**Universidad de Zaragoza**

Facultad de Ciencias

# Evolución de sistemas dinámicos mediante aprendizaje profundo

*Trabajo Fin de Grado*

Autor:
Natalia Robres Portella

Directores:
Sergio Gutiérrez Rodrigo
Pablo Calvo Barlés

Julio 2024

# Agradecimientos

*Me gustaría agradecer a mis directores, Sergio Gutiérrez y Pablo Calvo, por su apoyo constante y grandes consejos a lo largo de este proyecto.*

*A mis amigos y a mi familia por ser mi pilar durante estos cuatro años de carrera. Os debo gran parte de esto y os agradezco infinitamente haber creido y confiado en mí.*

# Resumen

En este trabajo exploramos el uso de las Redes Neuronales Informadas por Física (PINNs) para resolver ecuaciones diferenciales en sistemas dinámicos. Se abordan aplicaciones desde el modelo de Romeo y Julieta hasta las Ecuaciones de Lorenz, destacando la versatilidad de las PINNs, que combinan principios físicos con técnicas de aprendizaje automático para modelar fenómenos complejos.



# Abstract

In this work, we explore the use of Physics-Informed Neural Networks (PINNs) to solve differential equations in dynamic systems. Applications range from the Romeo and Juliet model to the Lorenz Equations, highlighting the versatility of PINNs, which integrate physical principles with machine learning techniques to model complex phenomena.

# Índice

# 1. Introducción

Hasta hace relativamente poco era impensable contar con un asistente virtual como Siri o Alexa, traducir idiomas con un dispositivo portátil o incluso que un coche pudiera conducir por su cuenta. La Inteligencia Artificial (IA) ha irrumpido en nuestras vidas y ahora convivimos diariamente con sus avances. Tareas que solamente podían ser abordadas por la inteligencia humana, como el reconocimiento de texto e imágenes, son ahora resueltas por la IA con gran éxito.

Una de las principales ramas de la IA es el denominado *Aprendizaje automático* (AA), que se enfoca en el desarrollo de algoritmos con los que una máquina puede aprender a realizar tareas a partir de un conjunto de datos. Se distingue principalmente entre aprendizaje supervisado y no supervisado. En el aprendizaje supervisado, la máquina aprende a partir de un conjunto de datos etiquetados, de manera que es guiada con ejemplos de cómo debe actuar. Por otro lado, en el aprendizaje no supervisado las máquinas aprenden patrones sobre conjuntos de datos sin etiquetas, sin recibir ninguna guía.

El *Aprendizaje Profundo* (AP) es, en particular, una de las ramas del AA [1]. Este subcampo utiliza Redes Neuronales (RN) profundas, que son "máquinas" computacionales cuya estructura se compone de múltiples capas de procesamiento. Utilizando aprendizaje supervisado, éstas son capaces de realizar tareas de gran complejidad. Existen distintos tipos de redes profundas. Concretamente, las *Physics-informed neural networks* (PINNs) son redes que pueden resolver ecuaciones diferenciales de cualquier tipo, combinando principios físicos con técnicas de AA [2]. Las PINNs se han utilizado en diversas áreas de la física. Por ejemplo, han resultado útiles en la resolución de ecuaciones de movimiento que describen sistemas complejos, como la dinámica de fluidos, la evolución de sistemas mecánicos, etc [3].

Una ventaja clave de las PINNs es su eficiencia computacional. No necesitan mallados finos ni discretizaciones específicas, por lo que el uso de las PINNs disminuye en gran medida el tiempo de cálculo y los recursos computacionales. Esto es especialmente notable en sistemas de grandes dimensiones. A pesar de ser un campo relativamente joven, las PINNs han captado considerable interés debido a su potencial para mejorar o reemplazar las herramientas tradicionales de cálculo numérico. Estas redes representan una promesa significativa en aplicaciones donde los métodos clásicos enfrentan limitaciones prácticas, aunque aún se encuentran en una fase inicial de desarrollo.

Por otro lado, los sistemas dinámicos son un área fundamental de la física y la matemática aplicada. Estos sistemas evolucionan con el tiempo y se describen a partir de ecuaciones diferenciales ordinarias (EDOs). Clasificamos los sistemas dinámicos en dos grandes categorías: sistemas no caóticos y sistemas caóticos. Los sistemas no caóticos son aquellos que presentan un comportamiento predecible, en el sentido de que las trayectorias apenas son sensibles a las condiciones iniciales. Dos ejemplos conocidos de sistemas

con esta naturaleza son el oscilador armónico o el problema de dos cuerpos con atracción gravitatoria. Los sistemas caóticos son aquellos sistemas dinámicos extremadamente sensibles a las condiciones iniciales. Las ecuaciones de Lorenz, desarrolladas por Edward Lorenz en 1963 [4] son un ejemplo emblemático de sistema caótico.

En este trabajo emplearemos las PINNs para resolver EDOs que describen sistemas dinámicos. Comenzaremos abordando los conceptos básicos de las redes neuronales. A continuación, se expondrán los resultados obtenidos a partir de las simulaciones realizadas. Finalmente, discutiremos las conclusiones derivadas del trabajo.

# 2. Objetivos y metodología

El objetivo principal de este trabajo es diseñar RNs capaces de resolver sistemas de ecuaciones diferenciales. Para lograr este objetivo, se implementarán PINNs y se evaluará su eficacia en distintos casos.

Comenzaremos con un caso sencillo, la ecuación de movimiento constante. Una vez verifiquemos que la PINN funciona correctamente, estudiaremos la ecuación del amor de Romeo y Julieta a modo de ejemplo de sistema no caótico, y posteriormente ampliaremos el estudio a las ecuaciones de Lorenz, que como ya se ha comentado, se tratan de un sistema caótico [5].

La implementación de las PINNs se ha llevado a cabo en el lenguaje de programación *Python*. En el Anexo A se presenta el código empleado. *Python* es conocido por su versatilidad y por ser gratuito. Ofrece numerosas bibliotecas que facilitan la implementación de algoritmos de AP. Las principales bibliotecas empleadas en el trabajo son *Tensorflow* y *Keras*. En cuanto a la primera de ellas, su nombre proviene de la unión de tensor y flujo, es una librería que permite ejecutar gráficos de flujo de datos de manera eficiente, lo que la hace especialmente adecuada para el manejo de grandes volúmenes de datos y la creación de modelos de AP complejos. *Keras*, por su parte, es una biblioteca que puede ejecutar sobre *Tensorflow*. Es una biblioteca de alto nivel para manejarse con multitud de herramientas propias de las RNs. *Keras* facilita la implementación de algoritmos de AP al ofrecer herramientas ya configuradas y módulos que se pueden reutilizar.

Otras bibliotecas adicionales que se utilizan durante el trabajo son *Numpy* y *Matplotlib*. *Numpy* es una biblioteca fundamental para el cálculo numérico en *Python*. Dispone de soporte para tratar con matrices multidimensionales además de una gran variedad de funciones matemáticas para operar con estos datos. *Matplotlib* se empleará para visualizar los datos y resultados de nuestro modelo. Permite crear figuras que facilitan la interpretación de los resultados.
Además de estas bibliotecas, se han empleado entornos de desarrollo integrados como *Jupyter Notebooks* para facilitar la tarea de programación.

# 3. Conceptos básicos sobre redes neuronales

## 3.1. Perceptrón

Nos remontamos a los años 50, cuando un científico llamado Frank Rosenblatt desarrolló el perceptrón [6]. El perceptrón es una idea inspirada en la neurona biológica, una célula caracterizada por poseer una serie de canales de entrada (inputs), llamados dendritas y un canal de salida (output), llamado axón. Las dendritas de una neurona recogen información de otra neurona y a través del axón se pasa esta información a las dendritas de la siguiente neurona. En este ejemplo, es claro que una neurona aislada carece de sentido, éstas adquieren importancia cuando funcionan como una red con distintas capas. Lo mismo ocurre en las RNs artificiales.

Un perceptrón recibe uno o más inputs binarios y produce un único output binario. Para calcular este output es necesario introducir el concepto de pesos. Cada input, $x_1, ..., x_i$, tiene asociado un peso, $w_1, ..., w_i$, que marca el nivel de importancia de dicha entrada a la hora de calcular el output.

El output del perceptrón es 0 si la suma de los pesos multiplicados por sus respectivas entradas es menor o igual que un valor umbral que hemos de establecer y es 1 si es mayor. Matemáticamente, esto quedaría de la siguiente manera:

$$output = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leq umbral \\ 1 & \text{si } \sum_j w_j x_j > umbral \end{cases} \tag{1}$$

En resumen, el perceptrón podría considerarse una manera de tomar decisiones midiendo o pesando la importancia de ciertas circunstancias.

Para simplificar la forma en la que describimos los perceptrones (1), definimos el sumatorio como un producto escalar y pasamos el término del umbral al otro lado, definiendo el sesgo, $b$, del perceptrón como $b \equiv -umbral$. Podemos ver el sesgo como la facilidad que tiene el perceptrón para proporcionar como salida 1. Introducimos ahora el concepto de neurona artificial a modo de generalización del perceptrón.

Una neurona artificial es una función que recibe un conjunto de números escalares, valores de input, y los lleva a un número real mediante la composición de transformaciones lineales con la acción de una función no lineal cualquiera, a diferencia del perceptrón, que tiene por definición únicamente la función escalón como activación.

Podemos resumir esto en el siguiente esquema

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

donde a es la función no lineal, conocida como función de activación.

## 3.2. Funciones de activación

Las funciones de activación son funciones no lineales que reciben el valor de $z$ y lo transforman en $a(z)$. La elección de la función de activación es crucial a la hora de desarrollar RNs eficientes.

Veamos cuales son las funciones de activación más populares en AP.

### 3.2.1. La función sigmoide

Esta función toma como entrada cualquier valor y lo transforma en un valor comprendido entre 0 y 1. Viene dada por la expresión 2 y se representa en la Figura 1a.

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

La función satura a 1 cuando los valores de entrada son muy altos, y a 0 cuando son muy bajos.

### 3.2.2. La función tangente hiperbólica

En este caso, los valores de salida están comprendidos entre -1 y 1. A diferencia que la función sigmoide, esta función proporciona una salida antisimétrica.

Viene dada por la expresión 3 y se representa en la Figura 1b.

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{3}$$

### 3.2.3. La función ReLU

Esta función es ampliamente utilizada en redes profundas. Esta función no satura, lo que mejora la velocidad de convergencia del algoritmo. Toma los valores de entrada negativos y los transforma en 0. Los valores positivos no se ven modificados por esta función. Se recomienda el uso de esta función para las capas ocultas de la red.

Viene dada por la expresión 4 y se representa en la Figura 1c.

$$relu(z) = \text{máx}(0, z) \tag{4}$$

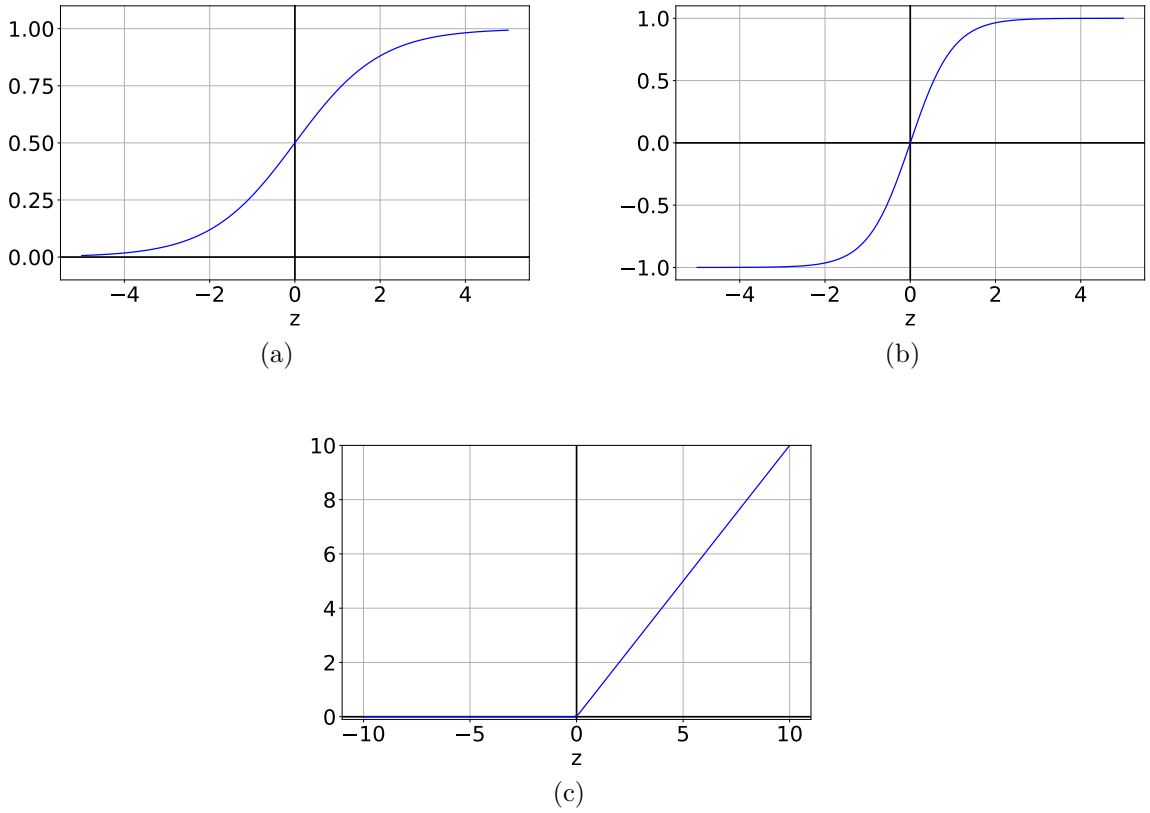


Figura 1: Representación gráfica de las funciones de activación. **(a)** Función sigmoide dada por la Ecuación 2 **(b)** Función tangente hiperbólica dada por la Ecuación 3 **(c)** Función ReLU dada por la Ecuación 4.

## 3.3. Arquitectura de una red neuronal

Una RN densa está compuesta por capas de neuronas. Una capa es un módulo de procesamiento de datos, podemos pensar que es una transformación o mapeo de datos en forma de vectores. A la primera capa, la denominamos capa de entrada y recibe un vector de datos de entrada, $\vec{x} \in \mathbb{R}^n$. A las capas intermedias las denominamos capas ocultas y a la última capa, capa de salida, que devuelve un vector $\vec{y} \in \mathbb{R}^k$, que serán las predicciones de la RN. En la Figura 2 se muestra la arquitectura de una RN densa, esto es, todas las neuronas de una capa están conectadas a todas las neuronas de la anterior. La RN de la Figura 2 presenta una capa de entrada con $n$ neuronas, dos capas ocultas con $m$ neuronas cada una y una capa de salida con $k$ neuronas.

Es importante destacar que la información de la RN fluye de izquierda a derecha. Estas RNs en las que el output de una capa sirve de input para la siguiente capa se denominan RNs de propagación directa.

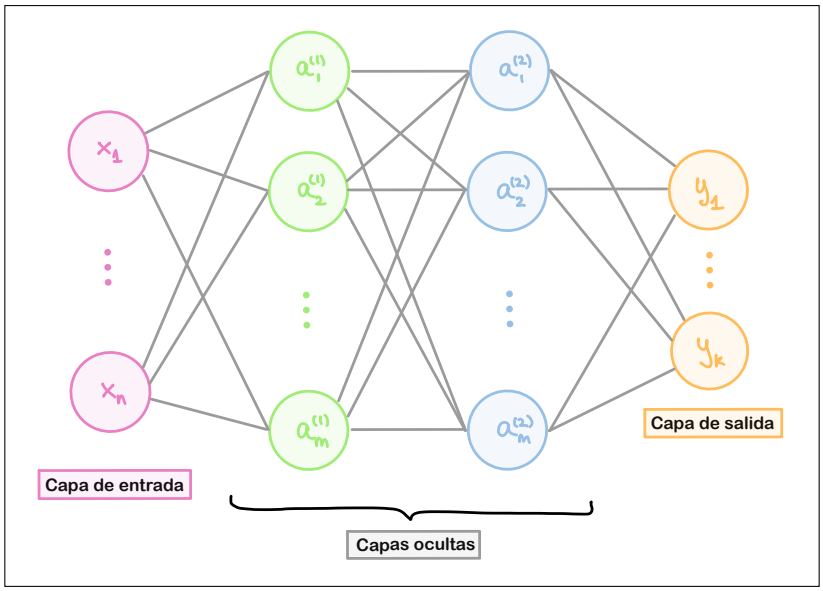


Figura 2: Esquema de una RN densa.

## 3.4. La función de coste

En el aprendizaje supervisado disponemos de un conjunto de pares de datos de entrenamiento $\{\vec{x}_i, \vec{y}_{ver}(\vec{x}_i)\}_{i=1}^{N}$. Por ello, es conveniente introducir una función que nos calcule el error entre el valor predicho por la RN y el valor verdadero para poder evaluar la eficiencia del entrenamiento. De esto se ocupa la función de coste. El objetivo del entrenamiento es minimizar esta función. Podría describirse como una medida del éxito de la tarea que tiene la RN de predecir los resultados.

Existen múltiples funciones de coste y en función del problema con el que estemos trabajando debemos escoger una función de coste u otra y esta tarea es extremadamente importante. Existen una serie de indicaciones a la hora de escoger esta función para los problemas más conocidos. Si estamos tratando con un problema de clasificación de dos clases es conveniente emplear la entropía cruzada binaria. Sin embargo, para un problema de clasificación de muchas clases conviene usar la entropía cruzada categórica. Para nuestro problema de regresión, emplearemos el error cuadrático medio (ECM).

$$C = ECM = \frac{1}{N}\sum_i |\vec{y}_{ver}(\vec{x_i}) - \vec{y}_{pred}(\vec{x_i})|^2 \tag{5}$$

donde $\vec{y}_{ver}$ son los valores verdaderos, $\vec{y}_{pred}$ los valores predichos por la red y N es el número de puntos empleado para calcular esta función.

Una métrica común en problemas de regresión es el error absoluto medio, EAM. Es el promedio del valor absoluto de la diferencia entre las predicciones y los valores verdaderos. Se empleará en tanto por ciento (EAM %). Viene definido por la siguiente ecuación:

$$EAM\,\% = \frac{1}{N}\sum_i \frac{|y_{pred} - y_{ver}|}{y_{ver}} \cdot 100 \tag{6}$$

## 3.5. Entrenamiento

Durante el proceso de entrenamiento se minimiza la función de coste, modificando los pesos y los sesgos. Para minimizar la función de coste, en cada iteración se calculan los gradientes de los pesos mediante el método de descenso del gradiente, en el que se actualizan los parámetros en dirección opuesta al gradiente (ya que el gradiente apunta hacia la dirección máxima y estamos buscando un mínimo). De esta manera se reduce el valor de la función de coste en cada iteración, hasta que alcance su mínimo. Para entender cómo opera el funcionamiento del algoritmo del descenso del gradiente, podemos compararlo con el descenso de una montaña hasta alcanzar el punto más bajo de la montaña. Es equivalente a observar si la pendiente de la montaña aumenta; si es así, avanzamos en dirección contraria para descender hacia el valle más cercano. Este algoritmo se ilustra en la Figura 3.

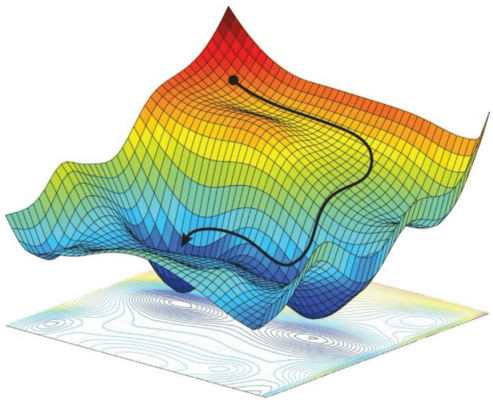

Figura 3: Ilustración del concepto descenso del gradiente (Fuente: [7]).

Matemáticamente, la actualización de los pesos y sesgos en cada iteración sigue las siguientes reglas,

$$w_{ji}^{l} \rightarrow w_{ji}^{l} - \eta \frac{\partial C}{\partial w_{ji}^{l}} \tag{7}$$

$$b_{j}^{l} \rightarrow b_{j}^{l} - \eta \frac{\partial C}{\partial b_{j}^{l}}, \tag{8}$$

donde $\eta$ es la tasa de aprendizaje (*learning rate*), $w_{ji}^{l}$ es el peso que conecta la neurona $i$-ésima de la capa $l-1$ con la neurona $j$-ésima de la capa $l$, $b_{j}^{l}$ es el sesgo de la neurona $j$-ésima en la capa $l$ y $C$ es la función de coste, que dependerá del problema que estemos resolviendo.

Definimos el conjunto de pesos y sesgos de todas las capas en un momento del entrenamiento particular como:

$$\theta = [W^{l}, b^{l}] \tag{9}$$

para 0<l<L, donde L es el número total de capas.
En este procedimiento estamos buscando un mínimo global. La tasa de aprendizaje puede entenderse como el tamaño de los pasos que damos hacia ese mínimo, por tanto con una tasa muy grande es posible que nos saltemos el mínimo y con una muy pequeña el método tardará mucho en encontrar ese mínimo.

Corremos el peligro de que el método se quede atascado en un mínimo local. El optimizador es quien se encarga de evitar esto, además de tener otras funciones. Un ejemplo de

optimizador eficaz es el Descenso de Gradiente Estocástico (DGE). Se basa en el método de descenso del gradiente pero el término estocástico significa que la actualización de los parámetros del modelo se realiza utilizando un lote o *mini-batch* del conjunto de datos. Se conoce como *mini-batch size* al tamaño de este lote y es un hiperparámetro más de nuestra RN que deberemos ajustar. Generalmente, el entrenamiento se divide en épocas de entrenamiento. Cada iteración sobre los datos de entrenamiento se denomina época. Una época es el conjunto de *N/mini-batch size* iteraciones (N es el número total de datos) que hace la red hasta que todos los datos de entrenamiento han sido utilizados.

De análisis matemático sabemos que una cadena de funciones puede derivarse empleando la regla de la cadena. Pues bien, aplicar la regla de la cadena para computar los gradientes de la red es lo que se conoce como el algoritmo de *Backpropagation*. Este algoritmo toma el valor final de la función de coste y realiza las derivadas desde la salida hasta la entrada de la RN, aplicando la regla de la cadena para calcular la contribución de cada parámetro en la función de coste. En resumen, es el descenso de gradiente empleando una técnica para calcular los gradientes automáticamente. Una vez que tiene estos gradientes, realiza un paso regular de descenso de gradiente y se repite de nuevo el proceso hasta que la red converja a la solución. El cálculo automático de los gradientes se conoce como Diferenciación Automática (DA). Existen distintas técnicas de DA. La que usa el algoritmo de *Backpropagation* se conoce como DA en modo reverso y es la solución que utiliza Tensorflow para calcular los gradientes de manera eficiente [8].

El DGE utiliza el siguiente algoritmo

1. Tomar aleatoriamente un lote de muestras de entrenamiento $\vec{x}_i$ (valores de entrada) y los correspondientes valores esperados que la red debería predecir $\vec{y}_{ver}(\vec{x}_i)$ .

2. Ejecutar la red sobre los datos $\vec{x}_i$ del lote, procesándolos a través de sus capas y produciendo las correspondientes salidas, $\vec{y}_{pred}(\vec{x}_i)$.

3. Calcular la pérdida de la red en el lote $\vec{x}_i$. Esto es, una medida de la discrepancia entre los valores verdaderos y los predichos por la red.

4. Calcular el gradiente de la pérdida con respecto a los parámetros de la red mediante el algoritmo de *Backpropagation*.

5. Mover los parámetros ligeramente en dirección opuesta al gradiente, reduciendo de esta manera el coste en el lote.

6. Seleccionar otro lote del conjunto de datos de entrenamiento y repetir el proceso. Cuando todos los datos hayan sido utilizados en lotes, diremos que se ha completado una época de entrenamiento.

Existen otros tipos de optimizadores con la tasa de aprendizaje adaptativa. Concretamente, el empleado en el trabajo se denomina Adam [9].

La Figura 4 resume el proceso de entrenamiento de una RN.

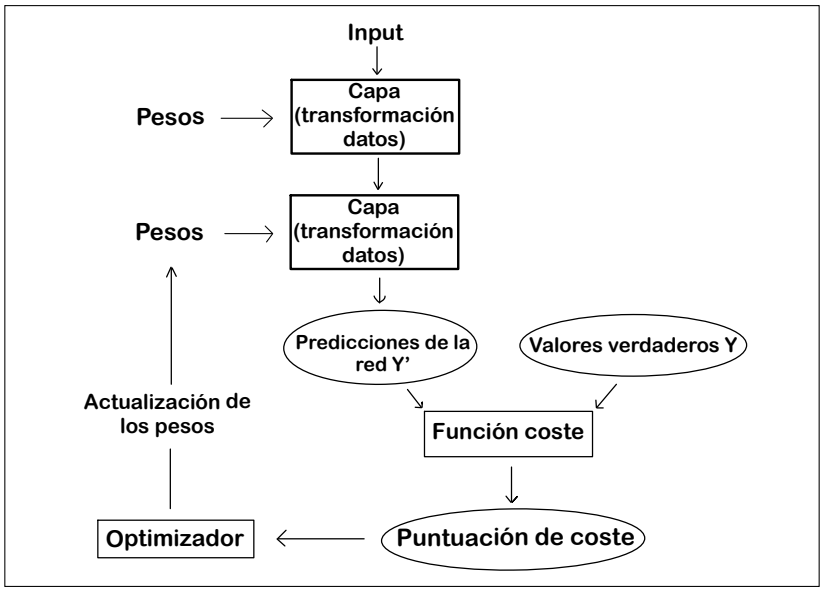


Figura 4: Esquema del funcionamiento de una RN.

## 3.6. Overfitting

El principal objetivo a la hora de entrenar una RN es que el modelo se ajuste a los datos de entrenamiento lo mejor posible y que, además, sea capaz de hacer predicciones correctas sobre datos que no han sido utilizados durante el entrenamiento. Esto se conoce como capacidad de generalización.

Para cuantificar cómo generaliza la red, se utilizan los llamados datos de validación. Son un subconjunto de los datos que se utilizan para evaluar el modelo después de cada época, es decir, se utilizan durante el entrenamiento para monitorizar el rendimiento del modelo. Estos datos no son utilizados para calcular los gradientes, no se utilizan para ajustar los pesos de la red, al contrario que los datos de entrenamiento. La importancia de utilizar sólo una parte de la base de datos para el entrenamiento y dejar otra parte de la misma para el proceso de validación se explica a continuación.

Al inicio del entrenamiento, optimización y generalización están estrechamente relacionadas: una reducción en la función de coste en los datos de entrenamiento, suele traducirse en una reducción similar en los datos de validación. Se dice entonces que el modelo está subentrenado, (*underfit*), lo que significa que existe un margen de mejora y el modelo necesita seguir aprendiendo. Sin embargo, después de un determinado número de iteraciones, la generalización deja de mejorar y las métricas de validación se estabilizan mientras que las métricas de entrenamiento continúan disminuyendo. Este fenómeno es uno de los principales problemas en las RNs, denominado sobreajuste (*overfitting*). Éste tiene lugar cuando la red se ajusta demasiado bien a los datos de entrenamiento, tanto que no será capaz de predecir buenos resultados para nuevos datos de entrenamiento, es decir, pierde la capacidad de generalización.

Una forma de evitar que el modelo aprenda patrones irrelevantes en los datos de entrenamiento es aumentar la cantidad de datos de entrenamiento disponibles. Sin embargo, esto no siempre es posible. Es posible prevenir y reducir el sobreajuste utilizando técnicas de regularización.

### 3.6.1. Técnicas de regularización

Una de las estrategias más simples para evitar el sobreajuste es reducir el tamaño de la RN. Determinar el número adecuado de capas y neuronas no tiene una respuesta definitiva. Lo que se suele hacer es empezar con una estructura sencilla, con pocas capas y parámetros e ir ajustándola según sea necesario durante el entrenamiento.

A medida que aumenta el número de parámetros, la RN adquiere mayor capacidad de representación de funciones. Esto se traduce en una mayor precisión en el ajuste y, por tanto, una función de coste baja. Sin embargo, si el número de parámetros es demasiado grande, esta misma capacidad también la hace más susceptible al sobreajuste, entrando en un régimen donde la diferencia entre la función de coste en los datos de entrenamiento y los de validación puede ser significativa. Teniendo esto en cuenta, es conveniente elegir un número de parámetros que sea suficientemente grande como para representar la función subyacente a los datos de entrenamiento, pero no lo suficiente como para permitir el sobreajuste.

Otra manera alternativa de mitigar el sobreajuste es utilizar técnicas de regularización de pesos. Estos métodos imponen restricciones a los valores de los pesos de la red, promoviendo que sean pequeños y distribuidos de manera más uniforme. Uno de los regularizadores más conocidos es el *dropout*. Esta técnica, propuesta por Geoffrey Hinton, consiste en apagar aleatoriamente un número de neuronas durante el entrenamiento de cada iteración [10].

Hinton describe la inspiración detrás del *dropout* al compararlo con la rotación aleatoria de cajeros en un banco, donde la rotación aleatoria evita posibles conspiraciones. Análogamente, el *dropout* introduce ruido en las salidas de las capas. Esto ayuda a evitar que la red memorice patrones insignificativos, lo que promueve una generalización más robusta al romper los patrones que podrían no ser relevantes para los datos de prueba.

En resumen, reducir el tamaño de la red y aplicar técnicas de regularización como el *dropout* son estrategias efectivas para evitar el sobreajuste.

# 4. Physics-Informed Neural Networks (PINNs)

En 2018, Raissi, Perdikaris y Karniadakis introdujeron el concepto de PINN [2]. Las PINNs podrían definirse como RNs que incluyen en la función de coste la ecuación diferencial y las condiciones iniciales o de contorno. Esto hace que las PINNs sean útiles a la hora de resolver una gran variedad de ecuaciones diferenciales, tanto ordinarias (EDOs) como ecuaciones en derivadas parciales (EDPs).

Una EDP es una relación de la forma

$$Z(\vec{x}, y(\vec{x}), \frac{\partial y}{\partial x_1}, ..., \frac{\partial y}{\partial x_n}, \frac{\partial^2 y}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 y}{\partial x_1 \partial x_n}, ...) = 0 \tag{10}$$

donde $\vec{x} = (x_1, ..., x_n)$ es un vector de n dimensiones definido en una región $\Omega \in \mathbb{R}^n$. Para describir completamente el proceso físico hace falta plantear el estado inicial del proceso (condiciones iniciales para problemas de evolución) y el régimen en la frontera $\partial\Omega$ de la región $\Omega$ donde tiene lugar el proceso (las condiciones de contorno).

En EDPs que modelan procesos dinámicos (en los que el tiempo es una de las variables independientes) hay que especificar una o más condiciones iniciales.

$y(\vec{x})$ es solución de la Ecuación 10 y satisface las condiciones de frontera, dadas por la Ecuación 11.

$$F(\vec{x}, y(\vec{x})) = 0 \quad \vec{x} \in \partial\Omega \tag{11}$$

Para una EDP, la capa de entrada se corresponde con $\vec{x}$ (n neuronas de entrada) y la salida de la red representa la solución $y(\vec{x})$ (1 neurona de salida). Cabe destacar que esta explicación se puede generalizar a ecuaciones diferenciales con solución vectorial, es decir, con varias neuronas en la salida, $\vec{y}(\vec{x})$.

Sea $N_{CI}$ un conjunto de puntos, denotados como $\Gamma_{CI}$ tales que $\Gamma_{CI} \in \partial\Omega$ y sea $N_{EDO}$ otro conjunto de puntos, denotados como $\Gamma_{EDO}$ tales que $\Gamma_{EDO} \in int(\Omega)$, donde $int(\Omega)$ denota el interior de $\Omega$.

Como hemos mencionado, la función de coste incluirá dos términos, uno debido a la EDP ($C_{EDO}$) y otro debido a las condiciones frontera ($C_{CI}$). Definimos pues la función de coste como la suma pesada de dos términos:

$$C_{PINN} = w_{EDO} \cdot C_{EDO} + w_{CI} \cdot C_{CI} \tag{12}$$

donde las funciones de coste son funciones del conjunto de puntos escogido y de los pesos y sesgos de las capas ocultas, es decir, $C = C(\theta, \Gamma), C_{EDO} = C_{EDO}(\theta, \Gamma_{EDO}), C_{CI} = C_{CI}(\theta, \Gamma_{CI})$, $\Gamma = \Gamma_{EDO} + \Gamma_{CI}$ y $\theta$ se define como en la Ecuación 9. En lo que sigue, se omiten las dependencias por facilitar notación.

Los hiperparámetros $w_{EDO}$ y $w_{CI}$ que aparecen en la Ecuación 12 se deben ajustar para optimizar el funcionamiento de la red. En nuestro trabajo, se tomará $w_{EDO} = w_{CI} = 1$.

Definimos las funciones de coste como sigue:

$$C_{EDO} = \frac{1}{N_{EDO}} \sum_{\vec{x} \in \Gamma_{EDO}} \left| Z\left(\vec{x}, y_{pred}(\vec{x}), \frac{\partial y_{pred}}{\partial x_1}, ..., \frac{\partial y_{pred}}{\partial x_n}, \frac{\partial^2 y_{pred}}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 y_{pred}}{\partial x_1 \partial x_n}, ...\right)\right|^2 \tag{13}$$

y

$$C_{CI} = \frac{1}{N_{CI}} \sum_{\vec{x} \in \Gamma_{CI}} |F(y_{pred}, \vec{x})|^2 \tag{14}$$

La función de coste se minimizará de la misma manera que se ha explicado anteriormente. Un valor de la función de coste de la ecuación diferencial ($C_{EDO}$) cercano a 0 indicará que las predicciones de la red satisfacen la EDP en el conjunto de puntos que hayamos escogido, $\Gamma_{EDO}$. Análogamente, un valor de la función de coste de las condiciones frontera $C_{CI}$ cercano a 0 indicará que se satisfacen las condiciones de frontera por las predicciones de la red en los puntos del conjunto $\Gamma_{CI}$.

Estas funciones siguen siendo una medida del éxito de la tarea que tiene la red de predecir los resultados pero cada una en "lugares" distintos.

# 5. Resultados

En esta sección, se mostrarán los resultados obtenidos para distintas EDOs. Comenzaremos mostrando un caso sencillo y posteriormente, ilustraremos dos ejemplos de ecuaciones dinámicas.

A lo largo del trabajo, se denotará al tiempo como x, en lugar de t, la forma habitual, por coherencia con la notación utilizada en el trabajo para RNs.

## 5.1. Ecuación de movimiento constante

Para comprobar el funcionamiento correcto de la PINN, es conveniente entrenarla con un caso sencillo. El sistema de EDOs que tratamos de resolver es el siguiente:

$$\begin{cases} \dfrac{dy_1}{dx} = v_1 \quad \text{con } y_1(0) = 0 \\ \dfrac{dy_2}{dx} = v_2 \quad \text{con } y_2(0) = 0 \\ \dfrac{dy_3}{dx} = v_3 \quad \text{con } y_3(0) = 0 \end{cases} \tag{15}$$

donde $v_1, v_2$ y $v_3$ son constantes. Se ha tomado $v_1 = 2, v_2 = 3$ y $v_3 = 5$.

La configuración de la red con la que trabajaremos es la presentada en la Tabla 1.

| Hiperparámetro | Valor |
|---|---|
| *nº de datos de entrenamiento* | 50 |
| *mini-batch size* | 5 |
| Función de activación | elu |
| nº neuronas por capa | 1,50,50,50,3 |
| Optimizador | Adam |
| *Tasa de aprendizaje* | 0.001 |

Tabla 1: Hiperparámetros para resolver el sistema de ecuaciones15.

En este caso, la capa de entrada tiene 1 neurona, las 3 capas densas ocultas tienen 50 neuronas cada una y la capa de salida tiene 3 neuronas.

En la Figura 5a se presenta la solución analítica y la solución predicha por la red. En la Figura 5b se muestra la función de coste total, es decir, la suma de la función de coste de la EDO ($C_{EDO}$) y la función de coste de las condiciones iniciales ($C_{CI}$). La convergencia de las pérdidas es un buen método para indicar que el entrenamiento de la red ha finalizado. En el caso de no converger, se deberían aumentar las épocas de entrenamiento. En este caso, el entrenamiento se ha realizado con 100 épocas.

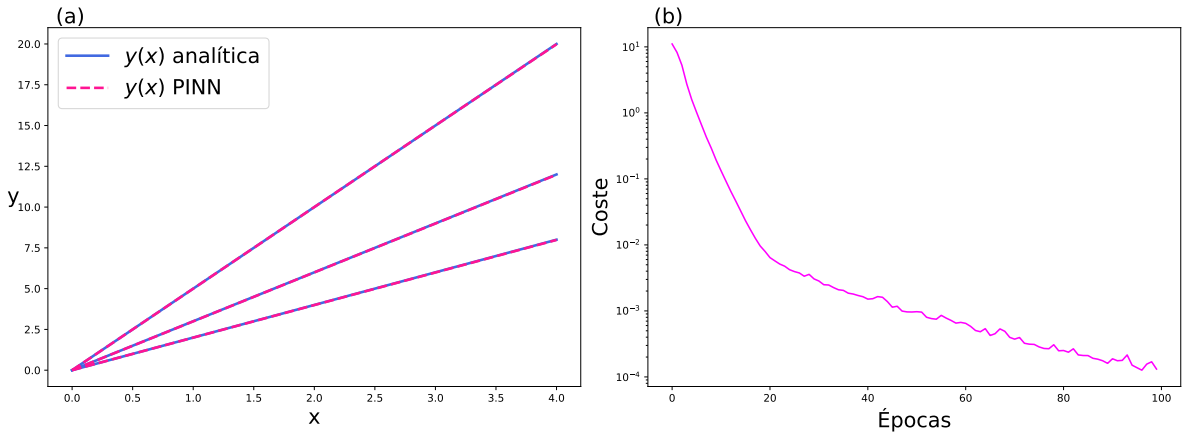


Figura 5: **(a)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas). **(b)** Evolución de la función coste con el número de épocas (escala logarítmica). La función de coste se corresponde con la suma de la contribución de la EDO y las condiciones iniciales (ver ecuación 12).

Se aprecia claramente cómo la solución analítica coincide con la predicha por la red, con un EAM % del 0.32 %. El tiempo de ejecución ha sido de 3.12 segundos.

Hemos comprobado que nuestra red entrena bien para los hiperparámetros mostrados en la Tabla 1 pero, ¿a qué se debe la elección de estos parámetros? Pues bien, la elección de estos parámetros ha sido realizada a base de prueba y error. No existe ninguna fórmula matemática que nos devuelva la receta para que la PINN funcione de manera óptima.

Podríamos pensar que si aumentamos el número de puntos de entrenamiento la PINN presentaría un comportamiento mejor. Fijando el doble de puntos de entrenamiento ($n^o$ *de puntos de entrenamiento = 100*) y realizando la misma simulación, obtenemos prácticamente el mismo error, 0.39 % y un tiempo de ejecución mayor, 4.56 segundos. Sin embargo, si disminuimos el número de puntos de entrenamiento a la mitad ($n^o$ *puntos de entrenamiento = 25*), obtenemos un tiempo de ejecución menor, 2.72 segundos pero un error del 1.79 %.

Concluimos que debe existir un valor óptimo para este parámetro y que tener más puntos de entrenamiento no implica un mejor funcionamiento de la PINN.

### 5.1.1. Extrapolación de la PINN con funciones lineales

En este apartado, queremos comprobar si la PINN es capaz de extrapolar o predecir el comportamiento de las ecuaciones en un dominio mayor. Entrenando la red en el intervalo $x \in (0, 4)$ y representando el dominio temporal para $x \in (-9, 9)$ obtenemos la Figura 6a. Se observa cómo la red es capaz de predecir con bastante exactitud en el intervalo $x \in (4, 9)$ y cómo devuelve la solución nula para $x \in (-9, 0)$. Esto no debería sorprendernos ya que la red no tiene información sobre el comportamiento de la función para intervalos de x negativos. Si entrenamos la red en el intervalo $x \in (-4, 4)$ y realizamos la extrapolación a $x \in (-9, 9)$ obtenemos la Figura 6b, que muestra un comportamiento simétrico.

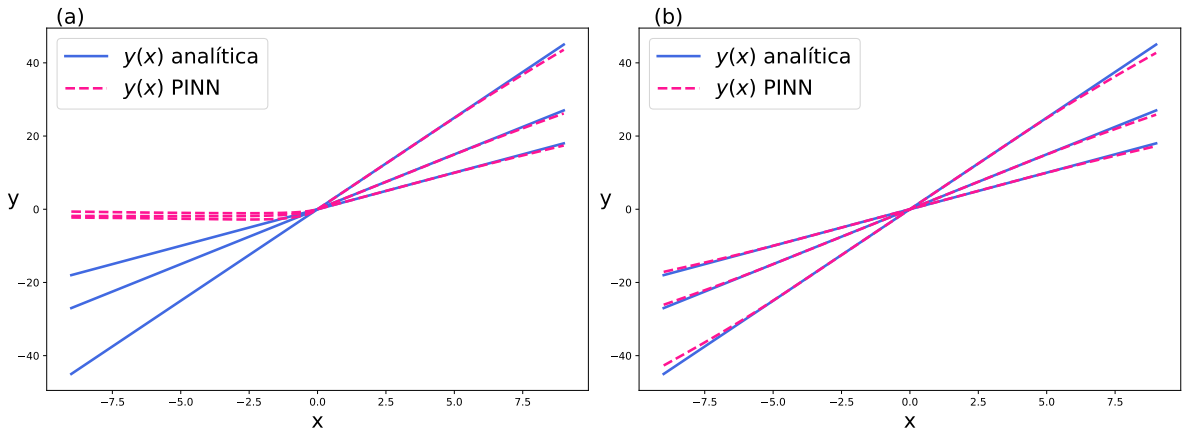


Figura 6: **(a)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) entrenando en el rango de $x \in (0, 4)$ y extrapolando al rango de $x \in (-9, 9)$. **(b)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) entrenando en el rango de $x \in (-4, 4)$ y extrapolando al rango de $x \in (-9, 9)$..

En la Figura 6a el error es del 41.7 % (con respecto al exacto) y en la Figura 6b obtenemos un EAM % del 0.19 % .

## 5.2. Modelo de Romeo y Julieta

El modelo de Romeo y Julieta es un sistema de ecuaciones diferenciales que simboliza de manera matemática la dinámica emocional entre dos individuos. A continuación, trataremos de encontrar una solución para el sistema de EDOs de Romeo y Julieta, que viene dado por el siguiente sistema de ecuaciones.

$$\begin{cases} \dfrac{dR}{dx} = -m \cdot J & \text{con } R(0) = 1 \\ \\ \dfrac{dJ}{dx} = n \cdot R & \text{con } J(0) = 0 \end{cases} \tag{16}$$

donde R(x) representa el amor (u odio si es negativo) de Romeo por Julieta a tiempo x y J(x) representa el amor (u odio si es negativo) de Julieta por Romeo a tiempo x.

Los parámetros m y n son constantes arbitrarias. Para nuestro estudio tomaremos m=3 y n=5.

Si nos fijamos en las ecuaciones, es un sistema de amor-odio. Cuanto más quiere Julieta a Romeo, más decrece el amor de Romeo. Cuanto más quiere Romeo a Julieta, más le quiere Julieta. Por tanto, la solución es cíclica: una elipse en el plano R-J. Esto causará una serie de problemas que trataremos de solucionar posteriormente.

En la Tabla 2 se muestran los hiperparámetros que emplearemos en nuestra PINN para resolver el sistema de ecuaciones 16. Una ventaja de las PINNs es que, con relativa facilidad, es posible resolver multitud de ecuaciones diferenciales, tan sólo modificando unas líneas de código. En este ejemplo, sólo se han modificado las líneas de código donde se define la función de coste asociada a la EDO. Los hiperparámetros utilizados son prácticamente los mismos que en la Tabla 1 salvo el número de puntos de entrenamiento que se ha aumentado en un factor 10 y en este caso, el número de neuronas de entrada sigue siendo 1 pero ahora el número de neuronas de salida es 2.

| **Hiperparámetro** | **Valor** |
|---|---|
| *$n^o$ de puntos de entrenamiento* | 500 |
| *mini-batch size* | 64 |
| Función de activación | elu |
| n$^o$ neuronas por capa | 1,50,50,50,2 |
| Optimizador | Adam |
| *Tasa de aprendizaje* | 0.001 |

Tabla 2: Hiperparámetros para resolver el sistema de ecuaciones 16.

En la Figura 7a se presenta la predicción de la red y la solución analítica para las dos componentes en un intervalo temporal $x \in (0, 2)$.

En la Figura 7b se representa la solución analítica y la predicha por la red en el plano R-J, que como hemos dicho anteriormente, es una elipse. El entrenamiento se ha realizado con 1000 épocas.

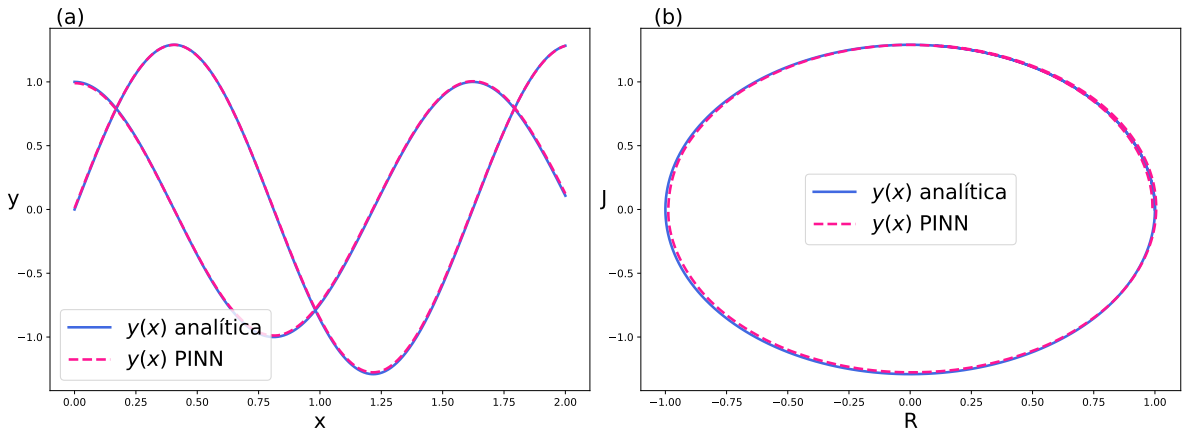


Figura 7: **(a)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) por componentes. **(b)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) en el plano R-J.

En este caso obtenemos un EAM % del 0.77 % que es considerablemente bajo.

Representamos en la Figura 8 la función coste total para comprobar la convergencia del método. Se trata de la suma de la función de coste de la ecuación diferencial y la función de coste de las condiciones iniciales. En este caso, el método ha requerido 1000 épocas para converger, mientras que en el caso de la ecuación de movimiento constante, la convergencia se logró en solo 100 épocas.

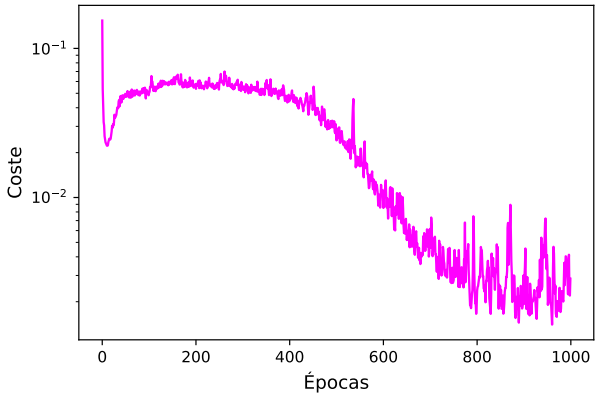


Figura 8: Evolución de la función coste con el número de épocas (escala logarítmica) para el sistema de ecuaciones 16, en el rango $x \in (0, 2)$. La función de coste se corresponde con la suma de la contribución de la EDO y las condiciones iniciales (ver ecuación 12).

### 5.2.1. Problemas asociados al dominio de entrenamiento

Las PINNs pueden presentar problemas cuando el dominio de entrenamiento crece. Este hecho lo podemos observar en la Figura 9, donde se ha aumentado el dominio temporal a x tales que $0 < x < 4$. También se ha aumentado el número de épocas a 3000 para asegurar la convergencia del cálculo. En la Figura 9a Se observa que la solución predicha por la PINN sigue la tendencia oscilatoria pero decae a 0 y no coincide con la solución analítica a pesar de que el método haya convergido.

Una forma de mitigar el problema es eligiendo las funciones de activación adecuadas. En la Figura 9a se ha empleado la función de activación *elu* y en la Figura 9b se ha empleado la función de activación *tanh*, manteniendo el resto de hiperparámetros iguales. En este caso, cambiar de la función de activación *elu* a la *tanh* corrige el problema en el intervalo elegido, como se ve en la Figura 9. La función *tanh* está acotada mientras que la *elu* no lo está. Por tanto, para el caso de las constantes de movimiento la función *elu* era adecuada, sin embargo, para funciones oscilantes, que están acotadas, conviene utilizar una función de activación que no tienda a infinito.

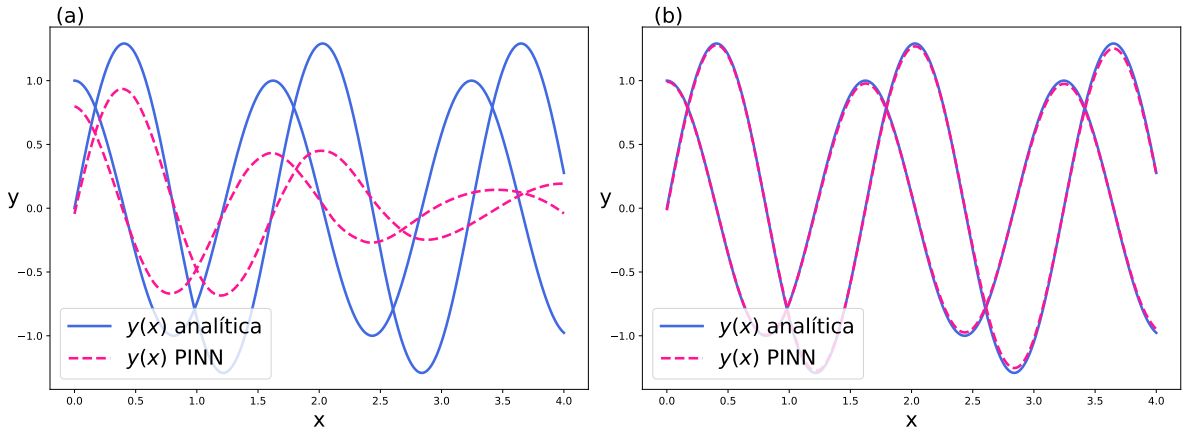


Figura 9: **(a)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) empleando la función de activación *elu*, en el rango $x \in (0, 4)$ **(b)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) empleando la función de activación *tanh*, en el rango $x \in (0, 4)$.

Una vez comprobado que la red entrena adecuadamente, podemos realizar una serie de cambios. Parece que para valores de x pequeños o cercanos a las condiciones iniciales, la predicción de la red muestra un comportamiento que coincide con la solución analítica. Sin embargo, cuando el rango de valores de x aumenta todavía más, el problema vuelve a aparecer como se ve en la Figura 10 incluso si se aumentan las épocas de entrenamiento.

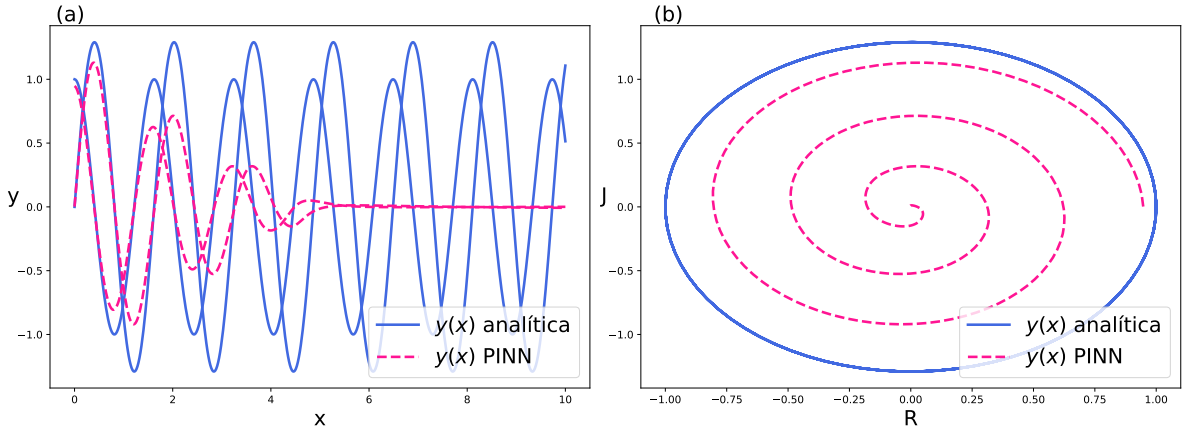


Figura 10: **(a)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) por componentes. **(b)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) en el plano R-J.

Observamos que, a pesar de que la PINN muestra un comportamiento oscilatorio, con el paso del tiempo la solución decae a 0. La hipótesis es que la PINN encuentra una solución a las ecuaciones pero para distintas condiciones iniciales en cada punto. Esto es típico con funciones oscilantes. La razón por la que la PINN obtiene esta solución a pesar de que la función de coste de la EDO es muy baja es la siguiente: localmente (en los puntos de entrenamiento), la red cumple bastante bien la EDO ya que, aunque tiene menos amplitud, la solución sigue oscilando. Esta amplitud va decayendo hasta que alcanza la solución y=0, la solución trivial, que también cumple la EDO, haciendo que la función de coste de la EDO siga siendo muy baja. Representamos en la Figura 11 la función de coste de la EDO y la función de coste de las condiciones iniciales, que dada la tendencia y los valores obtenidos producen la falta impresión de que la predicción de la PINN es correcta.

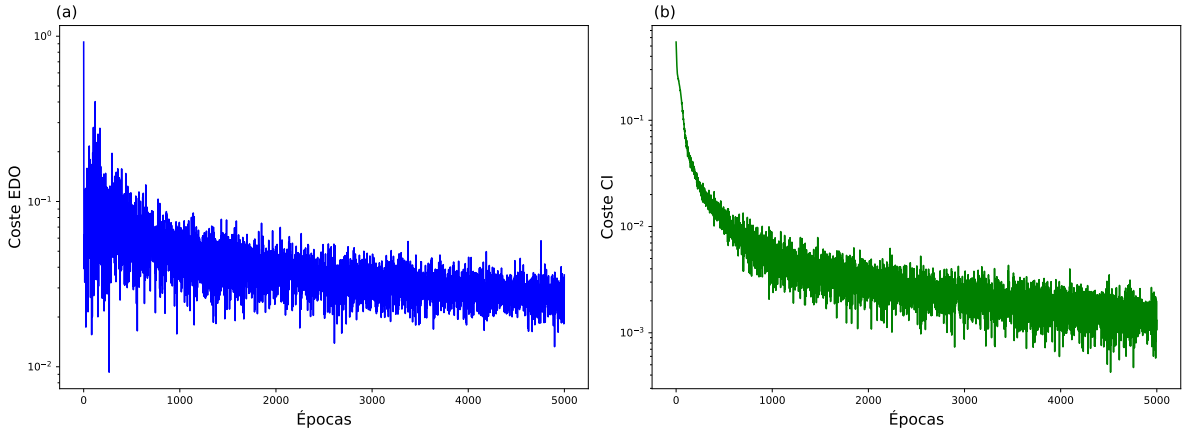


Figura 11: **(a)** Evolución de la función coste de la EDO con el número de épocas (escala logarítmica) (ver ecuación 13). **(b)** Evolución de la función coste de las CI con el número de épocas (escala logarítmica) (ver ecuación 14)

Podemos entonces concluir que la red entrena bien para intervalos pequeños de x, cercanos a la condición inicial. Cuando aumentamos el intervalo, la solución decae a 0. La función nula es solución de la EDO y aunque no cumpla las condiciones iniciales, la red es la única solución que encuentra.

### 5.2.2. Entrenamiento por subintervalos

Con el objetivo de mejorar el entrenamiento de la PINN para intervalos mayores y lejanos a las condiciones iniciales proponemos en este trabajo un algoritmo iterativo que consiste en dividir el dominio temporal en n subintervalos. Entonces, se implementa un bucle en el que entrenamos la red en el primer subintervalo y fijamos el último punto de entrenamiento como si fuera una condición de contorno. Estas nuevas condiciones de contorno son dinámicas y se generan con las predicciones de la PINN entrenada con los intervalos anteriores. Por tanto, las condiciones de contorno dinámicas (CCD) son pares de datos (x,$\vec{y}$) que se añaden a la función de coste de las condiciones iniciales ($C_{CI}$).

Posteriormente, entrenamos la red en el primer y segundo intervalo con la primera CCD fija, es decir, obligamos a la predicción de la red a pasar por ese punto. Y así sucesivamente hasta, en la última iteración, la red se entrena en todos los puntos del dominio, que es la unión de todos los subintervalos, forzándola a pasar por todas las CCD. Es decir, estamos realizando un entrenamiento acumulativo y forzando a la PINN a pasar por unos puntos que se asumen son solución de la EDO.
Introducimos pues un nuevo parámetro, el número de subintervalos, *n sub*. El código empleado se presenta en el Anexo A. Fijando 10 subintervalos y entrenando la red con los mismos parámetros que en la Tabla 2 y 5000 épocas, obtenemos la Figura 12. Se muestran las CCD generadas durante el entrenamiento de la PINN mediante símbolos.

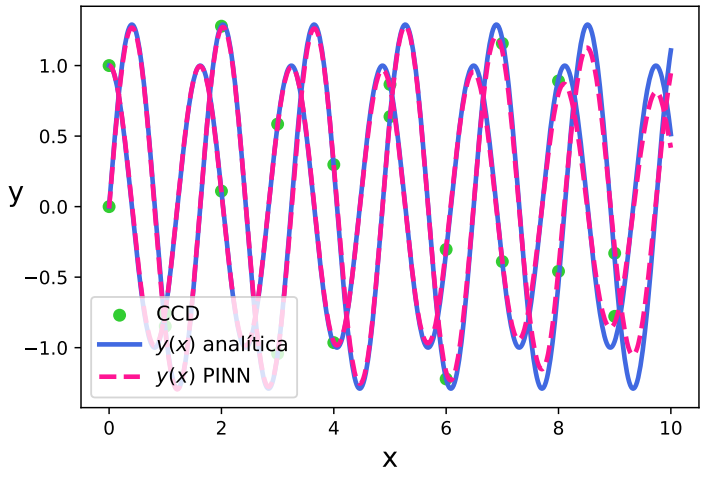


Figura 12: Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) junto a las CCD (círculos verdes) generadas en el entrenamiento. El entrenamiento se ha realizado en el rango $x \in (0, 10)$ utilizando 10 subintervalos y 5000 épocas.

Realizando este procedimiento de entrenar por subintervalos, conseguimos reducir el EAM % al 5.99 %. Cabe destacar también que para que la predicción de la red pase por todas las CCD, se ha de aumentar el número de épocas como se ha comentado anteriormente, con el consiguiente aumento del tiempo de ejecución un factor 10.

Al estar entrenando por subintervalos, conviene conocer las pérdidas de cada subintervalo. En la Figura 13a se muestra la función de coste de la ecuación diferencial para cada subintervalo y en la Figura 13b se muestra la función de coste de las condiciones iniciales para cada subintervalo. Cabe destacar que en el subintervalo 2 se incluye también el 1, en el subinervalo 3 se incluyen el 1 y el 2 y así sucesivamente. Como hemos dicho, el entrenamiento es acumulativo, y en el último subintervalo se entrena la red en todos los puntos del dominio.

Se observa en la Figura 13 que la función coste decrece mucho en el primer subintervalo, y en cada iteración este decrecimiento es menor. Esto podría ser por lo descrito en el párrafo anterior. En cada iteración, la PINN es entrenada en un dominio mayor, y por tanto la función de coste es mayor.

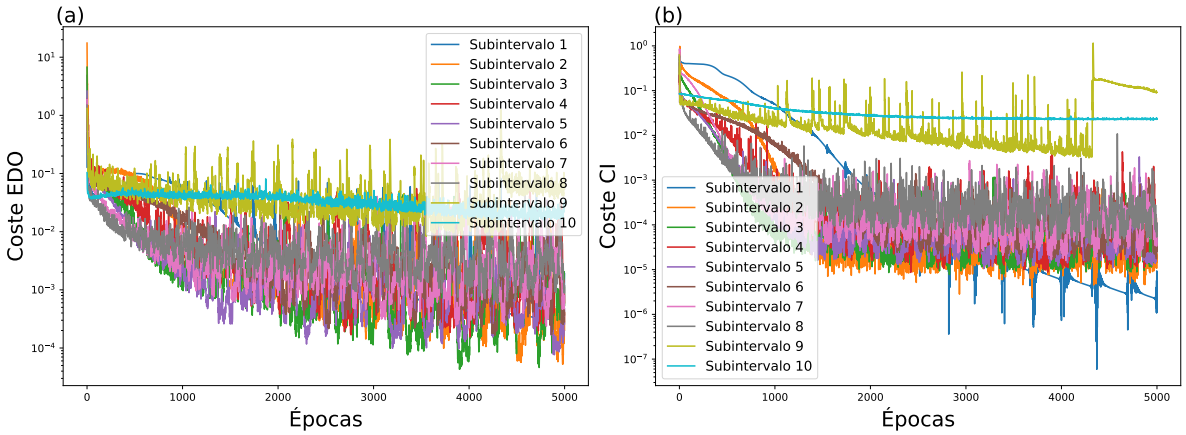


Figura 13: **(a)** Evolución de la función coste de la EDO con el número de épocas (escala logarítmica) por subintervalo (ver ecuación 13). **(b)** Evolución de la función coste de las CI con el número de épocas (escala logarítmica) por subintervalo (ver ecuación 14)

### 5.2.3. Extrapolación con funciones oscilantes

A continuación, entrenaremos la red en el intervalo $x \in (0, 2)$ y representamos la extrapolación de la predicción de la red en la Figura 14 al igual que se ha hecho en el apartado 5.1.1. Para el caso de las constantes de movimiento, la red era capaz de predecir el comportamiento de la función en un entorno del intervalo de entrenamiento. Notamos de la Figura 14 que en este caso, la red no es capaz de hacerlo. Esto se debe a que estamos trabajando con una función oscilante que tiene patrones más complejos que las funciones lineales. Los datos de entrenamiento no cubren suficientes periodos de oscilación y la red no es capaz de aprender estos patrones de oscilación.

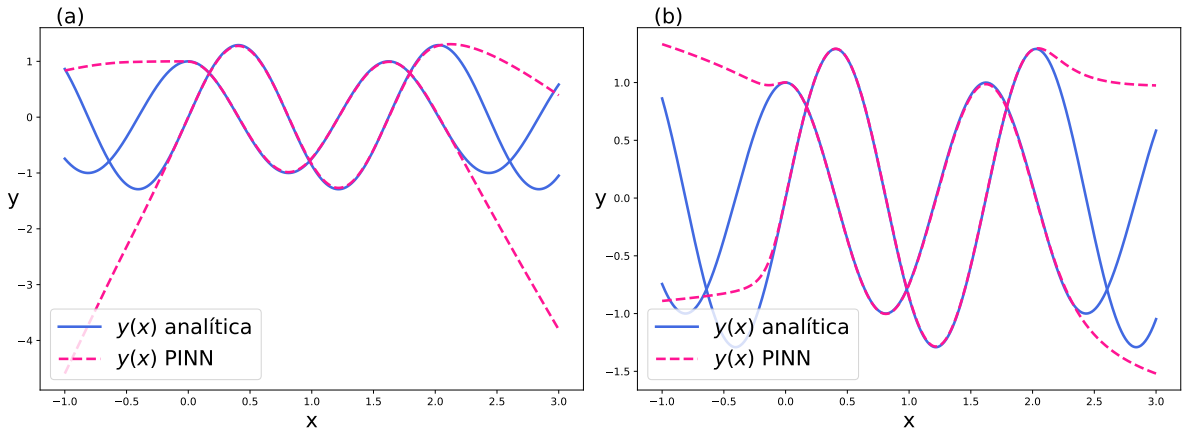


Figura 14: **(a)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) entrenando en el rango de $x \in (0, 2)$ y extrapolando al rango de $x \in (-1, 3)$ empleando la función de activación *elu*. **(b)** Comparación entre la solución analítica (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) entrenando en el rango de $x \in (0, 4)$ y extrapolando al rango de $x \in (-1, 3)$ empleando la función de activación *tanh*.

## 5.3. Ecuaciones de Lorenz

El sistema de Lorenz es el comienzo de la rama de las matemáticas y la física aplicada conocida como teoría del caos. Poincaré había afirmado que pequeñas variaciones en las condiciones iniciales de ciertos sistemas pueden suponer grandes cambios en la evolución a lo largo del tiempo del sistema [11]. Un claro ejemplo de este tipo de comportamientos se da en la evolución del tiempo atmosférico.

En 1963, Edward Lorenz estudiando la convección atmosférica, se encontró con un sistema de ecuaciones diferenciales en tres dimensiones que mostraban este comportamiento caótico. Para ciertos valores en los parámetros del sistema, las trayectorias tendían a formar una figura que hoy es conocida como atractor de Lorenz. Un atractor es una región del espacio de fases hacia la cual convergen las trayectorias posibles de un sistema. Además, corroboró la afirmación de Poincaré, ya que dicho sistema era extremadamente sensible a las condiciones iniciales. La contribución de Lorenz no se reduce únicamente a descubrir el comportamiento caótico de las ecuaciones, sino que reconoció un cierto orden en el caos. En su investigación mostró cómo un sistema determinista podía generar un comportamiento caótico.

Las ecuaciones de Lorenz vienen dadas por las ecuaciones siguientes.

$$\begin{cases} \dfrac{dy_1}{dx} = a \cdot (y_2 - y_1) & \text{con } y_1(0) = 1 \\ \dfrac{dy_2}{dx} = y_1 \cdot (b - y_3) - y_2 & \text{con } y_2(0) = 1 \\ \dfrac{dy_3}{dx} = y_1 y_2 - c y_2 & \text{con } y_3(0) = 1 \end{cases} \tag{17}$$

donde $a$ es el número de Prandtl y $b$ es el número de Rayleigh. Los tres parámetros $a$, $b$ y $c$ son estrictamente positivos. Normalmente se toma $a$=10, $c$=8/3 y $b$ variable. El sistema de Lorenz muestra un comportamiento caótico para $b$=28.

En la Figura 15 se muestra la solución al sistema de ecuaciones de Lorenz para los parámetros mencionados calculada mediante el método numérico de Runge-Kutta. Se ha calculado para $x \in (0, 40)$

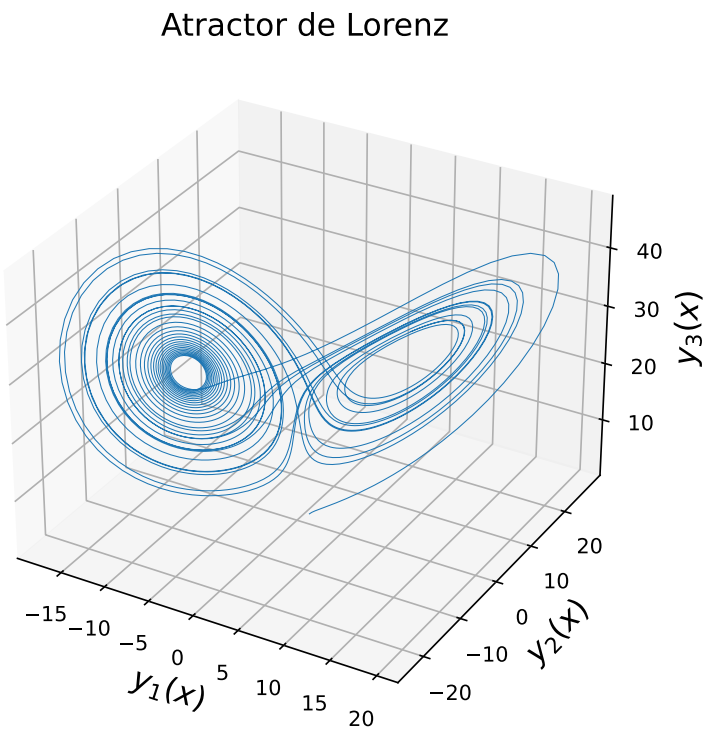


Figura 15: Representación gráfica del Atractor de Lorenz

A continuación, se describe el proceso que permite resolver la ecuación de Lorenz mediante el uso de la PINN construida. Los hiperparámetros de nuestra red vienen dados esencialmente por los de la Tabla 2. Lo único que varía es el número de neuronas de la capa de salida, que en este caso son 3. Nuevamente, destacamos la ventaja de la versatilidad de las PINN.

Realizaremos el entrenamiento de la PINN por subintervalos como se ha explicado en el apartado 5.2.2. Fijamos un dominio temporal x tales que 0<x<1.3. En este punto, el comportamiento de atractor ya ha comenzado y es de esperar que la PINN necesite muchas CCD para lograr imitar el comportamiento del atractor. Se ha realizado la simulación con 20 subintervalos y 10000 épocas para que el método converja y la PINN pase por todas las CCD generadas durante el entrenamiento de la PINN.

En la Figura 16 se muestra la solución predicha por la PINN y la obtenida mediante el método numérico Runge-Kutta para las tres componentes del sistema.

El tiempo de ejecución ha sido de 45 minutos. Reducir los tiempos de simulación constituye un desafío crucial para las PINNs, al igual que lo es en el resto de métodos computacionales. En los ejemplos descritos en este trabajo, el algoritmo de Runge-Kutta generalmente es más rápido, dado que las EDOs involucradas no son computacionalmente exigentes. No obstante, esta investigación en PINNs y el método de las CCD pueden ser relevantes en sistemas de ecuaciones parciales, donde los métodos clásicos suelen ser notablemente más lentos.

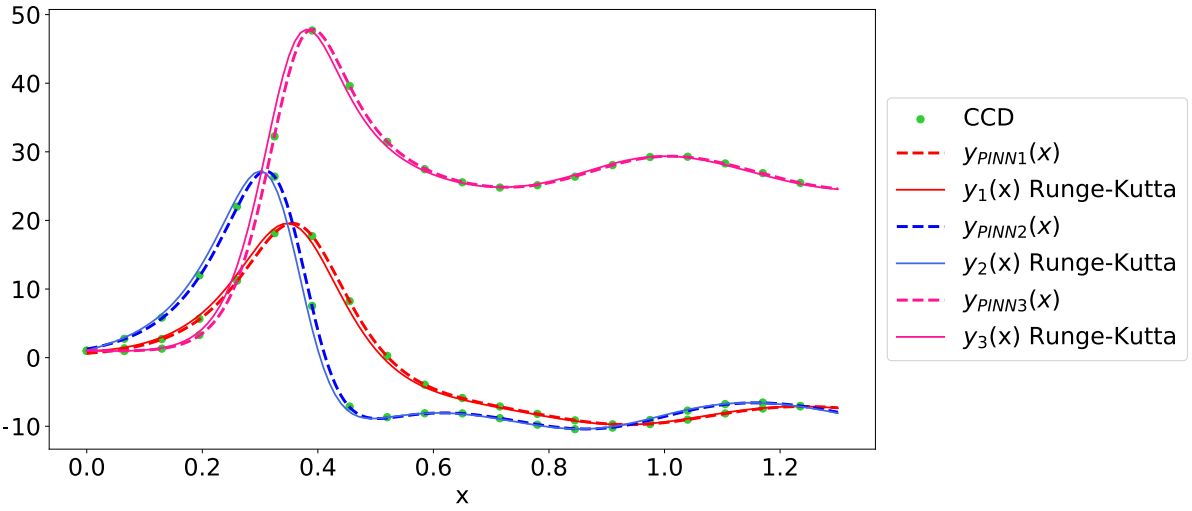


Figura 16: Comparación entre la solución obtenida mediante el método Runge-Kutta (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) junto a las CCD (círculos verdes) generadas en el entrenamiento. El entrenamiento se ha realizado en el rango $x \in (0, 1{,}3)$ utilizando 20 subintervalos y 10000 épocas.

En la Figura 17 se muestra la predicción de la PINN junto a la solución calculada mediante el método numérico de Runge-Kutta en tres dimensiones.

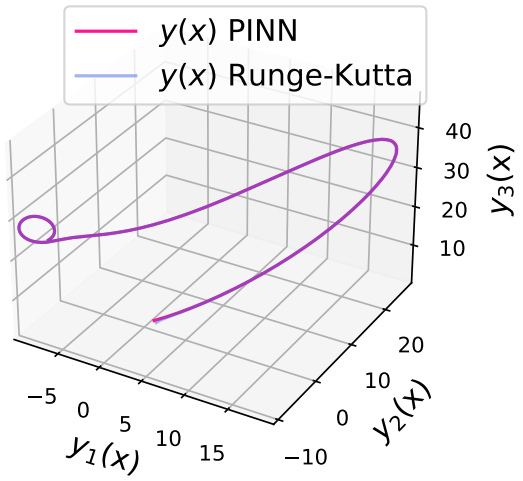


Figura 17: Comparación entre la solución obtenida mediante el método Runge-Kutta (líneas continuas) y los resultados predichos por la PINN (líneas discontinuas) en 3D

# 6. Conclusiones

En este trabajo se ha estudiado la evolución de sistemas dinámicos mediante el uso de RNs, que resultan muy útiles en la resolución de EDOs, fundamentales en la física.

Las PINNs presentan grandes ventajas en comparación con los métodos numéricos tradicionales como se ha visto en este trabajo. Por ejemplo, destacamos su versatilidad. En el transcurso de este trabajo se ha desarrollado una PINN capaz de abordar tanto ecuaciones lineales como oscilantes en dominios pequeños y cercanos a las condiciones iniciales. Además, se ha implementado el método de las CCD para que la misma PINN fuera capaz de resolver las ecuaciones en un dominio temporal mayor.

El método propuesto ha demostrado ser efectivo, especialmente en la resolución del sistema de ecuaciones de Lorenz. Sin embargo, se reconoce la necesidad de optimizar el tiempo de ejecución. Esta optimización constituye un área de desarrollo para futuras investigaciones.

En el trabajo también se ha visto la capacidad de extrapolación de las PINNs para sistemas de ecuaciones lineales. Sin embargo, no son capaces de predecir buenos resultados fuera del rango de entrenamiento para funciones oscilantes.

En conclusión, las PINNs pueden ser una herramienta poderosa en la resolución de EDOs en física gracias a su capacidad de generalización, flexibilidad y a su eficiencia computacional. Facilitan la resolución de sistemas complejos y es un campo que permite innovar y desarrollar el campo de la física aplicada y experimental.

# 7. Bibliografía

# Referencias


[1] François Chollet. *Deep Learning with Python*. Manning Publications, 2018.

[2] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[3] Luis Medrano Navarro et al. Solving differential equations with deep learning: a beginner's guide. *European Journal of Physics*, arXiv:2307.11237v1, 2023.

[4] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20:130–141, 1963.

[5] Steven H. Strogatz. Love affairs and differential equations. *Mathematics Magazine*, 61:35–42, 1988.

[6] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[7] Ángel Sánchez Ruiz. Descenso de gradiente estocástico. 2019.

[8] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow*. O'Reilly Media, 2017.

[9] Jimmy Lei Ba and Diederik P. Kingma. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015. Published as a conference paper at ICLR, arXiv:1412.6980v9.

[10] Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, and Nitish Srivastava. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[11] H. Poincaré. *Les Méthodes nouvelles de la Mécanique Céleste*. Gauthier-Villars et Fils, Imprimeurs-Libraires, 1892.

[12] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

# A. Anexo A: Código

Listing 1: Código en Python para resolver el sistema de ecuaciones de Romeo y Julieta entrenando la red por subintervalos.

```
1
2  # In[1]:#THE PINN CLASS
3
4  import tensorflow as tf
5  from tensorflow.keras.layers import Input, Dense
6  from tensorflow.keras.optimizers import Adam
7  import numpy as np
8  import matplotlib.pyplot as plt
9  import keras
10 import time
11
12 start_time = time.time()
13
14 m = tf.constant(3.0, dtype=tf.float32)
15 n = tf.constant(5.0, dtype=tf.float32)
16
17 # defino la PINN
18
19 class ODE_1st(tf.keras.Model):
20
21     def __init__(self, x0, y0_exact, *args, **kwargs):
22         super().__init__(*args, **kwargs)
23         self.loss_tracker = keras.metrics.Mean(name="loss")
24         self.loss_ode_tracker = keras.metrics.Mean(name="loss_ode")
25         self.loss_boundary_tracker = keras.metrics.Mean(name="
               loss_boundary")
26         self.x0 = x0
27         self.y0_exact = y0_exact
28
29
30     def train_step(self, data):
31         # Training points and the analytical (exact) solution at
               this points
32         x, y_exact = data  # entiendo q el y_exact ahora es un
               vector de 3 comps
33
34         # Calculate the gradients and update weights and bias
35         with tf.GradientTape() as tape:
36             # Calculate the gradients dy/dx #habr  que hacer esto
                   para las 3 comps del vector
37             with tf.GradientTape() as tape2:
38                 tape2.watch(self.x0)
39                 tape2.watch(x)
40                 y0_NN = self(self.x0, training=True)
41                 y_NN = self(x, training=True)  # la y_NN tiene 2
                       comps
```

```
42              print("forma␣y_NN", y_NN.shape)
43              print("forma␣de␣x", x.shape)
44
45              dy_dx_NN = tape2.batch_jacobian(y_NN, x)
46
47              print("Shape␣of␣dy_dx_NN:", dy_dx_NN.shape)
48              print("Shape␣of␣y_NN:", y_NN.shape)
49              print("Shape␣of␣x:", x.shape)
50              print(dy_dx_NN[:, 0, 0].shape)
51              print(y_NN[:, 1].shape)
52              print(y_NN[:, 0].shape)
53
54              loss_ode = self.compiled_loss(dy_dx_NN[:, 0, 0], -m*(
                    y_NN[:, 1])) \
55                  + self.compiled_loss(dy_dx_NN[:, 1, 0], (n*y_NN[:,
                        0]))
56
57              y0_NN = tf.reshape(y0_NN, shape=self.y0_exact.shape)
58
59              loss_boundary = self.compiled_loss(y0_NN, self.y0_exact
                    )
60
61              loss = tf.cast(loss_ode, dtype='float32') + \
62                  tf.cast(loss_boundary, dtype='float32')
63
64          gradients = tape.gradient(loss, self.trainable_weights)
65          self.optimizer.apply_gradients(zip(gradients, self.
                trainable_weights))
66          self.compiled_metrics.update_state(y_exact, y_NN)
67          self.loss_tracker.update_state(loss)
68          self.loss_ode_tracker.update_state(loss_ode)
69          self.loss_boundary_tracker.update_state(loss_boundary)
70          return {m.name: m.result() for m in self.metrics}
71
72
73  # In[2]: #INIT PARAMETERS
74  x0 = tf.constant([[0.0]], dtype=tf.float32)
75  x0 = tf.convert_to_tensor([[0.0]], dtype=tf.float32)
76  y0_exact = np.array([[1.0, 0.0]])
77  print(y0_exact.shape)
78
79  # ejecuto la PINN
80  n_train = 500
81  xmin = 0
82  xmax = 10.0
83
84  # Definition of the function domain
85  x_train = np.linspace(xmin, xmax, n_train)
86  x_train = np.reshape(x_train, (n_train, 1))
87  print(x_train.shape)  # (ntrain,1)
88
```

```
# The real solution y(x) for training evaluation
y_train = np.zeros((n_train, 2))  #2 neuronas de salida
print(y_train.shape)

# In[3]:#NEURAL NETWORK MODEL
# Input and output neurons (from the data)
input_neurons = 1
output_neurons = 2

epochs = 5000

minib_size = 64

# Definition of the the model
activation = 'tanh'
input_layer = Input(shape=(input_neurons,))
x = Dense(50, activation=activation)(input_layer)
x = Dense(50, activation=activation)(x)
x = Dense(50, activation=activation)(x)
output_layer = Dense(output_neurons, activation=None)(x)
print(x0.shape)  # (1,1)
print(y0_exact.shape)  # (1,2)
print(input_layer.shape)  # (None,1)
print(output_layer.shape)  # (None,2)


# Definition of the metrics, optimizer and loss
loss = tf.keras.losses.MeanSquaredError()
metrics = tf.keras.metrics.MeanSquaredError()
optimizer = Adam(learning_rate=0.001)

# In[4]:#ITERATIVE ALGORITHM
# Iterar sobre los subintervalos
n_subintervalos = 10

x0_list = []
print(x0_list)
y0_exact_list = [y0_exact]
print(y0_exact_list)

losses_ode = []
losses_boundary = []
losses = []
MSEs = []
predicciones = []

for i in range(n_subintervalos):
    xmin_sub = i * (xmax - xmin)/n_subintervalos
    xmax_sub = (i + 1) * (xmax - xmin)/n_subintervalos
    x_train_sub = np.linspace(xmin, xmax_sub, int(
        n_train/n_subintervalos) * (i + 1))
```

```
    x_train_sub = np.reshape(
        x_train_sub, (int(n_train/n_subintervalos) * (i + 1), 1))
    y_train_sub = np.zeros((len(x_train_sub), 2))
    # Crear y_train para este subintervalo si es necesario
    print("Dimensiones de x_train_sub para el subintervalo",
          i, ":", x_train_sub.shape)
    print("Dimensiones de y_train_sub para el subintervalo",
          i, ":", y_train_sub.shape)

    # Set weights from previous model
    if(i > 0):
        model.set_weights(w)

    x0_list.append(xmin_sub)
    x0_array = np.array(x0_list)
    print(x0_array.shape)
    x0_array = np.reshape(x0_array, (i+1, 1))
    print(x0_array.shape)
    print(x0_array[i])

    if(i > 0):
        y0_exact_i = model.predict(np.reshape(x0_array[i], (1, 1)))
        y0_exact_list.append(y0_exact_i)
        y0_exact_array = np.array(y0_exact_list)
    else:
        y0_exact_array = y0_exact
    print("y0_exact_array = ", y0_exact_array)

    x0_tensor = tf.constant(x0_array)
    y0_exact_tensor = tf.constant(y0_exact_array)
    print(x_train_sub.shape)
    print(np.min(x_train_sub))
    print(np.max(x_train_sub))

    print("SHAPE X0_ARRAY = ", np.shape(x0_array),
          " ; SHAPE Y0_EXACT = ", np.shape(y0_exact))
    model = ODE_1st(x0=x0_tensor, y0_exact=y0_exact_tensor,
                    inputs=input_layer, outputs=output_layer)
    model.compile(loss=loss,
                  optimizer=optimizer,
                  metrics=[metrics])
    history = model.fit(x_train_sub, y_train_sub,
                        batch_size=minib_size, epochs=epochs,
                           verbose=True)
    losses.append(history.history['loss'])
    losses_ode.append(history.history['loss_ode'])
    losses_boundary.append(history.history['loss_boundary'])
    MSEs.append(history.history['mean_squared_error'])

    w = model.get_weights()

```

```
# In[5]:Metrics: total loss and MSE
# Grafica las p rdidas por subintervalo
# loss total
plt.figure(figsize=(8, 5))
for i, loss_sub in enumerate(losses):
    plt.plot(loss_sub, label=f'Subintervalo {i+1}')

plt.xlabel(' pocas ',fontsize=12)
plt.ylabel('Coste',fontsize=12)
plt.yscale("log")
plt.title('Funci n coste durante el entrenamiento por subintervalo
    ',fontsize=15)
plt.legend(loc="upper right",fontsize=10)
plt.savefig('lossrjpartestocho.pdf',dpi=500)
plt.show()

for i, mse_sub in enumerate(MSEs):
    plt.plot(mse_sub, label=f'Subintervalo {i+1}')
plt.xlabel(' pocas ')
plt.ylabel('MSE')
plt.yscale("log")
plt.title('MSE durante el entrenamiento por subintervalo')
# plt.savefig('msesub_xmax=0,1_nsub=4_epochs=2000')
plt.legend(loc="upper right")
plt.show()

# In[6]:Metrics: ODE losses
# loss_ode
plt.figure(figsize=(8, 5))
for i, loss_ode_sub in enumerate(losses_ode):
    # SGR: loss_ode_sub instead of losses_ode
    plt.plot(loss_ode_sub, label=f'Subintervalo {i+1}')
plt.xlabel(' pocas ',fontsize=12)
plt.ylabel('Coste',fontsize=12)
plt.yscale("log")
plt.title('Funci n coste ODE durante el entrenamiento por
    subintervalo',fontsize=15)
plt.legend(loc="upper right", fontsize=11)
plt.savefig('loss_ode_subintervalo.pdf',dpi=500)
plt.show()

# In[7]:Metrics: Boundary condition losses
# loss_boundary
plt.figure(figsize=(8, 5))
for i, loss_boundary_sub in enumerate(losses_boundary):
    # SGR: loss_boundary_sub instead of lossess_boundary
    plt.plot(loss_boundary_sub, label=f'Subintervalo {i+1}')
plt.xlabel(' pocas ',fontsize=12)
plt.ylabel('Coste',fontsize=12)
plt.yscale("log")
```

```
238 plt.title('Funci n␣coste␣C.I␣durante␣el␣entrenamiento␣por␣
    subintervalo',fontsize=15)
239 plt.legend(loc="lower␣left", fontsize=11)
240 plt.savefig('loss_CI_subintervalo.pdf',dpi=500)
241 plt.show()
242
243 # summarize history for loss and metris (todo)
244 plt.rcParams['figure.dpi'] = 150
245 plt.plot(history.history['loss'], color='magenta',
246          label='Funci n␣coste␣($L_D␣+␣L_B$)')
247 plt.yscale("log")
248 plt.xlabel(' pocas ')
249 plt.legend(loc='upper␣right')
250 #plt.savefig('losstotalrjpartes')
251 plt.show()
252
253 # In[8]:Check PINN predictions
254
255 raiz15 = tf.sqrt(15.0)
256 print("raiz15", raiz15)
257 raiz15_3 = raiz15/3
258 print(raiz15_3)
259 n = 500
260
261 x = np.linspace(0, 10, n)
262 print(x.shape)
263 print(x)
264
265 y_exact = np.zeros((n, 2))
266 print(y_exact.shape)
267 y_exact[:, 0] = tf.cos(raiz15*x)
268 y_exact[:, 1] = raiz15_3*tf.sin(raiz15*x)
269 print(x.shape)
270 y_NN = model.predict(x)
271 print(y_NN.shape)
272
273 print(x0_array.shape)
274
275 print(y0_exact_array.shape)
276
277 print(x.shape, y_exact.shape)
278 print(np.squeeze(x0_array).shape, np.squeeze(y0_exact_array)[:, 0].
    shape)
279 print(x0_array)
280 print(y0_exact_array)
281
282 # Plot the results
283
284 # LAS DOS COMPS
285 plt.scatter(np.squeeze(x0_array), np.squeeze(
286     y0_exact_array)[:, 0], label='CCD',color="limegreen")
```

```
plt.scatter(np.squeeze(x0_array), np.squeeze(
    y0_exact_array)[:, 1], color="limegreen")
plt.plot(x, y_exact[:,0], color="royalblue",linestyle='solid',
                    linewidth=2.5,label="$y(x)$ anal tica")
plt.plot(x, y_exact[:,1], color="royalblue",linestyle='solid',
                    linewidth=2.5)
plt.plot(x, y_NN[:,0], color="deeppink",linestyle='dashed',
                    linewidth=2.5, label="$y(x)$ PINN")
plt.plot(x, y_NN[:,1], color="deeppink",linestyle='dashed',
                    linewidth=2.5)
# plt.legend()
# plt.title("RJPARTES")
plt.legend(loc='lower left',fontsize=11)
plt.title("Soluci n anal tica vs PINN",fontsize=15)
plt.xlabel("x",fontsize=15)
plt.ylabel("y",fontsize=15,rotation=0)
plt.savefig('rjpartes2comps_x=10_5000_10sub.pdf', dpi=500)
plt.show()



# ESPIRAL
plt.plot(y_exact[:,0] , y_exact[:,1],color="royalblue",linestyle='
    solid',
                   linewidth=2.5,label="$y(x)$ anal tica")
plt.plot(y_NN[:,0] , y_NN[:,1],color="deeppink",linestyle='dashed',
                   linewidth=2.5,label="$y(x)$ PINN")
plt.legend()
plt.xlabel("R(t)")
plt.ylabel("J(t)",rotation=0)
plt.title('Soluci n real vs PINN')
#plt.savefig('RJ_xmax=0,1_nsub=4_epochs=2000')
plt.savefig('rjparteselipse.pdf',dpi=400)
plt.show()


# Fin del cron metro
end_time = time.time()

# Tiempo de ejecuci n
execution_time = end_time - start_time
print(f"Tiempo de ejecuci n: {execution_time} segundos")



fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# Plot ODE loss
for i, loss_ode_sub in enumerate(losses_ode):
    ax1.plot(loss_ode_sub, label=f'Subintervalo {i+1}')

```

```
ax1.set_xlabel(' pocas ', fontsize=20)
ax1.set_ylabel('Coste␣EDO', fontsize=20)
ax1.set_yscale("log")
#ax1.set_title('Funci n coste ODE durante el entrenamiento por
    subintervalo', fontsize=15)
ax1.legend(loc="upper␣right", fontsize=13)

# Plot CI loss
for i, loss_boundary_sub in enumerate(losses_boundary):
    ax2.plot(loss_boundary_sub, label=f'Subintervalo␣{i+1}')

ax2.set_xlabel(' pocas ', fontsize=20)
ax2.set_ylabel('Coste␣CI', fontsize=20)
ax2.set_yscale("log")
#ax2.set_title('Funci n coste C.I durante el entrenamiento por
    subintervalo', fontsize=15)
ax2.legend(loc="lower␣left", fontsize=13)


# Agregar etiquetas (a) y (b) arriba de las subfiguras
fig.text(0.05, 0.97, '(a)', fontsize=20, verticalalignment='top',
    horizontalalignment='left')
fig.text(0.55, 0.97, '(b)', fontsize=20, verticalalignment='top',
    horizontalalignment='left')

# Ajustar el layout para dejar espacio en la parte superior
plt.subplots_adjust(top=0.85)
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.savefig('loss_ode_CI_panel.pdf', dpi=500)
plt.show()
```