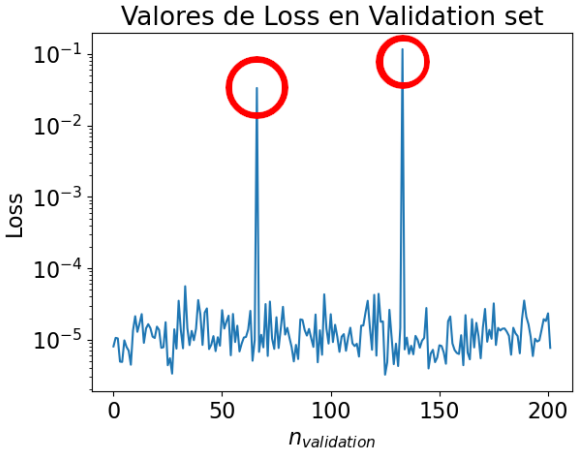


Figura 17: Loss en el conjunto de validación.

7.2. Filtrado de ruido

El filtrado de ruido es otra aplicación común de estos modelos. El ruido es un problema común en los datos, ya que puede afectar negativamente al rendimiento de los modelos de aprendizaje automático. La idea es que el ruido es diferente en cada caso y como el *autoencoder* está diseñado para comprimir datos buscando los patrones comunes, este debería ser capaz de eliminar el ruido. Al decodificar la representación comprimida, el *autoencoder* debería reconstruir los datos limpios y no el ruido.

Por otro lado, esta técnica requiere una gran cantidad de datos ya que, en caso contrario, corremos el riesgo de que nuestro modelo aprenda a reproducir el ruido. En nuestro caso, aplicamos un ruido que sea proporcional a $x(x-1)$, para asegurarnos de que los valores

que toma los datos de entrenamiento están entre 0 y 1. Esto se debe a que, aunque haya ruido, el detector no medirá valores negativos de intensidad y tampoco podremos obtener valores de transmisión mayores que 1, ya que supondría que se refleja mas luz de la que esta incidiendo. En el entrenamiento con datos ruidosos obtenemos un valor de la función de coste más elevado que en el caso sin ruido, pero esto se debe a la desviación provocada por el ruido. En la Figura 18 observamos el resultado del entrenamiento para un ejemplo en el conjunto de validación. Como el ruido lo hemos introducido artificialmente podemos comparar el valor original con la salida de la red. A pesar de la influencia del ruido, el *autoencoder* ha aprendido a reconocer los patrones subyacentes de los espectros que le suministramos, minimizando el efecto del ruido en la predicción de la red. Así pues, confirmamos que los *autoencoders* son una herramienta efectiva para reducir el ruido en espectros experimentales, ya que al tener que codificar toda la información en unos pocos parámetros, la red no puede "aprender" los patrones del ruido.

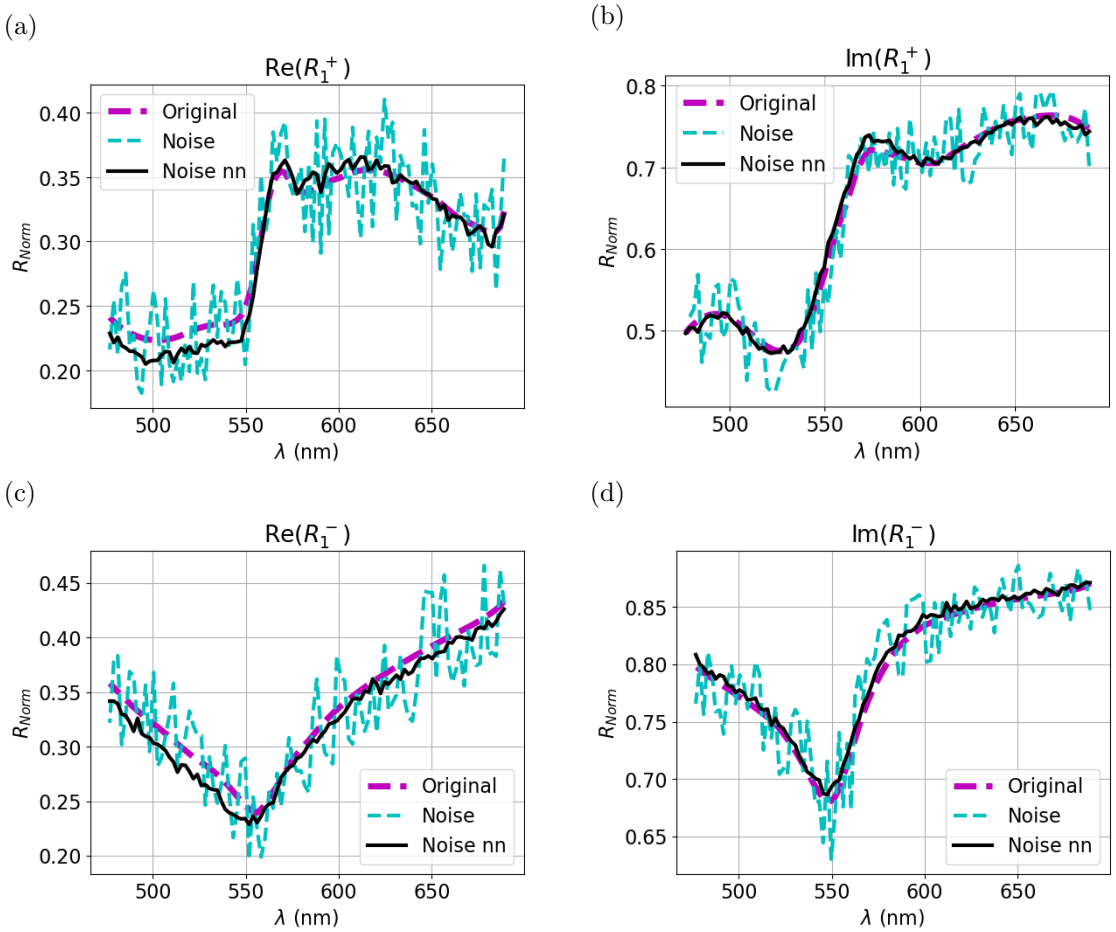


Figura 18: Comparación entre los datos originales (Original), los datos de entrenamiento (Noise) y la predicción de la red (Noise nn).

8. Conclusiones

En este trabajo hemos comprobado que las redes neuronales son capaces de "resolver las ecuaciones de Maxwell" para el sistema que hemos estudiado y nos han ayudado a comprender diferentes propiedades del sistema como sus simetrías y los grados de libertad. Toda la parte de programación y obtención de datos la hemos desarrollado con el lenguaje de programación *Python*, ya que con las librerías *TensorFlow-2.10* y *Keras* se simplifica mucho el entrenamiento y generación de las redes. Utilizamos *Matplotlib* para visualizar los datos, y las librerías *Numpy* y *Pandas* nos ayudaron a manejar grandes cantidades de información.

Primero, evaluamos el impacto de los hiperparámetros en la eficiencia del entrenamiento. Luego, con las propiedades de las redes neuronales, examinamos el impacto de las simetrías en el entrenamiento. Descubrimos las limitaciones de las redes al representar funciones no inyectivas y variables cíclicas. Por último, usamos *autoencoders* para analizar los grados de libertad en un conjunto de datos no etiquetados. Además, demostramos que estas redes pueden ser utilizadas para detectar anomalías y reducir el impacto del ruido en los datos de entrenamiento.

No hemos sido capaces de explicar el aparente aumento de grados de libertad de nuestro sistema en el estudio con *autoencoders*, punto sobre el que sería necesario investigar más. Además en casi todo este trabajo nos hemos limitado a estudiar un solo modo Bragg, ya que hemos realizado los entrenamiento con una CPU comercial. En el caso de hacer un estudio con más modos Bragg sería necesario realizarlo en un ordenador con más capacidad computacional, o realizar el entrenamientos en GPU's, ya que estas tienen un mayor rendimiento en tareas paralelizables.

9. Referencias

[1] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[2] F. Rosenblatt. The perceptron - a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York, January 1957.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[4] A. Fawzi, M. Balog, A. Huang, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[5] R. H. Ritchie. Plasma losses by fast electrons in thin films. *Phys. Rev.*, 106:874–881, June 1957.

[6] T. W. Ebbesen, H. J. Lezec, H. F. Ghaemi, T. Thio, and P. A. Wolff. Extraordinary optical transmission through sub-wavelength hole arrays. 391(6668):667–669, February 1998.

[7] F. Lorén, G. L. Paravicini-Bagliani, S. Saha, J.Gautier, M. Li, C. Genet, and L. Martín-Moreno. Microscopic analysis of the spin-momentum locking on a geometric phase metasurface. 2022. arXiv:2205.10245.

[8] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018.

Lista de Figuras