Facultad de Ciencias

Trabajo Fin de Grado

Operadores matemáticos en Física simulados mediante Inteligencia Artificial

Autor:

Javier Pardos Cardiel

Directores:

Dr. Sergio Gutiérrez Rodrigo

Dr. Luis Martín Moreno

Septiembre 2022

Resumen

En este trabajo se utilizan redes neuronales con el propósito de simular operadores matemáticos aplicados a la Física. Para ello, nos hemos basado en el Teorema Universal de Aproximación de Operadores, que dice que una red neuronal con una simple capa oculta puede aproximar con precisión cualquier operador continuo no lineal. Este teorema sugiere el uso de redes profundas en el aprendizaje de operadores continuos a partir de datos generados aleatoriamente.

En muchos problemas estudiados en Física se hace uso de operadores matemáticos a modo de aplicaciones entre funciones de un espacio de estados físicos en otro espacio de estados físicos. Estos operadores son de gran utilidad tanto en la mecánica clásica como en la cuántica. En el primer caso, la mayoría de ellos están relacionados con las simetrías, las cuales están a su vez ligadas a las leyes de conservación que caracterizan al sistema (Teorema de Noether). En el segundo, representan la base de la formulación matemática de la mecánica cuántica, donde cada observable está asociado a un operador hermítico con valores propios reales.

Hasta hace pocos años, la aplicación final de los operadores se realizaba mediante métodos clásicos de álgebra y cálculo. Ha sido recientemente cuando nuevas técnicas basadas en Inteligencia Artificial se han incorporado como una herramienta novedosa capaz de actuar como un operador matemático.

Como prueba del funcionamiento de estas técnicas, se ha elegido un operador relativamente sencillo como es el del oscilador armónico. Tras realizar diferentes ensayos, se ha obtenido un resultado bastante satisfactorio que ha sido aplicado a distintas fuentes de origen físico (función escalón, impulso...) para analizar varios casos y sacar conclusiones al respecto.

Índice

Apartado **Página**

1. Introducción

La Inteligencia Artificial (IA) se refiere al conjunto de actividades realizadas por una máquina que intentan replicar comportamientos considerados inteligentes por parte de los humanos. Hoy en día, abarca multitud de campos tales como el reconocimiento de imágenes, la conducción autónoma o los sistemas de recomendación. En todas estas tareas, las IAs deben pasar por un proceso de aprendizaje conocido como Aprendizaje Automático o *Machine Learning*, proceso en el que destaca el papel del aprendizaje profundo (*Deep Learning*) y el procesamiento del lenguaje natural [1].

A pesar de que el concepto de IA surgió en torno a 1950 cuando se planteó si se podría programar una computadora para que "pensase" por sí sola, en los últimos años ha experimentado un gran desarrollo debido a la accesibilidad a grandes cantidades de datos que le permite a la máquina procesarlos y reconocer patrones en los mismos. No menos importante ha sido el continuo avance de la potencia y la memoria computacionales, además del empeño de algunas multinacionales como Google, que han puesto parte de su conocimiento al alcance del público general a través de herramientas como *TensorFlow* y *Google Colab*, que se describen más adelante.

El abanico de áreas al que ha llegado la IA es de lo más amplio y destaca en tareas de regresión, clasificación y predicción. Además, más allá de facilitar el día a día a la población en general, se está empezando a extender su uso en ciencia. Por ejemplo, sobresalen las investigaciones en diagnóstico de enfermedades o predicción de estructuras proteicas a partir de la secuencia del DNA en medicina, así como la clasificación de galaxias en cosmología (tareas de *clustering* en IA) y la sustitución de costosas simulaciones numéricas empleadas para reproducir el comportamiento de fluidos [2].

En cuanto a la estructura del trabajo, se pueden distinguir los siguientes apartados: tras exponer los objetivos y la metodología empleada, en la sección 3 se explican algunos conceptos básicos, tales como las redes neuronales y la base matemática de los operadores, espacios en los que están definidos y teoremas en los que se ha basado el trabajo. A continuación, se presentan los resultados que se han ido obteniendo en el apartado 4, desarrollando el problema y su aplicación a diferentes casos de interés, así como las posibles debilidades del método que se hayan podido encontrar. Finalmente, se expondrán las conclusiones extraídas tras la elaboración del trabajo.

2. Objetivos y metodología

El principal objetivo del trabajo es generar una red neuronal que sea capaz de aprender la forma de cierto operador matemático continuo, esto es, una aplicación continua que "lleva" una determinada función de entrada a otra de salida, cada una en su espacio de funciones.

Para ello, el trabajo se enfoca en un operador relativamente sencillo, como es el del oscilador armónico, para plantear un problema abordable y determinar cómo de buena es la red neuronal entrenada. Se utilizan funciones generadas aleatoriamente para entrenar la red con la que se prentenden estudiar fuentes conocidas de origen físico con el fin de evaluar su funcionamiento. De esta manera, se pueden extraer conclusiones acerca de los puntos fuertes y débiles del problema resuelto.

La implementación de lo expuesto a partir de ahora ha sido realizada con el lenguaje de programación *Python*, que permite aprovechar la plataforma *TensorFlow* con sus librerías de *Keras* en el diseño y manejo de redes nueronales de forma más o menos sencilla. La ventaja que tiene es que permite "olvidarse" de la estructura más interna de la red, con el objetivo de poder centrarse en problemas más complicados y abstractos como el que se presenta. En particular, se ha utilizado *Google Colab* o *Colaboratory*, que permite programar y ejecutar *Python* en el navegador mediante un entorno interactivo o "cuaderno". Esta herramienta presenta diversas ventajas, especialmente en investigación en ciencia de datos e Inteligencia Artificial, y ejecuta código en los servidores en la nube de Google.

3. Conceptos básicos

Como paso previo al desarrollo del trabajo, es conveniente introducir algunos conceptos básicos que serán de gran utilidad para comprender las estructuras y el método empleados a lo largo del mismo. Por un lado, se explica toda la parte relacionada con la IA, desde las neuronas que conforman la red hasta los tipos de redes, pasando por las fases que constituyen el entrenamiento. Por el otro, se ven los operadores matemáticos y se formaliza el estudio en términos de espacios y conjuntos, más allá de ser tratados como "funciones entre funciones". Finalmente, se ve la teoría y los teoremas de *DeepONet*, en los que está basado todo el trabajo, así como la arquitectura de la red empleada en el mismo.

3.1. Redes neuronales

En el marco de las tareas de Inteligencia Artificial, y más concretamente de los procesos de *Machine Learning*, el uso de redes neuronales está muy extendido. La estructura de estas redes está inspirada en la que forman las neuronas en los seres vivos y de ahí el apellido de "neuronales". Si bien es cierto que queda mucho por descubrir dentro de la neurología, también lo es que el mecanismo neuronal se conoce desde hace más de 100 años y el hecho de imitarlo a la hora de que una máquina aprendiese una tarea pareció buena idea. Tanto es así que las redes neuronales han resultado ser una de las ramas de algoritmos de aprendizaje más utilizadas dentro del mundo del *Machine Learning*.

Partiendo de la neurona como elemento básico, estas se organizan en capas que se conectan entre sí dando lugar a la red, es decir, una red consta de una o varias capas consecutivas entre la información de entrada (*input*) y la de salida (*output*). Hoy en día, las redes más populares son las que acumulan muchas capas intermedias (*hidden layers*) en lo que se conoce como *Deep Learning*.

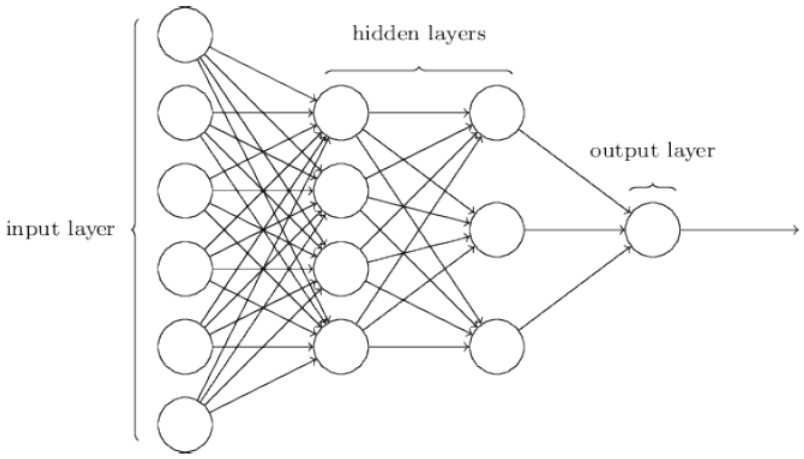


Figura 1: *Ejemplo de red con una capa de entrada, dos intermedias y una de salida mononeuronal. Fuente: [1]*

La popularidad del *Deep Learning* viene justificada por los buenos resultados que ofrece en diversos campos de aplicación. Esto se debe a que permite una mayor capacidad de abstracción y hace posible que una máquina realice tareas que un humano desarrolla de manera prácticamente inmediata, como puede ser el reconocimiento de los elementos de una imagen.

3.1.1. Neuronas

Como se ha comentado, la neurona es la unidad básica de procesamiento dentro de una red. Cada neurona recibe un *input* de la capa anterior y tras una serie de cálculos genera un valor de *output* que sirve de entrada para una o varias neuronas de la capa siguiente. El *input* que recibe cada neurona a través de las conexiones con las neuronas de la capa anterior depende del valor de salida de cada una de ellas x_j y del peso correspondiente a dicha conexión w_j, que marca el grado de intensidad con que una neurona afecta a la otra. Los cálculos realizados por la neurona consisten en una suma de los valores de entrada ponderada con sus respectivos pesos, a la que se aplica una función de activación $f(z)$, para dar lugar al *output* de la misma.

Utilizando notación vectorial, dada una capa $r \in 1, ..., R$ donde R es el número de capas de la red, se tiene para cada neurona k de dicha capa un vector de pesos $\mathbf{w_k^r} = (w_{k1}^r, ..., w_{kN}^r)$ que corresponde a las conexiones de la misma con cada una de las N neuronas de la capa anterior. También hay otro parámetro que caracteriza cada neurona llamado *bias* o sesgo, b_k^r.

De este modo, dado el *output* de una capa $r-1$, descrito por el vector $\mathbf{x^{r-1}} = (x_1^{r-1}, ...x_N^{r-1})$, se introduce un valor en la función de activación de la neurona $k \in 1, ..., N$ de la capa r dado por:

$$z_k^r = \sum_{j=1}^{N} w_{kj}^r \, x_j^{r-1} + b_k^r = \mathbf{w_k^r} \cdot \mathbf{x^{r-1}} + b_k^r \tag{1}$$

Se comprueba que este cálculo realizado por la neurona es una transformación lineal, que será seguida por una operación no lineal a través de la función $f(z)$. Esta es la base de las capas de neuronas, realizar sucesivas transformaciones lineales seguidas de no lineales hasta llegar al *output* (cada capa puede tener distinta función de activación). En el caso de no introducir las no linealidades, se podría sustituir una red de varias capas por una de una única capa al ser todas las operaciones lineales [3], de manera que no tendría sentido hablar de *Deep Learning*.

El *output* de una neurona k la capa r vendrá dado entonces por $f(z_k^r) \equiv a_k^r$. Tras atravesar toda la red, es decir, las R capas de neuronas, se tiene un valor a la salida $\mathbf{a^R}(\mathbf{w}, \mathbf{b})$ que puede ser un escalar o un vector, según si la capa de salida está formada por una o varias neuronas. Este valor es función de todos los pesos y *biases* de cada una de las capas, que a su vez son los parámetros que habrá que ajustar en el entrenamiento. Se puede

intuir que el número de parámetros se multiplica enseguida incluso en redes relativamente sencillas, por lo que el coste computacional puede incrementarse rápidamente.

3.1.2. Funciones de activación

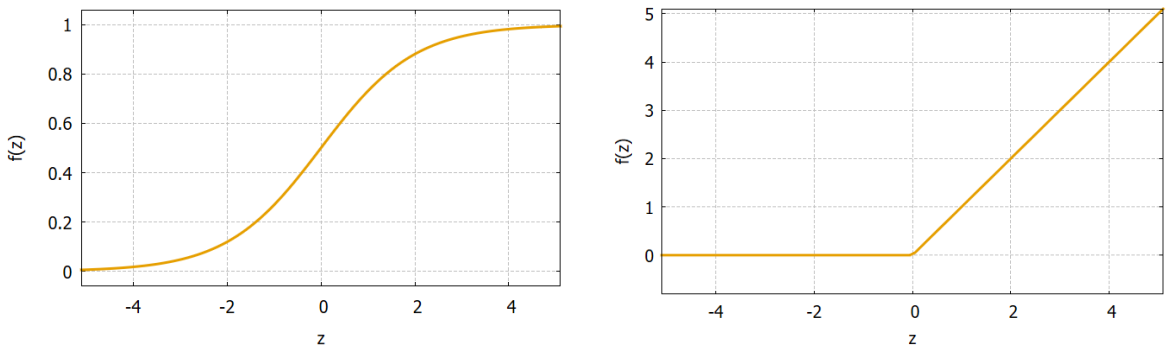
Las funciones de activación representan las transformaciones no lineales realizadas por cada una de las neuronas a los datos de entrada (dados por la expresión (1)) para dar lugar a una salida. Son muchas las funciones de activación existentes y según el objetivo de la red vienen mejor unas u otras. Entre las más extendidas se encuentran la *sigmoid*, la *ReLU* o la *softmax*.

La función de activación *sigmoid* tiene la forma de una sigmoide y toma todos sus valores entre 0 y 1. Para valores cada vez más negativos de z, la función tiende a 0, mientras que a medida que z crece positivamente, la función se acerca a 1. Es por ello por lo que presenta un buen rendimiento en la última capa a la hora de decidir entre pertenecer a una clase u otra. La forma de esta función es la siguiente y su representación aparece en la Figura (2a):

$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

Por otro lado, la palabra *ReLU* es el acrónimo de *Rectified Linear Unit* y es exactamente 0 para valores negativos de z, mientras que para los positivos es lineal $f(z) = z$. Presenta ventajas como su fácil implementación (frente a otros cálculos como la *tanh* o la exponencial en la sigmoide) y la ausencia de problemas de saturación, lo que la ha llevado a ser la más usada. Su forma funcional es la siguiente y aparece representada en la Figura (2b):

$$f(z) = máx\,(0, z) = \begin{cases} 0 & \text{si } z < 0 \\ z & \text{si } z \geq 0 \end{cases} \tag{3}$$



(a) *Función sigmoide.* (b) *Función ReLU.*

Figura 2: *Funciones de activación más utilizadas.*

Hoy en día, la función *ReLU* es la que se utiliza en la mayoría de problemas y, como regla general, se suele empezar probando por esta y en el caso de que no genere buenos resultados se cambia a otras. En este trabajo es la función que se ha empleado para las capas intermedias ya que proporcionaba buenos resultados. La función *sigmoid*, por su parte, se suele preferir en problemas de clasificación binaria, y la *softmax* en problemas de clasificación multiclase [4].

3.1.3. Función coste

Dentro de la fase de entrenamiento, se debe definir una función para analizar la eficiencia de la red y si esta está aprendiendo como debe. Es la llamada función coste $C(w, b)$ y da una medida de la desviación del valor predicho por la red $\mathbf{a^R}(\mathbf{w}, \mathbf{b})$ respecto del valor real $\mathbf{y}$. Cuanto más parecidos son estos valores, más se aproxima la función a 0. Este es el procedimiento que sigue el entrenamiento: variar los parámetros de los que depende la función (pesos y *bias*) con el fin de obtener una predicción óptima. Para ello, se calcula el gradiente del coste $\nabla C(w, b)$ respecto de sus parámetros con el objetivo de hallar el mínimo de dicha función coste. En problemas de regresión como el de este trabajo, la función más utilizada es la *mean squared error* que calcula el error cuadrático medio entre los valores reales y los predichos:

$$C = \frac{1}{m} \sum_{m} (\mathbf{a} - \mathbf{y})^2 \tag{4}$$

En la expresión (4) m es el número de datos utilizados en el entrenamiento y la suma se entiende sobre todos los *outputs* generados. Además, la operación vectorial $\mathbf{a} - \mathbf{y}$ se realiza elemento a elemento, es decir, $\mathbf{a} - \mathbf{y} = (a_1 - y_1, ..., a_N - y_N)$ donde N es el número de neuronas de la última capa.

A la vista del tratamiento de la función coste para encontrar su mínimo, se puede intuir que los cálculos relativos al gradiente pueden volverse pesados computacionalmente en cuanto el número de parámetros involucrados (y, por tanto, las derivadas a realizar) son elevados. Más aún, la propia estructura de las redes profundas hace que la variación de un parámetro influya en las siguientes capas de tal manera que se complique el proceso, aun cuando la función coste es sencilla como la (4). Para combatir esto se utiliza el llamado *backpropagation*, que calcula el gradiente recorriendo la capa en sentido inverso al natural, es decir, desde la última hasta la primera.

También hay que tener en cuenta que para obtener buenas predicciones se necesitan muchos datos. Esto supone un problema en relación al coste de cálculo, que se resuelve mediante la técnica de *stochastic gradient descent (SGD)*. En realidad, es una aproximación que consiste en asumir que el gradiente de la función coste tomando un subgrupo de los datos de entrada es similar al resultado de tomar todos los datos. El tamaño de dicho subgrupo se conoce como *mini-batch* (hiperparámetro de la red) y se debe elegir de tal forma que sea lo suficientemente grande como para arrojar buenos resultados pero no

tanto como para que requiera un tiempo largo en la fase de entrenamiento.

Este algoritmo del *SGD* introduce un nuevo hiperparámetro: el *learning rate* (designado con la letra η). Este controla la medida en la que cambia el modelo como respuesta al error calculado cada vez que los pesos son actualizados, es decir, cómo de rápido el modelo se adapta al problema. Normalmente, $\eta \in (0, 1)$ y su valor se debe elegir teniendo en cuenta que valores más pequeños necesitan más "épocas", debido a que las actualizaciones de los pesos en cada paso son pequeñas (pudiendo llegar a atascarse), mientras que valores más grandes dan lugar a cambios rápidos que pueden resultar en un peor aprendizaje o en procesos de entrenamiento inestables [5].

3.1.4. *Overfitting*

Durante el entrenamiento puede aparecer un problema conocido como *overfitting* o sobreajuste. Este se produce cuando la red pierde capacidad de generalización de las características de un conjunto de datos y extrae conclusiones concretas del subconjunto utilizado en el entrenamiento (*training set*). Esto puede ocurrir al incluir un gran número de parámetros que aumentan la complejidad de la red y da lugar a este hecho contraproducente. En estos casos, lo que sucede es que se obtiene una predicción muy buena en los datos de entrenamiento, mientras que al examinar otro conjunto distinto de datos (*validation set*) saldrían resultados distorsionados.

El problema del *overfitting* puede ser grave si no se es consciente del mismo, ya que *a priori* parece que la red entrena bien. Para evitarlo, lo ideal es aumentar el número de datos de entrada, de manera que la red no pueda incurrir en el error de particularizar las predicciones en torno a una serie de características comunes a ellos, al introducir una mayor dispersión. A su vez, conviene reducir el número de parámetros para que la red no sobreajuste y se vea forzada a generalizar los patrones encontrados en los datos. En lugar de reducir el tamaño y complejidad de la red, lo que se hace es recurrir a una técnica conocida como "regularización".

La regularización consiste en añadir un término extra a la función coste que penalice el hecho de haber muchos parámetros. Este término va multiplicado por un "regularizador" λ, que reduce el número de parámetros útiles cuanto mayor es. Es por ello que para elegir su valor hay que ser metódico y se suele escoger a partir de los datos del *validation set*. En caso de utilizar esta técnica, lo ideal es evaluar la red con un tercer conjunto de datos diferente al de entrenamiento y validación, conocido como *test set*. En conclusión, no se debe examinar el funcionamiento de la red en datos utilizados para ajustar parámetros (*training set*) o hiperparámetros (*validation set*).

Por último, se puede recurrir al *early stopping* para evitar el *overfitting*. Este consiste en detener el entrenamiento en el momento oportuno, lo que se realiza programando que se haga en distintas fases que se denominan "épocas" (*epochs*). El objetivo de esto es poder controlar el valor de la función coste y parar el entrenamiento cuando se considere que

es suficientemente pequeño y antes de que comience a perder generalidad sobreajustando particularidades de los datos de entrenamiento.

3.2. Operadores matemáticos y *DeepONet*

DeepONet o *Deep Operator Network* puede entenderse como un marco general de trabajo que utiliza el *Deep Learning* para el aprendizaje de operadores no lineales y continuos. Este se basa en una teoría que garantiza un error de aproximación pequeño. Además, las arquitecturas específicas que se utilizan también presentan pequeños errores de generalización (error de la red al analizar nuevos datos distintos a los de entrenamiento), siendo muy importante una representación adecuada del espacio de entrada del operador [6].

Más allá del teorema universal de aproximación de funciones por redes neuronales mediante aprendizaje supervisado, que consiste en que una red neuronal con una capa oculta y un número finito de neuronas puede aproximar funciones continuas con cualquier precisión [7], la idea de aproximar operadores no lineales por redes neuronales viene sustentada por un potente teorema menos conocido, el Teorema Universal de Aproximación de Operadores. Este teorema dice que una red con una simple capa intermedia puede aproximar con precisión cualquier funcional no lineal y continuo (aplicación de un espacio de funciones a los números reales) así como cualquier operador no lineal (aplicación de un espacio de funciones a otro).

3.2.1. Planteamiento del problema

Sea G un operador que tome una función u como entrada, y sea $G(u)$ la correspondiente función de salida. Para cualquier punto y en el dominio de $G(u)$, dicha salida es un número real $G(u)(y)$. De esta manera, la entrada de la red está compuesta por dos partes: u e y, y como salida tiene a $G(u)(y)$. En la práctica, lo que se hace es discretizar las funciones u de entrada para poder aplicar la aproximación de la red. La manera más sencilla de representar estas funciones en su espacio de entrada V es tomando el valor de las mismas en un número grande pero finito de puntos $\{x_1, ..., x_m\}$ que se conocen como "sensores", como se muestra en la Figura (3). Existen otras maneras de representar una función, como por ejemplo, mediante expansiones espectrales o por medio de una imagen. Se prevé que en el futuro dichas funciones serán representadas por otras redes neuronales más allá del *DeepONet*.

Para demostrar la capacidad y efectividad de esta técnica, se plantea el problema lo más general posible exigiendo las restricciones más débiles posibles tanto en los sensores como en los datos de entrada. Concretamente, la única condición requerida es que las posiciones de los sensores sean las mismas para todas las funciones de entrada u, mientras que para las localizaciones de salida y no se pide ninguna condición. Y aun dicha restricción

se puede relajar, por ejemplo, interpolando u en un conjunto común de localizaciones sensor o proyectando u en una base de funciones y usando los coeficientes a modo de representación de u.

3.2.2. Arquitectura de la red

Partiendo del objetivo de aprender operadores de una forma general, el único requerimiento para el *training set* va a ser la consistencia de los sensores $\{x_1, ..., x_m\}$ de las funciones de entrada, esto es, no será necesario que sus posiciones estén equiespaciadas. De esta manera, la entrada de la red está compuesta por dos componentes separadas: $(u(x_1), ..., u(x_m))^T$ e $\mathbf{y}$, y la salida se corresponde entonces con la aplicación de G sobre u tomada en el punto $\mathbf{y}$.

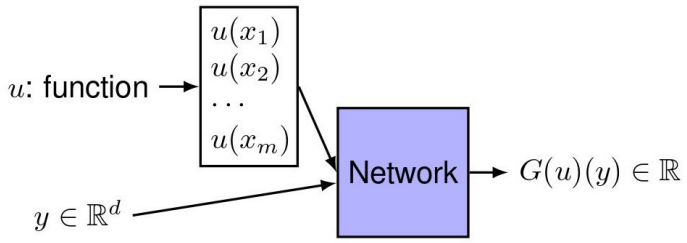
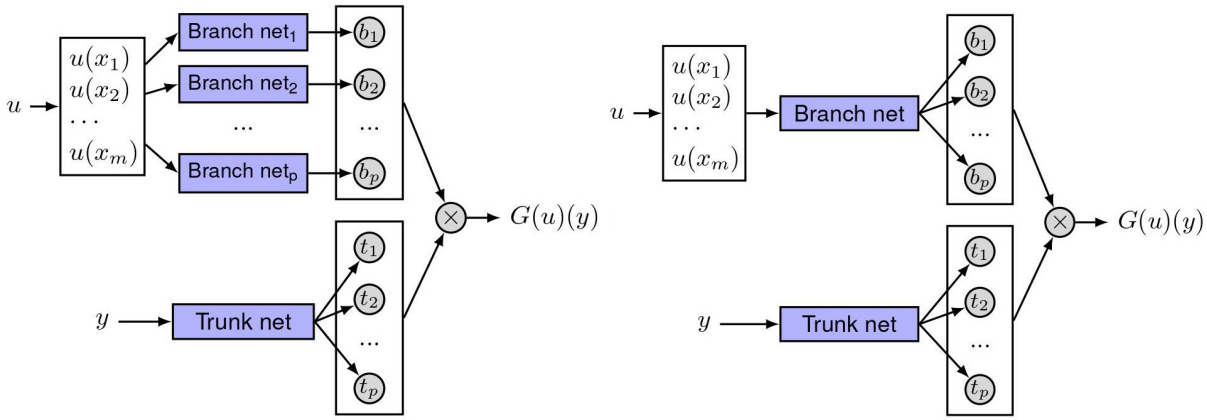


Figura 3: *Esquema de los inputs* $[(u(x_1), ..., u(x_m)), \mathbf{y}]$ *y el output* $G(u)(y)$ *para una red que aprenda un operador* $G : u \mapsto G(u)$. *Fuente: [6]*

Dentro de este apartado, se pretende diseñar una arquitectura para la red que dé lugar a una buena generalización. En problemas de varias dimensiones, $\mathbf{y}$ es un vector con d componentes, por lo que la dimensión de $\mathbf{y}$ ya no coincide con la de $\mathbf{u}(x_i)$ para $1 \leq i \leq m$. Este hecho evita tratar $\mathbf{u}(x_i)$ e $\mathbf{y}$ de igual manera y hace necesarias al menos dos subredes para considerar $\Big(u(x_1), ..., u(x_m)\Big)^T$ e $\mathbf{y}$ de manera separada.

Por un lado, parece necesario incluir una red (en adelante *trunk net*) que toma como entrada a $\mathbf{y}$ y como salidas $(t_1, ..., t_p)^T \in \mathbb{R}^p$. Por el otro, aparecen dos opciones. Se pueden colocar p redes (en adelante *branch nets*), de forma que cada una de ellas toma $\Big(u(x_1), ..., u(x_m)\Big)^T$ como entrada y tiene como salida un escalar $b_k \in \mathbb{R}$ para $1 \leq k \leq p$. Pero también, como en la práctica p suele ser de orden 10 o mayor, resulta más eficiente juntar todas estas redes en una única (*branch net*), es decir, una única red que tendrá como *output* el vector $(b_1, ..., b_p)^T \in \mathbb{R}^p$.

El primer caso, como presenta p redes apiladas en paralelo, recibe el nombre de *stacked DeepONet* como se ve en la Figura (4a), mientras que al segundo se le llama *unstacked* y aparece en la Figura (4b). La equivalencia entre ellos se puede entender viendo el segundo caso como uno apilado con todas las *branch nets* compartiendo el mismo conjunto de parámetros.



(a) *Stacked DeepONet.* (b) *Unstacked DeepONet.*

Figura 4: *Arquitecturas propuestas para la DeepONet. En (a) aparece una trunk net como una red de una capa y anchura p y p branch nets apiladas, cada una de ellas como una red de una capa oculta y anchura n. En (b) hay una trunk net y una branch net, que puede ser vista como una stacked DeepONet con todas las branch nets compartiendo el mismo set de parámetros. Fuente: [6]*

3.2.3. Teorema Universal de Aproximación de Operadores Generalizado

A la vista de las arquitecturas de red planteadas, tanto por claridad como por conveniencia, más allá del Teorema Universal de Aproximación de Operadores ya mencionado, se puede enunciar su versión generalizada. Este teorema prueba que las arquitecturas expuestas son también aproximadores universales de operadores y permite diferentes combinaciones de *branch/trunk nets*. A continuación, se enuncia sin demostrar:

Teorema 1 (Teorema Universal de Aproximación de Operadores Generalizado)**.** *Sea X un espacio de Banach*[1]*. $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ son dos subespacios compactos*[2] *en X y en $\mathbb{R}^d$, respectivamente, y V es un espacio compacto en $C(K_1)$. Sea $G : V \longrightarrow C(K_2)$ un operador no lineal y continuo. Entonces, para todo $\varepsilon > 0$ existen m y p enteros positivos, así como funciones vectoriales continuas $\mathbf{g} : \mathbb{R}^m \longrightarrow \mathbb{R}^p$ y $\mathbf{f} : \mathbb{R}^d \longrightarrow \mathbb{R}^p$, y $x_1, ..., x_m \in K_1$, tales que:*

$$\left| G(u)(y) - \langle \underbrace{\mathbf{g}(u(x_1), ..., u(x_m))}_{branch}, \underbrace{\mathbf{f}(y)}_{trunk} \rangle \right| < \varepsilon$$

se satisface para toda función $u \in V$ y todo $y \in K_2$, donde $\langle \cdot, \cdot \rangle$ denota el producto escalar en $\mathbb{R}^p$. Además, las funciones $\mathbf{g}$ y $\mathbf{f}$ pueden ser escogidas como distintas clases

[1]Un espacio de Banach es un espacio vectorial normado y completo (toda sucesión de Cauchy tiene límite en él) en la métrica definida por su norma. Típicamente, es un espacio de funciones de dimensión infinita.

[2]Un espacio compacto tiene propiedades similares a las de un conjunto finito en el sentido de que las sucesiones contenidas en un conjunto finito siempre contienen una subsucesión convergente. La noción de compacidad generaliza el concepto de cerrado y acotado de los espacios euclídeos.

de redes neuronales, las cuales cumplen el teorema clásico universal de aproximación de funciones.

Más allá de la formulación rigurosa, este teorema se puede interpretar teniendo en cuenta lo explicado hasta ahora. El operador G entre dos espacios compactos puede aproximarse por el producto escalar de dos funciones (cada una representada por una red, a través de las cuales se introducen los dos *inputs*), ya que la diferencia entre ambos términos está acotada por un valor ε que puede ser tan pequeño como se quiera.

También de acuerdo con el teorema (1), aunque $G(u)(y)$ tiene dos *inputs* independientes (de ahí que se utilice una red para cada uno), se puede entender que es una función de y condicionada a u. Por ello, las *DeepONets* pueden ser vistas como un modelo condicional, donde las salidas de ambas redes son unidas al final por medio de un producto escalar.

4. Resultados

A la hora de buscar resultados en base a toda esta teoría, lo que se ha hecho es escoger un problema que involucrase un operador y pensar en la manera de generar los datos con los que entrenar la red. El paso siguiente ha sido programar la red y asegurarse de que entrena bien. Por último, se ha aplicado a algún caso físico conocido para corroborar lo obtenido.

4.1. Definición del problema

Como se ha ido comentando, el problema estudiado pretende que una red neuronal aprenda la acción de un operador G. En concreto, se busca encontrar la solución de la ecuación diferencial del oscilador armónico sometido a una fuente de excitación u, es decir, la ecuación $L\,s = u \;\Leftrightarrow\; G\,u = s$, donde el operador L viene representado por:

$$G^{-1} \equiv L = A\,\frac{d^2}{dx^2} + B \tag{5}$$

Utilizando esta notación, u denota la función fuente y s su solución, $s\,(y) = G\,(u)(y)$. Para construir la red, se necesita un conjunto de funciones fuente y su solución $\{u\,(\mathbf{x}), s\,(\mathbf{y})\}$, así como las coordenadas donde se evalúa s, es decir, $\{y_1, ..., y_n\}$.

Mientras que la primera rama toma cada $u\,(\mathbf{x})$ en los mismos puntos $\{x_1, ..., x_m\}$, la segunda simplemente toma los puntos $\{y_1, ..., y_n\}$ en el dominio de definición de la solución $s\,(\mathbf{y})$. Cada una de estas soluciones $s\,(\mathbf{y})$ se evalúa en puntos aleatoriamente escogidos entre los de dicho dominio.

4.2. Generación de los datos

La manera de representar las funciones u y s se realiza expandiéndolas en una base de funciones adecuada para el problema. De esta manera, truncando la serie que representa cada función en $2\,N_s$ sumandos y utilizando cálculo matricial, se puede obtener la solución s para una excitación u dada. La base elegida por conveniencia es la de senos y cosenos, que puede ser expresada como una exponencial compleja, y da lugar a una expresión del operador en forma matricial que es diagonal. Se pueden expresar las funciones de la siguiente manera:

$$\begin{cases} u(x) = \sum_{l=0}^{N_s} u_l\, cos(2\pi l x) + \sum_{m=0}^{N_s} v_m\, sin(2\pi m x) \\ s(y) = \sum_{l=0}^{N_s} s_l\, cos(2\pi l y) + \sum_{m=0}^{N_s} t_m\, sin(2\pi m y) \end{cases} \tag{6}$$

Con la notación de Dirac, si se escribe la base de senos y cosenos como exponencial compleja y se denota al vector l-ésimo como $|l\rangle$, se obtiene el siguiente *bra-ket* $\langle x|l\rangle = e^{2\pi ilx}$, donde i es la unidad imaginaria. Debido a la elección de la base incluyendo el factor 2π, el dominio de las funciones a estudiar va a ser el $[0,1)$. Así, dado un $x \in [0,1)$ tal que el producto escalar de dos funciones está definido como $\langle f\,|\,g\rangle = \int_0^1 f(x)\overline{g(x)}dx$, se cumple que $\langle l\,|\,m\rangle = \delta_{lm}$ (delta de Dirac) por la ortonormalidad de la base escogida. El elemento de matriz L_{lm} de la matriz L se puede calcular de la siguiente forma:

$$L_{lm} = \langle l\,|L|\,m\rangle = \int_0^1 \langle l\,|x\rangle\langle x\,|L|\,m\rangle dx = \int_0^1 e^{-2\pi ilx}\left[\left(A\,\frac{d^2}{dx^2} + B\right)e^{2\pi imx}\right]\,dx =$$

$$= \int_0^1 e^{-2\pi ilx}\left(A\,(2\pi im)^2 + B\right)e^{2\pi imx}\,dx = \left(-4\pi^2 m^2 A + B\right)\underbrace{\langle l\,|\,m\rangle}_{\delta_{lm}}$$

De este modo, la matriz L toma la siguiente forma de matriz diagonal:

$$L = \begin{pmatrix} L_{00} & 0 & \cdots & 0 \\ 0 & L_{11} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & L_{N_sN_s} \end{pmatrix}$$

Por su parte, la matriz G es justo la inversa de L y sus elementos son justamente los inversos L^{-1}_{lm}. Por todo lo anterior, los elementos no nulos se dan para $l = m$ y, en ese caso, $L_{ll} = \left(-4\pi^2 l^2 A + B\right)^{-1}$. Se tiene entonces que el valor que toma la posición l de la solución s se calcula a partir de su correspondiente en u como:

$$s_l = L^{-1}_{ll}u_l = \frac{u_l}{(-4\pi^2 l^2 A + B)} \tag{7}$$

Con todo esto, cada función u se obtiene eligiendo valores aleatorios de $\{u_l, v_m\}$, que se van a tomar en un intervalo acotado $(-10, 10)$ para evitar problemas de escala con las funciones. Su correspondiente solución s se calcula a partir de ella como:

$$s(y) = \sum_{l=0}^{N_s} \frac{u_l}{(-4\pi^2 l^2 A + B)}\, cos(2\pi ly) + \sum_{m=0}^{N_s} \frac{v_m}{(-4\pi^2 m^2 A + B)}\, sin(2\pi my) \tag{8}$$

Una vez definidas las funciones que hacen los papeles de fuente y de solución, el método utilizado del *DeepONet* requiere definir las condiciones iniciales en estos casos de ecuaciones diferenciales, pues no hay otra forma de que la red las conozca. Es por ello por lo que se toman familias de funciones $\{u(x), s(y)\}$ que satisfagan las condiciones que se dan. En particular, en este trabajo se escogen por comodidad las condiciones iniciales nulas $s(0) = s'(0) = 0$.

La introducción de estas condiciones se realiza a través de los coeficientes de expansión de la solución $\{s_l, t_m\}$. Partiendo de las ecuaciones de las funciones u y s en (6):

$$s(0) = \sum_{l=0}^{N_s} s_l = s_0 + \sum_{l=1}^{N_s} s_l = 0 \quad \Longrightarrow \quad s_0 = -\sum_{l=1}^{N_s} s_l \tag{9}$$

Para calcular los coeficientes iniciales de los sumandos sinusoidales, se realiza la derivada de la expresión término a término:

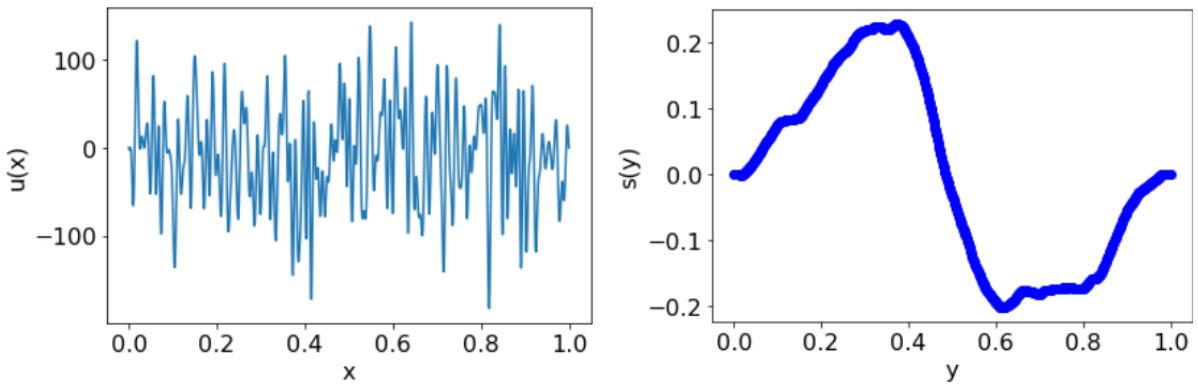
$$s'(y) = -2\pi \sum_{l=0}^{N_s} l\ s_l\ sin(2\pi l y) + 2\pi \sum_{l=0}^{N_s} m\ t_m\ cos(2\pi m y) \tag{10}$$

$$s'(0) = 0 \quad \Longrightarrow \quad t_0 = 0 \quad \wedge \quad t_1 = -\sum_{l=2}^{N_s} m\ t_m \tag{11}$$

El conjunto de estos coeficientes $\{s_0, t_0, t_1\}$ impone las condiciones iniciales, que juegan un papel fundamental en el problema planteado. Este es el caso general, pero para problemas concretos como los que se presentan en los apartados 4.5 y 4.6, habrá que estudiar con detenimiento la asignación de estos coeficientes.

4.3. Comprobación de los datos

Una vez generados los datos, se puede comprobar la forma que toman las funciones que se van a utilizar en la red. También conviene recuperar la función $u(x)$ a partir de la $s(y)$ mediante un método de aproximación numérico del estilo de *Runge-Kutta*, diferencias finitas... En las siguientes gráficas se puede ver un ejemplo de función fuente $u(x)$, combinación lineal de 100 sumandos sinusoidales y otros 100 cosenoidales, junto a su solución $s(y)$, que adquiere forma armónica.



(a) *Función $u(x)$.* (b) *Función $s(y)$.*

Figura 5: *Ejemplo de función fuente $u(x)$ y su solución $s(y)$ para 1000 puntos en el intervalo* $[0, 1)$.

En cuanto a la recuperación de la fuente a partir de la solución, se optó por utilizar el método de las diferencias finitas (MDF) al estar tratando con una ecuación en derivadas parciales de segundo orden. Lo que se hace es aproximar esta derivada segunda por su aproximación:

$$\frac{d^2 s}{dy^2}(y_i) = \frac{s(y_{i+1}) - 2\,s(y_i) + s(y_{i-1})}{h^2} \tag{12}$$

En este caso, se ha elegido el valor de h como $h = y_{i+1} - y_i$, de manera que $u_{FDM}(x)$ es la fuente sustituyendo la derivada segunda por la aproximación que aparece en la ecuación (12). En la siguiente figura se comprueba la concordancia entre la función $u(x)$ generada ($u(x)$ en negro) y la aproximada ($u(x)$ MDF en rojo).

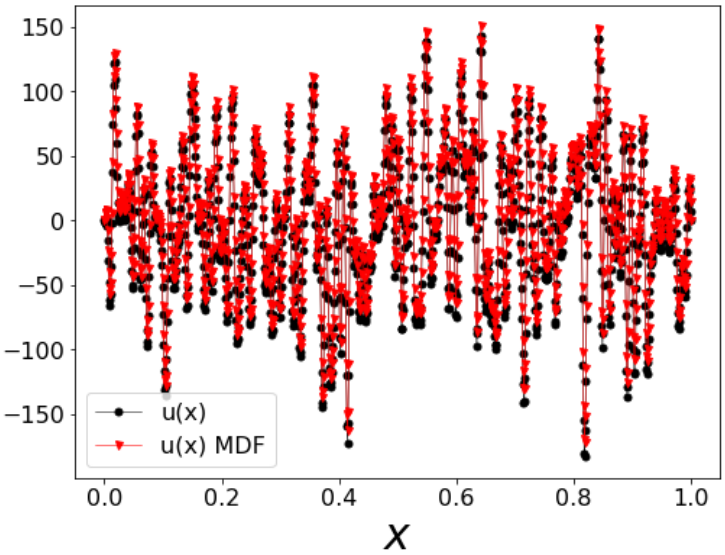


Figura 6: *Función $u(x)$ recuperada a partir de la función $s(y)$ de la Figura (5b) mediante el método de las diferencias finitas.*

4.4. Entrenamiento y predicción

Una vez generados los datos, estos se deben separar en dos conjuntos que se han introducido como *training* y *validation set*. En este caso, el primero está formado por el 90 % del total de datos, mientras que el 10 % restante está reservado para la validación. La elección de los hiperparámetros y la forma final de la red se ha hecho después de probar con distintas configuraciones y ver con cuál salían mejores resultados.

mini-batch	*epochs*	*learning rate*
32	50	0,01

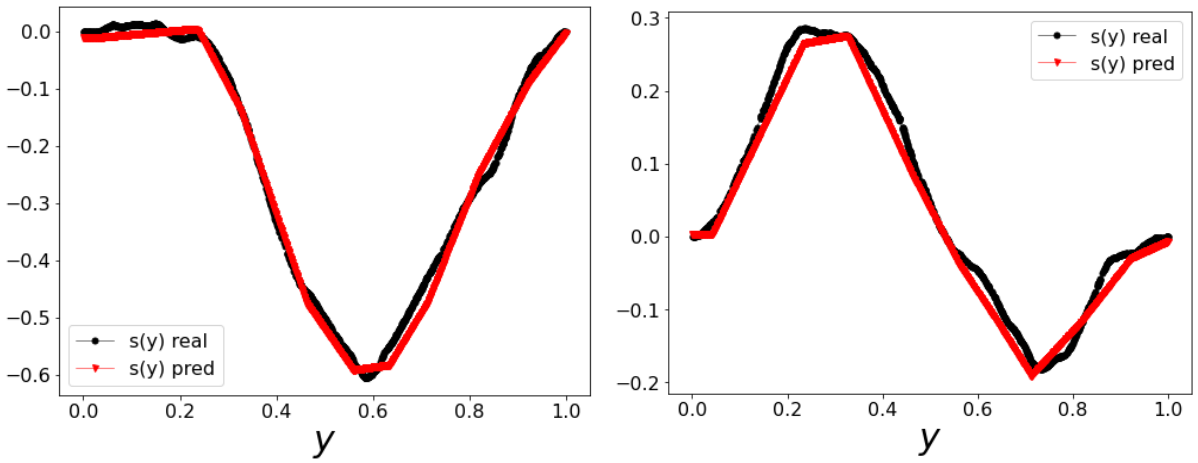
Tabla 1: *Valores utilizados para los hiperparámetros más importantes.*

En cuanto a la estructura de la red neuronal, se utilizan dos redes para la entrada como se había adelantado, cada una de ellas con una capa intermedia de 40 neuronas, y la función de activación *ReLU*. Las salidas de estas redes se concatenan mediante una capa de tipo *Dot*, la cual realiza el producto escalar entre las muestras de dos tensores [8].

Por otro lado, a la hora de compilar el modelo se utiliza el optimizador *rmsprop*, que considera solo los gradientes más recientes en lugar de considerar un acumulado de estos. Además, para valorar la actuación de dicho modelo, se emplea como función de pérdidas y métrica la ya comentada *mean squared error*.

El entrenamiento de la red fue realizado en un *cluster* de ordenadores de la Universidad de Zaragoza. De esta manera, se pudo generar una amplia base de datos con 50000 funciones para entrenar el modelo de manera más eficiente. En todos los casos se toman como valores de A y B en la expresión (5) del operador, $A = 1$ y $B = -1$.

A continuación, se pueden ver cuatro ejemplos de predicciones realizadas por dicha red neuronal. Se muestra en negro la función $s(y)$ generada y en rojo la predicción que hace la red para ella. En general, se aprecia cómo la red predice los distintos cambios y picos que se dan para la función aunque no llegue a reproducirla a la perfección. El error cuadrático medio del conjunto de todas las funciones es muy pequeño, tomando un valor de $m.s.e. = 0{,}00065$.



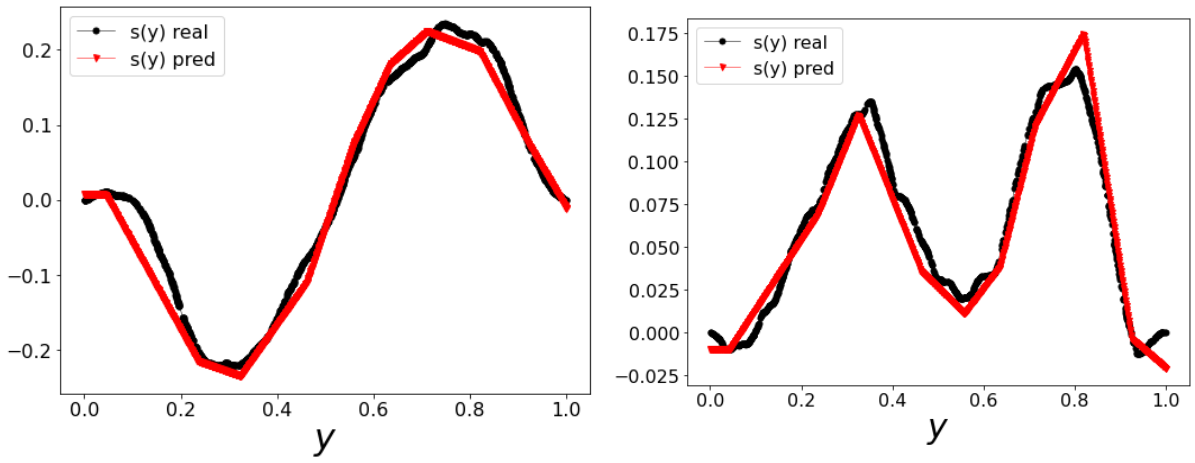


Figura 8: *Distintas predicciones realizadas por la red neuronal: las más precisas se dan para soluciones sencillas del operador, mientras que las funciones con más variaciones son reproducidas de manera más aproximada.*

En la siguientes dos gráficas, se muestra la comparativa entre los valores reales de $G(u)(y)$ y los predichos por la red para 100 funciones, cada una evaluada en un punto aleatorio de su dominio:

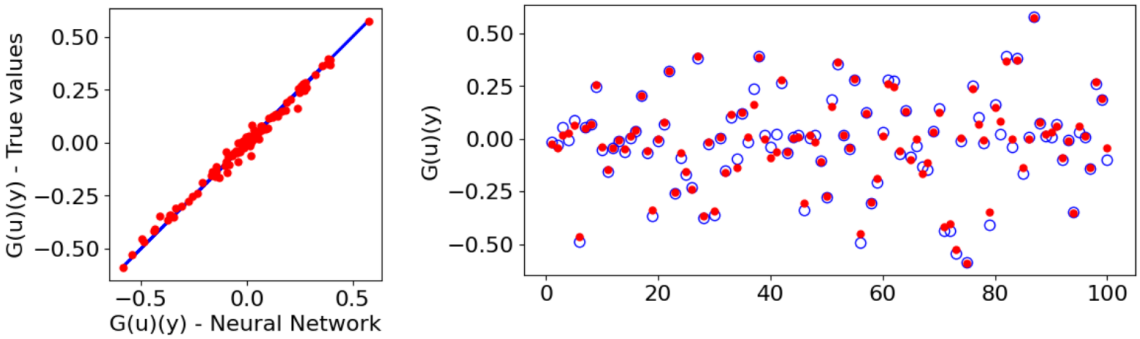


Figura 9: *Comparativa entre los valores reales (en azul) y los predichos (en rojo) de $G(u)(y)$ para un conjunto de 100 funciones en un punto de su dominio elegido aleatoriamente para cada una de ellas. A la izquierda, se observa que los valores reales forman una recta perfecta que pasa por el $(0, 0)$ y tiene pendiente 1, mientras que los valores predichos tienen cierta dispersión en torno a ella ya que no son exactamente iguales a los reales, como se puede comprobar en la gráfica de la derecha.*

4.5. Aplicación a fuentes de origen físico

Una vez estudiado el caso general del operador del oscilador armónico y entrenada la red para funciones aleatorias, parece adecuado aprovechar estos resultados para probar con distintas fuentes que tienen aplicación en diversas áreas de la Física. En particular, se presentan los casos de la función escalón y de la delta de Dirac como ejemplos válidos

para la red entrenada, mientras que se reserva el caso de la fuente triangular para mostrar alguna de las debilidades del problema que se ha resuelto.

4.5.1. Función escalón como fuente

La función escalón de Heaviside $H(x)$ es una función discontinua cuyo valor es 1 para el argumento positivo y 0 para el resto del intervalo. En problemas de física, es común encontrar problemas que corresponden a estados de sí o no (activo o inactivo). Este es el caso por ejemplo de una fuerza que actúa sobre un sistema mecánico o de una tensión eléctrica aplicada a un circuito, situaciones que podrían ser controladas por un interruptor para distinguir momento de aplicación y de no aplicación. También cobra especial importancia en ingeniería de procesamiento de señales, al representar una señal que se enciende en un instante determinado y queda encendida indefinidamente.

En el problema estudiado, la función está restringida al intervalo $[0, 1)$ y es por ello por lo que se centra en el punto medio $x = 0{,}5$. Además, se reescalan adecuadamente los coeficientes fuera del intervalo $(-10, 10)$ por un factor 10^4, ya que en caso contrario aparecen valores de la solución $s(y)$ de orden de magnitud 10^{-3}, que pueden causar problemas en la red. Por tanto, se elige como función fuente $u(x)$ a la siguiente:

$$\tilde{H}(x-0{,}5) = \begin{cases} -5000 & \text{si } x < 0{,}5 \\ 5000 & \text{si } x \geq 0{,}5 \end{cases}$$

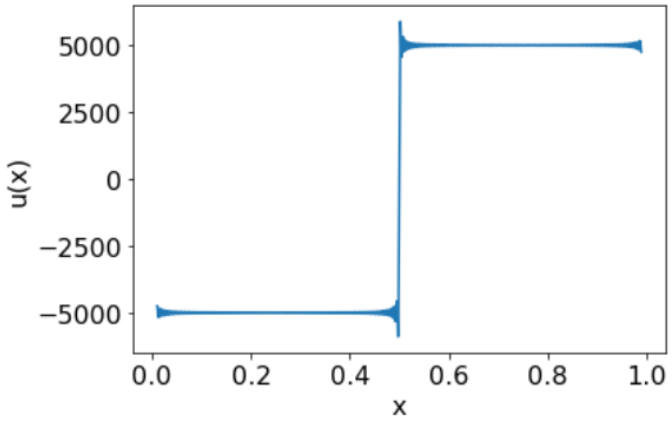


Figura 10: *Función escalón reescalada y centrada en* $x = 0{,}5$ *truncando la base en 200 términos.*

Se puede comprobar que la forma funcional es sencilla, simplemente una función a trozos con un valor constante para cada intervalo. Sin embargo, a la hora de introducirla en la red, conviene representarla en la base utilizada hasta el momento. La correspondiente serie de Fourier de este escalón solo tiene términos sinusoidales al ser una función impar, y se han elegido 200 de ellos como en las funciones de entrenamiento. Tras imponer las condiciones de contorno $s(0) = s'(0) = 0$, se obtiene la siguiente solución junto a la predicción de la red:

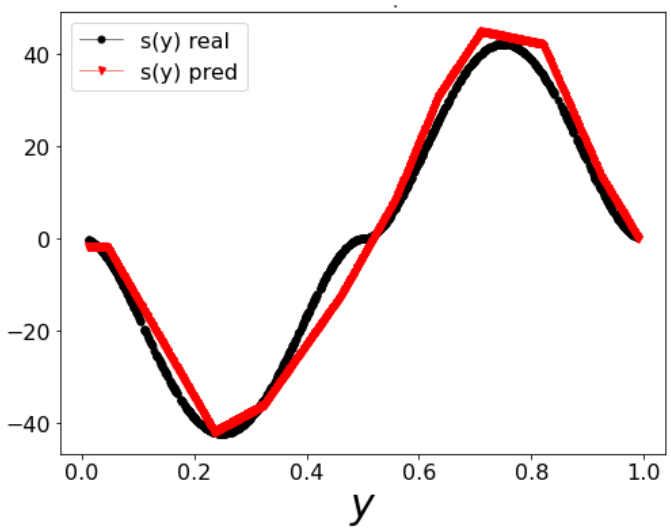


Figura 11: *Predicción realizada por la red para el caso de una fuente de tipo escalón como la de la Figura (10).*

A la vista de esta gráfica, se puede concluir que la predicción de la red para la solución del escalón es bastante precisa, excepto quizás en la parte central en torno a $x = 0{,}5$. Aunque en la Figura (10) aparece unido, este punto es de discontinuidad de salto finito y puede ser que la red no distinga esta singularidad.

4.5.2. Delta de Dirac como fuente

La delta de Dirac es un objeto matemático que admite dos interpretaciones: la del matemático puro que la considera como distribución, y la del físico y el ingeniero que la trata como función. Según el propio Dirac, este objeto se introduce "... para considerar cantidades que involucran cierta clase de infinitos. Para lograr una notación precisa en el manejo de estos infinitos, se introduce una cantidad $\delta(x)$ dependiendo de un parámetro x, que satisface las condiciones:

$$\int_{-\infty}^{\infty} \delta(x)\,dx = 1, \qquad \delta(x) = 0 \;\; para \;\; x \neq 0 \;\; " \tag{13}$$

También reconoce que "no es una función de x de acuerdo con la definición matemática usual, la que requiere que una función tenga un valor definido para cada punto de su dominio..." [9] En el sentido de distribuciones[3], define un funcional en forma de integral sobre un cierto espacio de funciones y puede estar centrada en un punto $x = a$ como $\delta_a(x) = \delta(x - a)$, de manera que tiende a infinito cuando $x = a$ y es nula para cualquier otro valor (siendo la integral a todo el espacio igual a 1).

[3]Se llama distribución sobre un espacio Ω a toda aplicación lineal y continua $T : \mathcal{D}(\Omega) \longrightarrow \mathbb{R}$ que asocia a cada función de prueba ϕ el valor $T(\phi) =< T, \phi >$. De esta manera, la delta de Dirac es la aplicación (funcional) lineal y continua $< \delta, \phi >= \phi(0)$.

En este sentido, la delta de Dirac recibe en ocasiones el nombre de función impulso y permite definir la derivada generalizada de funciones discontinuas. Este último hecho la relaciona con la función escalón, siendo la delta la derivada distribucional del escalón. De nuevo, su importancia viene dada por el hecho de que muchos problemas físicos y de ingeniería involucran sistemas mecánicos o eléctricos actuados por fuentes discontinuas o de impulso.

En este caso, para introducirla en la red se toman 200 términos de la serie de Fourier, que ahora son cosenoidales al ser una función par.

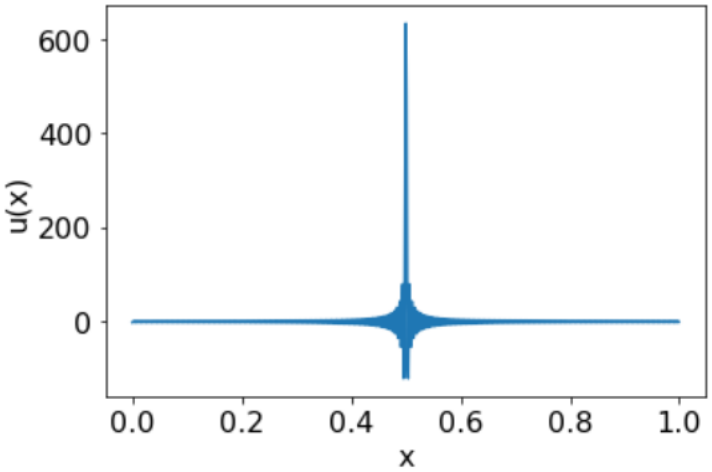


Figura 12: *Delta de Dirac centrada en* $x = 0{,}5$ *truncando la base en 200 términos.*

La solución a esta fuente se puede obtener analíticamente haciendo uso de la transformada de Laplace. Partiendo de la ecuación del operador (5):

$$\frac{d^2 s(x)}{dx^2} + s(x) = u(x) = \delta(x - 0{,}5)$$

Se puede tomar la transformada de Laplace $\mathcal{L}(s(x)) = S(t)$ de la expresión anterior:

$$(t^2 + 1)\, S(t) = e^{-0{,}5 \cdot t} \implies \mathcal{L}^{-1}(S(t)) = \tilde{H}(t - 0{,}5) \cdot sin(t - 0{,}5)$$

La solución $s(y)$ tiene la forma de un seno centrado en $y = 0{,}5$ escalado por la función $\tilde{H}$, la cual toma valores negativos para $y < 0{,}5$ y positivos para $y \geq 0{,}5$. Así, queda una función sinusoidal simétrica, con $s(0{,}5) \neq 0$ ya que se ha impuesto que $s(0) = 0$, como la que aparece en la Figura (13):

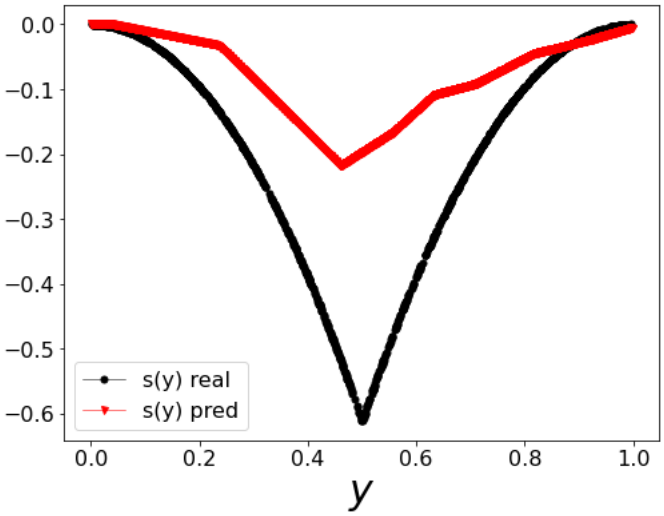


Figura 13: *Predicción realizada por la red para el caso de una fuente de tipo delta de Dirac como la de la Figura (12).*

Se observa que la predicción de la red intuye la forma de la solución pero no es capaz de replicarla exactamente. Este hecho puede tener que ver con las peculiaridades de la fuente, que no es una función por lo que se ha comentado, así como el punto tan marcado de no derivabilidad de la solución en $y = 0{,}5$.

4.6. Debilidades de la red entrenada

Como último caso, se va a analizar una fuente de tipo triangular con condición de contorno $s(0) = 0$, pero $s'(0) \neq 0$. Una onda triangular representa un tipo de señal periódica con velocidades de subida y bajada constantes, lo que se traduce en tiempos de subida y bajada iguales. Se caracteriza porque tiene un contenido en armónicos muy bajo, lo que se puede ver por su parecido a una onda sinusoidal. Además, presenta diversas aplicaciones en distintos campos, destacando el de la electrónica. Por ejemplo, se emplean en la creación de osciladores controlados por tensión, debido a que la relación lineal que presentan entre la amplitud y el tiempo supone una gran ventaja, al tener que comparar su nivel con la tensión de control. Asimismo, se pueden generar señales sinusoidales al conformar señales triangulares con redes de resistencias y diodos [10].

Para estudiar el comportamiento de la red ante esta fuente, se debe escribir de nuevo en la base utilizada hasta ahora. En este caso, la serie de Fourier solo presenta términos sinusoidales impares que se han reescalado por un factor 100, llegando a una función de entrada como la siguiente:

$$T(x) = \sum_{n=0}^{2\,N_{sum}} \frac{(-1)^n}{(2n+1)^2}\, sin\left((2n+1)\pi x\right)$$

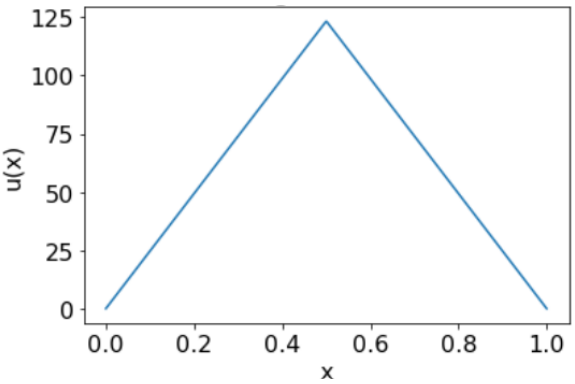


Figura 14: *Función triangular reescalada y truncando la base en 200 términos.*

Este ejemplo sirve para probar los límites de validez de las predicciones de la red. Si el caso de la delta de Dirac ya presentaba una predicción menos precisa, con la fuente triangular se puede apreciar la importancia de las condiciones de contorno. La respuesta $s(y)$ a esta fuente cumple que $s(0) = 0$, mientras que la primera derivada en el origen es $s'(0) \neq 0$.

En la siguiente figura se puede comprobar que la red es incapaz de predecir siquiera la forma ondulatoria de la respuesta:

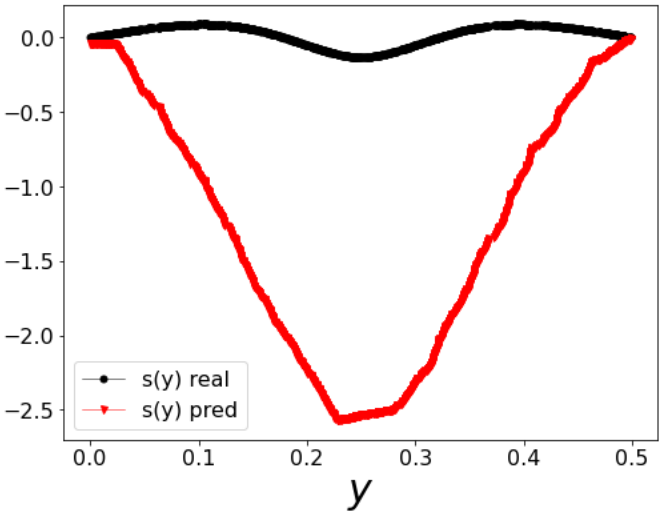


Figura 15: *Predicción realizada por la red para el caso de una fuente triangular como la de la Figura (14).*

Con este ejemplo, se puede comprobar que la red entrenada tiene una serie de limitaciones. Un requisito fundamental es el de las condiciones iniciales, ya que la red solo podrá reproducir soluciones cuyas condiciones sean las mismas a las de las funciones empleadas en el entrenamiento. Dicho de otra manera, el operador diferencial solo es capaz de aprender la aplicación entre las funciones $u(x)$ y un subespacio de aquellas soluciones que cumplen las condiciones iniciales de partida.

5. Conclusiones

En este trabajo se ha planteado un problema general que puede ser de gran utilidad para multitud de problemas físicos y matemáticos. Una vez concluido, se puede afirmar que las redes profundas son capaces de aprender y predecir la actuación de un operador entre dos espacios de funciones que pueden representar diferentes estados físicos.

Aunque el problema se ha realizado tomando un operador relativamente sencillo, la potencia de las *DeepONet* llega incluso al aprendizaje de operadores implícitos o a la predicción de la evolución de un sistema caracterizado por ecuaciones diferenciales estocásticas. Además de esto, hay otras tareas que podrían haberse probado, como por ejemplo la inclusión de las condiciones iniciales en una rama de entrada paralela a las otras dos que se han utilizado. De esta forma, se podría predecir la solución completa de una función junto a las condiciones iniciales correspondientes. Estos avances en la mejora de las *DeepONet* podrían hacer de ellas una herramienta muy útil a la hora de resolver problemas provenientes de ramas como la física o la ingeniería.

Por otro lado, se han examinado algunas de las limitaciones del método. Entre ellas destaca la concordancia de las condiciones iniciales de la función a estudiar con las de las funciones empleadas en el entrenamiento. También se han observado debilidades a la hora de predecir soluciones de funciones fuente que presentan un número finito de discontinuidades. Este es un problema pendiente que requiere una investigación más exhaustiva.

En definitiva, a lo largo de este trabajo se ha podido comprobar el potencial de la IA, y en particular, de las redes neuronales, para resolver problemas que tienen un enfoque mucho más abstracto y técnico que los que se suelen plantear, como son la clasificación o el reconocimiento de imágenes. En este sentido, todavía queda mucho por hacer e investigar, y ejemplo de ello es que se están buscando redes neuronales que sean capaces de demostrar resultados y teoremas matemáticos.

6. Bibliografía

[1] Nielsen, M.A. (2015) *Neural Networks and Deep Learning,* San Francsico, CA.

[2] Pérez Martínez, H., Martín Moreno, L. (2020) *Inteligencia Artificial aplicada a transiciones de fase,* TFG, Grado de Física, Universidad de Zaragoza, pp. 2-3

[3] Martínez Aznar, M., Gutiérrez Rodrigo, S., Martín Moreno, L. (2021) *Inteligencia Artificial aplicada a la Física Estadística,* TFG, Grado de Física, Universidad de Zaragoza, p. 5

[4] Martín Moreno, L., Zueco Lainez, D., Gutiérrez Rodrigo, S. (2021) *Curso de Introducción a la Inteligencia Artificial,* Capítulo 2: Redes Neuronales, QMAD, Universidad de Zaragoza

[5] Brownlee J. (25/01/2019) *Understand the Impact of Learning Rate on Neural Network Performance,* Machine Learning Mastery

[6] Lu, L., Jin, P., Pang, G. *et al* (2021) *Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators,* Nat Mach Intell 3, pp. 218-229

[7] Cybenko, G. (1989) *Approximation by Superposition of a Sigmoidal Function,* Mathematics of Control, Signals and Systems, Springer-Verlag New York Inc.,pp. 303-314

[8] TensorFlow (04/09/2022) *Información sobre capas: tf.keras.layers.*

[9] George, K., Imaz Jahnke, C. (1995) *La delta de Dirac como función,* Educación Matemática, Vol. 7 - No. 3, p. 49

[10] Portal de arquitectura Arqhys.com (09/09/2022) *Equipo de redacción profesional. (2012, 12). Ondas triangulares,* Arqhys Contenidos.