**UNIVERSIDAD DE SALAMANCA**
MÁSTER UNIVERSITARIO EN MODELIZACIÓN MATEMÁTICA

# Redes Neuronales Diferenciales: Fundamentos y Aplicaciones

AUTOR: Jaime Gabriel Vegas

TUTOR: Ángel María Martín del Rey

**Curso 2023-2024**



**UNIVERSIDAD DE SALAMANCA**
MÁSTER UNIVERSITARIO EN MODELIZACIÓN MATEMÁTICA

# Redes Neuronales Diferenciales: Fundamentos y Aplicaciones

Firmado por GABRIEL VEGAS JAIME (FIRMA) el día 06/07/2024 con un certificado emitido por AC DNIE 006

AUTOR: Jaime Gabriel Vegas

TUTOR: Ángel María Martín del Rey

**Curso 2023-2024**

MÁSTER UNIVERSITARIO EN MODELIZACIÓN MATEMÁTICA
Facultad de Ciencias
Universidad de Salamanca
e-mail: mumoma@usal.es
*https://www.usal.es/modelizacion-matematica*

# CERTIFICADO DEL TUTOR DE TRABAJO DE FIN DE MÁSTER

## MÁSTER UNIVERSITARIO EN MODELIZACIÓN MATEMÁTICA

D. Ángel María Martín del Rey, profesor del Departamento de Matemática Aplicada de la Universidad de Salamanca

HACE CONSTAR

Que el trabajo titulado “Redes Neuronales Diferenciales: Fundamentos y Aplicaciones”, que se presenta, ha sido realizado por Jaime Gabriel Vegas, con DNI número ***25300C y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo Fin de Máster del Máster Universitario en Modelización Matemática de la Universidad de Salamanca

Salamanca, a fecha de la firma electrónica

Firmado: Firmado por MARTIN DEL REY ANGEL MARIA - ***5320** el día 06/07/2024 con un certificado emitido por AC FNMT Usuarios

# Resumen

En este trabajo explicaremos los fundamentos de las *ecuaciones diferenciales ordinarias neuronales* (ODENets) y las *redes neuronales informadas en física* (PINNs), dos técnicas de aprendizaje profundo empleadas para resolver ecuaciones diferenciales ordinarias y en derivadas parciales, respectivamente.

Las ODENets son una familia de modelos de aprendizaje profundo en la que se emplea una red neuronal para parametrizar el campo vectorial. La salida de la red neuronal es calculada después por un algoritmo de resolución de ODEs que consideraremos una caja negra, de forma que la propagación hacia atrás de la red se hace sin acceder a las operaciones internas. Estos modelos tienen ventajas sobre otros métodos tradicionales de resolución de ODEs, como un coste de memoria constante, la adaptabilidad de la estrategia de evaluación a cada entrada o la posibilidad de intercambiar precisión numérica por velocidad, además de poder ser aplicados a datos con ruido y esparcidos en el tiempo. Explicaremos su funcionamiento y aplicaremos esta técnica a un ejemplo concreto.

En segundo lugar, las PINNs son redes neuronales entrenadas para resolver tareas de aprendizaje supervisado obedeciendo una cierta ley física descrita por una ecuación diferencial en derivadas parciales. Describimos su funcionamiento para el caso en tiempo continuo para sus dos modos: soluciones basadas en datos y descubrimientos basado en datos. Este nuevo modelo permite obtener aproximaciones de la solución de EDPs y estimaciones de sus parámetros. Explicaremos en qué consisten y cómo funcionan para después aplicar este método a un caso concreto.

Todos los ejemplos realizados en este trabajo pueden encontrarse en [1].

**Palabras clave:** *Inteligencia artificial-aprendizaje profundo-redes neuronales-ecuaciones diferenciales-ODENet-PINN.*

# Índice general

# Índice de figuras

# Capítulo 1

# *Introducción*

La inteligencia artificial es un área de las ciencias de la computación que ha logrado una enorme popularidad en los últimos años, pero cuyos orígenes se pueden remontar a la década de 1950. Aunque hubo trabajos en años anteriores, el nacimiento de este campo se suele establecer en 1956, donde un grupo de investigadores entre los que destacan nombres como John McCarthyh de Stanford, Trenchard More de Princeton, Arthur Samuel de IBM o Ray Solomonoff y Oliver Selfrindge del MIT, se reunieron durante el verano de ese año en la Universidad de Stanford. El objetivo era realizar un estudio en base a la conjetura de que cada aspecto del aprendizaje o cualquier otra característica de la inteligencia puede describirse con tanta precisión que una máquina puede ser hecha para simularlo. Este grupo no consiguió ningún logro rompedor durante ese verano, pero sí hizo que se conocieran sus integrantes, cuyos estudios dominarían el campo durante los próximos años. Fue un momento decisivo para la consideración de la inteligencia artificial como una rama independiente de la teoría de control o la teoría de decisiones. Ese mismo año dos investigadores de la actual Universidad Carnegie Mellon, Allen Newell y Herbert Simon, afirmaron "haber inventado un programa de ordenador capaz de pensar de manera no numérica y, por tanto, haber resuelto el venerable problema del cuerpo-mente", llegando a afirmar que el programa había llegado a probar teoremas sencillos. [2, 3].

Los primeros logros fueron recogidos muy positivamente ya que resultaba sorprendente ver a un ordenador realizar algo remotamente inteligente cuando unos años atrás se veían como máquinas capaces de hacer únicamente operaciones aritméticas. No obstante, pronto comenzaron a llegar las decepciones, acompañados con la publicación de informes que señalaban la incapacidad de los algoritmos en ese momento de cumplir con las expectativas, lo que llevó a una retirada de fondos generalizada y a un periodo conocido como "el invierno" de la inteligencia artificial, alrededor de los años próximos a 1980 [3, 4].

Siguiendo a años de subidas y bajadas del entusiasmo en el área [2], en la década de 1990 comenzó el aumento en popularidad que aún dura, gracias principalmente a la popularización de redes neuronales artificiales fomentada por la disponibilidad de grandes cantidades de datos de entrenamiento y a la disponibilidad de cada vez más elevado poder computacional. Algunos ejemplos son la victoria del algoritmo DeepBlue frente campeón del mundo de ajedrez Gary Kasparov (1997) [5], la introducción de arquitecturas como *LeNet* para el reconocimiento de imágenes (1998) [6], la invención de las *redes generativas adversarias* capaces de generar contenido nuevo como imágenes o audio [7] o, por supuesto, la arquitectura *Transformer* que está detrás de productos como ChatGPT [8].

Desde el inicio se ha tratado de aplicar estas técnicas a ámbitos científicos-matemáticos, siendo quizás uno de los más conocidos el proyecto AlphaFold, que usa aprendizaje profundo para predecir la estructura tridimensional de proteínas a partir de su secuencia de aminoácidos, logrando una precisión que compite directamente con los experimentos [9]. Las grandes organizaciones científicas, como el CERN o la ESA han sido tradicionalmente impulsores de estas nuevas técnicas, sobre todo enfocadas al análisis de datos [10, 11].

El objetivo de este trabajo será el estudio de dos técnicas de aprendizaje profundo para el modelado de ecuaciones diferenciales ordinarias, las *ecuaciones diferenciales ordinarias neuronales*, y parciales, las *redes neuronales informadas en física*. Siendo el capítulo 1 esta introducción, en el capítulo 2 estableceremos los fundamentos matemáticos de las *redes neuronales prealimentadas*, caracterizadas por la dirección del flujo de la información, únicamente hacia delante, entre sus capas. Primero introduciremos y explicaremos el concepto *neurona artificial*, el elemento constituyente de las redes neuronales. Después hablaremos de su organización dentro de la red y el funcionamiento de esta. Por último, hablaremos del proceso por el cual una red neuronal artifical prealimentada aprende, el entrenamiento, y de cómo lo hace gracias al uso del descenso del gradiente y la propagación hacia atrás. También detallaremos un ejemplo sencillo del uso de estos modelos, en concreto para clasificación.

El capítulo 3 discurrirá sobre las *ecuaciones diferenciales ordinarias neuronales* u ODENets. Estas son una familia de modelos de aprendizaje profundo, concretamente una aplicación de las redes neuronales prealimentas explicada en el capítulo 2, con el que modelar ecuaciones diferenciales ordinarias (EDOs). En lugar de apredener la solución de la ecuación, la red neuronal modela el campo vectorial, para luego obtener una solución mediante el uso del algoritmo de resolución de EDOs (ODESolver) que consideremos. Este algoritmo se trata como una caja negra, de forma que el entrenamiento de la red se lleva a cabo sin acceder a sus operaciones internas. Esto permite el entrenamiento de la red con un coste de memoria constante en función de la profundidad de la red al no tener que guardar los valores intermedios del ODESolver, lo que es un cuello de botella importante en el entrenamiento de modelos profundos. Este método también tiene asociadas otras ventajas, como la posibilidad de emplear un ODESolver con estrategias de evaluación adaptativas para cada entrada, una disminución del número de parámetros de la red necesarios para aprender una cierta tarea comparado con otras arquitecturas de redes neuronales, o la posibilidad de aprender a partir de datos reales esparcidos aleatoriamente en el tiempo, pudiendo incluso extrapolar fuera de este intervalo temporal [12].
Explicaremos los fundamentos matemáticos de esta técnica y la aplicaremos a un ejemplo concreto

En el capítulo 4 describiremos las *redes neuronales informadas en física* o PINNs, una aplicación de las redes neuronales prealimentadas. Esta será útil en casos donde no se disponga de muchos datos o la obtención de estos sea muy costosa, pero si dispongamos de conocimiento físico previo del sistema, concretamente la EDP que dicta su comportamiento. Esto amplifica la información contenida en los datos y nos ayuda a obtener el comportamiento del sistema, es decir, la solución de la EDP. También, como veremos más adelante, se puede usar este método para ajustar una EDP, es decir, estimar el valor de sus parámetros. Esta formulación tiene varias ventajas respecto a otros intentos previos de añadir información física a algoritmos de aprendizaje automático, como procesos de

regresión gaussiana [13]. Por ejemplo, intentos de extender esta técnica a problemas no lineales [14, 15] requieren linealizar localmente los términos de orden superior, limitando la aplicabilidad. Además se requiere hacer unas ciertas asunciones previas que pueden limitar la capacidad de representación del modelo [16].
Explicaremos el funcionamiento de este algoritmo y lo aplicaremos al problema de la ecuación del calor.

# Capítulo 2

# *Fundamentos matemáticos del aprendizaje automático*

## 2.1. El aprendizaje automático

La inteligencia artificial es un concepto para el que podemos encontrar varias definiciones, todas alrededor de la misma idea [17]:

> "La inteligencia artificial es el estudio y desarrollo de sistemas computacionales que puedan copiar el comportamiento humano inteligente."

Siendo esta definición tan general, podemos pensar que el concepto de inteligencia artificial no es algo nuevo. Al fin y al cabo, vivimos rodeado de dispositivos "inteligentes", que toman decisiones en base a una lógica pre-programada por nosotros. Por ejemplo, el termostato que tenemos en casa enciende o apaga la calefacción siguiendo una serie de reglas con un aspecto similar a las siguientes (siendo $T$ la temperatura y $h$ la hora):

- Si $T < 21^{\circ}$ C $\Rightarrow$ encender los radiadores.
- Si $T > 21^{\circ}$ C $\Rightarrow$ no encender los radiadores.
- Si $T < 21^{\circ}$ C y $00:00 < h < 07:00 \Rightarrow$ no encender los radiadores.
- ...

De esta forma, el programa que ejemplifica esta lista de reglas es inteligente de acuerdo a la definición dada. No obstante, no es la inteligencia artificial en la que nos vamos a centrar en este trabajo. Nosotros nos centraremos en una rama de la inteligencia artificial llamada *aprendizaje automático*, también conocido como por su nombre en ingles, *machine learning*, y más concretamente en el *aprendizaje profundo* o *deep learning*.

Arthur Lee Samuel, pionero del aprendizaje automático, lo definió como "el campo de estudio que da a los ordenadores la habilidad de aprender sin ser explícitamente programados" [18]. De esta forma, continuando con el ejemplo anterior, un algoritmo de aprendizaje automático aprendería cuándo encender o apagar los radiadores no mediante una lista de reglas, sino observando ejemplos y *aprendiendo* automáticamente las normas que gobiernan el fenómeno.

El paradigma del aprendizaje automático en que nos centraremos será el de las *redes neuronales artificiales*, las cuales describiremos más en profundidad en los siguientes apartados.

### 2.1.1. Conceptos generales

#### Conjunto de datos (dataset) de entrenamiento

Un *conjunto de datos* o *dataset* es una colección de información sobre un determinado fenómeno, como la variación de la temperatura o la velocidad de un objeto en función del tiempo, imágenes, audios, etc. Este conjunto de datos es la información a partir de la que nuestro algoritmo de aprendizaje automático aprenderá, de forma que para que este sea lo más efectivo posible, el conjunto de datos debe ser representativo del fenómeno que queremos modelar. En este sentido, si queremos crear un modelo que distinga imágenes de perros de cualquier otra cosa, no es óptimo que nuestro conjunto de datos de entrenamiento contenga imágenes solo de perros pequeños, o perros de un color, o de una sola raza, porque no sería representativo de la población total de perros. Muy ligado a esta cualidad, está el hecho de que el conjunto de datos sea lo mayor posible. Siguiendo con el ejemplo, sería recomendable tener fotos del mismo tipo de perro desde distintas perspectivas, distintas distancias, distinta luz ambiental, etc. Un buen conjunto de datos, abundante y de calidad, es crucial si queremos obtener buenos resultados. Muchas veces, conseguir el conjunto de datos de entrenamiento es de las partes más complejas de todo el proceso, y también de las más limitantes [19].

#### Entrenamiento y tipos de aprendizaje

El entrenamiento de un algoritmo de aprendizaje automático es el proceso por el cual este aprende sin intervención humana explícita. Una definición comúnmente aceptada de "aprender" en el contexto del aprendizaje automático es la siguiente: "Un algoritmo está aprendiendo de una experiencia $E$ con respecto a cierta clase de tareas $T$ y medida de desempeño $P$ si su desempeño en cierta clase de tareas $T$ medido por $P$ mejora con la experiencia $E$" [20].

Se distinguen principalmente los siguientes tipos de aprendizaje:

- Los algoritmos de *aprendizaje supervisado* son aquellos en los que el conjunto de entrenamiento contiene tanto la entrada $\mathbf{x}$ como la salida deseada para dicha entrada, $\mathbf{y}(\mathbf{x})$. En muchos casos las salidas $\mathbf{y}$ pueden ser difíciles de conseguir automáticamente, de forma que deben ser proporcionadas manualmente por un "supervisor" humano, aunque para el caso automático el nombre es el mismo. Un ejemplo de aprendizaje supervisado aplicado a la clasificación lo podemos encontrar en el apartado 2.4

- Los algoritmos de *aprendizaje no supervisado* son aquellos que observan características de los datos, pero no tienen una etiqueta como si ocurre en el caso supervisado. Un ejemplo clásico de este tipo de aprendizaje es el de encontrar la mejor representación de un conjunto de datos, donde "mejor representación" generalmente indica aquella que preserve tanta información acerca de $\mathbf{x}$ como sea posible al mismo tiempo que se obedece algún tipo de penalización o ligadura con el objetivo de mantener

la representación lo más simple posible, pero más accesible que $\mathbf{x}$ en sí. Otros casos serían obtener muestras de una distribución, eliminar el ruido de una distribución o encontrar cúmulos en los que agrupar subconjuntos del conjunto de datos [21]. Algunos ejemplos de aplicaciones serías algoritmos de agrupamiento espacial de datos (*clustering*) o detección de anomalías [22].

- El aprendizaje *semi supervisado* es aquel que combina datos etiquetados con no etiquetados. Un ejemplo es el auto entrenamiento, en el que un algoritmo clasificador es entrenado con una pequeña cantidad de datos etiquetados, como en el caso supervisado, y luego es usado para clasificar otro conjunto de datos no etiquetados. Los puntos de este último conjunto de datos que se hayan clasificado con mayor seguridad son añadidos al conjunto de entrenamiento. El algoritmo es reentrenado y el proceso se repite [23].

- En el *aprendizaje por refuerzo* un agente autónomo debe aprender a realizar una tarea por prueba y error, recibiendo una recompensa si lo hace correctamente o una penalización si lo hace incorrectamente. Este tipo de técnicas ha obtenido muy buenos resultados en problemas de toma de decisiones secuencial, siendo muy conocidos los logros en el ajedrez y el Go de AlphaZero, un algoritmo de aprendizaje reforzado que ha logrado capacidades superhumanas [24].
  Los videojuegos son un entorno muy cómodo para entrenar este tipo de algoritmos, ya que es sencillo tener acceso a numerosas métricas en todo momento y crear grandes bancos de datos. Un ejemplo del éxito en este ámbito es StarCraft, un videojuego de estrategia en tiempo real conocido por su complejidad, en el que hay que tomar decisiones sobre la economía de la partida mientras se controla a cientos de unidades individuales. AlphaStar fue clasificado como Gran Maestro en este juego, por encima del 99,8 % de jugadores humanos [25].

Otros conceptos importantes son:

- *Época*: Un paso completo sobre todo el conjunto de entrenamiento por el algoritmo de aprendizaje automático. Comúnmente un entrenamiento consiste en varias épocas [26].

- *Sobre ajuste*: situación en la que el modelo se ajusta demasiado ó exactamente a los datos de entrenamiento, lo que causa que no sea capaz de generalizar a datos que no ha visto nunca [26]. Suele ocurrir cuando el número de épocas de entrenamiento es exageradamente grande, y es la principal razón por las que se separa el conjunto de datos de entrenamiento en los subconjuntos de entrenamiento y validación, ya que un signo claro de sobre ajuste es obtener buenos resultados en el primer subconjunto y malos resultados en el segundo.

## 2.2. Las redes neuronales artificiales

Las *redes neuronales artificiales*, a partir de ahora simplemente *redes neuronales*, son un tipo de modelo dentro del aprendizaje automático que han logrado muy buenos resultados dentro de campos como el reconocimiento de imágenes [27] o procesamiento de

lenguaje natural [28], así como otros tan llamativos como ganar al campeón mundial de Ajedrez [5] o Go [29].

Matemáticamente, podemos definir a una red neuronal como una función $f(\mathbf{x}, \mathbf{y}(\bar{\mathbf{x}}), \boldsymbol{\theta})$ tal que $f : \mathbb{R}^{n_0} \times \mathbb{R}^{n_L} \times \mathbb{R}^{n_m} \to \mathbb{R}^{n_L}$. La red neuronal dependerá de unos ciertos parámetros internos $\boldsymbol{\theta}$ y de unos datos de entrenamiento $\mathbf{y}(\bar{\mathbf{x}})$, y trataremos de obtener su salida para una nueva entrada $\mathbf{x}$.

En los siguientes apartados explicaremos en detalle su funcionamiento y objetivo.

### 2.2.1. La neurona artificial

La *neurona artificial* (a partir de ahora, *neurona*) es el elemento constituyente de las redes neuronales. Una neurona es una función que recibe un número de entradas, cada una con un peso asociado, y produce una única salida. Veamos el funcionamiento de una neurona apoyándonos en una de las formas más sencillas, el *Perceptrón* [30, 31]. Se suele representar como vemos en la Figura 2.1.

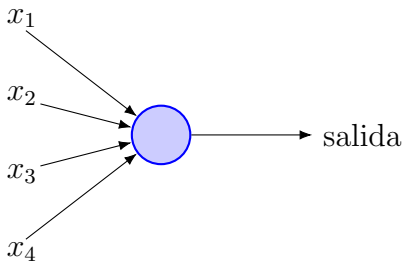


Figura 2.1: Neurona con cuatro entradas y una salida.

La salida del perceptrón se calcula de la siguiente forma:

$$\sigma(z) = \begin{cases} 0 & \text{si } z \leq 0, \\ 1 & \text{si } z > 0. \end{cases} \quad , \quad z = \mathbf{w} \cdot \mathbf{x} + b \in \mathbb{R}, \tag{2.1}$$

En la anterior ecuación, $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ es el vector de entradas de la neurona, $\mathbf{w} = (w_1, w_2, \ldots, w_n) \in \mathbb{R}^n$ el vector de pesos asociados a cada entrada, "$\cdot$" es el producto escalar entre ambos, y $b \in \mathbb{R}$ es el *sesgo* de la neurona. Cada neurona tiene como parámetros asociados $n$ pesos, siendo $n$ el número de entradas, y un sesgo. La función $\sigma(\cdot)$ que calcula la salida de la neurona se denomina *función de activación*, siendo su valor la *activación* de la neurona. Existen numerosas funciones de activación, compartiendo todas las propiedades de continuidad y no linealidad. En la tabla 2.1 podemos ver algunas de las más empleadas. El argumento de la función de activación siempre es la transformación afín $z = \mathbf{w} \cdot \mathbf{x} + b$, al que se suele llamar *suma ponderada*.

### 2.2.2. La arquitectura de una red neuronal prealimentada

Una red neuronal es una estructura organizada en capas, estando cada capa formada por un número determinado de neuronas. La primera capa se denomina *capa de entrada*, la última, *capa de salida*, y las restantes, *capas ocultas*. Las neuronas de la capa de entrada reciben los valores que hayamos introducido a la red y se lo pasan a todas las de la capa

| Nombre | Expresión | Rango |
|---|---|---|
| Función de Heaviside | $\sigma(z) = \begin{cases} 0 \text{ si } z < 0, \\ 1 \text{ en otro caso} \end{cases}$ | $\{0, 1\}$ |
| Sigmoide | $\sigma(z) = \dfrac{1}{1 + e^{-z}}$ | $(0, 1)$ |
| Tangente hiperbólica | $\sigma(z) = \tanh(z)$ | $(-1, 1)$ |
| Valor absoluto | $\sigma(z) = \text{abs}(z)$ | $[0, +\infty)$ |
| ReLU | $\sigma(z) = \text{máx}(0, z)$ | $[0, \infty)$ |
| Leaky ReLu | $\sigma(z) = \begin{cases} 0{,}01z \text{ si } z \leq 0, \\ z \text{ en otro caso} \end{cases}$ | $(-\infty, \infty)$ |
| Softmax | $f(\mathbf{z})_i = \dfrac{e^{z_i}}{\sum_{j=1}^{\#\mathbf{z}} e^{z_j}}$ | $[0, 1]$ |

Tabla 2.1: Algunas de las funciones de activación más empleadas [32].

siguiente sin realizar ninguna operación. Después, cada neurona recibe como entradas todas las salidas de aquellas de la capa anterior, y con sus pesos y sesgos, calculan una salida, la cual pasan a todas las neuronas de la capa siguiente. Podemos ver un ejemplo de red neuronal en la Figura 2.2.

En el ejemplo de la Figura 2.2, la red está formado por cinco capas: una de entrada, con tres neuronas; tres capas ocultas, cada una con cinco neuronas; y una capa de salida, conformada por tres neuronas. En este caso, todas las neuronas de las capas ocultas tienen el mismo número de neuronas, pero esto no tiene por qué ser así. Por otro lado, el número de neuronas de la capa de entrada y de salida sí están fijados para cada caso, debiendo ser igual al número de entradas y salidas de la función que deseamos aproximar.
Al número de capas se le suele denominar *profundidad*, mientras que al número de neuronas de cada capa se llama *anchura* [21]. De esta forma, una red neuronal más profunda tendrá más capas ocultas que una menos profunda, y de igual manera con la anchura.

Las redes neuronales con la arquitectura vista en la Figura 2.2 se denominan *redes neuronales prealientadas*, (*feedforward neural networks*), ya que la información siempre se mueve en la misma dirección, y de una capa a la siguiente. También se suele referir a las capas de este tipo de redes como *completamente conectadas*, porque todas las neuronas de capas consecutivas están conectadas entre sí. En otras arquitecturas, como en las *redes neuronales recurrentes* [21] o *redes neuronales de transformadores* [34], esto no ocurre, .

Respecto a la notación, la etiqueta $h_i^{(k)}$ se refiere a la activación de la neurona $i$-ésima de la capa $k$-ésima. De igual forma que $z_i^{(k)}$, el argumento de la función de activación, representa la suma ponderada de esa misma neurona, tal que $h_i^{(k)} = \sigma(z_i^{(k)})$. Los pesos individuales los denotaremos como $w_{i,j}^{(k)}$, indicando el peso de la neurona $i$-ésima de la capa $k$-ésima asociado a la entrada recibida de la $j$-ésima neurona de la capa anterior.

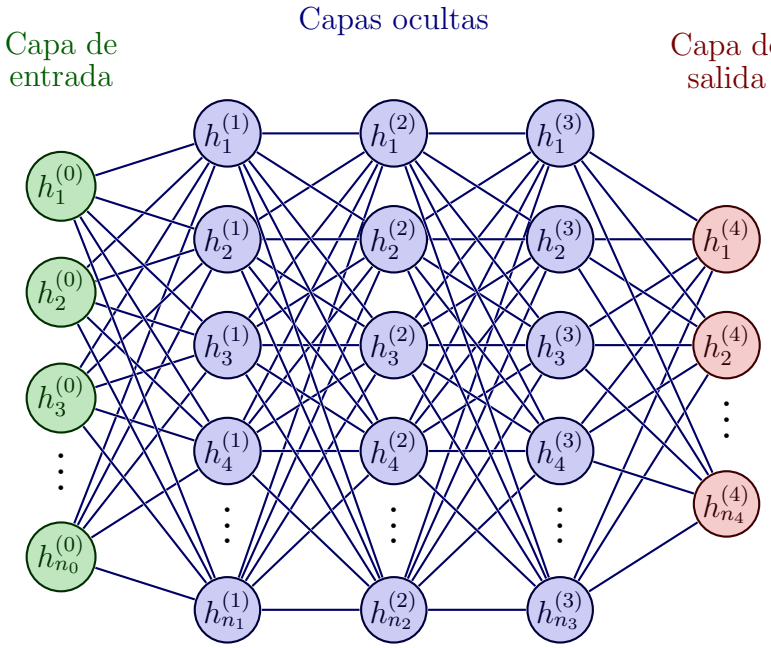


Figura 2.2: Ejemplo de red neuronal. Esta y todas las Figuras de redes neuronales a lo largo del trabajo han sido adaptadas de [33].

Podemos ver en la Figura 2.3 un diagrama que puede facilitar la compresión de esta notación. También, $b_i^{(k)}$ indica el sesgo de la neurona $i$-ésima de la capa $k$.
Usaremos la convención de empezar a numerar en el cero las capas pero no las activaciones. De esta forma, la $i$-ésima activación de la primera capa será $h_i^{(0)}$ (los datos de entrada), la de la segunda capa (primera capa oculta) será $h_i^{(1)}$, y así sucesivamente hasta la última capa, que denotaremos como capa $L$ y cuya salida $i$-ésima será $h_i^{(L)}$. Por último, $n_0, n_i, n_L$ se refieren al número de neuronas de la capa de entrada, de la $i$-ésima capa oculta y de la capa de salida, respectivamente, para el caso general.

La expresión general para la salida de la neurona $i$ de la capa $k$, sería

$$h_i^{(k)} = \sigma(z_i^{(k)}) = \sigma\left(\sum_{j=1}^{n_{k-1}} w_{i,j}^{(k)} h_j^{(k-1)} + b_i^{(k)}\right), \tag{2.2}$$

Para ser más concisos podemos adoptar una notación matricial, siendo $\mathbf{W}^{(k)} \in \mathbb{R}^{n_k \times n_{k-1}}$ la matriz con todos los pesos de las neuronas de la capa $k$. Sus elementos serán los pesos $w_{i,j}^{(k)}$ explicados anteriormente. De igual forma, $\mathbf{h}^{(k)}, \mathbf{b}^{(k)} \in \mathbb{R}^{n_k}$ denotan los vectores con las activaciones y los sesgos de las neuronas de la capa $k$. Con la notación matricial, las salidas de toda la capa sería la siguiente:

$$\begin{aligned}\mathbf{h}^{(k)} &= \sigma\left(\mathbf{W}^{(k)} \cdot \mathbf{h}^{(k-1)} - \mathbf{b}^{(k)}\right) \\ &= \sigma\left(\begin{pmatrix} w_{1,1}^{(k)} & w_{1,2}^{(k)} & \cdots & w_{1,n_{k-1}}^{(k)} \\ w_{2,1}^{(k)} & w_{2,2}^{(k)} & \cdots & w_{2,n_{k-1}}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_k,1}^{(k)} & w_{n_k,2}^{(k)} & \cdots & w_{n_k,n_{k-1}}^{(k)} \end{pmatrix} \begin{pmatrix} h_1^{(k-1)} \\ h_2^{(k-1)} \\ \vdots \\ h_{n_{k-1}}^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{n_k}^{(k)} \end{pmatrix}\right),\end{aligned} \tag{2.3}$$

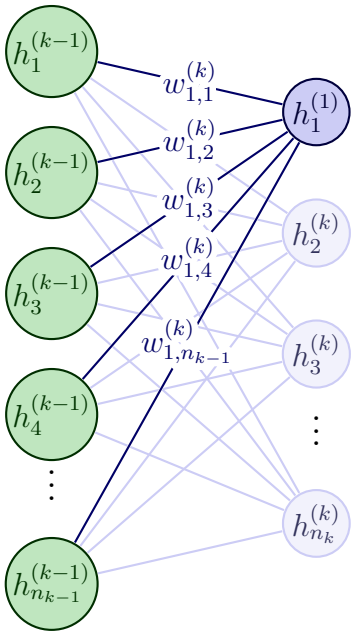


Figura 2.3: Conexiones entre dos capas ocultas de la red, donde se han resaltado las conexiones de la neurona 1 de la capa $k$ con todas las de la capa $k-1$.

entendiendo que en este caso la función $\sigma(\cdot)$ actúa elemento a elemento y que $\mathbf{h}^{(k-1)}$ es el vector transpuesto aunque no se explicite. En la ecuación anterior podemos ver claramente cómo la salida de las neuronas de una capa depende de las salidas de las capas anteriores.

## 2.3. Entrenamiento de una red neuronal

El objetivo de nuestra red neuronal será aproximar una cierta función de la que disponemos unos ciertos valores $\mathbf{y}(\mathbf{x}) \in \mathbb{R}^{n_L}$ para unas ciertas entradas $\mathbf{x} \in \mathbb{R}^{n_0}$. Estos datos se denominan *datos de entrenamiento*, y el proceso por el cual la red aprende a aproximar las reglas subyacentes que dominan a nuestra función, *entrenamiento*. El dato $\mathbf{y}(\mathbf{x})$ se denomina *objetivo*, y será la salida deseada de nuestra red para su entrada correspondiente, $\mathbf{x}$.

Para cuantificar la calidad de la salida de nuestra red, $\mathbf{h}(\mathbf{x})^{(L)} \in \mathbb{R}^{n_L}$, respecto a la salida deseada, $\mathbf{y}(\mathbf{x})$, necesitamos definir una *función de coste* o *función de pérdida*. Una función de pérdida ampliamente empleada es el error cuadrático medio,

$$L(\mathbf{h}^{(\mathbf{L})}, \mathbf{y}) = \frac{1}{n}\|\mathbf{y} - \mathbf{h}^{(\mathbf{L})}\|^2. \tag{2.4}$$

Cuanto más alejada esté la salida de nuestra red, $\mathbf{h}(\mathbf{x})^{(L)}$, de la salida deseada para la entrada correspondiente, $\mathbf{y}(\mathbf{x})$, mayor será el valor de la función de coste. El objetivo del proceso de entrenamiento será entonces minimizar su valor. Esto se hará encontrando unos valores de los pesos y los sesgos que hagan que la salida de red sea lo más parecido a su respectivo dato de entrenamiento para un cierto *conjunto de datos de entrenamiento* $(\mathbf{y}_1(\mathbf{x}_1), \mathbf{y}_2(\mathbf{x}_2), \ldots, \mathbf{y}_n(\mathbf{x}_n))$. Es una práctica muy común dividir el conjunto completo de entrenamiento en dos subconjuntos, el de entrenamiento y el de validación. De esta forma,

la red neuronal se entrena solo con los datos del subconjunto de entrenamiento, y una vez entrenada se comprueba su desempeño con el subconjunto de validación. De esta forma, podemos comprobar cómo de bien generaliza a datos que no ha visto nunca.

Minimizar la ecuación (2.4) analíticamente en la práctica será imposible, ya que el número de parámetros de una red neuronal puede ser del orden de millones. Aquí es donde hablamos de los *optimizadores*, algoritmos que emplearemos para minimizar numéricamente la función de coste.

### 2.3.1. Optimizadores

Como hemos dicho, necesitamos minimizar la función de coste (2.4), lo que haremos numérica e iterativamente. El algoritmo que decide cómo variar los parámetros para minimizar la función de coste se denomina *optimizador*. A continuación, explicamos brevemente algunos de los optimizadores más populares.

#### Descenso estocástico del gradiente (SGD)

El algoritmo SGD [26], el más sencillo de todos, consiste en calcular numéricamente el gradiente de la función de coste y modificar los valores de los pesos y los sesgos de la red en la "dirección" del gradiente negativo de la función de coste. Este proceso está representado de forma ilustrativa en la Figura 2.4.

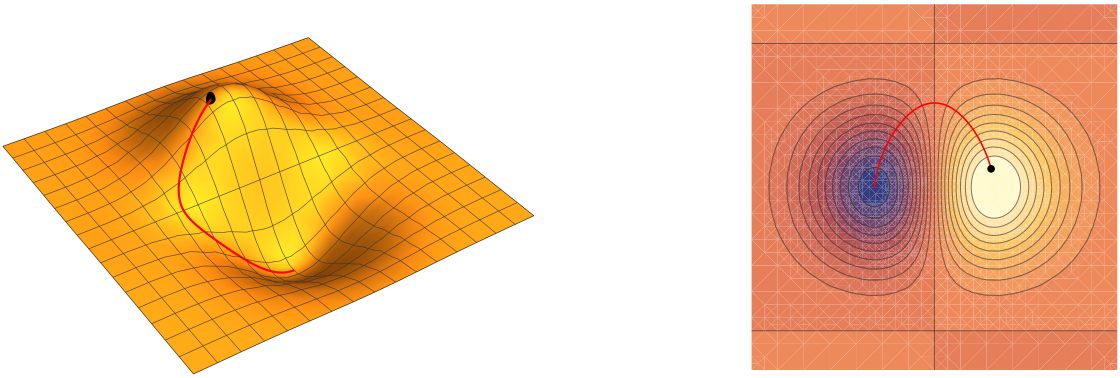

Figura 2.4: Uso del método del descenso del gradiente para hallar el mínimo de una función de dos dimensiones. Computación hecha en Mathematica adaptada de [35].

Sea $\boldsymbol{\theta} = (\theta_1, \theta_2, \dots, \theta_m) = (w^{(1)}_{1,1}, w^{(1)}_{1,2} \dots, w^{(L)}_{n_k,n_{k-1}}, b^{(1)}_1, b^{(1)}_2 \dots, b^{(L)}_{n_k})$ un vector con todos los parámetros de la red. Denotamos al número total de parámetros como $m$. A su vez, sea $\boldsymbol{\Delta\theta} = (\Delta\theta_1, \dots, \Delta\theta_m)$ otro vector con unos ciertos incrementos de los parámetros. Podemos ahora escribir el incremento de la función de coste como

$$\Delta L = \sum_{i=1}^{m} \frac{\partial L}{\partial \theta_i} \Delta\theta_i = \nabla \mathbf{L}_\theta \cdot \boldsymbol{\Delta\theta}, \tag{2.5}$$

donde $\nabla \mathbf{L}_\theta = (\frac{\partial L}{\partial \theta_1}, \dots, \frac{\partial L}{\partial \theta_m})$. De esta forma, podemos comprobar que si los incrementos de los parámetros tiene la forma

$$\boldsymbol{\Delta\theta} = -\eta \nabla \mathbf{L}_\theta \tag{2.6}$$

siendo $\eta \in \mathbb{R}^+$ un parámetro denominado *tasa de aprendizaje*, la variación de la función de coste será

$$\Delta L = \nabla \mathbf{L}_\theta \cdot \boldsymbol{\Delta\theta} = -\eta \|\nabla \mathbf{L}_\theta\|^2 < 0, \tag{2.7}$$

lo que garantiza que estamos disminuyendo su valor. La manera en que se actualizan los parámetros de la iteración $t-1$ a la $t$ es, por tanto,

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla L(\boldsymbol{\theta_{t-1}}). \tag{2.8}$$

La estocasticidad se introduce al darnos cuenta de que nuestra red puede tener miles o millones de parámetros, de forma que calcular el gradiente completo puede ser muy costoso. Lo que se hace entonces es realizar este proceso con un subconjunto de todos los datos de entrenamiento, lo que se suele llamar *minibatch*. De esta forma obtenemos una aproximación del gradiente que nos ayudará a llegar al mínimo más rápido. Esto es una práctica común, de forma que si no se indica lo contrario los gradientes de la función de coste no se hacen para todos los datos de entrenamiento, sino para un cierto *minibatch*.

Durante el proceso de entrenamiento se repite la operación (2.8) hasta que consideramos que estamos lo suficientemente cerca del mínimo.

### Descenso estocástico del gradiente con momento (SGD con momento)

Debido a la sencillez, el método anterior es sencillo de implementar, pero puede resultar en una convergencia muy lenta.

En el ejemplo de la Figura 2.4, vemos una trayectoria similar a la que podría seguir una partícula al descender esa misma pendiente. Intuitivamente, podemos pensar que a medida que la partícula desciende por una parte empinada, esta va acumulando velocidad o momento. No obstante, esto no ocurre con el descenso estocástico de gradiente, ya que si la partícula se desliza por una superficie de pendiente constante, esta lo hará siempre a la misma "velocidad". De esta forma, en el algoritmo de *descenso estocástico del gradiente con momento* [36] lo que se hace es añadir momento al descenso de la siguiente manera:

$$\mathbf{v}_0 = 0, \tag{2.9}$$

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla L(\boldsymbol{\theta}_{t-1}), \tag{2.10}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t, \tag{2.11}$$

donde el parámetro $\gamma \in \mathbb{R}^+$ se denomina *momento*, y suele fijarse en $\gamma = 0{,}9$ o un valor cercano [36]. De esta forma, el término del momento aumenta para dimensiones cuyos gradientes apuntan en la misma dirección durante varias iteraciones, mientras que se reduce para dimensiones cuyos gradientes cambian de dirección, lo que causa en muchos casos que la convergencia se acelere.

### Nesterov Acelerated Gradient (NAG)

La adición del momento ayuda a que la convergencia sea más rápida, pero sigue teniendo aspectos mejorables. Por ejemplo, siguiendo con la analogía de la partícula, a medida que esta baja por un valle va acumulando velocidad, lo que ocasiona que al alcanzar la vaguada esta comience a subir por la pendiente contraria durante algunas iteraciones, para después comenzar un pequeño movimiento oscilatorio alrededor del mínimo.

Estas oscilaciones no hacen sino aumentar el tiempo de convergencia, de forma que para solucionar este inconveniente podemos recurrir al *gradiente acelerado de Nesterov*, o *NAG* por sus siglas en inglés [37]. Este algoritmo obtiene una estimación de la posición futura del gradiente, de forma que si nos encontramos en la situación anterior podamos reducir el efecto no deseado. Esto es, en lugar de calcular el gradiente con respecto a los valores de los parámetros actuales, podemos calcularlos con respecto a la aproximación de los futuros valores,

$$\mathbf{v}_0 = 0, \tag{2.12}$$
$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla L(\boldsymbol{\theta}_{t-1} - \gamma \mathbf{v}_{t-1}), \tag{2.13}$$
$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t. \tag{2.14}$$

De esta forma, evitamos descender demasiado rápido en situaciones en las que esto nos podría perjudicar.

### Adaptive Gradient (Adagrad)

El siguiente algoritmo es el de *gradiente adaptativo*, y se suele denominar *Adagrad* [38].

Durante en el entrenamiento, no todos los parámetros de la red afectan de igual medida a la salida de la red. En este sentido, puede haber ciertos parámetros que solo intervengan, o que intervengan de una manera apreciable, en un número muy limitado de los casos del entrenamiento, y otros que lo hagan en casi la totalidad de los casos. Esto causa un problema, ya que habrá ciertos parámetros que converjan mucho más lentos que otros debido simplemente a que su valor se actualiza un número menor de veces. Adagrad permite adaptar la tasa de aprendizaje para cada parámetro, de forma que se realiza cambios más grandes en los parámetros que afectan con menos frecuencia y cambios más pequeños en los parámetros que afectan con más frecuencia. Recordemos que en los algoritmos anteriores actualizábamos todos los parámetros $\boldsymbol{\theta}$ al mismo tiempo debido a que todos los parámetros individuales $\theta_i$ tenían la misma tasa de aprendizaje $\eta$.

Denotemos como $g_{t,i}$ el gradiente de la función de coste con respecto al parámetro $\theta_i$ a tiempo $t$,

$$g_{t,i} = \nabla_{\theta_t} L(\boldsymbol{\theta_t}) = \frac{\partial L(\boldsymbol{\theta_t})}{\partial \theta_i}. \tag{2.15}$$

Por claridad, con esta notación, la regla de actualización para el caso del descenso estocástico del gradiente se escribiría como

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}. \tag{2.16}$$

La regla de actualización de Adagrad, en cambio, modifica el parámetro $\eta$ para cada iteración $t$ y cada parámetro $\theta_i$ basado en gradientes anteriores calculados para $\theta_i$:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\mathbf{G}_{t,ii} + \varepsilon}} \cdot g_{t,i}, \tag{2.17}$$

donde $\mathbf{G}_t \in \mathbb{R}^{d \times d}$ es una matriz cuya diagonal es la suma de los cuadrados de los gradientes con respecto a $\theta_i$ hasta la iteración $t$, mientras que $\varepsilon$ es un parámetro para evitar la división entre cero que suele establecerse en un valor del orden de $10^{-8}$ de forma predeterminada

[36]. No obstante, en algunos casos los resultados son sensibles a la modificación de este valor [39].
Tomando la multiplicación entre matriz y vector elemento a elemento, $\odot$, entre $\mathbf{G}_t$ y $\mathbf{g}_t$, podemos expresar la regla de actualización como

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta_t} - \frac{\eta}{\sqrt{\mathbf{G}_t + \varepsilon}} \odot \mathbf{g}_t. \tag{2.18}$$

Si no se indica lo contrario, todas las operaciones con vectores para el resto del apartado se harán elemento a elemento.

Uno de los grandes beneficios de Adagrad es el hecho de que elimina la necesidad de elegir un valor para la tasa de aprendizaje, que se suele dejar a $\eta = 0{,}01$ por efecto [36]. Por otro lado, la principal debilidad es la acumulación de gradientes al cuadrado en el denominador, ya que, al ser todos los términos positivos, la suma puede crecer tanto que se acabe dividiendo por un número tan grande que el aprendizaje acabe desapareciendo.

### Adadelta

*Adadelta* [40] es un algoritmo cuyo objetivo es eliminar el agresivo y monótono decrecimiento de la tasa de aprendizaje que ocurre en Adagrad. Esto se logra ponderando con un peso cada vez menor los gradientes más antiguos, de forma que estos se vayan "olvidando".

En lugar de almacenar los $d$ gradientes anteriores, la suma de los gradientes se define recursivamente como una media decreciente de todos los gradientes al cuadrado pasados. El promedio $E[g^2]_t$ a tiempo $t$ depende solo de las medias anteriores del gradiente actual:

$$E[\mathbf{g}^2]_t = \gamma E[\mathbf{g}^2]_{t-1} + (1-\gamma)\mathbf{g}_t^2, \tag{2.19}$$

donde $\gamma$, similar al momento y que se suele denominar *factor de olvido*, se establece alrededor de 0,9 [36]. Si el vector de actualización de los parámetros, $\boldsymbol{\Delta\theta}_t$, para el caso de Adagrad era

$$\boldsymbol{\Delta\theta}_t = -\frac{\eta}{\sqrt{G_t + \varepsilon}} \odot \mathbf{g}_t, \tag{2.20}$$

para Adadelta reemplazamos la matriz diagonal $\mathbf{G}_t$ con el promedio decreciente sobre los gradientes al cuadrado anteriores, $E[\mathbf{g}^2]_t$,

$$\boldsymbol{\Delta\theta}_t = -\frac{\eta}{\sqrt{E[\mathbf{g}^2]_t + \varepsilon}}\mathbf{g}_t = -\frac{\eta}{RMS[\mathbf{g}]_t}\mathbf{g}_t, \tag{2.21}$$

donde en la segunda igualdad hemos escrito el denominador como la media cuadrática, $RMS[\cdot]$, para simplificar la notación.

Es en este punto cuando el autor del algoritmo se da cuenta de que las unidades de la regla de actualización no son las mismas que aquellas de los parámetros (como por otra parte ocurre con el descenso estocástico del gradiente con y sin momento, o en Adagrad). Para arreglar esto, se define otro promedio exponencialmente decreciente, esta vez de las actualizaciones de los parámetros al cuadrado,

$$E[\boldsymbol{\Delta\theta}]_t = \gamma E[\boldsymbol{\Delta\theta}]_{t-1} + (1-\gamma)\boldsymbol{\Delta\theta}_t^2. \tag{2.22}$$

La media cuadrática de la actualización de los parámetros es entonces

$$RMS[\boldsymbol{\Delta\theta}]_t = \sqrt{E[\boldsymbol{\Delta\theta}^2]_t + \varepsilon}. \tag{2.23}$$

No sabemos cuánto vale $RMS[\boldsymbol{\Delta\theta}]_t$, así que lo aproximamos por la media cuadrática de las actualizaciones de los parámetros hasta la iteración en la que estamos, $t$. Si ahora reemplazamos la tasa de aprendizaje $\eta$ por la regla de actualización previa, $RMS[\boldsymbol{\Delta\theta}]_{t-1}$, obtenemos finalmente el algoritmo Adadelta,

$$\boldsymbol{\Delta\theta}_t = -\frac{RMS[\boldsymbol{\Delta\theta}]_{t-1}}{RMS[\mathbf{g}]_t} g_t, \tag{2.24}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{\Delta\theta}_t. \tag{2.25}$$

De esta forma, no tenemos que preocuparnos por establecer una tasa de aprendizaje inicial.

### RMSprop

*RMSprop* [41] es un algoritmo adaptativo que no ha sido publicado sino descrito en una clase de Coursera por el profesor Geoff Hinton . Es muy similar a Adadelta, ya que ambos fueron desarrollados para tratar de eliminar los gradientes menguantes de Adagrad. Su regla de actualización es la siguiente,

$$E[\mathbf{g}^2]_t = \gamma E[\mathbf{g}^2] + (1-\gamma)\mathbf{g}_t^2, \tag{2.26}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{E[\mathbf{g}^2]_t + \varepsilon}} \mathbf{g}_t. \tag{2.27}$$

El autor sugiere unos valores predeterminados de $\gamma = 0{,}9$ y $\eta = 0{,}001$.

### Adaptive Moment Estimation (Adam)

El algoritmo *Adam*, del inglés *Adaptative Moment Estimation* [42], es el más popular en la actualidad, y suele ser el predeterminado en la mayoría de casos. Es también un algoritmo que modifica la tasa de aprendizaje para cada parámetro. Además de almacenar un promedio exponencialmente decreciente de los gradientes al cuadrado, $v_t$, como hace Adadelta o RMSprop, Adam también emplea un promedio exponencialmente decreciente de los gradientes anteriores, $\mathbf{m}_t$, similar al momento,

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \tag{2.28}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2. \tag{2.29}$$

$\mathbf{m}_t$ y $\mathbf{v}_t$ son estimadores del primer momento (la media) y el segundo momento (la varianza no centrada) de los gradientes, respectivamente, de ahí el nombre del algoritmo. Los autores de Adam [42] se dieron cuenta de que al inicial $\mathbf{m}_t$ y $\mathbf{v}_t$ como vectores de ceros, se creaba un sesgo hacia el cero, sobre todo en los pasos iniciales y cuando las tasas de decrecimiento son pequeñas, es decir, $\beta_1$ y $\beta_2$ son cercanas a 1.

Para contrarrestar este efecto se halla las estimaciones del primer y segundo momento insesgados,

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \tag{2.30}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}, \tag{2.31}$$

para después actualizar los parámetros de manera muy similar a Adadelta o RMSprop,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_\mathbf{t}} + \varepsilon}\hat{\mathbf{m}}_t, \tag{2.32}$$

donde, de nuevo, todas las operaciones con vectores son elemento a elemento. Con $\beta_1^t$, $\beta_2^t$ se indica la potencia $t$ de los parámetros $\beta_1$ y $\beta_2$. Los autores proponen como valores predeterminados $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ y $\varepsilon = 10^{-8}$ [42].

En [42] los autores comparan varios optimizadores en bancos de prueba estándar para redes neuronales, obteniendo Adam el mejor resultado de entre todos los anteriores. En la Figura 2.5 podemos ver dos casos concretos.

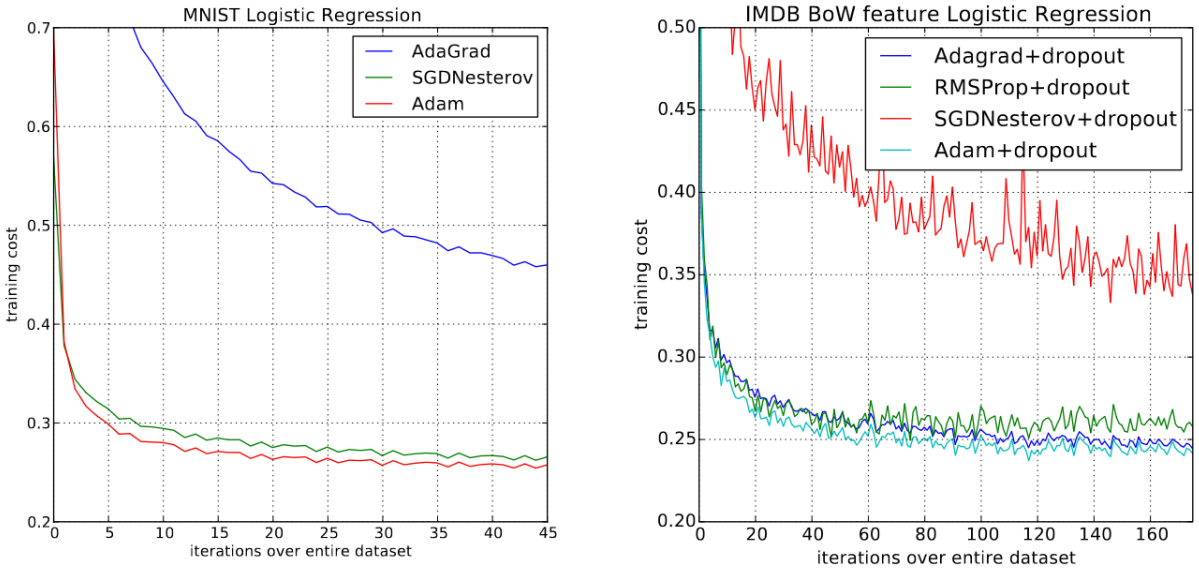


Figura 2.5: Comparación de varios optimizadores en bancos dos bancos de prueba estándar. Imagen sacada de [42].

La demostración inicial sobre la convergencia de Adam resulto ser incompleta, y se demostró que el algoritmo no converge para todas las funciones convexas [43]. En los últimos años han surgido varios algoritmos para solucionar esto [43, 44], no obstante, Adam continúa siendo de los más usados debido a sus buenos resultados prácticos.

### Nesterev-accelerated Adaptive Moment Estimation (Nadam)

El algoritmo *Nadam*, del inglés, *Nesterev-accelerated Adaptive Moment Estimation* [45], es una combinación entre Adam, que a su vez lo podíamos ver como una combinación de RMSprop y momento, con NAG. Para incorporar NAG en Adam, necesitamos modificar el término del momento $\mathbf{m}_t$.

Recordamos la regla de actualización del descenso estocástico del gradiente con momento con la notación actual,

$$\mathbf{g}_t = \nabla L(\boldsymbol{\theta}_t), \tag{2.33}$$
$$\mathbf{m}_t = \gamma \mathbf{m}_{t-1} + \eta \mathbf{g}_t, \tag{2.34}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{m}_t. \tag{2.35}$$

Si expandimos la tercera expresión obtenemos

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - (\gamma \mathbf{m}_{t-1} + \eta \mathbf{g}_t). \tag{2.36}$$

Utilizando lo aprendido con NAG, podemos realizar un paso más preciso si actualizamos los parámetros con el momento $\mathbf{m}$ antes de calcular el gradiente. Esto es,

$$\mathbf{g}_t = \nabla L(\boldsymbol{\theta}_t - \gamma \mathbf{m}_{t-1}), \tag{2.37}$$
$$\mathbf{m}_t = \gamma \mathbf{m}_{t-1} + \eta \mathbf{g}_t, \tag{2.38}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{m}_t. \tag{2.39}$$

La modificación del autor de Nadam [45] fue, en lugar de aplicar el paso del momento dos veces, una para actualizar el gradiente $\mathbf{g}_t$ y otras para actualizar los parámetros $\boldsymbol{\theta}_{t+1}$, aplicar la actualización del momento directamente al actualizar los parámetros actuales. Es decir,

$$\mathbf{g}_t = \nabla L(\boldsymbol{\theta}_t), \tag{2.40}$$
$$\mathbf{m}_t = \gamma \mathbf{m}_{t-1} + \eta \mathbf{g}_t, \tag{2.41}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - (\gamma \mathbf{m}_t + \eta \mathbf{g}_t). \tag{2.42}$$

De esta forma, en lugar de usar el momento anterior como en la ecuación (2.36), usamos el momento actual, $\mathbf{m}_t$, para modificar los parámetros $\boldsymbol{\theta}$. Podemos hacer algo similar con Adam, reemplazando el vector de momentos a timpo $t-1$ con aquel a tiempo $t$.

Recordemos la regla de actualización de Adam,

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \tag{2.43}$$
$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \tag{2.44}$$
$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}, \tag{2.45}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \varepsilon}\hat{\mathbf{m}}_t. \tag{2.46}$$

Expandiendo la segunda ecuación con la definición de $\hat{\mathbf{m}}_t$ y $\mathbf{m}_t$ obtenemos

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \varepsilon}\left(\frac{\beta_1 \mathbf{m}_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)\mathbf{g}_t}{1-\beta_1^t}\right) \\ &= \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \varepsilon}\left(\beta_1 \hat{\mathbf{m}}_{t-1} + \frac{(1-\beta_1)\mathbf{g}_t}{1-\beta_1^t}\right),\end{aligned} \tag{2.47}$$

donde al pasar de la primera expresión a la segunda hemos usado que $\frac{\mathbf{m}_{t-1}}{1-\beta_1^t}$ es la corrección no sesgada del vector de momentos en el paso previo $t-1$.

Ahora, de igual forma que pasamos de (2.36) a (2.42), reemplazamos la estimación no sesgada del vector de momentos en el paso anterior, $\hat{\mathbf{m}}_{t-1}$, con aquella en el último paso, $\hat{\mathbf{m}}_t$,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \varepsilon}\left(\beta_1 \hat{\mathbf{m}}_t + \frac{(1-\beta_1)\mathbf{g}_t}{1-\beta_1^t}\right), \tag{2.48}$$

Pese a la novedad, Adam sigue siendo el algoritmo más empleado por su sencillez y comprobada eficacia, aunque Nadam puede dar mejores resultados en determinados casos [45].

### 2.3.2. Propagación hacia atrás

Ahora ya sabemos de qué manera tenemos que modificar los pesos y los sesgos de la red neuronal para que esta aprenda, según la ecuación (2.6). Lo que nos falta ahora es saber cómo calcular el gradiente de la función de coste. Este proceso se lleva a cabo con el método de *propagación hacia atrás* (*backpropagation*) [26].

#### Condiciones para la función de coste

Para poder aplicar este método, la función de coste $L(\cdot)$ debe cumplir unas ciertas propiedades a saber:

1. La función de coste debe poder escribirse como un promedio de las funciones de coste individuales de cada dato de entrenamiento. Es decir, si tenemos un conjunto de entrenamiento de $p$ casos, siendo $L_i = L(\mathbf{h}_i^{(L)}, \mathbf{y}_i)$ con $\mathbf{h}_i^{(L)} = (\mathbf{h}_1^{(L)}, \ldots, \mathbf{h}_{n_L}^{(L)})$ la salida de la red neuronal para una cierta entrada $\mathbf{x}_i$, el valor de la función de coste para el caso individual $i$, debemos poder escribir

$$L = \frac{1}{p}\sum_{i=1}^{p} L_i. \tag{2.49}$$

2. La función de coste debe poder escribirse en función de las salidas de la red neuronal de la forma

$$L = L(\mathbf{h}^{(L)}). \tag{2.50}$$

#### Cálculo de gradiente de la función de coste

En esta apartado explicaremos un método para calcular el gradiente de la función de coste.

Suponiendo que nuestra función de coste cumple las condiciones expuestas en el apartado anterior, hacemos el razonamiento para un ejemplo individual del entrenamiento, $\mathbf{y}(\mathbf{x}) = (y_1, \ldots, y_{n_k})$. Para obtener la función de coste total solo tenemos que promediar sobre todos los datos de entrenamiento. La función de pérdida para este ejemplo individual será la siguiente:

$$L_0 = \|\mathbf{h}^{(L)} - \mathbf{y}\|^2 = \sum_{i=1}^{n_L}(h_i^{(L)} - y_i)^2. \tag{2.51}$$

Recordando la expresión de la salida de la red neuronal, ecuación (2.2), podemos aplicar la regla de la cadena para hallar la derivada de la función de coste respecto a los pesos y los sesgos de la última capa,

$$\frac{\partial L_0}{\partial w_{j,k}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{j,k}^{(L)}} \frac{\partial h_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial h_j^{(L)}} = h_k^{(L-1)} \sigma'(z_j^{(L)}) 2(h_j^{(L)} - y_j), \tag{2.52}$$

$$\frac{\partial L_0}{\partial b_j^{(L)}} = \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} \frac{\partial h_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial h_j^{(L)}} = \sigma'(z_j^{(L)}) 2(h_j^{(L)} - y_j). \tag{2.53}$$

Las derivadas respecto a los parámetros de la última capa son inmediatas. Sin embargo, si tratamos de hacer lo mismo para la capa previa obtenemos lo siguiente:

$$\frac{\partial L_0}{\partial w_{j,k}^{(L-1)}} = \frac{\partial z_j^{(L-1)}}{\partial w_{j,k}^{(L-1)}} \frac{\partial h_j^{(L-1)}}{\partial z_j^{(L-1)}} \frac{\partial L_0}{\partial h_j^{(L-1)}}, \tag{2.54}$$

$$\frac{\partial L_0}{\partial b_j^{(L-1)}} = \frac{\partial z_j^{(L-1)}}{\partial b_j^{(L-1)}} \frac{\partial h_j^{(L-1)}}{\partial z_j^{(L-1)}} \frac{\partial L_0}{\partial h_j^{(L-1)}}. \tag{2.55}$$

Recordando de nuevo la ecuación (2.2), todos los factores de la derecha de las ecuaciones anteriores son inmediatos excepto el último, $\frac{\partial L_0}{\partial h_j^{(L-1)}}$. No obstante, aplicando de nuevo la regla de la cadena podemos hallar este factor de la siguiente forma:

$$\frac{\partial L_0}{\partial h_k^{L-1}} = \sum_{i=1}^{n_L} \frac{\partial z_i^L}{\partial h_k^{L-1}} \frac{\partial h_i^L}{\partial z_i^L} \frac{\partial L_0}{\partial h_i^L}, \tag{2.56}$$

donde el sumatorio se introduce porque la salida $h_k^{(L-1)}$ modifica el valor de la salida de la red a través de todas las neuronas de la capa de salida, ya que envía su salida a todas ellas. Juntando la expresión anterior con (2.54) y (2.55) ya podemos hallar la derivada de la función de coste respecto a los parámetros de la penúltima capa.

De igual forma, para la antepenúltima capa tendríamos las expresiones

$$\frac{\partial L_0}{\partial w_{j,k}^{(L-2)}} = \frac{\partial z_j^{(L-2)}}{\partial w_{j,k}^{(L-2)}} \frac{\partial h_j^{(L-2)}}{\partial z_j^{(L-2)}} \frac{\partial L_0}{\partial h_j^{(L-2)}}, \tag{2.57}$$

$$\frac{\partial L_0}{\partial b_j^{(L-2)}} = \frac{\partial z_j^{(L-2)}}{\partial b_j^{(L-2)}} \frac{\partial h_j^{(L-2)}}{\partial z_j^{(L-2)}} \frac{\partial L_0}{\partial h_j^{(L-2)}}, \tag{2.58}$$

para las que tendríamos que emplear

$$\frac{\partial L_0}{\partial h_k^{(L-2)}} = \sum_{i=1}^{n_{L-1}} \frac{\partial z_i^{(L-1)}}{\partial h_k^{(L-2)}} \frac{\partial h_i^{(L-1)}}{\partial z_i^{(L-1)}} \frac{\partial L_0}{\partial h_i^{(L-1)}}. \tag{2.59}$$

Empleando las expresiones (2.51), (2.56) y (2.59) podemos hallar la derivada de la función de coste respecto a todos los parámetros de la antepenúltima capa.

Para hallar la derivada de la función de coste respecto a todos los parámetros de la red y construir su gradiente realizamos el mismo procedimiento para el resto de capas de la red.

### 2.3.3. Diferenciación automática

La propagación hacia atrás no es sino un caso concreto de una familia de técnicas para el cálculo eficiente y preciso de derivadas numéricas [46]: la *diferenciación automática* (AD, sus siglas en inglés, a partir de ahora). Muchos libros de texto solo explican el proceso de propagación hacia atrás sin referirse en ningún momento al concepto de AD. No obstante, en los últimos años ha demostrado su versatilidad y eficacia no solo en el mundo de la inteligencia artificial, sino en el de la computación numérica en general, como demuestra la aparición de populares bibliotecas como `JAX` ([47]) (antes `autograd`), `Chainer` [48] o `PyTorch` [49].

AD es un término referido a una familia de técnicas empleadas para calcular evaluaciones numéricas de derivadas a través de la acumulación de valores durante la ejecución del programa. Esto proporciona la forma de evaluar derivadas a la precisión de la máquina de manera eficiente evitando todo lo posible error de truncamiento o aproximaciones [46].

Para la gran mayoría de casos, las expresiones de las que queremos hallar su derivada son composiciones o combinaciones de elementos de un conjunto finito de operaciones cuya derivada es conocida. Combinando las derivadas de estos componentes a través de la regla de la cadena, la diferenciación automática permite generar algoritmos capaces de derivar expresiones arbitrarias sin necesidad, en la mayoría de casos, de modificación.

De esta forma, en la Tabla 2.2, podemos ver la representación del cálculo de la función de ejemplo $f(x_1, x_2) = e^{1+x_1} + x_1x_2 - \tan(x_2)$ en el punto $(x_1, x_2) = (-1, \frac{\pi}{4})$ como una *acumulación de evaluaciones* de operaciones elementales. El grafo computacional (grafo que incluye los pasos para el cálculo de la función a partir de operaciones elementales) para este caso lo podemos ver en la Figura 2.6. Adoptamos la notación en la que la función $f : \mathbb{R}^n \to \mathbb{R}^m$ se construye usando variables intermedias $v_i$ de tal forma que, siendo $i \in \mathbb{N}$,

- $v_{i-n} = x_i,\ 1 \leq i \leq n$ son las variables de entrada,
- $v_i,\ 1 \leq i \leq l$ son las variables intermedias,
- $y_{m-i} = v_{l+m-i},\ m-1 \geq i \geq 0$ son las variables de salida.

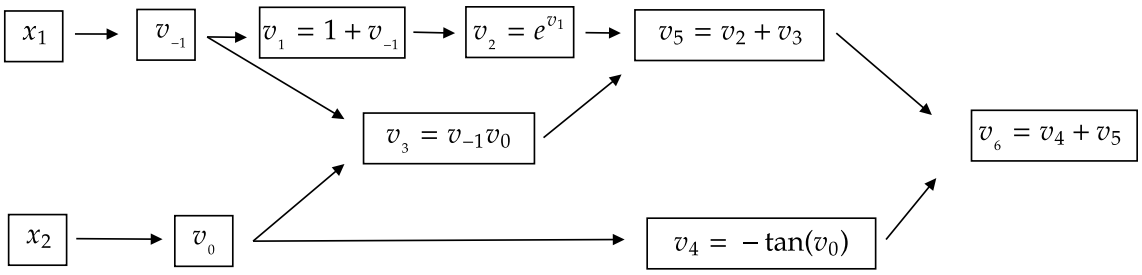


Figura 2.6: Gráfo computacional de la función $f(x_1, x_2) = e^{1+x_1} + x_1x_2 - \tan(x_2)$.

#### Modo de acumulación hacia delante

El *modo de acumulación hacia delante*, también llamado *modo lineal tangente* [46], es el más sencillo. Siguiendo con nuestra función de ejemplo $f(x_1, x_2) = e^{1+x_1} + x_1x_2 - \tan(x_2)$

en el punto $(x_1, x_2) = (-1, \frac{\pi}{4})$, la descomponemos en operaciones elementales:

$$v_{-1} = x_1 = -1, \tag{2.60}$$
$$v_0 = x_2 = \frac{\pi}{4}, \tag{2.61}$$
$$v_1 = 1 + v_{-1} = 0, \tag{2.62}$$
$$v_2 = e^{v_1} = 1, \tag{2.63}$$
$$v_3 = v_{-1} \cdot v_0 = -\frac{\pi}{4}, \tag{2.64}$$
$$\vdots$$

de forma que cada operación posterior esté en función de las anteriores. En el momento en que no podamos hacer más operaciones habremos calculado el valor buscado. Así se construye el grafo computacional asociado a la función, en nuestro caso la Figura 2.6. Los valores de las variables $v_i$ se van acumulando, en el sentido de que cada una está en función de las anteriores. En la Tabla 2.2 podemos ver todos los pasos necesarios para el cálculo de $f(-1, \frac{\pi}{4})$.

Para calcular la derivada con respecto a una de las variables, en nuestro caso respecto a $x_1$, empezamos asociando a cada variable intermedia $v_i$ la derivada

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1}. \tag{2.65}$$

Como cada variable intermedia $v_i$ es una función elemental, sabemos calcular su derivada de manera inmediata, de forma que podemos ir acumulando los valores de manera similar a como lo hemos hecho para el cálculo de $f(-1, \frac{\pi}{4})$, es decir:

$$\dot{v}_{-1} = \dot{x}_1 = \frac{\partial x_1}{\partial x_1} = 1, \tag{2.66}$$
$$\dot{v}_0 = \dot{x}_2 = \frac{\partial x_2}{\partial x_1} = 0, \tag{2.67}$$
$$\dot{v}_1 = \dot{v}_{-1} = 1, \tag{2.68}$$
$$\dot{v}_2 = e^{v_1}\dot{v}_1 = 1, \tag{2.69}$$
$$\dot{v}_3 = \dot{v}_{-1} \cdot v_0 + \dot{v}_0 \cdot v_{-1} = \frac{\pi}{4}, \tag{2.70}$$
$$\vdots$$

Mediante la aplicación de la regla de la cadena a cada operación elemental generamos la derivada que estamos buscando, $\dot{y} = \frac{\partial y}{\partial x_1}$. Esto lo podemos ver de forma completa en la parte derecha de la Tabla 2.2.

Este proceso se puede generalizar [46] para el cálculo del Jacobiano de una función $f : \mathbb{R}^n \to \mathbb{R}^m$, con $n$ variables de entrada $x_i$ y $m$ variables de salida $y_j$. Cada ejecución del modo de acumulación hacia delante es iniciado estableciendo una de las variables $\dot{x}_i = 1$ y el resto a cero. Para cada punto $\mathbf{a}$ donde queramos calcular la derivada obtenemos, tras una ejecución,

$$\dot{y}_j = \left.\frac{\partial y_j}{\partial x_i}\right|_{\mathbf{x}=\mathbf{a}}, \quad j = 1, \ldots, m, \tag{2.71}$$

lo que nos da una columna de la matriz Jacobiana

$$\mathbf{J}_f = \left( \begin{array}{ccc} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{array} \right) \Bigg|_{\mathbf{x}=\mathbf{a}} . \tag{2.72}$$

Esto hace que la AD en el modo de acumulación hacia delante sea muy eficiente para funciones $f : \mathbb{R} \to \mathbb{R}^m$, ya que todas las derivadas se pueden calcular con un solo paso. Por otro lado, para funciones $f : \mathbb{R}^n \to \mathbb{R}$ requerimos $n$ evaluaciones para calcular el gradiente

$$\nabla f = \left( \frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n} \right), \tag{2.73}$$

ya que se corresponde a la primera fila del Jacobiano de la función.

En general, para casos $f : \mathbb{R}^n \to \mathbb{R}^m$ donde $n \gg m$, es aconsejable el empleo de otra técnica, como la que explicaremos en el apartado siguiente.

| **Acumulación hacia delante** | | | | | **Acumulación tangente hacia delante** | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $v_{-1}$ | $=$ | $x_1$ | $=$ | $-1$ | $\dot{v}_{-1}$ | $=$ | $\dot{x}_1 = \frac{\partial x_1}{\partial x_1}$ | $=$ | $1$ |
| $v_0$ | $=$ | $x_2$ | $=$ | $\frac{\pi}{4}$ | $\dot{v}_0$ | $=$ | $\dot{x}_2$ | $=$ | $0$ |
| $v_1$ | $=$ | $1 + v_{-1}$ | $=$ | $0$ | $\dot{v}_1$ | $=$ | $\dot{v}_{-1}$ | $=$ | $1$ |
| $v_2$ | $=$ | $e^{v_1}$ | $=$ | $1$ | $\dot{v}_2$ | $=$ | $e^{v_1} \cdot \dot{v}_1$ | $=$ | $1$ |
| $v_3$ | $=$ | $v_{-1} \cdot v_0$ | $=$ | $-\frac{\pi}{4}$ | $\dot{v}_3$ | $=$ | $\dot{v}_{-1} \cdot v_0 + \dot{v}_0 \cdot v_{-1}$ | $=$ | $\frac{\pi}{4}$ |
| $v_4$ | $=$ | $-\tan(v_0)$ | $=$ | $-1$ | $\dot{v}_4$ | $=$ | $-\dot{v}_0 \cdot \tan(v_0)$ | $=$ | $0$ |
| $v_5$ | $=$ | $v_2 + v_3$ | $=$ | $1 - \frac{\pi}{4}$ | $\dot{v}_5$ | $=$ | $\dot{v}_2 + \dot{v}_3$ | $=$ | $1 + \frac{\pi}{4}$ |
| $v_6$ | $=$ | $v_4 + v_5$ | $=$ | $-\frac{\pi}{4}$ | $\dot{v}_6$ | $=$ | $\dot{v}_4 + \dot{v}_5$ | $=$ | $1 + \frac{\pi}{4}$ |
| $y$ | $=$ | $v_6$ | $=$ | $-\frac{\pi}{4}$ | $\dot{y}$ | $=$ | $\dot{v}_6$ | $=$ | $1 + \frac{\pi}{4}$ |

Tabla 2.2: Cálculo del valor de $f(x_1, x_2) = e^{1+x_1} + x_1 x_2 - \tan(x_2)$ y su derivada respecto a $x_1$ en el punto $(x_1, x_2) = (-1, \frac{\pi}{4})$.

### Modo de acumulación hacia atrás

El *modo de acumulación hacia atrás* de la AD, también llamado *modo adjunto* o *modo lineal cotangente* [46], se corresponde con una generalización del algoritmo de propagación hacia atrás. Esto es, propaga la derivada hacia atrás (de izquierda a derecha en un grafo computacional como el de la Figura 2.6) desde una salida dada. Se dice que es una generalización porque la propagación hacia atrás suele explicarse para el caso concreto de obtener la derivada de la función de coste respecto a los pesos y los sesgos de la red.

Comenzamos definiendo las nuevas variables intermedias $\bar{v}_i$ de la siguiente forma:

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}, \tag{2.74}$$

siendo $v_i$ las variables intermedias definidas para el modo de acumulación hacia delante e $y$ la función de la que queremos hallar su derivada, en nuestro caso $y = f(x_1, x_2)$.

Es importante señalar que para poder emplear este modo debemos conocer tanto el grafo computacional de la función $f(x_1, x_2)$ como el valor de las variables intermedias $v_i$ para un cierto punto $(x_1, x_2)$.

Para explicar el proceso, usaremos la red neuronal de la Figura 2.7. En esta red tenemos dos entradas, una salida, y solo una capa oculta con dos neuronas. Trataremos de hallar la derivada de la salida $h_1^{(2)}$, respecto a una de sus entradas, $h_1^{(0)} = x_1 = v_{-1}$, ya que esta operación será ampliamente empleada en el capítulo 4. El gráfico computacional de esta red lo podemos encontrar en la Figura 2.8, donde también podemos ver el proceso como combinación de funciones elementales de las variables intermedias $v_i$.

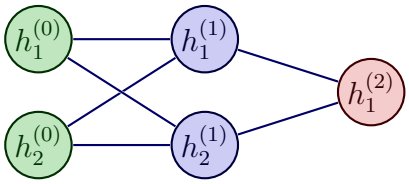


Figura 2.7: Red neuronal sencilla para la que emplearemos el modo de acumulación hacia atrás.

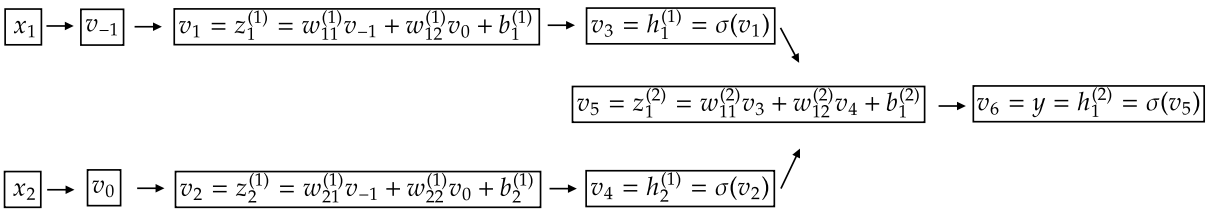


Figura 2.8: grafo computacional de la función de la salida de la red 2.7.

Estamos interesados en obtener la contribución $\bar{v}_i = \frac{\partial y}{\partial v_i}$ al cambio en la salida, $y$, debido al cambio de cada variable intermedia, $v_i$. Para el caso, por ejemplo, de $v_1$, podemos ver en la Figura 2.8 que la única manera que tiene de afectar a $y$ es afectando a $v_3$, de forma que su contribución al cambio de $y$ es

$$\frac{\partial y}{\partial v_1} = \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_1} \quad \text{o bien} \quad \bar{v}_1 = \bar{v}_3 \frac{\partial v_3}{\partial v_1}. \tag{2.75}$$

Debemos tener cuidado con las variables intermedias que dependen de más de una variable intermedia anterior. Por ejemplo, en la Figura 2.8 podemos ver que tanto $v_1$ como $v_2$ dependen de $v_0$, de forma que tenemos que tener en cuenta ambas contribuciones a la forma en que una variación en $v_0$ puede modificar el valor de $y$. Esto se hace calculando $\bar{v}_0$ en dos pasos,

$$\bar{v}_0 = \bar{v}_{0,1} + \bar{v}_{0,2}, \quad \text{donde} \quad \bar{v}_{0,1} = \bar{v}_1 \frac{\partial v_1}{\partial v_0}, \ \bar{v}_{0,2} = \bar{v}_2 \frac{\partial v_2}{\partial v_0}. \tag{2.76}$$

Lo mismo ocurre con $v_{-1}$ por lo que debemos hacer el mismo proceso en dos pasos. El proceso completo es el siguiente:

$$\bar{v}_6 = \bar{y} = 1 \tag{2.77}$$

$$\bar{v}_5 = \bar{v}_6 \frac{\partial v_6}{\partial v_5} = \sigma'(v_5) \tag{2.78}$$

$$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \sigma'(v_5) w_{12}^{(2)} \tag{2.79}$$

$$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \sigma'(v_5) w_{11}^{(2)} \tag{2.80}$$

$$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \sigma'(v_5) w_{12}^{(2)} \sigma'(v_2) \tag{2.81}$$

$$\bar{v}_1 = \bar{v}_3 \frac{\partial v_3}{\partial v_1} = \sigma'(v_5) w_{11}^{(2)} \sigma'(v_1) \tag{2.82}$$

$$\bar{v}_{0,1} = \bar{v}_1 \frac{\partial v_1}{\partial v_0} = \sigma'(v_5) w_{11}^{(2)} \sigma'(v_1) w_{12}^{(1)} \tag{2.83}$$

$$\bar{v}_0 = \bar{v}_{0,1} + \bar{v}_2 \frac{\partial v_2}{\partial v_0} = \sigma'(v_5) \left[ w_{11}^{(2)} \sigma'(v_1) w_{12}^{(1)} + w_{12}^{(2)} \sigma'(v_2) w_{22}^{(1)} \right] \tag{2.84}$$

$$\bar{v}_{-1,1} = \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = \sigma'(v_5) w_{11}^{(2)} \sigma'(v_1) w_{11}^{(1)} \tag{2.85}$$

$$\bar{v}_{-1} = \bar{v}_{-1,1} + \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} = \sigma'(v_5) \left[ w_{11}^{(2)} \sigma'(v_1) w_{11}^{(1)} + w_{12}^{(2)} \sigma'(v_2) w_{21}^{(1)} \right] \tag{2.86}$$

De esta forma, después del paso hacia delante, ejecutamos el paso hacia atrás calculando todas las variables $\bar{v}_i$, empezando con $\bar{v}_6 = \bar{y} = \frac{\partial y}{\partial y} = 1$. Así obtenemos la cantidad que buscábamos, $\bar{v}_{-1} = \frac{\partial y}{\partial v_{-1}} = \frac{\partial h_1^{(2)}}{\partial x_1}$.

No obstante, observamos que con todos los adjuntos intermedios que hemos ido calculando podemos obtener de igual forma la derivada respecto a la otra entrada, $\bar{v}_0$, en solo una ejecución del algoritmo. Esto hace que el modo hacia atrás sea más adecuado que el modo hacia adelante para hallar derivadas de funciones con varias entradas. Además, tenemos que tener en cuenta que todos los parámetros necesarios para hallar la derivada son aquellos de la red, de forma que una vez entrenada esta, tenemos todos los componentes necesarios para hallar la derivada.

En el caso extremo, $f : \mathbb{R}^n \to \mathbb{R}$, solo un paso del modo de acumulación hacia atrás es necesario para hallar el gradiente completo, comparado con los $n$ pasos necesarios con el modo de acumulación hacia atrás. Esto es por lo que es tan ampliamente usando en el campo de la inteligencia artificial, en concreto del aprendizaje profundo, ya que muchas veces es necesario hallar gradientes de funciones dependientes de millones de parámetros y con pocas salidas. Esto lo ha convertido en el algoritmo de propagación hacia delante más usado [46].

De forma general, para una función $f : \mathbb{R}^n \to \mathbb{R}^m$, si denotamos al número de operaciones necesarias para evaluar la función original como $\text{ops}(f)$, el tiempo que se tarda en calcular la matriz Jacobiana $m \times n$ por el modo de acumulación hacia adelante es $n \cdot c \cdot \text{ops}(f)$, mientras que la misma operación puede realizarse con el modo de acumulación hacia atrás en $m \cdot c \cdot \text{ops}(f)$ operaciones, siendo $c$ una constante tal que $c < 6$,

típicamente $c \in [2, 3]$ [50]. Esto hace que el modo de acumulación hacia atrás tenga un mejor desempeño cuando $m \ll n$.

## 2.4. Ejemplo ilustrativo

Con los fundamentos establecido, pongamos un ejemplo del uso de una red neuronal. Emplearemos el conjunto de datos *Iris*, uno de los datasets más empleados para la evaluación de métodos de clasificación. Está compuesto por 3 clases de 50 instancias cada una, siendo cada clase una especie de planta Iris: *Setosa*, *Versicolour* o *Virginica*. Una de las clases es linealmente separable de las otras, mientras que las dos restantes no son separebles linealmente entre sí [51]. Cada instancia del conjunto de datos tiene cuatro datos: el ancho y el largo del pétalo y el ancho y el largo del sépalo de cada flor, en centímetros. Por simplicidad, usarmos solo los datos para el pétalo. Además, cada elemento incluye un valor de objetivo, es decir, a qué especie pertenece. Esto está codificado en forma de entero, de forma que el valor del objetivo será $y = 0, 1, 2$ en función de si la instancia considerada pertence a la especie de Setosa, Versicolour o Virginica, respectivamente. No obstante, nosotros usaremos la llamada codificación *one-hot*, en la que cada etiqueta $0, 1$ o 2 viene representada con un vector de ceros menos el lugar cuyo índice es la etiqueta, es decir, $\mathbf{y} = (y_1, y_2, y_3) = (1, 0, 0), (0, 1, 0), (0, 0, 1)$, respectivamente. Podemos ver una representación de los datos en la Figura 2.9.

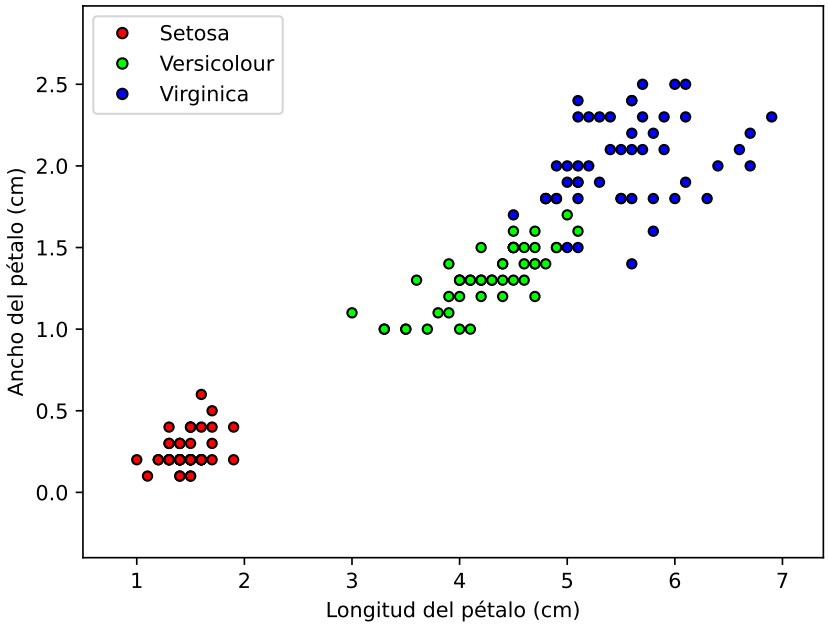


Figura 2.9: Anchura frente a longitud del pétalo del dataset Iris.

Nuestro objetivo será el siguiente: a partir del conjunto de datos anterior, entrenar a una red neuronal de forma que dada una nueva planta Iris que no se encuentra en el conjunto de datos, sea capaz de clasificar correctamente a qué especie pertenece. Trataremos,

por tanto, de contruir el mapa

$$\begin{aligned} f : \mathbb{R} \times \mathbb{R} &\to \mathbb{R}^3, \\ [x_1 \times x_2] &\mapsto (\bar{y}_1, \bar{y}_2, \bar{y}_3), \quad \bar{y}_i \in [0, 1], \end{aligned} \tag{2.87}$$

donde la salida sea un vector de probabilidades. Entonces, para una cierta entrada $\mathbf{x} = (x_1, x_2)$, la especie en la que se clasifique la entrada será $\text{argmax}(f(\mathbf{x})) = \text{argmax}(\bar{\mathbf{y}})$, donde

$$\underset{i \in \{1,\dots,n\}}{\text{argmax}} (\bar{\mathbf{y}}) = \{i \mid \bar{y}_i \geq \bar{y}_j \ \forall j \in \{1, \dots, n\}\}. \tag{2.88}$$

La red neuronal empleada tendrá cuatro capas. La capa de entrada tendrá dos neuronas, ya que hemos dicho que solo usaremos dos datos de entrada, la longitud y anchura del pétalo. Las dos capas ocultas tendrán diez neuronas cada una, y su función de activación será la función ReLU,

$$\text{ReLU}(z) = \text{máx}(0, z). \tag{2.89}$$

La capa de salida debe tener tres neuronas, una por cada elemento del vector de salida, y su función de activación será la función Softmax,

$$\text{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{\#\mathbf{z}} e^{z_j}}. \tag{2.90}$$

Esta función de activación nos convierte la salida en una densidad de probabilidad, de forma que la suma las salidas de las tres neuronas sume uno. Esto es, cada neurona calcula el valor de su suma ponderada $z_i$, y una vez tenemos la sumas ponderadas de todas las neuronas, aplicamos las función softmax al vector $\mathbf{z} = (z_1, z_2, z_3)$ con todas ellas. Después, nos quedaremos con el índice del mayor elemento con la función $\text{argmax}(f(\mathbf{x}))$. Esto causa que cada neurona de la última capa quede asociada a una especie, en el sentido de que si la primera neurona tiene una activación mayor que el resto, la entrada será entendida como perteneciente a la especie Setosa; si la neurona con mayor activación es la segunda, la entrada será clasificada como Versicolour, y como Virginica para la misma situación con la tercera neurona. Estos y otro parámetros de la red y el entrenamiento están recogidos a modo de resumen en la Tabla 2.3.

### Propagación hacia delante

De una manera más detallada, siendo $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$ los datos de la longitud y la anchura de los pértalos de una flor de la especie Iris, la salida de la primera capa oculta se halla de la siguiente forma:

$$\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{W}^{(1)} \cdot \mathbf{x} + \mathbf{b}^{(1)}) \in \mathbb{R}^{10}, \tag{2.91}$$

de acuerdo a la notación explicada en el apartado 2.2.2, y entendiendo que la función de activación actúa elemento a elemento. De igual forma, la salida de la siguiente capa oculta será

$$\begin{aligned} \mathbf{h}^{(2)} &= \text{ReLU}(\mathbf{W}^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}) \\ &= \text{ReLU}(\mathbf{W}^{(2)} \cdot \text{ReLU}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}) \in \mathbb{R}^{10}. \end{aligned} \tag{2.92}$$

| **Parámetro/característica** | **Valor/tipo** |
|---|---|
| Optimizador | Adam |
| Tasa de aprendizaje ($\eta$) | 0,01 |
| Capas ocultas | 2 |
| Neuronas de las capas ocultas | 10 |
| Función de activación de las capas ocultas | ReLU |
| Neuronas de la capa de salida | 3 |
| Función de activación de la capa oculta | Softmax |
| Épocas de entrenamiento | 50 |
| Datos de entrenamiento | 150 |
| Tamaño del minibatch | 5 |
| Función de pérdida | Error cuadrático medio |
| Parámetros totales de la red | 173 |

Tabla 2.3: Parámetros de la red y del entrenamiento.

En este punto, antes de obtener la salida de la red, cada neurona de la última capa calcula su respectiva suma ponderada:

$$\begin{aligned} z_i^{(3)} &= \sum_{j=1}^{n_{L-1}=10} w_{i,j}^{(3)} h_i^{(2)} + b_i^{(3)} \\ &= \sum_{j=1}^{n_{L-1}=10} w_{i,j}^{(3)} \left[\mathrm{ReLU}(\mathbf{W}^{(2)} \cdot \mathrm{ReLU}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)})\right]_j + b_i^{(3)}, \ i = 1, 2, 3. \end{aligned} \tag{2.93}$$

Una vez tenemos las activacion es de las neuronas de la capa de salida, aplicamos la función Softmax de la forma

$$\mathbf{h}^{(3)} = \text{softmax } (\mathbf{z}^{(3)}) = (\bar{y}_1, \bar{y}_2, \bar{y}_3) \in \mathbb{R}^3, \tag{2.94}$$

obteniendo un vector cuyos elementos cumples $\sum_{i=1}^{3} y_i = 1$. Si el mayor elemento del vector se salida es el primero, la entrada se clasifica como Setosa. Para el segundo, Versicolour, y para el tercero, Virginica.

### Cálculo de la pérdida y propagación hacia atrás

Una vez hemos hallado la salida de la red para un ejemplo en particular, podemos calcular el valor de la función de pérdida y hacer propagación hacia atrás. Como elejimos un minibatch de cinco elementos, la función de pérdida se calculará de la siguiente forma:

$$L = \frac{1}{5}\sum_{i=1}^{5} L_i = \frac{1}{5}\sum_{i=1}^{5}\left(\sum_{j=1}^{3}(h_j^{(L)}(\mathbf{x}_i) - y_j(\mathbf{x}_i))\right). \tag{2.95}$$

Para disminiur su valor necesitamos hallar su gradiente, lo cual se explicó en profundidad en el apartado 2.3.2. Como caso particular pongamos que queremos hallar la derivada de la función de coste respecto al primer peso de cada una de las primeras neuronas, es decir, el peso que cada neurona asigna a la primera neurona de la capa anterior. Lo haremos pora un caso particular $L_0$, ya que solo habrá que promediar después sobre todos los elementos del minibatch. Podemos comprobar que para la última capa tiene la forma

$$\begin{aligned}\frac{\partial L_0}{\partial w_{1,1}^{(3)}} &= \frac{\partial z_1^{(3)}}{\partial w_{1,1}^{(3)}}\frac{\partial h_1^{(3)}}{\partial z_1^{(3)}}\frac{\partial L_0}{\partial h_1^{(3)}} \\ &= h_1^{(2)}\mathrm{ReLU}'(z_1^{(3)})2(h_1^{(3)} - y_1).\end{aligned} \tag{2.96}$$

Para la penúltima capa, o segunda capa oculta, tendremos lo siguiente:

$$\begin{aligned}\frac{\partial L_0}{\partial w_{1,1}^{(2)}} &= \frac{\partial z_1^{(2)}}{\partial w_{1,1}^{(2)}}\frac{\partial h_1^{(2)}}{\partial z_1^{(2)}}\frac{\partial L_0}{\partial h_i^{(3)}} \\ &= \frac{\partial z_1^{(2)}}{\partial w_{1,1}^{(2)}}\frac{\partial h_1^{(2)}}{\partial z_1^{(2)}}\sum_{i=1}^{n_L=3}\frac{\partial z_i^{(3)}}{\partial h_1^{(2)}}\frac{\partial h_i^{(3)}}{\partial z_i^{(3)}}\frac{\partial L_0}{\partial h_i^{(3)}} \\ &= h_1^{(1)}\mathrm{ReLU}'(z_1)^{(2)}\sum_{i=1}^{n_L=3} w_{i,1}^{(3)}\mathrm{ReLU}'(z_i^{(3)}2(h_i^{(3)} - y_i)).\end{aligned} \tag{2.97}$$

Y para la antepenúltima capa, o primera capa oculta, tendríamos

$$\begin{aligned}\frac{\partial L_0}{\partial w_{1,1}^{1}} &= \frac{\partial z_1^{(1)}}{\partial w_{1,1}^{(1)}}\frac{\partial h_1^{(1)}}{\partial z_1^{(1)}}\frac{\partial L_0}{\partial h_1^{(1)}} \\ &= \frac{\partial z_1^{(1)}}{\partial w_{1,1}^{(1)}}\frac{\partial h_1^{(1)}}{\partial z_1^{(1)}}\sum_{j=1}^{n_{L-1}=10}\frac{\partial z_j^{(2)}}{\partial h_1^{(}1)}\frac{\partial h_j^{(2)}}{\partial z_j^{(2)}}\frac{\partial L_0}{\partial h_j^{(2)}} \\ &= \frac{\partial z_1^{(1)}}{\partial w_{1,1}^{(1)}}\frac{\partial h_1^{(1)}}{\partial z_1^{(1)}}\left[\sum_{j=1}^{n_{L-1}=10}\frac{\partial z_j^{(2)}}{\partial h_1^{(}1)}\frac{\partial h_j^{(2)}}{\partial z_j^{(2)}}\sum_{i=1}^{n_L=3}\frac{\partial z_i^{(3)}}{\partial h_j^{(2)}}\frac{\partial h_i^{(3)}}{\partial z_i^{(3)}}\frac{\partial L_0}{\partial h_i^{(3)}}\right] \\ &= x_1\mathrm{ReLU}'(z_1^{(1)})\left[\sum_{j=1}^{N_{L-1}=10} w_{j,1}^{(2)}\mathrm{ReLU}'(z_j^{(2)})\sum_{i=1}^{n_L=3} w_{i,j}^{(3)}\mathrm{ReLU}'(z_i^{(3)})2(h_i^{(3)} - y_i)\right]\end{aligned} \tag{2.98}$$

Una vez calculado estas cantidades para todos los parámetros de la red, empleamos el optimizador escogido, en este caso Adam, para actualizar los valores de los parámetros

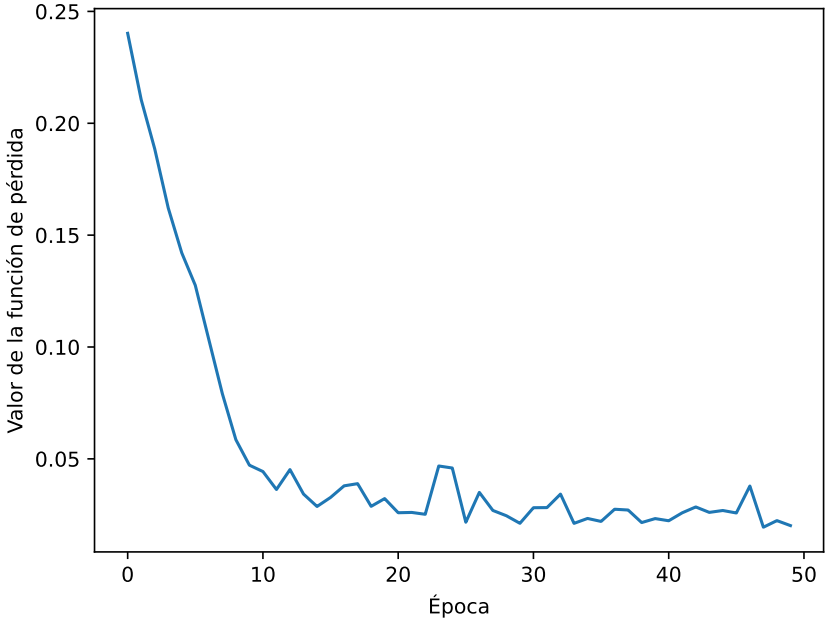


Figura 2.10: Evolución del valor de la pérdida durante el periodo de entrenamiento.

de forma que disminuyamos el valor de la función de coste. En este caso, la evolución del valor de la función de pérdida la podemos ver en la Figura 2.10.

Una vez finalizado el entrenamiento, podemos evaluar la red neuronal en una región del plano y sombrear de un color determinado las regiones en las que la entrada sea clasificada como una especie determinada. Si hacemos esto, obtenemos la Figura 2.11, en la que podemos observar que la red ha dividido el plano en tres regiones, una por cada posible opción. Esto es, cada flor cuyos datos de longitud y anchura de sus pétalos se encuentre en la región roja será clasificada como Setosa, en la región verde, Versicolour, y en la región morada, Virginica. Observamos que para la Setosa, la región sombreada envuelve completamente los datos de entrenamiento, mientras que para las otras dos especies esto no ocurre, ya que existe un intervalo donde ambas especies coinciden. Para consegur un mejor resultado, habría que mirar al resto de entradas, ya que entre Versicolour y Virginica no es posible distinguir solo teniendo los datos de longitud y anchura de los pétalos.

De esta forma, hemos logrado una red neuronal que ha aprendido a distinguir entre tres especies de plantas a partir de un pequeño conjunto de datos, de forma que dado un nuevo caso no presente en el conjunto de datos de entrenamiento es capa de clasificarlo de forma probablemente correcta.

## 2.5. Redes neuronales como aproximadores de funciones

### 2.5.1. Teorema de aproximación universal

Viéndolo desde una perspectiva más general, podemos pensar en una red neuronal como una "máquina" que trata de aproximar una función, de forma que recibe un cierto

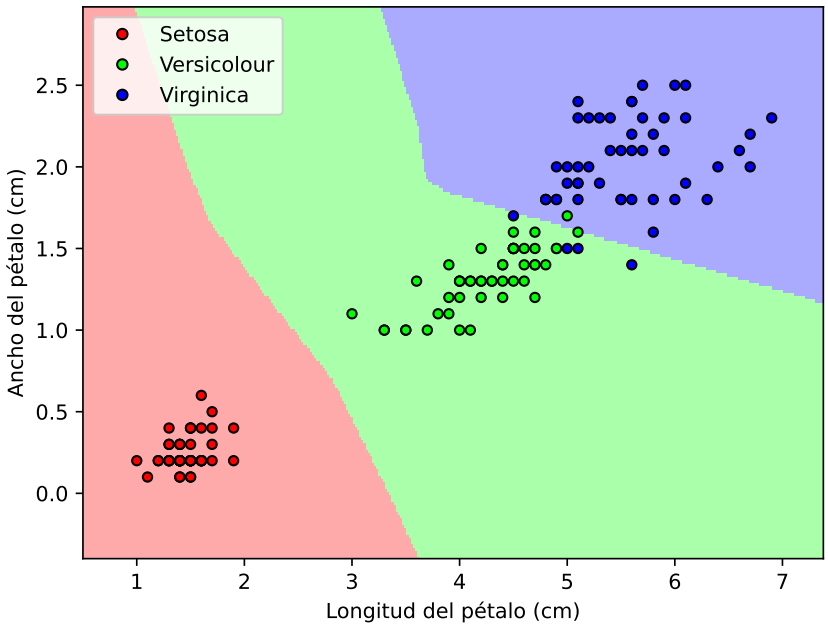


Figura 2.11: Clasificación realizada por la red neuronal una vez entrenada (zonas sombreadas).

número de entradas y da como resultado un cierto número de salidas basándose en las reglas subyacentes aprendidas de observar los datos de entrenamiento. De esta forma, nos podríamos preguntar cuál es el límite, es decir, cuáles son las funciones que pueden ser aproximadas por una red neuronal y cuáles no. La respuesta a esta pregunta nos la da el *Teorema de Aproximación Universal* [52, 53].

**Teorema 2.5.1** (Aproximación Universal)**.** *Sea* $\{x_1, \ldots, x_n\}$ *un conjunto de puntos distintos en* $\mathbb{R}^r$*, y sea* $g : \mathbb{R}^r \to \mathbb{R}$ *una función arbitraria.*

*Sea* $f : \mathbb{R}^r \longrightarrow \mathbb{R}$ *una red neuronal con una sola capa oculta, de forma que solo esta capa tiene función de activación. Sea* $\sigma : \mathbb{R} \longrightarrow [0, 1]$ *la función de activación de la capa oculta tal que* $\sigma(\cdot)$ *es no decreciente y cumple*

$$\lim_{x\to\infty} \sigma(x) = 1, \quad \lim_{x\to-\infty} \sigma(x) = 0. \tag{2.99}$$

*Si* $\sigma$ *alcanza* 0 *y* 1*, entonces existe una red neuronal* $f$ *con* $n$ *neuronas ocultas y un* $\varepsilon > 0$ *tal que* $|f(x_i) - g(x_i)| < \varepsilon$*, con* $i \in \{1, \ldots, n\}$*.*

Esto es, una red neuronal con al menos una capa oculta y un número arbitrario de neuronas en ella puede aproximar una función arbitraria con un error tan pequeño como queramos. La demostración del teorema se puede encontrar en [52].

Originalmente era necesario que la función de activación cumpliera las condiciones expuestas en el teorema 2.5.1. Sin embargo, se demostró que estos eran casos concretos de un resultado más general: mientras la función de activación sea continua a trozos y localmente acotada, una red neuronal prealimentada como la descrita en el teorema anterior puede aproximar cualquier función continua con precisión arbitraria si y solo si la función de activación no es un polinomio [54].

Es importante darse cuenta de que el teorema 2.5.1 nos habla solo del conjunto de puntos $\{x_1, \ldots, x_n\}$, que será el conjunto de entrenamiento. Esto es, no nos garantiza que si extrapolamos fuera de ese intervalo la aproximación de la red neuronal sea buena [21].

Por último, aunque el teorema pide que la red neuronal tenga al menos una capa oculta, empíricamente se comprueba que redes de mayor profundidad (mayor número de capas) obtienen, en general, mejores resultados con el mismo conjunto de entrenamiento. Esto se puede ver en la Figura 2.12, en cuyo artículo [55] se probó a reconocer dígitos en imágenes de Google Street View.

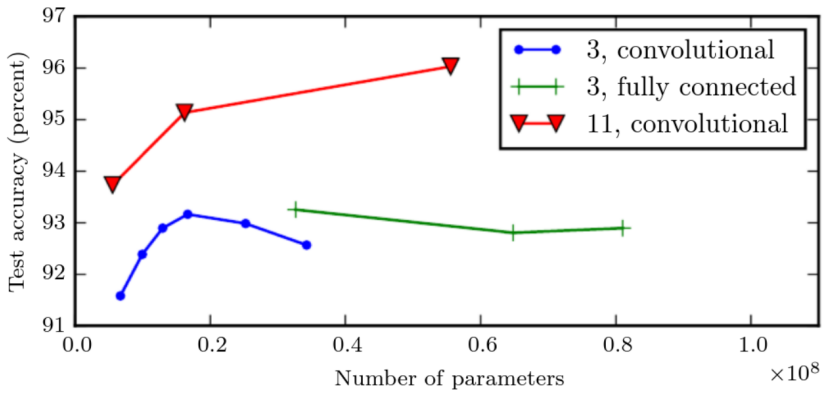


Figura 2.12: Precisión frente a número de parámetros para distintas arquitecturas de redes neuronales en función del número de capas y parámetros totales. La leyenda indica el número de capas de cada red neuronal. Figura de [21] adaptada de [55].

En la Figura 2.12 puede sorprender que la red neuronal con 3 capas empeore su precisión al aumentar el número de parámetros. Este comportamiento es un indicio de *sobrea juste*, descrito brevemente en el apartado 2.1.1.

### 2.5.2. El teorema de "no hay almuerzo gratis"

Mediante el proceso de entrenamiento, una red neuronal encuentra unas reglas que son probablemente correctas para la mayoría de los casos del conjunto de datos con el que se ha entrenado [21]. No obstante, nada garantiza que si aplicamos este algoritmo a un caso distinto para el que se ha entrenado, su desempeño deba ser bueno. Este razonamiento intuitivo viene reflejado en los resultados conocidos como *Teoremas de No Hay Almuerzo Gratis para la Optimización* (*No Free Lunch Theorems for Optimization*). En [56] se explica como para cualquier algoritmo, un desempeño superior sobre una clase de problemas es compensado por su desempeño en otra clase de problemas, de forma que no existe un algoritmo "universal" óptimo para todos los casos.

Esto hace que debamos elegir el tipo de algoritmo que mejor se adapta a nuestro problema para poder conseguir un buen resultado.

## 2.6. Inteligencia artificial en la Física

La Física ha sido una gran impulsora de técnicas de aprendizaje automático debido a la necesidad de analizar datos experimentales. Históricamente, laboratorios como el CERN

lideraron los intentos de atajar los problemas del *big data*, ya que en experimentos como el detector ATLAS se llegan a generar más de 60 terabytes por segundo [10]. Esto obliga a desallorar térnicas avanzadas de tratamiento de datos, muchas de las cuales están dentro del marco del aprendizaje automática y el aprendizaje profundo.

Las redes neuronales han sido un recurso ampliamente utilizado dentro de áreas de la Física como la física de altas energías desde finales de la década de 1980 [10, 57], siendo en la actualidad una herramienta ampliamente utilizada. Por ejemplo, en 2023 se logró entrenar una red neuronal para distinguir la generación del tetraquark top ($t\bar{t}t\bar{t}$), una partícula con baja probabilidad de producirse, del ruido de fondo causado por las cerca de 600 millones de colisiones por segundo que ocurren en el LHC [58]. También se está estudiando el uso de redes neuronales para reconstruir las colisiones con datos procedentes de los calorímetros de los detectores. En futuras actualizaciones se aumentará la luminosidad del colisionador (colisiones por unidad de tiempo), lo que obliga a emplear técnicas de análisis cada vez más rápidas. En este sentido, el uso de redes neuronales parece un camino a considerar debido a la baja latencia que se puede lograr [59].

En astrofísica también podemos encontrar el uso de térnicas de aprendizaje profundo. El lanzamiento en el año 2013 del satélite GAIA permitió, y continúa haciéndolo, la obtención de la posición, el movimiento, la luminosidad, la temperatura y la composición del orden de mil millones ($10^9$) de estrellas [60]. Esta cantidad de datos empuja al uso de técnicas lo más sofisticadas posibles para en análisis, entre las que se pueden encontrar las redes neuronales. Por ejemplo, en [11] se emplearemos técnicas de aprendizaje automático y profundo para derivar la metalicidad de 270905 estrellas RR Lyrae (estrellas que varían su brillo periódicamente) a partir de datos de variación del brillo el función del tiempo.

También, en [61] se comenta cómo redes neuronales pueden formar parte del sistema de control del plasma en los futuros reactores de fusión. En concreto, en estos reactores se debe controlar en tiempo real la estabilidad del plasma, cosa que era casi imposible debido al elevado coste computacional de los modelos teóricos. El uso de una red neuronal especialmente entrenada para esta tarea permitiría acelerar mucho este proceso, llegando incluso a predecir la inestabilidad 300 ms antes de que ocurra.

Los ejemplos anteriores se pueden incluir dentro de los campos de análisis de datos y clasificación, pero en los últimos años han surgido nuevas arquitecturas de redes neuronales que permiten su uso como método numérico, como son las *redes neuronales diferenciales* (del inglés, *Differential Neural Networks*, DNN) o las *redes neuronales informadas en la física* (*Physics-informed Neuronal Networks*, PINN), sobre las que hablaremos en los capítulos siguientes.

# Capítulo 3

# *Ecuaciones diferenciales ordinarias neuronales*

La gran mayoría de procesos en Física están modelados a través de una o varias ecuaciones diferenciales. Muchas veces es común encontrarse en la situación en la que el problema es demasiado complejo, de forma que encontrar su resolución analítica puede no resultar práctico (o posible). Ante esta tesitura el uso de métodos numéricos, algoritmos mediante los cuales podemos obtener valores numéricos aproximados de la solución a nuestro problema original, son una técnica muy empleada.

En 2018 se publicó un artículo [12] donde se explica un nuevo método numérico para ecuaciones diferenciales ordinarias basado en el uso de redes neuronales. Los autores denominaron a esta arquitectura *Ecuación Diferencial Ordinaria Neuronal* (*Neural Ordinary Differential Equations*, ODENet). En este capítulo explicaremos su funcionamiento.

## 3.1. ¿Qué es una ecuación diferencial ordinaria neuronal?

Una *ecuación diferencial ordinaria neuronal* (*neural ordinary differential equations*, ODENet) es una ecuación diferencial ordinaria (o sistema de ellas) donde se emplea una red neuronal para parametrizar el campo vectorial [62],

$$\frac{d\mathbf{y}}{dt}(t) = f(t, \mathbf{y}(t), \boldsymbol{\theta}), \tag{3.1}$$

$$\mathbf{y}(0) = \mathbf{y}_0 \tag{3.2}$$

En la anterior expresión, $\boldsymbol{\theta} \in \mathbb{R}^m$ representa un vector de parámetros de la red, $t \in [0, T] \in \mathbb{R}$ la variable independiente, $f : \mathbb{R} \times \mathbb{R}^{n_k} \times \mathbb{R}^m \to \mathbb{R}^{n_L}$ es la red neuronal, de forma que tenemos

$$f(t, \mathbf{y}(t), \boldsymbol{\theta}) = (f_1(t, \mathbf{y}(t), \boldsymbol{\theta}), \ldots, f_{n_L}(t, \mathbf{y}(t), \boldsymbol{\theta})), \tag{3.3}$$

y finalmente $\mathbf{y} : [0, T] \to \mathbb{R}^{n_L}$ es la solución.

### 3.1.1. Similitud con otras arquitecturas de redes neuronales

Este planteamiento puede resultar forzado a primera vista, pero lo cierto es que ciertos métodos numéricos en ecuaciones diferenciales ordinarias siguen una estructura similar a algunas redes neuronales, como las *redes neuronales residuales* (*Residual Neural Network*, ResNet).

Motivados por el hecho de que las redes neuronales más profundas pueden tener dificultades en aprender respecto a otras con menor número de capas, los autores de [27] propusieron la arquitectura ResNet.

Sea $H(\mathbf{x})$ la función que queremos aproximar con nuestra red neuronal. Teóricamente, no hay problema en considerar que la red aproxime $g(\mathbf{x}) := H(\mathbf{x}) - \mathbf{x}$, de forma que la función original se convierte en $H(\mathbf{x}) = g(\mathbf{x}) + \mathbf{x}$ (para el caso en que el número de entradas y salidas de la red sea distinto, el razonamiento es similar [27]). Aunque el resultado de aprender una u otra función pueda ser asintóticamente igual, el proceso de aprendizaje no tiene por qué, y de hecho experimentalmente se demuestra que no es así [27], lo que nos permite hacer esta traslación.

Este tipo de redes neuronales se organizan en capas agrupadas en *bloques residuales*. Cada bloque está formado por un conjunto de capas y una *conexión de atajo*, como podemos ver en la Figura 3.1. De esta forma, si llamamos $\mathbf{h}$ a la salida del bloque de la

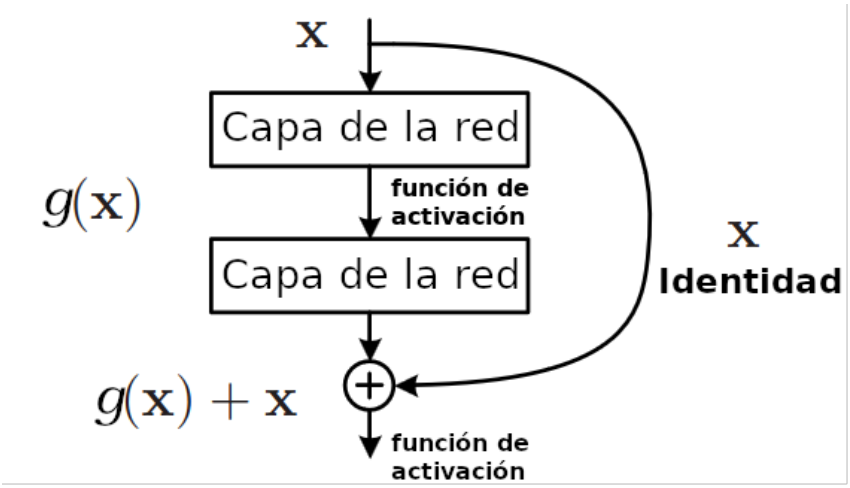


Figura 3.1: Bloque de una red residual. Imagen tomada de [27], traducida y adaptada a nuestra notación.

Figura 3.1, esta tendría la forma

$$\mathbf{h} = g(\mathbf{x}, \boldsymbol{\theta}) + \mathbf{x}, \tag{3.4}$$

donde hemos explicitado el vector de parámetros $\boldsymbol{\theta}$. Si iteramos esta misma estructura a lo largo de una red con suficientes capas como para tener varios bloques residuales, la estructura de las salidas de cada uno estos bloques sería

$$\mathbf{h}_{j+1} = \mathbf{h}_j + g(j, \mathbf{h}_j, \boldsymbol{\theta}_j), \tag{3.5}$$

donde $\mathbf{h}_j$ y $f(j, \cdot, \cdot)$ son el $j$-ésimo estado oculto y bloque residual, respectivamente.

Podemos comparar esta expresión con el método de Euler de resolución numérica de EDOs. Discretizando temporalmente la expresión (3.1), de forma que $t_{j+1} - t_j = \Delta t$ es el paso temporal, tenemos, para una cierta condición inicial $\mathbf{y}_0$:

$$\frac{\mathbf{y}(t_{j+1}) - \mathbf{y}(t_j)}{\Delta t} \approx \frac{d\mathbf{y}}{dt}(t_j) = f(t_j, \mathbf{y}(t_j), \boldsymbol{\theta}), \tag{3.6}$$

de forma que

$$\mathbf{y}(t_{j+1}) = \mathbf{y}(t_j) + \Delta t f(t_j, \mathbf{y}(t_j), \boldsymbol{\theta}). \tag{3.7}$$

Absorbiendo $\Delta t$ dentro de $f$ y renombrando $\mathbf{y} \to \mathbf{h}$, $f \to g$, recuperamos la formulación de la ecuación (3.5).

De esta forma, en el límite en que añadimos más capas y tomamos pasos temporales más pequeños, podemos parametrizar la dinámica "continua" de las capas ocultas de la red usando una ecuación diferencial ordinaria especificada por una red neuronal. Explícitamente, tendríamos

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \boldsymbol{\theta}), \ \mathbf{h}(t_0) = \mathbf{h_0}, \tag{3.8}$$

empezando por la capa de entrada a tiempo $t = 0$, $\mathbf{h}(0)$, y pudiendo definir la capa de salida $\mathbf{h}(T)$ como la solución de un problema de valores iniciales del tipo (3.1) para un tiempo $T$. Este valor puede ser calculado por un algoritmo de resolución numérica de EDOs, que consideraremos como una "caja negra", evaluando la dinámica de las capas ocultas, $f$, las veces que vea necesario para determinar con la precisión deseada nuestra solución a tiempo $T$. Con este punto de vista, el número de capas de la red neuronal podría verse como el número de evaluaciones que haga el algoritmo de resolución numérica que empleemos.

## 3.2. Propagación hacia atrás en una ecuación diferencial ordinaria neuronal

Toda red neuronal necesita ser entrenada. Para ello, debemos emplear propagación hacia atrás y un optimizador, por ejemplo, descenso del gradiente, explicado en el apartado 2.3.1, respectivamente. Sin embargo, aunque la idea es la misma, el proceso por el que se lleva a cabo es distinto.

Como hemos dicho antes, trataremos al algoritmo de resolución de EDOs como una caja negra, sin importar cómo realiza la resolución. Privarnos de la información de cómo funciona dicho algoritmo podría parecer ilógico, sin embargo se trata de una forma de generalizar. La resolución numérica de EDOs es una rama que lleva muchos años, en la que se han ideado algoritmos muy sofisticados y en la que se encontrarán mejores en el futuro. Particularizando a un método en concreto sería, a largo plazo, contraproducente. Además, este planteamiento tiene otros beneficies, como menos coste de memoria, mejor control de errores numéricos y un escalado lineal con el tamaño del problema [12].

Cuando nos refiramos al algoritmo de resolución numérica también lo haremos con el nombre de *ODESolver*.

### 3.2.1. Método de sensibilidad adjunta. Propagación hacia atrás continua en el tiempo

El método de sensibilidad adjunta nos servirá para realizar la propagación hacia atrás de nuestra función de pérdida [12]. En primer lugar, recordar que si tenemos un problema de valor inicial general como el de la expresión (3.8), la solución exacta para un tiempo $t_1$ será

$$\mathbf{h}(t_1) = \mathbf{h_1} = \mathbf{h_0} + \int_{t_0}^{t_1} f(\mathbf{h}(t), t, \boldsymbol{\theta})dt. \tag{3.9}$$

Consideramos ahora que queremos minimizar una cierta función de coste $L(\cdot)$, cuyas entradas serán la salida del algoritmo de resolución de ODEs que estemos usando, $\mathbf{h}(t_1) = \text{ODESolver}(\mathbf{h}(t_0), f, t_0, t_1)$, y el correspondiente dato de entrenamiento para ese mismo punto, $\hat{\mathbf{h}}(t_1)$. Explícitamente,

$$L(\mathbf{h_1}, \hat{\mathbf{h}}_1) = L\left(\mathbf{h_0} + \int_{t_0}^{t_1} f(\mathbf{h}(t), t, \boldsymbol{\theta})dt,\ \hat{\mathbf{h}}_1\right) = L[\text{ODESolver}(\mathbf{h}_0, f, t_0, t_1),\ \hat{\mathbf{h}}_1]. \tag{3.10}$$

Para optimizar $L$ necesitamos hallar su gradiente respecto los parámetros $\boldsymbol{\theta}$. En primer lugar, determinamos cómo la variación de la función de coste depende del estado $\mathbf{h}(t)$ para cada instante de tiempo. Llamaremos *adjunto* a esta cantidad,

$$\mathbf{a}(t) = -\frac{\partial L}{\partial \mathbf{h}(t)} = -\left(\frac{\partial L}{\partial h_1}, \ldots, \frac{\partial L}{\partial h_{n_L}}\right). \tag{3.11}$$

Después nos será útil la cantidad $\frac{d\mathbf{a}}{dt}$, por lo que la hallamos continuación. En el apartado 2.3.2 vimos que el gradiente de la capa $\mathbf{h}(t)$ depende de la capa $\mathbf{h}(t+1)$ aplicando la regla de la cadena:

$$\frac{dL}{d\mathbf{h}(t)} = \frac{dL}{d\mathbf{h}(t+1)}\frac{d\mathbf{h}(t+1)}{d\mathbf{h}(t)}. \tag{3.12}$$

Para el caso continuo de las ODENet podemos escribir el estado tras un tiempo $\varepsilon$ como

$$\mathbf{h}(t+\varepsilon) = \int_t^{t+\varepsilon} f(\mathbf{h}, t, \boldsymbol{\theta})dt + \mathbf{h}(t) = T_\varepsilon(\mathbf{h}(t), t), \tag{3.13}$$

de forma que la regla de la cadena puede ser aplicada de la misma manera,

$$\frac{dL}{d\mathbf{h}(t)} = \frac{dL}{d\mathbf{h}(t+\varepsilon)}\frac{d\mathbf{h}(t+\varepsilon)}{d\mathbf{h}(t)} \quad \text{o bien} \quad \mathbf{a}(t) = \mathbf{a}(t+\varepsilon)\frac{\partial T_\varepsilon(\mathbf{h}(t), t)}{\partial \mathbf{h}(t)}. \tag{3.14}$$

De esta forma podemos hallar la cantidad requerida [12]:

$$\frac{d\mathbf{a}(t)}{dt} = \lim_{\varepsilon\to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} \tag{3.15}$$

$$= \lim_{\varepsilon\to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial T_\varepsilon(\mathbf{h}(t))}{\partial \mathbf{h}(t)}}{\varepsilon} \tag{3.16}$$

$$= \lim_{\varepsilon\to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial}{\partial \mathbf{h}(t)}\left(\mathbf{h}(t) + \varepsilon f(\mathbf{h}(t), t, \boldsymbol{\theta}) + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \tag{3.17}$$

$$= \lim_{\varepsilon\to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\left(I + \varepsilon\frac{\partial}{\partial \mathbf{h}(t)} f(\mathbf{h}(t), t, \boldsymbol{\theta}) + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \tag{3.18}$$

$$= \lim_{\varepsilon\times 0^+} \frac{-\mathbf{a}(t+\varepsilon)\left(\varepsilon\frac{\partial}{\partial \mathbf{h}(t)} f(\mathbf{h}(t), t, \boldsymbol{\theta}) + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \tag{3.19}$$

$$= \lim_{\varepsilon\to 0^+} -\mathbf{a}(t+\varepsilon)\frac{\partial f(\mathbf{h}(t), t, \boldsymbol{\theta})}{\partial \mathbf{h}(t)} + \mathcal{O}(\varepsilon) \tag{3.20}$$

$$= -\mathbf{a}(t)\frac{\partial f(\mathbf{h}(t), t, \boldsymbol{\theta})}{\partial \mathbf{h}(t)}, \tag{3.21}$$

donde al pasar de (3.16) a (3.17) hemos usado la ecuación (3.14), y en el paso de (3.17) a (3.18) hemos hecho un desarrollo de Taylor alrededor de $\mathbf{h}(t)$.

Pasamos ahora a obtener los gradientes necesarios. En primer lugar, podemos ver a $\boldsymbol{\theta}$ y $t$ como estados con ecuaciones diferenciales constantes, de forma que

$$\frac{\partial \boldsymbol{\theta}(t)}{dt} = \left(\frac{\partial \theta_1}{\partial t}, \ldots, \frac{\partial \theta_m}{\partial t}\right) = \mathbf{0}, \quad \frac{dt(t)}{dt} = 1. \tag{3.22}$$

En la anterior expresión hemos usado $t$ como variable dependiente e independiente. Esto es un abuso de notación que mantendremos para no saturar, ya que el significado se entiende por el contexto. Lo que hacemos ahora será combinar las dos expresiones anteriores con la correspondiente en $\mathbf{h}$ para construir lo que se llama *estado aumentado* $[\mathbf{h}, \boldsymbol{\theta}, t]\,(t)$, de forma que su adjunto y ecuación diferencial correspondiente son, respectivamente,

$$\mathbf{a}_{\text{aug}} := \begin{bmatrix} \mathbf{a} \\ \mathbf{a}_\theta \\ \mathbf{a}_t \end{bmatrix} = \begin{bmatrix} \frac{dL}{d\mathbf{h}(t)} \\ \frac{dL}{d\boldsymbol{\theta}(t)} \\ \left(\frac{dL}{dt(t)}, 0, \ldots, 0\right) \end{bmatrix} \tag{3.23}$$

$$\frac{d}{dt}\begin{bmatrix} \mathbf{h} \\ \boldsymbol{\theta} \\ t \end{bmatrix}(t) = f_{\text{aug}}\left([\mathbf{h}, \boldsymbol{\theta}, t]\right) := \begin{bmatrix} f\left([\mathbf{h}, \boldsymbol{\theta}, t]\right) \\ \mathbf{0} \\ (1, 0, \ldots, 0) \end{bmatrix} \tag{3.24}$$

El Jacobiano de $f_{\text{aug}}$ tendrá la forma

$$\frac{\partial f_{\text{aug}}}{\partial [\mathbf{h}, \boldsymbol{\theta}, t]} = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{h}} & \frac{\partial f}{\partial \boldsymbol{\theta}} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}(t), \tag{3.25}$$

donde $\mathbf{0}$ es una matriz de ceros de las dimensiones adecuadas. Podemos comprobar [12] que el razonamiento que llevó a la expresión (3.21) sigue siendo cierta para el estado aumentado, de forma que tenemos

$$\frac{d\mathbf{a}_{\text{aug}}}{dt} = -\begin{bmatrix}\mathbf{a}(t) & \mathbf{a}_\theta(t) & \mathbf{a}_t(t)\end{bmatrix}\frac{\partial f_{\text{aug}}}{\partial[\mathbf{h},\boldsymbol{\theta},t]}(t) = -\begin{bmatrix}\mathbf{a}\frac{\partial f}{\partial \mathbf{h}} & \mathbf{a}\frac{\partial f}{\partial \boldsymbol{\theta}} & \mathbf{a}\frac{\partial f}{\partial t}\end{bmatrix}(t). \tag{3.26}$$

El primer elemento del resultado anterior es el resultado obtenido previamente para la evolución del adjunto, ecuación (3.21), de donde podemos obtener el gradiente de la función de coste respecto a cualquier de los estados ocultos integrando sobre todo el intervalo $[t_0, t_N],\ t_0 < t_N$. En particular, para el instante inicial $t_0$ y final $t_N$ obtenemos

$$\mathbf{a}(t_N) = \frac{dL}{d\mathbf{h}(t_N)}, \quad \mathbf{a}(t_0) = \mathbf{a}(t_N) + \int_{t_N}^{t_0}\frac{d\mathbf{a}(t)}{dt}dt = \mathbf{a}(t_N) - \int_{t_N}^{t_0}\mathbf{a}(t)\frac{\partial f(\mathbf{h}(t),t,\boldsymbol{\theta})}{\partial \mathbf{h}(t)}. \tag{3.27}$$

En la anterior expresión parece que los límites de integración deberían estar al revés, pero no es así. Hablaremos de esto al final del apartado.

El segundo elemento de (3.26) puede ser empleado para hallar el gradiente de la función de coste con respecto a los parámetros $\boldsymbol{\theta}$, integrando sobre todo el intervalo $[t_N, t_0],\ t_0 < t_N$, y estableciendo $\mathbf{a}_\theta(t_N) = \mathbf{0}$,

$$\frac{d}{dt}\mathbf{a}_\theta = -\mathbf{a}\frac{\partial L}{\partial \boldsymbol{\theta}} \tag{3.28}$$

$$\frac{d}{dt}\left(\frac{dL}{d\boldsymbol{\theta}}\right) = -\frac{dL}{d\mathbf{h}}\frac{\partial f}{\partial \boldsymbol{\theta}}, \tag{3.29}$$

$$\frac{dL}{d\boldsymbol{\theta}} = -\int_{t_N}^{t_0}\frac{dL}{d\mathbf{h}}\frac{\partial f}{\partial \boldsymbol{\theta}}dt = -\int_{t_N}^{t_0}\mathbf{a}(t)\frac{\partial f(\mathbf{h}(t),t,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}dt, \tag{3.30}$$

de forma que para el parámetro $k$-ésimo tendríamos

$$\frac{dL}{d\theta_k} = -\int_{t_N}^{t_0}\frac{dL}{d\mathbf{h}}\frac{\partial f}{\partial \theta_k}dt = \int_{t_N}^{t_0}\left(\frac{\partial L}{\partial h_1},\dots,\frac{\partial L}{\partial h_k}\right)\cdot\left(\frac{\partial f_1(\mathbf{h}(t),t,\boldsymbol{\theta})}{\partial \theta_k},\dots,\frac{\partial f_{n_L}(\mathbf{h}(t),t,\boldsymbol{\theta})}{\partial \theta_k}\right)dt, \tag{3.31}$$

Finalmente, también podemos hallar los gradientes de la función de coste con respecto a $t_0$ y $t_N$, obteniendo lo siguiente,

$$\frac{dL}{dt_N} = \mathbf{a}(t_N)f(\mathbf{h}(t_N),t_N,\boldsymbol{\theta}), \quad \frac{dL}{dt_0} = \mathbf{a}_t(t_0) = \mathbf{a}_t(t_N) - \int_{t_N}^{t_0}\mathbf{a}(t)\frac{\partial f(\mathbf{h}(t),t,\boldsymbol{\theta})}{\partial t}dt. \tag{3.32}$$

De esta forma, hemos hallado los gradientes de todas las posibles entradas de un algoritmo de resolución de un problema de valor inicial.

### 3.2.2. Algoritmo para el método de sensibilidad adjunta

Normalmente la forma de obtener la salida de nuestra red neuronal era introduciendo una entrada $\mathbf{h}(t_0)$ y haciendo la propagación hacia delante de la forma vista en el apartado 2.2.2. Sin embargo, este proceso ahora no es posible debido a que nuestra red neuronal no modela el comportamiento de $\mathbf{h}(t)$ sino de su variación con respecto del tiempo, $^{d\mathbf{h}}/_{dt}$.

No obstante, podemos usar el *ODESolver* en nuestra red para obtener el valor final a un tiempo $t_N$, $\mathbf{h}(t_N)$. Atendiendo a la parte superior de la Figura 3.2a, la forma en la que nos movemos de la parte izquierda (condición inicial $\mathbf{h}(t_0)$) a la parte derecha (punto donde queremos hallar el valor aproximado de $\mathbf{y}(t_N)$, $\mathbf{h}(t_N)$) es llamando a nuestro *ODESolver*.

Una vez obtenida nuestra estimación, lo que queremos hacer es propagar hacia atrás el error, de forma que minimicemos la función de pérdida $L(\cdot)$. Como nuestro *ODESolver* es una caja negra, nosotros no tenemos acceso a las operaciones que realiza para obtener la solución, por lo que la obtención del gradiente de la función de coste la realizaremos mediante el método del adjunto.

Comenzamos definiendo el parámetro $\mathbf{a}(t_N) = {}^{\partial L}/_{\partial \mathbf{h}(t_N)}$, cantidad que siempre podremos obtener porque la función de pérdida debe depender explícitamente de la salida $\mathbf{h}(t_N)$. De esta forma, podemos tratar a este adjunto como una función, tal que si hallamos cómo varía respecto del tiempo podemos integrar hacia atrás (con $\mathbf{a}(t_N)$ como valor inicial del PVI) y obtener la derivada de la función de coste respecto a la entrada, ${}^{\partial L}/_{\partial \mathbf{h}(t_0)}$. Esto viene reflejado en la parte inferior de la Figura 3.2a.

Generalizando el proceso por el cual hallamos cómo varía la cantidad $\mathbf{a} = {}^{\partial L}/_{\partial \mathbf{h}}$ podemos obtener el gradiente de la función de pérdida $L$ con respecto a los parámetros $\boldsymbol{\theta}$, ${}^{dL}/_{d\boldsymbol{\theta}}$, que es lo que realmente nos interesa para poder emplear el método del descenso del gradiente u otro que nos convenga más.

En resumen, teniendo la estimación $\mathbf{h}(t_N)$, el procedimiento de propagación hacia atrás es el Algoritmo 1.

**Algoritmo 1** Algoritmo de propagación hacia atrás con el método del adjunto [12]

**INPUT**: parámetros $\boldsymbol{\theta}$, tiempo inicial, $t_0$, y final, $t_N$, salida $\mathbf{h}(t_N)$, gradiente ${}^{\partial L}/_{\partial \mathbf{h}(t_N)}$

$s_0 = \left[\mathbf{h}(t_N), \frac{\partial L}{\partial \mathbf{h}(t_1)}, \mathbf{0}_{|\theta|}\right]$ ▷ Definimos el estado aumentado inicial

**def** dinamica_aug($[\mathbf{h}(t), \mathbf{a}(t), \cdot], t, \theta$) ▷ Dinámica del estado aumentado

**return** $\left[f(\mathbf{h}(t), t, \theta), -\mathbf{a}(t)\frac{\partial f}{\partial \mathbf{h}}, -\mathbf{a}(t)\frac{\partial f}{\partial \boldsymbol{\theta}}\right]$ ▷ Hallar los productos vectoriales

$\left[\mathbf{h}(t_0), \frac{\partial L}{\partial \mathbf{h}(t_0)}, \frac{\partial L}{\partial \boldsymbol{\theta}}\right] = \text{ODESolver}(s_0, \text{dinamica\_aug}, t_1, t_0, \boldsymbol{\theta})$ ▷ EDO con $t$ invertido

**OUTPUT**: $\frac{\partial L}{\partial \mathbf{h}(t_0)}, \frac{\partial L}{\partial \boldsymbol{\theta}}$ ▷ Obtenemos los gradientes

Si la función de pérdida $L(\cdot)$ dependiera directamente de múltiples observaciones, $(\mathbf{h}(t_1), \mathbf{h}(t_2), \ldots, \mathbf{h}(t_1))$, el adjunto debe ser actualizado en la dirección ${}^{\partial L}/_{\partial \mathbf{h}(t_i)}$ para cada $i = 1, 2, \ldots, N$. Este caso está reflejado en la Figura 3.2b.

## 3.3. Ejemplo

Para ilustrar el funcionamiento y los resultados que se pueden obtener con este formalismo, veamos un ejemplo.

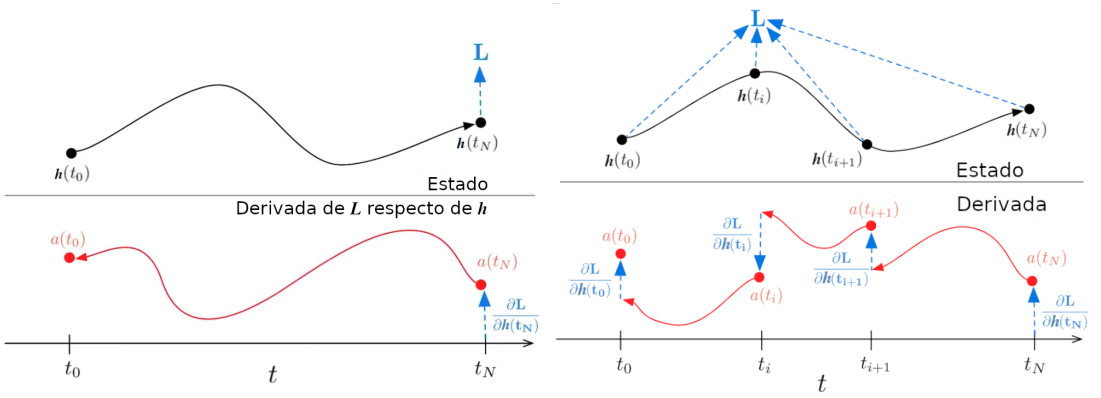


Figura 3.2: Propagación hacia atrás con el método del adjunto. Imágenes sacadas de [12], traducida y adaptadas a la notación empleada.

El PVI del que nos encargaremos será el siguiente,

$$\frac{dx}{dt} = -0{,}15x - y, \tag{3.33}$$

$$\frac{dy}{dt} = x - 2y^3, \tag{3.34}$$

$$(x_0, y_0) = (1, 0). \tag{3.35}$$

Podemos ver la trayectoria que usaremos de entrenamiento, junto al plano de fases alrededor de esta, en la Figura 3.3.

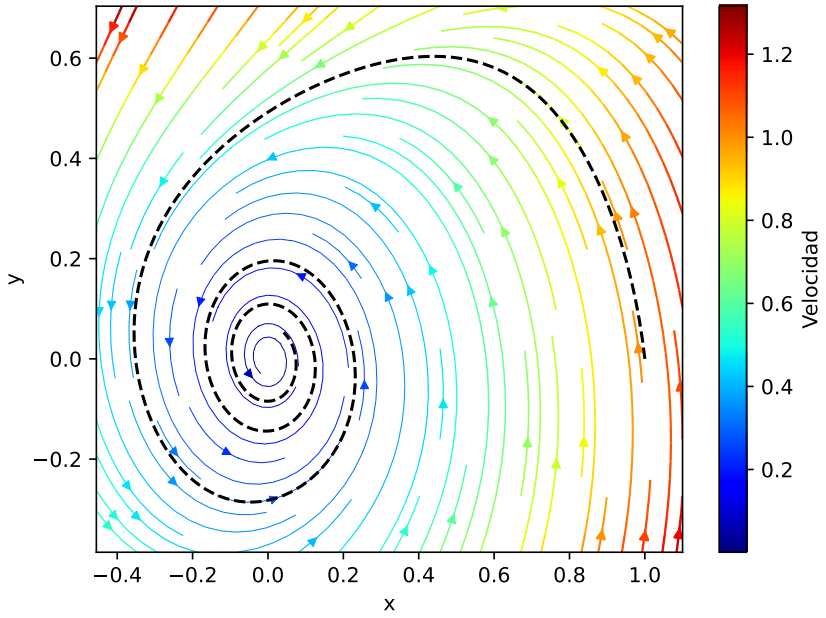


Figura 3.3: Plano de fases del sistema de ecuaciones diferenciales de ejemplo.

Para la implementación de la ODENet emplearemos la librería de Python `torchdiffeq`

[63], la cual implementa el método del adjunto explicado en el apartado anterior. El código empleado se puede encontrar en [1].

Trataremos de entrenar a la ODENet con el primer tercio de la trayectoria de la Figura 3.3, y comprobaremos la capacidad de extrapolar al resto de la trayectoria.

En primer lugar, tomamos 500 puntos equiespaciados en $t \in [0, 25/3]$, empezando nuestra trayectoria en el punto $(x, y) = (1, 0)$. En la Figura 3.4 podemos apreciar cómo a medida que va a avanzando el entrenamiento, el cual duró 400 épocas, la ODENet consigue aproximar cada vez mejor el plano de fases.

El resultado final, junto con la extrapolación en el intervalo temporal $t \in [25/3, 20]$ la podemos encontrar en la Figura 3.5.

Este caso está muy alejado de lo que nos podríamos encontrar en la realidad, ya que las medidas nunca coinciden con el valor real debido al error o ruido asociado. En el caso de la Figura 3.6 hemos planteado una situación más normal, con un número menor de medidas, 50, no equiespaciadas en el tiempo y a las que se les ha añadido un ruido gaussiano con desviación típica igual al 5 % del valor en cada punto.

En la Tabla 3.1 podemos ver algunos detalles más de la red empleada y de los entrenamientos.

| **Parámetro/característica** | **Valor/tipo** |
|---|---|
| Optimizador | Adam |
| Tasa de aprendizaje ($\eta$) (caso sin ruido - caso con ruido) | $10^{-3}$ - $10^{-2}$ |
| ODESolver | Runge-Kutta de orden 5 de Dormand-Prince-Shampine |
| Datos de entrenamiento (caso sin ruido - caso con ruido) | 500 - 100 |
| Capas ocultas | 2 |
| Neuronas de las capas ocultas | 20 |
| Función de activación de las capas ocultas | Tangente hiperbólica |
| Neuronas de la capa de entrada | 2 |
| Neuronas de la capa de salida | 2 |
| Función de pérdida | Error cuadrático medio |

Tabla 3.1: Parámetros de la red y del entrenamiento para las ODENet.

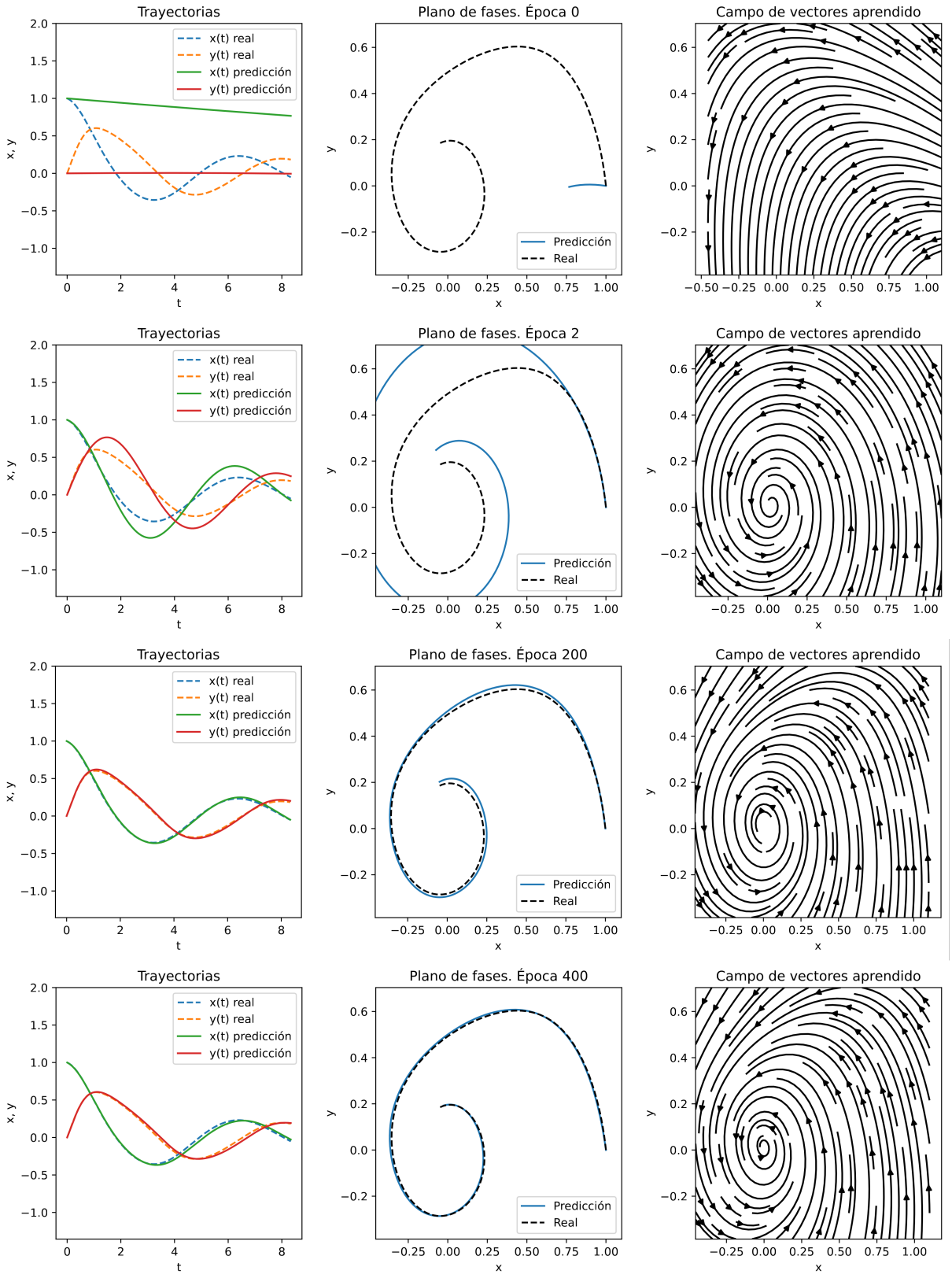


Figura 3.4: Evolución del aprendizaje de la ODENet.

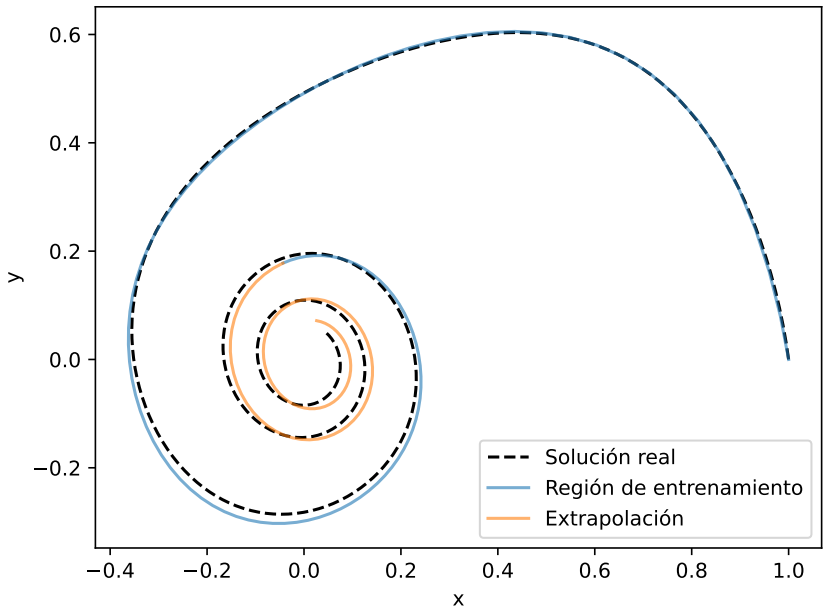


Figura 3.5: Estado final de la ODENet para el intervalo de entrenamiento y el de extrapolación.

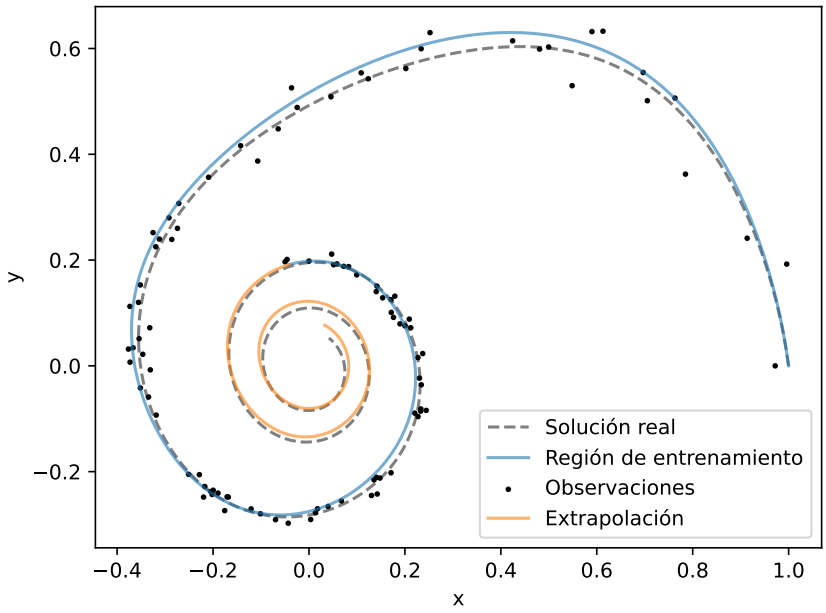


Figura 3.6: Estado final de la ODENet para el intervalo de entrenamiento y el de extrapolación para 50 medidas con un 5 % de ruido.

# Capítulo 4

# *Redes Neuronales Informadas en Física*

Muchos de los últimos avances del aprendizaje profundo, como pueden ser la generación de imágenes o de texto, han sido posibles en gran medida gracias a dos recursos que se han hecho muy accesibles en los últimos años: una elevada potencia computacional y una ingente cantidad de datos. Aun así, no siempre se está en la situación de disponer de todos los datos de los que uno gustaría, ya que la obtención de estos puede resultar muy costosa. En esta situación, sacar toda la información posible de los datos disponibles se vuelve una tarea fundamental.

Es muy habitual que al estudiar un fenómeno de naturaleza física dispongamos de un conocimiento previo. Desde el muy general como la conservación de la energía, al más particular como que cierto parámetro no pueda ser negativo. De implementar esta información dentro de nuestra red neuronal, podríamos amplificar la información que esta ve en los datos, pudiendo sacar mucho más provecho de estos. Este planteamiento, además de las conocidas capacidades de las redes neuronales como aproximadores universales, llevaron a la creación de las *redes neuronales informadas en física* [16] (*Physics Informed Neural Networks*, PINNs), pudiendo aparecer también como "basadas en física", "guiadas por física" o "guiadas por la teoría" [64]. A continuación describimos su funcionamiento.

## 4.1. Contexto y estado del arte

Las PINNs son una técnica científica de aprendizaje profundo empleado para resolver problemas en los que están involucradas ecuaciones diferenciales en derivadas parciales (EDPs). Una red neuronal informada en física aproxima la solución de una EDP minimizando una función de pérdida a la que incluimos un término extra respecto a lo que hemos explicado en apartados anteriores: además del término que ya hemos descrito de pérdida respecto a los datos de entrenamiento, que pueden ser condiciones iniciales, de frontera o dentro del propio dominio espacio-temporal, incluye un término donde se incorpora las leyes físicas que gobiernan el problema. Esto permite comparar el resultado obtenido con el esperado según nuestro conocimiento físico, penalizando aquellas soluciones que no concuerden con la ecuación del sistema. Esencialmente, es una técnica de resolución de EDPs que convierte el problema de resolver directamente la ecuación en un problema de minimizar una función de pérdida. En este método no es necesario definir un mallado del

dominio espacial, lo que puede simplificar mucho el proceso comparado con métodos más tradicionales como el método de elementos finitos [64].

La idea de incorporar conocimiento previo a una red neuronal no es nueva, pudiendo remontarse a mediados de la década de 1990 donde ya se convertía el problema de resolver la EDP es uno de minimización, calculando la pérdida usando un método quasi-Newtoniano y evaluando los gradientes mediante diferencias finitas [65]. Estos avances fue altamente inspirado por los resultados de la capacidad de aproximación universal de las redes neuronales, de la que hablamos en el apartado 2.5.1. En esa misma década se publicaron artículos en esta misma línea [64].

La combinación de la mejora del hardware, el descubrimiento de nuevas prácticas para mejorar el rendimiento durante el entrenamiento y la posibilidad de acceder a potentes bibliotecas de código abierto como `TensorFlow` [66] con técnicas como la diferenciación automática [46] implementadas, la investigación para resolver EDPs con redes neuronales no se detuvo [64]. Todos estos avances durante años permitieron en 2018 la publicación del artículo fundacional de las PINNs [16], cuyo formulación explicaremos más adelante. El formalismo introducido en este artículo permitía aplicar el método a cualquier problema modelado con una ecuación diferencial, lo que simplificó mucho la curva de accesibilidad y popularizó enormemente esta técnica [64].

En estos años se han publicado artículos acerca del potencial, las limitaciones y las aplicaciones de este nuevo método [67], como la modelización de fluidos, donde se han obtenido buenos resultados [68, 69]. También se ha tratado de ampliar la metodología para la resolución de ecuaciones integro-diferenciales [70, 71] y ecuaciones diferenciales estocásticas [72, 73]. Una revisión muchos más extensa sobre el tema puede encontrarse en [64].

## 4.2. Añadir información física a una red neuronal

En el artículo fundacional [16], realizan los razonamientos primero para modelos en tiempo continuo y luego para modelos en tiempo discreto. Nosotros nos centraremos en los primeros.

Numerosos sistemas físicos están descritos por una ecuación de la forma

$$f(\mathbf{x}, t) := \frac{d\mathbf{y}}{dt} + \mathcal{N}[\mathbf{y}; \boldsymbol{\lambda}] = \mathbf{0}, \tag{4.1}$$

$$\mathbf{y}(0) = \mathbf{y}_0, \tag{4.2}$$

donde $\mathcal{N}[\cdot]$ es un operador diferencial, $\mathbf{x} \in \Omega \subseteq \mathbb{R}^{n_0}$ una región del espacio, $t \in [0, T] \in \mathbb{R}$ la variable independiente, $\boldsymbol{\lambda} \in \mathbb{R}^p$ se refiere a los parámetros del modelo (no confundir con aquellos de la red neuronal, $\boldsymbol{\theta}$) e $\mathbf{y}(\mathbf{x}, t) \in \mathbb{R}^{n_L}$ es la solución buscada, la cual aproximaremos por una red neuronal, $\mathbf{h}(\mathbf{x}, t) \in \mathbb{R}^{n_L}$. Este supuesto, junto con la ecuación (4.1), nos define la *red neuronal informada en física* $f(x, t) : \mathbb{R}^{n_0} \times \mathbb{R} \to \mathbb{R}^{n_L}$, la cual podemos hallar una vez obtenida $\mathbf{y}$ mediante diferenciación automática, explicada en el capítulo 2.3.3.

La forma, por tanto, de incorporar el conocimiento físico a nuestra red neuronal es mediante la función de pérdida, la cual pasa a ser

$$L(\mathbf{h}(\mathbf{x}, t), \mathbf{y}(\mathbf{x}, t)) = L_h + L_f, \tag{4.3}$$

donde $L_h$ es la función de pérdida con respecto a los datos (la que hemos estado usando hasta ahora), y $L_f$ es la función de pérdida con respecto a la física, es decir, respecto a la expresión (4.1). Si estamos usando como función de pérdida el error cuadrático medio, tendríamos lo siguiente,

$$L_h = \frac{1}{N_y}\sum_{i=1}^{N_y} ||\mathbf{h}^{(L)}(\mathbf{x}_i, t_i) - \mathbf{y}_i||^2, \quad L_f = \frac{1}{N_f}\sum_{j=1}^{N_f} ||f(\bar{\mathbf{x}}_j, t_j)||^2. \tag{4.4}$$

En la anterior expresión $\{\mathbf{x}_i, t_i, \mathbf{y}_i\}_{i=1}^{N_y}$ denota el conjunto de datos de entrenamiento, siendo $N_y$ el número de intervalos temporales que tenemos en nuestro conjunto de entrenamiento. Por otra parte, $\{\mathbf{x}_j, t_j\}_{j=1}^{N_f}$ denota aquellos puntos en lo que evaluaremos la pérdida con respecto al conocimiento físico del sistema, expresión (4.1). Del mismo modo, $N_f$ es el número de intervalos temporales en lo que evaluaremos la pérdida física.

Incorporando la información física a la función de pérdida estamos perjudicando aquellos valores de la solución que no concuerden con nuestro conocimiento previo del sistema, lo que sumado a las observaciones $y_i$, permite a la red realizar mejores aproximaciones de la solución.

### 4.2.1. Soluciones frente a descubrimientos basados en datos

Con la configuración descrita en el apartado anterior y dados una serie de datos con los que estamos trabajando, podemos estar interesados en dos clases de resultados: *soluciones basadas en datos* o *descubrimiento basado en datos*. El primero se refiere a la situación en la que tenemos un conjunto de datos, posiblemente con ruido y no equiespaciados en el tiempo, y deseamos suavizarlos o hacer predicciones: dados unos valores para los parámetros $\boldsymbol{\lambda}$, ¿qué podemos decir acerca de la solución $y(\mathbf{x}, t)$? Por otro lado, el descubrimiento basado en los datos se asemeja más al ajuste de una ecuación a unos datos: ¿Qué valor de los parámetros $\boldsymbol{\lambda}$ describen mejor las observaciones? Esto lo podemos lograr incorporando los parámetros $\lambda$ del operador diferencial como un parámetro más de la red neuronal [16].

## 4.3. Ejemplo: Ecuación del calor unidimensional

El ejemplo en el que nos fijaremos en este apartado es el de la ecuación del calor para el caso unidimensional. En el tridimensional, dado un recinto espacial $\Omega \in \mathbb{R}^3$ con frontera $\Gamma$ y un intervalo temporal $[0, T]$, $T \in \mathbb{R}$ y una condición inicial $u_0 = u(\mathbf{x}, t)$, la llamada formulación fuerte del problema es la siguiente,

$$C\frac{\partial u}{\partial t} - \mathbf{K}\nabla^2 u = f(\mathbf{x}, t), \quad (\mathbf{x}, t) \in \Omega \times [0, T], \tag{4.5}$$

$$u = g(\mathbf{x}, t), \quad (\mathbf{x}, t) \in \Gamma \times [0, T], \tag{4.6}$$

donde $C$ es la capacidad calorífica del material (J/(kg·K)), $\rho$ la densidad del material (kg/m$^3$), y $\mathbf{K} = \frac{1}{\rho}\mathbf{k}$, donde $\mathbf{k}$ es la matriz de conductividades. Esta matriz tiene la siguiente forma,

$$\mathbf{k} = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{pmatrix}. \tag{4.7}$$

siendo $k_i$ es la conductividad térmica del material $(W/(m \cdot K))$ en el eje $i$. Por último, $f(\mathbf{x}, t)$ es el término de fuente, con el que podemos añadir energía al sistema, y $g(\mathbf{x}, t)$ es la condición de frontera.

Nosotros nos centraremos en el caso unidimensional siguiente,

$$C\frac{\partial u}{\partial t} - k_x\frac{\partial^2 u}{\partial x^2} = 0, \quad (\mathbf{x}, t) \in \Omega \times [0, T], \tag{4.8}$$

$$u = 0, \quad (\mathbf{x}, t) \in \Gamma \times [0, T], \tag{4.9}$$

$$u_0 = \sin(\pi x), \tag{4.10}$$

con $\Omega = [0, 2]$ y $T = 0{,}1$, es decir, una barra unidimensional con los extremos mantenidos a 0°C, sin término de fuente y cuyo perfil de temperaturas iniciales se puede ver en la Figura 4.1.

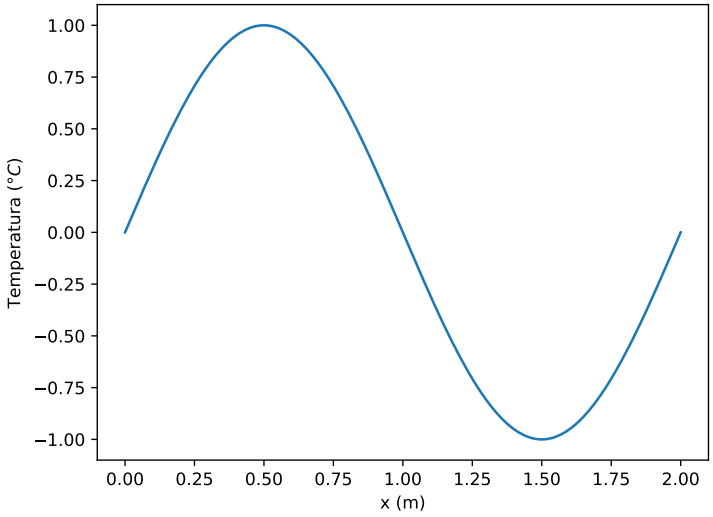


Figura 4.1: Condición inicial para nuestro caso.

### 4.3.1. Generación de los datos de entrenamiento

Para la generación de los datos usaremos diferencias finitas, concretamente el método de Euler explícito, para aproximar tanto la primera derivada temporal como la segunda derivada espacial.

Denotamos $\Delta t$ y $\Delta x$ al intervalo temporal y espacial, respectivamente, de forma que $t_{i+1} = t_i + \Delta t$, $x_{i+1} = x_i + \Delta x$. Se puede comprobar que obtenemos la siguiente regla de actualización:

$$u_i^{k+1} \approx u_i^k + \frac{\Delta t k_x}{C(\Delta x)^2}(u_{i-1}^k - 2u_i^k + u_{i+1}^k), \tag{4.11}$$

donde $u_i^k = u(x_i, t_k)$.

Durante la generación de los datos, los parámetros tomaron los valores recogidos en la Tabla 4.1, para los cuales se cumple la condición de estabilidad numérica [74]:

$$\Delta t \leq \frac{(\Delta x)^2}{2k_x} \tag{4.12}$$

.

| Parámetro | Valor |
|---|---|
| $\Delta t$ (s) | $10^{-8}$ |
| $\Delta x$ (m) | $10^{-2}$ |
| $C$ (J/(kg·K)) | 0,3 |
| $k_x$ ($m^2$/s) | 0,5 |

Tabla 4.1: Valores de los parámetros para la generación de los datos.

Una visualización de los datos de entrenamiento se puede ver en la Figura 4.2.

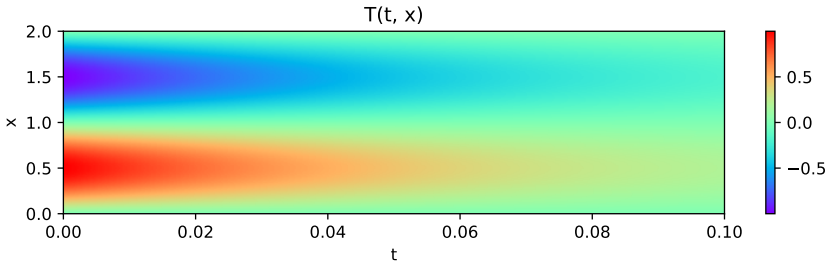


Figura 4.2: Datos de la simulación de la ecuación del calor unidimensional.

### 4.3.2. Implementación computacional

Pese a la relativa novedad de este tipo de redes, ya nos podemos encontrar algunas implementaciones de PINNs para librerías populares de Python, como `PyTorch` [75], `TensorFlow` [76] o `JAX` [77]. Nosotros usaremos la implementación en `JAX`, ya que parece ser la más rápida para casos similares al que estamos considerando [77]. De todas formas, como nuestro problema de ejemplo es relativamente simple, la diferencia entre implementar el problema en una librería o en otra no debería ser significativamente grande. Los programas empleados pueden encontrarse en [1].

### 4.3.3. Resultados

Como dijimos anteriormente, la función de pérdida que queremos minimizar está compuesta de dos términos, la función de pérdida respecto a los datos de entrenamiento y la función de pérdida respecto a la ecuación diferencial que domina el fenómeno. Esto es,

$$\begin{aligned} L(\mathbf{h}(\mathbf{x},t),\mathbf{y}(\mathbf{x},t)) = \frac{1}{N_y}\sum_{i=1}^{N_y} ||h^{(L)}(\mathbf{x}_i,t_i) - y_i(\mathbf{x}_i,t_i)||^2 \\ + \frac{1}{N_f}\sum_{j=1}^{N_f} \left|\left| C\,\frac{\partial h^{(L)}}{\partial t}\bigg|_{(\bar{\mathbf{x}}_j,t_j)} - k_x\,\frac{\partial^2 h^{(L)}}{\partial x^2}\bigg|_{(\bar{\mathbf{x}}_j,t_j)} \right|\right|^2, \end{aligned} \tag{4.13}$$

donde $\mathbf{x}_i$ son los puntos de los que tenemos datos de entrenamiento para tiempo $t_i$, mientras que $\bar{\mathbf{x}}_j$ son los puntos en lo que evaluaremos la pérdida física en el tiempo $t_j$.

En la Tabla 4.2 podemos ver algunas características comunes para los dos casos que vamos a tratar.

| **Parámetro/característica** | **Valor/tipo** |
|---|---|
| Optimizador | Adam |
| Tasa de aprendizaje ($\eta$) | $10^{-3}$ |
| Capas ocultas | 8 |
| Neuronas de las capas ocultas | 20 |
| Función de activación de las capas ocultas | Tangente hiperbólica |
| Neuronas de la capa de entrada | 2 |
| Neuronas de la capa de salida | 1 |
| Función de pérdida | Error cuadrático medio |

Tabla 4.2: Parámetros de la red y del entrenamiento para ambas PINNs.

### Soluciones basadas en datos

En primer lugar, tratamos de modelizar el comportamiento de la solución usando como datos de entrenamiento únicamente algunas temperaturas de la condición inicial y de frontera escogidos al azar. También seleccionamos una serie de puntos dentro de todo el dominio para evaluar la pérdida física, lo cual se hará conociendo tanto la ecuación que rige el fenómeno como el valor de sus parámetros $\boldsymbol{\lambda}$. Estos datos están recogidos en la Tabla 4.3.

La red neuronal que modeló la solución $u$ está formada por 10 capas, estando las 8 capas ocultas formadas por 20 neuronas cada una.

La Figura 4.3 recoge un resumen de los resultados obtenidos por la PINN, además de una comparación de la predicción de la red con los datos reales para tres momentos distintos. La red realizó 30000 épocas de entrenamiento.

Podemos calcular la diferencia media entre los datos de entrenamiento y la predicción de la red. Si hacemos esto, obtenemos el siguiente valor,

$$\frac{1}{N_t N_x} \sum_{i=1}^{N_t} \sum_{j=1}^{N_x} |u(x_j, t_i)_{\text{predicción}} - u(x_j, t_i)_{\text{entrenamiento}}| \approx 2 \cdot 10^{-3} \text{ °C}. \tag{4.14}$$

| Parámetros | Valor |
|---|---|
| Nº datos de temperatura inicial | $10^2$ |
| Nº datos de temperatura en la frontera | $10^2$ |
| Nº puntos de evaluación de la pérdida física | $10^4$ |
| Número de épocas de entrenamiento | $3 \cdot 10^4$ |
| Tiempo de ejecución | 226 s. |

Tabla 4.3: Información de los datos de entrenamiento empleados para obtener la solución basada en datos

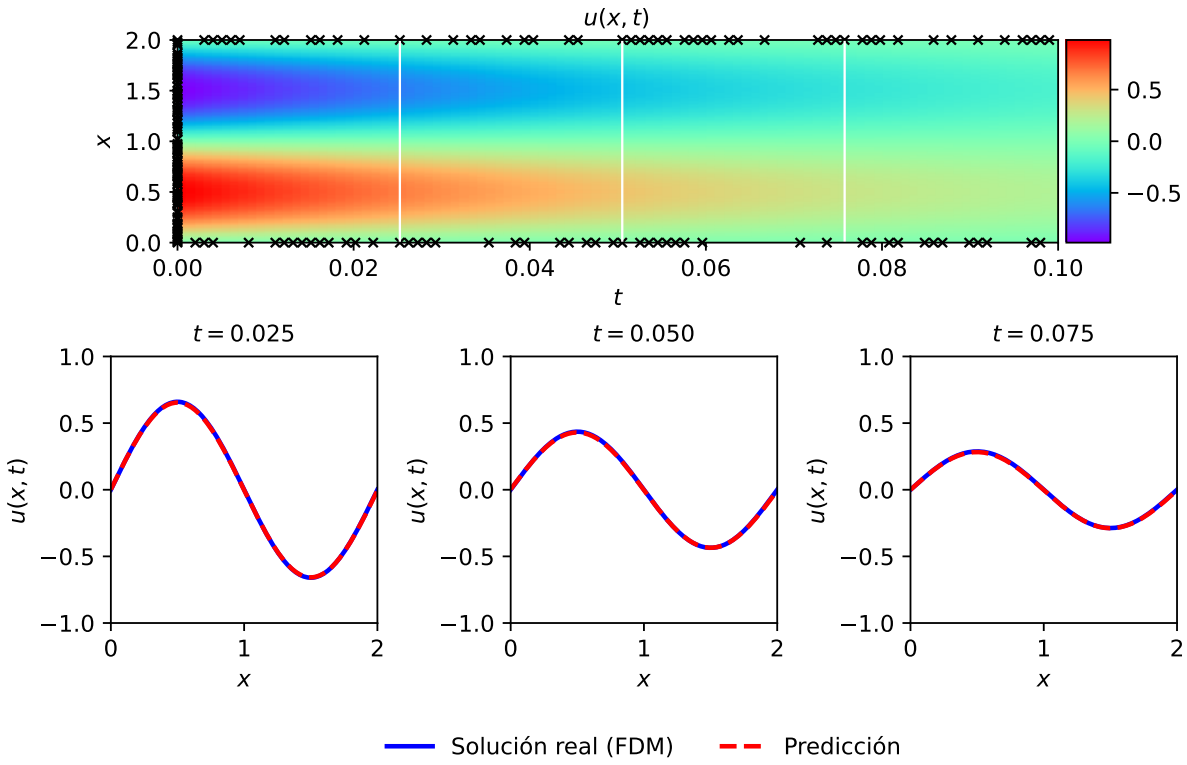


Figura 4.3: Salida de la PINN después del entrenamiento para la solución basada en datos.

Obtenemos que solo observando la condición inicial y la condición de frontera, la PINN es capaz de aproximar el comportamiento del sistema de forma correcta.

En la Tabla 4.4 hemos añadido cómo varía el error absoluto medio cuando los datos tienen ruido, algo esperable para el caso de medidas reales. En cada punto se escogió un valor al azar de una gaussiana con centro en el valor de la solución en ese punto y desviación típica igual a un porcentaje del valor en ese punto. Vemos una tendencia en

la que el error es en general mayor cuando también lo es el ruido, pero el aumento no es demasiado significativo, lo que indica que el método es resistente a la presencia de ruido en los datos. En [16] se hace un estudio similar variando el número de condiciones de contorno y condiciones iniciales que damos a la PINN, así como el número de capas y neuronas en cada capa. El resultado es el esperable: al dar más datos a la red, el error se reduce. Lo mismo ocurre al añadir más capas y más neuronas en cada una, aunque hay que tener en cuenta que al aumentar la profundidad de nuestra red aumentamos el coste computacional, de forma que el tamaño de esta debe ser un acuerdo entre el tiempo de computación y la precisión que queremos.

| **Ruido (%)** | **Error medio** |
|---|---|
| 10 | $4{,}7 \cdot 10^{-3}$ |
| 5 | $5{,}0 \cdot 10^{-3}$ |
| 3 | $1{,}8 \cdot 10^{-3}$ |
| 1 | $4{,}5 \cdot 10^{-3}$ |
| 0 | $3{,}1 \cdot 10^{-3}$ |

Tabla 4.4: Variación del error absoluto medio para las soluciones basadas en datos cuado los datos de entrenamiento tiene un ruido gaussiano.

### Descubrimiento basado en datos

Este caso será similar al del apartado anterior, pero no sabremos cuánto valen los parámetros $C$ y $k_x$ de la ecuación del calor, expresión (4.8). Además, en este caso no solo dispondremos de datos de la condición inicial, sino también de algunos puntos dentro del dominio, escogidos de nuevo al azar. El número de datos para caso está recogido en la Tabla 4.5.

De nuevo, la Figura 4.4 recoge un resumen de los resultados obtenidos, además de una comparación de los datos de entrenamiento con aquellos obtenidos de la PINN.

Si de nuevo calculamos la diferencia media entre las temperaturas de entrenamiento y las predichas por la red de la misma forma que hicimos en (4.14), obtenemos un valor aproximado de $4 \cdot 10^{-3}$ °C.

Un detalle importante sobre este ejemplo en particular es que al ser el término físico de la ecuación (4.18) suma de dos términos, cada uno multiplicado por un parámetro, la PINN se queda atrapada alrededor de la solución trivial $C = k_x = 0$, y no proporciona una solución válida. Una manera de solventar este problema es dividir la ecuación diferencial en (4.8) por uno de los parámetros, por ejemplo, $C$. De esta forma podemos reformular el problema,

$$\frac{\partial u}{\partial t} - \frac{K_x}{C}\frac{\partial^2 u}{\partial x^2} = 0, \quad (\mathbf{x}, t) \in \Omega \times [0, T], \tag{4.15}$$

$$u = 0, \quad (\mathbf{x}, t) \in \Gamma \times [0, T], \tag{4.16}$$

$$u_0 = \sin(\pi x), \tag{4.17}$$

| Parámetros | Valor |
|---|---|
| Nº datos de temperatura inicial | $10^2$ |
| Nº datos de temperatura dentro del dominio | $2 \cdot 10^3$ |
| Nº datos de evaluación de la pérdida física | $10^4$ |
| Número de épocas de entrenamiento | $2 \cdot 10^5$ |
| Tiempo de ejecución | 1623 s. |

Tabla 4.5: Información de los datos de entrenamiento empleados para obtener la solución basada en datos

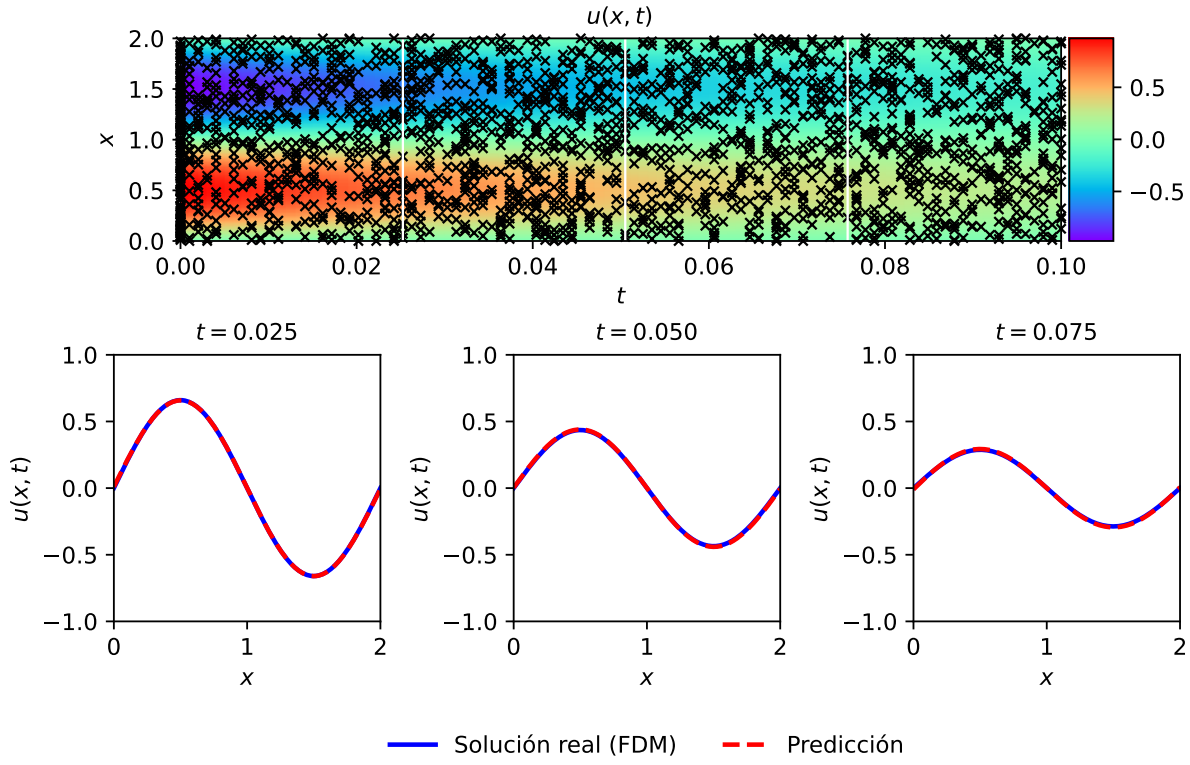


Figura 4.4: Salida de la PINN después del entrenamiento para el descubrimiento basada en datos.

y nuestra función de pérdida pasa a ser

$$L(\mathbf{h}(\mathbf{x},t),\mathbf{y}(\mathbf{x},t)) = \frac{1}{N_y}\sum_{i=1}^{N_y} ||h^{(L)}(\mathbf{x}_i,t_i) - y_i(\mathbf{x}_i,t_i)||^2 + \frac{1}{N_f}\sum_{j=1}^{N_f} \left|\left| \left.\frac{\partial h^{(L)}}{\partial t}\right|_{(\bar{\mathbf{x}}_j,t_j)} - \frac{k_x}{C}\left.\frac{\partial^2 h^{(L)}}{\partial x^2}\right|_{(\bar{\mathbf{x}}_j,t_j)} \right|\right|^2, \tag{4.18}$$

Con este pequeño cambio somos capaces de hallar una estimación del parámetro $k_x/C$, pese a que ahora estemos obteniendo menos información física del sistema.
El valor real de este parámetro es $\left(\frac{k_x}{C}\right)_{\text{real}} = 1.\bar{6}\ \text{m}^2/\text{s}$, mientras que la estimación obtenida por la PINN es $\left(\frac{k_x}{C}\right)_{\text{predicción}} = 1{,}6329\ \text{m}^2/\text{s}$.

Igual que hicimos en el apartado anterior, hemos hecho un pequeño estudio añadiendo ruido a los datos. Los resultados se pueden ver en la Tabla 4.6. Observamos que las estimaciones son muy cercanas para todos los casos, de forma que este modo parece permisivo con el ruido de los datos, similar al modo anterior.

| **Ruido (%)** | **Error medio** | **Valor de $k_x/C$** | **Error de $k_x/C$ (%)** |
|---|---|---|---|
| 10 | $6{,}6 \cdot 10^{-3}$ | 1,5784 | 5,6 |
| 5 | $4{,}3 \cdot 10^{-3}$ | 1,6509 | 0,9 |
| 3 | $4{,}3 \cdot 10^{-3}$ | 1,6382 | 1,7 |
| 1 | $3{,}1 \cdot 10^{-3}$ | 1,6385 | 1,7 |
| 0 | $3{,}7 \cdot 10^{-3}$ | 1,6329 | 2,1 |

Tabla 4.6: Variación del error absoluto medio para las soluciones basadas en datos cuando los datos de entrenamiento tienen un ruido gaussiano.

De nuevo, en [16] hacen un estudio similar variando el número de datos dado a la red para el entrenamiento, así como el número de capas de la red y el número de neuronas de cada capa. La tendencia que se observa es la misma que para el caso del apartado anterior: los resultados mejoras cuando más parámetros y más datos tiene la red en el entrenamiento.

# Capítulo 5

# *Conclusiones*

Tras la introducción en el capítulo 1, en el capítulo 2 obtuvimos los conocimientos matemáticos básicos sobre en funcionamiento de las redes neuronales, explicando su estructura y el proceso de entrenamiento en detalle, para luego aplicarlo a un ejemplo sencillo.

En el capítulo 3 se explicaron las ecuaciones diferenciales ordinarias neuronales, una metodología novedosa en la que se emplea una red neuronal prealimentada para modelar ecuaciones o sistemas de ecuaciones diferenciales ordinarias. Se explicó su funcionamiento y cómo se realiza el proceso de entrenamiento, y se terminó el apartado aplicado lo expuesto en ese capítulo a un caso concreto. Obtenemos resultados compatibles con aquellos de la bibliografía [12] incluso en el caso en que tenemos pocos datos y estos tienen un ruido considerable (5 %), lo que ejemplifica el potencial uso de esta técnica con medidas reales entre las que destaca la capacidad de extrapolar fuera de las regiones de entrenamiento.

En el capítulo 4 se introdujo el paradigma de las redes neuronales informadas en física, con el que podemos emplear nuestro conocimiento previo sobre la ecuación en derivadas parciales que debe obedecer un cierto sistema físico para modelar su comportamiento. De nuevo, explicamos su comportamiento para el caso en tiempo continuo, y aplicamos sus dos modos de funcionamiento primarios: soluciones basadas en datos y descubrimiento basado en datos. Aplicando esta metodología a la ecuación del calor, obtenemos que con en el primer modo podemos hallar la solución a la ecuación en derivadas parciales proporcionando una condición inicial y unas condiciones de contorno sin más trabajo extra como, por ejemplo, definir un mallado espacial; mientras en el segundo caso podemos obtener también un valor aproximando de los parámetros de la ecuación diferencial si también proporcionamos datos dentro del dominio espacio-temporal del problema, de nuevo sin tener que definir un mallado espacial, lo que ejemplifica la sencilla implementación del modelo. Obtenemos datos muy similares a la bibliografía sobre el tema [16] para ambos modos, incluyendo los casos en lo que nuestros datos tienen ruido para porcentajes de ruido variables, una situación esperable en el caso de querer trabajar con datos reales. Un paso que nos ha faltado, no obstante, es la consideración de incertidumbres en los resultados, ya sea por el uso de datos reales con incertidumbres y ruido o por la estocasticidad del proceso de entrenamiento de las redes neuronales (si ejecutamos el programa dos veces, obtendremos un resultado muy similar pero no igual). Ya ha habido avances en este sentido [73, 78, 79], lo que demuestra que la comunidad está dando importancia a este modelo y acerca cada vez más su posible uso en proyectos científicos reales.

Para casos relativamente sencillos, como el que hemos resuelto aquí, quizás es más aconsejable el empleo de otro software de resolución de EDPs, como `FreeFem` [80], sobre todo si no disponemos del equipo adecuado, ya que la principal desventaja de este tipo de método puede venir en el apartado del hardware. Por como son las operaciones realizadas durante el entrenamiento de redes neuronales, estas son adecuadas para ser paralelizadas. Es por esto que el uso de tarjetas gráficas (GPU) es muy común en este ámbito, ya que disponen de un mayor número de núcleos que los procesadores (CPUs) [81]. La diferencia entre realizar el entrenamiento en una GPU a una CPU es tan notable que la no disponibilidad de una tarjeta gráfica puede resultar en tiempos de entrenamiento prohibitivos, imposibilitando la aplicabilidad del modelo. La diferencia puede ser del orden de 100 veces más lento. La disponibilidad, por otra parte, de servicios de procesamiento en la nube en los que se alquilan ordenadores con los componentes adecuados es una muy buena opción si no disponemos del hardware necesario. Este fue el método al que se recurrió para ejecutar los programas de las PINN, pasando de un tiempo estimado del orden de decenas de horas a unos pocos minutos para el caso de soluciones basadas en física y media hora para los descubrimientos basados en física. Como se ve, disponer del hardware adecuado es importante.

# Bibliografía


[1] «Repositorio con los programas realizados en este trabajo». En: *GitHub* (2024). URL: https://github.com/JaimeGabriel/TFM_JaimeGabrielVegas.git [Visitado: 26/6/2024].

[2] Stuart J Russell y Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[3] W John Hutchins. «Machine translation: A brief history». En: *Concise history of the language sciences*. Elsevier, 1995, págs. 431-445.

[4] Jim Howe. «Artificial Intelligence at Edinburgh University : a Perspective». En: *The University of Edimburgh, School of Informatics* (Revisitado 2007). URL: https://www.inf.ed.ac.uk/about/AIhistory.html [Visitado: 25/6/2024].

[5] Murray Campbell, A Joseph Hoane Jr y Feng-hsiung Hsu. «Deep blue». En: *Artificial Intelligence* 134.1-2 (2002), págs. 57-83.

[6] Yann LeCun et al. «Gradient-based learning applied to document recognition». En: *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324.

[7] Antonia Creswell et al. «Generative adversarial networks: An overview». En: *IEEE Signal Processing Magazine* 35.1 (2018), págs. 53-65.

[8] Gokul Yenduri et al. «GPT (Generative Pre-Trained Transformer)— A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions». En: *IEEE Access* 12 (2023), págs. 54608-54649.

[9] John Jumper et al. «Highly accurate protein structure prediction with AlphaFold». En: *Nature* 596.7873 (2021), págs. 583-589.

[10] Julia Gonski. «Learning by machines, for machines: Artificial Intelligence in the world's largest particle detector». En: *ATLAS experiment* (2024). URL: https://atlas.cern/Updates/Feature/Machine-Learning [Visitado: 21/6/2024].

[11] Tatiana Muraveva Lorenzo Monti. «Machine Learning and Deep Learning algorithms for Gaia mission data analysis». En: *INAF - Osservatorio di Astrofisica e Scienza dello Spazio di Bologna* (2024). URL: https://indico.ict.inaf.it/event/2752/contributions/17179/attachments/8127/16802/ICSC_Monti_Elba2024.pdf [Visitado: 21/6/2024].

[12] Ricky T. Q. Chen et al. «Neural Ordinary Differential Equations». En: *Advances in Neural Information Processing Systems*. Ed. por S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018.

[13] Christopher KI Williams y Carl Edward Rasmussen. *Gaussian processes for machine learning*. Vol. 2. 3. MIT Press Cambridge, MA, 2006.

[14] Maziar Raissi, Paris Perdikaris y George Em Karniadakis. «Numerical Gaussian processes for time-dependent and nonlinear partial differential equations». En: *SIAM Journal on Scientific Computing* 40.1 (2018), A172-A198.

[15] Maziar Raissi y George Em Karniadakis. «Hidden physics models: Machine learning of nonlinear partial differential equations». En: *Journal of Computational Physics* 357 (2018), págs. 125-141.

[16] Maziar Raissi, Paris Perdikaris y George E Karniadakis. «Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations». En: *Journal of Computational Physics* 378 (2019), págs. 686-707.

[17] «Artificial Intelligence». En: *Oxford Reference* (2024). URL: https://www.oxfordreference.com/display/10.1093/oi/authority.20110803095426960 [Visitado: 24/6/2024].

[18] Gio Wiederhold y John McCarthy. «Arthur Samuel: Pioneer in Machine Learning». En: *IBM Journal of Research and Development* 36.3 (1992), págs. 329-331.

[19] Rui Ye et al. «OpenFedLLM: Training Large Language Models on Decentralized Private Data via Federated Learning». En: *arXiv preprint arXiv:2402.06954* (2024).

[20] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

[21] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep learning*. MIT Press, 2016.

[22] M Emre Celebi y Kemal Aydin. *Unsupervised learning algorithms*. Vol. 9. Springer, 2016.

[23] Xiaojin Jerry Zhu. «Semi-supervised learning literature survey». En: *Technical Report #1530. University of Wisconsin-Madison Department of Computer Sciences* (2005).

[24] Hongming Zhang y Tianyang Yu. «AlphaZero». En: *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Ed. por Hao Dong, Zihan Ding y Shanghang Zhang. Singapore: Springer Singapore, 2020, págs. 391-415.

[25] Oriol Vinyals et al. «Grandmaster level in StarCraft II using multi-agent reinforcement learning». En: *Nature* 575.7782 (2019), págs. 350-354.

[26] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination Press San Francisco, CA, USA, 2015.

[27] Kaiming He et al. «Deep residual learning for image recognition». En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, págs. 770-778.

[28] Daniel W Otter, Julian R Medina y Jugal K Kalita. «A survey of the usages of deep learning for natural language processing». En: *IEEE Transactions on Neural Networks and Learning Systems* 32.2 (2020), págs. 604-624.

[29] Fei-Yue Wang et al. «Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond». En: *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016), págs. 113-120.

[30] Warren S McCulloch y Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». En: *The Bulletin of Mathematical Biophysics* 5 (1943), págs. 115-133.

[31] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[32] Andrea Apicella et al. «A survey on modern trainable activation functions». En: *Neural Networks* 138 (2021), págs. 14-32.

[33] Izaak Neutelings. *Neural networks*. https://tikz.net/neural_networks/. [Visitado: 14/6/2024].

[34] Ashish Vaswani et al. «Attention is all you need». En: *Advances in Neural Information Processing Systems* 30 (2017).

[35] *Curves of Steepest Descent for 3D Functions*. https://demonstrations.wolfram.com/CurvesOfSteepestDescentFor3DFunctions/. [Visitado: 10/3/2024].

[36] Sebastian Ruder. «An overview of gradient descent optimization algorithms». En: *arXiv preprint arXiv:1609.04747* (2016).

[37] Yurii Nesterov. «A method for unconstrained convex minimization problem with the rate of convergence O (1/k2)». En: *Doklady Akademii Nauk SSSR*. Vol. 269. 3. 1983, pág. 543.

[38] John Duchi, Elad Hazan y Yoram Singer. «Adaptive Subgradient Methods for Online Learning and Stochastic Optimization». En: *Journal of Machine Learning Research* 12.61 (2011), págs. 2121-2159.

[39] Dami Choi et al. «On empirical comparisons of optimizers for deep learning». En: *arXiv preprint arXiv:1910.05446* (2019).

[40] Matthew D Zeiler. «Adadelta: an adaptive learning rate method». En: *arXiv preprint arXiv:1212.5701* (2012).

[41] Geoffrey Hinton, Nitish Srivastava y Kevin Swersky. «Neural networks for machine learning lecture 6a overview of mini-batch gradient descent». En: *Coursera lecture* 14.8 (2012).

[42] Diederik P Kingma y Jimmy Ba. «Adam: A method for stochastic optimization». En: *arXiv preprint arXiv:1412.6980* (2014).

[43] Sashank J Reddi, Satyen Kale y Sanjiv Kumar. «On the convergence of adam and beyond». En: *arXiv preprint arXiv:1904.09237* (2019).

[44] David Martınez Rubio. «Convergence analysis of an adaptive method of gradient descent». En: *University of Oxford, Oxford, M. Sc. thesis* (2017).

[45] Timothy Dozat. «Incorporating Nesterov Momentum into Adam». En: *Proceedings of the 4th International Conference on Learning Representations*. 2016, págs. 1-4.

[46] Atilim Gunes Baydin et al. «Automatic differentiation in machine learning: a survey». En: *Journal of machine learning research* 18.153 (2018), págs. 1-43.

[47] James Bradbury et al. «JAX: composable transformations of Python+NumPy programs». Ver. 0.3.13. En: (2018). URL: http://github.com/google/jax [Visitado: 24/6/2024].

[48] Seiya Tokui et al. «Chainer: A Deep Learning Framework for Accelerating the Research Cycle». En: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2019, págs. 2002-2011.

[49] *PyTorch*. https://pytorch.org/. [Visitado: 24/6/2024].

[50] Andreas Griewank y Andrea Walther. «Evaluating derivatives: principles and techniques of algorithmic differentiation». En: (2008).

[51] R. A. Fisher. «Iris». En: *UCI Machine Learning Repository* (1988). URL: https://archive.ics.uci.edu/dataset/53/iris [Visitado: 28/6/2024].

[52] Kurt Hornik, Maxwell Stinchcombe y Halbert White. «Multilayer feedforward networks are universal approximators». En: *Neural Networks* 2.5 (1989), págs. 359-366.

[53] George Cybenko. «Approximation by superpositions of a sigmoidal function». En: *Mathematics of Control, Signals and Systems* 2.4 (1989), págs. 303-314.

[54] Moshe Leshno et al. «Multilayer feedforward networks with a nonpolynomial activation function can approximate any function». En: *Neural Networks* 6.6 (1993), págs. 861-867.

[55] Ian Goodfellow et al. «Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks». En: *ICLR2014*. 2014.

[56] David H Wolpert y William G Macready. «No free lunch theorems for optimization». En: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), págs. 67-82.

[57] Liliana Teodorescu. «Artificial neural networks in high-energy physics». En: *CERN Document Server* (2008).

[58] Jonathan Butterworth y Collaboration ATLAS. «Observation of four-top-quark production in the multilepton final state with the ATLAS detector». En: *European Physical Journal C: Particles and Fields* 83 (2023), pág. 496.

[59] Georges Aad et al. «Artificial neural networks on FPGAs for real-time energy reconstruction of the ATLAS LAr calorimeters». En: *Computing and Software for Big Science* 5 (2021), págs. 1-11.

[60] *GAIA*. URL: https://www.esa.int/Science_Exploration/Space_Science/Gaia [Visitado: 21/6/2024].

[61] Jaemin Seo et al. «Avoiding fusion plasma tearing instability with deep reinforcement learning». En: *Nature* 626.8000 (2024), págs. 746-751.

[62] Patrick Kidger. *On Neural Differential Equations*. 2022. arXiv: 2202.02435 [cs.LG].

[63] Ricky T. Q. Chen. «torchdiffeq». En: (2018). URL: https://github.com/rtqichen/torchdiffeq [Visitado: 25/6/2024].

[64] Salvatore Cuomo et al. «Scientific machine learning through physics–informed neural networks: Where we are and what's next». En: *Journal of Scientific Computing* 92.3 (2022), pág. 88.

[65] MWMG Dissanayake y Nhan Phan-Thien. «Neural-network-based approximations for solving partial differential equations». En: *Communications in Numerical Methods in Engineering* 10.3 (1994), págs. 195-201.

[66] *TensorFlow*. https://www.tensorflow.org/. [Visitado: 25/6/2024].

[67] George Em Karniadakis et al. «Physics-informed machine learning». En: *Nature Reviews Physics* 3.6 (2021), págs. 422-440.

[68] Shengze Cai et al. «Physics-informed neural networks (PINNs) for fluid mechanics: A review». En: *Acta Mechanica Sinica* 37.12 (2021), págs. 1727-1738.

[69] Xiaowei Jin et al. «NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations». En: *Journal of Computational Physics* 426 (2021), pág. 109951.

[70] Guofei Pang, Lu Lu y George Em Karniadakis. «fPINNs: Fractional physics-informed neural networks». En: *SIAM Journal on Scientific Computing* 41.4 (2019), A2603-A2626.

[71] Lei Yuan et al. «A-PINN: Auxiliary physics informed neural networks for forward and inverse problems of nonlinear integro-differential equations». En: *Journal of Computational Physics* 462 (2022), pág. 111260.

[72] Liu Yang, Dongkun Zhang y George Em Karniadakis. «Physics-informed generative adversarial networks for stochastic differential equations». En: *SIAM Journal on Scientific Computing* 42.1 (2020), A292-A317.

[73] Dongkun Zhang et al. «Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems». En: *Journal of Computational Physics* 397 (2019), pág. 108850.

[74] Agnes Tourin. «Introduction to finite differences methods». En: *Optimal Stochastic Control, Stochastic Target Problems, and Backward SDE* (2013), págs. 201-212.

[75] Reza Akbarian Bafghi y Maziar Raissi. «PINNs-Torch: Enhancing Speed and Usability of Physics-Informed Neural Networks with PyTorch». En: *The Symbiosis of Deep Learning and Differential Equations III*. 2023.

[76] Reza Akbarian Bafghi y Maziar Raissi. *PINNs-TF2: Fast and User-Friendly Physics-Informed Neural Networks in TensorFlow V2*. 2023. arXiv: 2311.03626.

[77] Reza Akbarian Bafghi y Maziar Raissi. «Comparing PINNs Across Frameworks: JAX, TensorFlow, and PyTorch». En: *ICLR 2024 Workshop on AI4DifferentialEquations In Science*. 2024.

[78] Björn Lütjens et al. «Pce-pinns: Physics-informed neural networks for uncertainty propagation in ocean modeling». En: *Workshop on AI for Modeling Oceans and Climate Change* (2021).

[79] Apostolos F Psaros et al. «Uncertainty quantification in scientific machine learning: Methods, metrics, and comparisons». En: *Journal of Computational Physics* 477 (2023), pág. 111902.

[80] F. Hecht. «New development in FreeFem++». En: *J. Numer. Math.* 20.3-4 (2012), págs. 251-265. ISSN: 1570-2820. URL: https://freefem.org/.

[81] Ebubekir Buber y DIRI Banu. «Performance analysis and CPU vs GPU comparison for deep learning». En: *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. IEEE. 2018, págs. 1-6.