Universidad de Zaragoza

Facultad de Ciencias

Trabajo de Fin de Grado en Física

Physics-Informed Neural Networks

para resolver ecuaciones diferenciales de primer y segundo orden

---

Autor:

Luis Medrano Navarro

Directores:

Sergio Gutiérrez Rodrigo

Luis Martín Moreno

Septiembre de 2022

Agradecimientos

Me gustaría agradecer a mis directores, Sergio Gutiérrez y Luis Martín, por su apoyo y dedicación a lo largo de estos meses. Gracias a ellos la realización de este trabajo ha sido una experiencia muy enriquecedora. Quisiera agradecerles también el haberme enseñado los conceptos previos para abordar este trabajo en un campo tan novedoso e interesante.

Índice

Lista de acrónimos

IA: Inteligencia Artificial.

ML: *Machine Learning*, Aprendizaje Automático.

PINN: *Physics-Informed Neural Network*, Red Neuronal con Información Física.

PCA: *Principal Component Analysis*, Análisis de Componente Principal.

CNN: *Convolutional Neural Network*, Red Neuronal Convolucional.

GPU: *Graphics Processing Unit*, Unidad de Procesamiento Gráfico.

TPU: *Tensor Processing Unit*, Unidad de Procesamiento Tensorial.

Adam: *Adaptive Moment Estimation*, Estimación de Momento Adaptativo.

EDO: Ecuación Diferencial Ordinaria.

EDP: Ecuación en Derivadas Parciales.

AD: *Automatic Differentation*, Diferenciación Automática.

MSE: *Mean Square Error*, Error Cuadrático Medio.

API: *Application Programming Interface*, Interfaz de Programación de Aplicaciones.

LSTM: *Long Short-Term Memory*, Memoria a Corto Plazo.

1. Introducción

En los últimos años se ha extendido enormemente el uso de la Inteligencia Artificial (IA) y el *Machine Learning* (ML), desde el reconocimiento de imágenes, utilizado por ejemplo en *Google Maps*, hasta el procesamiento del lenguaje natural, utilizado por un gran número de traductores, pasando por el *Big Data*, utilizado por las grandes compañías para diseñar sus estrategias de marketing o hacernos sugerencias según nuestros gustos. A grandes rasgos, el objetivo de estas técnicas es reconocer patrones en datos para poder extraer información de otros datos no vistos previamente, permitiendo así automatizar muchas tareas. Por tanto, no es de extrañar que haya nacido todo un mundo de aplicaciones de la IA en la ciencia.

Desde hace mucho tiempo se ha estado conjeturando sobre la posibilidad de crear máquinas con inteligencia y consciencia. Son numerosas las referencias a la IA que encontramos en la ficción. Todo esto empezó a hacerse realidad con las primeras redes neuronales, que tienen su origen en el Perceptrón [1], desarrollado en 1958 por Frank Rosenblatt. No obstante, no cobraron relevancia hasta la década de los 80s y los 90s, debido a que al principio no existían ni técnicas eficientes de entrenamiento, ni ordenadores suficientemente potentes. En 1986 se presenta un artículo revolucionario introduciendo el algoritmo de *Backpropagation* [2], que dio lugar a un *boom* en el desarrollo de la IA al acelerar el aprendizaje. Sin embargo, ha sido en la última década, con el desarrollo del *Deep Learning*, cuando se ha vivido la gran revolución de la IA con la gran cantidad de datos accesibles.

Dentro del ML suele distinguirse entre aprendizaje supervisado y aprendizaje no supervisado. En el supervisado, los algoritmos aprenden a partir de datos de entrenamiento previamente etiquetados. El no supervisado tiene la ventaja de que no necesita etiquetar previamente un conjunto de datos. Ejemplos de este último son las tareas de agrupación/*clustering*, o las *Physics-Informed Neural Networks* (PINNs), objeto de este trabajo.

Algunas de las aplicaciones de la IA en la Física son [3]:

1. *Principal Component Analysis* (PCA) y reducción de dimensión. Esta técnica consiste en encontrar transformaciones de las variables de un sistema físico que permitan explicar el sistema con un menor número de grados de libertad. Podemos ver ejemplos de esta aplicación en la Física Estadística.

2. Regresión y clasificación. Utilizando redes neuronales se pueden construir modelos de ajuste de funciones con datos experimentales o clasificar datos, por ejemplo, imágenes.

3. Reconocimiento de imágenes. Estas técnicas potenciadas con la introducción de las Redes Neuronales Convolucionales (CNN) se han utilizado, por ejemplo, en Física de Partículas o en Astronomía. Pueden utilizarse para clasificar partículas en colisiones (utilizado en el LHC) o para analizar imágenes de telescopios. También

son utilizadas en microscopia y medicina, pudiendo entrenar redes neuronales para reconocer tumores.

4. *Clustering.* Estos algoritmos permiten agrupar objetos cercanos en grupos. Esto se ha utilizado, por ejemplo, en astronomía para clasificar nuevas galaxias.

Existen muchos otros ejemplos de aplicación de la IA en Física, pero quizás, uno de los más interesantes y objeto de este trabajo, son las *Physics-Informed Neural Networks* (PINNs). Este tipo de redes neuronales permite resolver todo tipo de ecuaciones diferenciales, ya sean ordinarias o parciales, con una o varias variables, lineales o no-lineales, ecuaciones individuales o sistemas de ecuaciones. Estas PINNs se han utilizado, por ejemplo, para resolver la ecuación de Schrödinger[4][5], en física de fluidos para resolver las ecuaciones de Navier-Stokes [6][7] o en fotónica [8].

2. Objetivos

Los objetivos de este trabajo son realizar un acercamiento a las PINNs, programando una red neuronal capaz de resolver algunos ejemplos básicos de ecuaciones diferenciales, y proponer nuevas soluciones a algunos problemas que nos hemos encontrado.

Esta nueva técnica de resolución de ecuaciones diferenciales es especialmente interesante porque complementa y soluciona algunos de los problemas de las técnicas clásicas de análisis numérico, como Runge-Kutta o el Método de Elementos Finitos. En este último lo que se hace es discretizar el dominio en una malla y reducir el problema a un sistema de ecuaciones algebraicas. En las PINNs [9] aprovechamos la capacidad que tienen las redes neuronales para representar cualquier función, haciendo que minimice el residuo o error de la ecuación diferencial y las condiciones de contorno. Además, utilizan aprendizaje no supervisado, por lo que pueden utilizarse casi sin necesidad de intervención del usuario, pues no es necesario un proceso de preparación y etiquetado de los datos.

Las principales ventajas de las PINNs son que no necesitan de un discretizado del dominio para resolver las ecuaciones, permiten extender el rango de validez de la solución para valores en los que la red no ha sido entrenada, y en general, hacen un menor uso de memoria. Sin embargo, tienen algunas limitaciones. La principal [9] es que actualmente necesitan, en muchos casos, un tiempo de ejecución relativamente alto, mayor que las técnicas de Elementos Finitos. No obstante, el uso de GPUs y otros chips dedicados (TPUs) ha conseguido reducir enormemente estos tiempos. También es cierto que es en problemas complejos, donde con Elementos Finitos se necesita un discretizado muy fino, donde destacan las PINNs. Además, otro inconveniente es que no existe una forma rigurosa y consistente de evaluar los errores cometidos. Por otro lado, utilizando PINNs, no está garantizada la unicidad de la solución, de hecho, la red puede converger a soluciones ligeramente distintas según como inicialicemos sus parámetros. En realidad, las PINNs nos dan una aproximación a la solución real.

3. Conceptos básicos sobre Redes Neuronales

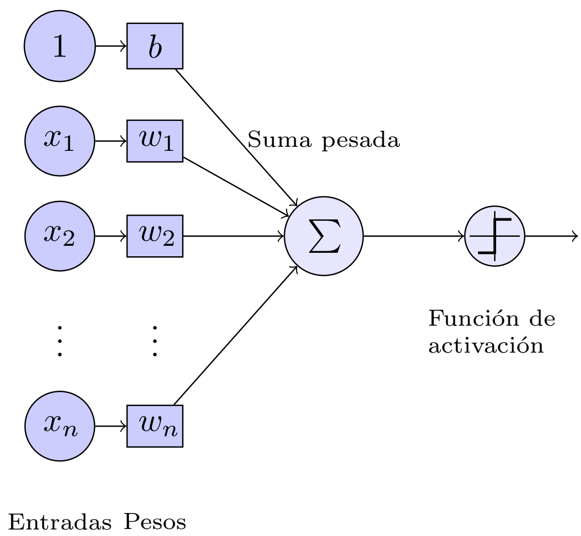
Antes de explicar en profundidad el funcionamiento de las PINNs es necesario entender el funcionamiento de una red neuronal básica [10].

3.1. Perceptrón

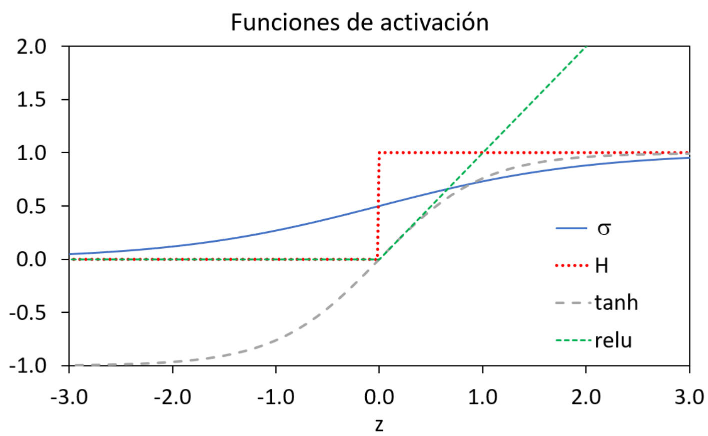
En analogía a las neuronas y al tejido cerebral descubierto por Ramón y Cajal, una red neuronal en IA es un conjunto de varias capas de neuronas matemáticas simples o perceptrones, introducidos por Frank Rosenblatt [1]. Una neurona matemática (que a partir de ahora llamaremos simplemente neurona) es una función

$$\begin{array}{ccccc} \mathbb{R}^n & \longrightarrow & \mathbb{R} & \longrightarrow & \mathbb{R} \\ \{x_n\} & \longrightarrow & z = \sum_{i=1}^{n} \omega_i x_i - b & \longrightarrow & \sigma(z) \end{array}$$

consistente en la composición de una función lineal con una no lineal. De manera que, dados un conjunto de valores de entrada $\{x_1, x_2, ..., x_n\}$, la neurona devuelve $\sigma(z)$ donde $z = \sum_{i=1}^{n} \omega_i x_i - b$ y σ es una función no lineal, denominada función de activación. A los coeficientes $\{\omega_i\}$ se les llama pesos y a b *bias* (umbral, sesgo). La forma de la función de activación determina el proceso de aprendizaje y es siempre una función creciente. Algunas de las más utilizadas son las que aparecen en la Gráfica 1. Cada una tiene distintos usos: la sigmoide para clasificación binaria, la tanh y la elu las que más se utilizan en PINNs, y la relu para regresión.



Esquema 1: *Esquema de un perceptrón (Fuente: stackexchange).*



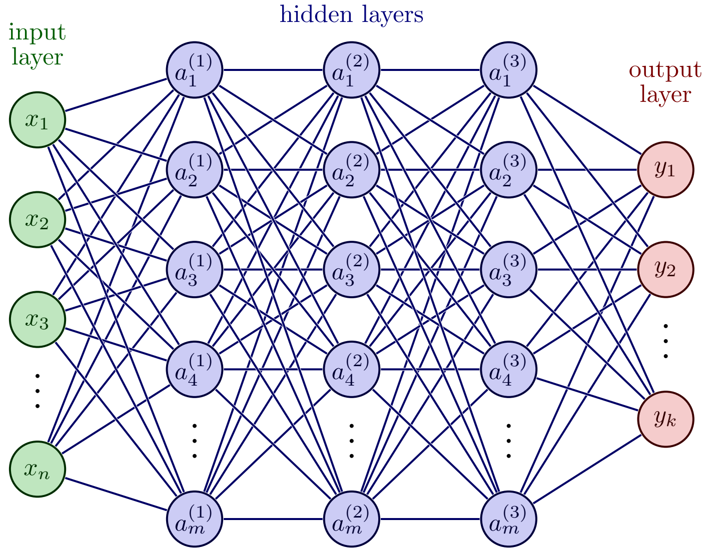
Gráfica 1: *Representación gráfica de algunas funciones de activación.*

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad H(z) = \begin{cases} 1, & \text{si } z > 0 \\ 0, & \text{si } z < 0 \end{cases} \quad \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \mathrm{relu}(z) = \begin{cases} z, & \text{si } z > 0 \\ 0, & \text{si } z < 0 \end{cases}$$

La función escalón $H(z)$ fue la primera función de activación en utilizarse. La idea es similar a la de una neurona cerebral: cuando los valores de entrada, pesados según su importancia, superan un cierto umbral ($\sum_{i=1}^{n} \omega_i x_i > b$), la neurona se activa. El resto de funciones representan este comportamiento de forma suavizada, de manera que pequeños cambios en los valores de entrada, producirán un cambio pequeño en la salida.

3.2. Red Neuronal

Una red neuronal es una concatenación de varias capas de neuronas. El orden de las capas se lee de izquierda a derecha. La primera capa es la capa de entrada (*input layer*), la última la capa de salida (*output layer*), y las capas intermedias se llaman capas ocultas (*hidden layers*).



Esquema 2: *Esquema de una red neuronal (Fuente: Izaak Neutelings).*

Una de las propiedades de las redes neuronales que vamos a usar en las PINNs es su teorema de universalidad [11] [12]. Al igual que todo circuito lógico/booleano se puede escribir sólo con puertas NAND, existe un teorema que nos dice que una red neuronal puede aproximarse uniformemente tan bien como queramos a cualquier función dada. En concreto:

Teorema 1 (Universalidad) *Sea $m^i \in \mathbb{Z}_+^d$, $i = 1, ..., s$ y sea $m = \max_i |m^i|$. Supongamos $\sigma \in C^m(\mathbb{R})$ y que σ no es un polinomio. Entonces, el espacio de redes neuronales con una sola capa oculta*

$$M(\sigma) := span\{\sigma(w \cdot x + b) : w \in \mathbb{R}^d, b \in \mathbb{R}\}$$

es denso en

$$C^{m^1,...,m^s}(\mathbb{R}^d) := \cap_{i=1}^{n} C^{m^i}(\mathbb{R}^d)$$

es decir, para toda $f \in C^{m^1,...,m^s}(\mathbb{R}^d)$, todo compacto $K \subset \mathbb{R}^d$ y todo $\epsilon > 0$, existe una red neuronal $g \in M(\sigma)$ tal que

$$\max_{x \in K} |D^k f(x) - D^k g(x)| < \epsilon$$

Una PINN no es más que una red neuronal representando una función, a la que haremos cumplir la ecuación diferencial y la condición inicial.

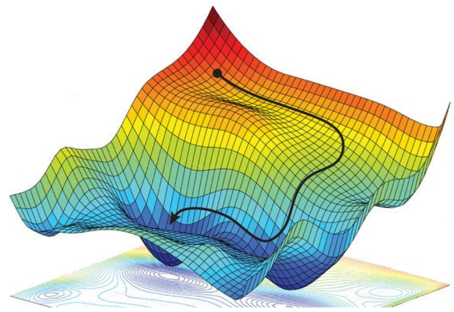
3.3. Función de coste "loss"

Cuando entrenamos de forma supervisada una red neuronal, lo que hacemos es minimizar los errores entre la salida de la red y las etiquetas correspondientes a los datos de entrada. A este error lo llamamos función de coste o *loss*, y es una función que depende de los pesos y los *bias* de todas las neuronas. Al igual que las funciones de activación, existen distintas funciones de coste. La más sencilla es considerar el error cuadrático medio entre la salida de la red $y(x)$ y los valores reales de las etiquetas e(x), para todos los datos de entrenamiento:

$$C(\omega, b) = \frac{1}{n}\sum_{x} |y(x) - e(x)|^2 \tag{1}$$

3.4. Descenso del gradiente

En el subapartado anterior hemos llegado a una función que hay que minimizar. Para hacer esta tarea se utiliza normalmente el algoritmo de descenso del gradiente. Es un algoritmo iterativo en el que, en cada paso, nos movemos (en el espacio multidimensional de pesos y *bias*) una pequeña distancia en dirección opuesta al gradiente, ya que el vector gradiente nos da la dirección de máxima variación de la función. De esta manera, en cada iteración actualizamos los valores de cada peso ω_i y cada *bias* b_i de las neuronas según las siguientes ecuaciones:



Gráfica 2: *Descenso del gradiente (Fuente: Ángel Sánchez Ruiz).*

$$\omega_i^{n+1} = w_i^n - \eta\frac{\partial C}{\partial \omega_i} \tag{2}$$

$$b_i^{n+1} = b_i^n - \eta\frac{\partial C}{\partial b_i} \tag{3}$$

donde el parámetro η se conoce como tasa de aprendizaje o *learning rate*. Cuanto más grande es este parámetro, más rápido se entrena la red, pero mayores son los errores cometidos. No obstante, ¿mayores respecto a qué? Son mayores en el sentido de que debemos imaginar este proceso como descender una montaña hasta el punto más bajo. Si diéramos pasos muy grandes, sería difícil encontrar ese punto, porque cuando estamos muy cerca y damos el siguiente paso, nos pasamos de largo y volvemos a subir ligeramente hacia arriba. Para mejorar las predicciones de la red, podemos reducir este parámetro a costa de aumentar el tiempo de ejecución. Además, esto también tiene el inconveniente de que el algoritmo puede quedar atrapado en mínimos relativos de la función, y no en el mínimo global.

Por otro lado, no siempre se utilizan todos los puntos de entrenamiento, sino que se eligen algunos al azar, formando subconjuntos de entrenamiento llamados *batches*. Otro parámetro del modelo es el tamaño de cada *batch*, llamado *batch size*.

La minimización de la función de coste es lo que se conoce como el entrenamiento de la red. Las derivadas parciales hay que evaluarlas en los puntos de entrenamiento. Cada paso de descenso en todos los puntos de entrenamiento se llama época de entrenamiento. Lo que suele hacerse es repetir este proceso durante varias épocas, es decir, el número de épocas es el número de veces que la red ve un dato de entrenamiento o aprende de este dato.

Además del descenso del gradiente, existen otros métodos para minimizar la función *loss*. Uno de los más usados es *Adam* (*Adaptive Moment Estimation*) [13], que es una versión mejorada del descenso del gradiente.

3.5. Backpropagation

El método del descenso del gradiente permite minimizar la función de coste, pero no nos dice cómo calcular el gradiente. Para calcular estas derivadas parciales se utiliza el algoritmo de *Backpropagation*. Este algoritmo [2] permitió la primera gran revolución de la IA. En principio, podríamos calcular las derivadas parciales de forma numérica:

$$\frac{\partial y}{\partial x_i} = \lim_{\Delta x_i \to 0} \frac{y(x_1, ..., x_i + \Delta x_i, ..., x_n) - y(x_1, ..., x_i, ..., x_n)}{\Delta x_i} \tag{4}$$

Sin embargo, este proceso requiere de un gran número de operaciones que hacen que el aprendizaje sea inasumiblemente lento. El nuevo algoritmo de *Backpropagation* calcula las derivadas utilizando la regla de la cadena y haciendo una actualización en dos pasos:

1. Calculamos hacia adelante, de izquierda a derecha, las activaciones de todas las neuronas.
2. Calculamos hacia atrás, de derecha a izquierda, las derivadas parciales utilizando la regla de la cadena.

Consideremos una red de L capas de neuronas, donde cada capa la representamos como un vector de neuronas $A^l = (a_1^l, ..., a_{N_l}^l)$:

$$\begin{cases} \text{Capa de entrada: } A^0(x) = x \in \mathbb{R}^{d_{in}} \\ \text{Capas ocultas: } A^l(x) = \sigma(W^l A^{l-1}(x) + b^l) \in \mathbb{R}^{N_l}, \text{ para } 1 \leqslant l \leqslant L-1 \\ \text{Capa de salida: } A^L(x) = \sigma(W^L A^{L-1} + b^L) \in \mathbb{R}^{d_{out}} \end{cases} \tag{5}$$

donde $W^l \in \mathbb{R}^{N_l \times N_{l-1}}$ es la matriz de pesos de la capa l y $b^l \in \mathbb{R}^{N_l}$ el vector de *bias*. Entonces, se define el error de las neuronas en la última capa como

$$\delta_j^L := \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \Leftrightarrow \delta^L = \nabla_a C \odot \sigma'(z^L) \tag{6}$$

donde $\odot$ es el producto de Hadamard (componente a componente) y $z^l = W^l A^{l-1} + b^l$. Haciendo pasos hacia atrás, los errores en el resto de capas son:

$$\delta^l := \frac{\partial C}{\partial z^l} = ((W^{l+1})^\intercal \delta^{l+1}) \odot \sigma'(z^l) \tag{7}$$

A partir de estas dos últimas ecuaciones es casi inmediato calcular las derivadas parciales que aparecen en (2) y (3):

$$\frac{\partial C}{\partial b_j^l} = \underbrace{\frac{\partial C}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}}_{\delta_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \cdot 1 = \delta_j^l \qquad\qquad \frac{\partial C}{\partial \omega_{ij}^l} = \underbrace{\frac{\partial C}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_i^l}}_{\delta_i^l}\frac{\partial z_i^l}{\partial \omega_{ij}^l} = \delta_i^l a_j^{l-1} \tag{8}$$

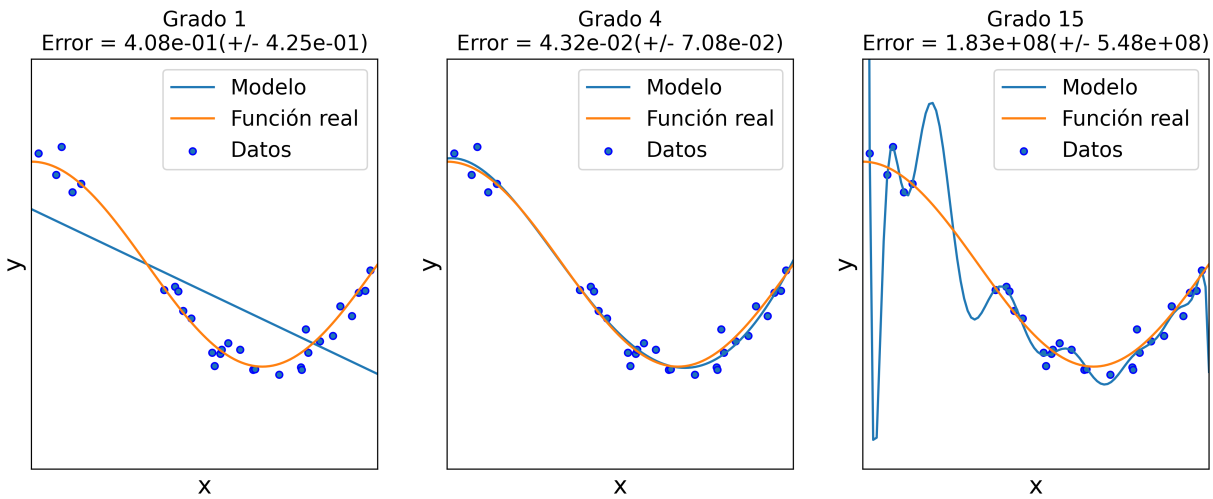
Para más detalles sobre este algoritmo se recomienda la lectura de [10], de la que se ha seguido la notación.

3.6. Overfitting

Por último, comentaremos brevemente uno de los problemas más importantes que puede ocurrir cuando entrenamos una red neuronal. Cuando hacemos un ajuste (regresión) de una función a unos valores experimentales, tenemos que elegir el número de grados de libertad de nuestro modelo. Si elegimos un número insuficiente, incurrimos en *underfitting*, el modelo no se ajustará bien a los datos, no logrará replicar la relación subyacente en ellos. Si elegimos un número demasiado alto se producirá *overfitting*, el modelo se ajustará muy bien a los datos, pero no reflejará correctamente la relación subyacente en ellos y, seguramente, producirá grandes errores cuando tratemos de explicar nuevos datos.

En el caso de redes neuronales, el número de neuronas (y por tanto de parámetros) de la red es el equivalente a los grados de libertad en regresión. Si elegimos un número muy grande de parámetros aparece el *overfitting*, el modelo se ajusta muy bien a los datos de ajuste/entrenamiento, pero no será capaz de generalizar el comportamiento de otros datos nuevos. En la Gráfica 3 mostramos los tres tipos de comportamientos posibles para un ajuste polinómico. En el primero, de grado 1, observamos *underfitting*: el modelo lineal no consigue reproducir el comportamiento de los datos. El segundo, de grado 4, consigue modelizar muy bien los datos. El tercero, con overfitting, aunque pasa perfectamente por los puntos de ajuste, no consigue generalizar bien el resto de puntos.

El *underfitting* es fácil de detectar (*loss* o error muy alto) y corregir (aumentar el número de parámetros). Para evitar el *overfitting*, existen numerosas técnicas. En este trabajo hemos empleado el *Dropout*, que consiste en eliminar algunas neuronas elegidas



Gráfica 3: *3 ajustes polinómicos en los que se observan underfitting y overfitting (Fuente: [14]).*

al azar en cada paso de entrenamiento. El porcentaje de neuronas que se eliminan es un parámetro que tenemos que elegir, normalmente un 50 %. Como se explica en [15], de esta manera la red aprende la información más robusta y no el ruido o las particularidades de ciertos datos. También puede interpretarse como en [16]: el *Dropout* es equivalente a entrenar varias redes neuronales distintas y después tomar un promedio, pero de una forma mucho más rápida.

Los parámetros que podemos modificar manualmente en la red, como el número de capas de neuronas, el número de neuronas por cada, la tasa de aprendizaje, el número de épocas de entrenamiento... se denominan hiperparámetros. Tendremos que jugar constantemente con estos valores para encontrar una red que funcione correctamente. El problema es que esto hay que hacerlo manual y heurísticamente, no existen reglas claras sobre como hacer esto. Es trabajo del programador de la red encontrar una combinación de hiperparámetros que proporcione resultados satisfactorios.

4. Physics-Informed Neural Networks (PINNs)

Varias han sido las propuestas realizadas para resolver ecuaciones diferenciales con redes neuronales. Destaca un primer artículo [17] de 1998, que trabaja de forma ligeramente distinta de la actual, desarrollada sobre todo a partir de 2020 [9].

4.1. Primeras PINNs en 1998

En el artículo pionero [17] de 1998 se trabaja con la condición inicial y la ecuación por separado. Partimos de una ecuación diferencial de la forma

$$G(\vec{x}, \psi(\vec{x}), \nabla\psi(\vec{x}), \nabla^2\psi(\vec{x})) = 0, \quad \vec{x} \in D \tag{9}$$

sujeta a unas ciertas condiciones de contorno, donde $\vec{x} = (x_1, ..., x_n) \in \mathbb{R}^n$ y $D \subset \mathbb{R}^n$ es el dominio en el buscamos la solución $\psi(\vec{x})$. Para resolver el problema, elegimos algunos puntos de D y su frontera S, definiendo los subconjuntos de entrenamiento $\hat{D}$ y $\hat{S}$. Entonces, el problema se convierte en un sistema de ecuaciones:

$$G(\vec{x}_i, \psi(\vec{x}_i), \nabla\psi(\vec{x}_i), \nabla^2\psi(\vec{x}_i)) = 0, \quad \vec{x}_i \in \hat{D} \tag{10}$$

sujetas a las condiciones de contorno. Para resolverlo, proponemos una función de test $\psi_t(\vec{x}, \vec{p})$, que depende de un conjunto de parámetros $\vec{p}$ que buscaremos de manera que se minimice:

$$\min_{\vec{p}} \sum_{\vec{x}_i \in \hat{D}} [G(\vec{x}_i, \psi(\vec{x}_i), \nabla\psi(\vec{x}_i), \nabla^2\psi(\vec{x}_i))]^2 \tag{11}$$

La manera en la que construimos la función de test debe ser tal que se cumplan las condiciones de contorno. Así, ψ_t tiene, en general, la forma:

$$\psi_t(\vec{x}, \vec{p}) = A(\vec{x}) + F(\vec{x}, N(\vec{x}, \vec{p})) \tag{12}$$

donde N es una red neuronal y A contiene únicamente la información de las condiciones de contorno. Veámoslo con algunos ejemplos:

Ejemplo 1

$$\frac{d\psi(x)}{dx} = f\,(x, \psi) \;\; \textit{con } x \in [0,1] \textit{ y tal que } \psi(0) = A$$

Entonces, una solución de test puede ser $\psi(x) = A + xN(x, \vec{p})$.

Ejemplo 2

$$\frac{d^2\psi(x)}{dx^2} = f\left(x, \psi, \frac{d\psi}{dx}\right) \;\; \textit{con } x \in [0,1] \textit{ y tal que } \psi(0) = A, \psi(1) = B$$

Entonces, una solución de test puede ser $\psi(x) = A(1-x) + Bx + x(1-x)N(x, \vec{p})$.

Este primer método de PINN quizás no es tan útil porque no es fácil de generalizar a cualquier ecuación o cualquier condición de contorno. Es necesario proponer una función de test que cumpla la condición de contorno, lo que puede ser difícil si el dominio es complicado.

4.2. PINNs en la actualidad

Una desventaja de las PINNs analizadas en el apartado anterior es que la generalización a EDPs más complejas es muy costosa. Cada tipo de EDP requiere reformular el problema. Recientemente se ha propuesto una versión de las PINNs que resuelve lo anterior, como se describe en [9]. En este caso, consideramos una ecuación diferencial

$$f\left(\vec{x}, \frac{\partial y}{\partial x_1}, ..., \frac{\partial y}{\partial x_d}, \frac{\partial^2 y}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 y}{\partial x_1 \partial x_d}, ...\right) = 0, \quad \vec{x} = (x_1, ..., x_d) \in \Omega \subset \mathbb{R}^d \tag{13}$$

con unas condiciones de contorno

$$B(y, \vec{x}) = 0 \text{ en } \partial\Omega \tag{14}$$

Ahora construimos directamente una red neuronal $\hat{y}(\vec{x}, \vec{\theta})$ dependiente de los pesos y *bias* $\vec{\theta} = \{W^l, b^l\}_{0 \leqslant l \leqslant L}$ de una red de L capas. De nuevo, $\hat{y}$ es la restricción de y a un conjunto de puntos de entrenamiento $\Gamma = \{x_1, ..., x_n\}$, donde Γ es a su vez la unión de $\Gamma_f \subset \Omega$ y $\Gamma_b \subset \partial\Omega$, este último representando el contorno. Para encontrar la solución entrenamos a la red para que minimice la función de coste o *loss*:

$$\mathbf{L}(\theta, \Gamma) = \omega_f \mathbf{L}_f(\theta, \Gamma_f) + \omega_b \mathbf{L}_b(\theta, \Gamma_b) \tag{15}$$

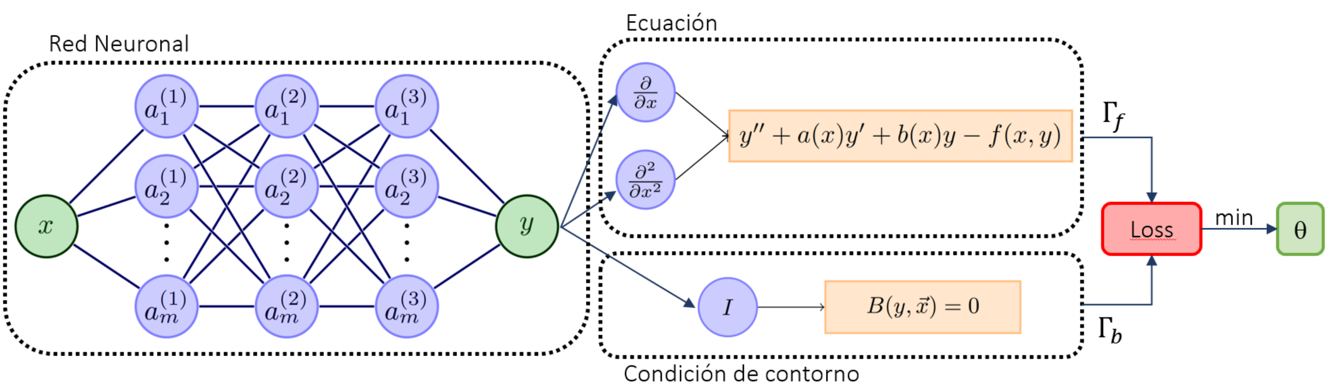
que es una suma pesada de la ecuación diferencial y las condiciones de contorno, lo que hace innecesario añadir capas adicionales para tener en cuenta las condiciones de contorno o iniciales. En nuestro caso, hemos tomado $\omega_f = \omega_b$. Matemáticamente:

$$\mathbf{L}_f(\theta, \Gamma_f) = \frac{1}{|\Gamma_f|} \sum_{\vec{x} \in \Gamma_f} \left| f\left(\vec{x}, \frac{\partial \hat{y}}{\partial x_1}, ..., \frac{\partial \hat{y}}{\partial x_d}, \frac{\partial^2 \hat{y}}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 \hat{y}}{\partial x_1 \partial x_d}, ...\right) \right|^2 \tag{16}$$

$$\mathbf{L}_b(\theta, \Gamma_b) = \frac{1}{|\Gamma_b|} \sum_{\vec{x} \in \Gamma_b} |B(\hat{y}, \vec{x})|^2 \tag{17}$$

donde $|\Gamma_f|$ es el número de puntos de entrenamiento y $|\Gamma_b|$ el número de puntos en la frontera utilizados para imponer las condiciones de contorno.

El Esquema 3 muestra resumidamente el funcionamiento de una PINN para resolver una ecuación diferencial ordinaria de orden 2. Para una ecuación de primer orden el esquema es similar, solo se modifica el recuadro de la ecuación, donde iría el nuevo tipo. Para una EDP habría que añadir más valores de entrada (x's) en la red neuronal. Por otro lado, la diferencia con el primer modelo de PINNs de 1998 es que este método es mucho más general, pues incluye las condiciones de contorno en la red neuronal y estas condiciones pueden ser tan complicadas como queramos.



Esquema 3: *Esquema de una PINN utilizada en este trabajo para EDOs de orden 2.*

5. Implementación a ordenador

Para programar la PINN hemos elegido el lenguaje de programación Python. Este es, en general, el más popular para programar *Machine Learning*, por su sencillez y versatilidad a la hora de programar, pero también por la gran cantidad de paquetes ya desarrollados: TensorFlow y Keras (de *Google*), Pandas, Scikit-learn o PyTorch. Existen otros lenguajes, como R, MATLAB o SQL, que también incluyen paquetes de ML, pero no son tan sencillos ni tan extensos como los disponibles en Python.

Lo que no abundan son los paquetes para programar PINNs, que son un tipo de red neuronal. Es por esto que hemos tomado la decisión de implementar nosotros el programa desde cero. Existe, por ejemplo, DeepXDE [9], en cuyo desarrollo está basado este trabajo y está muy bien para tareas académicas. Sin embargo, solo lo hemos utilizado en contadas ocasiones para verificar nuestros resultados. Uno de los problemas de DeepXDE es que es un paquete de muy alto nivel y no permite ver exactamente como funciona el código. Otro ejemplo de paquete para programar PINNs es NVIDIA Modulus, pero para poder utilizarlo hace falta tener equipo infórmatico de NVIDIA y es más difícil de utilizar.

En concreto, para implementar nuestras PINNs hemos utilizado las librerías de Keras y Tensorflow. Tensorflow consiste en un conjunto de librerías de programación para operar con tensores. Con Tensorflow es posible implementar redes neuronales desde primeros principios, sin embargo, al ser una librería de bajo nivel, su aprendizaje y uso es relativamente complejo. Por otro lado, Keras es una API (*Application Programming Interface*) de alto nivel, en la que es más sencillo crear arquitecturas complejas con redes neuronales, su gran ventaja. Desde 2020 Keras se incluye con los paquetes de Tensorflow, de tal forma que bajo la API de Keras opera Tensorflow. En la mayoría de los casos este hecho es transparente para el programador. En este trabajo se ha utilizado Keras como columna vertebral de la implementación, pero ha sido necesario utilizar Tensorflow en muchos casos, como se explica a continuación.

La mayor complicación en la implementación de las PINNs es que tenemos que modificar la función de coste *loss* para que tenga la forma de (15). En ella intervienen las derivadas de la salida de la red con respecto a las entradas (cuidado, no confundir con las derivadas que aparecen en *Backpropagation*, que son derivadas de la función de coste *loss* con respecto a los parámetros de la red). Para calcular estas derivadas se pueden utilizar varios métodos: calcular las derivadas analíticas manualmente, métodos numéricos (diferencias finitas), derivación simbólica o derivación automática, AD (*Automatic Differentation*). *Backpropagation* es un tipo de AD, que se basa en aplicar la regla de la cadena hacia atrás en la composición de varias funciones, como se ha explicado anteriormente, y es más eficiente que los métodos numéricos. Para calcular las derivadas de la salida con respecto a la entrada, utilizamos también AD a travás de la función GradientTape de TensorFlow.

6. Resultados

A continuación, mostramos los resultados obtenidos para algunas ecuaciones diferenciales, empezando por ecuaciones de primer orden y continuando con ecuaciones de segundo orden. Por último, explicaremos algunos problemas que nos hemos encontrado y posibles soluciones.

6.1. Ecuaciones de primer orden

Para resolver ecuaciones de primer orden, hemos configurado una red neuronal con los parámetros de la Tabla 1.

Hiperparámetro	Valor
n *train*	9
nº capas ocultas	3
nº neuronas/capa	1, 50, 50, 50, 1
nº parámetros	5251
Dropout	0 %
batch size	1
nº épocas	100
inicializador	*Glorot Uniform*
f. activación	elu
optimizador	Adam
learning rate	0,001

Tabla 1: *Configuración de la red neuronal (más información en el texto principal).*

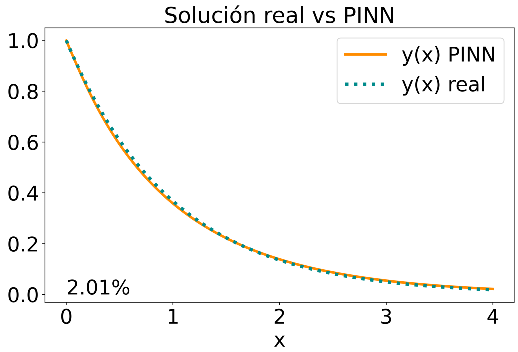
"n *train*" es el número de puntos de entrenamiento, es decir, el cardinal de Γ; "nº capas ocultas" es el número de capas que tiene la red, sin contar la de entrada y la de salida; "nº neuronas/capa" es la estructura topológica de la red, es decir, el número de neuronas que tiene cada capa; "nº parámetros" (grados de libertad) es el número total de pesos y *bias* de todas las neuronas, cuyos valores queremos obtener; "*Dropout*" es el % de neuronas que apagamos al hacer el *Dropout*, "*batch size*" es el número de subconjuntos de entrenamiento; "nº épocas" es el número de épocas de entrenamiento; "inicializador" es el método para introducir los valores iniciales de los pesos y *bias* de las neuronas para iniciar el entrenamiento, normalmente se hace de manera aleatoria siguiendo alguna distribución de probabilidad; "f. activación" es la función de activación; "optimizador" es el algoritmo utilizado para minimizar la *loss*; y "*learning rate*" es la tasa de aprendizaje, que nos indica lo rápido que se realiza el entrenamiento.

Todos estos valores son los hiperparámetros del sistema y deben ajustarse a mano, probando distintas combinaciones que funcionen. Aunque pueda parecer trivial, en ocasiones es difícil hacer esto y es una de las partes que más tiempo requiere.

En primer lugar, resolvemos la ecuación diferencial

$$\begin{cases} y'(x) + y(x) = 0 \text{ con } 0 \leqslant x \leqslant 4 \\ y(0) = 1 \end{cases} \tag{18}$$

de la cual sabemos que su solución real es $y(x) = e^{-x}$, la exponencial decreciente. Esta ecuación aparece en Física, por ejemplo, en los procesos de desintegración nuclear. El resultado calculado por la red neuronal de la Tabla 1 es prácticamente el mismo, como se puede ver en la Gráfica 4.



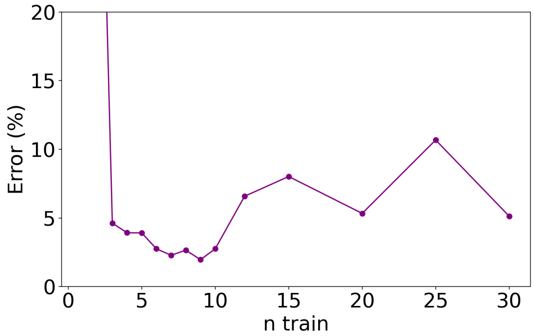
Gráfica 4: *Resultado de la ecuación (18).*

Vemos que la solución calculada por la red neuronal coincide prácticamente con la real. El error cometido es de tan solo un 2,01 %, y podría reducirse algo más si entrenáramos la red durante más épocas o afináramos un poco más los hiperparámetros.

Una de las características importantes a analizar es el tiempo que se tarda en calcular la solución. Como se menciona en la sección 2, un inconveniente de las PINNs es que este tiempo es muy superior al de métodos numéricos clásicos. Pero ¿cuánto mayor? Resolver esta ecuación mediante el método de Runge-Kutta con un discretizado que de un error similar ($\approx 2\,\%$), ha tardado 0,018s, mientras que la PINN ha tardado 13,318s, 740 veces más. Sin embargo, en otros casos en los que Runge-Kutta requiera un discretizado muy fino, esta diferencia no será tan grande.

Naturalmente, el tiempo de ejecución de la PINN también depende (como Runge-Kutta) del número de puntos de entrenamiento n *train*. Sin embargo, en contra de la intuición, el resultado no siempre mejora al aumentar n *train*. Hemos encontrado que una densidad de puntos de entrenamiento de 2 por cada unidad de x arroja en general los mejores resultados (aunque el valor óptimo puede ser ligeramente distinto para cada ecuación). Por tanto, en la mayoría de ocasiones hemos elegido n *train*$= 2(max(I) - min(I)) + 1$, donde I es el intervalo de x en el buscamos la solución. Así pues, en este caso, en el que buscamos la solución para $x \in I = (0, 4)$, elegimos $2 \cdot 4 + 1 = 9$ puntos de entrenamiento equiespaciados $\Gamma = \{0, 1/2, 1, 3/2, 2, 5/2, 3, 7/2, 4\}$. Para esta ecuación hemos estudiado más en profundidad cómo varía el error en función del número de puntos de entrenamiento. Los resultados se muestran en la Gráfica 5.

Como puede verse, se obtiene, aproximadamente, un mínimo entre 5 y 10, que se corresponde con tener entre 1 y 2 puntos de entrenamiento por unidad de x. Si tenemos muy pocos puntos es evidentemente que no funciona bien, pero lo curioso es que tampoco funciona bien para valores muy grandes de n *train*.

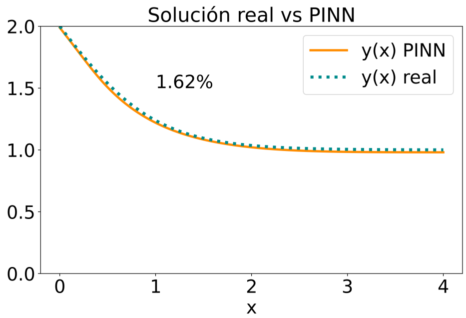


Gráfica 5: *Dependencia del error final en la solución con el número de puntos de entrenamiento.*

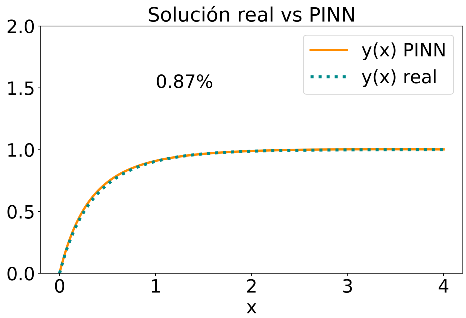
A continuación, consideremos una ecuación logística, ampliamente utilizada en sistemas dinámicos para modelizar poblaciones y epidemias:

$$\begin{cases} y'(x) - (y(x) - 1)(y(x) - 3) = 0 \text{ con } 0 \leqslant x \leqslant 4 \\ y(0) = 2 \end{cases} \tag{19}$$

cuya solución es $y(x) = 3 - \frac{2}{e^{-2x}+1}$. En este caso, utilizando la misma PINN que en la ecuación anterior, volvemos a obtener un buen resultado, como muestra la Gráfica 6.



(a) *Condición inicial $y(0) = 2$. El error cometido es de un 1,62 %.*



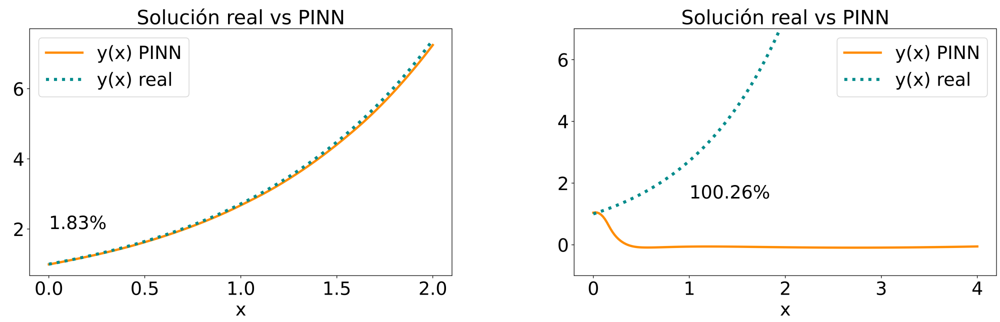
(b) *Condición inicial $y(0) = 0$. El error cometido es de un 0,87 %.*

Gráfica 6: *Resultado de la ecuación (19) en $x \in I = (0, 4)$ para dos condiciones iniciales.*

Por último, veamos el caso de una ecuación diferencial con solución no acotada:

$$\begin{cases} y'(x) - y(x) = 0 \text{ con } 0 \leqslant x \leqslant 2 \\ y(0) = 1 \end{cases} \tag{20}$$

cuya solución real es $y(x) = e^x$. Nótese que de momento se busca la solución en el intervalo $I = (0, 2)$. Con la misma configuración de la red que en el caso anterior, pero pasando de 100 épocas a 400, se obtiene la Gráfica 7a. En este dominio más pequeño se sigue obteniendo una buena solución. Sin embargo, para un dominio de $x \in I = (0, 4)$, Gráfica 7b, no hemos sido capaces de encontrar una combinación de hiperparámetros que funcione. La solución real diverge, pero la calculada por la red tiende rápidamente hacia 0. Este fenómeno nos lo hemos encontrado siempre que la solución de la ecuación no está acotada y queremos obtener la solución en un dominio demasiado grande. Más adelante, cuando hablemos de ecuaciones de segundo orden, nos extenderemos en profundidad sobre esto y plantearemos algunas posibles soluciones.



(a) *En* $x \in I = (0, 2)$.

(b) *En* $x \in I = (0, 4)$.

Gráfica 7: *Resultado de la ecuación 20.*

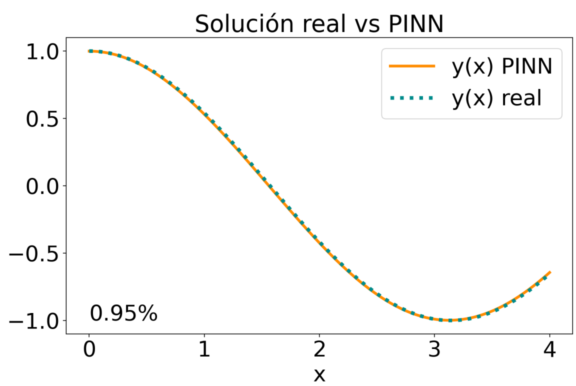
La conclusión que podemos sacar, de momento, es que el método de PINNs para resolver ecuaciones diferenciales ordinarias de primer orden funciona muy bien para ecuaciones cuyas soluciones están acotadas y en dominios no muy grandes.

6.2. Ecuaciones de segundo orden

Ahora veamos el caso de ecuaciones de segundo orden. En particular, se va a estudiar en profundidad el funcionamiento de la PINN en el oscilador armónico:

$$\begin{cases} y''(x) + y(x) = 0 \text{ con } 0 \leqslant x \leqslant 4 \\ y(0) = 1, y'(0) = 0 \end{cases} \tag{21}$$

cuya solución es $y(x) = \cos x$. Partimos de una PINN con los hiperparámetros de la Tabla 2. Notar que, a diferencia de la red utilizada en las ecuaciones de primer orden, ahora hemos obtenido mejores resultados con la función de activación tanh (aunque también funciona bien con elu). Ambas, elu y tanh, tienen en común que son de las pocas funciones de activación que toman valores negativos. La PINN consigue encontrar la solución con un error de tan solo un 1 %, como se puede ver en la Gráfica 8.

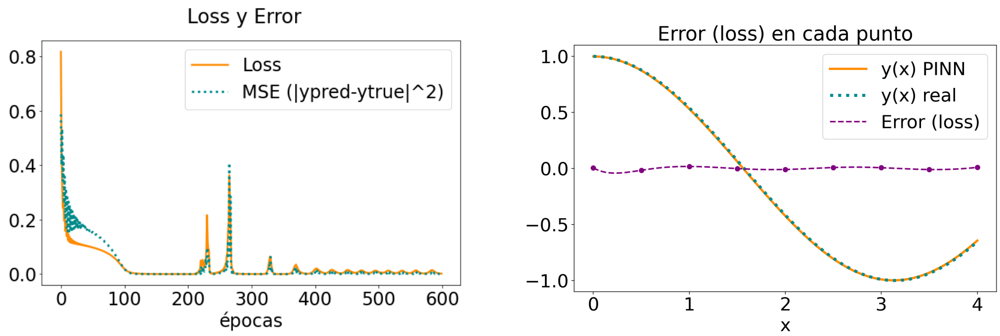


Gráfica 8: *Resultado de la ecuación (21) en* $x \in I = (0, 4)$.

Hiperparámetro	Valor
nº *train*	9
nº capas ocultas	3
nº neuronas/capa	1, 50, 50, 50, 1
nº parámetros	5251
Dropout	0 %
batch size	1
nº épocas	600
inicializador	Glorot Uniform
f. activación	tanh
optimizador	Adam
learning rate	0,001

Tabla 2: *Configuración de la red neuronal.*

Para estudiar en profundidad cómo funciona la red, echamos un vistazo a la Gráfica 9. En (9a) vemos los valores, para cada época de entrenamiento, de la función de coste *loss*, que tiene la forma de la ecuación (15), y del error cuadrático medio (MSE) entre la solución real y la calculada por la PINN. En (9b) vemos, además de las dos soluciones, el error en la ecuación diferencial en cada punto. Esto es, en cada punto representamos $y''(x) + y(x)$, que es el error o residuo, y que debería valer 0 si la PINN encontrara la solución real. Este valor es también la *loss* en cada punto, sin contar la condición inicial. Esta curva se calcula en todo el dominio, no solo en los puntos de entrenamiento, una vez entrenada la red.



(a) *Evolución de la loss y el MSE en el entrenamiento de la red.*

(b) *Ecuación diferencial (residuo o loss) en cada punto.*

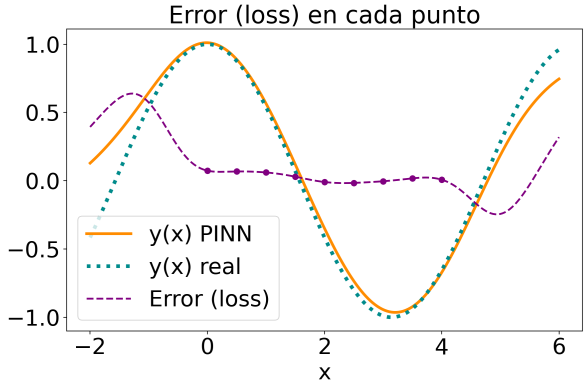
Gráfica 9: *Funcionamiento de la red.*

El MSE es el dato que realmente nos importa; sin embargo, solo lo podemos calcular si tenemos la solución real. Si no conocemos la solución real de la ecuación, lo cual es lo que nos interesa si queremos que las PINNs sean útiles, nos tenemos que limitar a calcular solo la *loss*. Como vemos, la red parece quedar bastante bien entrenada tras unas 100 épocas. Sin embargo, aparecen posteriormente algunos picos que pueden deberse a saltos entre mínimos locales. Por otro lado, respecto al cálculo de la *loss*, podría ocurrir que la red calcule una solución con una *loss* baja, pero que no coincida con la solución real. En el caso concreto que estamos estudiando, tanto la *loss* como el MSE son prácticamente cero.

Lo que sí nos aporta más información es el "Error (loss)" en cada punto, que aunque no es cero en este caso, es realmente pequeño en todo el dominio. Esto sí nos indica que hemos encontrado una buena solución y es una buena métrica porque para construir esta curva no necesitamos conocer la solución real.

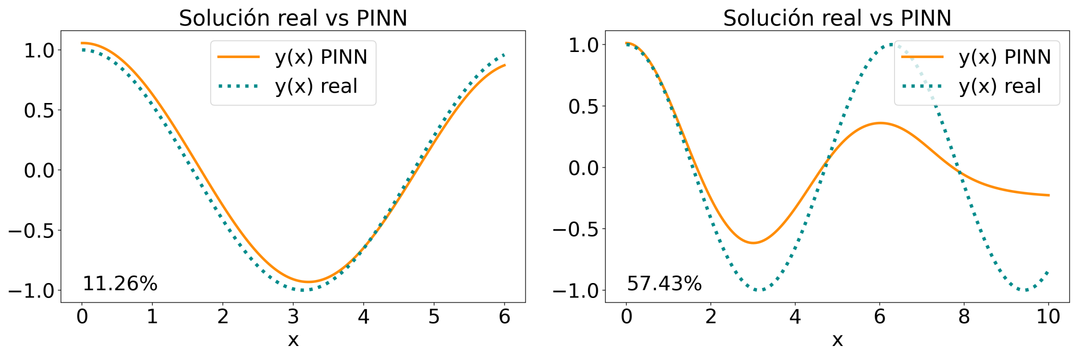
Otro parámetro a analizar es, como en el caso de primer orden, el tiempo necesario para entrenar a la red. Como vemos en la Gráfica 9, basta con entrenar la red durante unas 200 épocas, lo que tarda un tiempo de unos 40 segundos y nos da un error del 1 %. Ejecutar un programa de Runge-Kutta para calcular la solución de la misma ecuación, en el mismo dominio, y con un error similar de un 1 %, cuesta unos 0,064 segundos. Así pues, la PINN tarda aproximadamente 625 veces más. El mismo análisis en la ecuación de primer orden nos dio que la PINN tarda 740 veces más que Runge-Kutta. Como vemos, parece que, efectivamente, al ir complicando el problema, Runge-Kutta necesita un discretizado cada vez más fino, y la diferencia de tiempo entre un método y otro se reduce. No obstante, hay que destacar que invirtiendo más tiempo en la selección de hiperparámetros de la red, se podría reducir más el tiempo de entrenamiento. El objetivo del trabajo no era tanto esto, sino elegir una red más o menos general que se comporte bien en una gran variedad de ejemplos.

Por otro lado, una ventaja de las PINNs es que permiten extrapolar ligeramente la solución fuera del intervalo de entrenamiento. Esto es, que existen unos valores $0 \leqslant R_1, R_2 \in \mathbb{R}$ tales que $|y_{PINN}(x) - y_{real}(x)| < \epsilon$, $\forall x \in [0 - R_1, 4 + R_2]$, para algún $\epsilon > 0$. Es decir, que nos podemos salir un poco por fuera del intervalo de entrenamiento y la solución sigue siendo suficientemente buena. Esto lo podemos ver en la Gráfica 10, en la que hemos entrenado la red en $(0, 4)$, pero mostramos la solución en un rango mayor. Como vemos, especialmente en $(4, 6)$, la solución sigue siendo bastante buena. De nuevo, la curva Error(loss) nos indica las zonas en las que se obtiene casi la solución real. Esta extrapolación se puede hacer en un intervalo más grande que lo que podríamos hacer con Runge-Kutta. Esta importante ventaja aparece ya en el artículo de 1998 [17].



Gráfica 10: *Extrapolación de la solución a un intervalo mayor.*

Ahora veamos qué pasa si ampliamos el dominio en el buscamos la solución de $(0, 4)$ a $(0, 6)$ o $(0, 10)$, aumentando también *n train* como se ha comentado anteriormente. Como en el caso de ecuaciones de primer orden, conforme agrandamos el dominio y nos alejamos de la condición inicial, la red empieza a fallar. Mostramos los resultados en la Gráfica 11.



(a) *Solución de (21) en* $x \in I = (0, 6)$. (b) *Solución de (21) en* $x \in I = (0, 10)$.

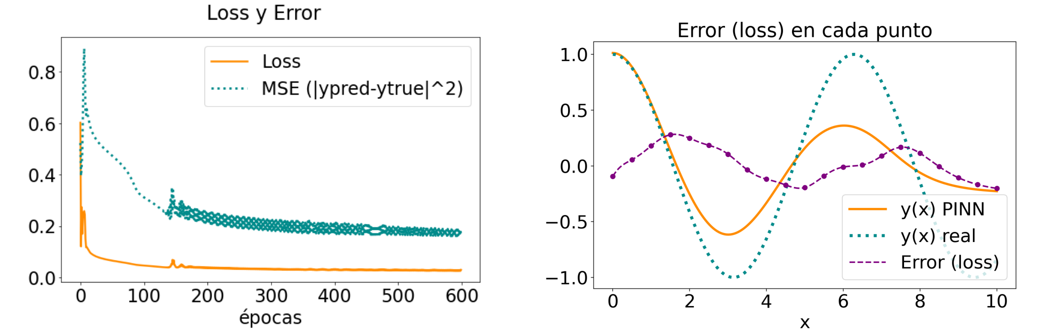
Gráfica 11: *Solución de (21) en varios dominios.*

Al extender el dominio a (0,6), la PINN sigue funcionando razonablemente bien. Sin embargo, en (0,10) la red nos vuelve a dar una mala solución que va tendiendo a cero.

6.3. Problemas de las PINNs

Como hemos ido viendo a lo largo del trabajo, el mayor problema al que nos estamos enfrentando es que en dominios muy grandes la PINN no es capaz de encontrar la solución real. En particular, cuando nos alejamos mucho de la condición inicial, la solución de la red cae a 0. En esta sección analizamos la causa.

No es algo que debería sorprendernos, ya que la función $y(x) = 0$, $\forall x \in I$ es siempre solución de estas ecuaciones diferenciales. Así que, de alguna manera, la red lo que hace es encontrar todo el rato la solución nula. El problema es que esta solución no satisface la condición inicial. Para estudiar mejor lo que está haciendo la red, reproducimos la Gráfica 9 para el caso de $x \in I = (0, 10)$, que se observa en 11b.



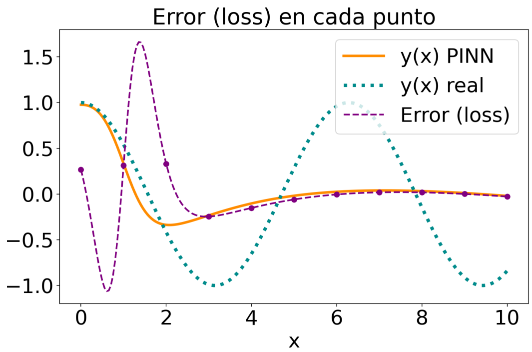
(a) *Evolución del entrenamiento de la red.* (b) *Ecuación diferencial (loss) en cada punto.*

Gráfica 12: *Funcionamiento de la red.*

Lo que observamos en la gráfica de la derecha es que en torno al $x = 2$ la curva Error (loss) aumenta y encuentra un máximo. En este pico la red salta a otra solución de la ecuación diferencial, pero que no satisface la condición inicial. A partir de ahí, en concreto a partir de $x \approx 9$, la curva Error(loss) vuelve a dar cero y la solución que da la red es la solución nula. Por tanto, es importante recalcar que el hecho de obtener Error(loss) casi nulo no nos garantiza que hayamos obtenido la solución que buscamos. Las conclusiones que podemos sacar son:

1. Si la curva Error(loss) es casi nula en todo el dominio, hemos encontrado una buena aproximación a la solución real.
2. Si esta curva tiene algún pico, a partir de ahí, podremos haber saltado a otra solución de la ecuación que no satisface la condición inicial, como es la solución nula. Así pues, la curva Error(loss) nos da una idea del rango de validez de la solución de la PINN.

Otro aspecto importante relacionado con este fenómeno es que este salto entre soluciones, y por tanto ese aumento en Error(loss), puede ocurrir entre dos puntos de entrenamiento consecutivos. Es decir, Error(loss) puede ser casi nulo en dos puntos de entrenamiento consecutivos, pero no entre medio. Esto lo podemos entender como un *overfitting*: la red funciona bien en los puntos de entrenamiento, pero no es capaz de generalizar el resto del dominio. Cabría esperar que este efecto será más visible en redes muy grandes (con muchos parámetros/neuronas) o con pocos puntos de entrenamiento. Esto es precisamente lo que sucede. Para una red con una topología (1, 500, 500, 500, 1) y 11 puntos de entrenamiento en (0,10) obtenemos la Gráfica 13.



Gráfica 13: *Overfitting y salto de la solución.*

Como vemos, los picos en Error(loss) suceden entre puntos de entrenamiento consecutivos. Vemos que la curva y_{PINN} se aproxima bastante bien a la real para valores muy pequeños de x. Después se producen los picos en Error(loss) y a partir de ahí, la solución de la PINN ha saltado a la solución nula, que cumple $loss = 0$ y por tanto satisface la ecuación diferencial, pero no la condición inicial. Si solo hubiéramos calculado Error(loss) en los puntos de entrenamiento podríamos haber pensado que hemos encontrado una buena solución. De modo que es de vital importancia asegurarnos de que Error(loss) es baja en todos los puntos del dominio y no solo en los de entrenamiento.

6.4. Posibles soluciones

A continuación, proponemos algunas soluciones a este problema. Algunas de ellas las hemos implementado y estudiado directamente en este trabajo, otras por su complejidad y las restricciones de tiempo propias del TFG, no se desarrollan aquí.

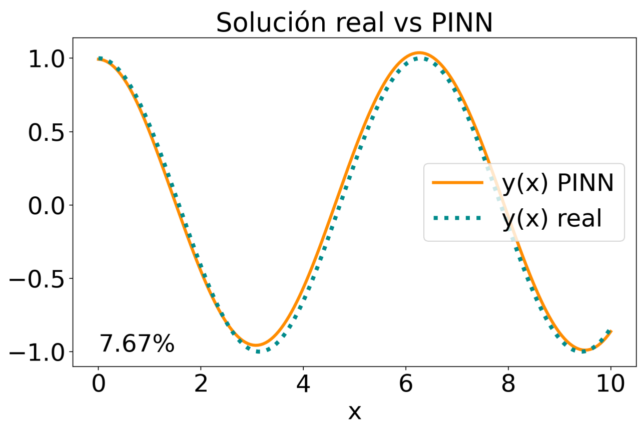
6.4.1. Dropout

Seleccionando un *Dropout* del 50 %, es decir, apagando al azar la mitad de las neuronas en cada paso de entrenamiento, conseguimos reducir en parte este *overfitting*. Sin embargo, se tarda más tiempo en llegar a la solución.

6.4.2. Añadir más condiciones de contorno

Una posible solución, utilizada en la mayoría de los ejemplos de [9], es añadir varias condiciones de contorno/iniciales. Sin embargo, esto no siempre es una buena práctica porque muchas veces no conocemos los valores de la solución en tantos puntos como nos gustaría. En muchos problemas físicos tenemos control sobre las condiciones iniciales: o las sabemos o las podemos imponer. De ahí que sea relevante resolver ecuaciones diferenciales donde solo conocemos un valor de la solución. No obstante, siempre que las conozcamos, añadiremos tantas condiciones de contorno como podamos.

Para ver el efecto de mejora de estas dos propuestas, calculamos la solución al problema (21), con la misma red de la Tabla 2, pero cambiando dos cosas: ponemos un *Dropout* del 50 % y añadimos las condiciones $y(x = 10) = \cos(10)$ y $y'(10) = -\sin(10)$. Como puede verse en la Gráfica 14, conseguimos reducir el error de un 57 % a un 8 %.



Gráfica 14: *Resultado de la ecuación (21) en* $x \in I = (0, 10)$ *con Dropout y dos condiciones de contorno.*

Tener una condición de contorno en el extremo izquierdo y otra en el derecho, es lo que ocurre, por ejemplo, cuando se resuelve la ecuación de Schrödinger en el pozo infinito de potencial. Además, para resolver la ecuación de Schrödinger, se podría imponer la condición de normalización del cuadrado de la función de onda, por ser una densidad de probabilidad. Esto evitaría que la PINN devuelva la solución nula en todos los puntos.

6.4.3. Penalizar la solución nula

Otra forma de evitar la solución nula es añadir en la función *loss* un término inversamente proporcional a la norma de la solución. Sin embargo, hemos rechazado esta propuesta porque la red no funcionaría en ecuaciones diferenciales cuya solución valga 0.

6.4.4. Redes recurrentes LSTM

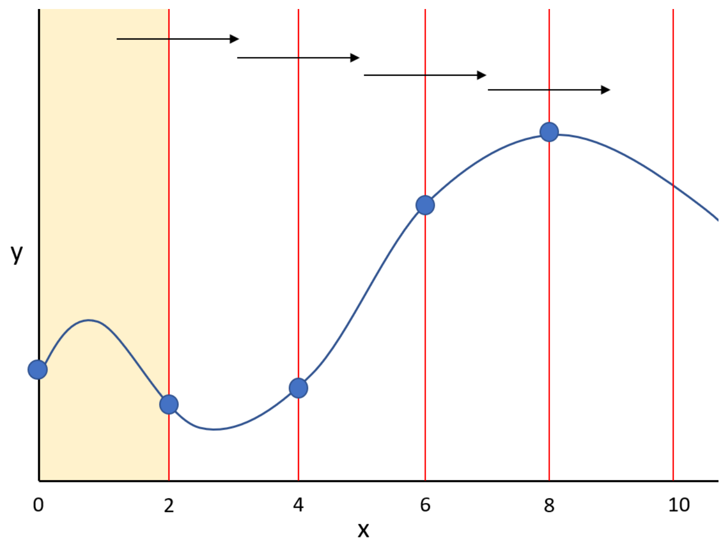
Algunos artículos científicos, por ejemplo [18][19], trabajan directamente con redes recurrentes del tipo LSTM. Este tipo de redes se utilizan, por ejemplo, en el reconocimiento del lenguaje (en traductores o asistentes personales como *Siri* o *Alexa*), donde el significado de una palabra afecta directamente a la palabra siguiente. De esta manera, se trata a los datos de forma secuencial. En el caso de las PINNs el valor de la solución en un punto dependerá directamente del valor de la solución en los puntos anteriores. De esta manera, la red no olvidará la condición inicial y se retrasarán los saltos entre soluciones de la ecuación con distintas condiciones iniciales.

6.4.5. Método iterativo

Dado que los problemas aparecen cuando entrenamos la red en dominios muy grandes y nos alejamos de la condición inicial, una forma de solucionar este problema es aprovechar que las PINNs funcionan bien en dominios pequeños. Lo que hacemos es partir el dominio, en este caso el intervalo $I = (0, 10)$, en subintervalos más pequeños, por ejemplo, $I_1 = (0, 2)$, $I_2 = (2, 4)$, $I_3 = (4, 6)$, $I_4 = (6, 8)$ y $I_5 = (8, 10)$. Ahora lo que hacemos es un proceso de entrenamiento iterativo en estos intervalos:

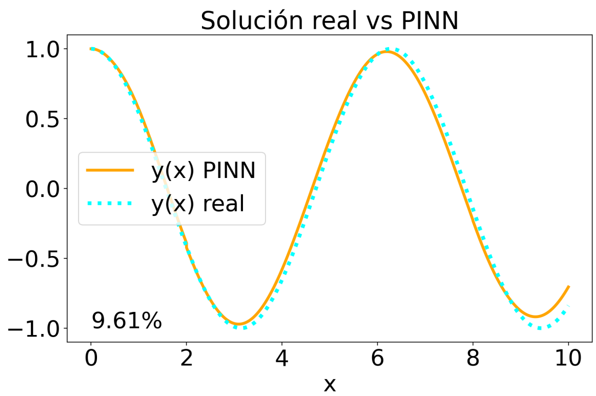
Algoritmo 1 Método iterativo

Elegir $0 \leqslant n =$ número de subintervalos $\in \mathbb{N}$
Entrenar la red en I_1 con c.i. en el $x = 0$
Guardar modelo
for i de 2 a n **do**
    Abrir modelo
    Reentrenar la red en I_i con c.i.=max$\{I_{i-1}\}$
    Guardar modelo



Esquema 4: *Esquema del método iterativo.*

Así pues, entrenamos la red en cada uno de los subintervalos por separado, moviéndonos de izquierda a derecha y tomando como condición inicial de cada uno el último punto del subintervalo anterior. De esta manera, al finalizar el entrenamiento, tenemos una sola red capaz de predecir la solución en todo el dominio. Así, conseguimos obtener una solución con un error menor del 10 %, como puede verse en la Gráfica 15. Notar que de esta manera no estamos aportando información extra a la red, como si hacemos cuando añadimos condiciones de contorno.



Gráfica 15: *Resultado de la ecuación (21) en* $x \in I = (0, 10)$ *con el método iterativo.*

Otra forma parecida de proceder sería, en lugar de partir de unos subintervalos preestablecidos, que sea el programa el que los elija de forma conveniente. Podríamos elegir que un intervalo dado acabe cuando el valor de la curva Error(loss) supere un cierto umbral o tenga un pico, como veíamos en la Gráfica 12b. De esta manera nos podríamos ahorrar tiempo de cálculo.

Por último, para comprobar la validez de la solución de la PINN, nos hemos planteado una segunda forma de calcular la solución a partir de la red. Esta forma consiste en utilizar solo las derivadas proporcionadas por la PINN entrenada y la definición de integral:

$$y_I(x) = y(0) + \int_0^x y'(s)ds \tag{22}$$

donde la integral la calculamos de forma numérica. Lo que nos hemos encontrado es que esta solución y_I coincide con la de la PINN, así que este método no mejora la solución. Esto lo podemos interpretar como que en realidad la PINN lo que está haciendo es realizar una integral. De hecho, se está investigando también actualmente la capacidad de las redes neuronales de representar operadores matemáticos, como la derivada o la integral. Al igual que una red neuronal puede representar cualquier función, también se pueden encontrar redes neuronales que realicen operaciones como derivar o integrar.

7. Conclusiones

En este trabajo hemos implementado una de las aplicaciones más novedosas e interesantes de la Inteligencia Artificial en la ciencia, las *Physics-Informed Neural Networks* (PINN), que son redes neuronales normales en las que modificamos la función de coste *loss* permitiendo así resolver ecuaciones diferenciales. El trabajo comienza con una introducción al funcionamiento básico de las redes neuronales. A continuación, se explica la primera versión histórica de las PINNs. Por último, hemos implementado y mostrado los resultados obtenidos con las PINNs más utilizadas en la actualidad. A pesar del uso extendido de librerías de *Machine learning* como *Keras* o *Tensorflow*, estas todavía no incluyen la posibilidad de programar PINNs de manera sencilla. No obstante, existen otras librerías más específicas, como *DeepXDE* que permiten hacer este trabajo. Sin embargo, en este trabajo hemos implementado el programa desde cero. Además, hemos explorado de manera exitosa algunas soluciones originales a los problemas que nos hemos ido encontrando.

En cuanto a los resultados obtenidos, por un lado, cabe destacar que, tanto en EDOs de primer como de segundo orden, hemos obtenido buenos resultados en dominios pequeños utilizando PINNs básicas. Por otro lado, para problemas con un dominio más grande, hemos conseguido implementar un nuevo método iterativo que permite obtener la solución. Por último, aunque en este trabajo nos hemos limitado a las EDOs más sencillas, esta técnica es igualmente valida para resolver EDPs o sistemas de ecuaciones, y se podría continuar en esa linea como trabajo futuro.

Pese a que el campo de la IA, en particular el de las PINNs, es relativamente nuevo (cuesta encontrar referencias con más de 3 años de antigüedad), ha despertado un gran interés con la esperanza de mejorar o sustituir las herramientas de cálculo numérico clásicas, que en algunas aplicaciones son casi imposibles de utilizar. A pesar de los grandes avances ya producidos por las PINNs, todavía queda un largo camino por recorrer para que sean ampliamente utilizadas.

8. Bibliografía

Referencias

[1] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[2] Rumelhart David E., Hinton Geoffrey E., and Williams Ronald J. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[3] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4), Dec 2019.

[4] Jan Hermann, Zeno Schätzle, and Frank Noé. Deep-neural-network solution of the electronic schrödinger equation. *Nature Chemistry*, 12(10):891–897, sep 2020.

[5] Antonio A. Gentile, Brian Flynn, Sebastian Knauer, Nathan Wiebe, Stefano Paesani, Christopher E. Granade, John G. Rarity, Raffaele Santagati, and Anthony Laing. Learning models of quantum systems from experiments. *Nature Physics*, 17(7):837–843, apr 2021.

[6] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. Machine learning for fluid mechanics. *Annual Review of Fluid Mechanics*, 52(1):477–508, 2020.

[7] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (pinns) for fluid mechanics: A review. Acta Mechanica Sinica, 37:1727-1738, 2021.

[8] Sunae So, Trevon Badloe, Jaebum Noh, Jorge Bravo-Abad, and Junsuk Rho. Deep learning enabled inverse design in nanophotonics. *Nanophotonics*, 9(5):1041–1057, 2020.

[9] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, jan 2021.

[10] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[11] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[12] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015, 10.48550/ARXIV.1412.6980.

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

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[16] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. Computing Research Repository (CoRR), abs/1207.0580, 2012.

[17] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[18] Murphy Yuezhen Niu, Lior Horesh, and Isaac Chuang. Recurrent neural networks in the eye of differential equations. arXiv, abs/1904.12933, 2019.

[19] Mansura Habiba and Barak A. Pearlmutter. Neural ordinary differential equation based recurrent neural network model, 2020. (Preprint), 10.48550/ARXIV.2005.09807.