

Vista previa
Vista previa 2016

INTRODUCCIÓN A LA SOLUCIÓN DE ECUACIONES DIFERENCIALES USANDO REDES NEURONALES

Miguel Uh Zapata
Reymundo Itzá Balam

Agradecimientos

"Ningún modelo se construye en soledad: detrás de cada ecuación hay preguntas compartidas, conversaciones y mucha paciencia."

Este libro es fruto de varios años de trabajo en el aula y en la investigación, y no habría sido posible sin el apoyo de muchas personas e instituciones.

Agradecemos a la Secretaría de Ciencia, Humanidades, Tecnología e Innovación (Secihti), que, a través del programa Investigadoras e Investigadores por México, ha hecho posible nuestra labor de investigación en Yucatán. Asimismo, agradecemos al Centro de Investigación en Matemáticas (Cimat), Unidad Mérida, por brindarnos un espacio de trabajo y de discusión académica.

A los estudiantes de nuestros cursos introductorios de redes neuronales, cuyas preguntas en clase dieron forma a muchas de las explicaciones de este texto. En particular, al Lic. en Matemáticas Eduardo Patrón Anchondo, quien nos ayudó con la transcripción de las notas originales, con la redacción de algunas secciones y con varias de las ilustraciones.

A nuestros colegas del Grupo de Matemática Computacional de Alto Rendimiento (HiperCoM), con quienes colaboramos, por propiciar un espacio constante de diálogo y discusión.

Por último, y de manera muy especial, a nuestras familias: su amor, su paciencia y su apoyo incondicional fueron el sostén de este proyecto durante las largas jornadas de escritura. Este libro también es suyo. Gracias, además, a nuestros amigos, por su ánimo y su compañía a lo largo del camino.

Los autores

Vista previa 2026

Acerca de los autores



angeluh@cimat.mx
ORCID: 0000-0001-8693-0310
SNII, nivel I

Miguel Ángel Uh Zapata

Investigador por México, Secihti
Cimat, Unidad Mérida

Obtuvo el doctorado en Matemáticas Computacionales y Aplicadas en Southern Methodist University (Dallas, Texas, EE. UU.) en 2012. Después realizó una estancia posdoctoral en el Laboratorio de Hidráulica Saint-Venant de la École des Ponts ParisTech, en Francia, y en 2014 se incorporó al CIMAT, Unidad Mérida, donde actualmente es Investigador por México de la Secihti. Su investigación se enfoca en el desarrollo de herramientas numéricas para la dinámica de fluidos computacional y la propagación de ondas, con aplicaciones a flujos multifásicos, transporte de sedimentos y flujos con superficie libre; en particular, métodos de volumen finito en mallas no estructuradas, diferencias finitas implícitas de alto orden y algoritmos paralelos en CPU y GPU. Ha impartido cursos de análisis numérico, solución numérica de EDPs y cómputo en paralelo, y participa en proyectos de divulgación como el laberinto interactivo táctil.



reymundo.itza@cimat.mx
ORCID: 0000-0002-7908-7262
SNII, nivel I

Reymundo Ariel Itzá Balam

Investigador por México, Secihti
Cimat, Unidad Mérida

Forma parte del programa Investigadoras e Investigadores por México, comisionado al Centro de Investigación en Matemáticas (CIMAT), Unidad Mérida. Es licenciado y maestro en Ciencias Matemáticas por la Universidad Autónoma de Yucatán (UADY) y doctor en Ciencias Matemáticas, con mención honorífica, por la Universidad Nacional Autónoma de México (UNAM); realizó dos estancias posdoctorales en el CIMAT. Su trabajo se centra en el análisis numérico y las matemáticas computacionales: esquemas de alto orden, el método de interfaces inmersas, ecuaciones con coeficientes discontinuos, funciones de base radial y cómputo en paralelo, con aplicaciones a la propagación de ondas y la dinámica de fluidos. Imparte cursos de cálculo, álgebra y ecuaciones diferenciales en la UADY, fue entrenador de la Olimpiada de Matemáticas en Yucatán y desarrolla actividades interactivas de divulgación. Más allá de los números, disfruta de un buen café, de conversar con amigos y de dejarse guiar por la curiosidad.

Una colaboración de casi una década. Los autores trabajan juntos desde 2017 en el diseño de métodos numéricos para ecuaciones diferenciales parciales: diferencias finitas de alto orden, interfaces inmersas y métodos de volumen finito. En 2020 dieron un paso hacia el aprendizaje automático con una red neuronal de frontera inmersa para ecuaciones elípticas con fuerzas singulares. Este libro reúne esa experiencia con el propósito de acercar las redes neuronales informadas por la física a estudiantes y profesionales.

Vista previa 2026

Prefacio

El crecimiento exponencial de los recursos computacionales ha impulsado avances significativos en las redes neuronales, con aplicaciones destacadas en múltiples disciplinas científicas. Muchas de estas áreas, como la física o la ingeniería, cuentan con un conocimiento previo expresado en leyes y ecuaciones fundamentales, el cual puede integrarse al proceso de aprendizaje.

En este contexto surgen las redes neuronales informadas por la física (PINNs, por sus siglas en inglés, *Physics-Informed Neural Networks*), un enfoque que incorpora explícitamente principios físicos dentro del entrenamiento de una red neuronal: la red no solo aprende de los datos, sino que se le exige respetar la ecuación diferencial, las condiciones iniciales y las condiciones de frontera del problema. Estas técnicas permiten mejorar la precisión en contextos con pocos datos e incrementan la interpretabilidad y la capacidad de generalización de los modelos. Así, las redes neuronales aplicadas a la solución de ecuaciones diferenciales se consolidan como herramientas versátiles y prometedoras para modelar sistemas complejos.

A pesar de su potencial, las PINNs aún presentan limitaciones importantes frente a los métodos numéricos tradicionales, especialmente en términos de costo computacional, estabilidad, convergencia y precisión en determinados problemas. Por ello, la investigación en este campo continúa siendo muy activa, con el objetivo de desarrollar metodologías más robustas y eficientes. No obstante, el creciente interés por estas técnicas ha propiciado que con frecuencia se utilicen como una "caja negra", sin una comprensión suficiente de sus fundamentos matemáticos, de sus supuestos ni de las condiciones bajo las cuales pueden fallar.

Esta situación motivó la escritura de este libro. Su propósito es conducir al lector desde los conceptos fundamentales de las redes neuronales hasta su aplicación en la aproximación de soluciones de ecuaciones diferenciales ordinarias y en derivadas parciales mediante PINNs. El texto combina la teoría matemática con ejemplos prácticos y desarrollos computacionales, con el fin de proporcionar una comprensión integral de la metodología, de sus alcances y también de sus límites. Se trata de un recurso de libre acceso, escrito íntegramente en español, con el que buscamos acercar estas herramientas a quienes se inician en el tema.

¿A quién va dirigido?

El libro está pensado para estudiantes de disciplinas STEM (ciencia, tecnología, ingeniería y matemáticas), tanto de licenciatura como de posgrado, interesados en el uso de redes neuronales para resolver problemas gobernados por ecuaciones diferenciales. También puede ser útil para profesores e investigadores que deseen una introducción ordenada al tema. Basta con conocer cálculo diferencial e integral, álgebra lineal y un curso básico de ecuaciones diferenciales; algo de experiencia en programación en Python será de gran ayuda, aunque no es indispensable.

Vista previa 2026

Contenido del libro

- **Capítulo 1.** Modelado matemático y computacional de fenómenos físicos. Recorre el camino que va de la observación de un fenómeno a su formulación matemática y a su simulación en la computadora, y sitúa a las redes neuronales dentro de este contexto.
- **Capítulo 2.** Introducción a las redes neuronales artificiales. Presenta los conceptos fundamentales a partir de su forma más simple, el perceptrón, y describe los elementos clave para construir y resolver problemas con esta técnica: la arquitectura de la red, el proceso de entrenamiento y los mecanismos de ajuste de parámetros, ilustrados con ejemplos.
- **Capítulo 3.** Ajustes de curvas usando redes neuronales. Compara los métodos clásicos de interpolación (polinomios de Lagrange y splines cúbicos) con una red neuronal: construcción del modelo, función de pérdida, entrenamiento e implementación en Python y PyTorch, así como el efecto del número de datos y de neuronas en la calidad del ajuste.
- **Capítulo 4.** Redes neuronales informadas por la física para ecuaciones en 1D. Introduce la metodología PINN con ecuaciones elípticas unidimensionales, la compara con el método de diferencias finitas y analiza la convergencia, el papel de los puntos de colocación y distintas arquitecturas; cierra con el caso de un oscilador armónico. El código completo se incluye en el Apéndice A.
- **Capítulo 5.** Redes neuronales informadas por la física para ecuaciones en 2D. Extiende el método a ecuaciones elípticas bidimensionales, desarrolla paso a paso el programa en Python y lo aplica a la ecuación de Laplace en dominios rectangulares —con mallas uniformes, puntos aleatorios y condiciones de frontera mixtas— y en dominios irregulares.

Cómo usar este libro

Cada capítulo abre con sus objetivos y un mapa de contenidos. Las notas al margen señalan definiciones, notación y datos históricos, y los recuadros de colores destacan las ideas centrales. Los fragmentos de código se presentan de forma breve y legible para que el lector pueda modificarlos y experimentar: creemos que la mejor forma de entender un método es verlo funcionar y, a veces, verlo fallar. Los primeros capítulos proporcionan las bases de modelado y de redes neuronales; los capítulos 4 y 5 se apoyan en ellas, por lo que recomendamos leerlos en orden.
Esperamos que el lector encuentre en estas páginas no solo un conjunto de técnicas, sino también una invitación a mirar con curiosidad los fenómenos que lo rodean, a preguntarse cómo podrían escribirse en el lenguaje de las matemáticas y a usar las PINNs con criterio, sabiendo qué hay dentro de la "caja".

Mérida, Yucatán, octubre de 2026

Miguel Uh Zapata

Reymundo Itzá Balam

Vista previa 2026

Índice general

CAPÍTULO 1:

Modelo matemático y computacional de fenómenos físicos 1

1.1 Modelando fenómenos fisicos . . . 3
1.2 Formalizando el modelado matemático . . . 4
1.3 Modelado matemático . . . 5
1.3.1 Modelando el cambio de temperatura . . . 6
1.3.2 Limitaciones de las soluciones analíticas . . . 10
1.4 Modelado computacional . . . 11
1.5 De las ecuaciones a la simulación . . . 12
1.5.1 Del continuo al discreto . . . 12
1.5.2 Métodos y simulación científica . . . 13

CAPÍTULO 2:

Introducción a las redes neuronales artificiales 17

2.1 Introducción . . . 19
2.1.1 El aprendizaje automático . . . 20
2.1.2 Las redes neuronales artificiales . . . 20
2.2 El Perceptrón . . . 22
2.2.1 Los perceptrones con sesgos . . . 23
2.2.2 Ejemplo: el problema de clasificación XOR . . . 24
2.3 Redes neuronales multicapa . . . 26
2.3.1 La red neuronal en términos de sus pesos y sesgos . . . 27
2.3.2 Solución del problema XOR usando una capa oculta . . . 29
2.3.3 Redes neuronales multicapa más complejas . . . 30
2.4 Funciones de activación . . . 32
2.4.1 Funciones de activación discontinuas . . . 32
2.4.2 Funciones de activación continuas . . . 32
2.4.3 Función de activación diferenciable: sigmoide . . . 33
2.4.4 Función de activación diferenciable: tanh . . . 34
2.5 Hacia el cálculo de los pesos y sesgos . . . 35
2.5.1 Modelo de regresión lineal . . . 35
2.5.2 El método de mínimos cuadrados . . . 36
2.6 Entrenamiento . . . 37
2.6.1 Los datos conocidos . . . 38
2.6.2 La función de pérdida . . . 38
2.6.3 El método de optimización . . . 40
2.7 Iteraciones con retropropagación . . . 44
2.7.1 Un ejemplo lineal con una sola neurona y un dato . . . 44
2.7.2 El ejemplo lineal con valores específicos . . . 46
2.7.3 El programa en Python para calcular más épocas . . . 48
2.7.4 Un ejemplo lineal con cinco datos conocidos . . . 52
2.7.5 El programa en Python para múltiples datos . . . 55

Vista previa 2026

2.8 Entrenamiento para redes complejas . . . 58
2.8.1 Cálculo manual de una primera época . . . 59
2.8.2 El programa en Python para calcular más épocas . . . 63
2.9 Librerías para redes neuronales . . . 67
2.9.1 Ejecución en la nube: Google Colab . . . 68

CAPÍTULO 3:
Ajustes de curvas usando redes neuronales 79
3.1 Introducción . . . 81
3.1.1 Herramientas matemáticas . . . 82
3.2 El ejemplo de interpolación . . . 84
3.3 Métodos clásicos de interpolación . . . 85
3.3.1 Interpolación usando polinomios de Lagrange . . . 85
3.3.2 Interpolación usando splines cúbicos . . . 87
3.4 Interpolación con redes neuronales . . . 89
3.4.1 Construcción del modelo . . . 90
3.4.2 Recopilación de datos . . . 91
3.4.3 Función de pérdida . . . 92
3.4.4 Entrenamiento de la red neuronal . . . 93
3.4.5 Evaluación del modelo . . . 93
3.5 Implementación en Python y PyTorch . . . 94
3.5.1 Datos conocidos para el entrenamiento de la red . . . 95
3.5.2 Construcción de la red neuronal . . . 96
3.5.3 Definición de la función de pérdida . . . 96
3.5.4 Entrenamiento de la red . . . 96
3.5.5 Evaluación con los datos de prueba . . . 96
3.6 Simulaciones usando redes neuronales . . . 97
3.6.1 Simulaciones 20 datos conocidos y 5 neuronas . . . 97
3.6.2 Convergencia del error cuadrático medio . . . 98
3.6.3 Solución explícita con pesos y sesgos entrenados . . . 100
3.6.4 Mayor número de datos conocidos . . . 101
3.6.5 Mayor número de neuronas . . . 102
3.6.6 Porcentaje de éxito del algoritmo . . . 104
3.7 Ajuste de una curva a datos aleatorios . . . 106
3.7.1 Resultados con datos aleatorios . . . 106

CAPÍTULO 4:
Redes neuronales informadas por la física para ecuaciones en 1D 115
4.1 Introducción . . . 117
4.2 Ecuaciones elípticas 1D . . . 118
4.3 Método de Diferencias Finitas . . . 119
4.4 Método de Redes Neuronales Informadas por la Física . . . 124
4.4.1 Definir el modelo . . . 125
4.4.2 Construir el modelo de red neuronal . . . 125
4.4.3 Recopilar datos . . . 126
4.4.4 Definir la función de pérdida . . . 128
4.4.5 Entrenar la red . . . 129
4.4.6 Evaluar el método . . . 130

Vista previa 2026

4.5 Implementación en Python . . . 131
4.5.1 Recopilación de datos . . . 133
4.5.2 Construcción del modelo . . . 133
4.5.3 Función de pérdida . . . 134
4.5.4 Optimizador y entrenamiento . . . 135
4.5.5 Evaluación de los datos de prueba . . . 135
4.6 Resultados numéricos usando las PINNs . . . 136
4.6.1 Exploración inicial del método . . . 136
4.6.2 Comparación con diferencias finitas . . . 137
4.6.3 Análisis de la convergencia del error . . . 140
4.6.4 Análisis con otras arquitecturas . . . 142
4.6.5 Caso de un oscilador armónico . . . 143

CAPÍTULO 5:
Redes neuronales informadas por la física para ecuaciones en 2D 151
5.1 Introducción . . . 153
5.2 Ecuaciones elípticas en 2D . . . 155
5.2.1 Condiciones de frontera . . . 155
5.3 El método de diferencias finitas . . . 157
5.4 Aproximaciones con redes neuronales . . . 160
5.4.1 El modelo basado en redes neuronales . . . 161
5.4.2 Incorporación de la EDP al modelo . . . 163
5.4.3 Datos conocidos: los puntos de entrenamiento . . . 164
5.4.4 Datos conocidos: los puntos de colocación . . . 166
5.4.5 La función de pérdida . . . 169
5.4.6 El entrenamiento de la red neuronal . . . 171
5.4.7 Evaluación del modelo . . . 172
5.5 El programa en Python . . . 173
5.5.1 Código para obtener los datos conocidos . . . 174
5.5.2 Código para definir la red neuronal . . . 176
5.5.3 Código para obtener el residuo . . . 176
5.5.4 Código para definir la función de pérdida . . . 178
5.5.5 Código para el entrenamiento . . . 178
5.5.6 Código para la validación del modelo . . . 180
5.5.7 Gráfica de los resultados . . . 181
5.6 Resultados en dominios rectangulares . . . 181
5.6.1 Ejemplo 1: ecuación de Laplace con mallas uniformes . . . 182
5.6.2 Ejemplo 2: ecuación de Laplace con puntos aleatorios . . . 187
5.6.3 Ejemplo 3: ecuación de Laplace con condiciones de frontera mixtas . . . 198
5.7 Resultados en dominios irregulares . . . 202
5.7.1 Ejemplo: ecuación de Laplace en dominios irregulares . . . 202

vista previa 2026

CAPÍTULO 1

MODELO MATEMÁTICO Y COMPUTACIONAL DE FENÓMENOS FÍSICOS

El modelado de fenómenos físicos surge de una capacidad fundamental del ser humano: reconocer patrones en la naturaleza y utilizarlos para anticipar su comportamiento. Desde las primeras observaciones empíricas de civilizaciones antiguas alrededor del mundo, como las plasmadas en los calendarios astronómicos de las culturas mesoamericanas, hasta la formalización matemática desarrollada durante la revolución científica, el objetivo ha sido el mismo: describir, entender y predecir el cambio.

La aparición del cálculo diferencial e integral permitió expresar leyes físicas en términos de tasas de variación, dando origen al lenguaje de las ecuaciones diferenciales. Estas ecuaciones constituyen hoy una herramienta central para modelar sistemas dinámicos en diversas áreas de la ciencia y la ingeniería. Sin embargo, muchos de estos modelos no admiten soluciones analíticas simples, lo que ha impulsado el desarrollo de métodos numéricos y, posteriormente, del modelado computacional. A través de la discretización y el uso de algoritmos, es posible aproximar soluciones y simular fenómenos complejos. Más recientemente, enfoques como las *redes neuronales informadas por la física* han ampliado las posibilidades del modelado, integrando datos y principios físicos en un mismo marco.

Este capítulo presenta una visión directa del modelado matemático y computacional, desde sus orígenes conceptuales hasta sus herramientas modernas, destacando su papel como puente entre la observación, la teoría y la simulación.

OBJETIVOS

Al finalizar este capítulo, el lector será capaz de:

- Comprender el origen del modelado matemático a partir de la observación de patrones en la naturaleza.
- Introducir las ecuaciones diferenciales como herramienta central para describir y analizar fenómenos físicos.
- Analizar las limitaciones de las soluciones analíticas y la necesidad de recurrir a métodos numéricos.
- Explorar el papel del modelado computacional como extensión del modelado matemático, incluyendo técnicas de discretización, simulación y enfoques modernos como redes neuronales informadas por la física.

Vista previa 2026

CONTENIDO

1.1 Modelando fenómenos fisicos

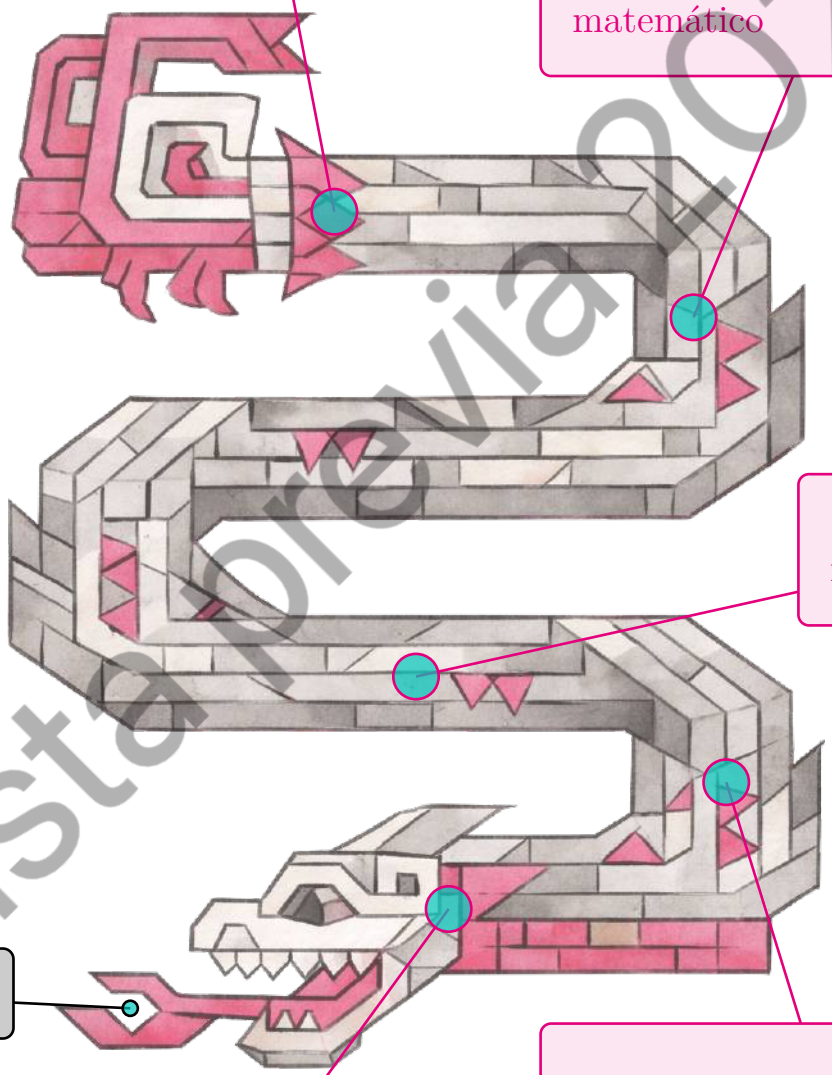
1.2 Formalizando el modelado matemático

1.3 Modelado matemático

1.4 Modelado computacional

1.5 De las ecuaciones a la simulación

Resumen



Vista previa 2026



Figura 1.1 Mesoamerica y la región maya.



Figura 1.2 La observación sistemática fue el inicio del modelado.

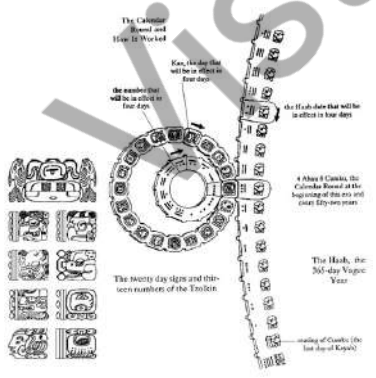


Figura 1.3 Representación del calendario maya.

1.1 MODELANDO FENÓMENOS FISICOS

La historia de la humanidad puede leerse como una larga secuencia de adaptaciones. Desde los primeros grupos nómadas hasta las sociedades sedentarias que surgieron tras la revolución agrícola (por lo menos hace 10,000 años), el ser humano ha debido ajustarse a entornos cambiantes: climas extremos, migraciones de fauna, disponibilidad de agua, ciclos estacionales. Sobrevivir no fue un acto de azar, sino de interpretación.

Entre las capacidades que resultaron decisivas destaca una en particular: la observación sistemática del entorno. Observar no solo para ver, sino para identificar regularidades. Cerca del agua abundaba la vegetación; donde escaseaba, predominaban zonas áridas. El sol seguía trayectorias predecibles en el horizonte; ciertos animales eran diurnos, otros nocturnos. Las nubes densas y oscuras precedían con frecuencia a la lluvia. La luna mostraba fases periódicas. Las estaciones regresaban.

Ese reconocimiento de patrones fue el primer paso hacia algo más profundo: la posibilidad de anticipar. Si ciertos fenómenos se repiten bajo condiciones similares, entonces quizá pueden predecirse. Esta idea, que hoy parece obvia, fue revolucionaria. Civilizaciones antiguas como la mesopotámica y la egipcia desarrollaron calendarios astronómicos precisamente para prever crecidas de ríos o ciclos agrícolas. No era aún ciencia en el sentido moderno, pero sí era modelación empírica: una representación simplificada de la realidad basada en regularidades observadas.

En Mesoamérica, la Cultura maya llevó esta observación astronómica a un nivel extraordinario. Sus astrónomos registraron con notable precisión los ciclos solares, lunares y planetarios, en particular el de Venus. El calendario Haab' (de 365 días) y el Tzolk'in (de 260 días), combinados en la llamada Rueda Calendárica, permitían sincronizar fenómenos astronómicos con actividades agrícolas y rituales. Más aún, el calendario de Cuenta Larga ofrecía un sistema posicional vigesimal para medir el tiempo en escalas extensas. No se trataba solo de contar días: era una estructura matemática que capturaba periodicidades observables del cielo. En términos modernos, habían construido un modelo temporal altamente consistente con los ciclos astronómicos visibles a simple vista.

Este paso –de reconocer patrones a expresarlos de forma sistemática– marca el inicio del pensamiento matemático. Para describir, comparar y predecir con mayor precisión, fue necesario desarrollar un lenguaje formal: la matemática. A partir de entonces, modelar dejó de ser una práctica intuitiva y se convirtió en un proceso estructurado. Este cambio da lugar al modelado matemático en sentido moderno, que abordamos a continuación.

Vista previa 2026

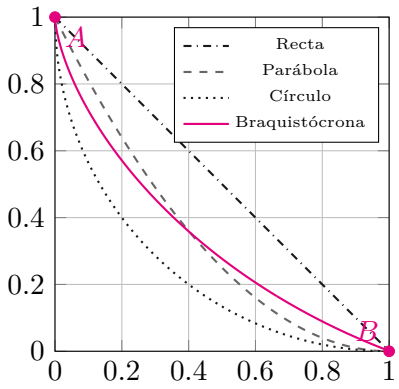


Figura 1.4 La braquistócrona es la curva en la que se desciende más rápido de A a B.

1.2 FORMALIZANDO EL MODELADO MATEMÁTICO

"Tanquam ex ungue leonem" (por la garra se reconoce al león), murmuró el matemático suizo Johann Bernoulli en 1697. La frase aludía a la solución de un problema que durante meses había desafiado a las mentes más brillantes de Europa: el problema de la *braquistócrona*. Bernoulli lo había propuesto públicamente en 1696, invitando a los matemáticos del continente a encontrar la curva por la cual una partícula desciende en el menor tiempo bajo la acción de la gravedad.

Entre las soluciones recibidas, una llegó de manera anónima desde Inglaterra. Al verla, Bernoulli reconoció de inmediato el estilo inconfundible de Isaac Newton. Según la tradición histórica (documentada en correspondencia de la época) Newton resolvió el problema en una sola noche después de regresar de la Casa de la Moneda, donde trabajaba entonces. Más allá del anecdotario, el episodio revela algo más profundo: una nueva herramienta matemática estaba ya consolidada.

De manera independiente, Gottfried Wilhelm Leibniz había desarrollado técnicas equivalentes, publicando en 1684 su método diferencial. Aunque la controversia sobre la prioridad es ampliamente conocida, lo históricamente verificable es que hacia finales del siglo XVII el Cálculo Diferencial e Integral era ya una herramienta operativa. La noción de límite (que recibiría una formalización rigurosa hasta el siglo XIX con Cauchy y Weierstrass) permitió dar sentido preciso a la idea de cambio instantáneo [56].

Vista previa 2026



Figura 1.5 De izquierda a derecha: Johann Bernoulli, Isaac Newton y Gottfried Wilhelm Leibniz. Tres figuras fundamentales en el desarrollo de la matemática moderna, en particular del Cálculo diferencial y variacional.

Notas

$$y'(x) = \frac{dy}{dx}(x),$$
$$y''(x) = \frac{d^2y}{dx^2}(x),$$
$$y^{(n)}(x) = \frac{d^ny}{dx^n}(x).$$

Figura 1.6 Notación de derivadas.

Sin embargo, el camino hacia ese punto comenzó antes. En la Grecia clásica, Eudoxo de Cnido introdujo el método de exhaución, antecedente conceptual del límite. Siglos más tarde, durante la revolución científica, Galileo Galilei dio un giro decisivo: sostuvo que el libro de la naturaleza "está escrito en lenguaje matemático". Galileo no solo observó el movimiento; lo midió. Estudió la caída de los cuerpos, la aceleración uniforme y la trayectoria parabólica de los proyectiles, estableciendo relaciones cuantitativas sistemáticas.

Con Isaac Newton, esa relación alcanzó una síntesis monumental. En los *Philosophiæ Naturalis Principia Mathematica* (1687) formuló las leyes del movimiento y la ley de gravitación universal mediante el aparato conceptual que hoy identificamos como Cálculo. El cambio dejó de ser una intuición geométrica para convertirse en una magnitud formalizable: la derivada. Aunque existieron intentos previos de modelar fenómenos físicos (desde los esquemas astronómicos de Ptolomeo hasta las leyes empíricas de Kepler) fue la invención del Cálculo lo que abrió definitivamente la puerta al modelado dinámico. Ya no se trataba solo de describir posiciones o configuraciones, sino de capturar cómo cambian en el tiempo. Con los años, la necesidad de cuantificar reemplazó a la simple descripción cualitativa. Las formas geométricas de la antigüedad dieron paso a ecuaciones que expresaban tasas de variación. La naturaleza comenzó a escribirse en términos de funciones y derivadas.

Ahí nace el corazón de las ecuaciones diferenciales: expresar leyes físicas como relaciones entre una cantidad y su cambio. Desde la caída de un cuerpo hasta la propagación del calor, desde el movimiento planetario hasta la dinámica de fluidos, el lenguaje matemático del cambio se convirtió en la herramienta fundamental para modelar el mundo. Y una vez que el cambio pudo escribirse, también pudo *analizarse*, *estimarse* y eventualmente *simularse*. El salto estaba dado: de reconocer al león por la garra, a describir su movimiento con una ecuación.

1.3 MODELADO MATEMÁTICO

Notas

$$u_t(t,x) = \frac{\partial u}{\partial t}(t,x),$$
$$u_x(t,x) = \frac{\partial u}{\partial x}(t,x),$$
$$u_{tt}(t,x) = \frac{\partial^2 u}{\partial t^2}(t,x),$$
$$u_{xx}(t,x) = \frac{\partial^2 u}{\partial x^2}(t,x),$$
$$u_{tx}(t,x) = \frac{\partial^2 u}{\partial x \partial t}(t,x).$$

Figura 1.7 Notación de derivadas parciales.

Existen muchas formas de modelar lo que observamos en la naturaleza. Podemos usar descripciones cualitativas, diagramas, reglas empíricas, etc. Sin embargo, cuando el fenómeno involucra cambio, es decir, cuando algo varía en el tiempo o en el espacio, una de las herramientas más poderosas y precisas que tenemos son las ecuaciones diferenciales.

DEFINICIÓN

Una *ecuación diferencial* es, en términos sencillos, una ecuación que relaciona una función desconocida con sus derivadas. En ella aparecen: una o más *variables independientes* (por ejemplo, el tiempo t o la posición x); una *variable dependiente* (como la temperatura, la velocidad o la concentración); y las *derivadas* de esa variable dependiente, que representan tasas de cambio.

Vista previa 2026

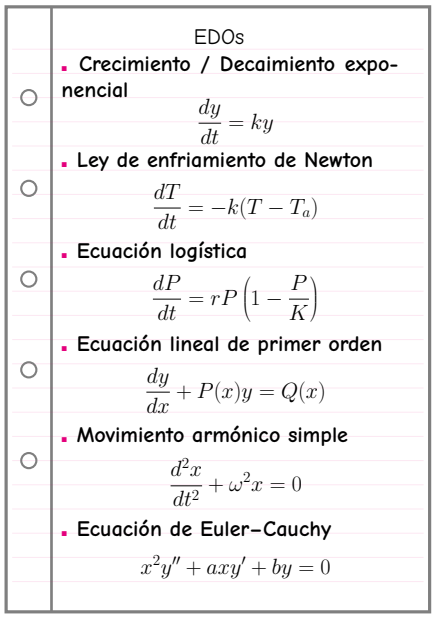


Figura 1.8 Algunas EDOs importantes.

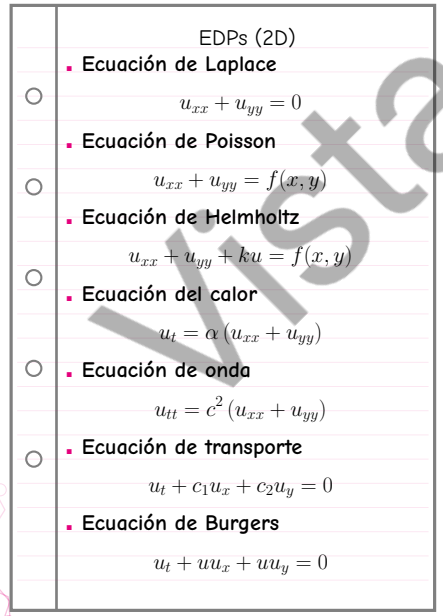


Figura 1.9 Algunas EDPs importantes.

¿Por qué son tan útiles? Porque muchas leyes físicas fundamentales se expresan precisamente como relaciones entre una cantidad y la forma en que cambia. Por ejemplo, la velocidad es la derivada de la posición respecto al tiempo, y la aceleración es la derivada de la velocidad. Cuando escribimos estas relaciones de manera sistemática, obtenemos ecuaciones diferenciales.

Las ecuaciones diferenciales pueden clasificarse de varias maneras:

- **Tipo:** depende del número de varables independientes involucradas en la ecuación.
  - Ecuaciones diferenciales ordinarias (EDO): involucran una sola variable independiente. Por ejemplo, cuando una cantidad depende únicamente del tiempo.
  - Ecuaciones diferenciales parciales (EDP): involucran dos o más variables independientes. Por ejemplo, cuando depende del tiempo y el espacio simultáneamente, en fenómenos como la propagación del calor o el movimiento de fluidos.
- **Orden:** se determina por la derivada de mayor grado que aparece en la ecuación. Si la derivada más alta es la primera, se dice que es de primer orden; si aparece la segunda derivada, es de segundo orden, y así sucesivamente.
- **Linealidad:** una ecuación es lineal si la función desconocida y sus derivadas aparecen sin multiplicarse entre sí ni elevarse a potencias distintas de uno. Si no cumple estas condiciones, se dice que es no lineal.

Estas clasificaciones no son meramente formales; anticipan el tipo de comportamiento que puede presentar el modelo y las técnicas matemáticas necesarias para analizarlo [57].

En esencia, modelar con ecuaciones diferenciales significa traducir un fenómeno físico a un lenguaje matemático que describe cómo cambia. A partir de ahí, podemos analizar su comportamiento, predecir su evolución y, en muchos casos, simularlo computacionalmente. La idea central es simple: si entendemos cómo cambia algo, podemos empezar a comprender cómo funciona.

1.3.1 Modelando el cambio de temperatura

Todo modelo matemático nace de una pregunta concreta. No comenzamos con una ecuación; comenzamos con una situación que nos intriga. Por ejemplo: imaginemos que vemos una taza de café sobre una mesa. Sabemos que fue servida por una máquina en el cuarto contiguo y conocemos la temperatura a la que sale el café. También sabemos la temperatura del cuarto donde ahora se encuentra. La pregunta surge de manera natural: ¿podemos estimar cuánto tiempo lleva ahí simplemente midiendo su temperatura actual?

Vista previa 2026



Figura 1.10 Taza de café y su modelado matemático. El parámetro k en la ley de enfriamiento de Newton encapsula propiedades tanto del material de la taza como de las características del fluido, determinando la rapidez con la que la temperatura evoluciona hacia el equilibrio térmico.

TEMPERATURA DE MI TAZA DE CAFE

Si suponemos que la taza es pequeña y que su temperatura es aproximadamente uniforme en todo su interior, una hipótesis razonable cuando el objeto no es muy grande y conduce bien el calor, entonces el fenómeno puede describirse mediante la ley de enfriamiento de Newton:

$$\frac{dT}{dt} = -k\,(T - T_a),$$

donde $T(t)$ es la temperatura del café en el tiempo t, T_a es la temperatura ambiente (que asumimos constante) y $k > 0$ es una constante que depende del material y de las condiciones de intercambio de calor con el entorno.

Aquí aparece un aspecto fundamental: la ecuación por sí sola no basta. Necesitamos saber la temperatura inicial. Es decir,

$$T(0) = T_0.$$

Con esa condición inicial, el problema queda bien planteado tanto en el sentido matemático como en el físico. Matemáticamente, garantiza que la ecuación tenga una solución única y determinada. Físicamente, incorpora la información concreta del caso que nos interesa: no hablamos de cualquier taza de café, sino de esa taza específica, con sus datos particulares y en ese entorno definido.

Observemos que en este modelo solo interviene el tiempo. Se trata de una ecuación diferencial ordinaria de primer orden. El espacio no juega ningún papel porque hemos supuesto que todo el objeto tiene la misma temperatura en cada instante.

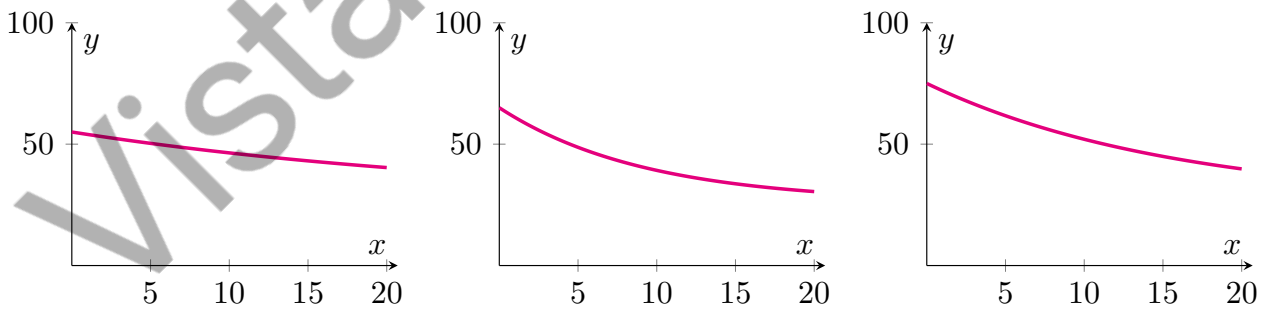


Figura 1.11 Modelado de la temperatura de una taza de café durante los primeros 15 minutos. A la izquierda se muestra el caso con $T_0 = 55$, al centro con $T_0 = 65$ y a la derecha con $T_0 = 75$. En los tres escenarios, tanto la temperatura ambiente como las propiedades de la taza se mantienen constantes.

Ahora pensemos en una situación distinta: un fenómeno un poco más complejo, la distribución de temperatura en una barra metálica. En este caso, se incluye no solo la variable temporal, sino también la variable espacial, lo que nos lleva a modelar el problema mediante una EDP.

Vista previa 2026



Figura 1.12 Retrato de Joseph Fourier.

CALENTAMIENTO DE UNA BARRA METALICA

Supongamos que tenemos una barra metálica larga, calentada inicialmente de manera no uniforme. Ya no podemos asumir que toda la barra tiene la misma temperatura, y por tanto la temperatura depende de la posición y del tiempo. Surge entonces la necesidad de un modelo más elaborado: la ecuación del calor unidimensional,

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2},$$

donde $u(x,t)$ representa la temperatura en la posición x y en el tiempo t, y $\alpha > 0$ es la difusividad térmica del material.

Esta ecuación fue estudiada sistemáticamente a comienzos del siglo XIX por *Joseph Fourier*, quien la introdujo en su investigación sobre la conducción del calor en sólidos. Su trabajo no solo permitió comprender cómo se difunde la temperatura, sino que también dio origen al análisis mediante series trigonométricas, hoy conocidas como series de Fourier.

La ecuación de calor traduce una idea física sencilla: el calor fluye desde las regiones más calientes hacia las más frías, y la rapidez con la que cambia la temperatura en un punto depende de cómo varía espacialmente a su alrededor. Si la temperatura es uniforme, no ocurre nada; si hay diferencias marcadas, el sistema evoluciona.

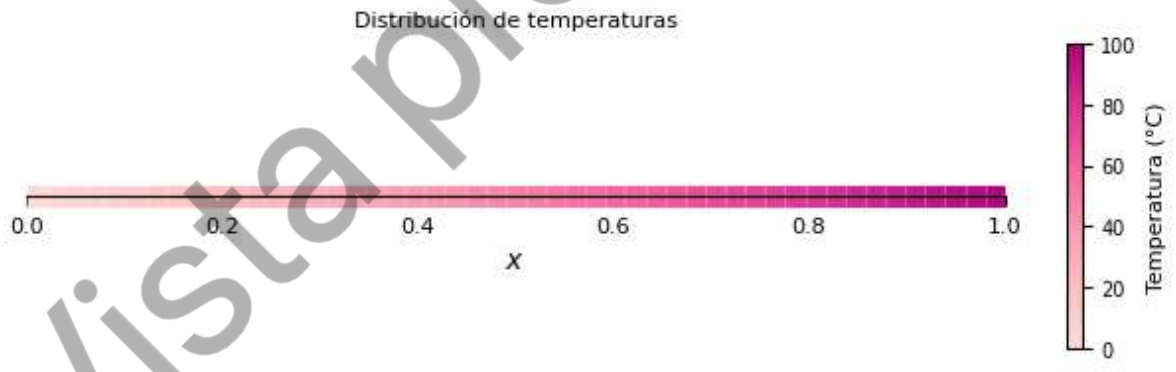


Figura 1.13 Distribución de temperatura en una barra de metal con condiciones de frontera de Dirichlet, donde un extremo se mantiene a 100 °C y el otro a 0 °C.

De nuevo aparece la necesidad de datos adicionales. Necesitamos conocer la distribución inicial de temperatura,

$$u(x,0) = f(x),$$

como en el ejemplo anterior, este planteamiento aún no es suficiente para modelar el fenómeno. Es necesario introducir condiciones adicionales que describan la interacción de la barra con su entorno.

Vista previa 2026

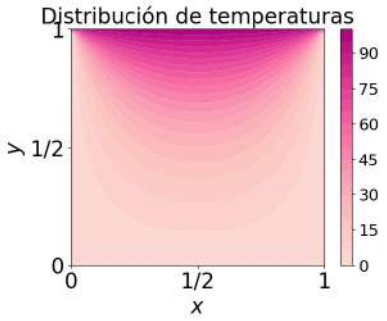


Figura 1.14 Distribución de temperatura en una placa rectangular con condiciones de frontera de Dirichlet, donde uno de sus lados se mantiene a 100 °C y los tres restantes a 0 °C.

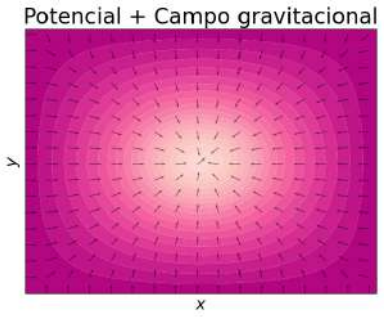


Figura 1.15 Potenicial y campo gravitacional en presencia de una masa.

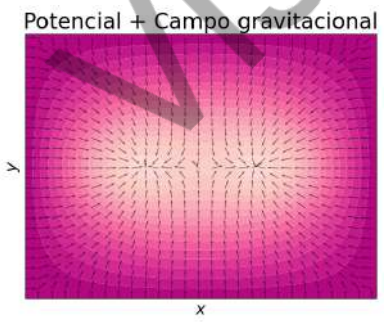


Figura 1.16 Potenicial y campo gravitacional en presencia de dos masas.

Por ejemplo, si los extremos se mantienen a temperaturas fijas (condiciones de frontera),

$$u(0,t) = T_1, \qquad u(L,t) = T_2.$$

Sin estas condiciones iniciales y de frontera, la ecuación no determina una solución única. La física nos da la forma de la ecuación; el entorno concreta el problema.

Ahora bien, si dejamos pasar suficiente tiempo, la barra puede alcanzar un equilibrio térmico. En ese momento la temperatura deja de depender del tiempo, es decir,

$$\frac{\partial u}{\partial t} = 0.$$

Al sustituir esta condición en la ecuación del calor obtenemos

$$\frac{\partial^2 u}{\partial x^2} = 0,$$

que es la versión unidimensional de la ecuación de calor estacionaria. En este estado, estacionario, la distribución de temperatura resulta ser una función lineal en x, determinada únicamente por las condiciones de frontera. El modelo dinámico describe cómo se alcanza el equilibrio; el modelo estacionario describe el equilibrio mismo.

Finalmente, si en lugar de una barra consideramos una placa metálica delgada, la temperatura depende de dos variables espaciales. En estado estacionario, el modelo natural es

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0,$$

que es la ecuación de Laplace en dos dimensiones. Si existen fuentes internas de calor (por ejemplo, un dispositivo que genera energía dentro de la placa) el modelo se convierte en la ecuación de Poisson:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x,y),$$

donde $g(x,y)$ representa la generación interna de calor. Ecuaciones de este tipo aparecieron originalmente en el estudio del potencial gravitacional y eléctrico en los trabajos de *Laplace* y *Poisson* a finales del siglo XVIII.

En ambos casos se requieren condiciones de frontera, como

$$u = u_D \quad \text{en la frontera,}$$

que pueden representar bordes mantenidos a temperatura fija, contacto con el exterior o interacción con sistemas de enfriamiento.

Vista previa 2026

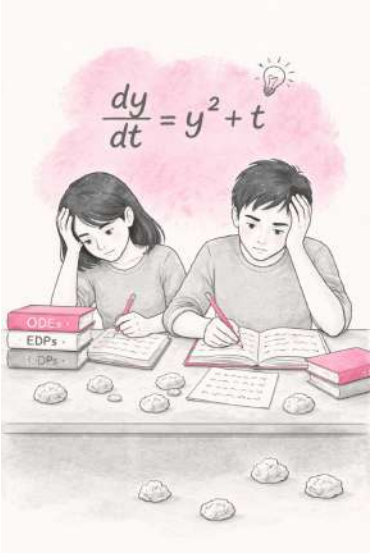


Figura 1.17 El trabajo duro a veces no basta.

En todos estos ejemplos aparece el mismo principio: la ecuación diferencial modela la ley física del fenómeno, pero son las condiciones iniciales y de frontera las que describen la situación concreta. Solo cuando ambas piezas están presentes el modelo deja de ser una expresión abstracta y se convierte en una herramienta para entender, y eventualmente estimar, lo que ocurre en el mundo real.

1.3.2 Limitaciones de las soluciones analíticas

Traducir un fenómeno físico a una ecuación diferencial es solo el primer paso. El siguiente desafío consiste en resolver esa ecuación y comprender qué describe el modelo. En algunos casos es posible encontrar soluciones exactas mediante técnicas analíticas, pero esto ocurre solo para un número limitado de ecuaciones relativamente simples.

Desde los trabajos de Leonhard Euler en el siglo XVIII se comprendió que muchas ecuaciones diferenciales podían estudiarse mediante aproximaciones sucesivas. Más tarde, matemáticos como Carl Runge y Martin Kutta desarrollaron métodos sistemáticos para aproximar soluciones con gran precisión.

Sin embargo, la búsqueda de soluciones analíticas tiene límites claros. Muchas ecuaciones diferenciales que surgen al modelar fenómenos reales no admiten expresiones cerradas en términos de funciones elementales. Un ejemplo sencillo aparece incluso en ecuaciones diferenciales ordinarias aparentemente simples. Considérese, por ejemplo,

$$\frac{dy}{dt} = y^2 + t, \qquad y(0) = 1.$$

Aunque la ecuación es de primer orden y su forma parece inocente, no existe una expresión conocida para su solución en términos de funciones elementales. Situaciones como esta son comunes cuando los modelos incorporan efectos no lineales o dependencias más complejas. En estos casos, el objetivo deja de ser encontrar una fórmula exacta y pasa a ser aproximar la solución con suficiente precisión para entender el comportamiento del sistema. Esta dificultad no es exclusiva de ejemplos unidimensionales: en dimensiones mayores surgen problemas aún más profundos, como las ecuaciones de Navier–Stokes para fluidos incompresibles, que relaciona la velocidad y la presión con el sistema de ecuaciones:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \Delta \mathbf{u}, \qquad \nabla \cdot \mathbf{u} = 0,$$

cuyo problema de existencia y regularidad en tres dimensiones permanece abierto y forma parte de los problemas del milenio. Para un panorama moderno de enfoques analíticos y computacionales, puede consultarse Lemarié-Rieusset, *The Navier–Stokes problem in the 21st century* [58].

¿Sabías que las ecuaciones de Navier–Stokes, usadas para describir el movimiento de fluidos como el aire y el agua, son tan complejas que aún no sabemos si siempre tienen soluciones suaves en tres dimensiones? Resolver este enigma es uno de los Problemas del Milenio, y quien lo logre podría ganar un millón de dólares... además de cambiar nuestra comprensión de los fluidos para siempre.

Figura 1.18 Desde los 40's se tenía la idea de redes neuronales.

Esta necesidad de aproximación motiva el desarrollo de métodos numéricos y conduce naturalmente al modelado computacional, que se abordará en la siguiente sección.

Vista previa 2026

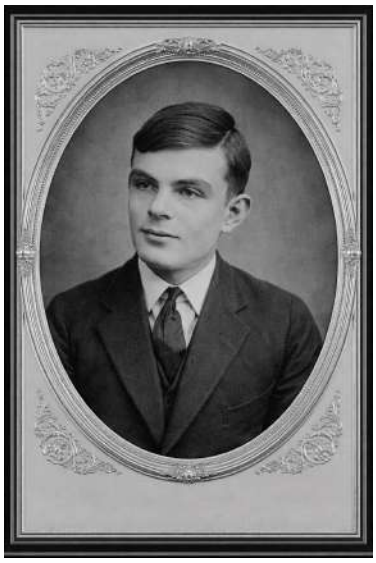


Figura 1.19 Retrato de Alan Turing.

1.4 MODELADO COMPUTACIONAL

"*Las horas pasaban y cada nuevo mensaje interceptado seguía sin descifrarse*". No se trataba solo de texto cifrado: detrás de cada señal de radio había convoyes en el Atlántico, ciudades bajo amenaza y miles de vidas en riesgo. La máquina alemana Enigma generaba millones de configuraciones posibles cada día, una cantidad imposible de revisar manualmente. Durante la Segunda Guerra Mundial, el matemático británico Alan Turing y su equipo en Bletchley Park diseñaron máquinas electromecánicas capaces de explorar sistemáticamente esas combinaciones y acelerar el proceso de descifrado. Aquellas máquinas, conocidas como *bombes*, mostraron de forma dramática el poder de delegar cálculos repetitivos a dispositivos capaces de realizarlos mucho más rápido que cualquier grupo de personas.

De manera similar, muchos de los problemas que surgen en el modelado matemático conducen a situaciones donde una solución exacta es difícil o imposible de obtener mediante una fórmula cerrada. En su lugar, el análisis del modelo requiere realizar una gran cantidad de cálculos: evaluaciones sucesivas, aproximaciones paso a paso o procedimientos iterativos que refinan progresivamente la solución. Aunque cada uno de estos cálculos puede ser sencillo por sí mismo, su número puede crecer rápidamente hasta volverse impracticable para realizarse manualmente. Es precisamente en este tipo de escenarios donde los métodos sistemáticos de aproximación y, hoy en día, el poder de las computadoras, se vuelven herramientas indispensables.



Figura 1.20 El camino al modelado es difícil y muchas veces se requiere estudiar trabajos previos aún cuando estén en otros idiomas.

Sin embargo, la idea de aproximar soluciones mediante cálculos sucesivos es mucho más antigua que las computadoras. Ya en el siglo XVIII, Leonhard Euler propuso un procedimiento sencillo pero profundo: construir la solución de una ecuación diferencial paso a paso a partir de su comportamiento local. Este principio, aproximar una solución mediante una secuencia de cálculos elementales, se convirtió con el tiempo en la base de una amplia familia de técnicas conocidas hoy como métodos numéricos. Con la aparición de las computadoras en el siglo XX, estas ideas adquirieron una nueva dimensión: las ecuaciones diferenciales podían transformarse en algoritmos capaces de ejecutarse millones de veces por segundo.

Así nació el modelado computacional: un enfoque en el que los modelos matemáticos ya no buscan únicamente fórmulas exactas, sino que también se traducen en algoritmos capaces de explorar y pronosticar el comportamiento de sistemas complejos. En este contexto, las ecuaciones matemáticas se transforman en procedimientos que una computadora puede ejecutar de manera eficiente, permitiendo realizar simulaciones que ayudan a comprender fenómenos cuya solución analítica es difícil o incluso imposible de obtener.

Vista previa 2026



Figura 1.21 El mundo continuo y el mundo discreto.

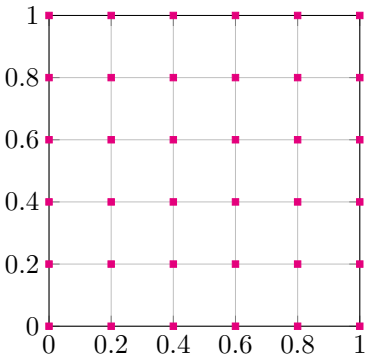


Figura 1.22 El dominio discreto consta de puntos estructurados.

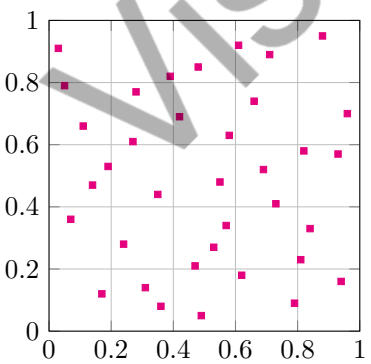


Figura 1.23 El dominio ahora consta de puntos aleatorios.

1.5 DE LAS ECUACIONES A LA SIMULACIÓN

El paso de las ecuaciones diferenciales a la simulación implica una transformación conceptual y técnica: traducir un modelo formulado en términos continuos a una representación que pueda ser manipulada por una computadora.

Este proceso no es inmediato, pues requiere, por un lado, adaptar la descripción del fenómeno a un conjunto finito de elementos manejables y, por otro, seleccionar estrategias numéricas que permitan aproximar su solución de manera eficiente. Tradicionalmente, esto se ha abordado mediante la discretización del problema y el uso de métodos numéricos bien establecidos; sin embargo, en años recientes han surgido enfoques alternativos que replantean esta transición desde una perspectiva distinta.

En las siguientes subsecciones se describen estos elementos, desde la construcción del modelo discreto hasta las herramientas clásicas de simulación y las propuestas más recientes basadas en aprendizaje automático.

1.5.1 Del continuo al discreto

Los modelos matemáticos que describen fenómenos físicos suelen formularse en términos de variables continuas. Por ejemplo, la temperatura en una barra metálica puede considerarse definida en cada punto del espacio y en cada instante del tiempo. Si la idea es resolver mediante aproximaciones, es necesrio apoyarse de una computadora, sin embargo, las computadoras no operan sobre cantidades continuas, sino sobre representaciones discretas de la información.

Por esta razón, el primer paso del modelado computacional consiste en transformar el problema continuo en uno discreto. Este proceso se conoce como *discretización*. En lugar de considerar todos los valores posibles de una variable, algo imposible de manejar directamente en una computadora, se selecciona un conjunto finito de puntos donde el modelo será evaluado.

Por ejemplo, el tiempo puede dividirse en intervalos de tamaño Δt, mientras que el espacio puede representarse mediante un conjunto de puntos distribuidos a lo largo del dominio de interés. Dependiendo del método numérico que se utilice, estos puntos pueden organizarse en mallas estructuradas, en distribuciones no estructuradas o incluso generarse de manera aleatoria. La elección de esta distribución forma parte de la estrategia de modelado y puede influir tanto en la precisión del método como en el costo computacional del cálculo.

Las derivadas que aparecen en las ecuaciones diferenciales deben ser tratadas de manera que puedan evaluarse computacionalmente; esto

Vista previa 2026

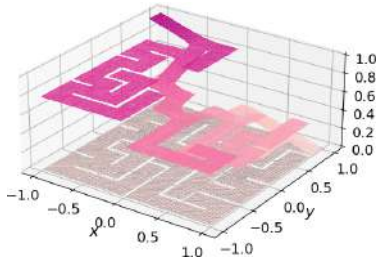


Figura 1.24 Distribución de temperatura en una placa recortada, donde la temperatura se representa como altura y toma valores en el intervalo $[0, 1]$.

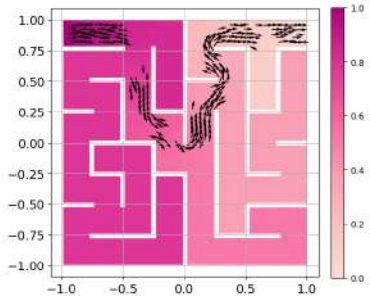


Figura 1.25 El gradiente de la temperatura puede interpretarse como la dirección de salida de un laberinto.

se logra mediante diversas aproximaciones que utilizan información en puntos discretos del dominio, ya sea aproximando directamente las derivadas, reformulando el problema o representando la solución mediante funciones base adecuadas. A partir de esta transformación, el problema original se convierte en un sistema de ecuaciones algebraicas que puede resolverse mediante cálculos sucesivos, permitiendo que el modelo sea tratado computacionalmente.

1.5.2 Métodos y simulación científica

Una vez que el modelo ha sido discretizado, es necesario emplear técnicas numéricas para aproximar la solución de las ecuaciones resultantes. A lo largo del tiempo se han desarrollado numerosos métodos con este propósito. En el caso de ecuaciones diferenciales ordinarias, son comunes esquemas como el método de Euler o los métodos de Runge–Kutta, mientras que para ecuaciones diferenciales parciales se utilizan con frecuencia técnicas como diferencias finitas, elementos finitos o volúmenes finitos. El desafío central radica en cómo tratar las derivadas que aparecen en las ecuaciones, ya que estas no pueden evaluarse directamente en una computadora cuando el modelo es continuo.

Una estrategia habitual consiste en aproximarlas mediante diferencias entre valores cercanos, lo que da lugar a los métodos de *diferencias finitas*. Sin embargo, esta no es la única posibilidad. También es posible reformular el problema en términos de derivadas débiles, lo que conduce a métodos como el de *elementos finitos* (FEM) o el de *volúmenes finitos* (FVM); otra alternativa consiste en aproximar la solución mediante funciones base elegidas de antemano, siguiendo un enfoque tipo *ansatz*, como ocurre en los métodos basados en *funciones de base radial* (RBF).

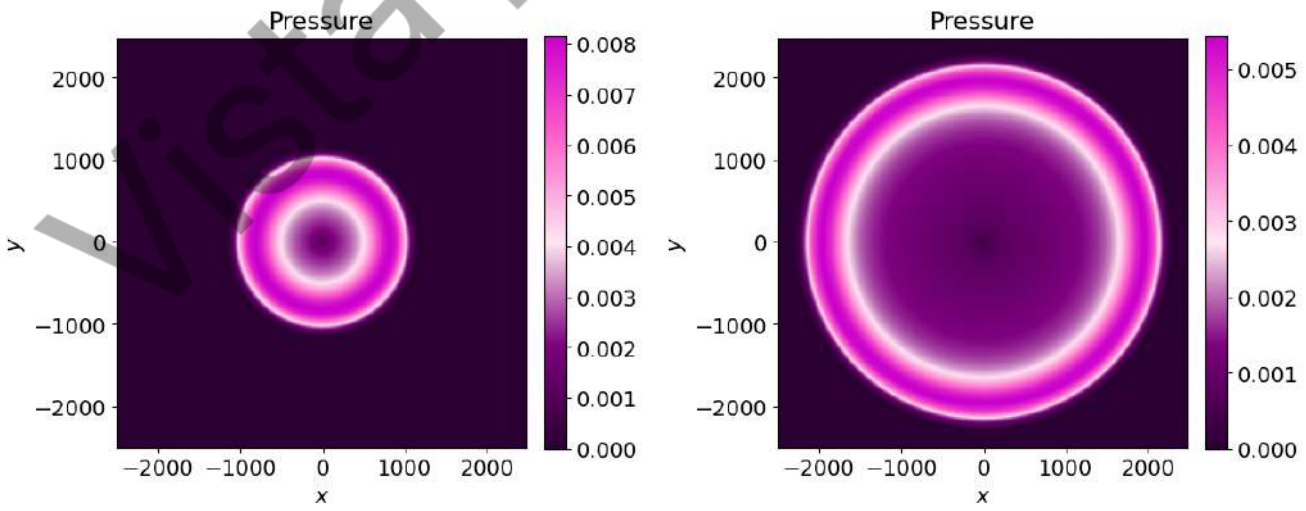


Figura 1.26 Izquierda: Propagación de una onda acústica en un medio homogéneo e isotrópico en el tiempo $t = t_1$.. Derecha: Propagación de una onda acústica en un medio homogéneo e isotrópico en el tiempo $t = t_2$.

Vista previa 2026

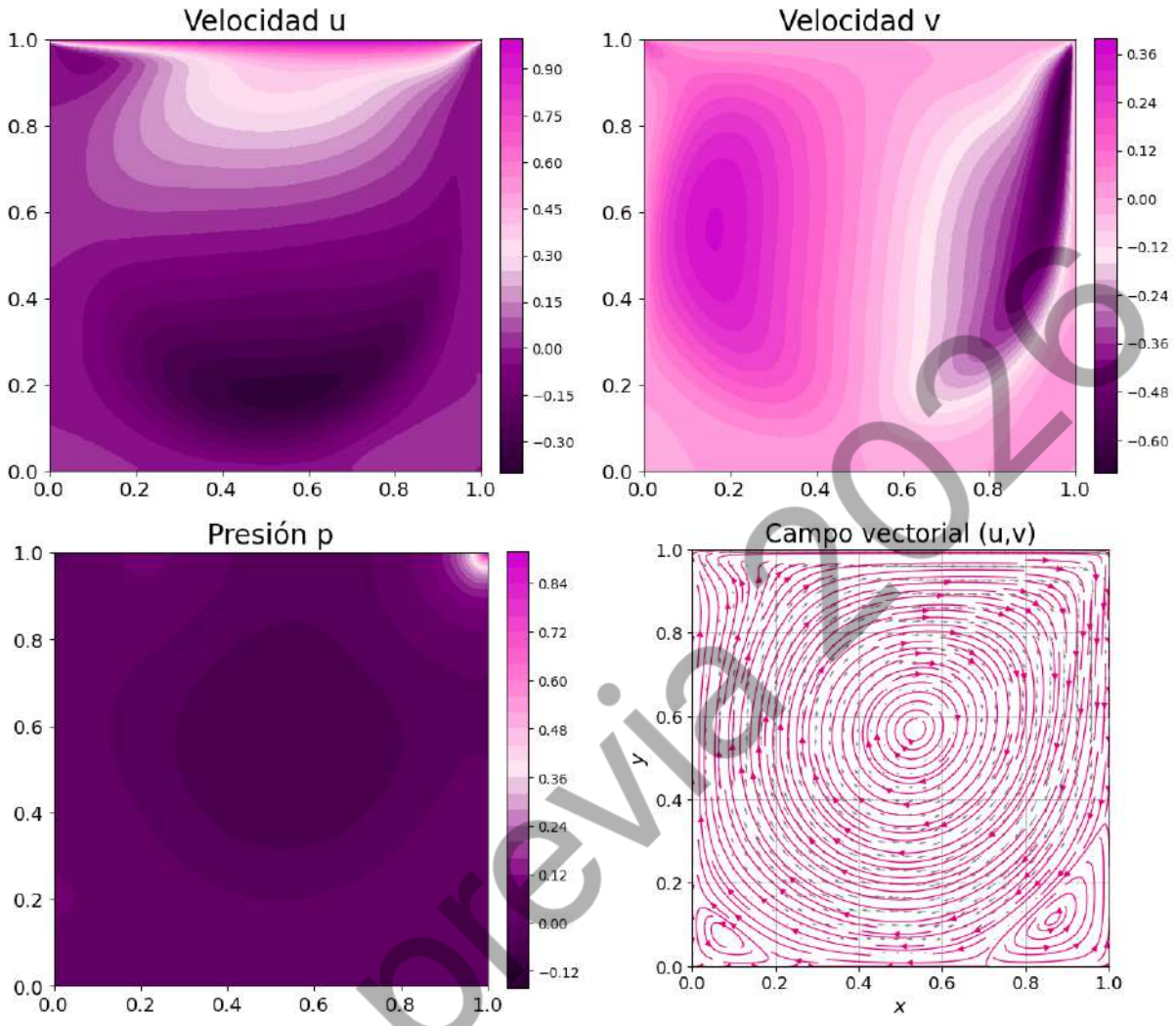


Figura 1.27 De izquierda a derecha y de arriba hacia abajo: contornos de nivel de la velocidad horizontal u, de la velocidad vertical v, de la presión p y el campo vectorial de velocidades (u, v) con líneas de corriente en la cavidad de Ghía.

Vista previa 2026

Uno de los antecedentes más conocidos es el modelo de neurona propuesto por McCulloch y Pitts en 1943, seguido por el *perceptrón* introducido por Rosenblatt en 1957.

Figura 1.28 Desde los 40's se tenía la idea de redes neuronales.

Cada una de estas estrategias traduce las ecuaciones diferenciales en una forma que puede manejarse computacionalmente. En todos los casos, el resultado final es un sistema de ecuaciones algebraicas que puede resolverse mediante cálculos directos o sucesivos. En este punto aparece otro problema de interés: cómo resolver de manera eficiente el sistema lineal resultante. La estructura de dicho sistema depende del método de discretización seleccionado; por lo tanto, el método utilizado para resolverlo también debe elegirse teniendo en cuenta esa estructura.

Las redes neuronales artificiales como nuevo modelo computacional

Una herramienta más reciente, que no sigue exactamente las estrategias clásicas de discretización pero que también permite aproximar soluciones de ecuaciones diferenciales, es el uso de *redes neuronales arti-*

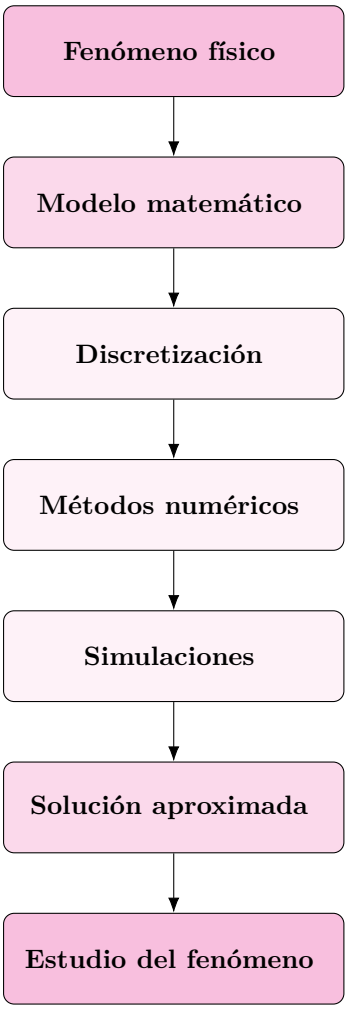


Figura 1.29 Esquema general del proceso estándar del modelado matemático y computacional de un fenómeno físico.

ficiales. Las redes neuronales artificiales tienen su origen en los pxrimeros modelos inspirados en el funcionamiento del cerebro. Con el paso de las décadas, y gracias al aumento en la capacidad de cómputo y al desarrollo de nuevos algoritmos de entrenamiento como la retropropagación, las redes neuronales se convirtieron en una herramienta poderosa para aproximar funciones complejas a partir de datos.

Más recientemente surgió una variante conocida como Redes neuronales artificiales informados por la física (o PINNs del inglés Physics Informed Neural Networks). En este enfoque, propuesto formalmente alrededor de 2017–2019, la red neuronal no se entrena únicamente con datos observados, sino también incorporando directamente las ecuaciones diferenciales que describen el fenómeno físico. En lugar de discretizar explícitamente los operadores diferenciales mediante mallas o esquemas numéricos tradicionales, la red se entrena para que su salida satisfaga la ecuación diferencial y las condiciones iniciales o de frontera. De esta forma, la ecuación física actúa como parte de la función de pérdida durante el entrenamiento.

Este enfoque ha abierto una línea de investigación activa en modelado científico y aprendizaje automático, donde los modelos combinan información de datos con principios físicos conocidos.

La elección del método, por lo tanto, no es únicamente técnica: forma parte del arte del modelado, pues depende del tipo de ecuación, de la geometría del dominio, de los datos disponibles y del equilibrio que se busque entre precisión y costo computacional. Cuando estos métodos se implementan en programas de computadora, dan lugar a *simulaciones* capaces de reproducir el comportamiento de sistemas complejos.

Hoy en día, la simulación científica se ha convertido en una herramienta fundamental para la investigación y la ingeniería. Gracias al modelado computacional es posible estudiar fenómenos que serían difíciles o imposibles de analizar únicamente mediante experimentos o soluciones analíticas, como la dinámica del clima, la propagación de ondas, el flujo de fluidos o la evolución de sistemas biológicos.

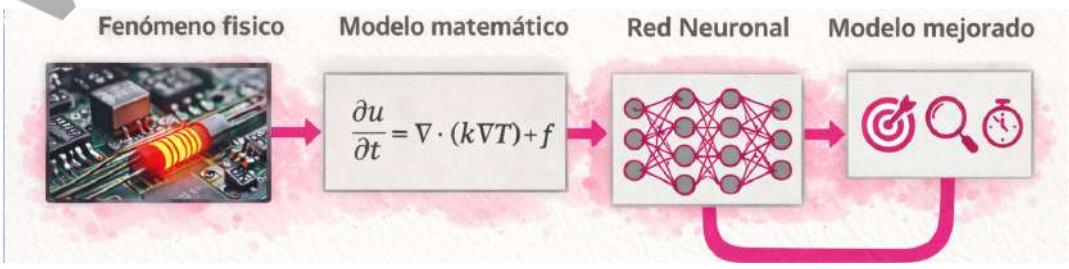


Figura 1.30 Flujo de modelado: del fenómeno físico al modelo mejorado mediante PINNs.

Vista previa 2026

RESUMEN

El recorrido presentado muestra que el modelado matemático no es una construcción abstracta aislada, sino el resultado de una evolución histórica impulsada por la necesidad de entender el entorno. Desde la observación de patrones en la naturaleza hasta la formulación de leyes físicas, y el desarrollo del cálculo marcó un punto de inflexión al permitir describir el cambio de manera precisa mediante ecuaciones diferenciales.

Estas ecuaciones constituyen el núcleo del modelado de fenómenos físicos, ya que permiten expresar relaciones fundamentales entre magnitudes y sus tasas de variación. Sin embargo, la complejidad de muchos sistemas reales conduce a ecuaciones diferenciales altamente complejas, lo que limita la obtención de soluciones analíticas y hace indispensable el uso de métodos numéricos, tales como diferencias finitas, elementos finitos o volúmenes finitos, entre otros.

El surgimiento del modelado computacional representa una extensión natural de esta necesidad: transformar modelos continuos en problemas discretos que puedan resolverse mediante algoritmos. Este enfoque ha permitido estudiar sistemas cada vez más complejos y ha consolidado la simulación como una herramienta esencial en la ciencia moderna.

Finalmente, la incorporación de técnicas recientes, como las redes neuronales informadas por la física, evidencia que el modelado continúa evolucionando. Hoy en día, no solo se busca resolver ecuaciones, sino integrar datos, teoría y cómputo en marcos más flexibles y potentes.

En esencia, modelar es traducir la realidad a un lenguaje que permita comprenderla. Y en ese proceso, las matemáticas, el cómputo y la observación siguen siendo piezas inseparables de una misma idea: entender cómo cambia el mundo.

Vista previa 2026

CAPÍTULO 2

INTRODUCCIÓN A LAS REDES NEURONALES ARTIFICIALES

En este capítulo introductorio se presentan los conceptos fundamentales comúnmente utilizados en el estudio de las redes neuronales artificiales. Iniciamos con una discusión general sobre qué es una red neuronal artificial, haciendo énfasis en su forma más simple: *el perceptrón*.

A partir de esta base, se introducirán los elementos clave involucrados en la construcción y resolución de problemas mediante esta red, tales como la arquitectura, el proceso de entrenamiento y los mecanismos de ajuste de parámetros. Finalmente, se revisarán algunos ejemplos ilustrativos que permitirán contextualizar y afianzar los conceptos presentados.

OBJETIVOS

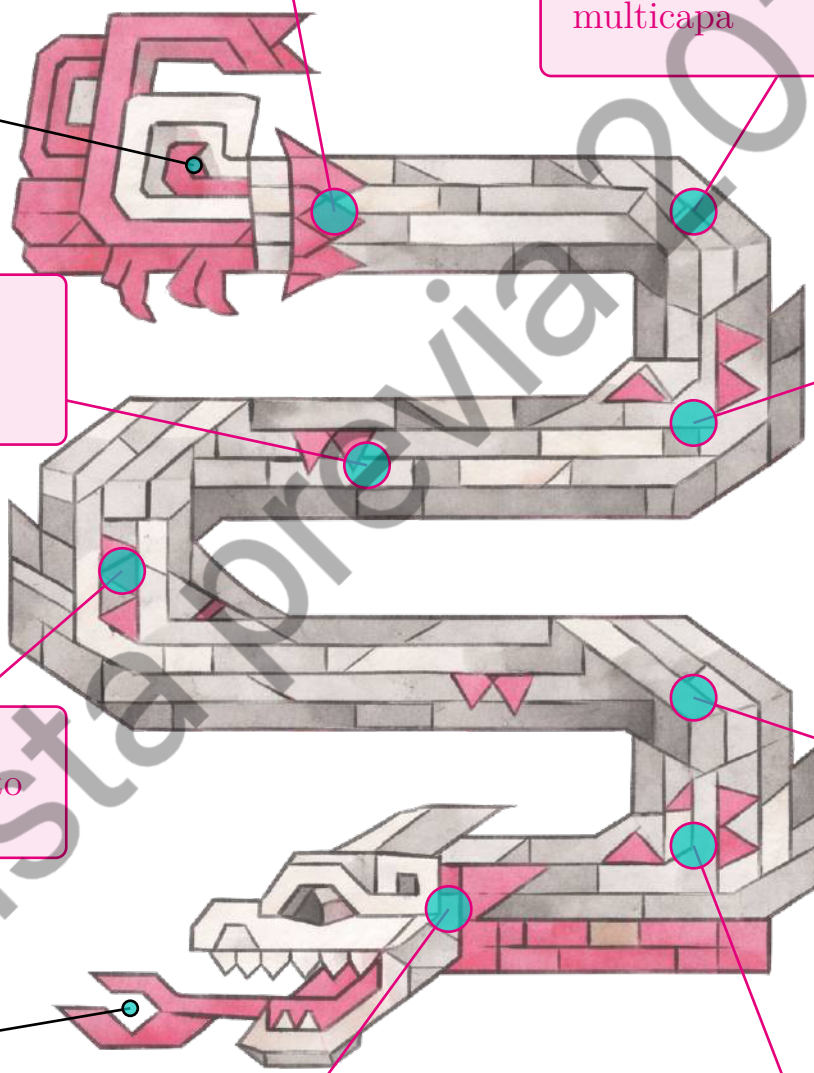
Al finalizar este capítulo, el lector será capaz de:

- Definir qué es una red neuronal artificial y describir sus componentes fundamentales.
- Explicar el funcionamiento y la estructura del perceptrón como el modelo más simple de una red neuronal.
- Identificar los conceptos básicos involucrados en la resolución de problemas mediante redes neuronales, tales como arquitectura, entrenamiento y ajuste de parámetros.
- Analizar ejemplos sencillos de aplicación de redes neuronales con el fin de reforzar los conceptos teóricos presentados.

Vista previa 2026

CONTENIDO

Introducción
2.2 El perceptrón
2.3 Redes neuronales multicapa
2.4 Funciones de activación
2.5 Hacia el cálculo de los pesos y sesgos
2.6 Entrenamiento
2.7 Iteraciones con retropropagación
2.8 Entrenamiento para redes complejas
2.9 Librerías para redes neuronales
Resumen



Vista previa 2026



Figura 2.1 El uso de plataformas basadas en inteligencia artificial se ha vuelto cada vez más común, particularmente entre las nuevas generaciones.

2.1 INTRODUCCIÓN

Hoy en día, la inteligencia artificial (*Artificial Intelligence*, AI) ocupa un lugar central en la ciencia, la tecnología y nuestra vida cotidiana. Su impacto ha sido tan profundo que hablar de ella resulta casi inevitable, no solo por la magnitud de sus avances, sino por la diversidad de áreas en las que ha transformado nuestra forma de trabajar, analizar y comprender problemas. Desde la salud hasta la ingeniería, e incluso en campos que durante mucho tiempo parecieron exclusivos del razonamiento humano, como las matemáticas, la inteligencia artificial ha demostrado un potencial extraordinario.

Resulta sorprendente pensar que, hasta hace relativamente pocos años, muchas de estas herramientas simplemente no existían. El ritmo de desarrollo ha sido tan vertiginoso que, en ocasiones, sus aplicaciones parecen adelantarse a la comprensión completa de los principios teóricos que las sustentan. Sin embargo, lejos de representar una barrera, esta aparente complejidad constituye una oportunidad invaluable para aprender, explorar y profundizar en un área fascinante. En particular, es estimulante observar su interacción con problemas tradicionalmente dominados por enfoques analíticos y numéricos, como la obtención de soluciones aproximadas de ecuaciones diferenciales. Este encuentro entre disciplinas abre nuevas perspectivas y plantea preguntas sumamente interesantes.

Vista previa 2026

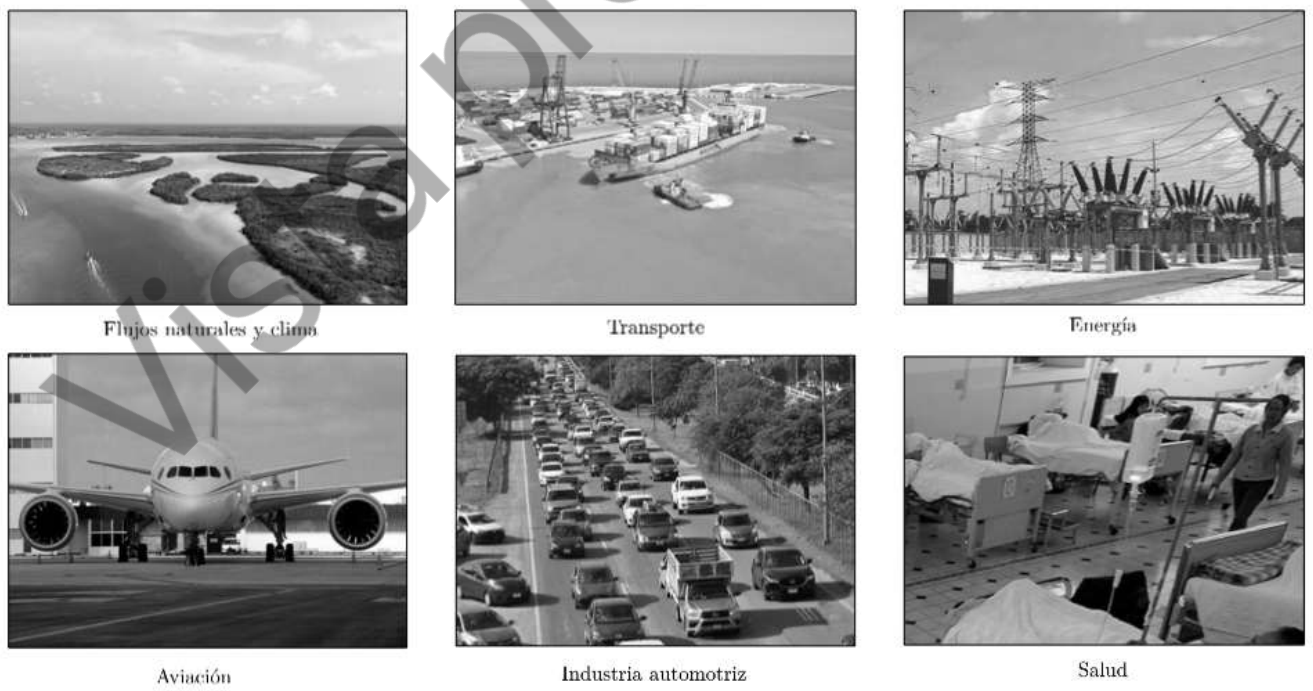


Figura 2.2 Hoy en día, la inteligencia artificial forma parte de múltiples avances en ciencia e ingeniería.

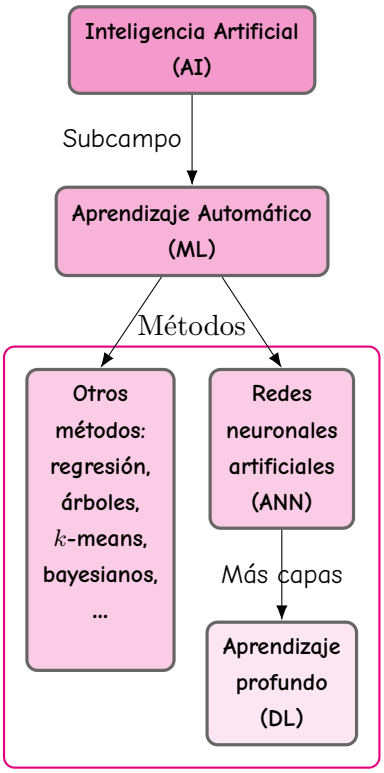


Figura 2.3 Visión general del panorama del aprendizaje automático.

2.1.1 El aprendizaje automático

El *Aprendizaje Automático* (mejor conocido como *Machine Learning*, ML) es un subcampo de la inteligencia artificial enfocado en el desarrollo de algoritmos capaces de identificar y aprender patrones a partir de datos, con el propósito de tomar decisiones o realizar predicciones sin necesidad de ser programados explícitamente para cada situación. Entre las técnicas más representativas de este campo se encuentran las *Redes Neuronales Artificiales* (*Artificial Neural Networks*, ANN), la regresión lineal, los árboles de decisión, las máquinas de soporte vectorial, el algoritmo k-means y los métodos bayesianos, entre otras.

Las redes neuronales artificiales difieren de otras técnicas de aprendizaje automático principalmente en dos aspectos: su flexibilidad y su interpretabilidad. Por un lado, las redes neuronales destacan por su capacidad para aproximar funciones de gran complejidad, aunque esta propiedad suele venir acompañada de una mayor exigencia en términos de datos y recursos computacionales. Por otro lado, otros modelos clásicos de aprendizaje automático ofrecen estructuras cuya interpretación resulta más transparente y estrechamente vinculada con el contexto del problema.

2.1.2 Las redes neuronales artificiales

Hasta hace algunos años, el término redes neuronales se asociaba casi exclusivamente con las células del cerebro, estructuras biológicas que se comunican entre sí y hacen posible su funcionamiento. Décadas de investigación han mostrado que el cerebro humano está formado por millones de estas células, cuya activación depende de la actividad, los estímulos y los procesos cognitivos involucrados.

Precisamente en este principio biológico encuentra su inspiración una de las herramientas más influyentes de la inteligencia artificial: las *redes neuronales artificiales* [1]. Estos modelos están compuestos por unidades básicas, denominadas neuronas (también denominadas nodos), interconectadas entre sí y organizadas por capas y cada neurona encargada de realizar operaciones simples. Sin embargo, al actuar de manera conjunta, dan lugar a sistemas capaces de abordar problemas complejos desde una perspectiva distinta a los enfoques tradicionales.

Cuando las neuronas se organizan y conectan a través de múltiples capas, se habla de *redes neuronales multicapa* (MLP). Dentro de este marco se encuentra el *Aprendizaje Profundo*, comúnmente conocido como *Deep Learning* (DL), el cual se caracteriza por el uso de redes neuronales con un número significativo de capas llamadas *ocultas*. Esta arquitectura proporciona una mayor capacidad de representación y flexibilidad al modelo, aunque a costa de una demanda significativamente mayor de recursos computacionales durante el proceso de entrenamiento.

Notas

- ANN $\subset$ ML
- No todo ML usa ANN
- DL $\subset$ ANN

Figura 2.4 Relaciones clave.

Vista previa 2026



Figura 2.5 Imagen ilustrativa del cerebro humano y sus neuronas.



Figura 2.6 Iniciamos nuestro recorrido por los conceptos fundamentales necesarios para comprender las redes neuronales.

Las redes neuronales artificiales, que en adelante denominaremos de forma abreviada como *redes neuronales*, constituyen una metodología que, sin duda, se ha consolidado como una herramienta fundamental debido a su versatilidad, facilidad de implementación y capacidad para manejar grandes volúmenes de datos. En términos generales, se emplean para extraer patrones y características a partir de conjuntos de datos previamente conocidos.

Una de las principales fortalezas de las redes neuronales radica en la diversidad de datos que pueden procesar, los cuales abarcan desde puntos asociados a funciones matemáticas e imágenes categorizadas, entre muchos otros. Esta flexibilidad permite abordar distintos tipos de problemas, como la clasificación de datos en categorías específicas o la estimación de resultados para nuevas observaciones. Asimismo, estas metodologías pueden aplicarse a contextos más complejos, como el estudio de ecuaciones diferenciales, tema central de este libro.

En las siguientes secciones iniciaremos un recorrido por los conceptos fundamentales necesarios para comprender estas herramientas. El objetivo no es únicamente aprender técnicas, sino también desarrollar una intuición clara que permita entender lo que, para muchos, aún se percibe como una *caja negra*. A lo largo de este capítulo, definiremos de manera gradual los principios, conceptos y herramientas matemáticas y computacionales que servirán como base para los desarrollos posteriores. Este enfoque permitirá, en capítulos subsecuentes, utilizar de forma efectiva los conceptos presentados.

Vista previa

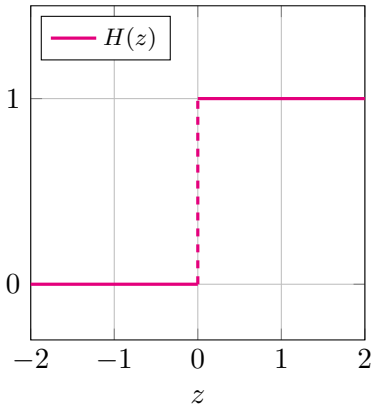


Figura 2.7 Función escalón usado como función de activación en el perceptrón.

2.2 EL PERCEPTRÓN

Desde una perspectiva matemática, una red neuronal puede interpretarse como la composición de diversas funciones matemáticas que, actuando de manera conjunta, permiten modelar y aproximar relaciones de elevada complejidad. Para comprender con mayor profundidad este concepto, resulta natural comenzar con la arquitectura más sencilla.

El *perceptrón*, introducido en 1957 por Frank Rosenblatt [2], representa una de las estructuras fundamentales dentro de las redes neuronales. En este modelo, tanto las entradas como la salida se expresan mediante valores numéricos, y cada conexión entre neuronas se encuentra asociada a un peso que determina la influencia de la información transmitida.

DEFINICIÓN

Sean $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ y $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)^T \in \mathbb{R}^n$ los vectores de entradas y pesos, respectivamente. El *perceptrón* es la función $\widetilde{u} : \mathbb{R}^n \to \mathbb{R}$ calculada como $\widetilde{u}(\boldsymbol{x}; \boldsymbol{w}) = H(z)$ donde

$$z = \boldsymbol{w}^T \boldsymbol{x} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n, \tag{2.1}$$

es la suma ponderada de sus entradas y

$$H(z) = \begin{cases} 1, & \text{si } z \geq 0, \\ 0, & \text{si } z < 0, \end{cases} \tag{2.2}$$

es una función escalón, denominada *función de activación*.

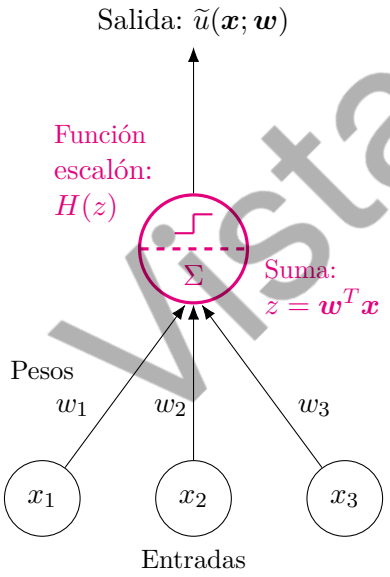


Figura 2.8 Esquema gráfico de las principales partes de un perceptrón con tres valores de entrada.

Una forma habitual de representar las redes neuronales es mediante esquemas gráficos, ya que estos permiten visualizar de manera intuitiva tanto la estructura del modelo como el flujo de información entre sus componentes. Si bien dichas representaciones pueden variar ligeramente según el contexto o la notación adoptada, en la Figura 2.8 se muestra una versión del modelo original propuesto por Rosenblatt, correspondiente a un perceptrón con tres entradas. El flujo de información en esta representación se considera de abajo hacia arriba. En el diagrama, el símbolo (Σ) denota la operación de suma ponderada que define la variable z, tal como se establece en la ecuación (2.1). Posteriormente, el símbolo ($\sqcap$) representa la aplicación de la función escalón dado en la ecuación (2.2), la cual determina la salida del modelo.

Originalmente, el perceptrón fue diseñado para resolver problemas de clasificación, lo que justifica el uso de una función escalón que genera salidas binarias. Sin embargo, la evolución de las redes neuronales ha permitido generalizar la arquitectura de la red y ha ampliado notablemente las capacidades del modelo, posibilitando su aplicación a problemas más complejos.

Vista previa 2026

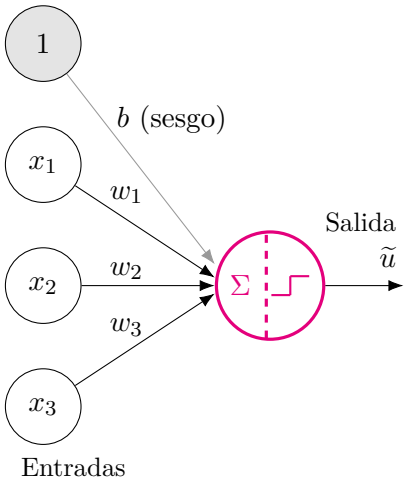


Figura 2.9 Esquema gráfico de las principales partes de un perceptrón con sesgo. Los pesos corresponden a los valores w_1, w_2 y w_3, mientras que el sesgo está determinado por b.

2.2.1 Los perceptrones con sesgos

La incorporación de un término de sesgo en el modelo del perceptrón responde a la necesidad de dotar al sistema de mayor flexibilidad en su capacidad de representación posibilitando un desplazamiento, dado que en ausencia de este término la salida del modelo depende exclusivamente de combinaciones de las entradas.

DEFINICIÓN

Una extensión natural del perceptrón consiste en introducir un *sesgo* (*bias*), el cual se modela mediante un parámetro real $b \in \mathbb{R}$. En este caso, la función $\widetilde{u} : \mathbb{R}^n \to \mathbb{R}$ se define a partir de una suma ponderada de las entradas y sumando el término de sesgo:

$$z = \boldsymbol{w}^T \boldsymbol{x} + b = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b, \tag{2.3}$$

seguida de la aplicación de la función de activación previamente definida en (2.2), obteniendo la salida:

$$\widetilde{u}(\boldsymbol{x}; \boldsymbol{w}, b) = H(z). \tag{2.4}$$

La Figura 2.9 muestra una versión generalizada del modelo ya propuesto. En esta representación se incorpora explícitamente el sesgo b. Ahora, el flujo de información se considera de izquierda a derecha, de acuerdo con la convención habitual empleada en la literatura. Notar como se incorpora el nodo gris del sesgo cuya entrada del valor siempre es 1. El nodo rosa del diagrama representa una neurona, a la cual se le asocian un conjunto de pesos, un término de sesgo y una función de activación. Las flechas indican el flujo de información, desde las entradas hacia la neurona, mientras que la salida generada puede emplearse como entrada de otras neuronas en estructuras más complejas.

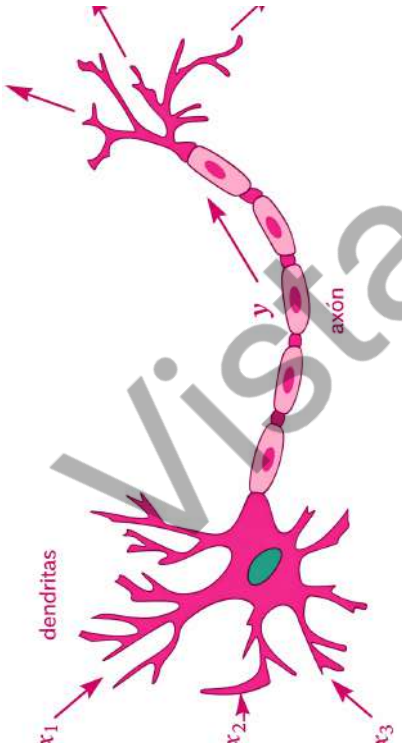


Figura 2.10 Representación clásica de lo que es una neurona biológica

En este punto, la inspiración biológica del modelo se vuelve más evidente. De manera simplificada, una neurona biológica consta de dendritas, cuerpo celular y axón. Las dendritas actúan como receptores. Estas señales son procesadas en el cuerpo celular. Al superar un umbral, se genera un impulso transmitido por el axón. En el modelo artificial, esta idea se refleja de forma natural: las entradas corresponden al vector $\boldsymbol{x}$, el nodo representa el mecanismo de integración, y la salida reproduce el comportamiento de activación. En particular, al emplear una función escalón, el modelo emite una respuesta cuando la señal total rebasa el umbral establecido.

Antes de continuar con el estudio de redes neuronales más complejas, es oportuno ejemplificar la aplicación de los modelos previamente presentados. Este ejercicio facilitará la comprensión de su funcionamiento y permitirá establecer una base sólida para abordar arquitecturas de mayor profundidad.

Vista previa 2026

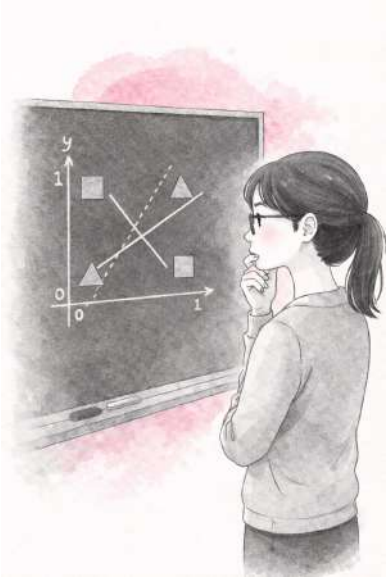


Figura 2.11 Representación geométrica del problema XOR. No existe una recta que separe correctamente las clases.

2.2.2 Ejemplo: el problema de clasificación XOR

Iniciaremos con uno de los ejemplos más representativos en redes neuronales y aprendizaje automático: el problema XOR (o exclusivo), que se plantea a continuación.

EL PROBLEMA XOR

Considérese un conjunto de datos compuesto por pares ordenados (x_1, x_2), donde $x_1, x_2 \in \{0, 1\}$. Se desea construir un modelo de clasificación binaria cuya salida $y \in \{0, 1\}$ satisfaga la siguiente regla:

- $y = 1$ cuando las entradas son distintas ($x_1 \neq x_2$),
- $y = 0$ cuando las entradas son iguales ($x_1 = x_2$).

El objetivo es determinar si un perceptrón simple es capaz de representar esta función de clasificación.

En 1969, Marvin Minsky y Seymour Papert, en su monografía *Perceptrons*, señalaron una limitación fundamental del perceptrón simple: su incapacidad para resolver problemas aparentemente elementales, como el problema XOR.

Desde una perspectiva geométrica, el problema XOR puede visualizarse en el plano bidimensional considerando cada patrón de entrada como un punto. En el caso de la función XOR, los patrones asociados a la clase 0, $(0, 0)$ y $(1, 1)$, se ubican en vértices opuestos del cuadrado unitario, mientras que los patrones de la clase 1, $(0, 1)$ y $(1, 0)$, ocupan los vértices restantes. La Figura 2.11 muestra este problema de clasificación usando cuadrados y triángulos.

Podemos ver que la disposición de las clases genera una configuración en la que no existe una única recta capaz de separar correctamente ambas clases. En otras palabras, las clases no son linealmente separables, lo que explica la incapacidad de un perceptrón simple, cuya capacidad de decisión es lineal, para representar la función XOR.

Para establecer una perspectiva matemática más rigurosa que el perceptrón simple no puede representar el problema XOR, recurriremos a un argumento por contradicción. Consideremos un perceptrón con dos variables de entrada dada por

$$\widetilde{u}(x_1, x_2; w_1, w_2, b) = H(z), \qquad \text{donde} \qquad z = w_1 x_1 + w_2 x_2 + b,$$

y cuya representación se muestra en la Figura 2.12. La demostración inicia suponiendo que este perceptrón resuelve el problema XOR. Bajo esta hipótesis, existirían los parámetros w_1, w_2 y b tales que el modelo satisface la Tabla 2.1.

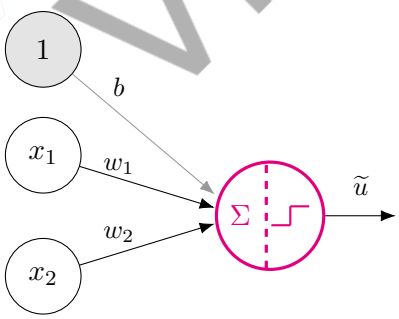


Figura 2.12 Perceptrón usado en el problema XOR.

Vista previa 2026

Tabla 2.1 Tabla de salidas del problema XOR.

x_1	x_2	Salida
0	0	0
0	1	1
1	0	1
1	1	0

Bajo la hipótesis de que el perceptrón representa la función XOR, analizamos primero los patrones cuya salida esperada es 0, es decir $H(z) = 0$.

- Al evaluar la entrada $(0, 0)$, se obtiene $z_{(0,0)} = w_1 \cdot 0 + w_2 \cdot 0 + b = b$. Dado que $H(z_{(0,0)}) = 0$, por definición de H debe cumplirse
$$b < 0.$$
- De manera análoga, al evaluar la entrada $(1, 1)$, resulta $z_{(1,1)} = w_1 + w_2 + b$. Como $H(z_{(1,1)}) = 0$, se concluye que
$$w_1 + w_2 + b < 0.$$

Sumando ambas desigualdades, se obtiene la condición

$$w_1 + w_2 + 2b < 0.$$

Consideremos ahora los patrones cuya salida es 1, es decir $H(z) = 1$.

- Al evaluar la entrada $(1, 0)$, se tiene $z_{(1,0)} = w_1 + b$. Como $H(z_{(1,0)}) = 1$, debe cumplirse
$$w_1 + b \geq 0.$$
- De forma similar, para la entrada $(0, 1)$ se obtiene $z_{(0,1)} = w_2 + b$, y la condición correspondiente es
$$w_2 + b \geq 0.$$

Sumando nuevamente las desigualdades anteriores, se deduce

$$w_1 + w_2 + 2b \geq 0.$$

Se observa que las condiciones obtenidas son incompatibles

$$w_1 + w_2 + 2b < 0 \quad \text{y} \quad w_1 + w_2 + 2b \geq 0,$$

lo cual constituye una contradicción. Por consiguiente, se concluye que un perceptrón simple no puede representar la función XOR. Esto concluye la demostración.

□



Figura 2.13 La clave para abordar problemas complejos reside en el trabajo organizado y coordinado en equipo.

Era de esperarse las limitaciones de la estructura descrita anteriormente dado que corresponde a la arquitectura más elemental dentro del campo de las redes neuronales. Si bien su formulación es conceptualmente simple, su importancia radica en que introduce la unidad fundamental de estos modelos: *la neurona artificial*. A partir de esta pieza básica es posible construir arquitecturas más sotisficadas, capaces de representar comportamientos considerablemente más complejos.

La principal fortaleza de las redes neuronales no reside en la neurona aislada, sino en la posibilidad de interconectar múltiples neuronas organizadas en capas. Esta idea conduce de manera natural a las *redes neuronales multicapa*. Bajo esta perspectiva, el problema XOR, que resulta irresoluble mediante un perceptrón simple, puede resolverse exitosamente introduciendo una *capa oculta*, como veremos en la siguiente sección.

Vista previa 2026



Figura 2.14 Mediante la interconexión estructurada de numerosas neuronas, red multicapa, se busca capturar relaciones que un modelo simple no podría representar.

2.3 REDES NEURONALES MULTICAPA

Como se expuso en la sección anterior, el perceptrón constituye un modelo fundamental que, si bien representa un punto de partida en el estudio de las redes neuronales, presenta limitaciones al abordar ciertos tipos de problemas. Sin embargo, al combinar múltiples neuronas es posible dotar al modelo de la flexibilidad necesaria para abordar tareas más complejas. Esta es, precisamente, la idea central del *perceptrón multicapa* (*Multilayer Perceptron*, MLP) o las *redes neuronales multicapa*. A partir de la unidad básica ahora definiremos una red neuronal contemplando únicamente el orden, cantidad y conexiones de las neuronas.

DEFINICIÓN
Se denomina *capa* al conjunto de neuronas que no presentan conexiones entre sí, pero que sí están conectadas con neuronas pertenecientes de otros niveles de la red. De manera gráfica, todos los nodos de una capa estarán ordenados en una misma columna.

La Figura 2.15 muestra dos ejemplos de redes neuronales multicapa que emplean como función de activación la misma función escalón previamente definida. Estos ejemplos de redes neuronales multicapa ilustran algunas de las múltiples configuraciones que pueden construirse a partir de la misma unidad básica.

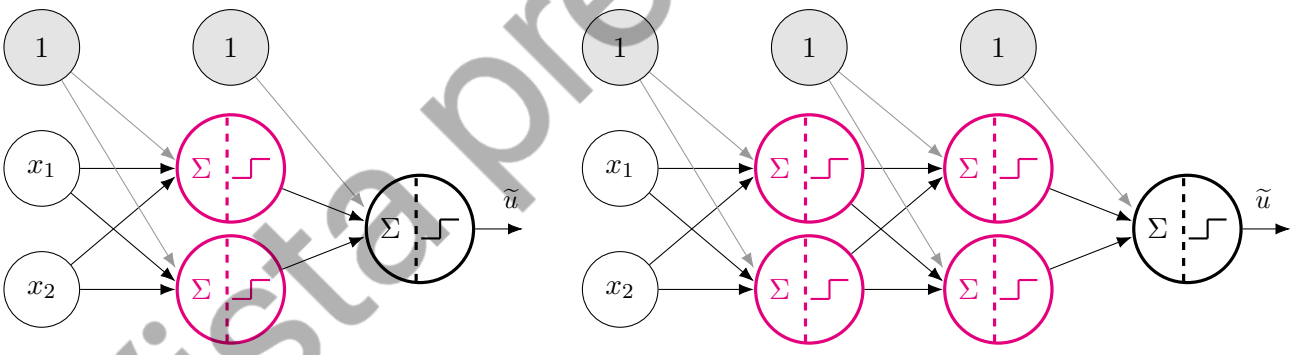


Figura 2.15 Ejemplos de redes neuronales multicapa. En las arquitecturas se ha omitido, por claridad visual, la notación correspondiente a los pesos (w_i) y sesgos (b) en las conexiones entre neuronas.

En nuestros ejemplos, ambas redes neuronales comparten las mismas dos entradas, x_1 y x_2, que hemos venido utilizando en el ejemplo de la sección anterior. Asimismo, en ambos casos se considera una única salida, representada por la función real $\widetilde{u}$. En contraste, la principal diferencia entre ambas configuraciones radica en el número de capas:

- En el primer caso se consideran la *capa de entrada* y dos capas de procesamiento: una capa intermedia, formada por dos neuronas, a la que denominaremos *capa oculta*, y una *capa de salida* compuesta por una sola neurona.

Vista previa 2026

Notas

- Capas de entrada.
- Capas ocultas.
- Capa de salida.

Figura 2.16 Tipo de capas.

- En el segundo ejemplo se consideran la capa de entrada y se incorporan tres capas de procesamiento: dos capas ocultas y una capa adicional de salida.

De manera general, y en función del papel que desempeñan dentro de la arquitectura, pueden distinguirse diversos tipos de capas:

- **Capa de entrada**: contiene las neuronas que representan las variables independientes. Su función es recibir y transmitir la información hacia las capas siguientes, sin realizar transformaciones adicionales.
- **Capa oculta**: está formada por un número determinado de neuronas ubicadas en el interior de la red. Puede haber una o varias capas ocultas, cada una con distinto número de neuronas. Con ellas se definen las características principales del modelo.
- **Capa de salida**: contiene las neuronas que produce la salida de la red. El número de ellas depende del problema considerado.

A medida que la red neuronal incrementa su complejidad, resulta más difícil mantener una representación detallada de todos sus elementos dado que el número de los parámetros crece considerablemente a medida que se incorporan más capas a la red. Si bien los detalles son fundamentales para el análisis matemático y la implementación computacional del modelo, su inclusión sistemática en todas las figuras dificultaría la interpretación visual de las arquitecturas más complejas. Por esta razón, en las dos arquitecturas presentadas en la Figura 2.15 se ha omitido, por claridad visual, la notación correspondiente a los pesos y sesgos asociados a las conexiones entre neuronas.

2.3.1 La red neuronal en términos de sus pesos y sesgos

Desde el punto de vista de funciones, la red neuronal puede describirse mediante las ecuaciones que se presentan a continuación, donde se detallan las operaciones realizadas en cada capa y el papel que desempeñan los pesos, los sesgos y la función de activación.

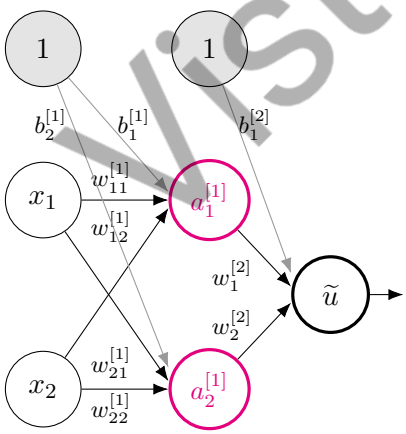


Figura 2.17 Pesos y sesgos de la red neuronal con dos capas

La Figura 2.17 presenta con mayor detalle la notación correspondiente a los pesos y sesgos de la red neuronal con una capa oculta con dos neuronas. En ella se hace explícita la estructura algebraica del modelo, mostrando cómo cada conexión entre neuronas está asociada a un peso específico y cómo cada neurona incorpora además un término de sesgo. Esta representación permite apreciar con claridad la cantidad de parámetros involucrados incluso en arquitecturas relativamente simples.

Para este ejemplo, consideremos el siguiente vector de entrada, la matriz de pesos y el vector de sesgos dados por:

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \qquad W^{[1]} = \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} \end{pmatrix}, \quad \text{y} \quad \boldsymbol{b}^{[1]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \end{pmatrix}. \tag{2.5}$$

Vista previa 2026

La red neuronal de dos capas puede escribirse como una expresión funcional de las entradas x_1 y x_2 de la forma:

$z_1^{[1]} = w_{11}^{[1]}x_1 + w_{12}^{[1]}x_2 + b_1^{[1]}$,

$a_1^{[1]} = H(z_1^{[1]})$,

$z_2^{[1]} = w_{21}^{[1]}x_1 + w_{22}^{[1]}x_2 + b_2^{[1]}$,

$a_2^{[1]} = H(z_2^{[1]})$,

$z_1^{[2]} = w_1^{[2]}a_1^{[1]} + w_2^{[2]}a_2^{[1]} + b_1^{[1]}$,

$\breve{u} = H(z_1^{[2]})$.

Figura 2.18 Red neuronal con una capa oculta de dos neuronas y dos valores de entrada.



Figura 2.19 Resulta pertinente preguntarse si esta red neuronal de una capa oculta y dos neuronas puede emplearse para resolver el problema XOR.

Entonces, la primera capa oculta está definida por

$$\boldsymbol{z}^{[1]} = W^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]} = \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \end{pmatrix}. \tag{2.6}$$

Aplicando la función de activación componente a componente, tenemos

$$\boldsymbol{a}^{[1]} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \end{pmatrix}. = H(\boldsymbol{z}^{[1]}) = \begin{pmatrix} H(z_1^{[1]}) \\ H(z_2^{[1]}) \end{pmatrix}. \tag{2.7}$$

Para la capa de salida, definimos

$$W^{[2]} = \begin{pmatrix} w_1^{[2]} & w_2^{[2]} \end{pmatrix}, \qquad b^{[2]} \in \mathbb{R}. \tag{2.8}$$

Entonces,

$$z^{[2]} = W^{[2]}\boldsymbol{a}^{[1]} + b^{[2]} = \begin{pmatrix} w_1^{[2]} & w_2^{[2]} \end{pmatrix}\begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \end{pmatrix} + b^{[2]}. \tag{2.9}$$

Finalmente, la salida de la red es

$$\breve{u} = H(z^{[2]}). \tag{2.10}$$

De manera resumida, las ecuaciones (2.5)–(2.10) pueden ser escritas como en la siguiente definición.

DEFINICIÓN

Una red neuronal multicapa $\breve{u}(\boldsymbol{x}; W, \boldsymbol{b})$ con dos entradas, una capa oculta compuesta por dos neuronas y una única salida se define mediante las siguientes ecuaciones:

$$\begin{aligned} \boldsymbol{z}^{[1]} &= W^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}, & \boldsymbol{a}^{[1]} &= H(\boldsymbol{z}^{[1]}), \\ z^{[2]} &= W^{[2]}\boldsymbol{a}^{[1]} + b^{[2]}, & \breve{u} &= H(z^{[2]}), \end{aligned}$$

donde $\boldsymbol{z}^{[1]} \in \mathbb{R}^2$ es el vector de la capa oculta antes de aplicar la función de activación, $\boldsymbol{a}^{[1]} \in \mathbb{R}^2$ es el vector resultante de aplicar la función de activación de manera componente a componente, el valor $z^{[2]} \in \mathbb{R}$ es la capa de salida antes de aplicar la función de activación y $\breve{u} \in \mathbb{R}$ representa la salida final de la red neuronal. Además tenemos que:

- $W^{[1]} \in M_{2\times 2}(\mathbb{R})$ es la matriz de pesos que conecta la capa de entrada con la capa oculta. Cada fila representa los pesos asociados a una de las 2 neuronas ocultas con respecto a las dos entradas x_1 y x_2.
- $\boldsymbol{b}^{[1]} \in \mathbb{R}^2$ es el vector de sesgos correspondiente a cada una de las neuronas de la capa oculta.
- $W^{[2]} \in \mathbb{R}^2$ es la matriz de pesos que conecta las salidas de las 2 neuronas ocultas con la única neurona de salida.
- $b^{[2]} \in \mathbb{R}$ es el sesgo de la neurona de salida.

Vista previa 2026

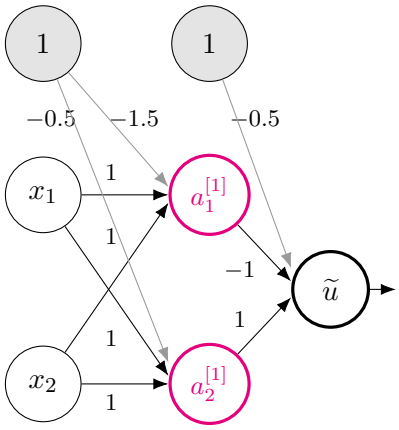


Figura 2.20 Pesos y sesgos de la red neuronal con dos capas para la solución del problema XOR.

2.3.2 Solución del problema XOR usando una capa oculta

Tal como se señaló al final del capítulo anterior, al incorporar redes multicapa es posible resolver el problema XOR. De hecho, la red neuronal con una sola capa oculta compuesta por dos neuronas, analizada previamente, resulta suficiente para este propósito. La solución está dada por:

$$\begin{aligned} z_1^{[1]} &= x_1 + x_2 - 1.5, \qquad a_1^{[1]} = H(z_1^{[1]}), \\ z_2^{[1]} &= x_1 + x_2 - 0.5, \qquad a_2^{[1]} = H(z_2^{[1]}), \\ z_1^{[2]} &= -a_1^{[1]} + a_2^{[1]} - 0.5, \quad \widetilde{u} = H(z_1^{[2]}). \end{aligned}$$

La representación gráfica correspondiente a esta configuración de pesos se muestra en la Figura 2.20.

Fácilmente puede verificarse que la red neuronal satisface los valores indicados en la Tabla 2.1. En efecto, al evaluar cada una de las posibles entradas se obtiene:

$$\begin{aligned} (0,0) \;&\rightarrow\; z_1^{[1]} = -1.5,\ z_2^{[1]} = -0.5,\ a_1^{[1]} = 0,\ a_2^{[1]} = 0,\ z_1^{[2]} = -0.5, \\ &\quad\ \widetilde{u}(0,0) = 0. \\ (0,1) \;&\rightarrow\; z_1^{[1]} = -0.5,\ z_2^{[1]} = 0.5,\ a_1^{[1]} = 0,\ a_2^{[1]} = 1,\ z_1^{[2]} = 0.5, \\ &\quad\ \widetilde{u}(0,1) = 1. \\ (1,0) \;&\rightarrow\; z_1^{[1]} = -0.5,\ z_2^{[1]} = 0.5,\ a_1^{[1]} = 0,\ a_2^{[1]} = 1,\ z_1^{[2]} = 0.5, \\ &\quad\ \widetilde{u}(1,0) = 1. \\ (1,1) \;&\rightarrow\; z_1^{[1]} = 0.5,\ z_2^{[1]} = 1.5,\ a_1^{[1]} = 1,\ a_2^{[1]} = 1,\ z_1^{[2]} = -0.5, \\ &\quad\ \widetilde{u}(1,1) = 0. \end{aligned}$$

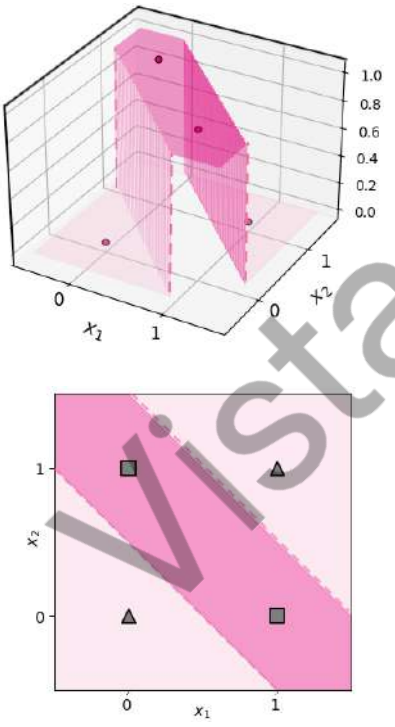


Figura 2.21 Solución de la red neuronal para el problema XOR. La superficie es discontinua (tipo escalón) del cual se forman dos regiones planas con valores 0 y 1.

Es así que puede apreciarse cómo la red neuronal, mediante la acción combinada de las dos neuronas en la capa oculta, logra generar una solución no lineal capaz de separar correctamente los datos del problema XOR. La Figura 2.21 ilustra la superficie de la solución determinada por $\widetilde{u} : \mathrm{R}^2 \rightarrow \mathrm{R}$, al igual que la región resultante que clasifica los puntos.

En este punto, la gran incógnita debe ser cómo se determinaron los valores de los pesos y los sesgos que permiten resolver el problema XOR. Sin embargo, aún no estamos en condiciones de responder plenamente a esta pregunta, pues es necesario profundizar primero en algunos conceptos fundamentales de las redes neuronales y en las herramientas matemáticas y computacionales que hacen posible su entrenamiento.

En las secciones siguientes desarrollaremos estos elementos con mayor detalle, lo que nos permitirá comprender no solo cómo se obtienen dichos parámetros, sino también por qué el procedimiento conduce efectivamente a una solución adecuada del problema planteado.

Vista previa 2026

Notas

La función ϕ denotará una función de activación general.

Figura 2.22 Función de activación.

2.3.3 Redes neuronales multicapa más complejas

No resulta difícil proponer redes neuronales $\widetilde{u}(\boldsymbol{x}; W, \boldsymbol{b})$ de mayor complejidad, ya sea incrementando el número de capas, aumentando la cantidad de neuronas en cada una de ellas o incorporando distintas funciones de activación, denominadas desde ahora con la función ϕ.

Por ejemplo, la Figura 2.23 presenta una red neuronal que incluye dos capas ocultas, formadas por tres y dos neuronas, respectivamente, además de una capa adicional de salida. Una diferencia relevante en esta configuración es que en la capa de salida se aplica únicamente la combinación lineal, esto es, la suma ponderada de las entradas mas el sesgo, sin introducir una función de activación. Si bien es posible diseñar arquitecturas con múltiples salidas como el presentado en la Figura 2.23, en las que la red aproxima simultáneamente varias cantidades de interés, en este texto no abordaremos este tipo de configuraciones.

Dado que nuestro objetivo es la aproximación de soluciones de ecuaciones diferenciales, centraremos el análisis en modelos con una sola salida, correspondientes a una función escalar $\widetilde{u}$.

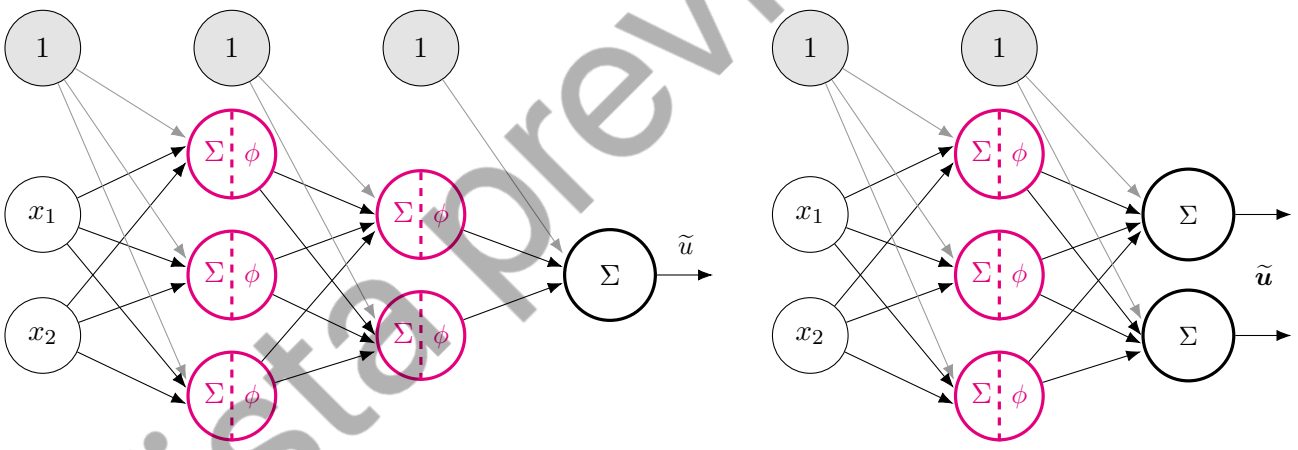


Figura 2.23 Más ejemplos de redes neuronales multicapa. En el primer ejemplo tenemos distintos números de neuronas por capa y en el segundo tenemos dos salidas en vez de una. Notar que la capa de salida, en ambos casos, se aplica únicamente la suma ponderada de las entradas mas el sesgo, sin introducir una función de activación.

Como se mencionó al presentar gráficamente el perceptrón, la representación gráfica de las redes neuronales puede variar ligeramente según la literatura consultada. A partir de este punto, para redes complejas, adoptaremos una versión más simplificada y uniforme de dicha representación: conservaremos el uso de colores para distinguir los distintos tipos de capas, pero omitiremos la notación explícita de pesos, sesgos y operadores de suma dentro de las capas ocultas. Asimismo, utilizaremos una representación más esquemática para los nodos asociados a los bias, con el fin de evitar sobrecargar visualmente las figuras.

Vista Previa 2026

La Figura 2.24 muestra un ejemplo de una red neuronal con múltiples capas, ilustrando la estructura que emplearemos para redes multicapas. Esta simplificación responde únicamente a razones didácticas y de claridad gráfica; no implica ninguna modificación en el modelo matemático subyacente. Las expresiones analíticas seguirán incorporando de manera explícita todos los parámetros y operaciones correspondientes.

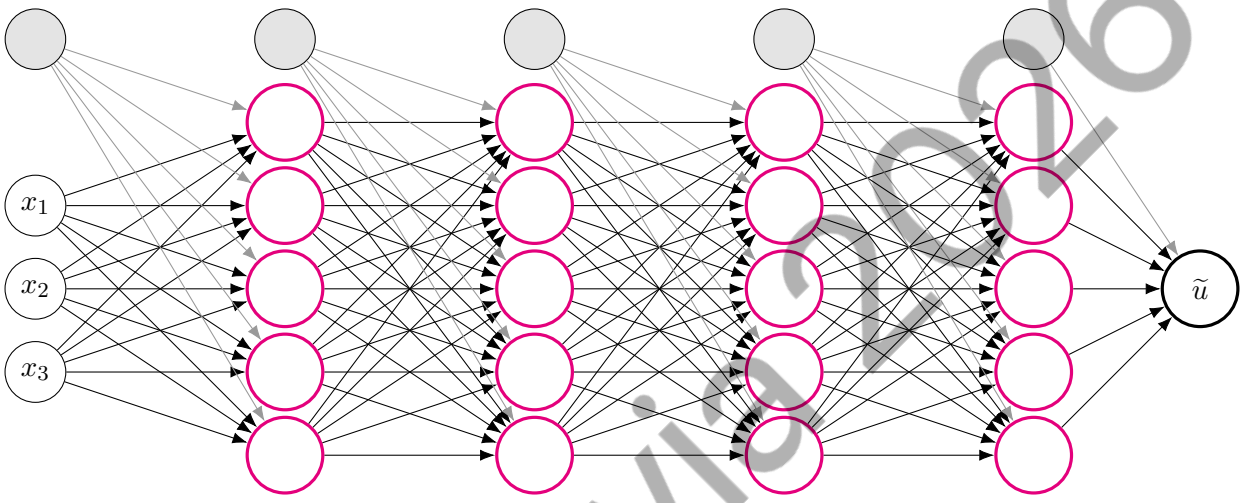


Figura 2.24 Un ejemplo de una red neuronal con cuatro capas ocultas de cinco neuronas cada una, ilustrando la estructura general que emplearemos desde ahora.

Vista previa 2026



Figura 2.25 La idea es simple: abrigar bien a un bebé no significa llenarlo de capas; aunque varias capas lo protegen del frío, el exceso puede limitar su movilidad sin aportar beneficios adicionales. De forma análoga, en redes neuronales, añadir más capas no garantiza un mejor desempeño.

A medida que avanzamos, surgen nuevas preguntas: ¿cómo determinamos cuántas neuronas utilizar?, ¿cuántas capas debe tener la red? En otras palabras, ¿cómo se elige la arquitectura adecuada? Lamentablemente, no existe una respuesta única ni una regla universal para ello. La elección dependerá en gran medida del problema que se desea resolver y de la información previa disponible acerca de su naturaleza.

Lo que sí podemos afirmar desde ahora es que una arquitectura más compleja no garantiza necesariamente una mejor aproximación. Más aún, incrementar el número de capas o de neuronas implica aumentar la cantidad de parámetros (pesos y sesgos) que deben determinarse, lo que conlleva un mayor costo computacional. Además, hasta este punto aún no hemos explicado el procedimiento mediante el cual se calculan dichos parámetros.

Antes de abordar el problema de encontrar los pesos y sesgos, es necesario considerar otro aspecto clave del diseño de la red: la elección de la función de activación. Existen diversas alternativas, cada una con propiedades particulares que influyen en la capacidad de aproximación del modelo. En la siguiente sección analizaremos estas opciones, las cuales se suman a la cuestión central de cómo decidir si la arquitectura seleccionada es la más adecuada para el problema en estudio.

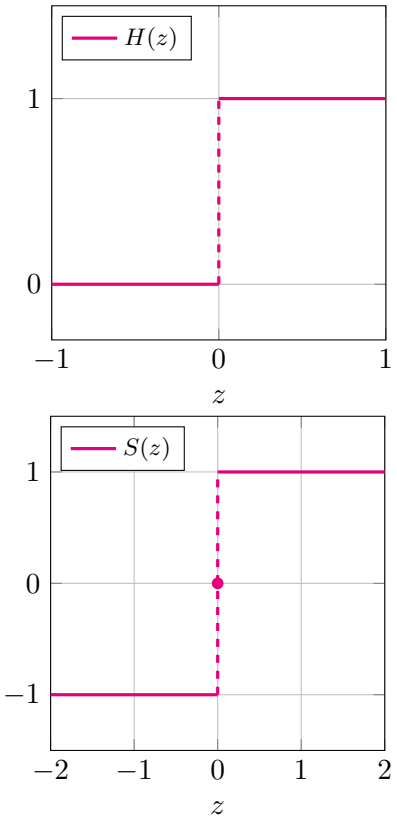


Figura 2.26 Gráfica de la función signo empleada en problemas de clasificación.

2.4 FUNCIONES DE ACTIVACIÓN

Las funciones de activación desempeñan un papel fundamental en el modelado de redes neuronales multicapa, ya que introducen la no linealidad necesaria para que la red pueda aproximar relaciones complejas. De manera general, pueden distinguirse dos grandes categorías: funciones empleadas en problemas de clasificación y funciones utilizadas en tareas de interpolación o aproximación continua.

2.4.1 Funciones de activación discontinuas

En el contexto de clasificación, suelen utilizarse funciones cuya salida pertenece a un conjunto finito de valores. Entre las más representativas se encuentran la función escalón y la función signo. La función escalón o Heaviside definida en (2.2), que hemos considerado en secciones anteriores, se emplea principalmente en modelos teóricos debido a su simplicidad conceptual.

Por su parte, la *función de activación signo* devuelve el signo del valor evaluado, produciendo como salida el conjunto $\{-1, 0, 1\}$. Esta función se utiliza en problemas de clasificación binaria y en procesos de cuantización. La ecuación (2.11) se presenta su expresión matemática.

$$S(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0. \end{cases} \tag{2.11}$$

Por otro lado, su representación gráfica se muestra en la Figura 2.26.

2.4.2 Funciones de activación continuas

Para tareas de interpolación o aproximación de funciones continuas, se prefieren funciones de activación que sean continuas y fáciles de evaluar, ya que esto facilita el proceso de obtención de los pesos y sesgos, como veremos en las siguientes secciones. A continuación, se presenta una breve introducción a algunas de las funciones más utilizadas en este contexto.

La *función de activación ReLU* (*Rectified Linear Unit*), mostrada en la Figura 2.28, se define como

$$\text{ReLU}(z) = \text{máx}(0, z). \tag{2.12}$$

Es una de las funciones más populares debido a su bajo costo computacional y a que evita el problema de saturación en la región $z > 0$, donde su derivada es constante. No obstante, presenta el inconveniente de las llamadas *neuronas muertas*, es decir, neuronas que dejan de activarse para cualquier entrada porque producen siempre un valor constante (cero). En tal caso, la neurona deja de contribuir al modelo.

Notas

- Clasificación.
- Interpolación.

Figura 2.27 Tipos de funciones de activación.

Vista previa 2026

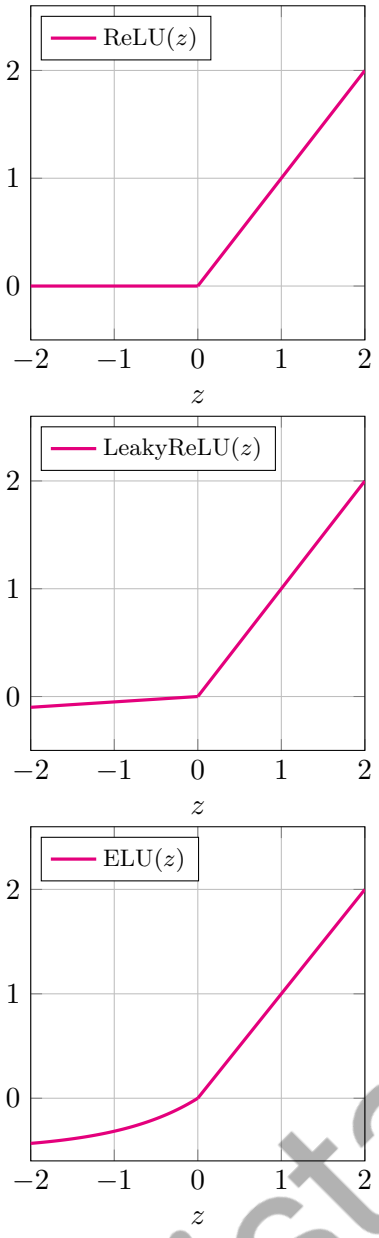


Figura 2.28 Gráfica de las funciones de activación ReLU, Leaky ReLU y ELU con $\alpha = 0.5$.

Para mitigar el problema de neuronas muertas de la función ReLU se introdujo la *función de activación Leaky ReLU*, cuya gráfica se muestra en la Figura 2.28. Esta se define como

$$\text{LeakyReLU}(z) = \begin{cases} \alpha z, & z < 0, \\ z, & z \geq 0, \end{cases} \tag{2.13}$$

donde $\alpha > 0$ es un parámetro pequeño, típicamente $\alpha = 0.01$. Esta modificación permite que los valores negativos no se anulen completamente, reduciendo así la probabilidad de generar neuronas muertas. Cabe señalar que el parámetro α puede considerarse un *hiperparámetro*. En algunas variantes incluso puede tratarse como un parámetro a calcular junto con los pesos y sesgos.

Otra opción es la *función de activación ELU* (*Exponential Linear Unit*), cuya expresión se presenta en la ecuación (2.14) y cuya representación gráfica se muestra en la Figura 2.28.

$$\text{ELU}(z) = \begin{cases} \alpha\left(\exp(z) - 1\right), & z < 0, \\ z, & z \geq 0. \end{cases} \tag{2.14}$$

La función ELU suaviza la región correspondiente a valores negativos, produciendo gradientes no nulos cuando $z < 0$ y contribuyendo así a un entrenamiento más estable. No obstante, su evaluación es ligeramente más costosa que la de la ReLU debido al cálculo de la función exponencial.

2.4.3 Función de activación diferenciable: sigmoide

Por otra parte, cuando se busca que la función sea diferenciable y además su imagen sea acotada, una de las opciones más utilizadas es la *función de activación sigmoide logística*. Esta se define mediante la ecuación

$$\sigma(z) = \frac{1}{1 + \exp(-z)}. \tag{2.15}$$

Su imagen es el intervalo $(0, 1)$, lo que la hace especialmente adecuada para problemas de clasificación binaria, donde la salida puede interpretarse como una probabilidad.

La representación gráfica de la función sigmoide logística se muestra en la Figura 2.29. Esta función pertenece a la familia de las funciones sigmoides, ya que satisface que su límite cuando $x \to -\infty$ es 0, mientras que cuando $x \to \infty$ es 1. Notar que esta función puede interpretarse como una versión diferenciable de la función escalón. Sin embargo, a diferencia de esta última, la transición es suave y la función se encuentra centrada en $z = 0.5$.

Además, la sigmoide logística satisface la ecuación diferencial

$$\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z)), \tag{2.16}$$

Vista previa 2026

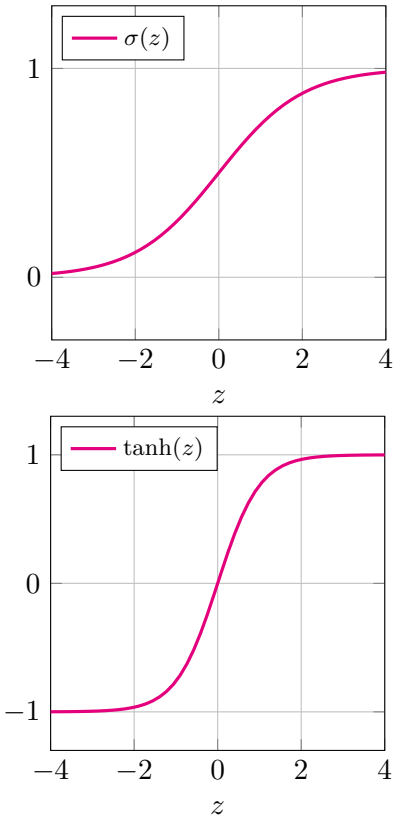


Figura 2.29 Gráfica de las funciones de activación sigmoide logística y tangente hiperbólica.

lo que implica que su derivada puede expresarse como un polinomio de grado dos en términos de la propia función. En consecuencia, derivadas de orden superior pueden obtenerse aplicando la regla de la cadena y reescribiéndose nuevamente como polinomios en σ. Esta estructura cerrada simplifica considerablemente el análisis algebraico del modelo y el cálculo eficiente de gradientes.

No obstante, esta función presenta el fenómeno de saturación cuando $|z|$ es grande, ya que su derivada tiende a cero en los extremos, lo que puede provocar el problema del *desvanecimiento del gradiente*. Este problema ocurre durante el entrenamiento de una red neuronal, cuando los gradientes utilizados para actualizar los pesos se vuelven progresivamente más pequeños a medida que se propagan desde las últimas capas hacia las primeras. Como consecuencia, los pesos de las primeras capas apenas se actualizan.

2.4.4 Función de activación diferenciable: tanh

Una de las funciones de activación más utilizadas es la *tangente hiperbólica*, definida por

$$\phi(z) = \tanh(z).$$

Su imagen es el intervalo $(-1, 1)$ y está centrada en cero. La Figura 2.29 presenta su forma característica con una transición suave entre valores asintóticos cuando $|z|$ es grande, así como su simetría respecto al origen.

De manera similar a la sigmoide logística, la función tangente hiperbólica satisface la ecuación diferencial:

$$\frac{d\phi}{dz} = 1 - \phi^2(z), \tag{2.17}$$

de modo que su derivada también puede expresarse como un polinomio en ϕ. Más aún, las derivadas de orden superior pueden escribirse como combinaciones polinomiales de ϕ.

Esta propiedad es particularmente relevante en el contexto de la solución de ecuaciones diferenciales con redes neuronales, donde se requieren derivadas de orden superior de la función aproximante. La posibilidad de reescribir dichas derivadas únicamente en términos de la propia activación conduce a expresiones algebraicamente estructuradas y computacionalmente eficientes.

Finalmente, estamos en condiciones de describir el procedimiento para la estimación de los parámetros del modelo, es decir, los pesos y los sesgos. Si bien existe libertad para elegir la complejidad de la red neuronal (número de capas, cantidad de neuronas por capa y tipo de función de activación), dicha elección influye directamente en la dificultad asociada al cálculo y ajuste de estos parámetros. En particular, arquitecturas más profundas o con mayor número de neuronas implican un incremento significativo en el número de parámetros a determinar.

Vista previa 2026



Figura 2.30 Existe una controversia sobre la autoría del método de mínimos cuadrados: Adrien-Marie Legendre lo publicó en 1805, mientras que Carl Friedrich Gauss sostuvo haberlo utilizado antes de su publicación en 1809. Retrato de Gauss (1777-1855).

2.5 HACIA EL CÁLCULO DE LOS PESOS Y SESGOS

Antes de estudiar en detalle los componentes fundamentales del cálculo de los pesos y sesgos de una red neuronal en el caso general, conviene recordar que el problema que estamos planteando no es completamente nuevo. En realidad, la idea de proponer una función con parámetros desconocidos que intenten modelar un fenómeno a partir de datos es algo que se ha estudiado desde hace mucho tiempo en el análisis matemático y en la estadística.

Este enfoque es precisamente el fundamento del método de regresión lineal usando *mínimos cuadrados*. En esencia, se propone un modelo paramétrico y se determinan sus parámetros de manera que el error entre las predicciones del modelo y los datos observados sea lo más pequeño posible.

2.5.1 Modelo de regresión lineal

Desde la perspectiva de redes neuronales, el modelo de regresión lineal puede interpretarse como una red con una sola entrada y una única neurona, en la cual la salida es simplemente una combinación lineal de la entrada, sin aplicar ninguna función de activación no lineal. Es decir, se trata de una suma ponderada básica.

EL PROBLEMA DE REGRESIÓN LINEAL

Consideremos uno de los ejemplos más sencillos: el ajuste de una función lineal

$$\widetilde{u} = wx + b,$$

donde w representa la pendiente y b la ordenada al origen. Supongamos ahora que se dispone de un conjunto de datos

$$\{(\hat{x}_i, \hat{u}_i)\}_{i=1}^{N},$$

donde cada $\hat{x}_i$ es un valor conocido de la variable independiente y $\hat{u}_i$ es el valor observado correspondiente. El problema consiste entonces en determinar los parámetros w y b de manera que la recta propuesta describa lo mejor posible dichos datos.

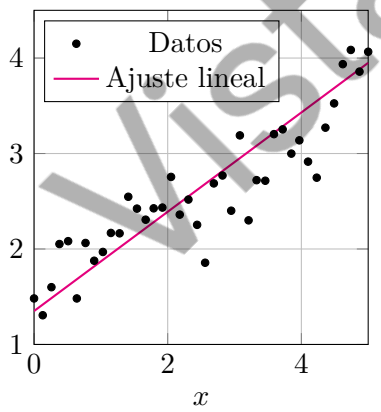


Figura 2.31 Ejemplo de ajuste lineal a un conjunto de datos mediante el método de mínimos cuadrados.

En el método de mínimos cuadrados, el ajuste de los datos se logra definiendo una función de error que mide la discrepancia entre las predicciones del modelo y los valores observados, y buscando los valores de w y b que minimicen dicha función. Este procedimiento es, conceptualmente, el mismo que utilizaremos en el entrenamiento de una red neuronal: proponer un modelo paramétrico $\widetilde{u}(x; w, b)$, definir una medida de error y ajustar los parámetros para reducirlo.

La diferencia fundamental radica en que, mientras en el caso lineal el problema puede resolverse explícitamente mediante fórmulas cerradas,

Vista previa 2026

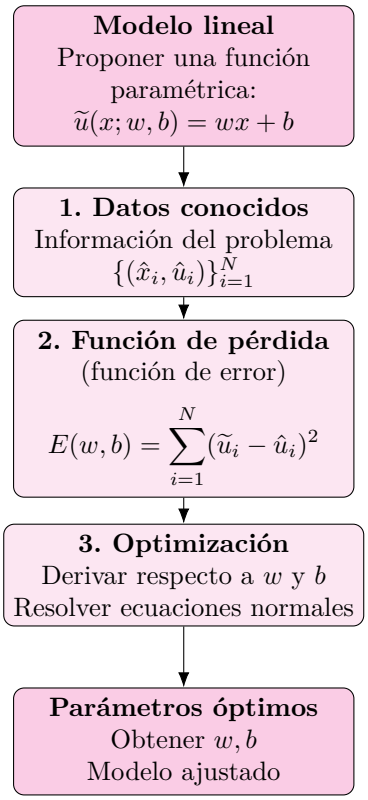


Figura 2.32 Esquema general del proceso para obtener los parámetros de un modelo lineal por medio de la metodología de mínimos cuadrados.

en el caso de redes neuronales más complejas el número de parámetros crece considerablemente y ya no es posible obtener soluciones analíticas simples. Por ello, se recurre a métodos iterativos de optimización, como el descenso de gradiente, que generalizan la filosofía de los mínimos cuadrados al caso no lineal.

2.5.2 El método de mínimos cuadrados

El método de mínimos cuadrados emplea el *error cuadrático* como medida del error entre las predicciones del modelo y los datos observados. En el caso del modelo lineal la función de pérdida se define como

$$E(w,b) = \sum_{i=1}^{N} (\widetilde{u}_i - \hat{u}_i)^2 = \sum_{i=1}^{N} (w\hat{x}_i + b - \hat{u}_i)^2 . \tag{2.18}$$

Esta función presenta dos ventajas fundamentales en el proceso de minimización: es diferenciable respecto a los parámetros y admite una solución analítica única.

Entonces, el mínimo de la función (2.18) se obtiene imponiendo que sus derivadas parciales respecto a los parámetros se anulen. Esto conduce a las llamadas *ecuaciones normales*:

$$\frac{\partial E}{\partial b} = -2\sum_{i=1}^{N}(\widetilde{u}_i - w\hat{x}_i - b) = 0, \quad \frac{\partial E}{\partial w} = -2\sum_{i=1}^{N}\hat{x}_i(\widetilde{u}_i - w\hat{x}_i - b) = 0.$$

La resolución de este sistema lineal de dos ecuaciones con dos incógnitas permite determinar explícitamente los valores óptimos w y b, es decir, los parámetros que minimizan el error cuadrático total.

Finalmente, observemos que en el problema de determinar los parámetros de este modelo intervienen tres elementos fundamentales:

- **Una función de error**, que mide qué tan bien el modelo se ajusta a los datos.
- **Un método de optimización**, que indica cómo obtener los parámetros para reducir dicha función de error.
- **El gradiente de la función de error**, que proporciona la dirección en la cual los parámetros deben ajustarse para disminuir el error.

En el caso de la regresión lineal simple, este proceso es relativamente directo lo que incluso permite obtener una solución analítica cerrada. Sin embargo, en el caso general de una red neuronal artificial, el modelo incorpora funciones de activación que introducen no linealidad. Esto provoca que la relación entre los parámetros y la salida sea más compleja y, por lo tanto, el cálculo del gradiente requiera un procedimiento sistemático, como el algoritmo de *retropropagación* (*backpropagation*).

Vista previa 2026

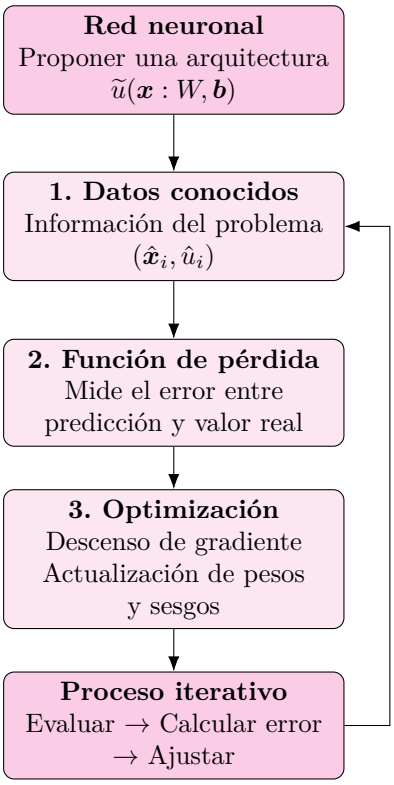


Figura 2.33 Esquema general del proceso de entrenamiento de una red neuronal.



Figura 2.34 ¡Cuidado! El proceso de entrenamiento puede requerir una cantidad considerable de tiempo de procesamiento.

2.6 ENTRENAMIENTO

El entrenamiento de una red neuronal puede interpretarse como una extensión natural del método de mínimos cuadrados: seguimos definiendo una función de error y buscamos minimizarla, pero ahora el modelo es más sofisticado y con nombres ligeramente diferentes, además el cálculo del gradiente depende explícitamente de su arquitectura. Esta analogía nos permitirá comprender el proceso de entrenamiento de redes neuronales como una generalización de ideas que ya conocemos.

Una vez definida la arquitectura de la red neuronal, surge la pregunta fundamental: ¿cómo se determinan los valores adecuados de los pesos y los sesgos? Recordemos que estos parámetros son los responsables de que la red produzca salidas correctas o, al menos, lo más cercanas posible a los valores esperados.

El proceso mediante el cual se calculan los parámetros recibe el nombre de *entrenamiento*. Este procedimiento se apoya en varios componentes fundamentales que deben definirse de manera coherente.

- **Datos conocidos**. En primer lugar, es necesario contar con información del problema en forma de datos. Estos datos representan ejemplos que la red debe aprender a reproducir.

- **Función de pérdida**. En segundo término, se define una función de pérdida (también llamada función de costo). Esta función mide qué tan grande es el error cometido por la red al comparar sus predicciones con los valores reales. Si el error es grande, la función de costo toma valores altos; si el ajuste es bueno, su valor disminuye. De esta manera, la función de costo actúa como una guía que indica qué tan bien está aprendiendo el modelo.

- **Optimización**. El siguiente paso es elegir un método que permita reducir ese error. Para ello se emplean técnicas de optimización, siendo el descenso de gradiente una de las más utilizadas. La idea básica es sencilla: calcular cómo cambia el error cuando se modifican ligeramente los pesos y sesgos, y luego ajustar estos parámetros en la dirección que más disminuya la función de costo.

Finalmente, el entrenamiento consiste de un proceso iterativo: evaluar la red, calcular el error, ajustar los parámetros y volver a evaluar. Se espera que con suficientes iteraciones, los pesos y sesgos se van refinando progresivamente hasta que la red logra un desempeño satisfactorio.

A continuación, se describirán cada una de estas etapas fundamentales del proceso de entrenamiento, resaltando sus aspectos más relevantes sin profundizar exhaustivamente en los detalles técnicos. Esta sección concluirá con un ejemplo ilustrativo en el que se muestra el carácter iterativo del entrenamiento de la red neuronal.

Vista previa 2026



Figura 2.35 Por lo general, se dispone únicamente de una cantidad limitada de datos conocidos. Esto se debe a que, en muchas problemas físicos la recopilación de información puede ser compleja, costosa o difícil de realizar.

2.6.1 Los datos conocidos

El primer paso consiste en contar con información del problema en forma de datos. Estos datos pueden provenir de mediciones experimentales, simulaciones numéricas o condiciones impuestas por el propio modelo matemático, y representan el conocimiento disponible que la red deberá aprender a reproducir o aproximar.

Con el fin de ejemplificar las definiciones relacionadas con el proceso de entrenamiento, introducimos el siguiente concepto.

DEFINICIÓN

Sea $\boldsymbol{x}$ el vector de variables independientes del problema y supóngase que se dispone de un conjunto de N observaciones de la forma

$$\{(\hat{\boldsymbol{x}}_1, \hat{u}_1), (\hat{\boldsymbol{x}}_2, \hat{u}_2), \ldots, (\hat{\boldsymbol{x}}_N, \hat{u}_N)\}\,, \tag{2.19}$$

donde cada $\boldsymbol{x}_i$ representa una entrada conocida y $\hat{u}_i$ el valor observado o deseado asociado a dicha entrada.

Estas observaciones constituyen el conjunto de datos con el cual se entrenará la red neuronal, es decir, la información que el modelo debe aprender a reproducir mediante el ajuste adecuado de sus pesos y sesgos. Por ejemplo, en un problema de clasificación los datos pueden estar formados por las entradas acompañadas de su etiqueta correspondiente. Así en el problema XOR, los cuatro datos proporcionados constituyen la única información a partir de la cual se ajustarán los pesos y sesgos.

Datos del problema XOR

En el problema XOR, los datos conocidos están dados por el conjunto

$$\{(0,0,0),\ (0,1,1),\ (1,0,1),\ (1,1,0)\},$$

donde los dos primeros valores de cada terna representan las entradas y el tercer valor corresponde a la salida deseada.

En este caso, la red neuronal debe aprender a asociar correctamente cada par de entradas binarias con su resultado esperado, reproduciendo el comportamiento lógico de la operación XOR.

Figura 2.36 Datos conocidos en el problema XOR.

2.6.2 La función de pérdida

Existen diversas opciones de *función de pérdida* o *función de costo* que pueden emplearse para medir el error en el proceso de minimización. La elección de una u otra depende de las características específicas del problema y de la naturaleza de los errores, particularmente si ciertos tipos de desviaciones resultan más perjudiciales que otros. Para concretar estas ideas, consideremos el caso ya utilizado como función de pérdida en el problema de regresión lineal: la función de errores cuadrático.

DEFINICIÓN

El *error cuadrático medio* (*Mean Squared Error*, MSE) se define mediante la siguiente función de costo:

$$E(W, \boldsymbol{b}) = \frac{1}{N}\sum_{i=1}^{N} (\tilde{u}_i - \hat{u}_i)^2\,. \tag{2.20}$$

Aquí, $\tilde{u}_i = \tilde{u}(\hat{\boldsymbol{x}}_i; W, \boldsymbol{b})$ representa la salida de la red neuronal evaluada en los mismos puntos donde se conocen los datos $\hat{u}_i$.

Vista previa 2026

Notas

$$E = \frac{1}{N}\sum_{i=1}^{N}(\widetilde{u}_i - \hat{u}_i)^2\,,$$

$$E_1 = \frac{1}{N}\sum_{i=1}^{N}|\widetilde{u}_i - \hat{u}_i|$$

Figura 2.37 Error cuadrático medio (MSE) y error abosluto medio (MAE).

Obsérvese que, a diferencia del caso clásico de regresión lineal, la ecuación (2.20) divide la suma de los errores entre el número total de datos. Esta normalización permite obtener una medida promedio de la discrepancia, independiente del tamaño de la muestra.

Ventajas del error cuadrático medio

El error cuadrático medio es una de las funciones de pérdida más utilizadas, debido a varias propiedades que la hacen especialmente conveniente en problemas de optimización:

- En primer lugar, es una función diferenciable respecto a los parámetros del modelo, lo que permite aplicar de manera directa métodos de optimización basados en el gradiente.
- Además, posee una característica fundamental: penaliza con mayor severidad los errores grandes que los pequeños. Para entender esto un poco más, definamos el residuo como $r_i = \widetilde{u}_i - \hat{u}_i$, entonces se cumple que

  $$r^2 \leq |r| \text{ cuando } |r| \leq 1, \qquad \text{y} \qquad r^2 > |r| \text{ cuando } |r| > 1.$$

  Esto significa que, para errores pequeños ($|r| \leq 1$), el valor cuadrático es incluso menor que el error absoluto, mientras que para errores grandes ($|r| > 1$), el término cuadrático crece más rápidamente. En consecuencia, los errores de gran magnitud se amplifican y pueden dominar la función de pérdida.

Como consecuencia, el error cuadrático medio tiende a estar fuertemente influenciado por *valores atípicos* (*outliers*). Sin embargo, esta misma propiedad lo hace especialmente adecuado cuando se asume que el ruido en los datos sigue una distribución Gaussiana, caso en el cual la norma L_2 (misma definición que MSE) surge de manera natural desde un punto de vista probabilístico.

Otras funciones de pérdida

Por otro lado, si se desea asignar un peso más uniforme a todas las discrepancias, puede emplearse la función de costo del error absoluto medio (*Mean Absolute Error*, MAE), la cual no amplifica de forma tan pronunciada los errores grandes. Sin embargo, presenta la dificultad de no ser diferenciable en el punto donde el error se anula, lo que requiere técnicas específicas de optimización convexa.

Asimismo, existen funciones de pérdida robustas que combinan ventajas de distintos enfoques mediante definiciones por tramos, como las funciones de Huber y de Tukey [59]. Sin embargo, la función de pérdida basada en el error cuadrático medio definida en (2.20) será la que emplearemos a lo largo de este libro, salvo indicación explícita de lo contrario. No obstante, reiteraremos su definición cada vez que se introduzca un nuevo problema, con el fin de mantener claridad y precisión en la exposición.

Funciones de pérdida robustas que combinan ventajas de distintos enfoques mediante definiciones por tramos:

- Función de Huber

$$J(r) = \begin{cases} \frac{1}{2}r^2, & |r| \leq \delta, \\ \delta\left(|r_i| - \frac{\delta}{2}\right), & |r| > \delta. \end{cases}$$

- Función de Tukey

$$J(r) = \begin{cases} \frac{c^2}{6}\left[1 - \left(1 - \left(\frac{r}{c}\right)^2\right)^3\right], & |r| \leq c, \\ \frac{c^2}{6}, & |r| > c. \end{cases}$$

Aquí $\delta > 0$ es un umbral que se interpreta como el radio de separación, c es una tolerancia a los valores atípicos, y r representa cada residuo definido por

$$r_i = \widetilde{u}_i - \hat{u}_i.$$

Figura 2.38 Otras funciones de pérdida.

Vista previa 2026

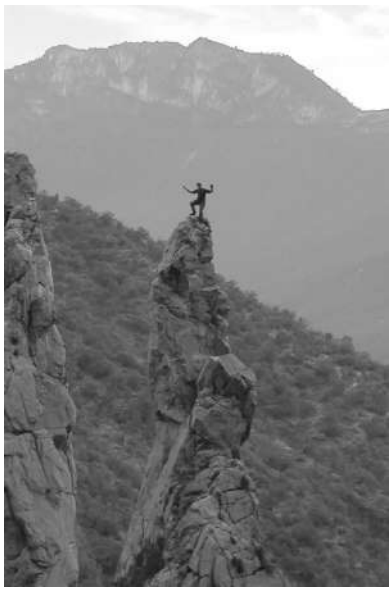


Figura 2.39 ¿Cómo descender ahora? La función de pérdida puede visualizarse como un paisaje montañoso: la posición actual representa los pesos del modelo y su altura el valor de la pérdida. El objetivo es encontrar el valle más profundo, es decir, el mínimo de la función.

2.6.3 El método de optimización

Continuando con el proceso de entrenamiento, el siguiente paso es definir cómo se ajustarán los parametros del modelo. Para entrenar una red buscamos minimizar la función de perdida con respecto a los pesos y sesgos a partir de los datos conocidos. Para ello, utilizaremos un método de optimización iterativo: en cada iteración, los pesos se actualizan ligeramente en la dirección que más reduce el error, siguiendo el principio del *descenso de gradiente*. Además, en este proceso es donde entra en juego la *tasa de aprendizaje*, la cual controla que tan rápido se ajusta la red a los datos.

Un ejemplo de una red neuronal sencilla

Para comprender con claridad el método de descenso de gradiente, ejemplificaremos el descenso de gradiente utilizando una red neuronal sencilla como la presentada en la Figura 2.40. El objetivo es identificar y analizar sus componentes esenciales, poniendo especial atención en el papel fundamental que desempeña el gradiente en la actualización de los parámetros.

EJEMPLO DE UNA RED SENCILLA

Consideremos un perceptrón de una sola entrada, sin función de activación; es decir, únicamente la suma ponderada (como en regresión lineal), dada por

$$\widetilde{u}(x; w, b) = wx + b. \tag{2.21}$$

El objetivo es encontrar los parámetros w y b que minimizan la función de pérdida E, definida como el error cuadrático medio.

Para comprender por qué el descenso de gradiente funciona, analizaremos el comportamiento local de la función de pérdida E utilizando herramientas básicas de cálculo en varias variables. Esta breve discusión permite entender la lógica matemática que justifica la regla de actualización de los parámetros.

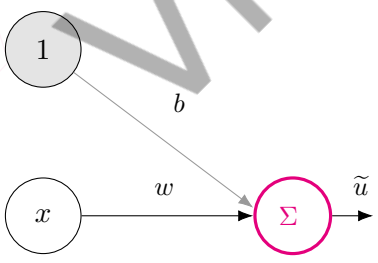


Figura 2.40 Red neuronal de una capa y una neurona para ejemplificar el método de descenso de gradiente.

Deducción del algoritmo de descenso de gradiente

Si el lector no está familiarizado con el cálculo multivariable, puede omitir esta explicación sin inconveniente y pasar directamente a la definición del algoritmo, donde se describe el procedimiento de manera operativa e intuitiva.

El objetivo es minimizar $E(\theta)$, definida como el error cuadrático medio proporcionada por la ecuación (2.20), donde $\theta = (w, b)^\top$ es el vector que contiene los dos parámetros del modelo propuesto.

Vista previa 2026

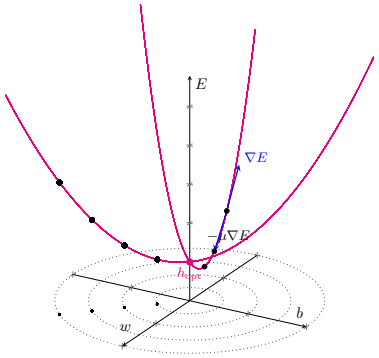


Figura 2.41 Bosquejo de la dirección de máximo descenso en un paraboloide.

Para empezar, si nos encontramos en un punto θ y realizamos un pequeño desplazamiento $\Delta\theta$, podemos aproximar el valor de la función E en el nuevo punto

$$\theta' = \theta + \Delta\theta, \tag{2.22}$$

mediante una expansión de Taylor de primer orden de la forma:

$$E(\theta') = E(\theta + \Delta\theta) \approx E(\theta) + \nabla E(\theta)^\top \Delta\theta.$$

Esta expresión nos dice que, para desplazamientos pequeños, el cambio en la función depende esencialmente del producto punto entre el gradiente $\nabla E(\theta)$ y el vector de desplazamiento $\Delta\theta$. Observemos ahora que, dado que el error cuadratico medio siempre es positivo , entonces:

- $E(\theta')$ es mayor que $E(\theta)$ si $\nabla E(\theta)^\top \Delta\theta > 0$, es decir, el error aumenta,
- $E(\theta')$ es menor que $E(\theta)$ si $\nabla E(\theta)^\top \Delta\theta < 0$, es decir, el error dismninuye.

Por lo tanto, si queremos reducir el valor de $E(\theta')$, debemos elegir $\Delta\theta$ de modo que

$$\nabla E(\theta)^\top \Delta\theta < 0.$$

Por otro lado, desde el cálculo multivariable sabemos que el gradiente $\nabla E(\theta)$ apunta en la dirección de máximo crecimiento de la función. En consecuencia, la dirección de máximo descenso es justamente la dirección opuesta, es decir, $-\nabla E(\theta)$. Para controlar el tamaño del paso que damos en esa dirección, introducimos un parámetro positivo $\eta > 0$, llamado *tasa de aprendizaje*. Así, elegimos

$$\Delta\theta = -\eta \nabla E(\theta). \tag{2.23}$$

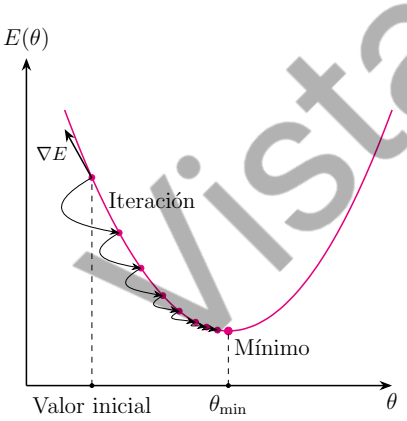


Figura 2.42 En el descenso de gradiente, en cada iteración calculamos el gradiente y nos movemos un pequeño paso en la dirección contraria, reduciendo así el valor de la función de pérdida.

Entonces, tenemos que

$$\nabla E(\theta)^\top \Delta\theta = -\eta \|\nabla E(\theta)\|^2 < 0, \tag{2.24}$$

Finalmente, sustituyendo este desplazamiento en la actualización de los parámetros (2.22), obtenemos la regla fundamental del descenso de gradiente:

$$\theta' = \theta - \eta \nabla E(\theta). \tag{2.25}$$

En la Figura 2.42 se muestra un diagrama que ilustra de manera visual la regla de actualización del descenso de gradiente. La idea geométrica es sencilla: el gradiente indica la dirección de mayor crecimiento de la función, por lo que al movernos en la dirección contraria descendemos hacia un mínimo.

Vista previa 2026

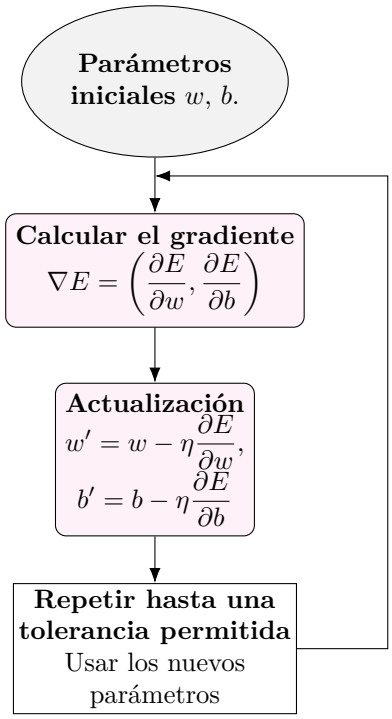


Figura 2.43 Esquema general del algoritmo de descenso de gradiente.



Figura 2.44 Las iteraciones en el proceso de aprendizaje se les denómina épocas.

El algoritmo de descenso de gradiente

Por lo tanto, en el método de descenso de gradiente, en cada iteración calculamos el gradiente y nos movemos un pequeño paso en la dirección contraria, reduciendo así progresivamente el valor de la función de pérdida E.

ALGORITMO

El algoritmo de descenso de gradiente para obtener los parámetros w y b correspondientes a una red neuronal de una sola entrada y la suma ponderada como salida está dada por:

$$w' = w - \eta \frac{\partial E}{\partial w}, \tag{2.26}$$

$$b' = b - \eta \frac{\partial E}{\partial b}, \tag{2.27}$$

donde (w, b) y (w', b') indican los valores anteriores y los actualizados de los parámetros del modelo y $\eta > 0$ es un *hiperparámetro* positivo llamado *tasa de aprendizaje*.

Es importante notar que el algoritmo de descenso de gradiente es un método iterativo; es decir, cada nuevo valor depende de los valores previamente calculados. Por ello, es necesario partir de valores iniciales w y b, los cuales deben ser elegidos o propuestos para dar inicio al proceso iterativo. En la Figura 2.42 se bosqueja este algoritmo.

Asimismo, es fundamental establecer un criterio de paro para el algoritmo; por ejemplo, una tolerancia y/o un número máximo de iteraciones. Estos elementos, junto con la tasa de aprendizaje, constituyen los hiperparámetros estrechamente vinculados al método de optimización seleccionado. Por cierto, en el contexto de redes neuronales artificiales, es común referirse a estas iteraciones como épocas.

DEFINICIÓN

Se conoce como *épocas* a las iteraciones que se realizan en el entrenamiento (proceso de optimización).

La tasa de aprendizaje y el número de épocas

La tasa de aprendizaje η controla el tamaño del paso que damos en la dirección de menor crecimiento. Intuitivamente, su función es regular cuánto avanzamos en cada iteración. Si el paso es demasiado grande, podemos sobrepasar el mínimo y comenzar a oscilar alrededor de él, dificultando la convergencia. En cambio, si η es pequeño, los pasos serán más cortos, lo que favorece una convergencia más estable, aunque potencialmente más lenta.

Vista previa 2026

Notas

El algoritmo Adam es utilizado en los códigos para optimizar en los ejemplos presentados en este libro. No es un acrónimo y no se escribe como "ADAM" [3].

Figura 2.45 Método de optimición.

Un par de observaciones sobre la tasa de aprendizaje y la convergencia del algoritmo son:

- En la práctica, un valor común inicial es $\eta = 0.01$. Sin embargo, este parámetro no es universal: debe ajustarse observando el comportamiento del entrenamiento.
- Si el error oscila o diverge, conviene disminuir η; si la convergencia es demasiado lenta, puede intentarse un valor ligeramente mayor.

Dado que se toman pasos pequeños, el entrenamiento consta de miles de iteraciones. Sobre estas existen métodos que en base al error de las iteraciones van reduciendo la tasa de aprendizaje utilizada. Estos permiten utilizar pasos mayores cuando la red empieza a aprender y tener pasos pequeños para cuando se alcanza una aproximación cercana al mínimo.

Decenso de Gradiente Estocásticos (SGD)

Una red neuronal profunda puede llegar a tener millones de parámetros. En este contexto, el método clásico de descenso de gradiente se vuelve computacionalmente muy costoso, ya que cada actualización requiere recorrer todo el conjunto de entrenamiento. Para superar esta limitación se han desarrollado los métodos de *descenso de gradiente estocástico* (*Stochastic Gradient Descent*, SGD).



Figura 2.46 Para comprender con claridad el concepto de retropropagación y su papel en las iteraciones del optimizador, es conveniente detenernos y realizar los cálculos manualmente, con calma y paso a paso.

La idea fundamental del descenso de gradiente estocástico es aproximar el gradiente utilizando únicamente una pequeña parte de los datos en cada iteración. De esta manera, se obtiene una estimación con menor costo computacional, pero suficientemente informativa para continuar descendiendo en promedio hacia el mínimo.

Aunque cada paso de esta estrategia es más ruidoso que en el caso determinista, se ha demostrado que, bajo condiciones adecuadas, estos métodos también convergen hacia un mínimo de la función de pérdida. Uno de los métodos más utilizados y efectivos dentro de esta familia es *Adam* (es el nombre del algoritmo derivado de *adaptive moment estimation*) [3], que combina ideas de momento y escalamiento adaptativo del gradiente para lograr una convergencia más estable y rápida en la práctica. En este libro emplearemos esta técnica en nuestros códigos.

Cálculo del gradiente

Finalmente, la dirección en la que deben modificarse los pesos (el gradiente) se calcula de manera eficiente mediante el algoritmo de retropropagación (*backpropagation*), que aprovecha la estructura interna de la red para propagar el error desde la salida hacia las capas anteriores. Gracias a esta combinación, es posible entrenar el modelo de forma eficiente, incluso cuando se dispone de una gran cantidad de datos. En la siguiente sección profundizaremos más en este tema.

Vista previa 2026



Figura 2.47 Retropropagación (*backpropagation*) se usa para determinar el valor de las derivadas en el recorrido que toma un valor de entrada.

2.7 ITERACIONES CON RETROPROPAGACIÓN

A lo largo de la sección anterior se ha mostrado que entrenar una red neuronal consiste, esencialmente, en minimizar el error del modelo con respecto a los datos disponibles. Más precisamente, el proceso de optimización requiere calcular el gradiente de la función de pérdida con respecto a todos los parámetros del modelo; este cálculo es el elemento central que permite ajustar los pesos y sesgos de la red.

Una red neuronal puede adoptar arquitecturas muy diversas (múltiples capas, número de neuronas, distintas funciones de activación); por ello, se necesita un esquema general que permita calcular el gradiente independientemente de la estructura específica del modelo. El método de descenso de gradiente con *retropropagación* (*backpropagation*) cumple precisamente esta función: se adapta a cualquier arquitectura de red y permite realizar el proceso de optimización de manera sistemática y eficiente.

Desde el punto de vista teórico, la retropropagación es un procedimiento eficiente para calcular el gradiente mediante la aplicación sistemática de la regla de la cadena sobre el grafo computacional del modelo. En arquitecturas pequeñas, este mecanismo puede analizarse de forma explícita y resulta relativamente sencillo de estructurar, lo que lo convierte en un excelente punto de partida para su comprensión.

2.7.1 Un ejemplo lineal con una sola neurona y un dato

Para ilustrar el proceso iterativo del entrenamiento con claridad, comenzaremos con el caso más simple posible: una sola neurona sin función de activación, exactamente como se definió en la ecuación (2.21). A partir de este ejemplo básico, identificaremos el mecanismo iterativo mediante el cual la retropropagación ajusta los parámetros del modelo para aproximar cada vez mejor un problema con un sólo dato.

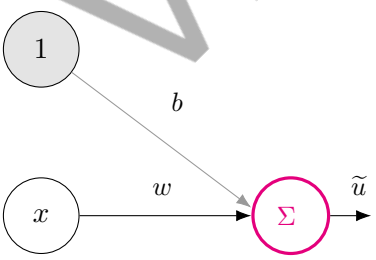


Figura 2.48 Red neuronal de una capa y una neurona para ejemplificar retropropagación.

EJEMPLO CON UN DATO

Consideremos el caso de una red neuronal con una sola entrada y sin función de activación, descrita por el modelo lineal

$$\tilde{u}(x; w, b) = wx + b, \tag{2.28}$$

y supongamos que se dispone de un único dato de entrenamiento dado por

$$(\hat{x}_1, \hat{u}_1), \tag{2.29}$$

por lo que la función de pérdida se define simplemente como el error cuadrático asociado a ese punto. En consecuencia,

$$E(w, b) = (\tilde{u}(\hat{x}_1; w, b) - \hat{u}_1)^2 . \tag{2.30}$$

Vista previa 2026

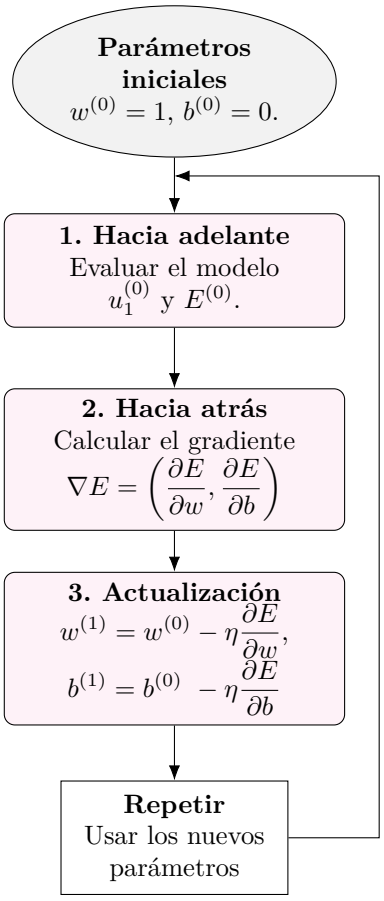


Figura 2.49 Esquema general del proceso de entrenamiento de una red neuronal.

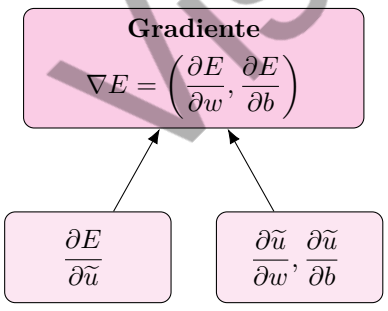


Figura 2.50 Esquema general del proceso para obtener el gradiente de la función de pérdida con dos parámetros.

De manera general, la implementación del proceso de optimización se puede dividir en los siguientes pasos: *el paso hacia adelante* (*foward phase*), *el paso hacia atrás* (*backward phase*) y la *actualización* de los parámetros.

El primer paso del proceso iterativo, al que hemos denominado *paso hacia adelante*, consiste en evaluar el modelo con los datos de entrada disponibles y, a partir de la salida obtenida, calcular el valor de la función de pérdida.

Para nuestro ejemplo (2.28), dado que los valores $\hat{x}_1$ y $\hat{u}_1$ son conocidos, es necesario fijar valores iniciales para los parámetros w y b con el fin de obtener un valor concreto de $\widetilde{u}$ y, en consecuencia, de la pérdida E. Estos valores iniciales son elegidos por el usuario (o definidos mediante algún esquema de inicialización) y su selección influye de manera importante en el comportamiento del proceso de optimización, particularmente en la velocidad y estabilidad de la convergencia. Denotaremos los valores iniciales para el peso y el sesgo como $w^{(0)}$ y $b^{(0)}$.

Siguiendo con el proceso iterativo de entrenamiento, el *paso hacia atrás* corresponde al recorrido de la red desde la salida hacia los parámetros con el objetivo de calcular el gradiente de la función de pérdida. La idea fundamental es aplicar la regla de la cadena de manera sistemática, evaluando las derivadas locales en cada nodo. Esta descomposición permite organizar y hacer más eficiente el cálculo numérico del gradiente.

En el caso del ejemplo (2.28), debido a su sencillez, el proceso de *backpropagation* se puede expresar explícitamente como

$$\nabla E = \left(\frac{\partial E}{\partial w}, \frac{\partial E}{\partial b}\right) = \left(\frac{\partial E}{\partial \widetilde{u}}\frac{\partial \widetilde{u}}{\partial w}, \frac{\partial E}{\partial \widetilde{u}}\frac{\partial \widetilde{u}}{\partial b}\right). \tag{2.31}$$

Aquí, $\frac{\partial E}{\partial \widetilde{u}}$ mide cómo la salida de la neurona influye en la pérdida; mientras que $\frac{\partial \widetilde{u}}{\partial w}$ y $\frac{\partial \widetilde{u}}{\partial b}$ cuantifican cómo el peso y sesgo afecta la salida. De esta manera, el gradiente total con respecto a cada parámetro se obtiene como el producto de derivadas intermedias. En el ejemplo considerado de un solo dato conocido, tenemos que

$$\begin{aligned}
\frac{\partial E}{\partial \widetilde{u}} &= \frac{\partial}{\partial \widetilde{u}}\left((\widetilde{u} - \hat{u}_1)^2\right) = 2(\widetilde{u} - \hat{u}_1), \\
\frac{\partial \widetilde{u}}{\partial w} &= \frac{\partial}{\partial w}(\hat{x}_1 w + b) = \hat{x}_1, \\
\frac{\partial \widetilde{u}}{\partial b} &= \frac{\partial}{\partial b}(\hat{x}_1 w + b) = 1.
\end{aligned} \tag{2.32}$$

Una vez calculado el gradiente, el último paso de cada iteración consiste en actualizar los parámetros del modelo mediante el algoritmo de descenso de gradiente, descrito en (2.26) y (2.27), utilizando una tasa de aprendizaje previamente establecida.

Vista previa 2026



Figura 2.51 Ahora sí es momento de tomar papel y lápiz para realizar los cálculos manualmente. Se recomienda intentar resolver el ejemplo por cuenta propia y, posteriormente, cotejar el resultado obtenido.

2.7.2 El ejemplo lineal con valores específicos

A continuación, se ilustrarán los pasos descritos anteriormente utilizando valores específicos tanto para el dato conocido como para los parámetros iniciales y la tasa de aprendizaje, con el propósito de mostrar explícitamente cómo se desarrolla una iteración completa del proceso de entrenamiento.

EJEMPLO CON VALORES ESPECÍFICOS

Consideremos los parámetros iniciales

$$w^{(0)} = 1, \quad b^{(0)} = 0,$$

para la red neuronal definida en (2.28). Supongamos además que se dispone de un único dato de entrenamiento

$$(\hat{x}_1, \hat{u}_1) = (2.1, 4),$$

y que el proceso de optimización se lleva a cabo mediante el método de descenso de gradiente con una tasa de aprendizaje

$$\eta = 0.01.$$

- El primer paso consiste en evaluar la salida del modelo en $\hat{x}_1$ y calcular la pérdida asociada mediante el error cuadrático.
- Posteriormente, calcular las derivadas de la función de pérdida con respecto a los parámetros w y b.
- Una vez obtenido el gradiente, actualizar los parámetros utilizando descenso de gradiente con la tasa de aprendizaje proporcionada.
- Finalmente, con los nuevos valores de w y b, evaluar nuevamente la pérdida para verificar la reducción del error.

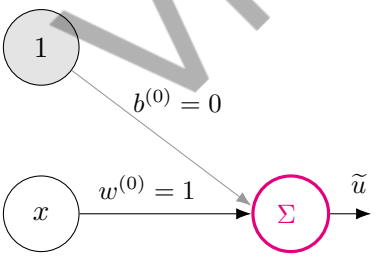


Figura 2.52 Red neuronal de una capa y una neurona con valores iniciales $w^{(0)}$ y $b^{(0)}$ para ejemplificar retropropagación.

Al sustituir los valores específicos del dato disponible, $\hat{x}_1 = 2.1$ y $\hat{u}_1 = 4$, la función de pérdida se expresa como

$$E(w, b) = \left(\widetilde{u}_1(w, b) - 4\right)^2, \tag{2.33}$$

donde

$$\widetilde{u}_1(w, b) = \widetilde{u}(\hat{x}_1; w, b) = \widetilde{u}(2.1; w, b) = 2.1\, w + b. \tag{2.34}$$

es la evaluación del modelo en $\hat{x}_1$.

En consecuencia, a partir de estas expresiones podemos calcular el gradiente con respecto a w y b, y proceder a la actualización de los parámetros mediante el esquema de descenso de gradiente de la siguiente forma.

Vista previa 2026

- Obsérvese que las derivadas parciales de $\bar{u}$ con respecto a los parámetros w y b no dependen de dichos parámetros.

$$\frac{\partial \widetilde{u}}{\partial w} = \hat{x}_1, \qquad \frac{\partial \widetilde{u}}{\partial b} = 1$$

- No obstante, esta situación no es general; se trata simplemente de una consecuencia de que el modelo considerado no incluye una función de activación.
- En arquitecturas más complejas, las derivadas suelen depender explícitamente de los propios parámetros.

Figura 2.53 Sobre el gradiente de E para un modelo lineal y un dato conocido $(\hat{x}_1, \hat{u}_1)$.

La primera época

- En el ejemplo considerado, al evaluar el modelo (2.33) con los parámetros iniciales $w^{(0)} = 1$ y $b^{(0)} = 0$, se obtiene

$$\begin{aligned} \widetilde{u}_1^{(0)} &= \widetilde{u}_1\left(w^{(0)}, b^{(0)}\right) = 2.1w^{(0)} + b^{(0)} \\ &= 2.1(1) + 0 \\ &= 2.1. \end{aligned}$$

$$\begin{aligned} E^{(0)} &= E(w^{(0)}, b^{(0)}) = \left(\widetilde{u}_1^{(0)} - 4\right)^2 \\ &= (2.1 - 4)^2 = 3.61 \end{aligned}$$

Por lo tanto tenemos que $E^{(0)} = 3.61$ con los parámetros iniciales.

- Para calcular el gradiente usamos (2.32), entonces tenemos que las derivadas locales estan dados por

$$\frac{\partial E}{\partial \widetilde{u}} = 2(2.1 - 4) = -3.8, \qquad \frac{\partial \widetilde{u}}{\partial w} = 2.1, \qquad \text{y} \qquad \frac{\partial \widetilde{u}}{\partial b} = 1.$$

Entonces

$$\begin{aligned} \frac{\partial E}{\partial w} &= \frac{\partial E}{\partial \widetilde{u}}\frac{\partial \widetilde{u}}{\partial w} = (-3.8)(2.1) = -7.98, \\ \frac{\partial E}{\partial b} &= \frac{\partial E}{\partial \widetilde{u}}\frac{\partial \widetilde{u}}{\partial b} = (-3.8)(1) = -3.8 \end{aligned}$$

- Ahora, se realiza la actualización de los parámetros con descenso de gradiente. Se tiene que los valores actualizados son:

$$\begin{aligned} w^{(1)} &= w^{(0)} - \eta\frac{\partial E}{\partial w} = 1 - 0.01(-7.98) \\ &= 1.0789, \\ b^{(1)} &= b^{(0)} - \eta\frac{\partial E}{\partial b} = 0 - 0.01(-3.8) \\ &= 0.038 \end{aligned}$$

donde la tasa de aprendizaje se ha tomado como $\eta = 0.01$.

- Finalmente, con los nuevos valores de $w^{(1)}$ y $b^{(1)}$, se evalúa nuevamente el modelo y la función de pérdida resultando:

$$\begin{aligned} \widetilde{u}_1^{(1)} &= \widetilde{u}_1\left(w^{(1)}, b^{(1)}\right) = 2.1w^{(1)} + b^{(1)} \\ &= 2.1(1.0789) + 0.038 \\ &= 2.30558, \end{aligned}$$

$$\begin{aligned} E^{(1)} &= E(w^{(1)}, b^{(1)}) = \left(\widetilde{u}_1^{(1)} - 4\right)^2 \\ &= (2.30558 - 4)^2 \\ &= 2.87. \end{aligned}$$

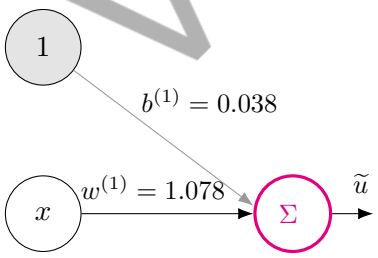


Figura 2.54 Red neuronal con parámetros $w^{(1)}$ y $b^{(1)}$ correspondientes a la primera época.

Verificamos así que la función de pérdida se reduce efectivamente, pasando del valor inicial $E^{(0)} = 3.61$ al nuevo valor $E^{(1)} = 2.87$, lo que confirma que la actualización de los parámetros produjo una disminución del error.

Vista previa 2026

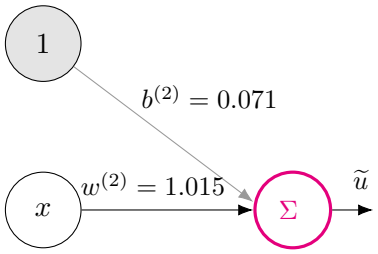


Figura 2.55 Red neuronal con parámetros $w^{(2)}$ y $b^{(2)}$ correspondientes a la segunda época.

La segunda época

- Repetimos el procedimiento utilizando ahora los valores $w^{(1)}$ y $b^{(1)}$. En esta segunda época, ya tenemos evualado el modelo y la pérdida:

$$\widetilde{u}_1^{(1)} = 2.30558, \qquad E^{(1)} = 2.87.$$

- Ahora calculamos las derivadas locales:

$$\frac{\partial E}{\partial \widetilde{u}} = 2(2.30558 - 4) = -3.38884, \qquad \frac{\partial \widetilde{u}}{\partial w} = 2.1, \qquad \frac{\partial \widetilde{u}}{\partial b} = 1.$$

Por lo tanto:

$$\begin{aligned} \frac{\partial E}{\partial w} &= (-3.38884)(2.1) = -7.116564, \\ \frac{\partial E}{\partial b} &= -3.38884. \end{aligned}$$

- Ahora actualizamos los parámetros:

$$\begin{aligned} w^{(2)} &= w^{(1)} - \eta \frac{\partial E}{\partial w} = 1.0798 - 0.01(-7.116564) \\ &= 1.15097, \\ b^{(2)} &= b^{(1)} - \eta \frac{\partial E}{\partial b} = 0.038 - 0.01(-3.38884) \\ &= 0.07189 \end{aligned}$$

- Finalmente, evaluamos nuevamente:

$$\begin{aligned} \widetilde{u}_1^{(2)} &= 2.1(1.15097) + 0.07189 \\ &= 2.48893, \\ E^{(2)} &= (2.48893 - 4)^2 \\ &= 2.28. \end{aligned}$$

Se observa que la función de pérdida continúa disminuyendo de manera consistente a lo largo de las iteraciones, pasando de $E^{(0)} = 3.61$ a $E^{(1)} = 2.87$ y posteriormente a $E^{(2)} = 2.28$. Este comportamiento confirma que el método de descenso de gradiente está actualizando los parámetros en la dirección opuesta al gradiente, acercando progresivamente la salida del modelo al valor objetivo.

Notas

$E^{(0)} = 3.61,$
$E^{(1)} = 2.87,$
$E^{(2)} = 2.28.$

Figura 2.56 La función de pérdida continúa disminuyendo.

2.7.3 El programa en Python para calcular más épocas

Como se ha podido observar, realizar las iteraciones de manera manual resulta rápidamente engorroso y poco práctico, especialmente si se desea analizar el comportamiento del método a lo largo de un número considerable de iteraciones. Por ello, es conveniente implementar un pequeño programa en Python que automatice el cálculo de la función de pérdida y la actualización de los parámetros en cada iteración.

Vista previa 2026



Figura 2.57 Es conveniente implementar un pequeño programa en Python que automatice los cálculos.

Además de simplificar los cálculos, esta implementación permite estudiar de manera más clara la evolución del error y responder a una pregunta natural: ¿seguirá disminuyendo la función de pérdida conforme aumentamos el número de iteraciones? A través del programa, podremos verificar empíricamente la convergencia del método de descenso de gradiente. La implementación se encuentra en el Código 2.1.

```python
import numpy as np
import matplotlib.pyplot as plt

# Datos iniciales
x1, u1 = 2.1, 4.0
w, b   = 1.0, 0.0
eta    = 0.01
epochs = 100

# Historial
lossV, wV, bV = [], [], []

for epoch in range(epochs):
    # Forward y funcion de perdida
    u = w*x1 + b
    error = u - u1
    loss = error**2
    # Gradientes
    dw = 2*error*x1
    db = 2*error
    # Actualizacion
    w -= eta*dw
    b -= eta*db
    # Guardar
    lossV.append(loss)
    wV.append(w)
    bV.append(b)
    print(f"Epoch {epoch+1}: Loss={loss:.6f}")

# Graficas
plt.plot(lossV)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
plt.plot(wV, label="w")
plt.plot(bV, label="b")
plt.legend()
plt.show()
```

Código 2.1 Implementación en *Python* para el Ejemplo 2.7.2.

Notas

```
x1     = 2.1
u1     = 4.0
w      = 1.0
b      = 0.0
eta    = 0.01
epochs = 100
```

Figura 2.58 Los parámetros del programa son los mismos que del ejemplo con valores específicos.

El programa sigue exactamente los mismos pasos descritos anteriormente. En primer lugar, evalúa el modelo en el dato proporcionado y calcula la función de pérdida mediante el error cuadrático. Posteriormente, determina el gradiente con respecto a los parámetros del modelo y, finalmente, actualiza dichos parámetros utilizando el algoritmo de descenso de gradiente con la tasa de aprendizaje especificada. Notar que a diferencia del procedimiento realizado manualmente, el programa no vuelve a calcular la función de pérdida inmediatamente después de actualizar los parámetros, ya que este cálculo se realiza al inicio de la siguiente iteración. De esta manera, se evitan operaciones redundantes del algoritmo.

Vista previa 2026

```
Epoch  0: Loss = 3.610000
Epoch  1: Loss = 2.871059
Epoch  2: Loss = 2.283374
Epoch  3: Loss = 1.815984
Epoch  4: Loss = 1.444265
Epoch  5: Loss = 1.148635
Epoch  6: Loss = 0.913517
Epoch  7: Loss = 0.726527
Epoch  8: Loss = 0.577812
Epoch  9: Loss = 0.459538
Epoch 10: Loss = 0.365474
Epoch 11: Loss = 0.290664
Epoch 12: Loss = 0.231167
Epoch 13: Loss = 0.183849
Epoch 14: Loss = 0.146216
```

Figura 2.59 Comportamiento del error en las primeras épocas durante el proceso de entrenamiento.

En este caso, el proceso se repite a lo largo de 100 épocas, lo que permite analizar la evolución del error y el comportamiento de la convergencia del método. En la Figura 2.59 se muestran los primeros valores de la función de pérdida. Se puede observar que los tres primeros coinciden con los obtenidos previamente de manera manual, lo cual confirma la correcta implementación del algoritmo.

Asimismo, se aprecia que el error disminuye conforme aumenta el número de iteraciones, evidenciando el efecto del descenso de gradiente. Por ejemplo, alrededor de la época 15 se obtiene un valor de $E \approx 0.146$, lo que indica una reducción significativa respecto al error inicial. Para analizar con mayor claridad esta tendencia decreciente, resulta conveniente visualizar la evolución completa de la función de pérdida, tal como se muestra en la Figura 2.60 a lo largo de las 100 épocas.

Los resultados son claros y consistentes con lo esperado: la función de pérdida decrece rápidamente y, después de aproximadamente 30 iteraciones, los cambios se vuelven poco perceptibles en escala lineal. Sin embargo, al considerar una escala logarítmica, se observa que el error continúa disminuyendo de manera sostenida. De hecho, para 100 épocas, la función de pérdida alcanza valores inferiores a 10^{-9}, lo que evidencia la convergencia del método hacia el mínimo.

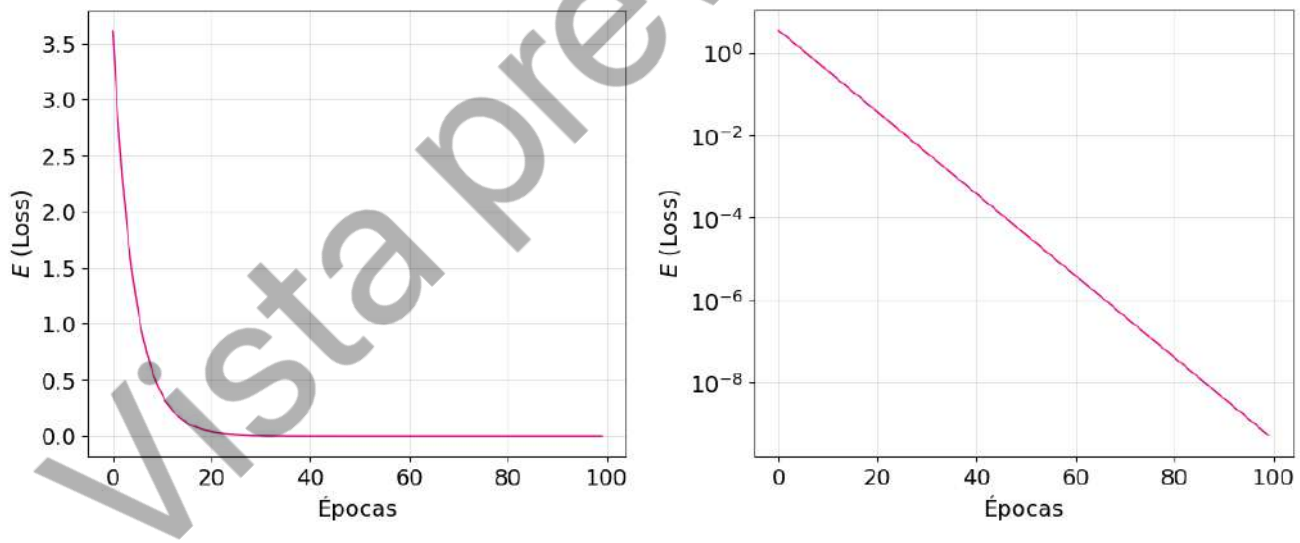


Figura 2.60 Resultados en escala lineal y logarítmica de la función de pérdida a lo largo de 100 épocas.

De manera análoga a la función de pérdida, también es posible graficar la evolución de los parámetros w y b en función del número de épocas. Esta información se presenta en la Figura 2.61. Los resultados muestran que ambos parámetros experimentan cambios significativos durante las primeras iteraciones; sin embargo, a partir de aproximadamente la época 40, sus valores comienzan a estabilizarse, presentando variaciones cada vez menores.

Vista previa 2026

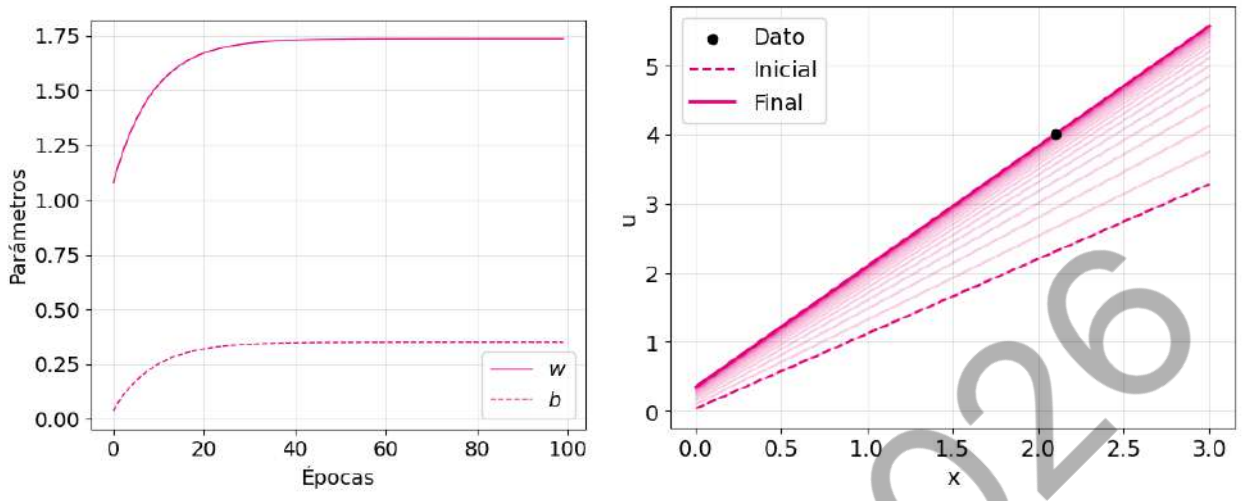


Figura 2.61 Resultados de los parámetros del modelo a lo largo de 100 épocas. La segunda figura corresponde a la evolución de la recta $\widetilde{u} = wx + b$ durante el entrenamiento.

```
Epoch  0: w=1.000, b=0.000
Epoch  1: w=1.080, b=0.038
Epoch  2: w=1.151, b=0.072
Epoch  3: w=1.214, b=0.102
Epoch  4: w=1.271, b=0.129
Epoch  5: w=1.322, b=0.153
...
Epoch 50: w=1.735, b=0.350
Epoch 51: w=1.735, b=0.350
Epoch 52: w=1.735, b=0.350
...
Epoch 88: w=1.737, b=0.351
Epoch 89: w=1.737, b=0.351
Epoch 90: w=1.737, b=0.351
```

Figura 2.62 Parámetros del modelo en distintas épocas.

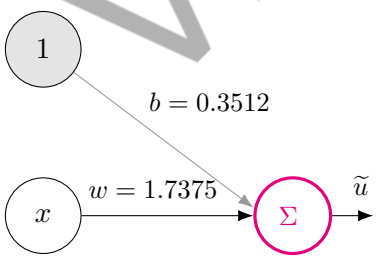


Figura 2.63 Red neuronal con parámetros w y b tras la convergencia del entrenamiento.

En particular, se observa que los parámetros tienden hacia los valores $w \approx 1.7375$ y $b \approx 0.3512$, lo cual es consistente con la convergencia del método y con la minimización de la función de pérdida. En la misma figura también se aprecia la evolución de $\widetilde{u} = wx + b$ a lo largo del proceso de entrenamiento: partiendo de una configuración inicial alejada del dato, las rectas intermedias muestran de manera gradual cómo el modelo se ajusta progresivamente, modificando tanto su pendiente como su intersección. Este comportamiento refleja de forma clara el efecto del entrenamiento, en donde cada actualización acerca la recta al punto conocido, hasta alcanzar una configuración prácticamente estable.

A través de este programa, también es posible analizar la influencia de la tasa de aprendizaje en la rapidez y estabilidad de la convergencia del método de descenso de gradiente. Como hemos mencionado anteriormente, este parámetro controla el tamaño de los pasos en cada actualización y, por lo tanto, juega un papel fundamental en el comportamiento del algoritmo: valores pequeños pueden conducir a una convergencia lenta, mientras que valores demasiado grandes pueden provocar oscilaciones o incluso la divergencia del método. Explorar estos efectos mediante experimentación numérica resulta altamente ilustrativo, por lo que se deja como ejercicio para el lector.

Antes de introducir arquitecturas de redes neuronales más complejas, abordaremos un segundo ejemplo en el que utilizaremos la misma arquitectura compuesta por una sola neurona, pero considerando ahora múltiples datos de entrenamiento. Este ejemplo nos permitirá profundizar en la comprensión del procedimiento descrito previamente. De esta manera, consolidaremos los conceptos fundamentales antes de incorporar una función de activación al modelo neuronal.

Vista previa 2026

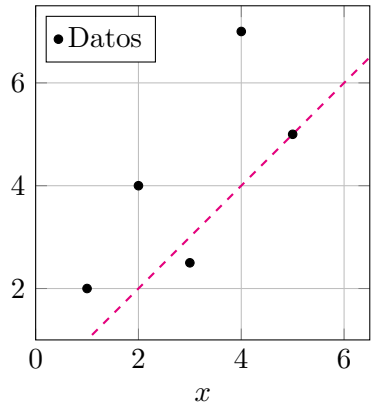


Figura 2.64 Datos y recta $\widetilde{u}$ dada por los parámetros iniciales de la red neuronal.

2.7.4 Un ejemplo lineal con cinco datos conocidos

En el ejemplo anterior, estudiamos el proceso de entrenamiento de una red neuronal lineal utilizando un único dato conocido. Esta simplificación nos permitió comprender cómo se calcula la función de pérdida, cómo se obtienen sus derivadas mediante la regla de la cadena y cómo se actualizan los parámetros utilizando el método de descenso de gradiente.

Sin embargo, en la práctica, el entrenamiento de una red neuronal se realiza generalmente utilizando un conjunto de datos. Por ello, en este segundo ejemplo extenderemos el procedimiento anterior para considerar cinco datos de entrenamiento, manteniendo la misma arquitectura y los mismos parámetros iniciales.

El objetivo será mostrar que el procedimiento de entrenamiento sigue siendo esencialmente el mismo, con la diferencia de que ahora debemos considerar la contribución de todos los datos al calcular la función de pérdida y su gradiente.

EJEMPLO CON CINCO DATOS CONOCIDOS

Consideremos nuevamente la red neuronal lineal definida por

$$\widetilde{u}(x; w, b) = wx + b,$$

con los parámetros iniciales

$$w^{(0)} = 1, \qquad b^{(0)} = 0.$$

Supongamos que disponemos de cinco datos de entrenamiento:

$$(\hat{x}_1, \hat{u}_1) = (1, 2), \quad (\hat{x}_2, \hat{u}_2) = (2, 4), \quad (\hat{x}_3, \hat{u}_3) = (3, 2.5),$$
$$(\hat{x}_4, \hat{u}_4) = (4, 7), \quad (\hat{x}_5, \hat{u}_5) = (5, 5).$$

El proceso de optimización se llevará a cabo mediante el método de descenso de gradiente con una tasa de aprendizaje

$$\eta = 0.01.$$

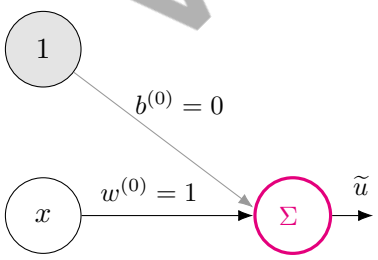


Figura 2.65 Red neuronal con parámetros iniciales.

A diferencia del ejemplo anterior, en el que la función de pérdida estaba definida para un único dato, ahora utilizaremos el error cuadrático medio, definido por

$$E(w, b) = \frac{1}{5}\sum_{i=1}^{5} E_i, \qquad \text{donde} \quad E_i = (\widetilde{u}_i - \hat{u}_i)^2 \tag{2.35}$$

y

$$\widetilde{u}_i = \widetilde{u}(\hat{x}_i; w, b) = w\hat{x}_i + b \tag{2.36}$$

representa la salida del modelo evaluada en el dato $\hat{x}_i$.

Vista previa 2026

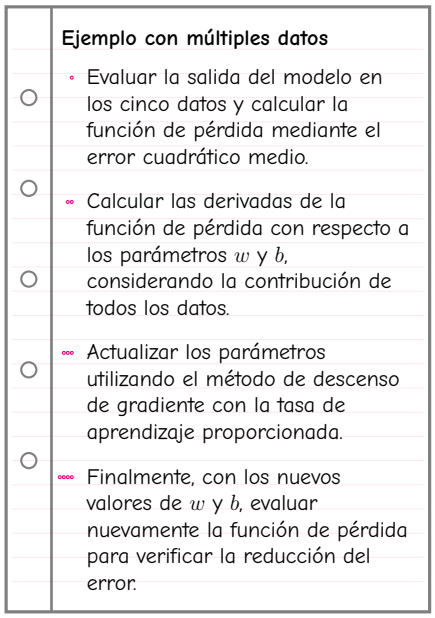


Figura 2.66 Entrenamiento de una red neuronal con múltiples datos.

Esta función mide el promedio de los errores cuadráticos correspondientes a los cinco datos. De esta manera, el entrenamiento buscará determinar los parámetros w y b que permitan ajustar la recta $\widetilde{u}$ al conjunto de observaciones, en lugar de ajustarla a un único punto. Al igual que en el ejemplo anterior, calcularemos manualmente las dos primeras épocas para comprender cómo se desarrolla el proceso de entrenamiento.

La primera época

- El primer paso consiste en evaluar el modelo en los cinco datos utilizando los parámetros iniciales $w^{(0)} = 1$ y $b^{(0)} = 0$. Como

$$\widetilde{u}_i^{(0)} = w^{(0)}\hat{x}_i + b^{(0)},$$

obtenemos $\widetilde{u}_1^{(0)} = 1$, $\widetilde{u}_2^{(0)} = 2$, $\widetilde{u}_3^{(0)} = 3$, $\widetilde{u}_4^{(0)} = 4$, y $\widetilde{u}_5^{(0)} = 5$. Sustituyendo estos valores en la función de pérdida (2.35), tenemos

$$\begin{aligned} E^{(0)} &= E(w^{(0)}, b^{(0)}) \\ &= \frac{1}{5}\left[(1-2)^2 + (2-4)^2 + (3-2.5)^2 + (4-7)^2 + (5-5)^2\right] \\ &= \frac{1}{5}\left[1 + 4 + 0.25 + 9 + 0\right] = 2.85. \end{aligned}$$

Por lo tanto, tenemos que $E^{(0)} = 2.85$ con los parámetros iniciales.

- Para calcular el gradiente utilizamos nuevamente la regla de la cadena. Sin embargo, ahora debemos considerar la contribución de los cinco datos. De acuerdo con (2.35), tenemos

$$\frac{\partial E}{\partial w} = \frac{1}{5}\sum_{i=1}^{5} \frac{\partial E_i}{\partial \widetilde{u}_i}\frac{\partial \widetilde{u}_i}{\partial w}, \qquad \text{y} \qquad \frac{\partial E}{\partial b} = \frac{1}{5}\sum_{i=1}^{5} \frac{\partial E_i}{\partial \widetilde{u}_i}\frac{\partial \widetilde{u}_i}{\partial b}.$$

Las derivadas parciales están dadas por

$$\frac{\partial E_i}{\partial \widetilde{u}_i} = 2(\widetilde{u}_i - \hat{u}_i), \qquad \frac{\partial \widetilde{u}_i}{\partial w} = \hat{x}_i, \qquad \frac{\partial \widetilde{u}_i}{\partial b} = 1.$$

En consecuencia,

$$\frac{\partial E}{\partial w} = \frac{2}{5}\sum_{i=1}^{5}(\widetilde{u}_i - \hat{u}_i)\hat{x}_i, \qquad \text{y} \qquad \frac{\partial E}{\partial b} = \frac{2}{5}\sum_{i=1}^{5}(\widetilde{u}_i - \hat{u}_i). \tag{2.37}$$

Sustituyendo los valores obtenidos anteriormente, calculamos

$$\begin{aligned} \frac{\partial E}{\partial w} &= \frac{2}{5}\left[(-1)(1) + (-2)(2) + (0.5)(3) + (-3)(4) + (0)(5)\right] = \text{-}6.2, \\ \frac{\partial E}{\partial b} &= \frac{2}{5}\left[-1 - 2 + 0.5 - 3 + 0\right] = \text{-}2.2. \end{aligned}$$

Por lo tanto, el gradiente de la función de pérdida, evaluado en los parámetros iniciales, es

$$\nabla E(w^{(0)}, b^{(0)}) = (-6.2, -2.2).$$

Obsérvese que cada dato contribuye al cálculo del gradiente, pero todas estas contribuciones se combinan antes de actualizar los parámetros del modelo.

Notas

$$\begin{aligned} \widetilde{u}_1^{(0)} &= 1(1) + 0 = 1, \\ \widetilde{u}_2^{(0)} &= 1(2) + 0 = 2, \\ \widetilde{u}_3^{(0)} &= 1(3) + 0 = 3, \\ \widetilde{u}_4^{(0)} &= 1(4) + 0 = 4, \\ \widetilde{u}_5^{(0)} &= 1(5) + 0 = 5. \end{aligned}$$

Figura 2.67 Evaluación de los datos en la red neuronal con los parámetros iniciales.

Vista previa 2026

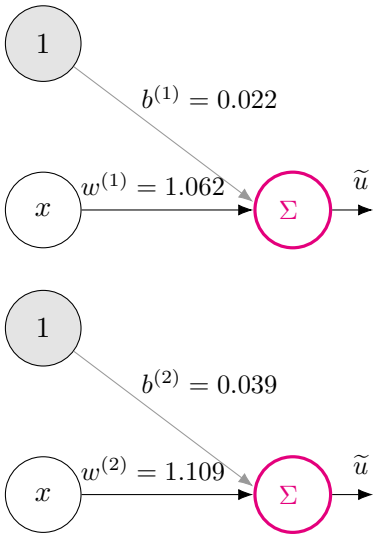


Figura 2.68 Red neuronal con parámetros correspondientes a la primera y segunda época usando múltiples datos.

∘∘∘ Ahora realizamos la actualización de los parámetros mediante descenso de gradiente. Los nuevos valores son

$$w^{(1)} = w^{(0)} - \eta\frac{\partial E}{\partial w} = 1 - 0.01(-6.2) = 1.062,$$

$$b^{(1)} = b^{(0)} - \eta\frac{\partial E}{\partial b} = 0 - 0.01(-2.2) = 0.022.$$

Así, después de la primera época, el modelo actualizado es

$$\widetilde{u}(x) = 1.062x + 0.022.$$

∘∘∘∘ Finalmente, con los nuevos valores de $w^{(1)}$ y $b^{(1)}$, evaluamos nuevamente el modelo y la función de pérdida. En este caso,

$$\widetilde{u}_i^{(1)} = 1.062\hat{x}_i + 0.022,$$

de donde obtenemos

$$\widetilde{u}_1^{(1)} = 1.084,\ \widetilde{u}_2^{(1)} = 2.146,\ \widetilde{u}_3^{(1)} = 3.208,\ \widetilde{u}_4^{(1)} = 4.270,\ \widetilde{u}_5^{(1)} = 5.332.$$

Por lo tanto,

$$\begin{aligned} E^{(1)} &= E(w^{(1)}, b^{(1)}) \\ &= \frac{1}{5}\Big[(1.084-2)^2 + (2.146-4)^2 \\ &\quad +(3.208-2.5)^2 + (4.270-7)^2 + (5.332-5)^2\Big] \\ &= 2.468152. \end{aligned}$$

Verificamos así que la función de pérdida se reduce efectivamente, pasando del valor inicial $E^{(0)} = 2.85$ al nuevo valor $E^{(1)} = 2.468152$, lo que confirma que la actualización de los parámetros produjo una disminución del error promedio.

Para la segunda época, repetimos el procedimiento utilizando ahora los parámetros $w^{(1)}$ y $b^{(1)}$. Calculamos nuevamente el gradiente y actualizamos los parámetros obteniendo $w^{(2)} = 1.10904$ y $b^{(2)} = 0.03984$. Finalmente, tenemos que la función de pérdida continúa disminuyendo a lo largo de las iteraciones, pasando de $E^{(0)} = 2.85$ a $E^{(1)} = 2.468152$ y posteriormente a $E^{(2)} = 2.244743$. Este comportamiento confirma que el método de descenso de gradiente está actualizando los parámetros en la dirección opuesta al gradiente, reduciendo progresivamente el error promedio del modelo.

Notas

$$\begin{aligned} E^{(0)} &= 2.850, \\ E^{(1)} &= 2.468, \\ E^{(2)} &= 2.244. \end{aligned}$$

Figura 2.69 Error determinada por la función de pérdida para el ejemplo con múltiples datos.

Existe una diferencia importante entre el ejemplo de un único y varios datos. Cuando se dispone de un único dato, es posible encontrar múltiples rectas que pasan exactamente por dicho punto, haciendo que la función de pérdida sea cero. En cambio, cuando se consideran varios datos que no pertenecen a una misma recta, como ocurre en este ejemplo, no es posible anular simultáneamente todos los errores. Por ello, el entrenamiento busca encontrar la recta que minimice el error cuadrático medio.

Vista previa 2026

```
Epoch  0:   Loss=2.850000
Epoch  1:   Loss=2.468152
Epoch  2:   Loss=2.244743
Epoch  3:   Loss=2.113671
Epoch  4:   Loss=2.036413
Epoch  5:   Loss=1.990522
Epoch  6:   Loss=1.962916
Epoch  7:   Loss=1.945972
Epoch  8:   Loss=1.935246
Epoch  9:   Loss=1.928149
Epoch 10:   Loss=1.923174
Epoch 11:   Loss=1.919440
Epoch 12:   Loss=1.916436
Epoch 13:   Loss=1.913862
Epoch 14:   Loss=1.911544
```

Figura 2.70 Comportamiento del error en las primeras épocas durante el proceso de entrenamiento.

```
Epoch  0: w=1.062, b=0.022
Epoch  1: w=1.109, b=0.040
Epoch  2: w=1.145, b=0.055
Epoch  3: w=1.172, b=0.067
Epoch  4: w=1.192, b=0.077
Epoch  5: w=1.207, b=0.086
...
Epoch 50: w=1.209, b=0.284
Epoch 51: w=1.208, b=0.288
Epoch 52: w=1.207, b=0.292
...
Epoch 97: w=1.164, b=0.449
Epoch 98: w=1.163, b=0.452
Epoch 99: w=1.162, b=0.455
```

Figura 2.71 Parámetros del modelo en distintas épocas.

2.7.5 El programa en Python para múltiples datos

Al igual que en el ejemplo anterior, realizar manualmente un número considerable de iteraciones resulta poco práctico. Por ello, implementaremos nuevamente un programa en *Python* que automatice los cálculos. La principal diferencia con respecto al programa anterior consiste en que ahora los datos se representan mediante arreglos de *NumPy*. Esto permite evaluar el modelo simultáneamente en los cinco datos y calcular el promedio de los errores utilizando la función `np.mean()`. La implementación se presenta en el Código 2.2.

```python
import numpy as np
import matplotlib.pyplot as plt

# Datos de entrenamiento
x1 = np.array([1, 2, 3,   4, 5], dtype=float)
u1 = np.array([2, 4, 2.5, 7, 5], dtype=float)
# Parametros iniciales
w, b   = 1.0, 0.0
eta    = 0.01
epochs = 100

# Historial
lossV, wV, bV = [], [], []

for epoch in range(epochs):
    # Forward y funcion de perdida
    u = w*x1 + b
    error = u - u1
    loss = np.mean(error**2)
    # Gradientes
    dw = 2*np.mean(error*x)
    db = 2*np.mean(error)
    # Actualizacion
    w -= eta*dw
    b -= eta*db
    # Guardar
    lossV.append(loss)
    wV.append(w)
    bV.append(b)
    print(f"Epoch {epoch}: Loss={loss:.6f}")

# Graficas
plt.plot(lossV)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
plt.plot(wV, label="w")
plt.plot(bV, label="b")
plt.legend()
plt.show()
```

Código 2.2 Implementación en *Python* para el ejemplo con cinco datos.

El programa sigue exactamente los mismos pasos descritos anteriormente. En primer lugar, evalúa el modelo en los cinco datos y calcula la función de pérdida mediante el error cuadrático medio. Posteriormente, determina el gradiente con respecto a los parámetros del modelo y, finalmente, actualiza dichos parámetros utilizando el método de descenso de gradiente con la tasa de aprendizaje especificada.

Vista previa 2026

En este caso, realizar únicamente 100 épocas no resulta suficiente para apreciar con claridad la convergencia de la función de pérdida y la estabilización de los parámetros del modelo, como se observa en la Figura 2.72. Por esta razón, extendemos el entrenamiento a 2000 épocas, lo que permite observar un comportamiento de convergencia más evidente.

Al igual que en el ejemplo anterior, la función de pérdida disminuye progresivamente; sin embargo, el error no converge a cero, debido a que los cinco datos no se encuentran sobre una misma recta. En este caso, los datos determinan una solución única de mínimos cuadrados, hacia la cual el algoritmo de descenso de gradiente aproxima gradualmente los parámetros del modelo. De esta manera, podemos observar cómo el entrenamiento busca encontrar la recta que mejor se ajusta al conjunto de datos como se muestra en la Figura 2.73.

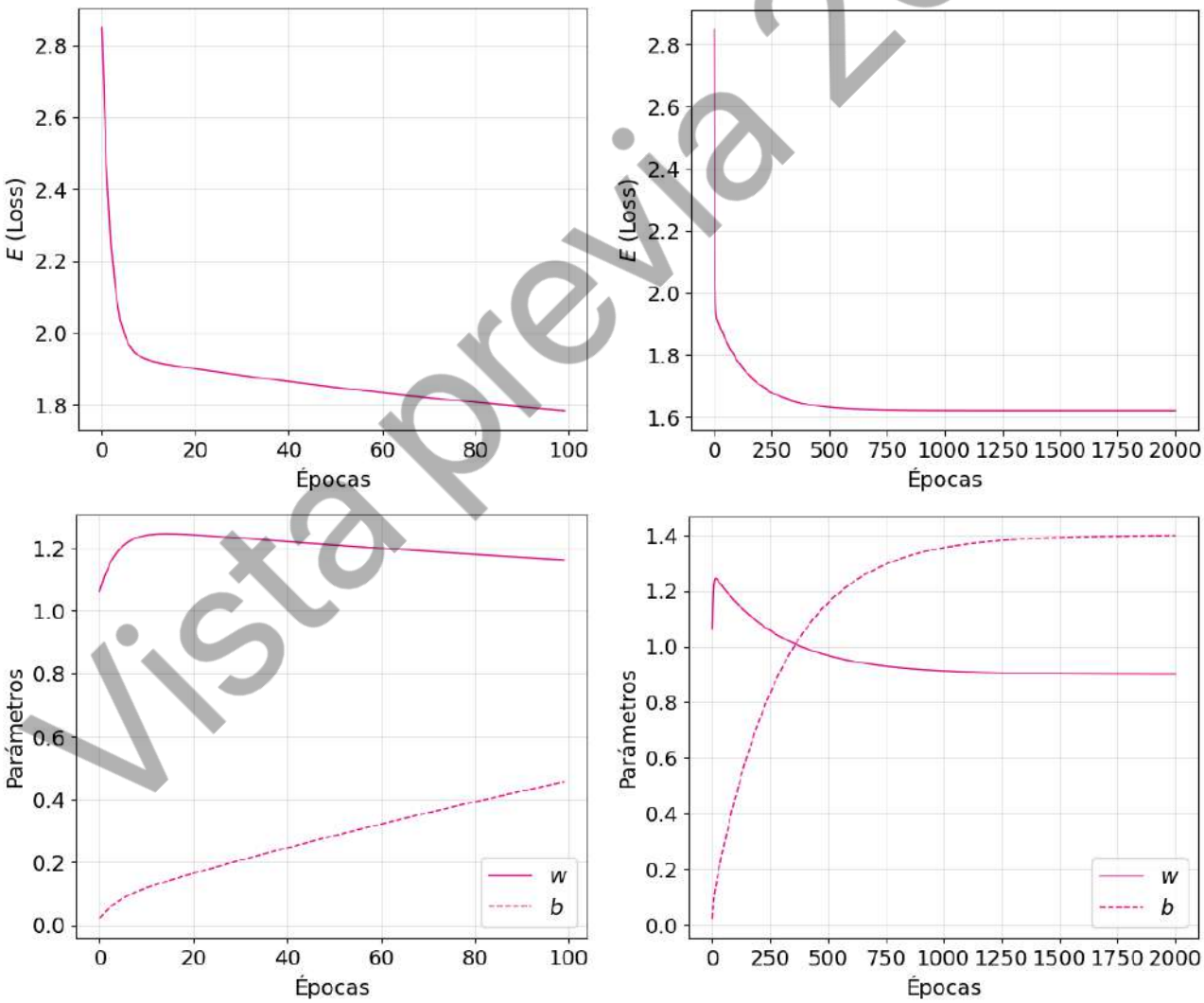


Figura 2.72 Resultados de la función de pérdida y de los parámetros a lo largo de 100 y 2000 épocas de entrenamiento.

Vista previa 2026

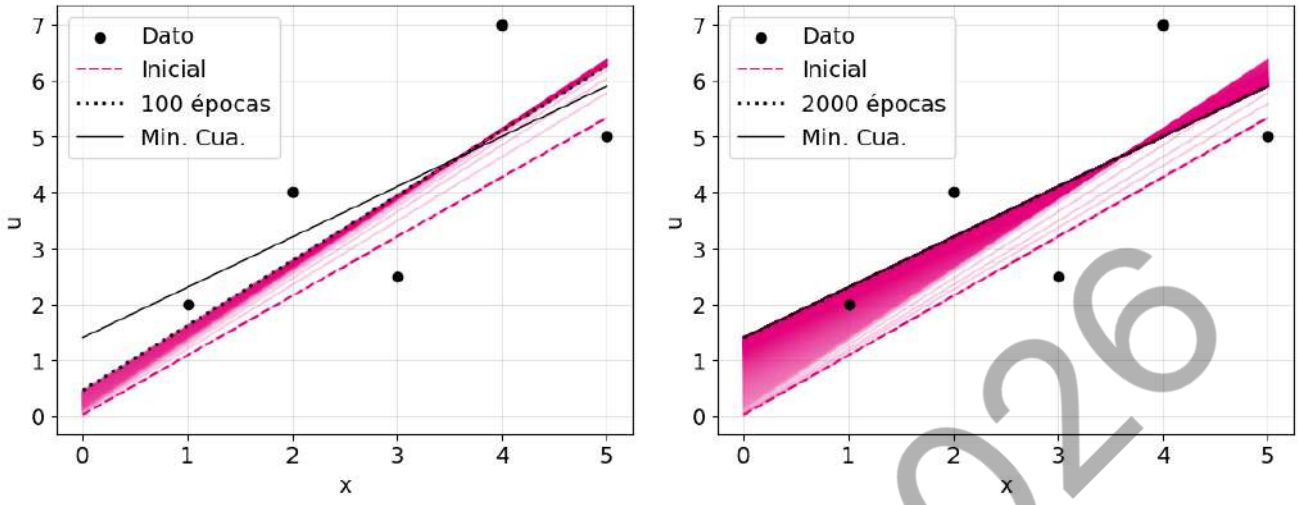


Figura 2.73 Evolución de la red $\widetilde{u}$ a lo largo de 100 y 2000 épocas de entrenamiento. La línea de color negro corresponde a la recta que minimiza el error cuadrático medio obtenida mediante mínimos cuadrados.

```
Epoch 1000 w=0.912 b=1.355
Epoch 1001 w=0.912 b=1.355
Epoch 1002 w=0.912 b=1.356
Epoch 1003 w=0.912 b=1.356
Epoch 1004 w=0.912 b=1.356
Epoch 1005 w=0.912 b=1.356
...
Epoch 1500 w=0.902 b=1.392
Epoch 1501 w=0.902 b=1.392
Epoch 1502 w=0.902 b=1.392
...
Epoch 1997 w=0.900 b=1.398
Epoch 1998 w=0.900 b=1.398
Epoch 1999 w=0.900 b=1.398
```

Figura 2.74 Parámetros del modelo en distintas épocas.

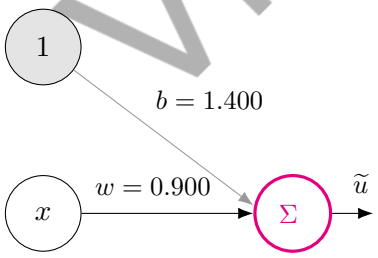


Figura 2.75 Red neuronal con parámetros w y b tras la convergencia de los parámetros durante el entrenamiento.

De hecho, para los cinco datos considerados, la recta que minimiza el error cuadrático medio está dada por $\widehat{u}(x) = 0.9x + 1.4$. Estos valores pueden obtenerse analíticamente mediante el método de mínimos cuadrados y constituyen una referencia para analizar la convergencia del proceso de entrenamiento.

Observamos que, después de 2000 épocas, los parámetros del modelo alcanzan aproximadamente los valores $w = 0.9004$, $b = 1.3984$. Aunque estos resultados son muy cercanos a los valores teóricos, todavía presentan pequeñas diferencias, lo que indica que el proceso de entrenamiento continúa aproximándose a la solución óptima. Al incrementar el número de épocas a 10,000, obtenemos $w = 0.900000$, $b = 1.3999999$, valores que coinciden con la solución analítica dentro de la precisión numérica utilizada.

Por último, este ejemplo permite observar que la metodología de entrenamiento no cambia al incrementar el número de datos. El procedimiento sigue consistiendo en evaluar el modelo, calcular la función de pérdida, determinar el gradiente mediante la regla de la cadena y actualizar los parámetros.

Es importante destacar que la metodología para redes neuronales más complejas no introduce una idea distinta a la presentada en este ejemplo; por el contrario, se trata exactamente del mismo mecanismo, evaluación hacia adelante y aplicación de la regla de la cadena hacia atrás, pero aplicado a estructuras de mayor tamaño y complejidad. La diferencia radica únicamente en la escala: más capas, más neuronas y, en consecuencia, más términos intermedios en el proceso de propagación del gradiente.

Vista previa 2026



Figura 2.76 *Un gran poder conlleva una gran responsabilidad.* Las redes neuronales multicapa ofrecen una mayor capacidad de modelación, también implican un incremento en la complejidad computacional.

2.8 ENTRENAMIENTO PARA REDES COMPLEJAS

En general, una red neuronal multicapa puede interpretarse como un número grande de composiciones de funciones elementales del tipo $\boldsymbol{z} = W \cdot \boldsymbol{x} + \boldsymbol{b}$, seguidas de funciones de activación $\phi(\boldsymbol{z})$ que introducen no linealidad en el modelo. Es precisamente esta estructura encadenada la que permite aplicar de manera sistemática la regla de la cadena durante el proceso de retropropagación.

Desde el punto de vista práctico, es fundamental que las funciones de activación posean derivadas sencillas. Por ejemplo, la función de activación $\phi(z) = \tanh(z)$ satisface

$$\frac{\partial \phi}{\partial z} = 1 - \phi(z)^2,$$

lo que significa que su derivada puede expresarse en términos de la propia función evaluada en el mismo punto. Esta propiedad es particularmente conveniente, ya que el valor de la activación calculado durante el paso hacia adelante puede reutilizarse directamente para obtener la derivada en el paso hacia atrás. A continuación analizaremos un ejemplo sencillo, pero que incorpora esta función de activación.

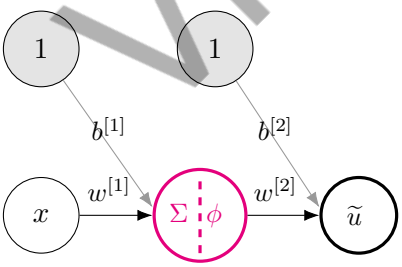


Figura 2.77 Red neuronal de dos capas y una neurona para ejemplificar el entrenamiento de la red neuronal.

EJEMPLO CON FUNCIÓN DE ACTIVACIÓN

Consideremos una red neuronal de dos capas de una sola neurona cada una, donde la primera capa utiliza la función de activación tanh y la segunda es lineal:

$$a^{[1]} = \tanh(w^{[1]}x + b^{[1]}),$$
$$\widetilde{u} = w^{[2]}a^{[1]} + b^{[2]}.$$

Se dispone de tres datos de entrenamiento:

$$(\hat{x}_1, \hat{u}_1) = (0,0), \quad (\hat{x}_2, \hat{u}_2) = (1,1), \quad (\hat{x}_3, \hat{u}_3) = (2,0),$$

y los parámetros iniciales:

$$w^{[1]} = 1,\ b^{[1]} = 0, \quad w^{[2]} = 2,\ b^{[2]} = 0.$$

El proceso de optimización se llevará a cabo mediante el método de descenso de gradiente con una tasa de aprendizaje

$$\eta = 0.01.$$

Utilizaremos como función de pérdida el error cuadrático medio:

$$E(\theta) = \frac{1}{3}\sum_{i=1}^{3}(\widetilde{u}_i - \hat{u}_i)^2,$$

donde $\theta = [w^{[1]}, w^{[2]}, b^{[1]}, b^{[2]}]^\top$ y $\widetilde{u}_i = \widetilde{u}(\hat{x}_i; \theta)$ $(i = 1,2,3)$.

Vista previa 2026

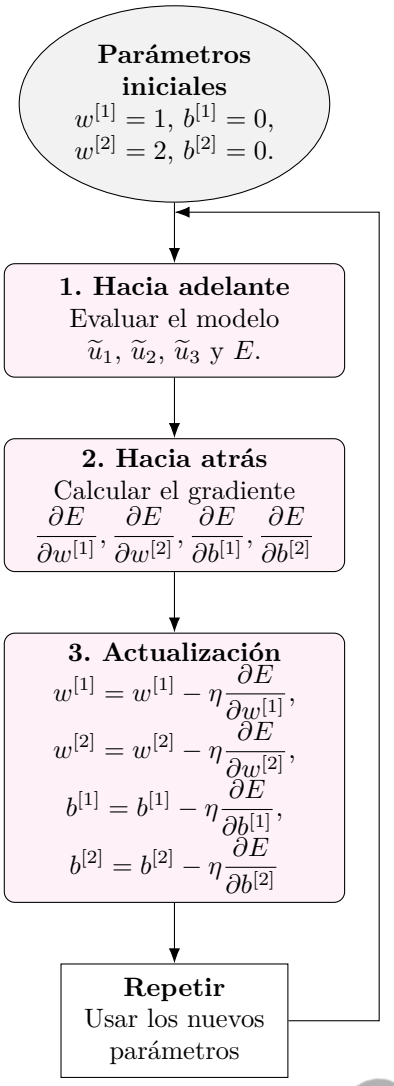


Figura 2.78 Esquema general del proceso de entrenamiento de la red neuronal con dos capas propuesta.

Notas

Notar que las iteraciones están denotadas por valores entre paréntesis, $E^{(0)}$, mientras que las capas entre corchetes, $w^{[1]}$, $w^{[2]}$, $b^{[1]}$, $b^{[2]}$.

Figura 2.79 Notación.

2.8.1 Cálculo manual de una primera época

En base a los datos anteriores y de manera consistente al ejemplo anterior procederemos a realizar los siguientes pasos.

- ◦ Evaluar la salida del modelo para cada uno de los datos y calcular la función de pérdida.
- ◦◦ Calcular las derivadas de la función de pérdida con respecto a los parámetros de ambas capas mediante retropropagación.
- ◦◦◦ Actualizar los parámetros utilizando descenso de gradiente y evaluar nuevamente la función de pérdida para verificar la reducción del error.

Al sustituir los valores de los datos disponibles, la salida del modelo para cada entrada $\hat{x}_i$ está dada por

$$\widetilde{u}_i = \widetilde{u}(\hat{x}_i; w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}), \qquad i = 1, 2, 3.$$

En consecuencia, a partir de estas expresiones podemos calcular el gradiente y proceder a la actualización del modelo.

Hacia adelante: evaluar el modelo

En el ejemplo considerado, al evaluar el modelo con los parámetros iniciales $w^{[1]} = 1$, $b^{[1]} = 0$ y los tres datos conocidos, se obtiene la salida de la primera capa:

$$\begin{aligned} a_1^{[1]} &= \tanh\left(w^{[1]}\hat{x}_1 + b^{[1]}\right) = \tanh(1\cdot 0 + 0) = \tanh(0) = 0, \\ a_2^{[1]} &= \tanh\left(w^{[1]}\hat{x}_2 + b^{[1]}\right) = \tanh(1\cdot 1 + 0) = \tanh(1) \approx 0.76159, \\ a_3^{[1]} &= \tanh\left(w^{[1]}\hat{x}_3 + b^{[1]}\right) = \tanh(1\cdot 2 + 0) = \tanh(2) \approx 0.96403. \end{aligned}$$

Ahora evaluamos la segunda capa usando $w^{[2]} = 2$ y $b^{[2]} = 0$:

$$\begin{aligned} \widetilde{u}_1 &= w^{[2]}a_1^{[1]} + b^{[2]} = 2(0) + 0 = 0, \\ \widetilde{u}_2 &= w^{[2]}a_2^{[1]} + b^{[2]} = 2(0.76159) + 0 = 1.52319, \\ \widetilde{u}_3 &= w^{[2]}a_3^{[1]} + b^{[2]} = 2(0.96403) + 0 = 1.92806. \end{aligned}$$

Entonces, la función de pérdida inicial está dada por

$$\begin{aligned} E^{(0)} &= \frac{1}{3}\left[(\widetilde{u}_1 - \hat{u}_1)^2 + (\widetilde{u}_2 - \hat{u}_2)^2 + (\widetilde{u}_3 - \hat{u}_3)^2\right] \\ &= \frac{1}{3}\left[(0-0)^2 + (1.52319 - 1)^2 + (1.92806 - 0)^2\right] \\ &= \frac{1}{3}\left[0 + 0.27373 + 3.71739\right] = 1.330374. \end{aligned}$$

Por lo tanto, $E^{(0)} = 1.330374$.

Vista previa 2026



Figura 2.80 La regla de la cadena es una de las herramientas fundamentales para poder calcular el gradiente con retropropagación.

Hacia atrás: calcular el gradiente

Para calcular el gradiente usamos la regla de la cadena. Para esto procederemos inicialmente calculando las derivadas locales para posteriormente calcular el gradiente con respecto a los parámetros de la segunda y primera etapa, respectivamente.

Derivadas parciales con respecto a $\widetilde{u}_i$

En primer lugar, se tiene que las derivadas locales con respecto a cada salida de la función de pérdida son

$$E(\theta) = \frac{1}{3}\sum_{i=1}^{3}(\widetilde{u}_i - \hat{u}_i)^2, \quad \Longrightarrow \quad \frac{\partial E}{\partial \widetilde{u}_i} = \frac{2}{3}(\widetilde{u}_i - \hat{u}_i), \quad i = 1, 2, 3.$$

Entonces:

$$\begin{aligned}
\frac{\partial E}{\partial \widetilde{u}_1} &= \frac{2}{3}(0 - 0) = 0, \\
\frac{\partial E}{\partial \widetilde{u}_2} &= \frac{2}{3}(1.52319 - 1) = \frac{2}{3}(0.52319) = 0.34879, \\
\frac{\partial E}{\partial \widetilde{u}_3} &= \frac{2}{3}(1.92806 - 0) = \frac{2}{3}(1.92806) = 1.28537.
\end{aligned}$$

Gradiente con respecto a los parámetros w y b.

Como la salida de nuestra red neuronal es lineal tenemos que

$$\widetilde{u}_i = w^{[2]}a_i^{[1]} + b^{[2]}, \quad \Longrightarrow \quad \frac{\partial \widetilde{u}_i}{\partial w^{[2]}} = a_i^{[1]}, \quad \text{y} \quad \frac{\partial \widetilde{u}_i}{\partial b^{[2]}} = 1.$$

Por tanto,

$$\begin{aligned}
\frac{\partial E}{\partial w^{[2]}} &= \sum_{i=1}^{3}\frac{\partial E}{\partial \widetilde{u}_i}\frac{\partial \widetilde{u}_i}{\partial w^{[2]}} \\
&= (0)(0) + (0.34879)(0.76159) + (1.28537)(0.96403) \\
&= 0 + 0.26564 + 1.23913 = 1.50477, \\
\frac{\partial E}{\partial b^{[2]}} &= \sum_{i=1}^{3}\frac{\partial E}{\partial \widetilde{u}_i}\frac{\partial \widetilde{u}_i}{\partial b^{[2]}} \\
&= 0 + 0.34879 + 1.28537 = 1.63416.
\end{aligned}$$

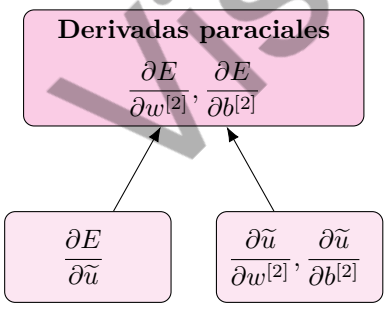


Figura 2.81 Esquema general del proceso para obtenerlas derivadas de la función de pérdida con respecto a los parámetros de la segunda capa.

Gradiente con respecto a la primera capa.

Aquí se hace más evidente el uso de la *retropropagación*. Como

$$\widetilde{u}_i = w^{[2]}a_i^{[1]} + b^{[2]}, \quad \text{donde} \quad a_i^{[1]} = \tanh(w^{[1]}\hat{x}_i + b^{[1]}),$$

por la regla de la cadena se tiene

Vista previa 2026

Notas

Recordemos que la función de activación $\phi(z) = \tanh(z)$ satisface

$$\frac{\partial \phi}{\partial z} = 1 - \phi(z)^2.$$

Figura 2.82 Propiedades de la función tanh.

$$\frac{\partial \widetilde{u}_i}{\partial w^{[1]}} = \frac{\partial \widetilde{u}_i}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial w^{[1]}}, \quad \text{y} \quad \frac{\partial \widetilde{u}_i}{\partial b^{[1]}} = \frac{\partial \widetilde{u}_i}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial b^{[1]}},$$

Por un lado, usando que $\tanh'(z) = 1 - \tanh^2(z)$ y la definición de $a_i^{[1]}$, obtenemos

$$\frac{\partial a_i^{[1]}}{\partial w^{[1]}} = \left(1 - \left(a_i^{[1]}\right)^2\right) \hat{x}_i \quad \text{y} \quad \frac{\partial a_i^{[1]}}{\partial b^{[1]}} = \left(1 - \left(a_i^{[1]}\right)^2\right).$$

Por otro lado,

$$\frac{\partial \widetilde{u}_i}{\partial a_i^{[1]}} = w^{[2]}.$$

Por tanto, juntando ambas expresiones tenemos que

$$\frac{\partial \widetilde{u}_i}{\partial w^{[1]}} = w^{[2]} \left(1 - \left(a_i^{[1]}\right)^2\right) \hat{x}_i, \quad \text{y} \quad \frac{\partial \widetilde{u}_i}{\partial b^{[1]}} = w^{[2]} \left(1 - \left((a_i^{[1]}\right)^2\right).$$

Entonces, Sustituyendo valor por valor:

$$\begin{aligned}
\frac{\partial E}{\partial w^{[1]}} &= \sum_{i=1}^{3} \frac{\partial E}{\partial \widetilde{u}_i} \frac{\partial \widetilde{u}_i}{\partial w^{[1]}} = \sum_{i=1}^{3} \frac{\partial E}{\partial \widetilde{u}_i} w^{[2]} \left(1 - (a_i^{[1]})^2\right) \hat{x}_i. \\
&= (0)(2)(1 - 0^2)(0) \\
&\quad +(0.34879)(2)(1 - 0.76159^2)(1) \\
&\quad +(1.28537)(2)(1 - 0.96403^2)(2) \\
&= 0 + (0.34879)(2)(0.41997) + (1.28537)(2)(0.07065)(2) \\
&= 0 + 0.29297 + 0.36325 \\
&= 0.65622.
\end{aligned}$$

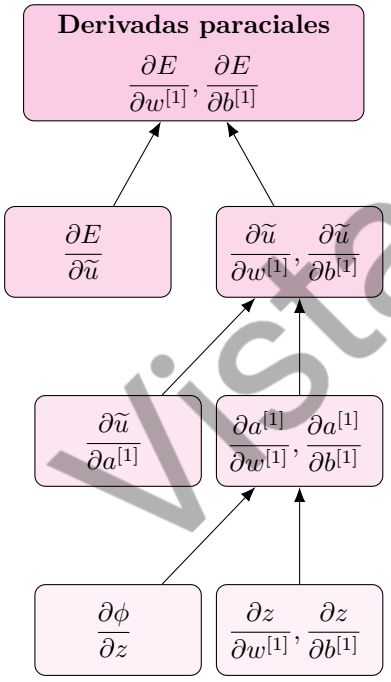


Figura 2.83 Esquema general del proceso para obtenerlas derivadas de la función de pérdida con respecto a los parámetros de la primera capa.

Análogamente,

$$\begin{aligned}
\frac{\partial E}{\partial b^{[1]}} &= \sum_{i=1}^{3} \frac{\partial E}{\partial \widetilde{u}_i} \frac{\partial \widetilde{u}_i}{\partial b^{[1]}} = \sum_{i=1}^{3} \frac{\partial E}{\partial \widetilde{u}_i} w^{[2]} \left(1 - (a_i^{[1]})^2\right) \\
&= (0)(2)(1 - 0^2) \\
&\quad +(0.34879)(2)(1 - 0.76159^2) \\
&\quad +(1.28537)(2)(1 - 0.96403^2) \\
&= 0 + (0.34879)(2)(0.41997) + (1.28537)(2)(0.07065) \\
&= 0 + 0.29297 + 0.18162 \\
&= 0.47459.
\end{aligned}$$

En consecuencia, el gradiente con respecto a todos los parámetros es aproximadamente

$$\nabla E \approx (0.65622,\ 0.47459,\ 1.50477,\ 1.63416)^\top.$$

Vista previa 2026

Actualización de parámetros

Ahora se realiza la actualización de los parámetros mediante descenso de gradiente con $\eta = 0.01$. Se tiene:

$$\begin{aligned} w^{[1]} &= w^{[1]} - \eta \frac{\partial E}{\partial w^{[1]}} = 1 - 0.01(0.65622) = 0.99344, \\ b^{[1]} &= b^{[1]} - \eta \frac{\partial E}{\partial b^{[1]}} = 0 - 0.01(0.47459) = -0.00475, \\ w^{[2]} &= w^{[2]} - \eta \frac{\partial E}{\partial w^{[2]}} = 2 - 0.01(1.50477) = 1.98495, \\ b^{[2]} &= b^{[2]} - \eta \frac{\partial E}{\partial b^{[2]}} = 0 - 0.01(1.63416) = -0.01634. \end{aligned}$$

Finalmente, tenemos la evaluación de los valores de la primera capa con los nuevos parámetros

$$\begin{aligned} a_1^{[1]} &= \tanh\left(w^{[1]}\hat{x}_1 + b^{[1]}\right) = \tanh(0.99344 \cdot 0 - 0.00475) \\ &= \tanh(-0.00475) \approx -0.00475, \\ a_2^{[1]} &= \tanh\left(w^{[1]}\hat{x}_2 + b^{[1]}\right) = \tanh(0.99344 \cdot 1 - 0.00475) \\ &= \tanh(0.98869) \approx 0.75680, \\ a_3^{[1]} &= \tanh\left(w^{[1]}\hat{x}_3 + b^{[1]}\right) = \tanh(0.99344 \cdot 2 - 0.00475) \\ &= \tanh(1.98213) \approx 0.96274. \end{aligned}$$



Figura 2.84 Aunque los pasos han sido pocos, ya se han establecido muchos conceptos fundamentales para aplicar redes neuronales en la resolución de ecuaciones diferenciales.

Ahora evaluamos la segunda capa obteniendo:

$$\begin{aligned} \widetilde{u}_1 &= w^{[2]}a_1^{[1]} + b^{[2]} = 1.98495(-0.00475) - 0.01634 \approx -0.02576, \\ \widetilde{u}_2 &= w^{[2]}a_2^{[1]} + b^{[2]} = 1.98495(0.75680) - 0.01634 \approx 1.48588, \\ \widetilde{u}_3 &= w^{[2]}a_3^{[1]} + b^{[2]} = 1.98495(0.96274) - 0.01634 \approx 1.89466. \end{aligned}$$

Entonces, la función de pérdida después de la primera época está dada por

$$\begin{aligned} E^{(0)} &= \frac{1}{3}\left[(\widetilde{u}_1 - \hat{u}_1)^2 + (\widetilde{u}_2 - \hat{u}_2)^2 + (\widetilde{u}_3 - \hat{u}_3)^2\right] \\ &= \frac{1}{3}\left[(-0.02576 - 0)^2 + (1.48588 - 1)^2 + (1.89466 - 0)^2\right] \\ &= 1.275489. \end{aligned}$$

Por lo tanto, $E^{(1)} = 1.275489$, la cual resulta ligeramente menor que $E^{(0)}$, confirmando la correcta dirección del descenso de gradiente.

Finalmente, es importante notar que varios términos intermedios, como $a_i^{[1]}$, se reutilizaron en diferentes partes del cálculo, lo cual constituye la base del algoritmo de *retropropagación* y permite una implementación computacional eficiente.

Vista previa 2026



Figura 2.85 Es conveniente implementar un pequeño programa en Python que automatice los cálculos.

2.8.2 El programa en Python para calcular más épocas

Ahora que contamos con una red neuronal ligeramente más compleja, podemos observar que realizar las iteraciones de manera manual se vuelve rápidamente un proceso engorroso y poco práctico. Por esta razón, también implementaremos un programa básico en Python que automatice el cálculo de la función de pérdida, la evaluación del modelo y la actualización de los parámetros en cada iteración del entrenamiento. La implementación de este método se encuentra en el Código 2.3.

```python
import numpy as np
import matplotlib.pyplot as plt

# Datos
x1 = np.array([0.0, 1.0, 2.0])
u1 = np.array([0.0, 1.0, 0.0])

# Parametros iniciales
w1, b1 = 1.0, 0.0
w2, b2 = 2.0, 0.0
eta    = 0.01
epochs = 200

# Historial
lossV = []
w1V, b1V, w2V, b2V = [], [], [], []

# Entrenamiento
for epoch in range(epochs):

    # Forward
    a1 = np.tanh(w1*x1 + b1)       # Primera capa
    u  = w2*a1 + b2                # Segunda capa

    # Funcion de perdida
    loss = (1/3.0) * np.sum((u - u1)**2)

    # Gradientes segunda capa
    error = (2.0/3.0) * (u - u1)
    dw2 = np.sum(error * a1)
    db2 = np.sum(error )

    # Gradientes primera capa
    da1 = error * w2
    dtanh = (1 - a1**2)
    dw1 = np.sum(da1 * dtanh * x1)
    db1 = np.sum(da1 * dtanh)

    # Actualizacion
    w1 -= eta * dw1
    b1 -= eta * db1
    w2 -= eta * dw2
    b2 -= eta * db2

    # Guardar
    lossV.append(loss)
    w1V.append(w1)
    b1V.append(b1)
    w2V.append(w2)
    b2V.append(b2)
```

Código 2.3 Implementación en *Python* para el ejemplo con función de activación tanh.

Notas

```
x1[0]=0, u1[0]=0,
x1[1]=1, u1[1]=1,
x1[2]=2, u1[2]=0,
w1 = 1, b1 = 0,
w2 = 2, b2 = 0,
eta    = 0.01
epochs = 200
```

Figura 2.86 Los parámetros del programa son los mismos que del ejemplo calculado manualmente.

Vista previa 2026

```
Epoch   0: Loss = 1.330374
Epoch   1: Loss = 1.275489
Epoch   2: Loss = 1.224515
Epoch   3: Loss = 1.177118
Epoch   4: Loss = 1.132997
Epoch   5: Loss = 1.091877
Epoch   6: Loss = 1.053509
Epoch   7: Loss = 1.017667
Epoch   8: Loss = 0.984146
Epoch   9: Loss = 0.952758
Epoch  10: Loss = 0.923332
...
Epoch 496: Loss = 0.191983
Epoch 497: Loss = 0.191952
Epoch 498: Loss = 0.191921
Epoch 499: Loss = 0.191890
Epoch 500: Loss = 0.191859
```

Figura 2.87 Comportamiento del error en las primeras épocas durante el proceso de entrenamiento.

El programa implementa de manera directa cada uno de los pasos que hemos descrito previamente de forma manual. Primero, evalúa la red neuronal en los tres datos de entrenamiento y, con ello, calcula la función de pérdida utilizando el error cuadrático medio. A continuación, determina el gradiente de dicha función con respecto a los parámetros del modelo mediante el proceso de *retropropagación*. Finalmente, con esta información, actualiza los parámetros aplicando el algoritmo de descenso de gradiente con la tasa de aprendizaje especificada.

En este caso, el entrenamiento requiere un mayor número de épocas para visualizar la evolución del error. En la Figura 2.87 se muestran los primeros valores de la función de pérdida. Puede observarse que los dos primeros coinciden con los obtenidos previamente de manera manual, lo cual confirma que la implementación del algoritmo es correcta. Se puede observar que el error disminuye a medida que aumenta el número de épocas. Sin embargo, para apreciar con mayor claridad la evolución del entrenamiento a lo largo de todas las iteraciones, resulta conveniente graficar la función de pérdida, como se muestra en la Figura 2.88, tanto en escala lineal como en escala logarítmica.

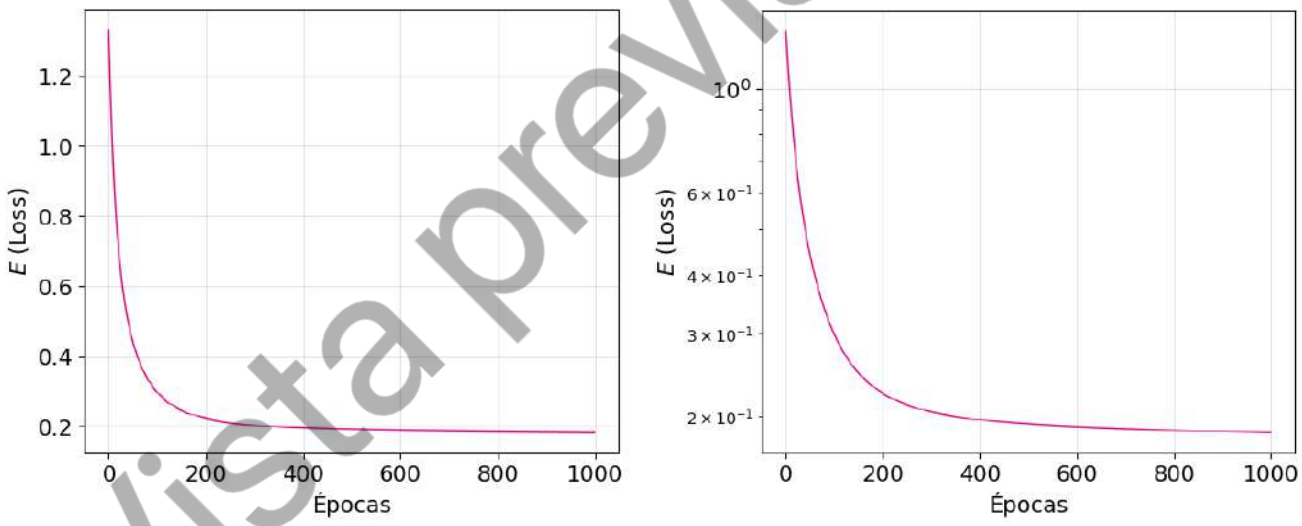


Figura 2.88 Resultados en escala lineal y logarítmica de la función de pérdida a lo largo de 1000 épocas.

Los resultados muestran un comportamiento interesante: la función de pérdida disminuye de manera apreciable durante las primeras iteraciones, lo que indica que el algoritmo de descenso de gradiente logra ajustar inicialmente los parámetros en una dirección adecuada. Sin embargo, después de cierto número de épocas, esta reducción se vuelve cada vez menor hasta que el error prácticamente deja de disminuir. De hecho, puede observarse que incluso después de 1000 épocas la pérdida permanece casi sin cambios significativos.

vista previa 2026

```
...
Epoch  985: Loss= 0.184276
Epoch  986: Loss= 0.184267
Epoch  987: Loss= 0.184257
Epoch  988: Loss= 0.184248
Epoch  989: Loss= 0.184239
Epoch  990: Loss= 0.184230
Epoch  991: Loss= 0.184221
Epoch  992: Loss= 0.184211
Epoch  993: Loss= 0.184202
Epoch  994: Loss= 0.184193
Epoch  995: Loss= 0.184184
Epoch  996: Loss= 0.184175
Epoch  997: Loss= 0.184166
Epoch  998: Loss= 0.184157
Epoch  999: Loss= 0.184148
Epoch 1000: Loss= 0.184139
```

Figura 2.89 Comportamiento del error en las últimas épocas del entrenamiento.

Esto ocurre porque, en este caso, la arquitectura de la red y la forma de la función de activación tanh imponen ciertas limitaciones sobre el tipo de función que el modelo puede aproximar. En consecuencia, llega un momento en el que el algoritmo alcanza una región donde ya no puede mejorar el ajuste. Dicho de otra forma, el método ha convergido hacia un mínimo, pero dicho mínimo no corresponde necesariamente a un error cercano a cero. Este comportamiento pone de manifiesto una idea importante en el entrenamiento de redes neuronales: no siempre aumentar el número de iteraciones garantiza una mejora adicional, ya que la calidad final del ajuste también depende de la arquitectura del modelo, de la elección de la función de activación y de la capacidad de representación de la red.

De manera análoga al análisis de la función de pérdida, también es posible graficar la evolución de los parámetros $w^{[1]}$, $w^{[2]}$, $b^{[1]}$ y $b^{[2]}$ en función del número de épocas. Esta información se presenta en la Figura 2.90. Los resultados muestran que la mayoría de los parámetros tienden a estabilizarse conforme avanza el entrenamiento, aunque el comportamiento de $w^{[1]}$ revela con mayor claridad las limitaciones de la arquitectura empleada.

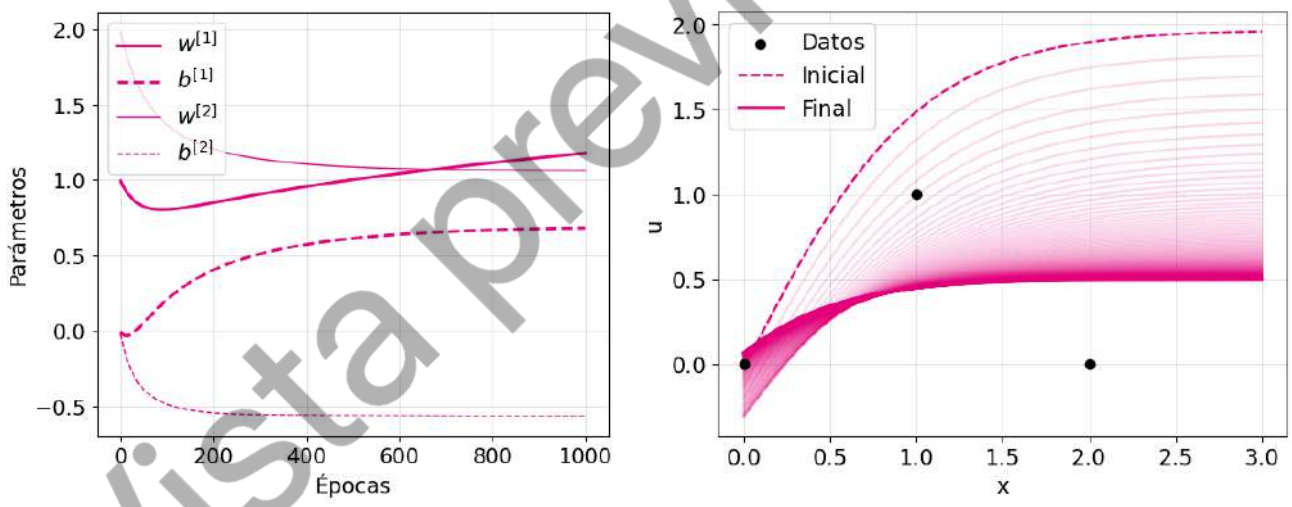


Figura 2.90 Resultados de los parámetros del modelo a lo largo de 1000 épocas. La segunda figura corresponde a la evolución de la curva dada por $\widetilde{u}$ durante el entrenamiento.

En efecto, los datos de entrenamiento presentan una forma tipo campana, es decir, una tendencia que primero crece y posteriormente decrece. Sin embargo, la red neuronal considerada en este ejemplo está construida a partir de una única función de activación tanh seguida de una salida lineal. Como la función tanh es esencialmente monótona, la familia de funciones que el modelo puede representar también queda severamente restringida. En consecuencia, aunque el algoritmo de descenso de gradiente logra ajustar parcialmente la red a los datos, no puede reproducir adecuadamente la forma global del conjunto de entrenamiento.

Vista previa 2026

Para observar un proceso de entrenamiento más estable, conviene elegir datos cuya forma sea compatible con la arquitectura propuesta. Por ejemplo, si se consideran los valores $(\hat{x}_1, \hat{u}_1) = (0, 0)$, $(\hat{x}_2, \hat{u}_2) = (1, 0.75)$ y $(\hat{x}_3, \hat{u}_3) = (2, 0.95)$, entonces se obtiene un comportamiento mucho más consistente con lo que se espera del algoritmo de entrenamiento. En este caso, la red neuronal logra ajustarse de manera más adecuada a los datos, lo que se traduce en una convergencia más estable del método y en una reducción progresiva de la función de pérdida, ver Figura 2.91.

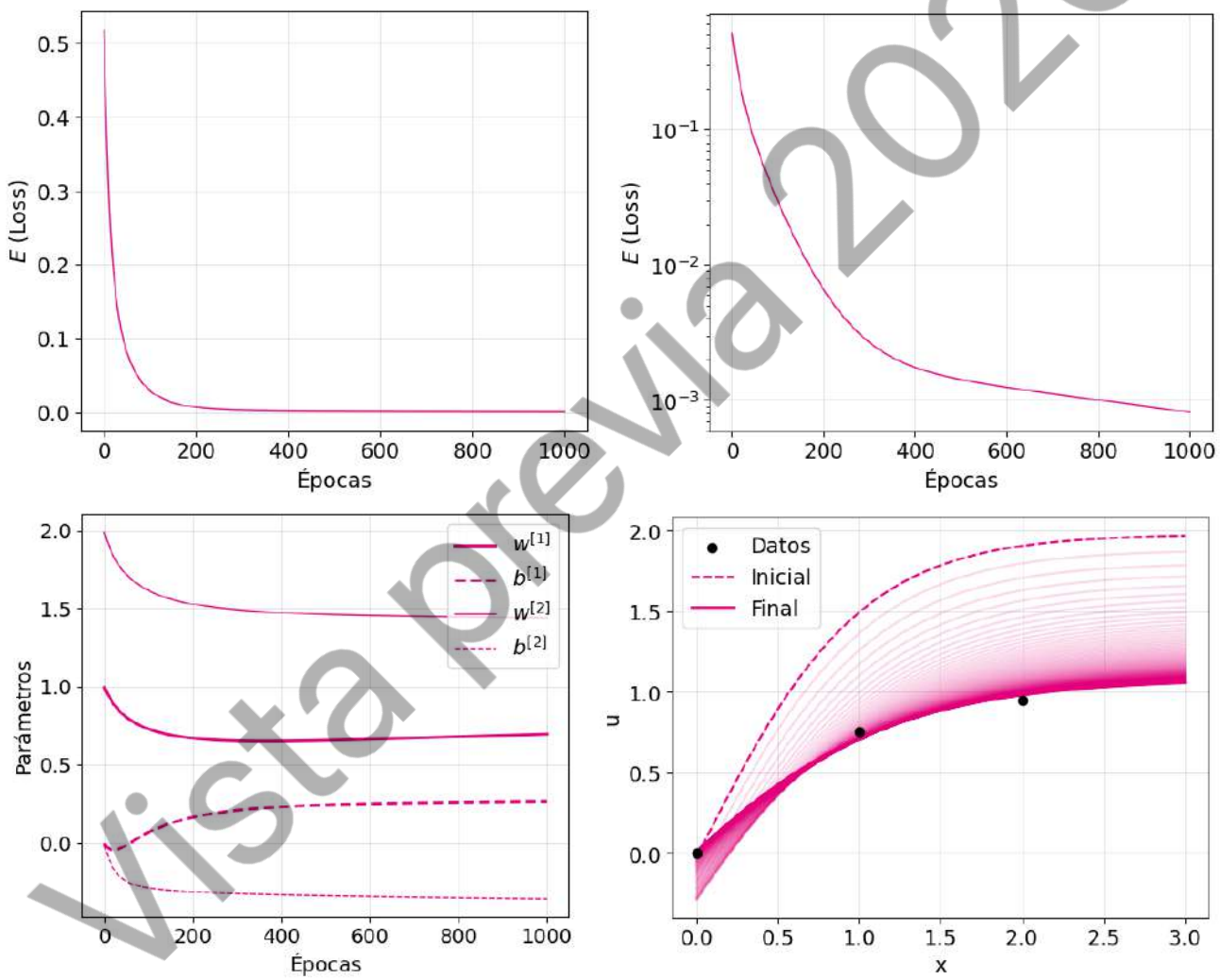


Figura 2.91 Resultados del modelo a lo largo de 1000 épocas empleando los nuevos datos de entrenamiento.

Este ejemplo ilustra cómo, al incrementar ligeramente la complejidad de la arquitectura y el número de datos, el cálculo del gradiente se vuelve considerablemente más elaborado. Sin embargo, en la práctica, no es necesario implementar manualmente estos procedimientos, ya que existen librerías especializadas como `PyTorch` y `TensorFlow`, las cuales automatizan estos cálculos mediante el uso de grafos computacionales.

Vista previa 2026



Figura 2.92 Lenguage de programación empleada en los códigos proporcionados en este libro.

2.9 LIBRERÍAS PARA REDES NEURONALES

Dada la elevada demanda computacional que implica el entrenamiento de redes neuronales, existen diversas librerías de uso libre diseñadas específicamente para este propósito. En este libro utilizaremos el lenguaje de programación `Python`, y en particular dos de las librerías más populares para aprendizaje profundo: `PyTorch` y `TensorFlow`. Ambas ofrecen herramientas de alto nivel para definir arquitecturas de redes, calcular gradientes de manera automática y realizar procesos de optimización. Además, aunque el usuario programa en `Python`, internamente muchas de las operaciones se ejecutan en código compilado (principalmente en `C/C++` y CUDA), lo que permite un desempeño considerablemente más eficiente.

Estas librerías tienen como estructura central el concepto de *tensor*. En este contexto, un tensor puede entenderse como un arreglo n-dimensional que admite operaciones vectorizadas eficientes. Para aprovechar al máximo el rendimiento de estas herramientas, es fundamental que los datos (incluyendo parámetros, variables intermedias e incluso constantes asociadas a un modelo o ecuación diferencial) se representen en formato tensorial. Asimismo, ambas librerías incluyen implementaciones optimizadas de múltiples funciones de activación y componentes estándar, por lo que es recomendable utilizar estas versiones integradas para evitar errores y mejorar la estabilidad numérica.

Una diferencia importante, tanto técnica como filosófica, entre ambas librerías radica en la construcción del grafo computacional, que representa de manera organizada las operaciones que componen el modelo. En `PyTorch`, el grafo es dinámico: se construye en tiempo de ejecución conforme el programa avanza. Esto brinda gran flexibilidad y facilita la experimentación. En cambio, en `TensorFlow` (en su concepción original), el grafo es estático: se define previamente y luego se ejecuta como una estructura ya compilada, lo que favorece la optimización y la escalabilidad en entornos de producción.



Figura 2.93 Principales librerías usadas para entrenar redes neuronales.

En términos prácticos, `PyTorch` suele ser especialmente conveniente para investigación, desarrollo de prototipos y modelos no convencionales, mientras que `TensorFlow` destaca en aplicaciones industriales, despliegue en producción y escalabilidad a gran escala. A lo largo del libro se presentarán ejemplos implementados en ambas librerías, señalando los detalles relevantes cuando sea necesario.

Para profundizar en el funcionamiento específico de objetos, clases, métodos o submódulos, se recomienda consultar la documentación oficial de cada librería. La documentación de `PyTorch` está disponible en `https://docs.pytorch.org/docs/stable/index.html`, mientras que la de `TensorFlow` puede consultarse en `https://www.tensorflow.org/guide`.

Vista previa 2026



Figura 2.94 Entorno en la nube que permite ejecutar códigos en `Python` directamente desde el navegador.

2.9.1 Ejecución en la nube: Google Colab

Para realizar las implementaciones y experimentar con los ejemplos presentados en este libro, el lector necesita contar con un editor o entorno de desarrollo para `Python`. Si se tiene poca experiencia en la instalación y configuración de software, se recomienda utilizar un editor de código en la nube.

Los entornos en la nube ofrecen dos ventajas fundamentales: no es necesario instalar programas en el equipo local y permiten compartir fácilmente el trabajo a través de internet. Entre las opciones disponibles, destacan los servicios gratuitos proporcionados por Google. En particular, *Google Colab* es un entorno de desarrollo integrado (IDE) en la nube que permite ejecutar código `Python` directamente desde el navegador.

Una de sus principales ventajas es que incluye muchas librerías preinstaladas, entre ellas `PyTorch` y `TensorFlow`, lo que facilita trabajar de inmediato sin preocuparse por dependencias o configuraciones adicionales. Por esta razón, se recomienda especialmente a los usuarios que no estén familiarizados con la instalación y administración de entornos de desarrollo locales.

Además, existen tutoriales introductorios y ejemplos de uso de fácil acceso que permiten familiarizarse rápidamente con la plataforma y a desarrollar código de manera práctica. Un ejemplo de la interfaz de programación en Google Colab se muestra en la Figura 2.95, donde puede apreciarse la estructura típica del entorno: el editor de celdas de código, las herramientas de ejecución y los paneles de gestión de archivos y recursos.

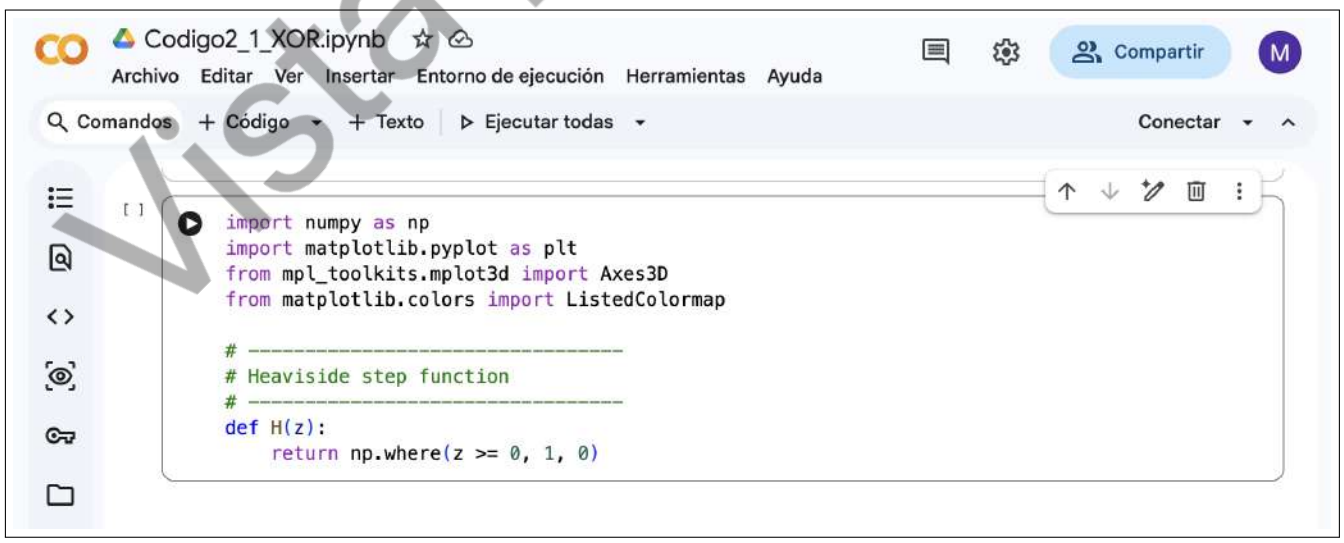


Figura 2.95 Un ejemplo de la interfaz de programación en Google Colab (2026).

Vista previa 2026

RESUMEN

En este capítulo partimos de la unidad más simple de una red neuronal: el perceptrón. A partir de esta pieza elemental fuimos construyendo, paso a paso, redes neuronales multicapa, entendiendo cómo se conectan las neuronas y cómo se incrementa la capacidad del modelo. Además, estudiamos con detalle los elementos que intervienen en el proceso de entrenamiento: la función de pérdida que mide el error, el gradiente que indica la dirección de ajuste y el método de optimización que actualiza los parámetros. De esta manera, no solo describimos la estructura de una red neuronal, sino que comprendimos cómo se determinan sus parámetros y cómo, mediante el entrenamiento, el modelo llega a proporcionar una aproximación a la solución del problema planteado.

Proponer y entrenar una red neuronal, desde el punto de vista técnico, es un proceso relativamente directo: basta con definir la arquitectura (capas y número de neuronas), elegir la función de activación, fijar el número de épocas, seleccionar la tasa de aprendizaje, la función de pérdida y el optimizador. Sin embargo, una pregunta fundamental permanece abierta: ¿es esta arquitectura la más adecuada para el problema en cuestión?

En general, no existe un método determinista que garantice a priori cuál es la mejor arquitectura. El diseño de redes neuronales implica un proceso iterativo de experimentación, en el que se comparan distintas configuraciones y se evalúa su desempeño en términos de precisión, estabilidad y capacidad de generalización.

Además del la capacidad de predicción, deben considerarse factores computacionales como el tiempo de procesamiento, la memoria disponible y el consumo energético. Por ello, en la práctica no solo se busca una red que funcione bien, sino una que sea lo más eficiente posible: un modelo de tamaño adecuado, correctamente entrenado, que satisfaga los requerimientos del problema sin incurrir en un costo computacional innecesario.

En los siguientes capítulos nos adentraremos en problemas específicos donde se ilustrarán, de manera detallada, cada uno de los pasos aquí descritos. El objetivo final será consolidar estos conocimientos para emplear redes neuronales como una herramienta eficaz en la resolución de ecuaciones diferenciales, integrando tanto los fundamentos teóricos como los aspectos computacionales necesarios para su implementación.

PARÁMETROS CLAVE

Al emplear redes neuronales para la solución de problemas, es importante considerar los siguientes parámetros clave:

- **Arquitectura de la red:**
  - Número de capas ocultas.
  - Número de neuronas por capa.
  - Tipo de función de activación.
- **Datos de entrenamiento:**
  - Cantidad de datos disponibles.
- **Función de pérdida (costo):**
  - El error cuadrático medio.
- **Algoritmo de optimización:**
  - Decenso de gradiente.
  - Tasa de aprendizaje.

Vista previa 2026

EJERCICIOS

Realiza los siguientes ejercicios con atención, asegurándote de comprender cada uno de los pasos y de aplicar correctamente los conceptos estudiados. Dedica el tiempo necesario para resolverlos con cuidado y, en caso de surgir dudas, consulta las notas correspondientes para reforzar tu aprendizaje. En el libro de soluciones se pueden encontrar las respuestas correspondientes, así como los códigos utilizados. Sin embargo, se recomienda recurrir a este material únicamente para verificar los resultados obtenidos o en caso de presentar dificultades durante la resolución de los mismos.

Ejercicio 2.1: Visualización del problema XOR. Desarrollar un programa en `Python` que permita visualizar geométricamente la solución del problema XOR obtenida mediante una red neuronal con una capa oculta de dos neuronas. En particular, se considera la arquitectura dada por

$$\begin{aligned} z_1^{[1]} &= x_1 + x_2 - 1.5, & a_1^{[1]} &= H(z_1^{[1]}),\\ z_2^{[1]} &= x_1 + x_2 - 0.5, & a_2^{[1]} &= H(z_2^{[1]}),\\ z_1^{[2]} &= -a_1^{[1]} + a_2^{[1]} - 0.5, & \widetilde{u} &= H(z_1^{[2]}), \end{aligned}$$

donde H es la función tipo escalón definida por

$$H(z) = \begin{cases} 1, & z \geq 0,\\ 0, & z < 0. \end{cases}$$

El objetivo de este ejercicio es que el lector pueda observar de manera explícita cómo una red neuronal muy sencilla es capaz de representar la función XOR mediante la combinación de regiones lineales en el plano de entrada. Asimismo, esta actividad servirá como una primera introducción práctica al uso de `Python` y del entorno de trabajo en Google Colab, permitiendo familiarizarse con la ejecución de código en un entorno interactivo.

El programa deberá incluir, al menos, los siguientes elementos:

- La generación de los cuatro puntos correspondientes al conjunto XOR.
- La evaluación de la salida de la red neuronal para cada uno de dichos puntos.
- Una gráfica tridimensional de la superficie generada por la red neuronal, es decir, de la función $\widetilde{u}(x_1, x_2)$.
- Una gráfica bidimensional en el plano (x_1, x_2) que permita identificar las regiones de clasificación inducidas por la red, de forma análoga a la presentada en el texto.

Ejercicio 2.2: Problema de la paridad de tres bits. Se ha demostrado que un perceptrón no puede resolver el problema XOR debido a que este no es linealmente separable. Existen otras funciones que tampoco pueden ser representadas por un único perceptrón. Un ejemplo clásico es la *función de paridad de tres bits*.

Considere un perceptrón con entradas $x_1, x_2, x_3 \in \{0, 1\}$, pesos $w_1, w_2, w_3 \in \mathbb{R}$, sesgo $b \in \mathbb{R}$ y función de activación tipo escalón. Entonces, la salida es

$$\widetilde{u} = H(w_1x_1 + w_2x_2 + w_3x_3 + b).$$

Sea la *función de paridad* definida como

$$f(x_1, x_2, x_3) = \begin{cases} 1, & \text{si } x_1 + x_2 + x_3 \text{ es impar},\\ 0, & \text{si } x_1 + x_2 + x_3 \text{ es par}. \end{cases}$$

Realiza lo siguiente:

- Construir explícitamente la tabla de verdad de f para los ocho posibles patrones de entrada.
- Identificar cuáles puntos del cubo $\{0, 1\}^3$ pertenecen a la clase 1 y cuáles a la clase 0.
- Interpretar geométricamente el problema en $\mathbb{R}^3$ y analizar si es posible separar ambos conjuntos mediante un único plano.
- Demostrar que no existen pesos w_1, w_2, w_3 y sesgo b tales que el perceptrón satisfaga $\widetilde{u} = f(x_1, x_2, x_3)$ para los ocho patrones de entrada.
- Concluir que la función de paridad de tres bits no es linealmente separable y, por lo tanto, no puede ser implementada exactamente por un solo perceptrón.

Sugerencia: Observe que los puntos con salida 1 y los puntos con salida 0 se alternan en los vértices del cubo unitario, lo que impide que un único hiperplano los separe completamente.

Vista previa 2026

Ejercicio 2.3: Visualización del problema de paridad de tres bits. Siguiendo con el ejercicio anterior, ahora nuestro objetivo es desarrollar un programa en `Python` que permita visualizar geométricamente en la computadora que los puntos correspondientes a la función de paridad no pueden serpararse linealmente mediante un solo hiperplano.

Además, este ejercicio servirá como una introducción práctica al uso de `Python` y del entorno de trabajo en Google Colab. A través de su implementación, el lector podrá familiarizarse la ejecución de código en un entorno interactivo en la nube.

El código debe incluir lo siguiente.

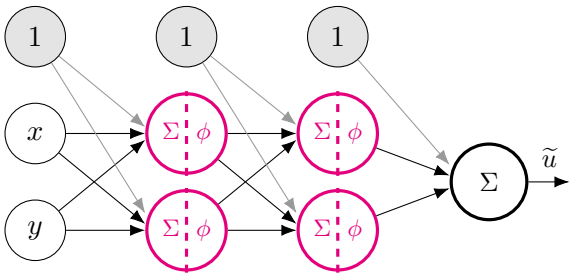
- Generar los ocho puntos del conjunto $\{0,1\}^3$ y evaluar la función de paridad f en cada uno de los ocho puntos.
- Construir una gráfica tridimensional donde:
  - Los puntos con salida $f = 1$ se representen con un color.
  - Los puntos con salida $f = 0$ se representen con otro color.
- Intentar añadir un plano (correspondiente a un perceptrón)

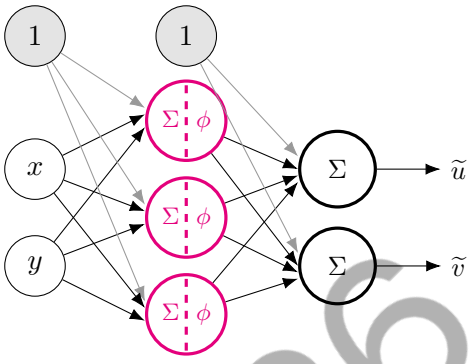
$$w_1x_1 + w_2x_2 + w_3x_3 + b = 0,$$

  y observar que no es posible separar correctamente ambas clases con un único plano.

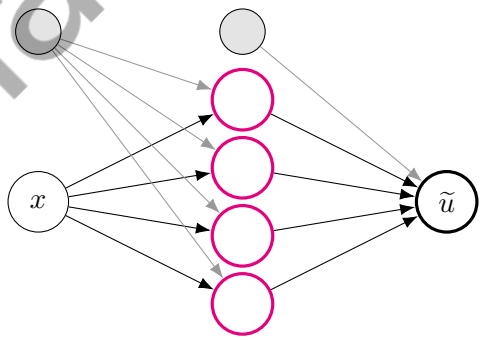
Finalmente, elaborar un breve reporte en el que se describan los principales componentes del programa desarrollado e incluir los resultados gráficos obtenidos del mismo.

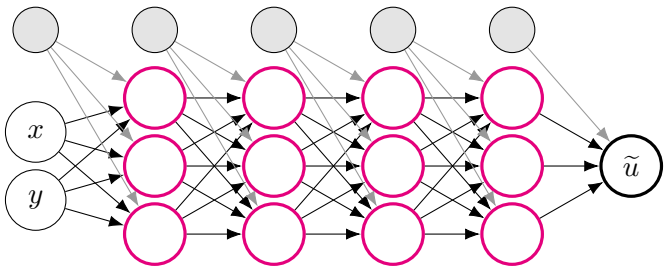
Ejercicio 2.4: Expresión matemática de una red neuronal. Escribir de manera explícita la expresión matemática de las siguientes redes neuronales en términos de sus pesos y sesgos, donde ϕ es la función tangente hiperbólica.

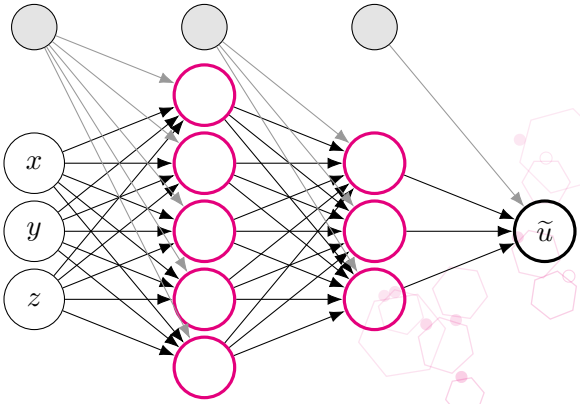




Ejercicio 2.5: Expresión matemática de redes neuronales más complejas. Escribir de manera explícita la expresión matemática de las siguientes redes neuronales en términos de sus pesos y sesgos, donde ϕ es la función de activación general. Ahora usaremos la notación más simplista que estaremos manejando en los siguientes capítulos.







Vista previa 2026

Ejercicio 2.6: Diagrama de una red neuronal. A continuación, consideraremos el problema en sentido inverso. En lugar de comenzar con el diagrama de la red neuronal, se proporcionará directamente su formulación matemática, y a partir de ella se deberá reconstruir la arquitectura correspondiente.

- Considerar primero la red neuronal que representa la siguiente función:

$$
\begin{aligned}
z_1^{[1]} &= 2x_1 - x_2 + 1, & a_1^{[1]} &= \sigma\left(z_1^{[1]}\right),\\
z_2^{[1]} &= -x_1 + 3x_2 - 2, & a_2^{[1]} &= \sigma\left(z_2^{[1]}\right),\\
z_1^{[2]} &= a_1^{[1]} + 2a_2^{[1]} - 1, & a_1^{[2]} &= \sigma\left(z_1^{[2]}\right),\\
\widetilde{u} &= 4a_1^{[2]} + 0.5.
\end{aligned}
$$

  donde σ es la función sigmoide logística.

- Ahora consideremos una segunda red neuronal, un poco más compleja:

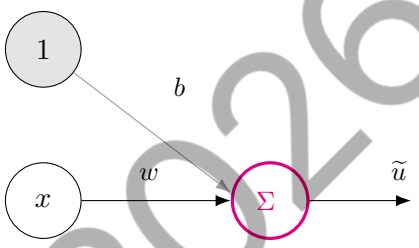
$$
\begin{aligned}
z_1^{[1]} &= 2x_1 - x_2 + 1, & a_1^{[1]} &= \phi\left(z_1^{[1]}\right),\\
z_2^{[1]} &= -x_1 + 3x_2 - 2, & a_2^{[1]} &= \phi\left(z_2^{[1]}\right),\\
z_3^{[1]} &= x_1 + x_2 - 1, & a_3^{[1]} &= \phi\left(z_3^{[1]}\right),\\
z_1^{[2]} &= a_1^{[1]} + 2a_2^{[1]} - a_3^{[1]} + 0.5, & a_1^{[2]} &= \phi\left(z_1^{[2]}\right),\\
z_2^{[2]} &= -2a_1^{[1]} + a_2^{[1]} + 3a_3^{[1]} - 1, & a_2^{[2]} &= \phi\left(z_2^{[2]}\right),\\
\widetilde{u} &= 3a_1^{[2]} - a_2^{[2]} + 0.25.
\end{aligned}
$$

  donde ϕ representa una función de activación no lineal.

A partir de estas representaciones algebraicas, dibujar el diagrama correspondiente indicando las conexiones entre neuronas, así como los pesos y sesgos involucrados en cada capa.

- Además, identificar de manera explícita:
  - el número de entradas de la red,
  - el número de capas ocultas,
  - el número de neuronas en cada capa,
  - los pesos y sesgos asociados a cada neurona,
  - y la función de activación utilizada.

Ejercicio 2.7: Estudio de la tasa de aprendizaje. Utilizando el programa desarrollado para la red neuronal lineal con un solo dato de entrenamiento, analizar la influencia de la tasa de aprendizaje en el comportamiento del algoritmo de descenso de gradiente con los mismos datos proporcionados por el problema.



Para ello, realizar los siguientes experimentos numéricos:

- Ejecutar el programa con distintos valores de la tasa de aprendizaje, por ejemplo:

$$\eta = 0.001, \quad 0.01, \quad 0.1, \quad 0.5, \quad 1.$$

- Para cada valor de η, graficar la evolución de la función de pérdida en términos del número de épocas.

- Analizar el comportamiento de la convergencia en cada caso.

Con base en los resultados obtenidos, responder las siguientes preguntas:

- ¿Qué ocurre cuando la tasa de aprendizaje es muy pequeña?

- ¿Qué sucede cuando la tasa de aprendizaje es demasiado grande?

- ¿En qué casos el método converge más rápidamente?

- ¿Se presentan oscilaciones o divergencia en algún caso? Justificar.

Finalmente, redactar una breve conclusión en la que se describa cómo influye la tasa de aprendizaje en la rapidez y estabilidad del método de descenso de gradiente.

Vista previa 2026

Ejercicio 2.8: Red neuronal lineal con dos datos. Consideremos la siguiente red neuronal lineal con una entrada y una salida:

$$\widetilde{u}(x; w, b) = wx + b,$$

donde w y b representan, respectivamente, el peso y el sesgo de la neurona. Supongamos además que se dispone de dos datos de entrenamiento

$$(\hat{x}_1, \hat{u}_1) = (2.1, 4), \qquad (\hat{x}_2, \hat{u}_2) = (3, 5),$$

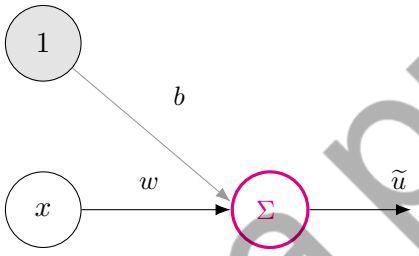
y que la calidad del ajuste se mide mediante la función de pérdida dada por el error cuadrático medio

$$E = \frac{1}{2}\left[(\widetilde{u}(\hat{x}_1; w, b) - \hat{u}_1)^2 + (\widetilde{u}(\hat{x}_2; w, b) - \hat{u}_2)^2\right].$$

Consideremos ahora los parámetros iniciales

$$w^{(0)} = 1, \qquad b^{(0)} = 0,$$

y supongamos que el proceso de entrenamiento se lleva a cabo mediante el método de descenso de gradiente con una tasa de aprendizaje $\eta = 0.01$.



Realizar manualmente una iteración (época) del algoritmo de descenso de gradiente siguiendo los pasos que se indican a continuación:

- Evaluar la salida del modelo en los dos datos de entrenamiento y calcular la función de pérdida inicial.
- Determinar las derivadas parciales de la función de pérdida con respecto a los parámetros w y b.
- Calcular el gradiente total de la función de pérdida, considerando la contribución de ambos datos.
- Actualizar los parámetros utilizando el algoritmo de descenso de gradiente con la tasa de aprendizaje proporcionada.
- Con los nuevos valores de w y b, evaluar nuevamente la red neuronal en ambos datos y calcular la nueva función de pérdida.
- Verificar si la función de pérdida disminuye después de la actualización.

Ejercicio 2.9: Programa de la red neuronal lineal con dos datos Desarrollar un programa en `Python` que implemente el entrenamiento de la siguiente red neuronal lineal con una entrada y una salida con los datos proporcionados en el ejercicio anterior. El programa deberá realizar múltiples iteraciones del entrenamiento y permitir un análisis detallado del comportamiento del método. En particular, se pide lo siguiente:

- Implementar la evaluación de la red neuronal.
- Implementar la función de pérdida.
- Calcular el gradiente con respecto a los parámetros w y b.
- Actualizar los parámetros mediante descenso de gradiente.
- Guardar en cada época los valores de:
  - la función de pérdida,
  - los parámetros w y b,
  - y la salida de la red en los datos de entrenamiento.

Más aún, el programa podría generalizarse para considerar un número arbitrario de datos de entrenamiento proporcionados por el usuario, lo que permitiría extender de manera natural el procedimiento presentado hasta ahora. Aquí un ejemplo del esqueleto del mismo:

```
# Evualuacion inicial
yhat = forward(x, w, b)
loss = mse(yhat, y)

# Entrenamiento
for epoch in range(epochs):
    # Calculo del gradientes
    dw, db = backward(x, y, yhat)

    # Actualizacion de parametros
    w, b = update(w, b, dw, db, lr)

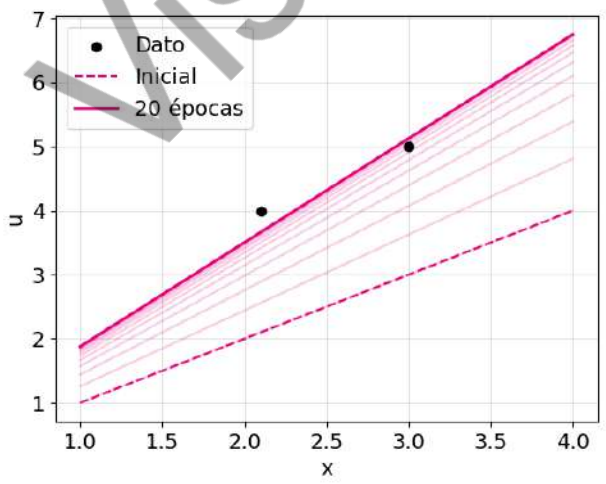
    # Evaluacion
    yhat = forward(x, w, b)
    loss = mse(yhat, y)
```

Vista previa 2026

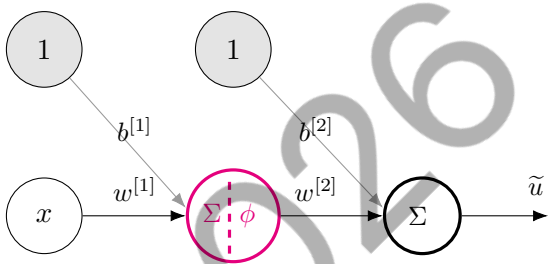
Ejercicio 2.10: Análisis de la red neuronal lineal con dos datos. A partir de la implementación del programa anterior, realizar el siguiente análisis:

- Mostrar los primeros valores de la función de pérdida y verificar que las primeras iteraciones coinciden con los cálculos realizados manualmente.
- Graficar la evolución de la función de pérdida en función del número de épocas, tanto en escala lineal como en escala logarítmica.
- Graficar la evolución de los parámetros w y b a lo largo del entrenamiento.
- Representar geométricamente el proceso de ajuste, graficando:
  - los dos datos de entrenamiento,
  - la recta inicial,
  - varias rectas intermedias durante el entrenamiento,
  - y la recta final obtenida después de la convergencia.
- Analizar si la red neuronal converge a una recta que aproxime adecuadamente los datos proporcionados.
- Discutir cómo cambia el comportamiento del entrenamiento al pasar de un solo dato a dos datos de entrenamiento.

En la siguiente curva podemos observar los datos, la recta inicial y la recta final después de 100 épocas.



Ejercicio 2.11: Programa de una red neuronal con dos capas y una neurona. Considerar el programa desarrollado en este capítulo para la una red neuronal de dos capas de una sola neurona cada una, donde la primera capa utiliza la función de activación tanh y la segunda es lineal.

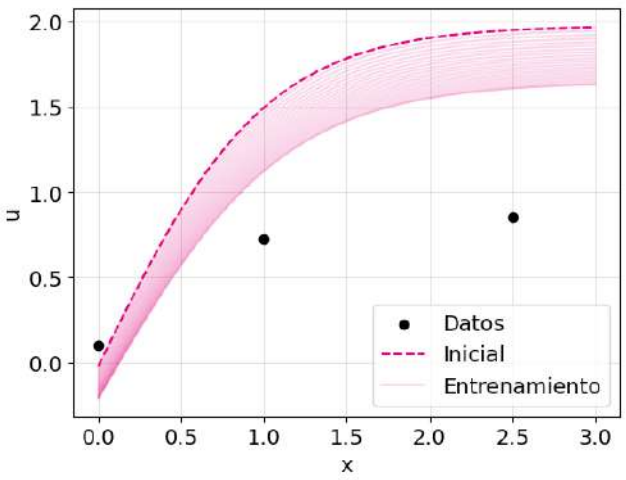


Para ilustrar mejor la relación entre la arquitectura de la red neuronal y la forma de los datos, considerar dos nuevos conjuntos de entrenamiento:

- $(0, 0.10)$, $(1, 0.72)$, $(2.5, 0.8)$.
- $(0, 0)$, $(1, 1.2)$, $(3, 1.2)$.

A partir de estas modificaciones al programa, analizar el comportamiento del entrenamiento y comparar los resultados con los dos conjuntos de datos. En particular, discutir la evolución de la función de pérdida, la estabilidad del entrenamiento, el comportamiento de los parámetros y la calidad del ajuste final de la red neuronal.

En la siguiente figura podemos observar los datos, la curva inicial y su evolución durante 20 épocas del primer conjunto de datos.



Vista previa 2026

Ejercicio 2.12: Red neuronal lineal con dos entradas. Consideremos la siguiente red neuronal lineal con una entrada y una salida:

$$\widetilde{u}(x, y; w, b) = w_1 x + w_2 y + b,$$

donde w_1, w_2 y b representan, respectivamente, los pesos y el sesgo de la neurona. Supongamos además que se dispone de un único dato de entrenamiento

$$(\hat{x}_1, \hat{y}_1, \hat{u}_1) = (3, 2, 0),$$

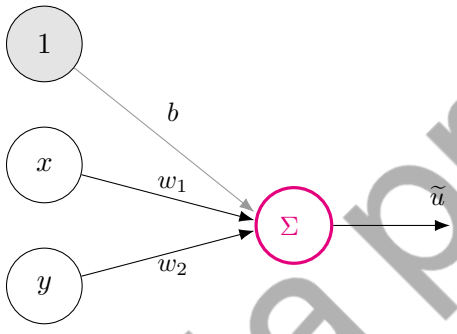
y que la calidad del ajuste se mide mediante la función de pérdida dada por el error cuadrático medio

$$E = \left(\widetilde{u}(\hat{x}_1, \hat{y}_1; w_1, w_2, b) - \hat{u}_1\right)^2.$$

Consideremos ahora los parámetros iniciales

$$w_1^{(0)} = 0.5, \quad w_2^{(0)} = 1, \quad b^{(0)} = 1,$$

y supongamos que el proceso de entrenamiento se lleva a cabo mediante el método de descenso de gradiente con una tasa de aprendizaje $\eta = 0.01$.



Realizar manualmente una época usando el descenso de gradiente siguiendo los pasos que se indican a continuación:

- Evaluar la salida del modelo en el dato de entrenamiento y calcular la función de pérdida inicial.
- Determinar las derivadas parciales de la función de pérdida con respecto a los parámetros w_1, w_2 y b.
- Calcular el gradiente total de la función de pérdida.
- Actualizar los parámetros utilizando el algoritmo de descenso de gradiente con la tasa de aprendizaje proporcionada.
- Con los nuevos valores de w_1, w_2 y b, evaluar nuevamente la red neuronal y calcular la nueva función de pérdida.
- Verificar si la función de pérdida disminuye después de la actualización.

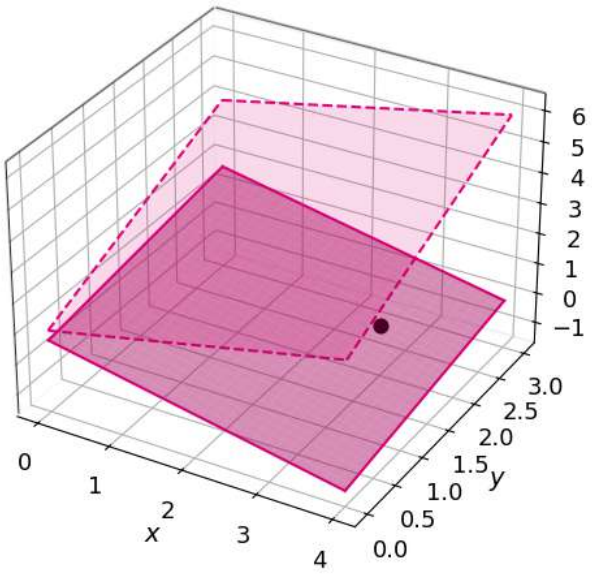
Ejercicio 2.13: Programa de una red neuronal lineal con dos entradas. Considere la red neuronal lineal con dos entradas y una salida dada en el ejercicio anterior. El objetivo de este ejercicio es desarrollar un programa que permita automatizar el proceso de entrenamiento de esta red neuronal y visualizar geométricamente la evolución de la solución conforme avanzan las épocas.

El programa debe incluir, al menos, los siguientes elementos:

- Definir funciones o rutinas que permitan:
  - evaluar la salida de la red neuronal,
  - calcular la función de pérdida,
  - determinar el gradiente con respecto a w_1, w_2 y b,
  - y actualizar los parámetros mediante descenso de gradiente.
- Ejecutar el entrenamiento durante un número suficiente de épocas (por ejemplo, 100 o 200).
- Almacenar en vectores los valores de la función de pérdida, los parámetros w_1, w_2 y b y la salida aproximada $\widetilde{u}$ en cada época.
- Graficar la evolución de la función de pérdida en términos del número de épocas.
- Graficar la evolución de los parámetros w_1, w_2 y b durante el entrenamiento.
- Realizar una visualización geométrica del proceso de entrenamiento, representando:
  - el dato de entrenamiento $(\hat{x}_1, \hat{y}_1, \hat{u}_1)$,
  - el plano inicial asociado a los parámetros iniciales,
  - algunos planos intermedios,
  - y el plano final obtenido después del entrenamiento.

En la siguiente figura podemos observar el dato conocido, el plano inicial y el plano al final después de 50 épocas de entrenamiento.

Vista previa 2026



Finalmente, redacte un breve análisis de los resultados obtenidos, en el que se describa el comportamiento de la función de pérdida, la evolución de los parámetros, y la interpretación geométrica del ajuste realizado por la red neuronal.

Ejercicio 2.14: Red neuronal lineal con una entrada y dos salidas. Consideremos la siguiente red neuronal lineal con una entrada y una salida:

$$\begin{aligned} \widetilde{u}(x;w,b) &= w_1 x + b_1, \\ \widetilde{v}(x;w,b) &= w_2 x + b_2, \end{aligned}$$

donde w_1, b_1 y w_2, b_2 representan, respectivamente, los pesos y sesgos de la red neuronal. Supongamos además que se dispone de un dato de entrenamiento con dos salidas:

$$(\hat{x}_1, \hat{u}_1, \hat{v}_1) = (1, 2, -2),$$

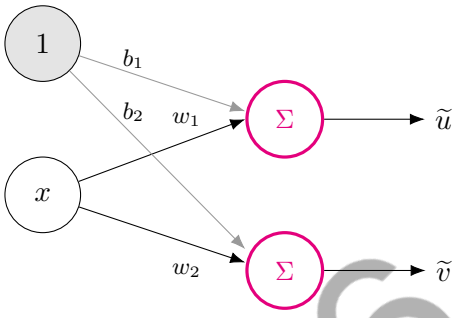
y que la calidad del ajuste se mide mediante la función de pérdida dada por

$$\begin{aligned} E(w_1, w_2, b_1, b_2) &= \frac{1}{2}\left(\widetilde{u}(\hat{x}_1; w_1, b_1) - \hat{u}_1\right)^2 \\ &+ \frac{1}{2}\left(\widetilde{v}(\hat{x}_1, w_2, b_2) - \hat{v}_1\right)^2. \end{aligned}$$

Consideremos ahora los parámetros iniciales

$$w_1^{(0)} = 1, \quad b_1^{(0)} = 0, \quad w_2^{(0)} = 2, \quad b_2^{(0)} = -2$$

y supongamos que el proceso de entrenamiento se lleva a cabo mediante el método de descenso de gradiente con una tasa de aprendizaje $\eta = 0.01$.



Realizar manualmente una época usando el descenso de gradiente y el algoritmo de retropropagación. Basar este procedimiento en los mismos pasos descritos en ejercicios anteriores.

Ejercicio 2.15: Programa de una red neuronal lineal con dos salidas. Considere la red neuronal lineal con una entrada y dos salidas $\widetilde{u}$ y $\widetilde{v}$ presentada en el ejercicio anterior donde w_1, b_1, w_2 y b_2 representan los parámetros del modelo. Utilizar los mismos datos conocidos para el entrenamiento y los parámetros iniciales planteados anteriormente.

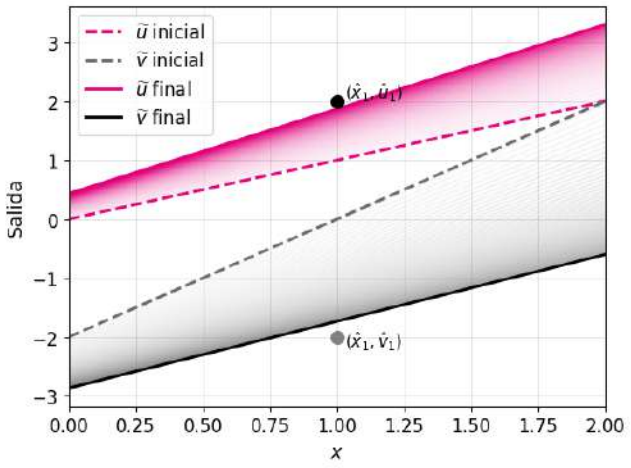
Desarrolle un programa, preferentemente en Python, que permita automatizar el proceso de entrenamiento de esta red neuronal. El programa debe incluir, al menos, lo siguiente:

- Evaluar las salidas $\widetilde{u}$ y $\widetilde{v}$ del modelo.
- Calcular la función de pérdida en cada época.
- Determinar el gradiente de E con respecto a los parámetros w_1, w_2, b_1 y b_2.
- Actualizar los parámetros utilizando descenso de gradiente.
- Ejecutar el entrenamiento durante varias épocas.
- Graficar la evolución de la función de pérdida.
- Graficar la evolución de los parámetros durante el entrenamiento.
- Visualizar la evolución de las rectas $\widetilde{u}(x)$ y $\widetilde{v}(x)$, mostrando sus estados inicial, intermedios y final.

Vista previa 2026

Finalmente, redacte un breve análisis en el que describa el comportamiento del error, la evolución de los parámetros y la interpretación geométrica del entrenamiento.

En la siguiente figura podemos observar el dato conocido, las rectas iniciales y las rectas finales después de 100 épocas de entrenamiento.



Ejercicio 2.16: Red neuronal lineal con dos entradas y dos salidas. Consideremos la siguiente red neuronal lineal con una entrada y una salida:

$$\begin{aligned} \tilde{u}(x, y; w, b) &= w_{11}x + w_{12}y + b_1, \\ \tilde{v}(x, y; w, b) &= w_{21}x + w_{22}y + b_2, \end{aligned}$$

donde w_{11}, w_{12}, b_1 y w_{21}, w_{22}, b_2 representan, respectivamente, los pesos y sesgos de la red neuronal. Supongamos además que se dispone de un solo dato de entrenamiento con dos salidas

$$(\hat{x}_1, \hat{y}_1, \hat{u}_1, \hat{v}_1) = (1, 2, 2, 1),$$

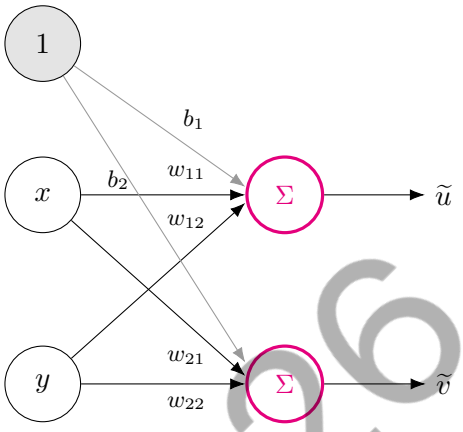
y que la calidad del ajuste se mide mediante la función de pérdida dada por

$$\begin{aligned} E &= \frac{1}{2}(\tilde{u}(\hat{x}_1, \hat{y}_1; w_{11}, w_{12}, b_1) - \hat{u}_1)^2 \\ &+ \frac{1}{2}(\tilde{v}(\hat{x}_1, \hat{y}_1; w_{21}, w_{22}, b_2) - \hat{v}_1)^2. \end{aligned}$$

Consideremos ahora los parámetros iniciales

$$w_{11}^{(0)} = 1, \quad w_{12}^{(0)} = -1, \quad b_1^{(0)} = 0,$$
$$w_{21}^{(0)} = 2, \quad w_{22}^{(0)} = 1, \quad b_2^{(0)} = -2$$

y supongamos que el proceso de entrenamiento se lleva a cabo mediante el método de descenso de gradiente con una tasa de aprendizaje $\eta = 0.01$.



Realizar manualmente una época usando el descenso de gradiente y retropropagación usando los mismos pasos descritos en ejercicios anteriores.

Ejercicio 2.17: Programa de una red neuronal lineal con dos entradas y dos salidas. Considere la red neuronal descrita en el ejercicio anterior. Desarrolle un programa, usando el mismo datos conocido y los mismos parámetros iniciales, que permita automatizar el proceso de entrenamiento de esta red neuronal. El programa debe incluir, al menos, lo siguiente:

- Evaluar las salidas $\tilde{u}$ y $\tilde{v}$ del modelo.
- Calcular la función de pérdida en cada época.
- Determinar el gradiente de E con respecto a todos los parámetros del modelo.
- Actualizar los parámetros utilizando descenso de gradiente.
- Ejecutar el entrenamiento durante varias épocas.
- Graficar la evolución de la función de pérdida.
- Graficar la evolución de los parámetros durante el entrenamiento.
- Visualizar la evolución de los planos $\tilde{u}(x, y)$ y $\tilde{v}(x, y)$, mostrando sus estados inicial y final en una misma gráfica.

Finalmente, redacte un breve análisis en el que describa el comportamiento del error y la evolución de los parámetros durante el entrenamiento.

Vista previa 2026

Vista previa 2026

CAPÍTULO 3

AJUSTE DE CURVAS USANDO REDES NEURONALES

Antes de adentrarnos directamente en la resolución de ecuaciones diferenciales mediante redes neuronales, resulta conveniente presentar un ejemplo sencillo de una red neuronal multicapa. Con este ejemplo introductorio se busca familiarizar al lector con la notación, los componentes estructurales y el funcionamiento básico de este tipo de modelos. En particular, se abordará cómo se representan las capas, las funciones de activación, los parámetros de la red (pesos y sesgos) y el proceso de optimización que permite generar predicciones precisas.

El ajuste de curvas a un conjunto de datos conocidos es el ejemplo seleccionado. Este ejemplo es especialmente interesante en la comunidad científica debido a sus múltiples aplicaciones. Iniciaremos explicando brevemente métodos clásicos de solución, como la interpolación y los mínimos cuadrados, para posteriormente abordar el método basado en redes neuronales y comparar los resultados obtenidos. Este ejemplo, muestra la idea básica y que sirve de base conceptual para comprender, más adelante, cómo se adaptan y extienden estas redes al contexto de problemas de ecuaciones diferenciales.

Vista previa 2026

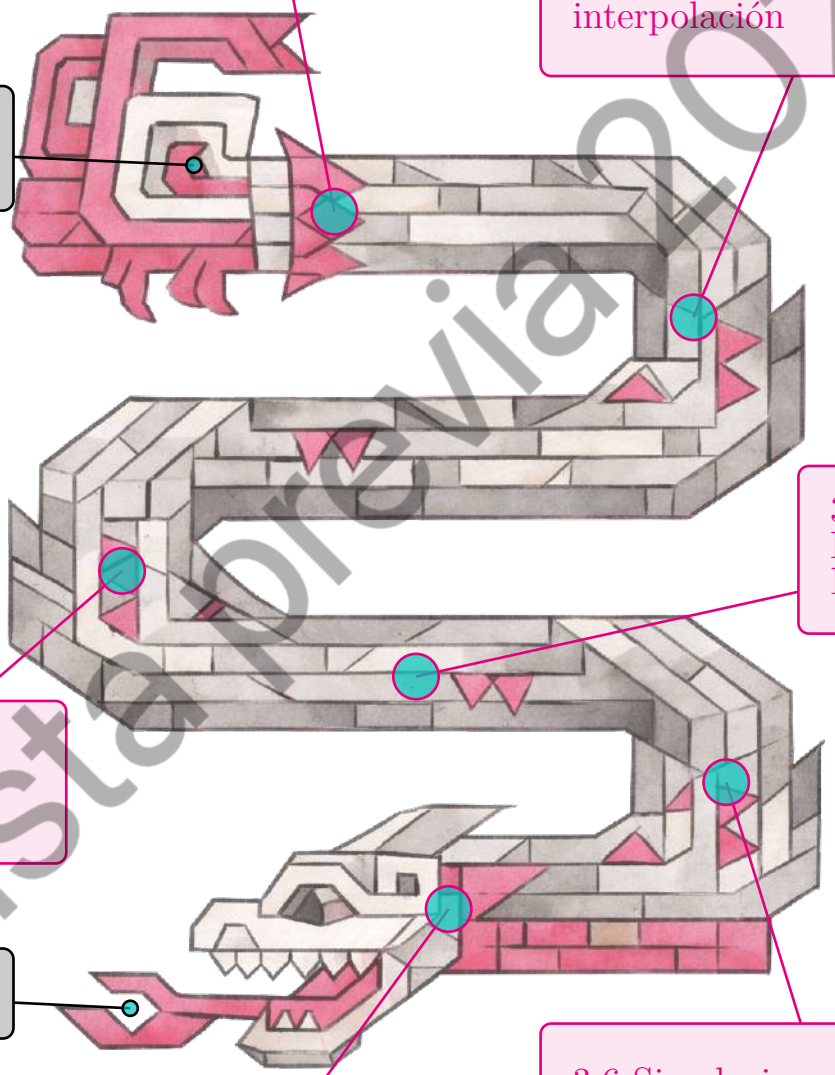
OBJETIVOS

Al finalizar este capítulo, el lector será capaz de:

- Describir la estructura de redes neuronales multicapa, comprendiendo la notación, las capas, las funciones de activación y los parámetros fundamentales (pesos y sesgos).
- Comprender el proceso de optimización en redes neuronales, identificando cómo se ajustan los parámetros de la red para generar predicciones precisas.
- Comparar métodos clásicos y basados en redes neuronales para el ajuste de curvas, evaluando las técnicas tradicionales como interpolación y mínimos cuadrados frente a los resultados obtenidos mediante redes neuronales.
- Analizar ejemplos sencillos de aplicación de redes neuronales con el fin de reforzar los conceptos teóricos presentados.

CONTENIDO

3.1 Introducción
3.2 El ejemplo de interpolación
3.3 Métodos clásicos de interpolación
3.4 Interpolación con redes neuronales
3.5 Implementación en Python y PyTorch
3.6 Simulaciones usando redes neuronales
3.7 Ajuste de una curva a datos aleatorios
Resumen



Vista previa 2026



Figura 3.1 Los censos proporcionan mediciones precisas del tamaño poblacional, aunque únicamente en intervalos de tiempo específicos.

3.1 INTRODUCCIÓN

El problema de ajustar curvas a un conjunto de datos es fundamental en la ciencia y la ingeniería, porque permite describir, entender y predecir el comportamiento de distintos fenómenos a partir de información observada, ya sea experimental o generada por simulaciones.

En física y química se utiliza para caracterizar relaciones entre variables experimentales; en ingeniería, para diseñar sistemas de control y calibrar sensores; en economía y finanzas, para modelar tendencias de mercado; y en biología y medicina, para analizar respuestas fisiológicas o la concentración de fármacos en el organismo, lo que es clave para evaluar tratamientos, comprender fenómenos biológicos y tomar decisiones basadas en evidencia cuantitativa.

> En este contexto, cuando hablamos de una curva, nos referimos generalmente a la gráfica de una función real valuada, es decir, una función que asigna a cada valor de entrada un único valor de salida. Así, el objetivo del ajuste es encontrar una función cuya gráfica pase lo más cerca posible de los puntos dados. Para medir qué tan buena es esta aproximación, se define un error entre los valores que predice la función y los datos observados, el cual se cuantifica utilizando la idea de distancia en matemáticas.

Un ejemplo relevante del uso del ajuste de curvas se encuentra en el estudio del crecimiento poblacional, el cual se basa en los datos proporcionados por censos realizados en intervalos de tiempo discretos. Por ejemplo en la Figura 3.2 podemos observar la evolución del crecimiento demográfico de la ciudad de Mérida en el período 1950–2011. Notar que solo contamos con un número reducido de datos durante esos 60 años trascurridos. Sin embargo, a partir de estas mediciones, podemos construir modelos matemáticos que permiten interpolar los valores entre censos y extrapolar tendencias futuras, como el crecimiento, la estabilización o el decrecimiento de una población. Este tipo de análisis es fundamental en demografía, economía y planeación urbana, ya que facilita la estimación de demandas futuras de recursos, como vivienda, infraestructura, servicios de salud y educación.



Figura 3.2 Evolución del crecimiento demográfico de la ciudad de Mérida. Período 1950 – 2011.

No solo es posible ajustar curvas. Cuando aumenta el número de parámetros o variables involucradas, también se pueden ajustar superficies a los datos. En este caso, la función ya no depende de una sola variable, sino de varias, es decir, se trata de una función real valuada de múltiples variables. Aunque la complejidad del problema aumenta, la idea fundamental sigue siendo la misma: encontrar una función que se aproxime lo mejor posible a los datos disponibles. Sin embargo, los cálculos se vuelven más elaborados, y este incremento en complejidad resulta esencial para modelar fenómenos más realistas en diversas aplicaciones.

Vista previa 2026

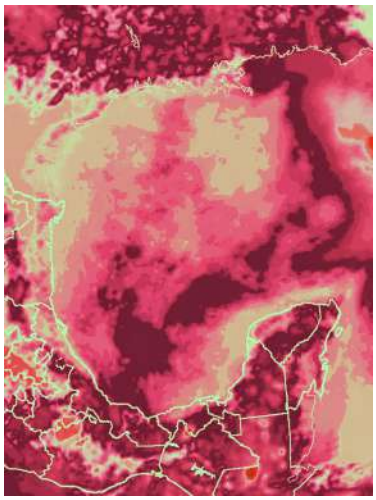


Figura 3.3 Mapa de temperaturas del Golfo de México y del territorio nacional, generado a partir de datos medidos en estaciones meteorológicas.

Un ejemplo concreto de la importancia de ajustar superficies a datos se encuentra en la elaboración de mapas de temperatura. Aquí, las mediciones se toman en estaciones meteorológicas distribuidas en el territorio, y luego se utilizan técnicas de interpolación y ajuste de superficies para extrapolar esos valores y obtener un mapa completo de temperaturas que represente todo un territorio como la mostrada en la Figura 3.3 para la región del Golfo de México. Este tipo de métodos permite visualizar tendencias, detectar anomalías y realizar predicciones espaciales, demostrando la utilidad en aplicaciones reales.

En este capítulo nos centraremos en el ajuste de curvas a partir de un conjunto de datos unidimensionales. En las siguientes subsecciones abordaremos este problema con mayor detalle, analizando los fundamentos y procedimientos necesarios para aproximar los datos mediante diferentes modelos.

3.1.1 Herramientas matemáticas

Una de las técnicas clásicas para ajustar curvas a un conjunto de datos es la interpolación.

DEFINICIÓN
La interpolación de puntos consiste en construir una función que pase exactamente por cada uno de los puntos conocidos del conjunto de datos.

Entre los métodos más comunes se encuentran la interpolación lineal, que conecta los puntos con segmentos rectos; la interpolación polinómica, que ajusta un polinomio de grado adecuado a todos los datos; y la interpolación spline, que utiliza polinomios por tramos para obtener curvas suaves y continuas. En la Figura 3.4 podemos observar un ejemplo de los métodos de interpolación más comunes para ajustar curvas a un mismo conjunto de datos.

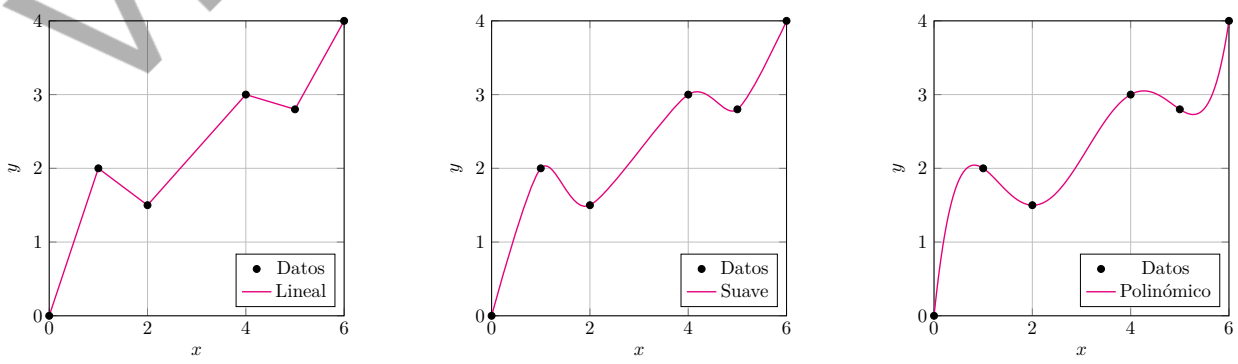


Figura 3.4 Métodos de interpolación más comunes para ajustar curvas a un conjunto de datos.

Vista previa 2026

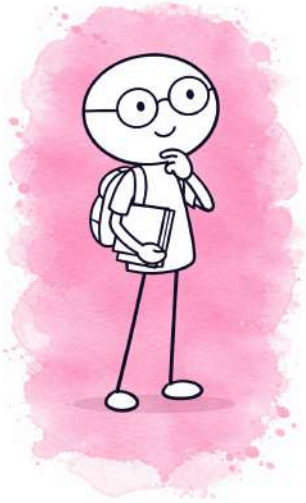


Figura 3.5 Será interesante analizar el comportamiento de las redes neuronales en distintos escenarios, tanto en aquellos donde la interpolación resulte más adecuada como en aquellos donde sea preferible emplear un ajuste por mínimos cuadrados.

La interpolación es particularmente útil cuando se desea estimar valores intermedios entre puntos de medición o analizar la tendencia local de los datos sin introducir modelos demasiado complejos. Sin embargo, cuando los conjuntos de datos son grandes o presentan un nivel significativo de ruido, las técnicas de interpolación pueden volverse inestables o producir oscilaciones no deseadas. Esto motiva el uso de métodos más flexibles y robustos, como los basados en mínimos cuadrados tal como se muestra en la Figura 3.6.

Con respecto a los métodos clásicos, el método de mínimos cuadrados consiste en determinar, dentro de una familia de funciones dada, los coeficientes que minimizan la suma de los cuadrados de los residuos entre los datos observados y el modelo aproximante. Fue publicado por primera vez por Legendre en 1805 y posteriormente justificado desde un punto de vista estadístico por Gauss en 1809. A partir de estas contribuciones, el método se consolidó como una herramienta fundamental en el análisis de datos. Su desarrollo numérico moderno se dio a finales de la década de 1960 con la introducción de la descomposición QR mediante transformaciones de Householder por Golub. De ahí se siguen desarrollando nuevas metodologías, siendo este campo de investigación uno de los más desafiantes dentro del análisis numérico.

En el remanente de este capítulo se realizará un breve repaso de las técnicas clásicas de interpolación, las cuales permiten comprender los fundamentos del ajuste de funciones a partir de datos y sirven como punto de partida para el análisis de métodos de aproximación. Posteriormente, se explorará el uso de redes neuronales para el tratamiento del mismo problema. Finalmente, se abordará el método de mínimos cuadrados y redes neuronales, a problemas más desafiantes, donde la complejidad de los datos hacen que los enfoques tradicionales resulten insuficientes.

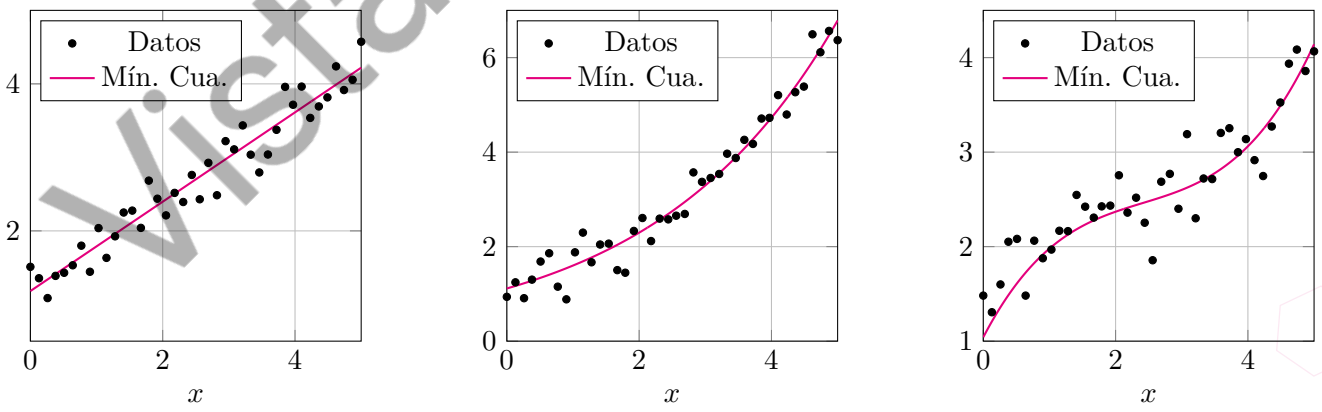


Figura 3.6 Ejemplos de ajuste de curvas a un conjunto de datos mediante LSM. En el primer caso tenemos una función linea, en el segundo caso una función exponencial y en el tercer caso un polinomio de tercer orden.

Vista previa 2026

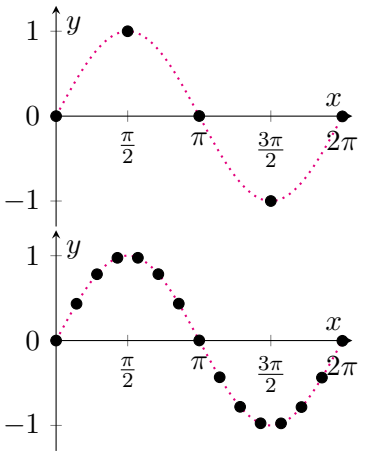


Figura 3.7 Ejemplo de 5 y 15 observaciones $\hat{u}_i$ correspondientes al muestreo de la función seno.

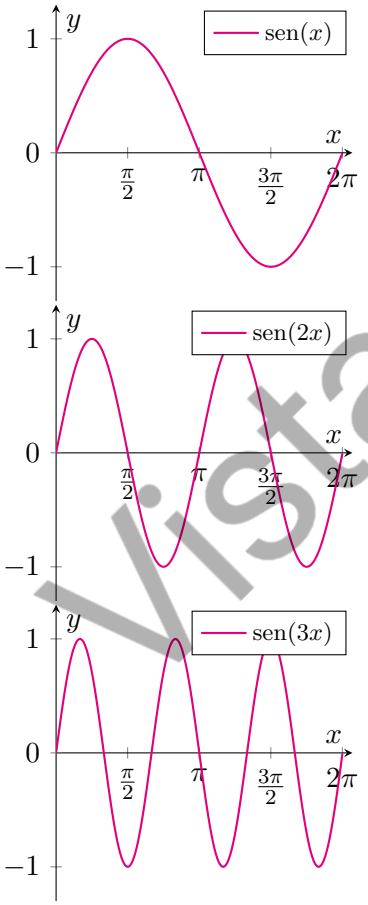


Figura 3.8 Gráfica de la función seno con diferentes valores m.

3.2 EL EJEMPLO DE INTERPOLACIÓN

La idea es comenzar con un ejemplo sencillo de interpolación que permita al lector seguir, de manera clara y ordenada, cada uno de los pasos necesarios para implementar una metodología de aproximación mediante redes neuronales artificiales. Con este objetivo, se generará un conjunto de datos a partir de una función conocida y se evaluará cómo, a partir de estos datos, el modelo es capaz de recuperar la función original.

EL PROBLEMA

Sea x una variable independiente y sea u una función real desconocida, cuyo valor en x se denota por $u(x)$ y que se desea aproximar. Para este propósito, supóngase que se dispone de N observaciones

$$(x_1, \hat{u}_1), (x_2, \hat{u}_2), \ldots, (x_N, \hat{u}_N), \tag{3.38}$$

donde $\hat{u}_i$, para $i = 1, 2, \ldots, N$, representan mediciones de $u(x_i)$.

EL EJEMPLO

A lo largo de esta sección, y a menos que se especifique lo contrario, se utilizará como ejemplo base la función definida por:

$$u(x) = \text{sen}(mx), \quad x \in [0, 2\pi], \tag{3.39}$$

donde m representa un número entero que indica la frecuencia de la onda. Inicialmente, se considerarán el valor de $m = 1$. Entonces x_i es un valor dentro del dominio de interés y $\hat{u}_i = \sin(x_i)$ es el valor correspondiente de la muestra.

El número de muestras es obtenido inicialmente de N puntos equiespaciados en el intervalo $[0, 2\pi]$ de la forma:

$$x_i = 2\pi(i-1)/(N-1), \quad \text{para} \quad i = 1, 2, \ldots, N.$$

La Figura 3.7 presenta dos ejemplos correspondientes a 5 y 15 observaciones $\hat{u}_i$, obtenidas mediante el muestreo de $\text{sen}(x)$ en el mismo número de puntos equidistantes x_i. Como puede observarse, cuando el número de puntos es reducido no resulta evidente que sea posible recuperar de manera unívoca la función original, ya que distintos modelos podrían interpolar adecuadamente los mismos datos discretos, sin coincidir necesariamente con la función que los generó.

Esta ambigüedad se vuelve aún más pronunciada al considerar casos más desafiantes, por ejemplo, al incrementar la frecuencia de la función subyacente, situación que se ilustra en la Figura 3.8. En estos escenarios, la limitada información proporcionada por un muestreo escaso puede conducir a interpretaciones erróneas del comportamiento real de la función, lo que pone de manifiesto la necesidad de métodos de aproximación robustos y flexibles.

Vista previa 2026

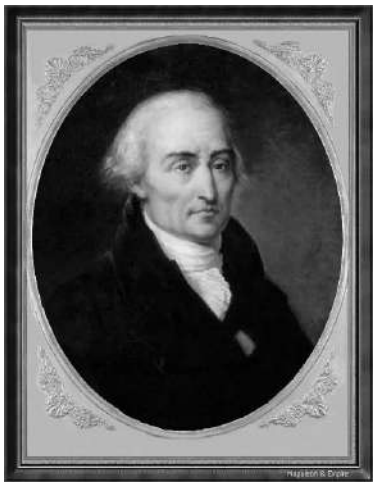


Figura 3.9 Retrato de Joseph-Louis Lagrange.

3.3 MÉTODOS CLÁSICOS DE INTERPOLACIÓN

Comprender los resultados obtenidos mediante métodos clásicos, como la interpolación de Lagrange y los splines cúbicos, resulta fundamental para evaluar y validar la eficacia de los modelos basados en redes neuronales que se estudiarán posteriormente. Estos métodos tradicionales cuentan con una sólida base teórica y han demostrado un desempeño confiable en la aproximación de funciones suaves, por lo que constituyen un punto de referencia natural para comparar enfoques más modernos.

A continuación, se presentan de manera resumida los métodos de interpolación de Lagrange y de splines cúbicos. Ambos pertenecen al núcleo del análisis numérico clásico y han sido ampliamente estudiados, lo que ha dado lugar a una extensa bibliografía al respecto. Para una exposición clara y detallada de estos métodos puede consultarse, por ejemplo, el libro de Cleve Moler [52]. En ambos casos se proporcionan implementaciones en `Python` que permiten realizar la interpolación utilizando N puntos equidistantes de la función seno.

3.3.1 Interpolación usando polinomios de Lagrange

El método de Lagrange es una herramienta fundamental para aproximar funciones conocidas en puntos discretos, permitiendo estimar su comportamiento en otros valores del dominio. Este construye un polinomio de grado $N-1$ que interpola exactamente un conjunto de N puntos dados. El polinomio de interpolación de Lagrange se expresa de la siguiente forma:

$$P(x) = \sum_{j=0}^{N} \hat{u}_j \ell_j(x), \quad \text{donde} \quad \ell_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{N} \frac{x - x_i}{x_j - x_i}.$$

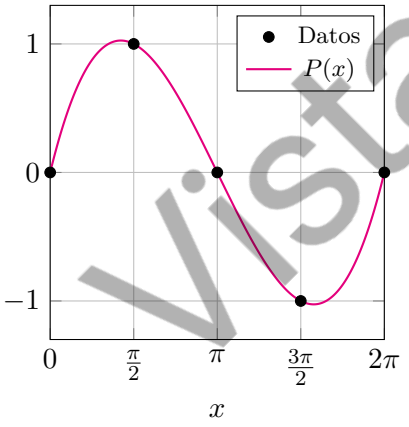


Figura 3.10 Si tomamos cinco puntos equidistantes de $\text{sen}(x)$ en el intervalo $[0, 2\pi]$, entonces el polinomio cúbico $P(x) = \frac{8}{3\pi^3}\, x\,(x-\pi)\,(x-2\pi)$ pasa exactamente por estos puntos.

Este polinomio cumple que $P(x_i) = \hat{u}_i$ para todos los nodos de interpolación, es decir el polinomio considera de manera exacta los valores en estos puntos.

La implementación de este método en *Python* es directa, ya que el módulo `scipy.interpolate` proporciona la siguiente rutina que construye automáticamente el polinomio de interpolación de Lagrange a partir del conjunto de datos dado.:

```
lagrange(x_nodes, u_nodes)
```

El parámetro `x_nodes` es un arreglo que contiene las coordenadas x_i de los *nodos de interpolación*, es decir, los puntos donde se conoce el valor de la función. Por su parte, `u_nodes` es un arreglo que contiene los valores correspondientes $\hat{u}_i$ para cada x_i. La función retorna un objeto de tipo `numpy.poly1d`, que representa un polinomio que interpola exactamente todos los puntos $(x_i, \hat{u}_i)$.

Vista previa 2026

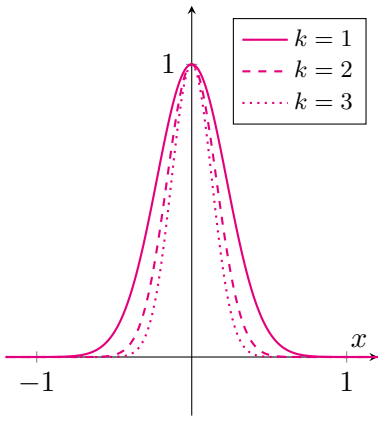


Figura 3.11 Otra función de interés se obtiene al considerar $\exp(-k\pi^2 x^2)$ para distintos valores de k. Esta familia corresponde a funciones gaussianas (de tipo campana), cuya forma se vuelve progresivamente más estrecha y pronunciada a medida que aumenta el valor de k.

```python
import numpy as np
from scipy.interpolate import lagrange

# Definicion de la funcion exacta
def u(x):
    return np.sin(x)

# Puntos equidistantes de interpolacion
N = 10  # numero de puntos
x_nodes = np.linspace(0, 2*np.pi, N)
u_nodes = u(x_nodes)

# Polinomio de Lagrange
poly_lagrange = lagrange(x_nodes, u_nodes)

# Evaluacion de la aproximacion
x_plot = np.linspace(0, 2*np.pi, 100)
u_exact = u(x_plot)
u_interp = poly_lagrange(x_plot)
```

Código 3.4 Implementación en *Python* para una interpolación de N puntos usando polinomios de Lagrange.

La parte superior de la Figura 3.12 muestra los resultados numéricos obtenidos para la función seno al considerar distintos números de puntos de interpolación. Aunque las aproximaciones aparentan mejorar conforme aumenta el número de datos, este comportamiento corresponde a un caso particular asociado a la función seleccionada. En contraste, al aplicar el mismo procedimiento a la función definida por $\exp(-\pi^2 x^2)$, los resultados presentados en la parte inferior de la figura evidencian el comportamiento típico de la interpolación de Lagrange: cuando el número de puntos es grande, este método tiende a presentar inestabilidades numéricas y oscilaciones no deseadas.

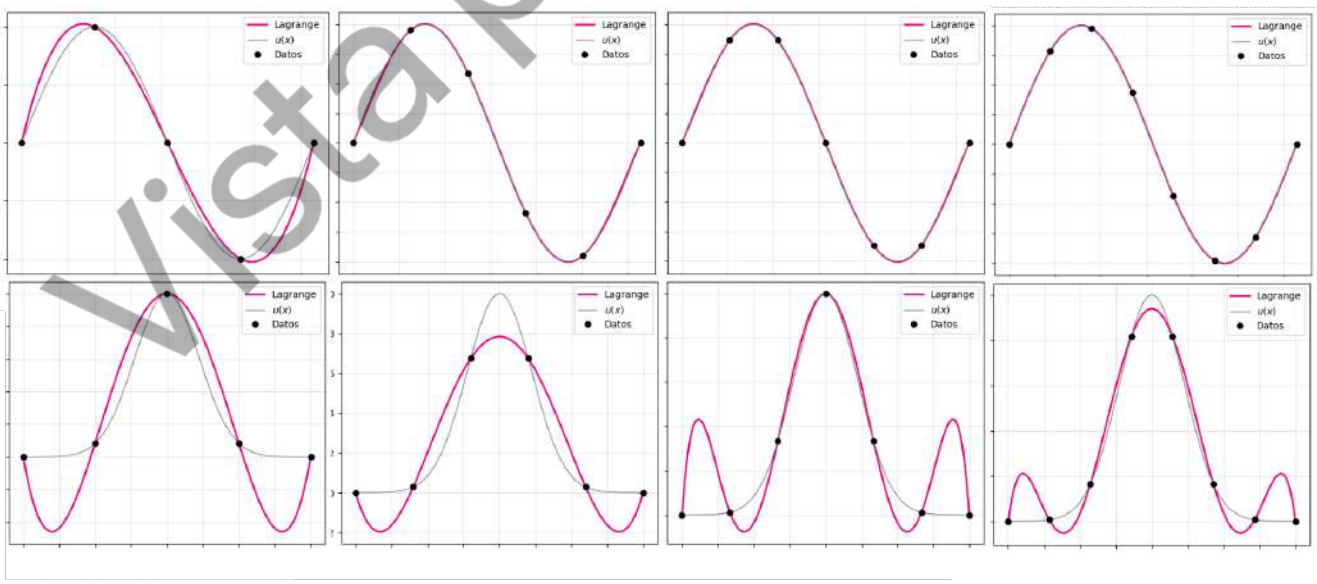


Figura 3.12 Resultados del método de interpolación de Lagrange aplicados a las funciones seno (arriba) y gaussiana (abajo), empleando 5, 6, 7 y 8 puntos de interpolación.

Vista previa 2026



Figura 3.13 En la actualidad, los splines se mantienen como uno de los métodos de interpolación más utilizados en diversas áreas científicas y de ingeniería.

3.3.2 Interpolación usando splines cúbicos

Este método es especialmente útil para funciones que requieren una interpolación con suavidad, evitando oscilaciones indeseadas que pueden ocurrir con polinomios de alto grado, como sucede con la interpolación de Lagrange.

Dado el conjunto de nodos de interpolación $\{(x_i, \hat{u}_i)\}_{i=1}^{N}$ con $x_1 < x_2 < \cdots < x_N$, el método de splines cúbicos construye una función s definida por partes, donde cada segmento en el intervalo $[x_i, x_{i+1}]$ está dado por un polinomio cúbico

$$s(x) = s_i(x), \quad x \in [x_i, x_{i+1}], \quad i = 1, \ldots, N-1,$$

donde

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3.$$

Los coeficientes a_i, b_i, c_i, d_i se determinan de modo que $s(x)$ interpole exactamente los datos, además sea suave y que garantice la continuidad de la función y de su primera y segunda derivadas en todo el intervalo. Para asegurar la unicidad de la solución, se imponen condiciones de frontera, siendo una opción común las llamadas condiciones naturales, que requieren que la segunda derivada del spline se anule en los extremos: $s''(x_0) = s''(x_n) = 0$.

La implementación de este método en *Python* es directo mediante el uso del comando:

```
CubicSpline(x_nodes, u_nodes)
```

Este también se encuentra en `scipy.interpolate` y se construye a partir del mismo conjunto de datos descritos anteriormente.

```python
import numpy as np
from scipy.interpolate import CubicSpline

# Definicion de la funcion exacta
def u(x):
    return np.sin(x)

# Puntos de interpolacion
N = 10  # numero de puntos
x_nodes = np.linspace(0, 2*np.pi, N)
u_nodes = u(x_nodes)

# Construccion del interpolador cubico
cs = CubicSpline(x_nodes, u_nodes)

# Evaluacion en una malla mas fina
x_plot = np.linspace(0, 2*np.pi, 100)
y_exact = u(x_plot)
y_interp = cs(x_plot)
```

Código 3.5 Implementación en *Python* para una interpolación de N puntos usando cubic splines.

Vista previa 2026

La Figura 3.14 presenta los resultados numéricos de las mismas funciones usadas con el método de Lagrange, pero ahora considerando splines cúbicos. La calidad de la aproximación mejora conforme aumenta el número de datos; aunque a simple vista no resulta evidente para $N > 10$. Una forma de evidenciarlo consiste en graficar el error absoluto calculada como la diferencia entre la solución esperada y la aproximación obtenida, es decir, $|u(x_p) - s(x_p)|$, donde x_p representa un punto arbitrario dentro del dominio de interés. Las gráficas de los errores para la función seno se muestran en la Figura 3.15, donde se observa claramente cómo los errores disminuyen a medida que aumenta el número de puntos empleados, en concordancia con las propiedades teóricas del método. Aunque estos resultados no se muestran explícitamente, el mismo comportamiento se presenta al considerar la función gaussiana.

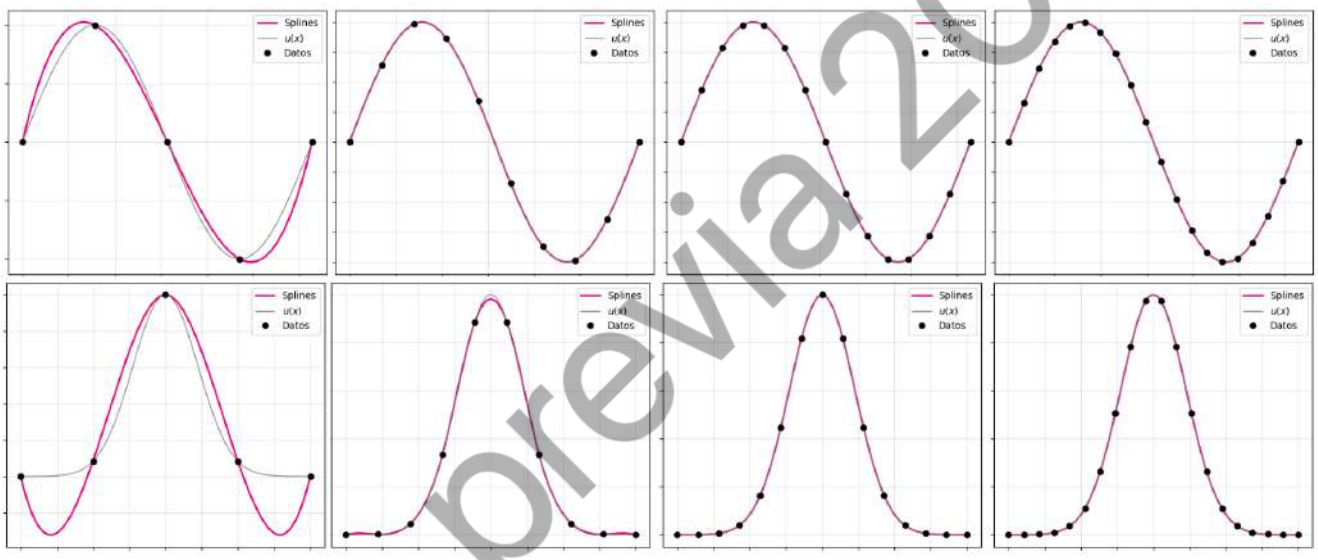


Figura 3.14 Resultados del método de interpolación usando splines cúbicos aplicados a las funciones seno (arriba) y gaussiana (abajo), empleando 5, 10, 15 y 20 puntos de interpolación.

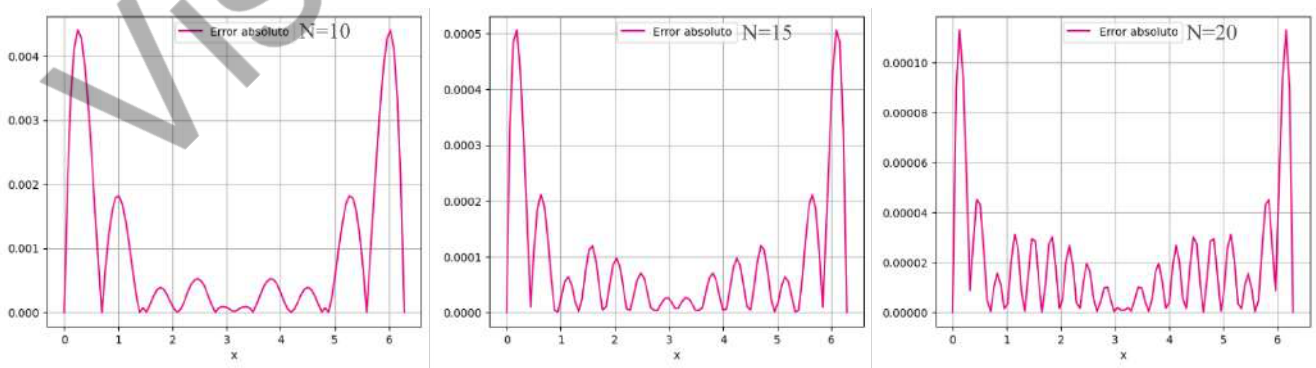


Figura 3.15 Errores absolutos del método de interpolación usando splines cúbicos aplicados a la función seno empleando 10, 15 y 20 puntos de interpolación.

Vista previa 2026



Figura 3.16 Es momento de dar un paso más en nuestro recorrido y aplicar los conceptos fundamentales aprendidos de redes neuronales al ajuste de curvas.

3.4 INTERPOLACIÓN CON REDES NEURONALES

Como se mencionó al inicio de este capítulo, el propósito de abordar el problema de ajuste de curvas es el de familiarizar al lector con la notación, los componentes estructurales y el funcionamiento básico del método de redes neuronales artificiales. La Figura 3.17 presenta un diagrama ilustrativo de los pasos necesarios para abordar este problema.

El problema considerado es el mismo que el tratado mediante los métodos clásicos de interpolación; por ello, en las secciones siguientes nos centraremos en detallar cada una de las etapas restantes. En primer lugar, se abordará la construcción del modelo, incluyendo la representación de las capas, las funciones de activación y los parámetros de la red (pesos y sesgos). Posteriormente, se analizarán los pasos restantes, que comprenden la recopilación de los datos, la definición de la función de pérdida, así como el entrenamiento y la evaluación del modelo.

Es importante señalar que los fundamentos de las redes neuronales fueron estudiados con detalle en el capítulo anterior, donde se introdujeron sus principales componentes, las diferentes arquitecturas y el papel de las funciones de activación. Asimismo, se explicó el procedimiento de entrenamiento mediante retropropagación y el método de descenso de gradiente para la optimización de los parámetros. Por esta razón, en el presente capítulo no profundizaremos nuevamente en estos conceptos, sino que nos concentraremos en su aplicación al problema de ajuste de curvas.

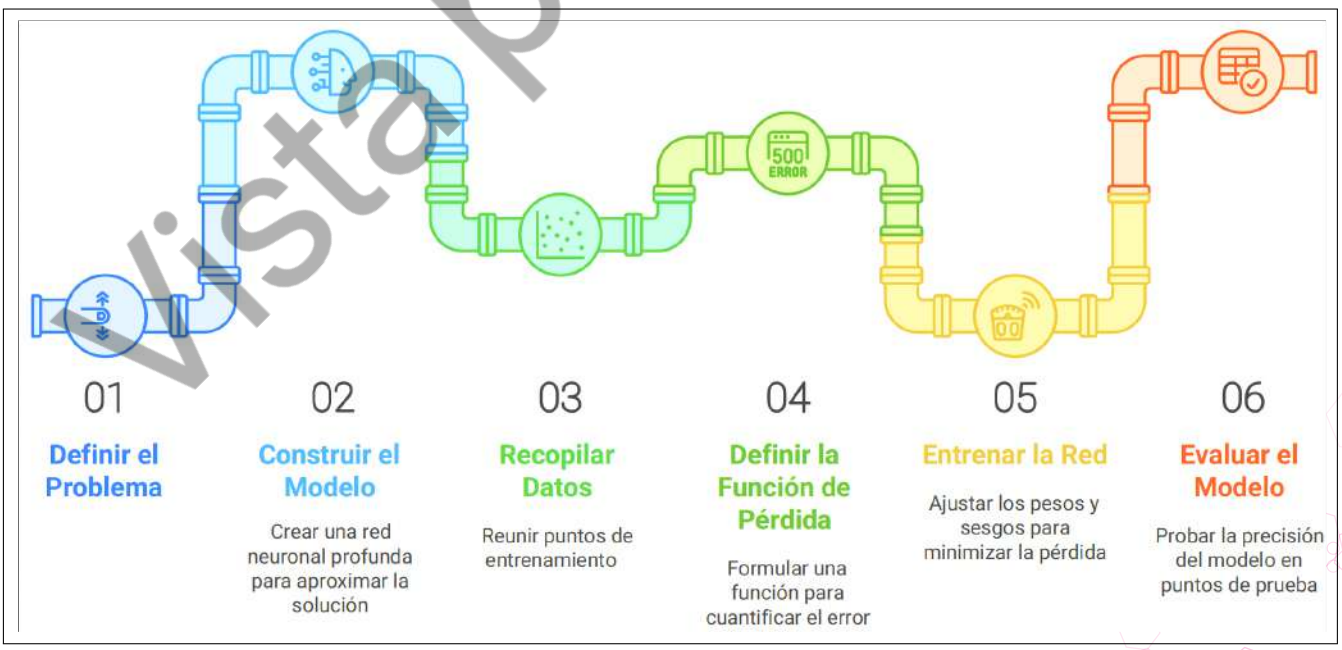


Figura 3.17 Diagrama ilustrativo de los pasos necesarios para resolver un problema utilizando redes neuronales.

Vista previa 2026

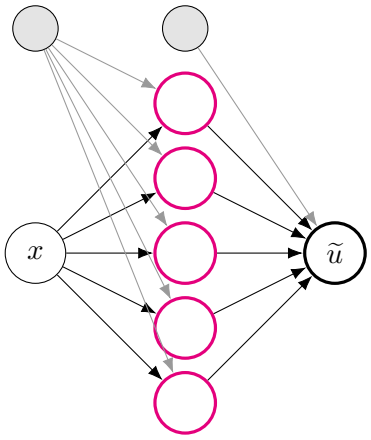


Figura 3.18 Arquitectura de una red neuronal con una entrada, una capa oculta con 5 neuronas y una salida.

3.4.1 Construcción del modelo

La red neuronal propuesta para el ajuste de curvas tiene como objetivo aproximar una función u de una sola variable; en consecuencia, adopta una arquitectura simple. La Figura 3.18 presenta un esquema de la red neuronal empleada en este ejemplo, la cual cuenta con cinco neuronas en una sóla capa oculta. Esta elección permite ilustrar de manera clara los elementos fundamentales de una red neuronal artificial sin introducir una complejidad innecesaria.

En el ajuste de curvas por medio de redes neuronales, se propone representar $u(x)$ mediante una red neuronal multicapa de la siguiente forma:

$$u(x) \approx \tilde{u}(x; W, b), \tag{3.40}$$

donde W y b representan los pesos y sesgos (*biases*) de la red, respectivamente. Lo parámetros W y b se ajustan durante el entrenamiento del modelo para minimizar una función de pérdida definida posteriormente.

La arquitectura de la red neuronal propuesta está compuesta por:

- Capa de entrada: contiene una única neurona que representa la variable independiente x.
- Capa oculta: una única capa oculta con un número N_e de neuronas. En nuestro ejemplo, tenemos $N_e = 5$. La función de activación tangente hiperbólica (tanh) se aplica sobre una combinación lineal de las entradas.
- Capa de salida: contiene una única neurona que produce la salida de la red (la aproximación de $u(x)$). Esta capa no aplica ninguna función de activación, se le denomina lineal.

Desde el punto de vista matemático, el comportamiento de este modelo neuronal puede describirse mediante las ecuaciones que se presentan a continuación, donde se detallan las operaciones realizadas en cada capa y el papel que desempeñan los pesos, los sesgos y la función de activación. La forma más usual de escribirlo es mediante capas:

$$\boldsymbol{z}^{[1]} = W^{[1]}x + \boldsymbol{b}^{[1]}, \qquad a^{[1]} = \tanh(\boldsymbol{z}^{[1]}), \qquad \tilde{u} = W^{[2]}\boldsymbol{a}^{[1]} + b^{[2]}, \tag{3.41}$$

donde $\boldsymbol{z}^{[1]} \in \mathbb{R}^5$ es el vector de activaciones lineales de la capa oculta antes de aplicar la función no lineal, $\boldsymbol{a}^{[1]} \in \mathbb{R}^5$ es el vector resultante de aplicar la función de activación tanh de manera componente a componente sobre $\boldsymbol{z}^{[1]}$ y finalmente $\tilde{u} \in \mathbb{R}$ representa la salida final de la red neuronal para una entrada escalar x. Además tenemos que:

- $W^{[1]} \in M_{5\times 1}(\mathbb{R})$ la matriz de pesos que conecta la capa de entrada con la capa oculta. Cada fila representa los pesos asociados a una de las 5 neuronas ocultas con respecto a la única entrada x.

Equivalentemente el modelo puede escribirse como

$$\tilde{u}(x) = W^{[2]}\tanh(W^{[1]}x + \boldsymbol{b}^{[1]}) + b^{[2]}.$$

La red neuronal también puede escribirse como una expresión funcional de la entrada x de la forma:

$$\begin{aligned}\tilde{u}(x) = & w_1^{[2]}\tanh(w_1^{[1]}x + b_1^{[1]}) + \\ & w_2^{[2]}\tanh(w_2^{[1]}x + b_2^{[1]}) + \\ & w_3^{[2]}\tanh(w_3^{[1]}x + b_3^{[1]}) + \\ & w_4^{[2]}\tanh(w_4^{[1]}x + b_4^{[1]}) + \\ & w_5^{[2]}\tanh(w_5^{[1]}x + b_5^{[1]}) + b^{[2]}.\end{aligned}$$

Entonces, el modelo completo consta de 16 parámetros desconocidos que determinan completamente el comportamiento de la red neuronal.

Figura 3.19 Formas equivalentes de escribir matemáticamente la red neuronal propuesta.

Vista previa 2026

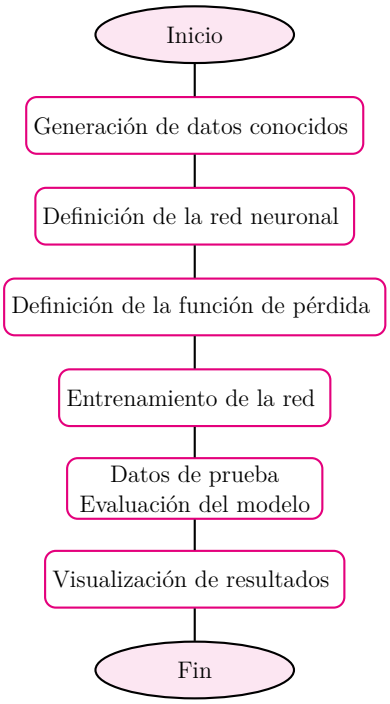


Figura 3.20 Diagrama de flujo para el ajuste de curvas usando redes neuronales.



Figura 3.21 En muchas áreas científicas, especialmente al abordar problemas del mundo real, la recopilación de datos puede resultar costosa o difícil. Por esta razón, generalmente se dispone de un número limitado de datos conocidos.

- $\boldsymbol{b}^{[1]} \in \mathbb{R}^5$ el vector de sesgos (bias) correspondiente a cada una de las neuronas de la capa oculta. Estos valores permiten desplazar el resultado de las neuronas antes de aplicar la función de activación.
- $W^{[2]} \in M_{5\times 1}(\mathbb{R})$ es la matriz de pesos que conecta las salidas de las 5 neuronas ocultas con la única neurona de salida.
- $b^{[2]} \in \mathbb{R}$ es el sesgo de la neurona de salida.

El lector puede obsrvar que no es difícil generalizar este procedimiento y proponer arquitecturas más complejas, por ejemplo, incorporando un mayor número de capas, más neuronas o distintas funciones de activación. Sin embargo, en esta etapa se opta por trabajar inicialmente con un modelo sencillo, dejando el análisis de configuraciones más elaboradas para secciones posteriores.

Por otra parte, a diferencia del capítulo anterior, donde todas las operaciones asociadas a las redes neuronales se realizaron de manera manual, en este caso dichas operaciones se llevarán a cabo utilizando paquetería especializada en *Python*. No obstante, dado que estos procedimientos requieren una explicación más detallada, su implementación se presenta en una sección independiente, una vez que se han descrito completamente todos los pasos involucrados en esta formulación.

3.4.2 Recopilación de datos

Para determinar los pesos (W) y sesgos (b) del método propuesto, es necesario disponer de un conjunto de datos conocidos sobre los cuales se construye el modelo. En el problema de ajuste de curvas, esta información se obtiene exclusivamente a partir de las N observaciones iniciales disponibles. En particular, para la función seno, el conjunto de datos considerado está formado por (3.38), es decir:

$$\{(x_i, \hat{u}_i)\}_{i=1}^{N}, \quad x_i \in [0, 2\pi], \quad \hat{u}_i = \operatorname{sen}(x_i). \tag{3.42}$$

Como es de esperarse, la cantidad de datos conocidos, denominados también como puntos de entrenamiento, jugará un papel fundamental en la precisión de la aproximación realizada por la red neuronal.

- En general, si el número de datos es muy reducido, la red puede no contar con suficiente información para aprender la forma general de la función, lo que puede llevar a una mala generalización en puntos no observados, incluso si el error sobre los datos de entrenamiento es pequeño. Esto se conoce como sobreajuste a pocos datos.
- Por otro lado, un mayor número de datos permite capturar con mayor fidelidad las características de la función objetivo, especialmente en regiones con variaciones rápidas. Sin embargo, este aumento de datos también conlleva un incremento en el costo computacional y podría requerir arquitecturas más complejas para obtener una mejor aproximación.

Vista previa 2026

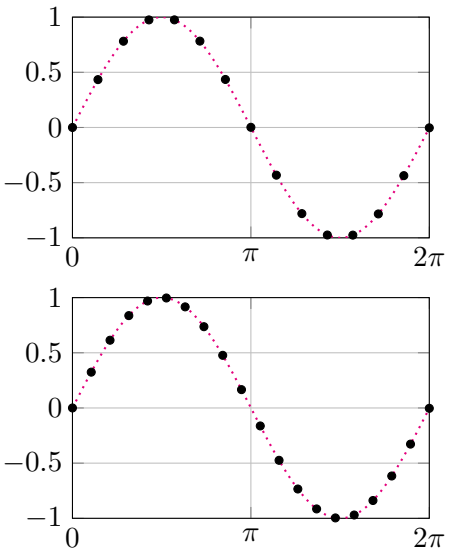


Figura 3.22 Ejemplos de 15 y 20 observaciones correspondientes al muestreo de la función seno con puntos equidistantes en el intervalo $[0, 2\pi]$.

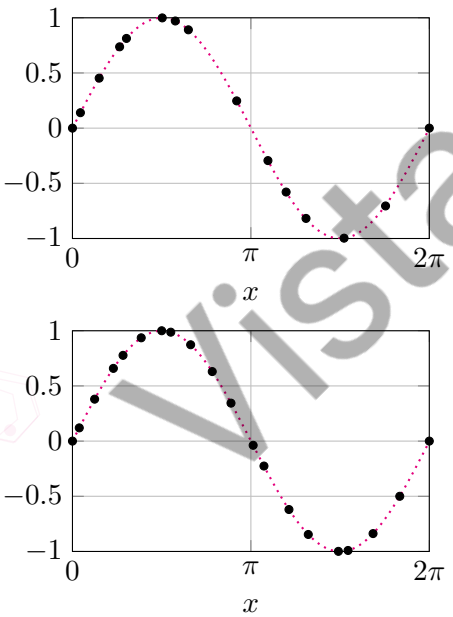


Figura 3.23 Ejemplos de 15 y 20 observaciones correspondientes al muestreo de la función seno de manera aleatoria.

En consecuencia, existe un equilibrio entre la cantidad de datos conocidos, la complejidad del modelo y la capacidad de generalización. Este balance es crucial al diseñar redes neuronales, y motiva la necesidad de experimentar con distintos tamaños de conjuntos de datos.

En un primer momento, los valores de x_i se han elegido de manera equidistante dentro del dominio de interés, ya que esta elección facilita la comprensión inicial del método y su implementación. No obstante, es importante destacar que esta selección no representa una limitación inherente al método, puesto que los puntos de entrenamiento también pueden escogerse de forma aleatoria. La Figura 3.22 muestra unos ejemplos de conjuntos de datos generados de manera aleatoria usados para entrenar nuestra red.

En la implementación realizada en *Python*, la generación de puntos equidistantes se lleva a cabo mediante la instrucción :

```
np.linspace(xI, xF, N)
```

la cual distribuye igualmente espaciado los N puntos en el intervalo $[x_I, x_F]$. Si se desea trabajar con puntos distribuidos aleatoriamente en el mismo dominio, basta con sustituir dicha línea por la siguiente instrucción:

```
np.random.uniform(low=xI, high=xF, size=(N,1))
```

De esta forma, se generan N puntos de entrenamiento independientes y aleatorios con la distribución uniforme en el intervalo considerado, sin modificar el resto del procedimiento numérico.

3.4.3 Función de pérdida

Una vez definida la arquitectura del modelo $\widetilde{u}(x_i; W, b)$ y recopilados los datos $\{(x_i, \hat{u}_i)\}_{i=1}^{N}$ a partir de los cuales se determinarán sus parámetros, se procede al proceso de entrenamiento. Este proceso consiste en ajustar los parámetros de la red neuronal —es decir, los pesos y sesgos de cada capa— de manera que la salida generada por el modelo se aproxime lo mejor posible a los valores conocidos de la función objetivo.

El entrenamiento se lleva a cabo minimizando una función de pérdida, o también llamada función de costo, que cuantifica el error entre la predicción y el valor real. Para el problema de ajuste de curvas usaremos el error cuadrático medio (MSE, por sus siglas en inglés), definido como:

$$E(W, b) = \frac{1}{N} \sum_{i=1}^{N} \left(\widetilde{u}(x_i; W, b) - \hat{u}_i\right)^2 . \tag{3.43}$$

En la fórmula anterior, la predicción proporcionada por la red neuronal se evalúa en el mismo punto x_i para el cual se conocen los datos.

Vista previa 2026

La calidad del entrenamiento depende de:
- El conjunto de datos.
- La arquitectura de la red neuronal.
- Los hiperparámetros del algoritmo de optimización (como la tasa de aprendizaje).
- La inicialización de los parámetros.

Figura 3.24 Sobre el entrenamiento.

3.4.4 Entrenamiento de la red neuronal

Determinar los parámetros de la red que producen predicciones lo más cercanas posible a los valores reales conocidos equivale a minimizar la función (3.43). Como se analizó en el capítulo anterior, este problema de optimización se aborda habitualmente mediante algoritmos basados en el descenso de gradiente, tales como el *descenso de gradiente estocástico* (SGD) y sus variantes más avanzadas. En particular, para el problema de ajuste de curvas se empleará el método *Adam*.

Durante el proceso de ajuste, se evalúa el valor de la función de pérdida y se calculan sus derivadas con respecto a cada uno de los parámetros del modelo. Estos gradientes permiten actualizar de manera iterativa los pesos y sesgos de la red en la dirección que reduce el error. Dicho cálculo se realiza mediante el algoritmo de backpropagation, el cual aplica de forma eficiente la regla de la cadena a lo largo de las distintas capas de la red. Todos estos procedimientos se llevan a cabo utilizando paquetería especializada en *Python*, la cual se describe en la siguiente sección.

3.4.5 Evaluación del modelo

Finalmente, un entrenamiento exitoso se traduce en una red neuronal capaz de generalizar el comportamiento de la función objetivo, más allá de los puntos en los que fue entrenada. Es decir, el modelo no solo debe ajustar adecuadamente los datos del conjunto de entrenamiento, sino también ser capaz de predecir con precisión valores de la función para nuevas entradas no vistas previamente.

Para evaluar esta capacidad de generalización, se introduce un conjunto de datos de prueba, conformado por un número de puntos N_p que no fueron utilizados durante el proceso de entrenamiento:

$$\{(x_j, u(x_j))\}_{j=1}^{N_p}, \quad x_j \in [0, 2\pi]. \tag{3.44}$$

En el ejemplo de ajuste de curvas asociado a la función seno, los puntos de prueba se toman como $N_p = 100$ valores equidistantes x_j dentro del intervalo dado. Esto permite observar visualmente, mediante gráficos, qué tan bien la red logra capturar el comportamiento de la función en todo el dominio. El error que el modelo comete sobre este conjunto de prueba ofrece una medida objetiva de su capacidad de generalización.

- Una red neuronal bien entrenada mostrará un error bajo tanto en el conjunto de entrenamiento como en el de prueba, lo cual indica que ha aprendido correctamente la función objetivo.
- En contraste, si el error en el conjunto de prueba es significativamente mayor que en el de entrenamiento, se dice que el modelo sufre de sobreajuste, lo que sugiere que sólo ha memorizado los datos en lugar de capturar una regla general.

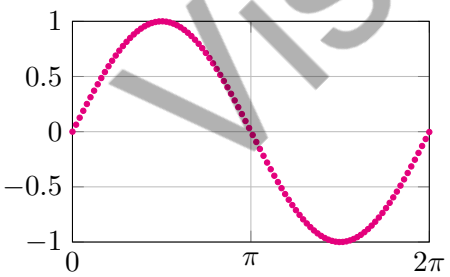


Figura 3.25 Ejemplo de 100 datos de prueba correspondientes al muestreo de la función seno con puntos equidistantes en el intervalo $[0, 2\pi]$.

Vista previa 2026



La librería `torch`, la cual se importa junto con el módulo `torch.nn`, contiene las clases y los métodos correspondientes a redes neuronales.

Figura 3.26 Librerías.



Figura 3.27 A programar.

3.5 IMPLEMENTACIÓN EN PYTHON Y PYTORCH

A continuación se presenta el programa completo en *Python* usando *PyTorch* de una red neuronal con una sola capa oculta, utilizada para el ajuste de curvas basado en la función seno. El código puede copiarse y ejecutarse directamente. Asimismo, la misma estructura puede extenderse de manera sencilla para trabajar con otras funciones.

```python
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

# 1. Datos de entrenamiento (N puntos uniformes)
N = 20
x_train = np.linspace(0, 2*np.pi, N).reshape(-1, 1)
u_train = np.sin(x_train)
x_tensor = torch.tensor(x_train, dtype=torch.float32)
u_tensor = torch.tensor(u_train, dtype=torch.float32)

# 2. Red neuronal simple
Ne = 5
class SimpleMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1, Ne)
        self.output = nn.Linear(Ne, 1)
    def forward(self, x):
        x = torch.tanh(self.hidden(x))
        return self.output(x)
model = SimpleMLP()

# 3. Funcion de perdida y optimizador
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# 4. Entrenamiento de la red
for epoch in range(1000):
    optimizer.zero_grad()
    output = model(x_tensor)
    loss = criterion(output, u_tensor)
    loss.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print(f"Epoch {epoch}: loss = {loss.item():.6f}")

# 5. Datos de prueba (100 puntos uniformemente distribuidos)
Np = 100
x_p = np.linspace(0, 2*np.pi, Np)
u_p = np.sin(x_p)
x_p_tensor = torch.tensor(x_p, dtype=torch.float32).view(-1, 1)
u_p_pred = model(x_p_tensor).detach().numpy()

# 6. Graficar resultados
plt.plot(x_train, u_train, 'o', label="Datos")
plt.plot(x_p, u_p, label="u(x)")
plt.plot(x_p, u_p_pred, '--', label="Red neuronal")
plt.xlabel('x')
plt.ylabel('u')
plt.grid(True)
plt.legend()
plt.show()
```

Código 3.6 Implementación usando *PyTorch* para el ajuste de curvas.

Vista previa 2026

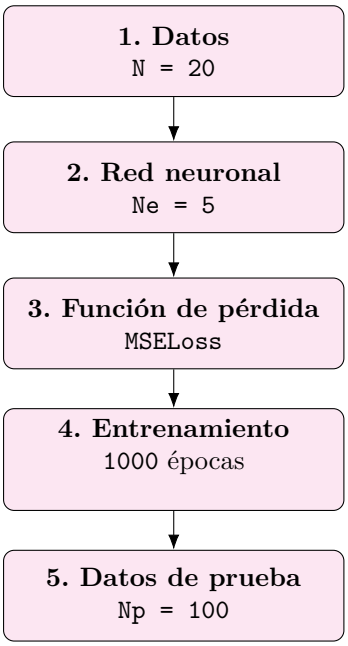


Figura 3.28 Esquema general del ajuste de curvas usando una red neuronal de una capa oculta con 5 neuronas.

El programa anterior fue diseñado con una estructura clara que permite identificar fácilmente cada una de las etapas involucradas en el ajuste de una curva mediante una red neuronal. La implementación sigue el mismo orden en el que se presentaron los conceptos teóricos, de modo que el lector puede relacionar directamente la teoría con el código. Cada sección del programa cumple una función específica desde la generación de los datos hasta la visualización de los resultados. A continuación, se describe cada una de estas partes de manera detallada, enfatizando su propósito y la forma en que contribuyen al funcionamiento global del método.

Nótese además que, para los experimentos realizados, se han identificado y definido explícitamente los tres parámetros principales que se modificarán de manera sistemática: el número de datos de entrenamiento (`N`), el número de neuronas en la única capa oculta de la red neuronal (`Ne`) y el número de puntos utilizados para la evaluación del modelo (`Np`).

Las librerías básicas utilizadas en este programa son `numpy`, empleada para realizar de manera eficiente las operaciones numéricas, y `matplotlib`, que se utiliza para la generación y visualización de gráficas. Adicionalmente, se emplea la librería `torch`, la cual se importa junto con el módulo `torch.nn`. Este último contiene las clases y métodos necesarios para la construcción, evaluación y entrenamiento de redes neuronales, como capas, funciones de activación y funciones de pérdida.

En las siguientes secciones describiremos brevemente los principales componentes del código, comenzando con la presentación de los datos conocidos y la construcción de la red neuronal. Posteriormente, abordaremos la definición de la función de pérdida y el proceso de entrenamiento, para finalizar con la evaluación del modelo utilizando los datos de prueba.



Figura 3.29 La visualización clara de resultados es esencial en métodos numéricos. En el fondo, una buena gráfica es un trabajo artístico, guiado por la creatividad y el criterio personal.

3.5.1 Datos conocidos para el entrenamiento de la red

El programa inicia con un muestreo uniforme `x_train` de `N=20` puntos en el intervalo $[0, 2\pi]$. Cada punto se evalúa en la función seno sobre `u_train` para completar los pares de entrenamiento. Posteriormente, estos datos se transforman al formato de *PyTorch* mediante `torch.tensor`, lo que permite un procesamiento eficiente durante el entrenamiento.

En *PyTorch*, cada muestra debe representarse como una fila y cada característica como una columna. Dado que los datos suelen generarse como vectores unidimensionales de N elementos, se usa `reshape(-1, 1)` para convertirlos en una matriz con N filas y una columna. Omitir este ajuste puede provocar una interpretación incorrecta de las dimensiones y errores durante la evaluación del modelo.

Vista previa 2026

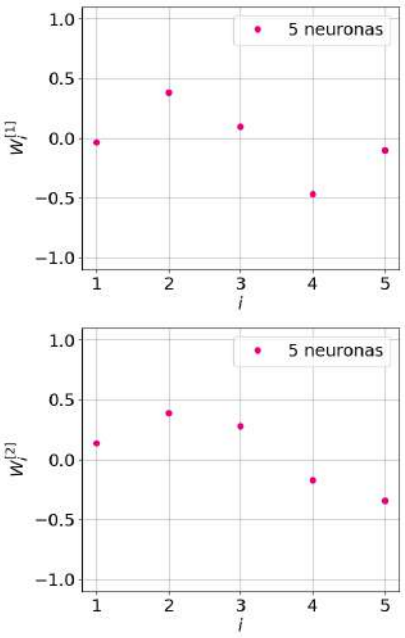


Figura 3.30 Valores iniciales de los pesos $W^{[1]}$ y $W^{2]}$ usando 5 neuronas.

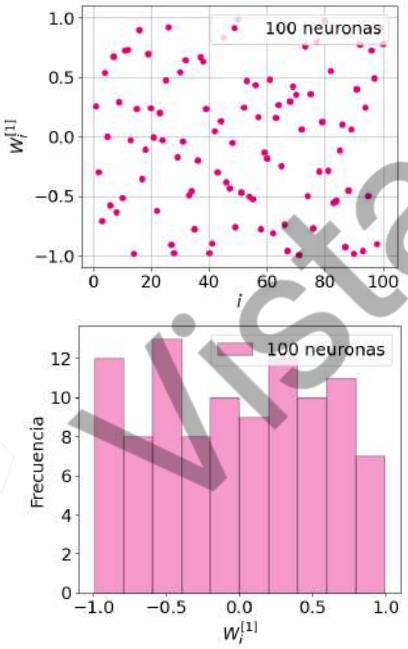


Figura 3.31 Distribución de los valores iniciales de los pesos $W^{[1]}$ correspondientes a la capa oculta usando 100 neuronas.

3.5.2 Construcción de la red neuronal

A continuación, se define la arquitectura de la red de una sola capa con `Ne=5` neuronas mediante la clase `SimpleMLP`. Al usar esta clase, se ejecuta el método `__init__`, que inicializa los pesos y sesgos de la red, creando así el objeto correspondiente. La función `forward` se encarga de propagar una entrada a través de la red para obtener la salida correspondiente.

Cabe destacar que, al crear la red, *PyTorch* asigna valores iniciales a los pesos y sesgos de manera automática usando una distribución uniforme. Esto permite evaluar la red incluso antes de realizar cualquier entrenamiento. La Figura 3.30 presenta los valores iniciales de los pesos de la capa oculta $W^{[1]}$ y la capa de salida $W^{[2]}$ para redes con 5 neuronas. Al aumentar el número de neuronas, la distribución de los pesos se aprecia con mayor claridad, como se muestra en la Figura 3.31. En estas figuras, la frecuencia se obtiene dividiendo el rango de valores en intervalos y contabilizando el número de pesos en cada uno, los cuales se representan mediante barras.

3.5.3 Definición de la función de pérdida

Se selecciona como la función de pérdida de *error cuadrático medio* (MSE) mediante `nn.MSELoss` y se define el optimizador `Adam` con una tasa de aprendizaje de `0.01`. Esta combinación es ampliamente utilizada en problemas de regresión, ya que permite un ajuste eficiente de los parámetros de la red.

3.5.4 Entrenamiento de la red

Se define el número de épocas de entrenamiento en la variable `epochs`, fijada en 1000. A continuación, se implementa un ciclo `for` que, en cada iteración, realiza los siguientes pasos: evalua la red con los datos de entrenamiento y calcula la salida (`output`), determina la pérdida comparando la salida de la red con los valores esperados (`loss`), calcula los gradientes mediante el método `backward` y actualiza los parámetros de la red usando `optimizer.step`. Adicionalmente, cada 100 épocas se imprime el valor de la pérdida, lo que permite monitorear el progreso del entrenamiento y detectar posibles problemas, como gradientes desbordados o un aprendizaje inadecuado.

3.5.5 Evaluación con los datos de prueba

Una vez finalizado el entrenamiento, se evalúa el desempeño de la red. En el código se genera un conjunto de validación mediante `Np=100` puntos equiespaciados con distribución uniforme en $[0, 2\pi]$. La función seno se evalúa en estos puntos y así obtener `u_p` correspondiente a los valores exactos en el conjunto de validación. Los mismos puntos son usados pero ahora en el modelo para obtener `u_p_pred` que corresponden a los valores aproximados generados por la red.

Vista previa 2026



Figura 3.32 Las simulaciones numéricas se llevaron a cabo utilizando la plataforma *Google Colab*, la cual proporciona un entorno accesible para la ejecución de código en *Python*.

3.6 SIMULACIONES USANDO REDES NEURONALES

Finalmente, en esta sección se evalúa la robustez de la metodología propuesta basada en redes neuronales. Para ello, se llevarán a cabo diversas pruebas enfocadas en los principales parámetros del método, tales como el número de datos disponibles y el número de neuronas de la red. Modificaciones de mayor alcance, como el incremento en el número de capas, se abordarán en el siguiente capítulo, donde se extenderá el análisis a la resolución de ecuaciones diferenciales.

3.6.1 Simulaciones 20 datos conocidos y 5 neuronas

Salvo que se indique lo contrario, en todas las simulaciones se utilizan $N = 20$ datos conocidos, distribuidos uniformemente en el dominio de interés, una red neuronal sencilla con una única capa oculta compuesta por $N_e = 5$ neuronas y un conjunto de $N_p = 100$ puntos de prueba, también distribuidos de manera uniforme.

Notas

```
N = 20,
Ne = 5,
Np = 100.
```

Figura 3.33 Parámetros.

La Figura 3.34 muestra los resultados de ejecutar el mismo código tres veces, realizadas en la plataforma *Colab*. En la misma gráfica se presentan de manera conjunta los resultados obtenidos por la red neuronal, la función objetivo y los datos utilizados durante el entrenamiento. Asimismo, en la parte inferior de la figura se incluye la gráfica del error absoluto entre la aproximación neuronal y la solución exacta, lo que permite una mejor visualización del ajuste logrado por el modelo.

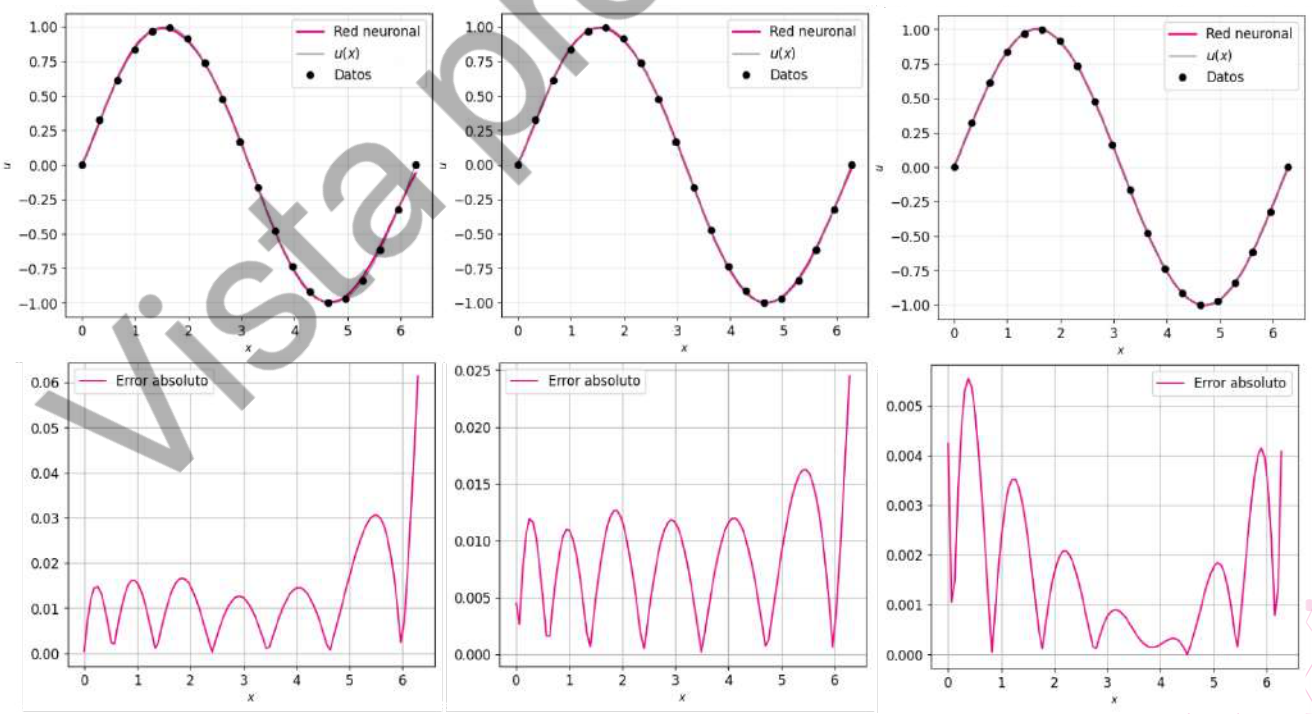


Figura 3.34 Resultados usando una red de una capa oculta con 5 neuronas y con 20 datos conocidos.

Vista previa 2026

```
Epoch   0: loss = 0.484411
Epoch 100: loss = 0.115193
Epoch 200: loss = 0.018183
Epoch 300: loss = 0.001162
Epoch 400: loss = 0.000128
Epoch 500: loss = 0.000040
Epoch 600: loss = 0.000024
Epoch 700: loss = 0.000016
Epoch 800: loss = 0.000011
Epoch 900: loss = 0.000010
```

Figura 3.35 Ejemplo del comportamiento del error cuadrático medio a lo largo de las épocas durante el proceso de entrenamiento.

A pesar de que la arquitectura de la red, los datos de entrada y los hiperparámetros se mantuvieron constantes, se observa una diferencia entre los resultados de las distintas ejecuciones. Podemos observar que en todos estos casos, la red neuronal logra capturar el comportamiento de la función objetivo; no obstante, la calidad del ajuste puede variar al ejecutar varias veces el mismo código. Esto se debe a la naturaleza estocástica del entrenamiento, especialmente cuando se utilizan inicializaciones aleatorias de los pesos y sesgos.

Nótese que, en el tercer experimento numérico, el error máximo es de 6×10^{-3}, lo que da lugar a diferencias prácticamente imperceptibles a simple vista entre la solución aproximada y la función objetivo. En este caso, el error medio es de aproximadamente 1.6×10^{-3}.

3.6.2 Convergencia del error cuadrático medio

Por otra parte, si analizamos la evolución del error de entrenamiento (`loss`), este alcanza un valor cercano a 10^{-5}. Un análisis más detallado de este comportamiento se presenta en la Figura 3.36, donde se grafica el error en función del número de épocas. En dicha figura puede observarse que, después de aproximadamente 300 épocas, el error de entrenamiento disminuye de manera significativa. Este comportamiento se aprecia con mayor claridad al representar los mismos resultados en escala logarítmica, donde se evidencia que el error continúa decreciendo incluso en etapas avanzadas del entrenamiento.

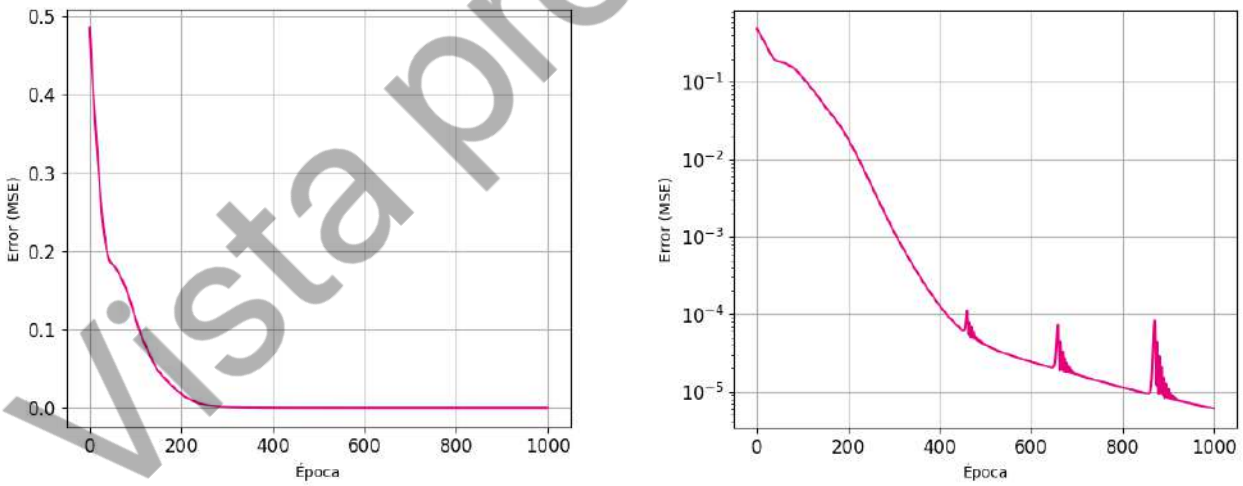


Figura 3.36 Error cuadrático medio calculada en cada época del entrenamiento usando 5 neuronas y 20 datos conocidos. La segunda gráfica presenta los mismos resultados que la primera, pero empleando una escala logarítmica en el eje vertical para facilitar la visualización de la disminución del error.

La Figura 3.37 muestra los resultados de varios entrenamientos del modelo convergiendo a una solución incorrecta. Estos resultados evidencian la sensibilidad del modelo a los parámetros iniciales usados y subrayan la importancia de realizar múltiples entrenamientos para evaluar la capacidad de generalización del modelo.

Vista previa 2026

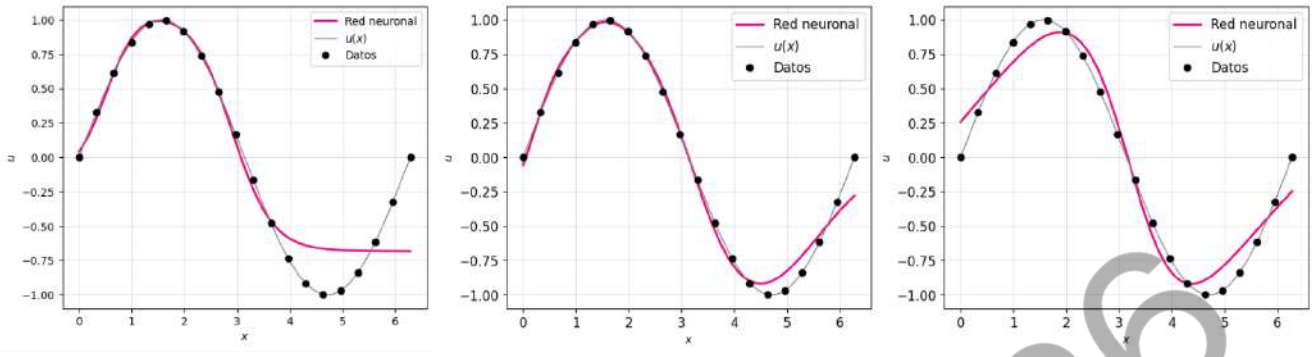


Figura 3.37 Resultados de la red neuronal cuando la curva no se ajusta correctamente a la función objetivo.



Figura 3.38 Más gráficas para analizar.

Si bien a partir de las gráficas anteriores es posible identificar que los resultados no ajustan adecuadamente la función objetivo, no es necesario llegar a esta etapa para concluir que la simulación no es correcta. Desde las primeras etapas del entrenamiento, el valor del error cuadrático medio proporciona una indicación inicial del desempeño del modelo al no seguir disminuyendo mientras aumenta el número de épocas. Cabe destacar que un error pequeño no garantiza que la red neuronal tenga una buena capacidad de generalización. Sin embargo, el hecho de que el error no disminuya o permanezca elevado es una señal clara de que el proceso de ajuste se está llevando a cabo de manera incorrecta, lo que se traduce en una aproximación deficiente de la función objetivo.

Un análisis detallado de este comportamiento se presenta en la Figura 3.39, donde se grafica el error en función del número de épocas para soluciones como las mostradas en la Figura 3.37. En dicha figura se observa que, tras cierto número de épocas, el error deja de disminuir o permanece en valores elevados, lo que indica un entrenamiento deficiente del modelo.

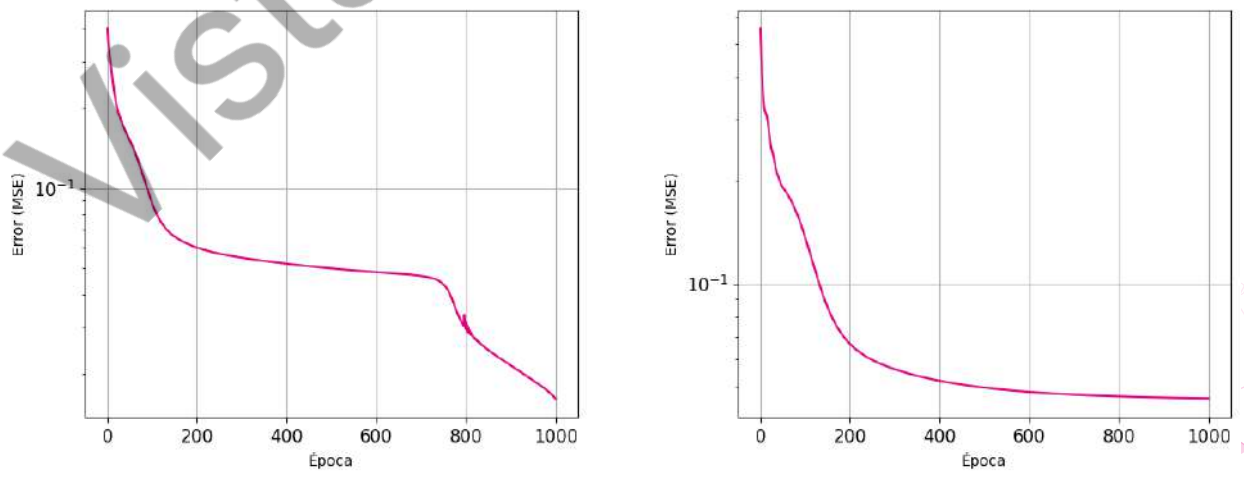


Figura 3.39 Comportamientos irregulares del error cuadrático medio a lo largo de las épocas de entrenamiento. Podemos observar como en ambas gráficas el MSE se mantiene contante en varias etapas del entrenamiento.

Vista previa 2026

$$
\begin{aligned}
W^{[1]} &= [-0.5336171 \\
&\quad -1.0336202 \\
&\quad -0.7277835 \\
&\quad 0.21641794 \\
&\quad -0.58277076] \\
\boldsymbol{b}^{[1]} &= [1.153382 \\
&\quad 0.4629482 \\
&\quad 2.4944947 \\
&\quad 0.59521943 \\
&\quad 3.5728998]^T \\
W^{[2]} &= [0.39996964 \\
&\quad -1.0358754 \\
&\quad 1.4290258 \\
&\quad 0.1900594 \\
&\quad -1.7688528]^T \\
b^{[2]} &= 0.3863224
\end{aligned}
$$

Figura 3.40 Valores de los pesos y sesgos entrenados de red la red neuronal propuesta.

El resultado del entrenamiento no es únicamente un conjunto de parámetros almacenados en un archivo, sino una función que podemos evaluar directamente en cualquier punto de su dominio. De esta manera, es posible calcular aproximaciones sin necesidad de ejecutar nuevamente el algoritmo de entrenamiento ni utilizar las herramientas con las que originalmente se construyó la red.

Figura 3.41 Entrenamiento.

3.6.3 Solución explícita con pesos y sesgos entrenados

Antes de proseguir con el análisis de los parámetros de la red, resulta conveniente hacer un paréntesis para reflexionar sobre los resultados obtenidos hasta este punto. Recordemos que uno de los objetivos centrales de esta formulación es determinar los valores de los pesos y sesgos de la red neuronal propuesta. Bajo pequeñas modificaciones al programa de entrenamiento, es posible acceder de manera directa a estos parámetros (lo cual se deja como ejercicio para el lector).

Una vez finalizado el proceso de entrenamiento, el programa permite guardar la red neuronal junto con los valores de sus pesos y sesgos. Sin embargo, es importante recordar que una red neuronal no deja de ser una función matemática, cuya expresión queda completamente determinada por su arquitectura, las funciones de activación utilizadas y los parámetros obtenidos durante el entrenamiento. Por lo tanto, siempre que se conozcan estos elementos, es posible reconstruir explícitamente la función que representa la red neuronal, sin necesidad de recurrir nuevamente al programa utilizado para entrenarla.

En nuestro caso, la aproximación resultante puede expresarse como una combinación lineal de cinco funciones tangentes hiperbólicas. En particular, al sustituir los pesos y sesgos obtenidos durante el entrenamiento en la expresión general del modelo, obtenemos

$$\hat{u}(x) = \sum_{j=1}^{5} W_{1j}^{[2]} \tanh\left(W_{j1}^{[1]} x + b_j^{[1]}\right) + b^{[2]}, \tag{3.45}$$

donde $W^{[1]} \in M_{5\times 1}(\mathbb{R})$, $\boldsymbol{b}^{[1]} \in \mathbb{R}^5$, $W^{[2]} \in M_{5\times 1}(\mathbb{R})$ y $b^{[2]} \in \mathbb{R}$. Al sustituir los coeficientes entrenados en la expresión anterior está dada por

$$
\begin{aligned}
\tilde{u}(x) = {} & 0.39996964 \tanh(-0.53361710x + 1.15338200) \\
& -1.0358754 \tanh(-1.03362020x + 0.46294820) \\
& +1.4290258 \tanh(-0.72778350x + 2.49449470) \\
& +0.1900594 \tanh(\ \ 0.21641794x + 0.59521943) \\
& -1.7688528 \tanh(-0.58277076x + 3.57289980) + 0.3863224.
\end{aligned}
$$

Esta expresión constituye una representación explícita de la función aprendida por la red neuronal.

Para comprobar esta equivalencia, en la Figura 3.42 se compara la gráfica de la función anterior con los resultados obtenidos directamente de la red neuronal entrenada. Como era de esperarse, ambas curvas coinciden prácticamente en todo el dominio considerado. De hecho, si se utilizaran exactamente los mismos parámetros y la misma precisión aritmética, ambas evaluaciones serían matemáticamente equivalentes. Las pequeñas diferencias que pudieran observarse se deben principalmente al redondeo de los coeficientes al escribirlos con un número limitado de cifras decimales y a la precisión numérica de los cálculos.

Vista previa 2026

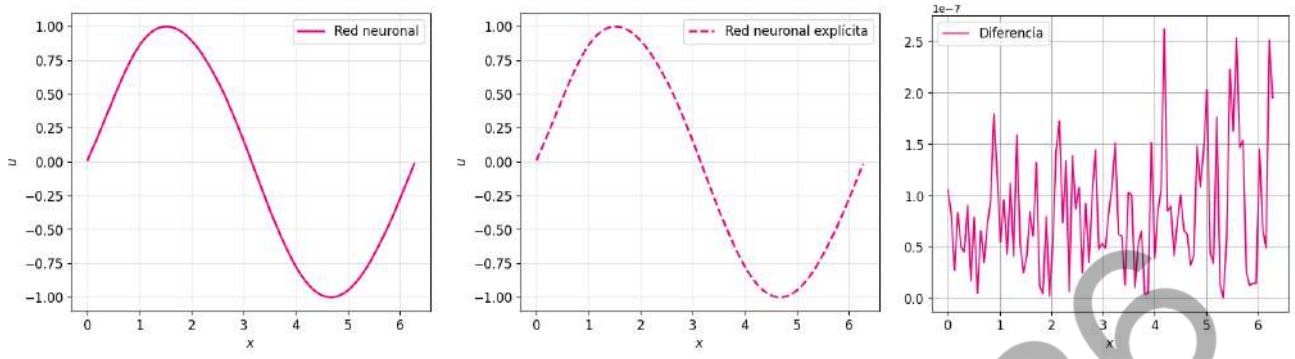


Figura 3.42 Solución numérica y su diferencia usando redes neuronales bajo los dos formatos presentados.

Notas

```
N  = 40,
Ne = 5,
Np = 100.
```

Figura 3.43 Parámetros.

3.6.4 Mayor número de datos conocidos

Con el objetivo de reducir el número de entrenamientos fallidos se incrementó el número de datos conocidos. En la Figura 3.44 se muestran los resultados obtenidos al emplear $N = 40$. En estos experimentos se obtuvieron aproximaciones precisas, alcanzándose errores de entrenamiento del orden de magnitud de 10^{-5}. No obstante, como se observa en la Figura 3.45, el modelo también converge en múltiples ocasiones a soluciones incorrectas. Este comportamiento pone de manifiesto que, si bien el aumento en la cantidad de datos mejora el desempeño en algunos casos, no es suficiente para garantizar resultados robustos, lo que evidencia la necesidad de emplear arquitecturas de red más complejas.

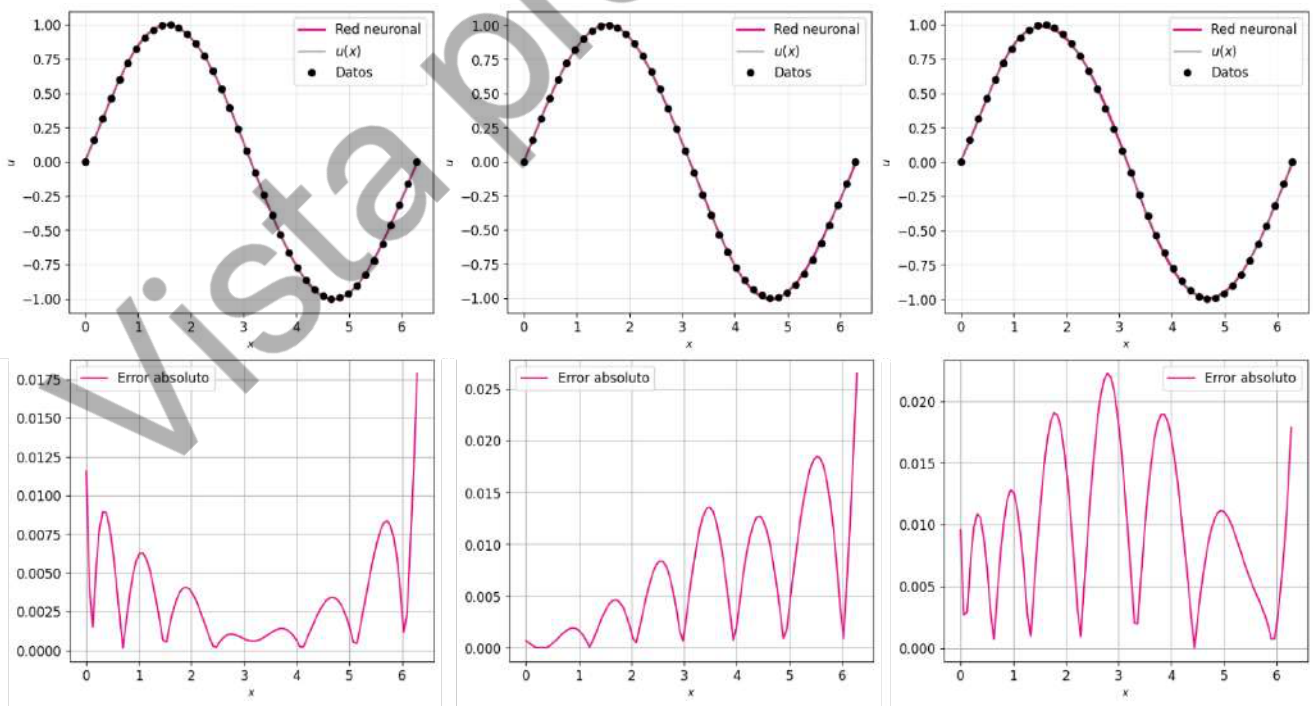


Figura 3.44 Resultados usando redes neuronales con 40 datos conocidos.

Vista previa 2026

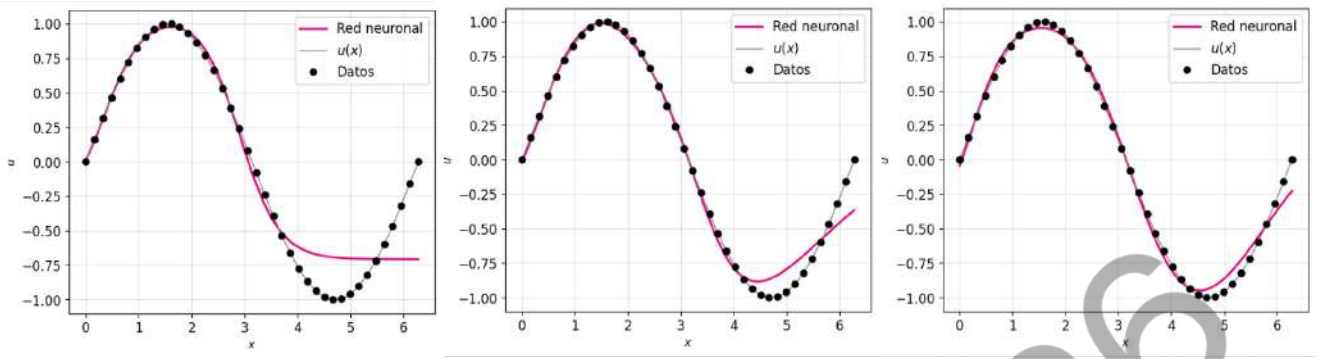


Figura 3.45 Resultados cuando la curva no se ajusta correctamente a la función objetivo con 40 datos conocidos.

Notas

```
N  = 20,
Ne = 10,
Np = 100.
```

Figura 3.46 Parámetros.

3.6.5 Mayor número de neuronas

Con el objetivo de mejorar la capacidad de aproximación de la red neuronal, se exploró una arquitectura más compleja incrementando el número de neuronas en la capa oculta. Esta modificación permite dotar al modelo de una mayor capacidad de representación, ya que al aumentar el número de neuronas se incrementa el número de parámetros libres (pesos y sesgos) que pueden ajustarse durante el entrenamiento.

Las Figura 3.47 muestra los resultados obtenidos con una red de $N_e = 10$ neuronas en la única capa oculta y 20 puntos de entrenamiento. Notar que la red neuronal nuevamente logra capturar la forma general de la función objetivo; sin embargo, la magnitud de los errores obtenidos es prácticamente la misma que en el caso con $N_e = 5$ estudiado inicialmente. En la Figura 3.48 podemos observar como la función de pérdida también se comporta de manera similar al caso de 5 neuronas ilustrado en la Figura 3.36.

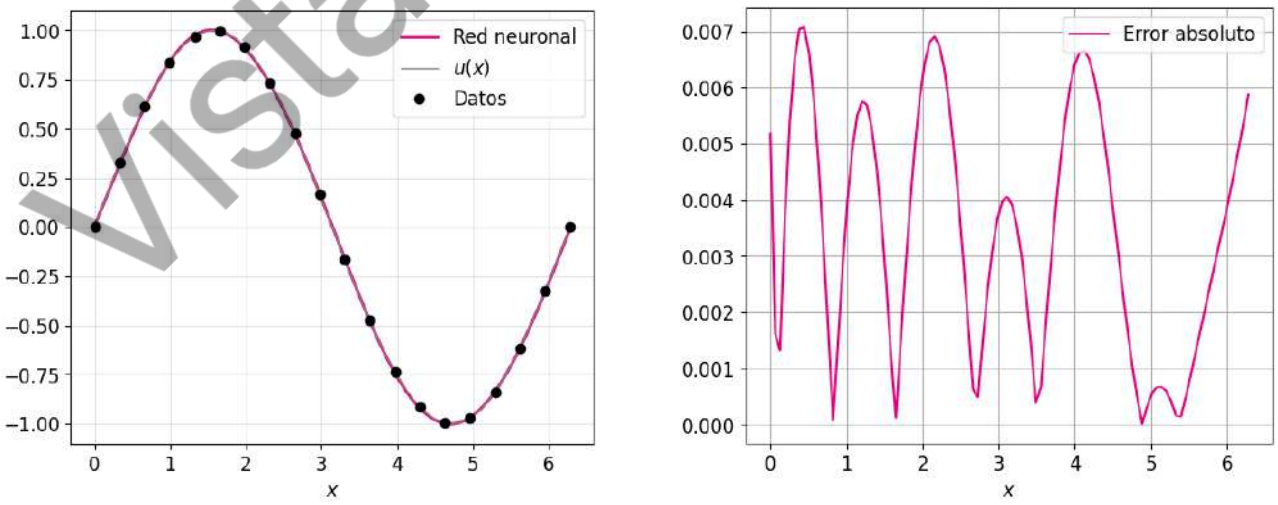


Figura 3.47 Resultados usando una red de una capa oculta con 10 neuronas y con 20 datos conocidos.

Vista previa 2026

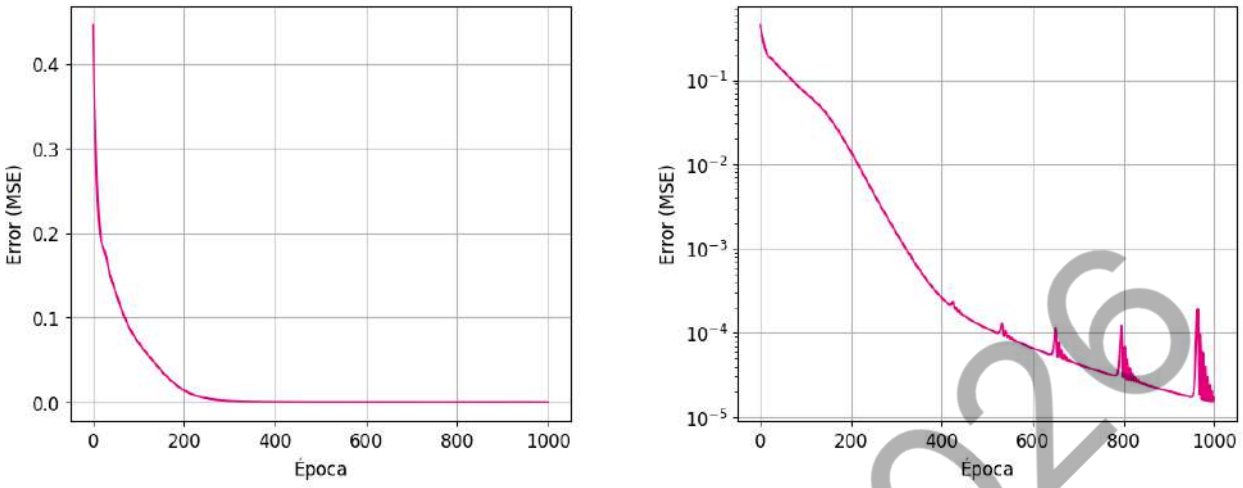


Figura 3.48 Error cuadrático medio del entrenamiento usando 10 neuronas y 20 datos conocidos. La segunda gráfica presenta los mismos resultados que la primera, pero empleando una escala logarítmica en el eje vertical.

Notas

```
N  = 5, 10, 20,
Ne = 100,
Np = 100.
```

Figura 3.49 Parámetros.

El mismo experimento puede repetirse incrementando aún más el número de neuronas en la capa oculta. En la Figura 3.50 se muestran los resultados correspondientes a una red con $N_e = 100$ neuronas, considerando distintas cantidades de puntos de entrenamiento. Es importante destacar que, incluso con un número muy reducido de datos de entrenamiento (tan solo cinco puntos), la red neuronal es capaz de aproximar razonablemente bien la función seno.

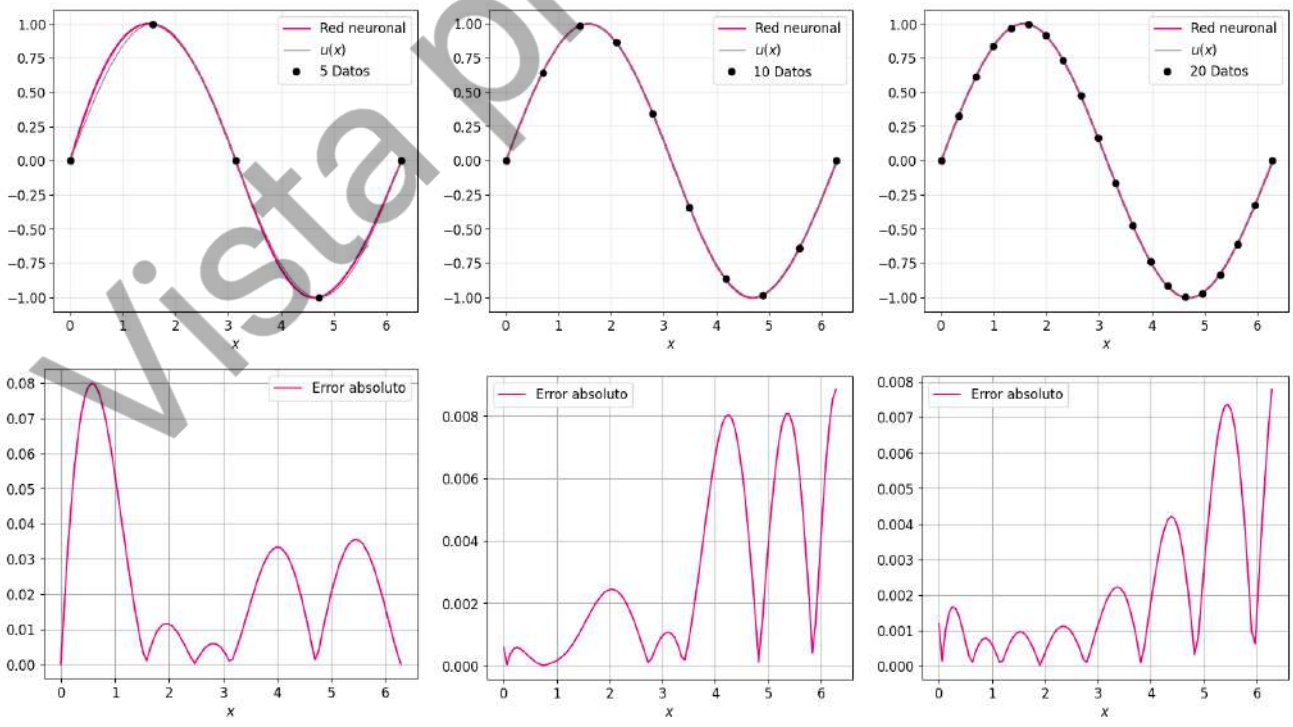


Figura 3.50 Resultados usando una red neuronal de una sola capa oculta con 100 neuronas.

Vista previa 2026

```
Run   1: Max=2.52e-02 -> ÉXITO
Run   2: Max=1.96e-02 -> ÉXITO
Run   3: Max=1.80e-02 -> ÉXITO
Run   4: Max=1.21e-02 -> ÉXITO
Run   5: Max=2.40e-02 -> ÉXITO
Run   6: Max=2.83e-02 -> ÉXITO
Run   7: Max=2.62e-02 -> ÉXITO
Run   8: Max=1.07e-01 -> FALLO
...
Run  98: Max=4.04e-02 -> ÉXITO
Run  99: Max=2.39e-02 -> ÉXITO
Run 100: Max=3.58e-02 -> ÉXITO

==============================
Corridas exitosas : 81/100
Porcentaje éxito  : 81.0%
==============================
```

Figura 3.51 Ejemplo del comportamiento del error máximo en múltiples simulaciones independientes usando 100 neuronas. Se definió como simulación exitosa aquella cuya norma máxima del error es inferior a 0.05.

No obstante, los resultados mantienen el comportamiento observado en experimentos anteriores: debido a la aleatoriedad en la inicialización de los pesos, pueden obtenerse soluciones distintas al ejecutar repetidamente el programa, algunas de las cuales presentan imprecisiones locales. Aun así, se observa un incremento en el número de simulaciones exitosas, es decir, en aquellas donde la red logra un ajuste adecuado de la función objetivo, como veremos a continuación.

3.6.6 Porcentaje de éxito del algoritmo

Finalmente, para analizar el nivel de éxito del algoritmo, se ejecutó un conjunto de simulaciones independientes que difieren únicamente en la inicialización aleatoria de los pesos de la red neuronal. Las Figuras 3.52 y 3.53 presenta los resultados correspondientes a 100 ejecuciones utilizando una red neuronal con 5 y 100 neuronas en la única capa oculta y 40 puntos de entrenamiento. En dichas figuras, uno de los ejes horizontales representa el dominio de interés de la función objetivo, $[0, 2\pi]$, mientras que el otro eje horizontal indica el índice de la simulación $\{1, 2, \ldots, 100\}$. El eje vertical muestra tanto la solución $\widetilde{u}$ como el error absoluto, evaluados sobre los mismos 100 puntos de prueba.

Se observa que la red neuronal con $N_e = 5$ presenta un número considerable de casos fallidos. Sin embargo, cuando los valores iniciales de los parámetros son adecuados, la red puede alcanzar una solución precisa; no obstante, dado que dichos valores se eligen de manera aleatoria en el Código, no es posible controlar este aspecto del proceso de entrenamiento. Por otro lado, prácticamente en todos los casos, la red neuronal logra recuperar correctamente la forma general de la función seno usando $N_e = 100$ neuronas, aunque con distintos niveles de precisión.

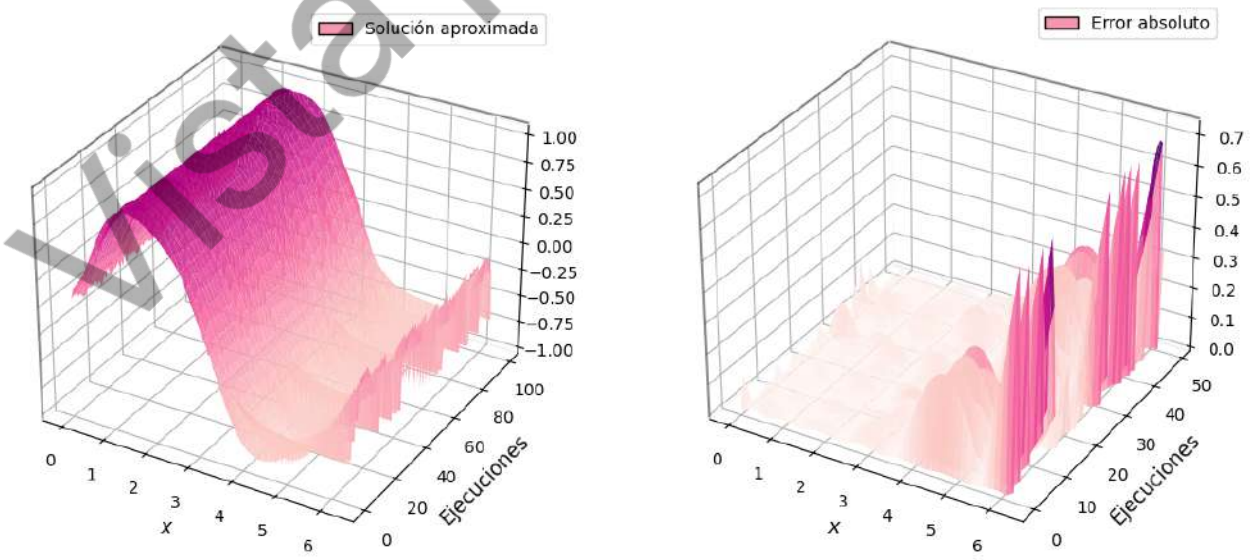


Figura 3.52 Aproximación de la función objetivo y su error absoluto usando una red con 5 neuronas en la capa oculta y 40 datos conocidos. Los resultados corresponden a 100 simulaciones independientes

Vista previa 2026

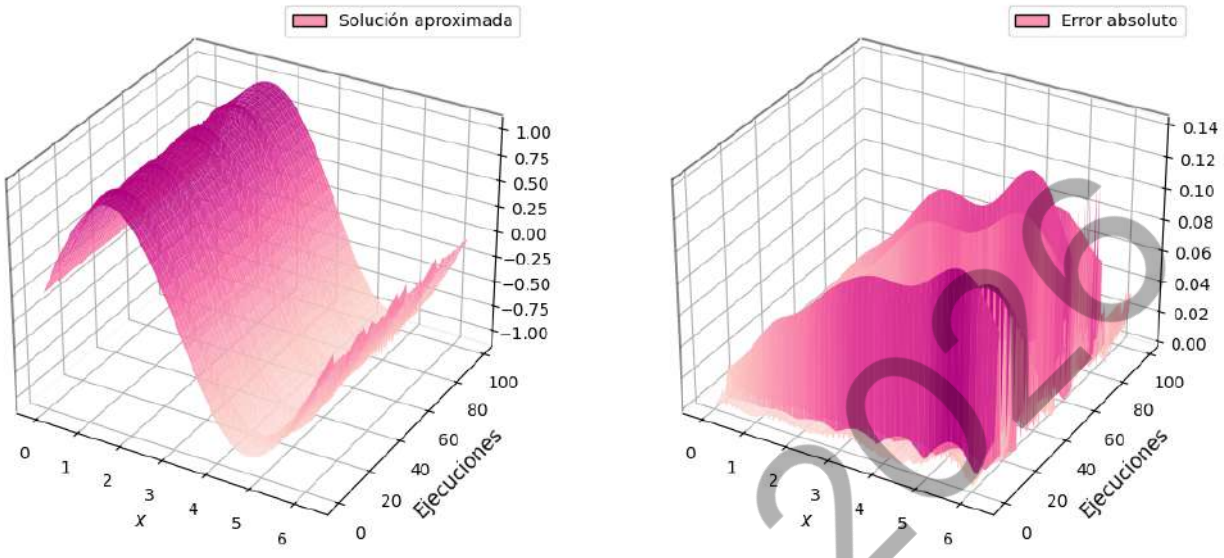


Figura 3.53 Aproximación de la función objetivo y su error absoluto usando una red con 100 neuronas en la capa oculta y 40 datos conocidos. Los resultados corresponden a 100 simulaciones independientes.

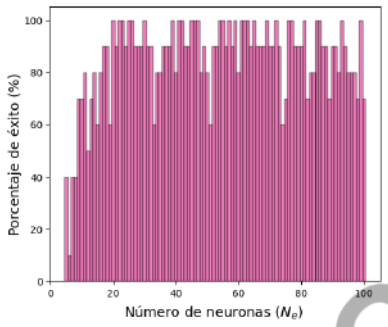


Figura 3.54 Análisis del número de neuronas fijando los demás parámetros. Las estadísticas corresponden a 100 simulaciones independientes usando 40 datos de entrenamiento.

Para cuantificar el grado de éxito, se introduce un criterio basado en un umbral para el error máximo. Aquellas simulaciones cuyo error máximo se encuentra por debajo de dicho umbral se consideran exitosas, mientras que las restantes se clasifican como fallidas. Por ejemplo, en la Figura 3.51 se muestran algunas estadísticas obtenidas al fijar este umbral en 5×10^{-2} para el ejemplo con 100 neuronas. Cabe destacar que, al relajar el criterio a 10^{-1}, prácticamente todas las simulaciones pueden considerarse exitosas, alcanzando un porcentaje de éxito cercano al 100 %.

Gracias al porcentaje de éxito, es posible realizar un análisis de sensibilidad de los parámetros de la red neuronal propuesta. En este trabajo, nos enfocamos únicamente en el estudio del efecto del número de neuronas, manteniendo fijos los demás parámetros del modelo. Análisis similares para otros casos, como el número de datos conocidos, se dejan como ejercicio para el lector.

La Figura 3.54 presenta el análisis correspondiente al número de neuronas bajo esta configuración. Las estadísticas mostradas se obtuvieron a partir de 100 simulaciones independientes, utilizando 40 datos de entrenamiento. Se puede observar que, en general, un mayor número de neuronas conduce a un mayor porcentaje de éxito. No obstante, esta tendencia no constituye una regla estricta en este caso, ya que se presentan comportamientos atípicos, como el correspondiente a $Ne = 20$.

Vista previa 2026



Figura 3.55 Efectivamente, la formulación para analizar datos aleatorios es prácticamente la misma.

3.7 AJUSTE DE UNA CURVA A DATOS ALEATORIOS

La última sección de este capítulo está dedicada al ajuste de una curva a partir de un conjunto de datos aleatorios, de manera análoga a como se realizaría mediante el método de mínimos cuadrados. En este contexto, se evidencia una de las principales ventajas de las redes neuronales: no es necesario introducir una metodología adicional distinta a la ya presentada en las secciones anteriores.

3.7.1 Resultados con datos aleatorios

En el ejemplo que se presenta a continuación, los datos de entrenamiento se generan de manera aleatoria, lo cual nos permitirá observar el comportamiento de la red neuronal en un escenario controlado.

EL EJEMPLO
Consideremos la función definida por

$$u(x) = \sin(x) + \eta, \quad x \in [0, 2\pi], \tag{3.46}$$

donde $\eta \sim \mathcal{N}(0, \sigma^2)$ denota un término de ruido gaussiano. La cantidad de ruido se determina eligiendo diferentes valores de σ.

Es importante mencionar que pequeñas modificaciones al Código 3.6 (utilizado para interpolar) son suficientes para abordar este problema. En particular, basta con reemplazar la generación de los datos basada en la función seno por un procedimiento que lea directamente los datos disponibles o que los genere de forma aleatoria, como lo presenta el Código 3.7.

```
# 1. Training data (20 points)
N = 40
x_train = np.linspace(0, 2*np.pi, N).reshape(-1, 1)

sigma = 0.2               # Desviacion estandar (ruido)
noise = np.random.normal(0.0, sigma, size=x_train.shape)

u_train = np.sin(x_train) + noise

x_tensor = torch.tensor(x_train, dtype=torch.float32)
u_tensor = torch.tensor(u_train, dtype=torch.float32)
```

Código 3.7 Modificación del programa usando *PyTorch* para el ajuste de curvas con datos aleatorios.

Notas

```
N  = 40,
Ne = 100,
Np = 100,
sigma = 0.1, 0.2.
```

Figura 3.56 Parámetros.

Nuestro análisis se centra en dos conjuntos de puntos aleatorios, los cuales se generan introduciendo ruido gaussiano aditivo $\eta \sim \mathcal{N}(0, \sigma^2)$, considerando los valores $\sigma = 0.1$ y $\sigma = 0.2$. Por otro lado, se emplea una red neuronal con una sola capa oculta compuesta por 100 neuronas y se utilizan 40 datos conocidos para el entrenamiento, en concordancia con la configuración presentada en las secciones anteriores.

Vista previa 2026

Los resultados numéricos correspondientes a este experimento se muestran en la Figura 3.57. Como era de esperarse, a medida que aumenta la magnitud del ruido, el ajuste de la curva se vuelve más complejo para la red neuronal. En particular, se observa un deterioro progresivo en la calidad de la aproximación conforme se incrementa el valor de σ.

Aunque los resultados cuantitativos detallados no se presentan aquí, el porcentaje de éxito disminuye de manera considerable, lo cual pone de manifiesto la necesidad de emplear arquitecturas más complejas, tales como redes neuronales con múltiples capas ocultas, para capturar adecuadamente la estructura de los datos en presencia de ruido significativo. El análisis de dichos casos se deja como ejercicio para el lector.

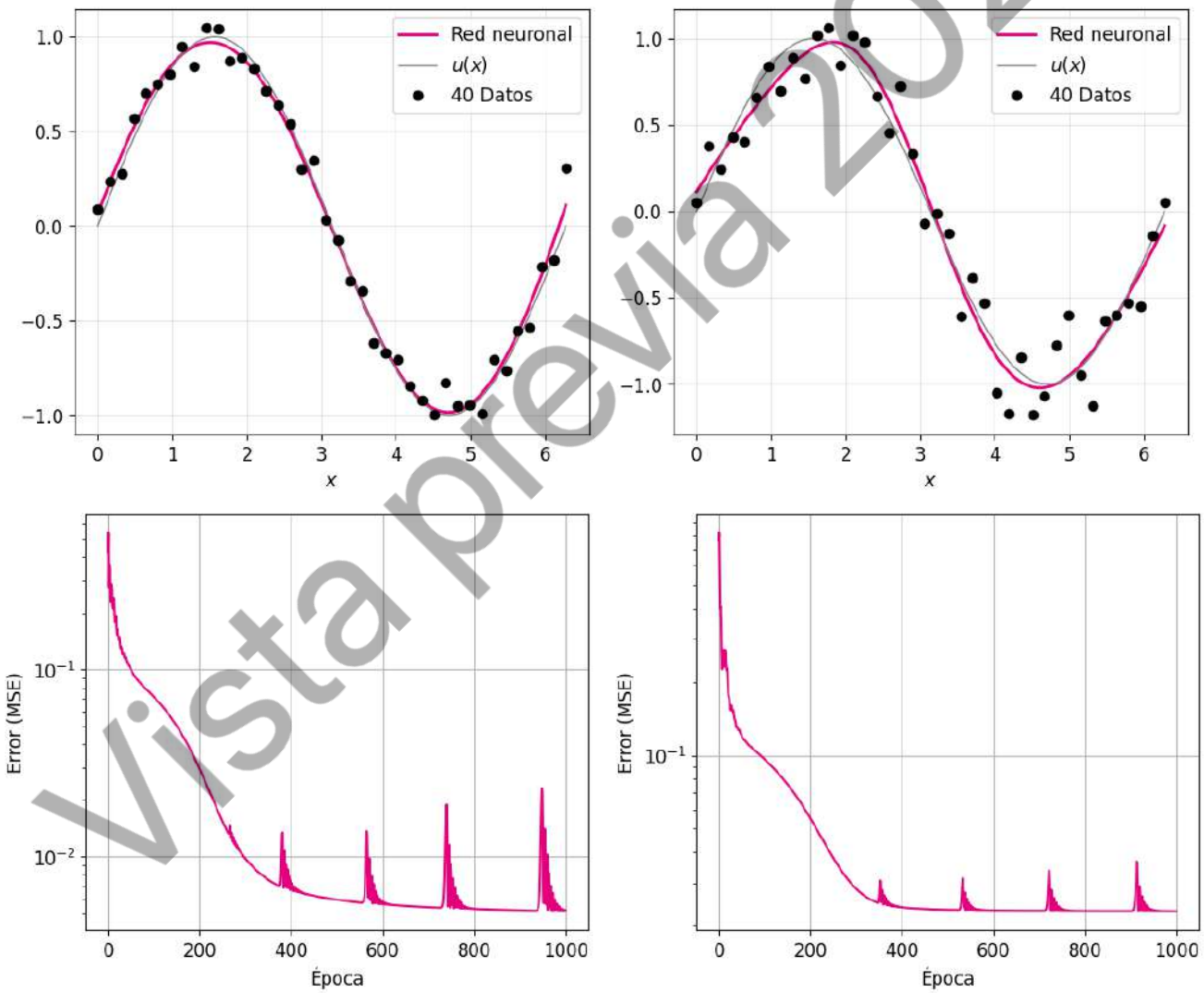


Figura 3.57 Resultados usando una red de una capa oculta con 100 neuronas y con 40 datos conocidos. Los puntos fueron generados introduciendo ruido gaussiano $\eta \sim \mathcal{N}(0, \sigma^2)$ con $\sigma = 0.1$ y $\sigma = 0.2$.

Vista previa 2026

RESUMEN

Los resultados numéricos obtenidos demuestran que una red neuronal simple de una sola capa es capaz de ajustar una curva de manera exitosa. En particular, la red logra capturar la forma general de la función objetivo incluso cuando se dispone de una cantidad limitada de datos de entrenamiento. En particular, el modelo logra reproducir su comportamiento general incluso cuando se dispone de un número limitado de datos de entrenamiento. No obstante, se observan variaciones entre distintas ejecuciones, principalmente debido a la inicialización aleatoria de los pesos y sesgos.

El incremento en el número de puntos de entrenamiento proporciona un mejor ajuste a la curva. Sin embargo, el modelo también converge en múltiples ocasiones a soluciones incorrectas. De manera similar, modificaciones en la arquitectura del modelo, como el aumento del número de neuronas, contribuyen a mejorar la calidad de la aproximación. Sin embargo, aun con estas mejoras, la red neuronal de una sola capa oculta presenta limitaciones en la tasa de éxito, lo que pone de manifiesto la necesidad de considerar arquitecturas más complejas, como redes neuronales con múltiples capas.

PARÁMETROS CLAVE

Al emplear redes neuronales para la aproximación de funciones, es importante considerar los siguientes parámetros clave:

- **Arquitectura de la red:**
  - Número de capas ocultas.
  - Número de neuronas por capa.
  - Función de activación utilizada.
- **Datos de entrenamiento:**
  - Cantidad de datos disponibles.
  - Distribución de los puntos.
  - Calidad de los valores conocidos.
- **Inicialización de pesos:**
  - Método utilizado.
- **Función de pérdida:**
  - El error cuadrático medio.
- **Algoritmo de optimización:**
  - Tipo de optimizador.
  - Número total de épocas de entrenamiento.
- **Evaluación del modelo:**
  - Comparación con los valores reales.
  - Visualización del error.
  - Uso de datos de prueba distintos a los de entrenamiento.

Vista previa 2026

EJERCICIOS

Realiza los siguientes ejercicios con atención, asegurándote de comprender cada uno de los pasos y de aplicar correctamente los conceptos estudiados. Dedica el tiempo necesario para resolverlos con cuidado. En el libro de soluciones se pueden encontrar las respuestas correspondientes, así como los códigos utilizados. Sin embargo, se recomienda recurrir a este material únicamente para verificar los resultados obtenidos o en caso de presentar dificultades durante la resolución de los mismos.

Ejercicio 3.1: Código de la función de pérdida. Modifica el Código 3.6 para registrar y graficar la evolución de la función de pérdida durante el entrenamiento, como se muestra en la Figura 3.36. Realiza la gráfica en escala lineal y en escala logarítmica.

- Utiliza este código para analizar el ejemplo de aproximación de sen(x), empleando una red neuronal con una sola capa oculta de 10 neuronas y 10 puntos conocidos. Entrena la red durante al menos 1000 épocas.
- Finalmente, analiza los resultados obtenidos respondiendo: ¿la función de pérdida converge?, ¿qué se puede concluir sobre la calidad de la aproximación con este número de puntos de entrenamiento?

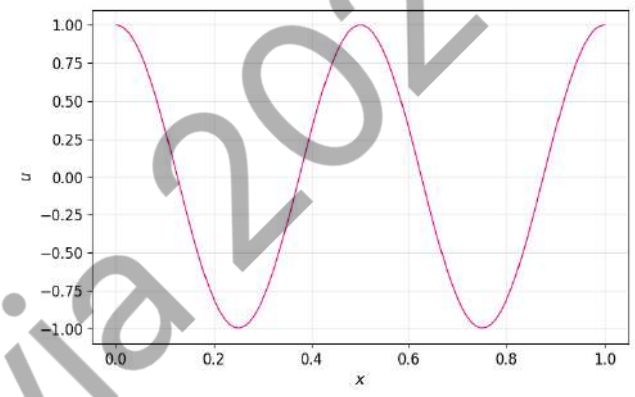
Ejercicio 3.2: Valores explícitos de los pesos y sesgos. Modifica el Código 3.6 para extraer de manera explícita los pesos y sesgos entrenados de la red neuronal. Con estos valores, construye una nueva función utilizando la estructura analítica de la red dada en la ecuación (3.45):

$$\hat{u}(x) = \sum_{j=1}^{N_e} W_{1j}^{[2]} \tanh\left(W_{j1}^{[1]} x + b_j^{[1]}\right) + b^{[2]},$$

- Escribe de forma explícita la función resultante, sustituyendo directamente los coeficientes obtenidos.
- Verifica que esta representación explícita reproduce, con buena precisión, la solución obtenida previamente mediante la red neuronal.
- Grafica la función reconstruida utilizando un programa distinto de *Python*, con el fin de corroborar que los valores aprendidos permiten recuperar la función objetivo de manera independiente del entorno de entrenamiento.

Ejercicio 3.3: Ajuste de una curva con datos obtenidos de la función coseno. En este ejercicio se analizará la aproximación de la función $\cos(4\pi x)$ en el intervalo $[0, 1]$ mediante una red neuronal artificial con una sola capa oculta compuesta por 10 neuronas. Se considerarán distintos conjuntos de puntos de entrenamiento y, en cada caso, se deberán graficar tanto la aproximación obtenida como el error absoluto asociado.

- Varía el número de puntos de entrenamiento equidistantes y analiza los resultados para $N = 10$, 20 y 40. Discute si es necesario emplear un mayor número de puntos para lograr una buena aproximación.
- Repite el experimento utilizando puntos de entrenamiento generados de forma aleatoria con `np.random.uniform`. Compara los resultados con los obtenidos al usar puntos equidistantes y comenta las diferencias observadas.



Ejercicio 3.4: Análisis del número de puntos de entrenamiento. En este ejercicio se estudia la influencia del número de puntos de entrenamiento en la calidad de la aproximación obtenida. Para ello, considere $\sin(x)$ en el intervalo $[0, 2\pi]$ y una red neuronal artificial con una sola capa oculta y $N_e = 10$ neuronas. Construya cuatro conjuntos de entrenamiento con $N = 10, 20, 40$ y 100 puntos equidistantes.

- Al finalizar el entrenamiento, registre el valor final de la función de pérdida (*loss*) y la norma del error L_2 entre la aproximación y la solución exacta evaluada en el conjunto de prueba.
- Para cada valor de N, represente gráficamente la función aproximada junto con la función exacta, incorporando también los datos de entrenamiento, de forma análoga a la presentada en la sección de resultados.
- Analice los resultados obtenidos comparando el desempeño de la red para cada configuración. En particular, discuta cómo influye el número de puntos de entrenamiento en la calidad de la aproximación.

Vista previa 2026

Ejercicio 3.5: Análisis de la arquitectura de la red. A continuación, se analiza la influencia de la arquitectura de la red utilizando el mismo ejemplo desarrollado en este capítulo.

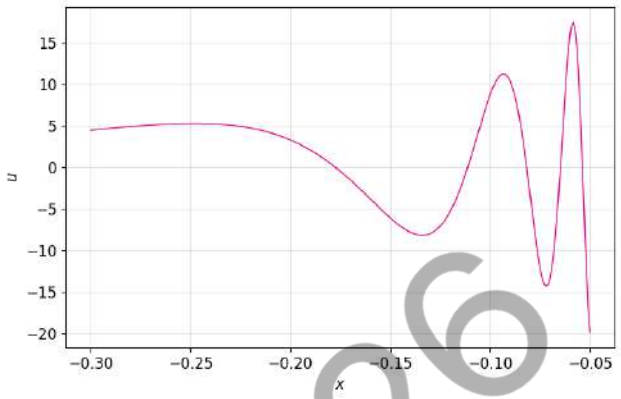
- Manteniendo fijo el número de puntos de entrenamiento, entrena redes neuronales con dos capas ocultas empleando $N_e = 5$, 10, 20 y 100 neuronas en cada capa.
- Repite el experimento considerando una red neuronal con tres capas ocultas, cada una compuesta por 10 neuronas.
- Para cada configuración, reporta la norma del error L_2 y analiza cómo la arquitectura de la red influye en la precisión de la aproximación.

Ejercicio 3.6: Ajuste de una curva altamente oscilatoria. Considere la siguiente función en el intervalo $[-0.3, -0.05]$:

$$u(x) = \left(2x^2 - 1\right)\frac{1}{x}\sin\left(\frac{1}{x}\right) - 4\cos\left(\frac{1}{x}\right).$$

Aunque $u(x)$ es una función suave, presenta un comportamiento altamente oscilatorio al aproximarse al origen. Este tipo de funciones representa un reto para las redes neuronales.

- Analice las dificultades de aproximar esta función y proponga una arquitectura de red neuronal adecuada, justificando el número de capas, neuronas y funciones de activación.
- Diseñe un esquema de entrenamiento que contemple una mayor densidad de puntos cerca del origen para capturar las oscilaciones.
- Realice la simulación, grafique la función exacta y la aproximación obtenida, y comente los resultados.



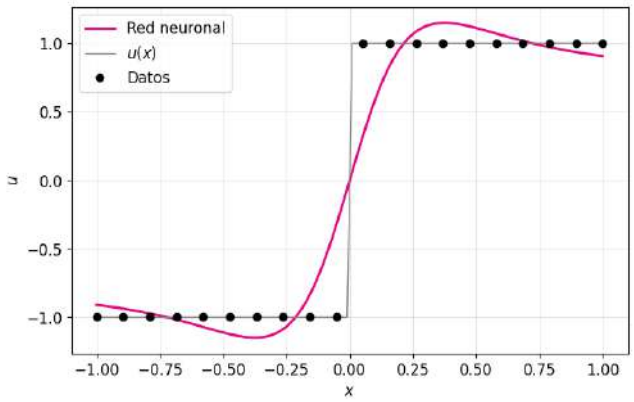
Ejercicio 3.7: Ajuste de una curva con datos de comportamiento discontinuo. Consideremos la siguiente función en el intervalo $[-1, 1]$:

$$u(x) = \begin{cases} -1 & x \leq 0, \\ 1 & x > 0. \end{cases}$$

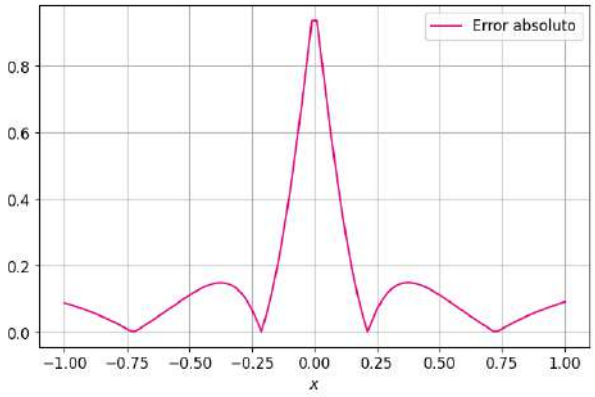
Esta función presenta una discontinuidad en $x = 0$, lo que la distingue de manera significativa de los ejemplos analizados previamente, en los cuales las soluciones eran suaves.

- Analizar los retos que implica aproximar funciones discontinuas mediante redes neuronales, en contraste con el caso de funciones suaves. Considere, en particular, la capacidad del modelo para capturar transiciones bruscas, así como el impacto de la discontinuidad en el comportamiento de la función de pérdida durante el entrenamiento.

A modo de ilustración, en las figuras que se presentan a continuación se muestran algunos resultados obtenidos al emplear una red neuronal con una sola capa oculta de 5 neuronas y 100 puntos de entrenamiento.

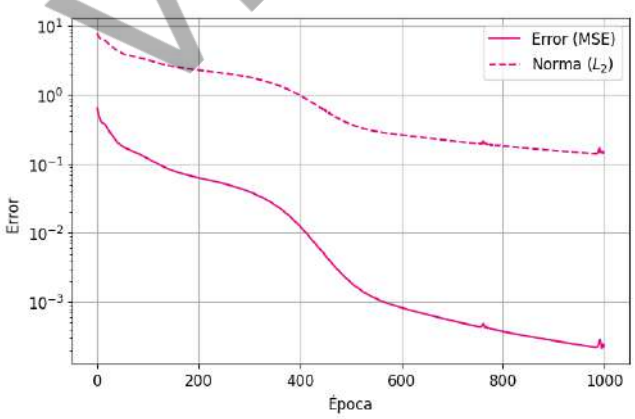


Vista previa 2026



Ejercicio 3.8: Análisis del número de épocas seleccionado. Uno de los hiperparámetros clave en el proceso de optimización es la elección del número de épocas. Hasta ahora se ha utilizado un valor de 1000, el cual fue seleccionado de manera arbitraria. Aunque el error cuadrático medio puede seguir disminuyendo al incrementar el número de épocas, esto no garantiza necesariamente una mejor aproximación de la función objetivo. Con el fin de analizar este comportamiento, en este ejercicio se te pide:

- Modificar el Código 3.6 para graficar, además de la evolución de la función de pérdida durante el entrenamiento, la norma del error L_2 entre la solución obtenida por la red neuronal y la solución analítica, como se muestra en la gráfica inferior.
- Utilizar este código para estudiar el problema de aproximación de $\text{sen}(x)$, empleando una red neuronal con una sola capa oculta de N_e neuronas y N puntos de entrenamiento.
- Variar de manera progresiva el número de épocas para analizar la evolución de la solución durante el entrenamiento.



Ejercicio 3.9: Comparación con otros métodos de interpolación. La función gamma se define como

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt, \quad x > 0.$$

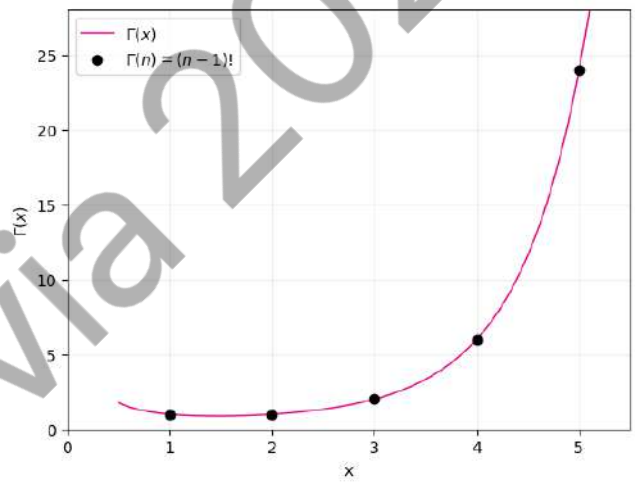
Para un argumento entero n, la función gamma satisface

$$\Gamma(n) = (n-1)!.$$

Interpolando los puntos de datos

$$t = 1, 2, 3, 4, 5, \qquad y = 1, 1, 2, 6, 24,$$

se obtiene una aproximación de la función gamma en el rango dado.



- Calcular el polinomio de grado cuatro que interpola estos cinco puntos. Graficar el polinomio resultante junto con los valores de la función gamma en el dominio $[1, 5]$.
- Usar una rutina de *cubic spline* para interpolar los mismos datos y graficar la curva resultante junto con los valores de la función gamma en el dominio $[1, 5]$.
- Usar la rutina de red neuronal para interpolar los mismos datos y graficar la curva resultante junto con los valores de la función gamma en el dominio $[1, 5]$.
- Determinar cuál de los tres interpolantes es más preciso en la mayor parte del dominio.
- Determinar cuál de los tres interpolantes es más preciso en el intervalo entre 1 y 2.

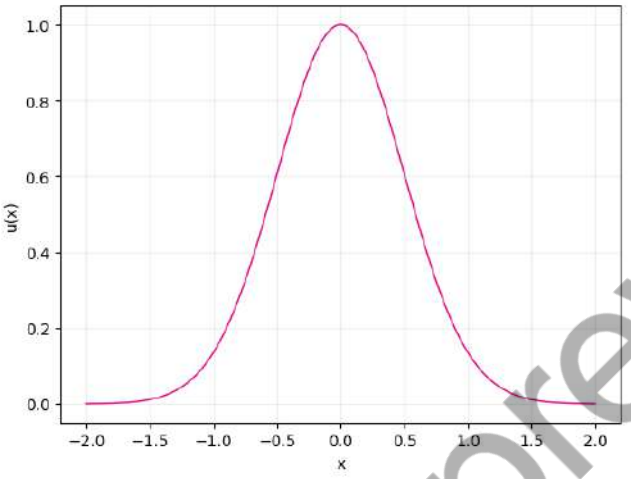
La función gamma está definida en *Python* en la paquetería *Scipy*.

Vista previa 2026

Ejercicio 3.10: Análisis sistemático de parámetros en una red neuronal en el ajuste de curvas. En este ejercicio se estudia a modo de resumen, de manera experimental y controlada, cómo distintos parámetros de una red neuronal influyen en su capacidad para aproximar la función

$$f(x) = \exp(-2x^2), \qquad x \in [-2, 2].$$

El análisis se realiza variando *dos parámetros a la vez*, manteniendo el resto fijo, mediante múltiples experimentos independientes. El objetivo final es identificar la configuración que produce el mejor desempeño global.



Definición de éxito. Cada configuración se evalúa mediante 100 entrenamientos independientes. Una ejecución se considera exitosa si cumple simultáneamente: MSE final $< 10^{-3}$, error absoluto promedio en validación $< 5 \times 10^{-3}$.

Definición de porcentaje de éxito. El porcentaje de éxito se define como el número de ejecuciones exitosas dividido entre 100, y se utiliza como puntaje principal de cada experimento.

Configuración base. Salvo que se indique lo contrario, todos los experimentos comparten la siguiente configuración:

- ☐ Tipo de red: perceptrón multicapa.
- ☐ Entrada y salida: una dimensión.
- ☐ Función de activación: tanh en capas ocultas.
- ☐ Función de pérdida: error cuadrático medio.
- ☐ Optimizador: Adam.
- ☐ Intervalo: $[-2, 2]$.
- ☐ Puntos de validación: 100 equidistantes.
- ☐ Número de pruebas por configuración: 100.

Etapa 1: influencia del número de épocas. Se estudia el efecto del número de épocas en el desempeño de la red, manteniendo fija la arquitectura.

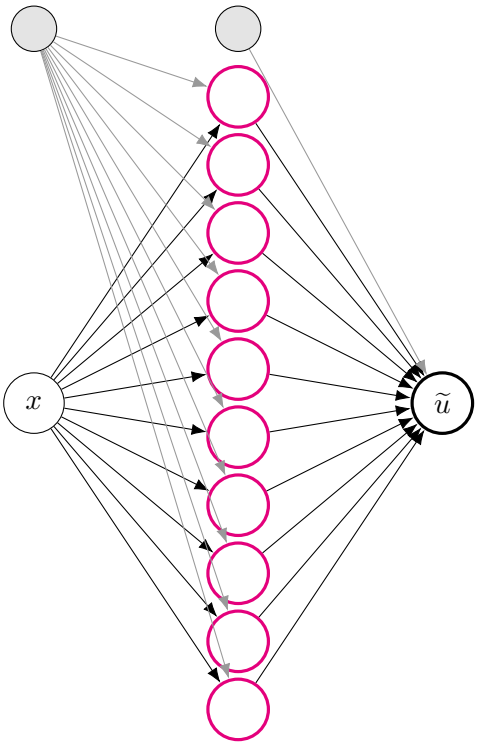
Parámetros fijos.

- Capas ocultas: 1.
- Neuronas en la capa oculta: 10.
- Tasa de aprendizaje: 0.01.

Parámetros variables.

- Número de épocas: 1000 a 5000 con incrementos de 500.
- Puntos de entrenamiento: $10, 20, 30, 40, 50$.

Para cada combinación (puntos,épocas) se calcula el porcentaje de éxito. Se identifica el número de épocas que maximiza el desempeño sin sobreentrenamiento.



Vista previa 2026

- Etapa 2: influencia de la tasa de aprendizaje. En esta etapa se analiza cómo la tasa de aprendizaje afecta tanto el porcentaje de éxito como el tiempo de entrenamiento.
  - Parámetros fijos.
    - Épocas: 1000.
    - Capas ocultas: 1.
    - Neuronas por capa: 10.
  - Parámetros variables.
    - Tasa de aprendizaje: $0.001, 0.01, 0.05, 0.1$.
    - Puntos de entrenamiento: $10, 20, 30, 40, 50$.

  Para cada configuración se reporta: Porcentaje de éxito y tiempo promedio de entrenamiento.

- Etapa 3: puntos de entrenamiento contra profundidad. Se estudia la relación entre la cantidad de datos y la profundidad de la red.
  - Parámetros fijos.
    - Neuronas por capa oculta: 10.
    - Tasa de aprendizaje: 0.01.
    - Épocas: 1000.
  - Parámetros variables.
    - Puntos de entrenamiento: $10, 20, 30, 40, 50$.
    - Capas ocultas: 1 a 10.

    .

  Se analiza cómo el aumento de la complejidad afecta la generalización y se discute la presencia de sobreajuste

- Etapa 4: neuronas contra puntos de entrenamiento. Se evalúa el efecto combinado del número de neuronas y la densidad de datos.
  - Parámetros fijos.
    - Capas ocultas: 1.
    - Tasa de aprendizaje: 0.01.
    - Épocas: 1000.
  - Parámetros variables.
    - Neuronas en la capa oculta.
    - Puntos de entrenamiento.

  Los resultados se organizan en una tabla donde las filas representan N_e y las columnas el número de neuronas.

- Etapa 5: balance entre complejidad y datos. Se profundiza el análisis de la relación entre arquitectura y tamaño del conjunto de entrenamiento.
  - Parámetros variables.
    - Capas ocultas: 1 a 10.
    - Puntos de entrenamiento: 10 a 50.

  Se analiza simultáneamente: Porcentaje de éxito y error promedio de validación.

- Etapa 6: selección de la mejor red. A partir de las etapas anteriores, se construye una red final combinando:
  - El número de épocas óptimo.
  - La tasa de aprendizaje más estable.
  - La arquitectura con mayor porcentaje de éxito.

  Esta red se entrena nuevamente y se compara su desempeño contra configuraciones base para validar la mejora obtenida.

- Etapa final. Ahora se analizan aspectos adicionales del entrenamiento de redes neuronales que no fueron considerados previamente.
  - Parte A: sensibilidad a la inicialización. Se fija una arquitectura óptima y se entrena la red múltiples veces con distintas inicializaciones de pesos. Se analiza la variabilidad del error final y del porcentaje de éxito.
  - Parte B: robustez al ruido. Se agrega ruido gaussiano a los datos de entrenamiento y se estudia cómo se modifica el desempeño de la red.
  - Parte C: comparación de funciones de activación. Se repite el experimento principal utilizando las funciones ReLU y sigmoide, comparando convergencia y estabilidad.
  - Parte D: curvas de aprendizaje. Se analizan las curvas de pérdida de entrenamiento y validación para detectar sobreajuste o subajuste.

Vista previa 2026

Vista previa 2026

CAPÍTULO 4

REDES NEURONALES INFORMADAS POR LA FÍSICA PARA ECUACIONES EN 1D

Vista previa 2026

Este capítulo introduce la metodología de Redes Neuronales Informadas por la Física (PINNs) aplicada a problemas diferenciales unidimensionales. Se desarrolla el marco necesario para aproximar soluciones mediante redes neuronales, incorporando tanto la información de frontera como las restricciones diferenciales en el interior del dominio. Debido a que en esta región usualmente no se dispone de datos observacionales, la ecuación diferencial se integra directamente en la función de pérdida como mecanismo de regularización física.

Una vez establecida la metodología, se resuelven problemas de referencia y se comparan los resultados con métodos clásicos, como diferencias finitas, con el propósito de validar la implementación y analizar las diferencias entre ambas estrategias numéricas. Posteriormente, se estudia el comportamiento del modelo al modificar hiperparámetros, arquitecturas y condiciones de entrenamiento, permitiendo evaluar el impacto de estas decisiones en la calidad de la solución aproximada. Este análisis permite identificar fortalezas, limitaciones y posibles líneas de mejora dentro del esquema propuesto.

El estudio en una dimensión proporciona un entorno controlado para comprender los mecanismos fundamentales de la metodología, facilitando el análisis del comportamiento de las PINNs antes de abordar problemas de mayor complejidad.

OBJETIVOS

Al finalizar este capítulo, el lector será capaz de:

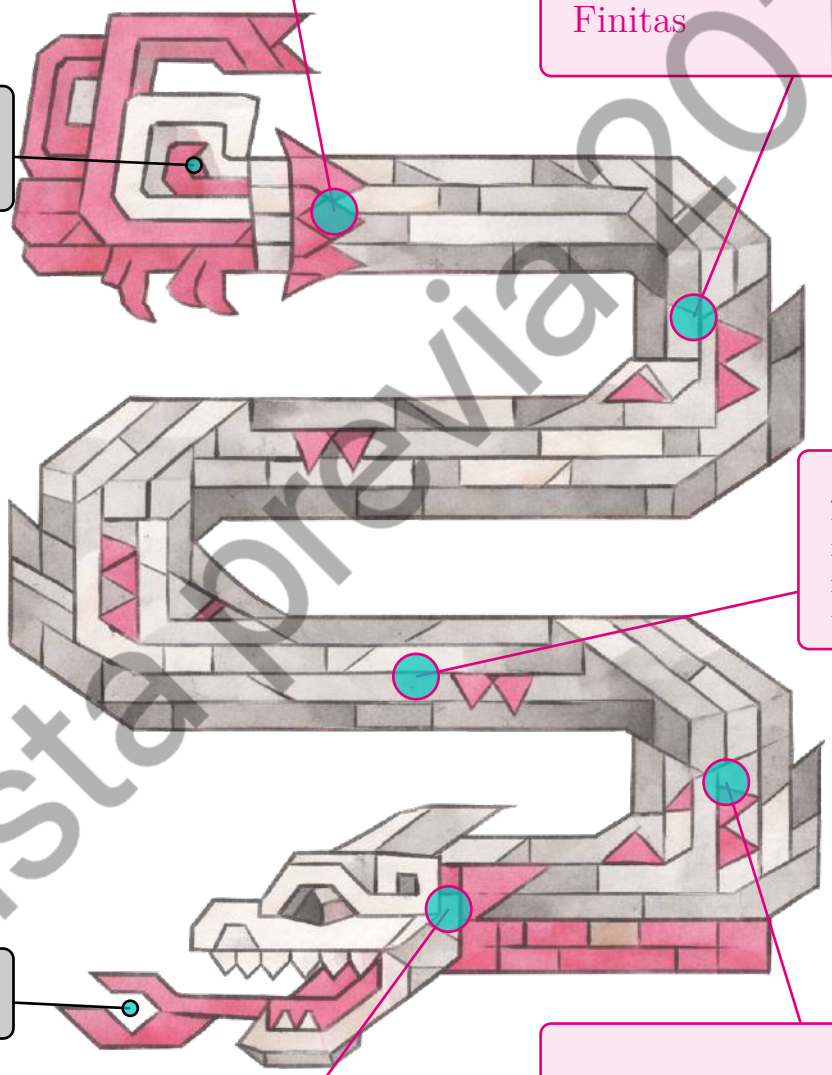
- Comprender los fundamentos de las Redes Neuronales Informadas por la Física aplicadas a ecuaciones diferenciales en una dimensión.
- Implementar y modificar una PINN básica para aproximar soluciones de problemas diferenciales 1D.
- Comparar el desempeño de las PINNs con métodos numéricos clásicos, como diferencias finitas.
- Analizar el efecto de distintas configuraciones de entrenamiento y arquitectura en la calidad de la aproximación.

CONTENIDO

4.2 Ecuaciones Elípticas 1D

4.3 Método de Diferencias Finitas

4.1 Introducción



4.4 Método de Redes Neuronales Informadas por la Física

Resumen

4.5 Implementación en Python

4.6 Resultados Numéricos usando las PINNs

Vista previa 2026



Figura 4.1 Clásico o moderno.

4.1 INTRODUCCIÓN

Como se describió en el Capítulo 1, el modelado constituye una parte fundamental del desarrollo científico, ya que permite describir fenómenos físicos con precisión y estimar magnitudes asociadas como la presión, la velocidad o la temperatura. Estas cantidades ofrecen una representación del fenómeno y, en general, mientras más preciso es el modelo, mayor es la confiabilidad de los resultados obtenidos.

Sin embargo, el refinamiento de los modelos matemáticos suele venir acompañado de un incremento en su complejidad. En muchos casos, esto conduce de manera natural a ecuaciones diferenciales más elaboradas, cuya solución analítica resulta difícil o incluso inalcanzable. Es en este contexto donde los métodos numéricos se vuelven indispensables, al proporcionar aproximaciones fieles de los problemas planteados. A su vez, problemas más complejos demandan métodos numéricos más robustos, eficientes y, en consecuencia, más sofisticados.

En esta dirección, el desarrollo completo de estas herramientas puede extenderse hacia problemas de mayor complejidad. No obstante, en esta unidad se opta por trabajar en el caso unidimensional, con el objetivo de evitar que la dificultad del problema matemático opaque la comprensión de la metodología. De esta forma, la atención se centra en los principios del enfoque numérico más que en complejidad del modelo.

Tradicionalmente, gran parte de las ecuaciones diferenciales se estudian como problemas de valor inicial (PVI), ampliamente utilizados en contextos como el modelado poblacional, la dinámica de sistemas físicos y otros procesos evolutivos. Sin embargo, dado que el enfoque de este trabajo está orientado hacia problemas con condiciones de frontera en dimensiones mayores, se prioriza el estudio de problemas de valor en la frontera (PVF) en una dimensión. Este enfoque permite establecer las bases metodológicas necesarias en un entorno más controlado, antes de abordar escenarios de mayor complejidad.

A diferencia de los enfoques tradicionales basados en datos, las PINNs incorporan directamente las leyes físicas del problema, expresadas como ecuaciones diferenciales, dentro del proceso de entrenamiento.

Figura 4.2 Las redes neuronales informadas por la física y las leyes físicas.

En este capítulo se estudian dos enfoques para aproximar la solución de los problemas planteados: un método clásico, basado en diferencias finitas, y un enfoque moderno fundamentado en redes neuronales informadas por la física (*Physics-Informed Neural Networks*, PINNs), introducidas por Raissi et al. [12], que constituyen el eje central de este libro. Con el objetivo de presentar la metodología de manera clara y sistemática, se considera un problema modelo sencillo de tipo elíptico. Este ejemplo permite ilustrar paso a paso la implementación de ambos enfoques y, en particular, facilita identificar cómo se adapta el esquema de interpolación mediante redes neuronales artificiales para aproximar la solución de la ecuación diferencial conocida como PINNs.

Vista previa 2026

Las condiciones de frontera describen la interacción del sistema modelado con su entorno en los extremos del dominio. Mientras que las condiciones de tipo Dirichlet especifican el valor de la solución, las de tipo Neumann determinan el comportamiento de su derivada, es decir, el flujo o la transferencia de información.

Figura 4.3 Las condiciones de frontera son importantes.

4.2 ECUACIONES ELÍPTICAS 1D

Para contar con un problema de referencia que permita evaluar los métodos implementados en esta sección, se propone trabajar con una ecuación diferencial de tipo elíptico. Este problema se analizará posteriormente junto con su solución exacta, la cual servirá como base para la comparación y validación de los resultados numéricos. Asimismo, permitirá cuantificar los errores de aproximación y analizar la precisión de los métodos considerados bajo distintas condiciones de discretización.

PROBLEMA ELÍPTICO

Consideramos el problema de aproximar la función u que satisface:

$$\frac{d^2u}{dx^2}(x) + \kappa(x)u(x) = g(x), \quad x \in \Omega = (a, b), \tag{4.47}$$

donde g y κ son funciones conocidas y definidas sobre el intervalo cerrado $[a, b]$. La función g se le conoce como término fuente y κ es el coeficiente de reacción.

Cabe aclarar que la función solución u depende de una única variable independiente x y es diferenciable en el intervalo (a, b). Dado que se trata de un intervalo acotado, su frontera está formada por los puntos $\{a, b\}$, que corresponden a sus extremos.

Además de la ecuación diferencial, es necesario especificar las condiciones de frontera para garantizar la existencia y unicidad de la solución. Estas condiciones incorporan la interacción del sistema con el entorno en los extremos del dominio. En general, pueden ser de tipo Dirichlet, Neumann o Robin; sin embargo, en esta primera aproximación se consideran únicamente condiciones de tipo Dirichlet y Neumann, así como combinaciones simples de ambas, con el fin de mantener el enfoque en los casos más básicos.

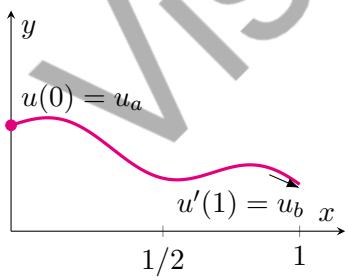


Figura 4.4 Las condiciones de frontera de Dirichlet especifica el valor en el $x = 0$ y las de Neumann especifican la pendiente de la tangente en $x = 1$.

CONDICIONES DE FRONTERA

Para el problema (4.47), se pueden considerar algunas de las siguientes condiciones de frontera:

$$u(a) = u_a, \quad u(b) = u_b, \quad \text{(Dirichlet)} \tag{4.48}$$
$$u'(a) = q_a, \quad u'(b) = q_b, \quad \text{(Neumann)} \tag{4.49}$$
$$u'(a) = q_a, \quad u(b) = u_b, \quad \text{(mixta)} \tag{4.50}$$
$$u(a) = u_a, \quad u'(b) = q_b, \quad \text{(mixta)} \tag{4.51}$$

donde u_a, u_b, q_a, q_b son constantes conocidas que representan, respectivamente, el valor de la función o su derivada en los extremos del dominio.

Vista previa 2026

Notas

Si $\kappa = 0$, la ecuación (4.52) se reduce a la ecuación de Poisson en una dimensión.
Si además $g(x) \equiv 0$, se obtiene la ecuación de Laplace en una dimensión.

Figura 4.5 Las condiciones de frontera son importantes.

Bajo condiciones adecuadas de regularidad sobre los datos del problema, la ecuación diferencial (4.47) junto con condiciones de frontera de tipo Dirichlet (4.48) admite una única solución u. De manera similar, combinaciones bien planteadas de condiciones de tipo Neumann o mixtas también conducen a problemas bien definidos, aunque pueden requerir condiciones adicionales para asegurar unicidad.

Para fijar ideas, se define un problema concreto que servirá como caso de estudio. Este se resolverá primero mediante una técnica estándar y, posteriormente, utilizando PINNs.

EJEMPLO PARTICULAR

Consideramos el problema de valores en la frontera:

$$\frac{d^2u}{dx^2}(x) + \kappa u(x) = g(x), \quad x \in (0,1), \tag{4.52}$$

$$u(0) = 1, \tag{4.53}$$

$$u(1) = e, \tag{4.54}$$

donde

$$g(x) = e^x \left[(1 - 4\pi^2 + \kappa)\cos(2\pi x) - 4\pi \sin(2\pi x)\right] \tag{4.55}$$

y κ es una constante. La solución analítica de este problema es

$$u(x) = e^x \cos(2\pi x). \tag{4.56}$$

Obsérvese que el problema anterior fue seleccionado de manera que se conoce su solución exacta, lo cual permite utilizarla como referencia para verificar la correcta implementación tanto del método de diferencias finitas como de las PINNs. Además, este planteamiento facilita el análisis del orden de precisión del método numérico, así como la evaluación del error absoluto.

En la siguiente sección se describe la implementación del método de diferencias finitas para este problema en particular, considerando κ como un parámetro.



Figura 4.6 Retrato de Leonhard Euler.

4.3 MÉTODO DE DIFERENCIAS FINITAS

Uno de los métodos numéricos más clásicos y ampliamente utilizados para la aproximación de ecuaciones diferenciales es el método de diferencias finitas (DFs). Sus orígenes se remontan a los trabajos de Leonhard Euler en el siglo XVIII, y desde entonces ha evolucionado hasta convertirse en una herramienta fundamental en el análisis numérico. Su permanencia en el tiempo no es casual: destaca por su simplicidad conceptual, su facilidad de implementación y su versatilidad para abordar una amplia variedad de problemas.

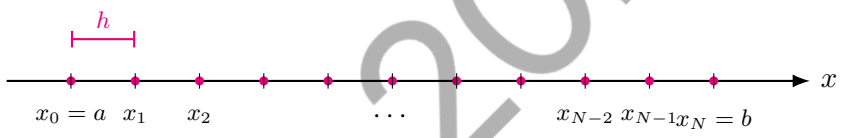
vista previa 2026



Figura 4.7 Retrato de Brook Taylor.

El método de diferencias finitas consiste en aproximar las derivadas presentes en una ecuación diferencial mediante expresiones discretas evaluadas sobre una malla. Esta malla se construye como un conjunto ordenado de puntos en el dominio, donde se busca aproximar la solución. Al sustituir las derivadas por estas aproximaciones, el problema continuo se transforma en un sistema de ecuaciones algebraicas que puede resolverse mediante técnicas computacionales. En lo que sigue, se presenta la formulación y aplicación de este método al problema considerado.

Iniciamos esta discusión dividiendo el intervalo $[a, b]$ en $N+1$ puntos equiespaciados $x_i = a + ih$, donde $h = (b-a)/N$, para $i = 0, \ldots, N$. Definimos $u_i := u(x_i)$ y $g_i := g(x_i)$, y denotamos por U_i la aproximación numérica de u_i, es decir, $U_i \approx u_i$.



Usando el teorema de Taylor, se puede demostrar que la segunda derivada de u en el punto x_i se aproxima mediante

$$\frac{d^2u}{dx^2}(x_i) = \frac{u(x_i+h) - 2u(x_i) + u(x_i-h)}{h^2} + \mathcal{O}(h^2), \tag{4.57}$$



Figura 4.8 Una de las principales ventajas de los métodos numéricos tradicionales es que su error de truncamiento es conocido. En particular, a mayor orden del término de error (es decir, mayor potencia de h), mejor es la calidad de la aproximación al refinar la malla.

donde el término $\mathcal{O}(h^2)$ representa el error de truncamiento, lo que indica que la aproximación es de segundo orden. Sustituyendo (4.57) en la ecuación diferencial (4.52) y evaluando en los puntos de la malla x_i, se obtiene

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \kappa u_i = g_i + \mathcal{O}(h^2), \quad i = 1, \ldots, N-1,$$

donde $u_i = u(x_i)$ y $g_i = g(x_i)$. Despreciando el término de error, se obtiene el siguiente esquema en diferencias finitas:

$$U_{i+1} + (\kappa h^2 - 2)U_i + U_{i-1} = h^2 g_i, \quad i = 1, \ldots, N-1, \tag{4.58}$$

donde U_i aproxima a u_i. Este esquema da lugar a un sistema lineal tridiagonal, en el cual se incorporan las condiciones de frontera $u(0) = u_a$ y $u'(1) = u_b$. Las ecuaciones (4.58) resultan en el siguiente sistema lineal

$$\begin{pmatrix} \kappa h^2 - 2 & 1 & 0 & \cdots & 0 \\ 1 & \kappa h^2 - 2 & 1 & \cdots & 0 \\ 0 & 1 & \kappa h^2 - 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 \\ 0 & 0 & 0 & 1 & \kappa h^2 - 2 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{N-1} \end{pmatrix} = \begin{pmatrix} h^2 g_1 - u_a \\ h^2 g_2 \\ h^2 g_3 \\ \vdots \\ h^2 g_{N-2} - u_b \end{pmatrix}$$

El sistema lineal resultante puede resolverse mediante eliminación gaussiana o algoritmos especializados para matrices tridiagonales. Para más detalles del método de diferencias finitas se puede consultar un libro clásico de análisis numérico como el Burden [60].

Vista previa 2026



NumPy es una biblioteca de Python para cálculos numéricos eficientes con arreglos y operaciones matemáticas avanzadas. La función `np.linspace` genera un arreglo de N datos equidistantes en un intervalo $[a, b]$ considerando extremos.

Figura 4.9 Librería Numpy.

El sistema lineal $A\mathbf{x} = \mathbf{b}$ se resuelve utilizando rutinas de la biblioteca NumPy. No obstante, dado que la matriz A es tridiagonal, también es posible emplear un método más eficiente, conocido como el método de Thomas, diseñado específicamente para este tipo de sistemas.

Figura 4.10 Las condiciones de frontera son importantes.

A continuación se presenta la implementación del método, detallando pasos importantes, como la matriz asociada al sistema y la rutina para obtener la solución numérica del problema. En este caso se emplea $\kappa = 0$, sin embargo el mismo código permite explorar el comportamiento del método numérico al modificar el valor de κ.

```python
import numpy as np
# ========================================================
# Dominio y entradas del modelo

# 1.A Dominio, malla uniforme
a = 0.0
b = 1.0
N = 100   # Número de sub-intervalos
h = (b - a) / N
x = np.linspace(a, b, N+1)

# 1.B Parámetro
kappa = 0.0

# 1.C Función lado derecho y condiciones de frontera
def g(x):
     return np.exp(x)*((1 - 4*np.pi**2 + kappa)*
            np.cos(2*np.pi*x) - 4*np.pi*np.sin(2*np.pi*x))

u_a = 1.0,
u_b = np.exp(1)

# ========================================================
# Solución

# 2.A Construcción matriz
A     = np.zeros((N-1, N-1))
b_vec = np.zeros(N-1)
for i in range(N-1):
    A[i, i] = h**2 * kappa -2
    if i > 0:
        A[i, i-1] = 1
    if i < N-2:
        A[i, i+1] = 1
    b_vec[i] = h**2 * g(x[i+1])

# 2.B Ajuste del vector b por condiciones de frontera
b_vec[0]  =  b_vec[0] -  u_a
b_vec[-1] =  b_vec[-1] - u_b

# 2.C Resolver sistema
u_interior = np.linalg.solve(A, b_vec)

# ========================================================
# 3. Solución del problema
u = np.zeros(N+1)
u[0], u[1:-1], u[-1] = u_a, u_interior, u_b
```

Código 4.8 Implementación en *Python* del método de diferencias finitas para aproximar la solución numérica de la ecuación (4.52).

En la Figura 4.12 se presentan los resultados numéricos del método de diferencias finitas para los casos $N = 10, 20, 40$. En la parte superior se muestra la comparación entre la solución analítica y la aproximación numérica, mientras que en la inferior se presenta el error absoluto correspondiente.

Vista previa 2026

Notas

El error absoluto en x_i se define como $|u_i - U_i|$, donde u_i denota el valor exacto y U_i la aproximación numérica correspondiente.

Figura 4.11 Una forma de medir la aproximación de forma directa.

Notar que al refinar progresivamente la malla, la solución numérica se aproxima cada vez más a la solución exacta, lo que evidencia la precisión del esquema utilizado. Debido a este comportamiento se dice que el método es consistente, es decir, el error disminuye conforme el tamaño de paso h se reduce. Además, a partir de las gráficas del error absoluto, se aprecia que su valor máximo se reduce aproximadamente a una cuarta parte cada vez que h se divide entre dos. Este comportamiento es consistente con un método de segundo orden, ya que el error es proporcional a h^2.

En la Figura 4.12 se observa una implementación consistente del método numérico, ya que los errores de mayor magnitud no se concentran en las fronteras del dominio. En particular, el error en dichas regiones es cercano a cero y de magnitud comparable al observado en el interior, lo cual sugiere un comportamiento estable y una adecuada imposición de las condiciones de frontera. Este comportamiento es consistente con la teoría subyacente del método. En los esquemas clásicos, el error puede controlarse en función del tamaño de paso, y su orden de convergencia está determinado por el término de truncamiento en la expansión de Taylor asociada.

Resulta natural complementar el análisis teórico con un estudio experimental del error, a fin de verificar empíricamente la tasa de convergencia esperada.

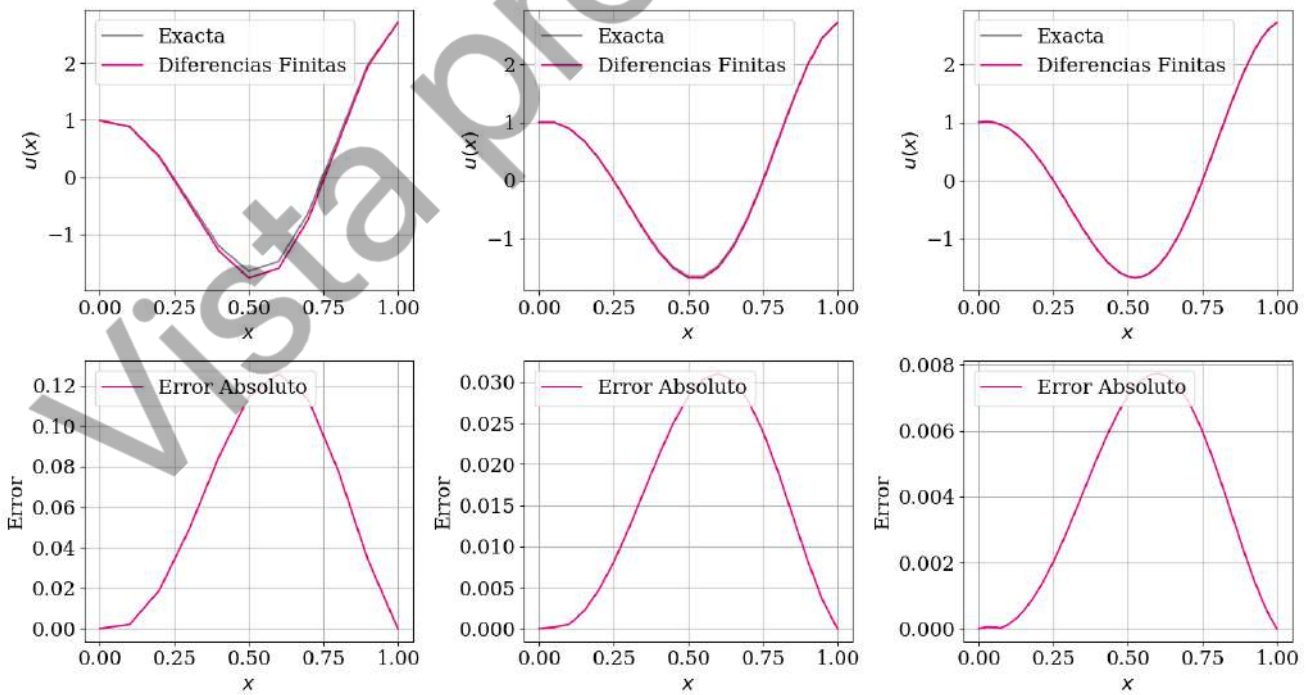


Figura 4.12 Resultados del método de diferencias finitas con $\kappa = 0$ y $N = 10$, 20 y 40 subdivisiones. En la parte superior se muestra la comparación con la solución exacta, mientras que en la inferior se presenta el error absoluto.

Vista previa 2026

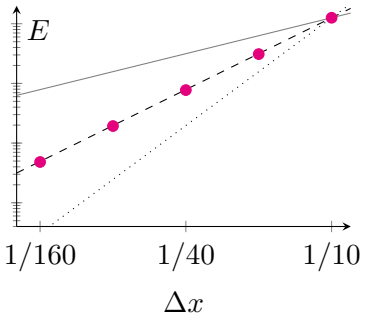


Figura 4.13 Los errores medidos en la norma L_∞ se alinean sobre una recta de pendiente 2 en la gráfica log-log. Con fines comparativos, se incluyen también rectas de referencia con pendientes 1 y 3. En consecuencia, el ajuste de los datos confirma una pendiente cercana a 2, en concordancia con el orden de precisión teórico del método.

La Tabla 4.1 presenta las normas L_2 y L_∞ del error absoluto para distintos valores de N. Como es de esperarse, al refinar la discretización (esto es, al aumentar N y disminuir el tamaño de paso), ambas normas decrecen de manera sistemática. Asimismo, se incluyen columnas adicionales que reportan el orden de precisión numérica observado.

DEFINICIÓN

El orden p de un método numérico se define a partir de la relación asintótica $E(h) \approx Ch^p$, donde $E(h)$ es el error asociado a un tamaño de paso h y C es una constante independiente de h. A partir de esta relación, el orden puede estimarse experimentalmente mediante la expresión

$$p \approx \frac{\log(E_1/E_2)}{\log(h_1/h_2)},$$

donde E_1 y E_2 corresponden a los errores obtenidos con tamaños de paso h_1 y h_2, respectivamente. Esta estimación permite cuantificar la convergencia y contrastarla con el orden teórico.

Tabla 4.1 Errores absolutos y órdenes de convergencia observados del método de diferencias finitas bajo diferentes valores de N para el ejemplo (4.52) con $\kappa = 0$, con las normas L_2 y L_∞.

N	Norma L_2	Orden	Norma L_∞	Orden
10	7.67×10^{-2}	–	1.26×10^{-1}	–
20	1.89×10^{-2}	2.02	3.10×10^{-2}	2.02
40	4.71×10^{-3}	2.00	7.73×10^{-3}	2.00
80	1.17×10^{-3}	2.00	1.93×10^{-3}	2.00
160	2.94×10^{-4}	2.00	4.82×10^{-4}	2.00

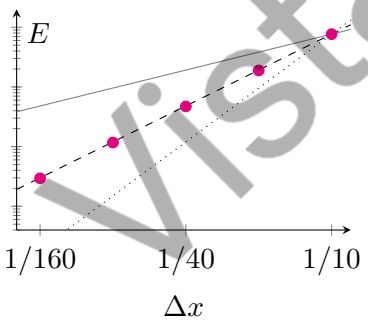


Figura 4.14 Los errores medidos en la norma L_2 se alinean sobre una recta de pendiente aproximadamente 2 en la gráfica log–log. Con fines comparativos, se incluyen también rectas de referencia con pendientes 1 y 3.

La Tabla 4.2 presenta las normas L_2 y L_∞ del error absoluto usando $\kappa = 1$ para distintos valores de N. Se obtiene resultados similares, incluso en el caso cuando el coeficiente de reacción es distinto de cero. Dado que el análisis detallado de la influencia de este parámetro κ escapa a los objetivos del presente texto, no se profundiza en este punto. No obstante, esta herramienta podrá emplearse más adelante cuando sea necesario establecer comparaciones con otros enfoques, como las redes neuronales informadas por la física.

Tabla 4.2 Errores absolutos y órdenes de convergencia observados del método de diferencias finitas bajo diferentes valores de N para el ejemplo (4.52) con $\kappa = 1$, con las normas L_2 y L_∞.

N	Norma L_2	Orden	Norma L_∞	Orden
10	8.48×10^{-2}	–	1.38×10^{-1}	–
20	2.09×10^{-2}	2.02	3.39×10^{-2}	2.02
40	5.21×10^{-3}	2.01	8.45×10^{-3}	2.01
80	1.30×10^{-3}	2.00	2.11×10^{-3}	2.00
160	3.25×10^{-4}	2.00	5.28×10^{-4}	2.00

Vista previa 2026



Figura 4.15 Una frase popular en la región para iniciar una tarea sin temor a involucrarse directamente en el trabajo es: "manos a la obra".

4.4 MÉTODO DE REDES NEURONALES INFORMADAS POR LA FÍSICA

En esta sección se presenta la metodología para utilizar redes neuronales en la resolución de ecuaciones diferenciales, siguiendo una idea análoga a la empleada previamente en el ajuste de curvas mediante interpolación. Sin embargo, en este nuevo contexto la ecuación diferencial adquiere un papel tan relevante como los datos disponibles, ya que ambos contribuyen de manera conjunta a la formulación del problema.

Las Redes Neuronales Informadas por la Física (PINNs, por sus siglas en inglés) no se construyen únicamente a partir de observaciones, sino que se propone una función parametrizada basada en la ecuación diferencial, la cual se incorpora en la función de pérdida. De esta forma, la optimización del modelo se realiza de manera que la red satisfaga, en la mayor medida posible, las condiciones impuestas por dicha ecuación, así como los datos disponibles correspondientes a condiciones iniciales o de frontera.

De forma similar al caso del ajuste de curvas, se introduce una arquitectura de red neuronal. En particular, se emplea una red neuronal multicapa para aproximar la solución de ecuaciones diferenciales. El proceso general se resume en la Figura 4.16.

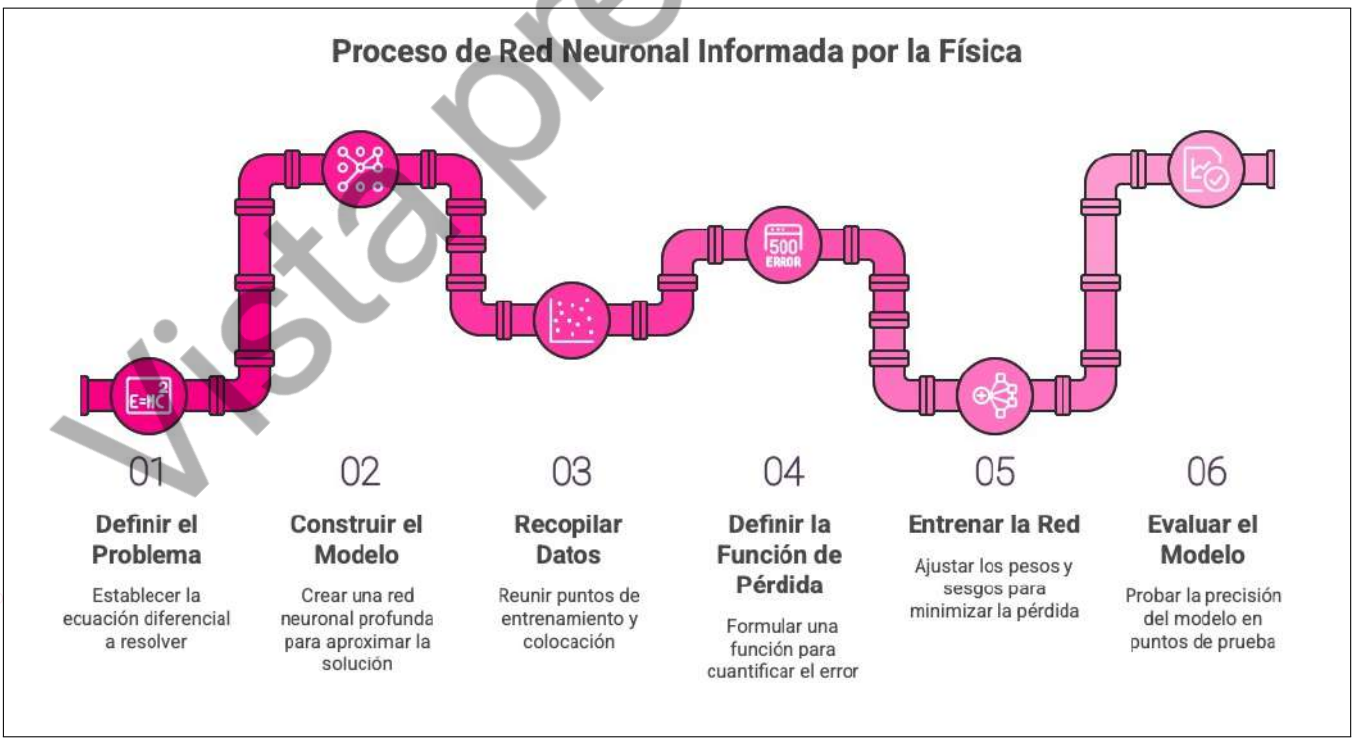


Figura 4.16 Proceso para la metodología de Redes Neuronales Informadas por la Física.

Vista previa 2026

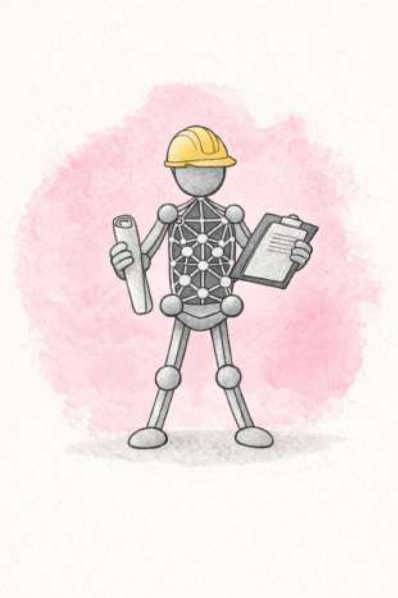


Figura 4.17 Construcción del modelo teórico.

Aunque el desarrollo de esta sección se centra en el caso específico de la ecuación (4.47), el esquema es general y su implementación puede adaptarse de manera directa a otras ecuaciones elípticas. En este sentido, se destacan los elementos que deben modificarse en el modelo propuesto, tales como la formulación de la ecuación, la generación o selección de datos y el procedimiento de entrenamiento de la red.

4.4.1 Definir el modelo

El primer ejemplo que resolveremos es (4.52). No obstante, como se verá, la metodología se adapta sin mayor dificultad a variantes del problema. En particular, se describirán los ajustes necesarios para considerar condiciones de frontera mixtas, combinando una condición de tipo Dirichlet con otra de tipo Neumann. Con este propósito, se analiza la siguiente modificación de (4.52), en la cual se sustituye la condición (4.54).

CONDICIÓN TIPO NEUMANN

Junto con el problema (4.52) y (4.53), consideramos el valor de frontera:

$$\frac{du}{dx}(x) = q_{\mathfrak{n}}(x), \quad x \in \partial\Omega. \tag{4.59}$$

En nuestro caso particular, si esta condición es aplicada únicamente al lado derecho del dominio, entonces $q_{\mathfrak{n}}(x) = e$ para $x = 1$.

Como se verá en las siguientes subsecciones, la metodología se presentará en el contexto de problemas unidimensionales y mediante un caso concreto. Sin embargo, el procedimiento que se ilustra con este ejemplo puede extenderse a otras problemáticas, esto es, a distintas ecuaciones y dimensiones, realizando las adaptaciones en cada situación.

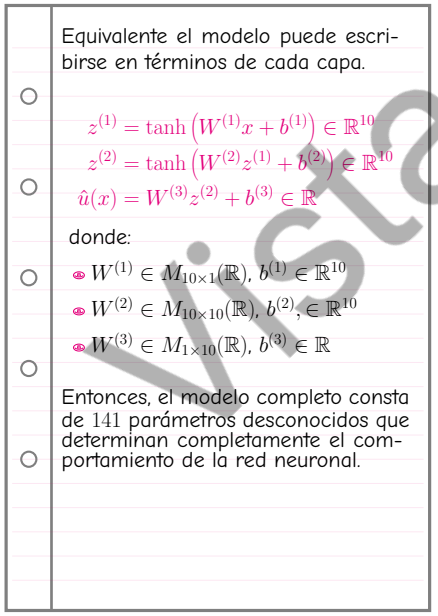


Figura 4.18 Formas equivalentes de escribir matemáticamente la red neuronal propuesta.

4.4.2 Construir el modelo de red neuronal

Se inicia definiendo la estructura básica del modelo, es decir, la arquitectura de la red neuronal. Esta red corresponde a una arquitectura totalmente conectada. La red se utiliza como una función parametrizada que recibe la variable independiente como entrada y produce una aproximación de la solución del problema a través de una secuencia de transformaciones entre sus capas.

Se propone aproximar $u(x)$ mediante una red neuronal multicapa de la siguiente forma:

$$u(x) \approx \tilde{u}(x; W, b), \tag{4.60}$$

donde W y b representan los pesos y sesgos de la red neuronal, respectivamente. Lo parámetros W y b se ajustan durante el entrenamiento del modelo para minimizar una función de pérdida definida posteriormente.

Vista previa 2026

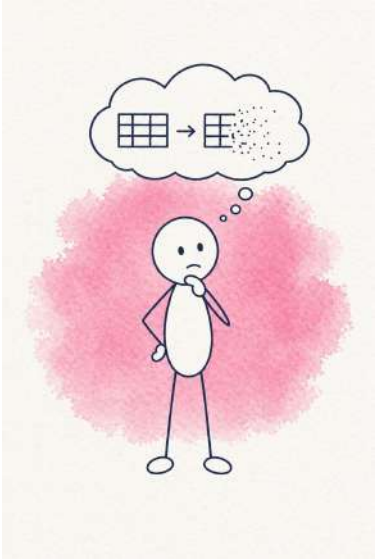


Figura 4.19 No se dispone de una gran cantidad de datos, como teniamos en el problema de ajuste de curvas.

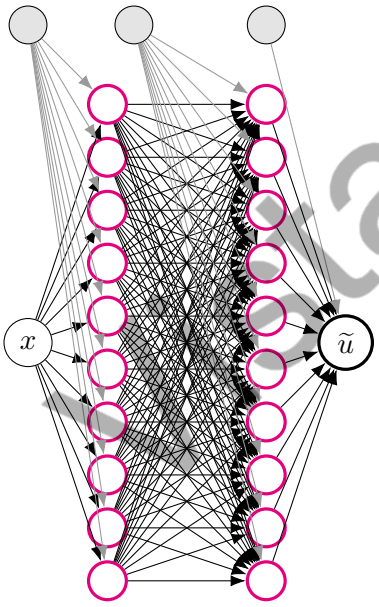


Figura 4.20 Arquitectura de la red neuronal: dos capas ocultas, cada una con 10 neuronas y función de activación tanh, seguidas de una capa de salida lineal.

Dado que el objetivo es aproximar una función de una sola variable independiente, la arquitectura puede describirse de manera más precisa como sigue:

- Capa de entrada: una única neurona asociada a la variable independiente del problema.
- Capas ocultas: dos capas densamente conectadas, cada una con 10 neuronas y activación tanh, lo que introduce no linealidad y permite capturar estructuras complejas en la solución.
- Capa de salida: una neurona sin función de activación, lo que equivale a una transformación lineal que produce la aproximación final de la función buscada.

La Figura 4.20 presenta de forma esquemática la arquitectura de la red neuronal multicapa utilizada. Esta representación permite visualizar claramente la estructura jerárquica del modelo y el flujo de información desde la entrada hasta la salida, facilitando la interpretación de su funcionamiento.

Con la arquitectura propuesta, es necesario contar con información que permita ajustar los parámetros de la red durante el entrenamiento. Sin embargo, en este tipo de problemas no siempre se dispone de un conjunto de datos en el sentido clásico. En particular, lo único que se conoce de forma explícita son las condiciones iniciales o de frontera, que pueden representarse como pares $(\hat{x}_i, \hat{u}_i)$.

A partir de esta información parcial, el resto del conocimiento del problema se incorpora directamente desde la ecuación diferencial que lo gobierna. De esta manera, la información física deja de depender exclusivamente de datos observados y pasa a estar codificada en la propia formulación del problema.

En lo que sigue, se describe cómo esta información proveniente de la ecuación diferencial se transforma en una forma adecuada para ser utilizada durante el entrenamiento de una red neuronal tipo PINN, integrando tanto las condiciones conocidas como las restricciones impuestas por el modelo físico.

4.4.3 Recopilar datos

A diferencia de la unidad anterior, donde se abordó el problema de ajuste de curvas con datos disponibles en todo el dominio, en este caso la información es considerablemente más limitada. En particular, los datos conocidos se restringen a las condiciones de frontera. Más aún, cuando se imponen condiciones de tipo Neumann, no se conoce directamente el valor de la solución, sino únicamente el de sus derivadas, lo que reduce aún más la información disponible para reconstruirla.

vista previa 2026



Figura 4.21 No se dispone de una gran cantidad de datos, por lo que resulta necesario incorporar la ecuación diferencial en el proceso de entrenamiento

Sin embargo, el problema no está completamente desprovisto de información. En todo el dominio se conoce la función g, correspondiente al lado derecho de la ecuación (4.52). Esto permite incorporar el conocimiento del modelo físico directamente en el proceso de entrenamiento.

DEFINICIÓN

Se introduce una función auxiliar f, que mide qué tan bien una función candidata satisface la ecuación diferencial. En particular, se define el residuo como

$$f(x) := \frac{d^2u}{dx^2}(x) + \kappa u(x) - g(x). \tag{4.61}$$

De esta forma, el proceso de recolección de datos se amplía: ya no se basa únicamente en pares $(\hat{x}_i, \hat{u}_i)$ en la frontera, sino también en información distribuida en el dominio a través de la ecuación diferencial. Esta idea es precisamente la que da origen a las PINNs, donde la ecuación que gobierna el fenómeno se integra como parte del entrenamiento.

Es posible interpretar el residuo como una segunda red neuronal asociada $f(x) \approx \widetilde{f}(x; W, b)$, donde

$$\widetilde{f}(x; W, b) = \frac{d^2u}{dx^2}(x; W, b) + \kappa u(x; W, b) - g(x), \tag{4.62}$$

es decir, $\widetilde{f}$ depende de la aproximación $\widetilde{u}$ y de sus derivadas. Estas derivadas se calculan de manera eficiente mediante *diferenciación automática* [15], lo que permite incorporar la ecuación diferencial sin necesidad de discretizarla explícitamente.

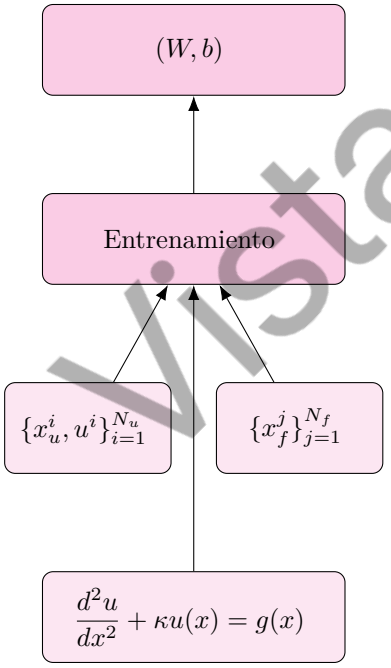


Figura 4.22 Insumos en una PINN.

Es importante destacar que los parámetros (W, b) son los mismos en ambas expresiones, por lo que no se trata de dos redes independientes. En realidad, $\widetilde{f}$ se obtiene a partir de $\widetilde{u}$ mediante una transformación inducida por la ecuación diferencial. Esta formulación permite integrar de manera natural la información física en el proceso de entrenamiento, sin modificar la arquitectura original de la red.

Para determinar los parámetros W y b, se emplean dos conjuntos finitos de puntos asociados a cada tipo de condición:

- Puntos de entrenamiento $\{x_u^i\}_{i=1}^{N_u}$, correspondientes a las condiciones de frontera, para los cuales se conocen los valores de la solución: $\{u^i\}_{i=1}^{N_u}$. En este caso particular, el conjunto consta únicamente de dos puntos, $\{0, 1\}$.

- Puntos de colocación $\{x_f^j\}_{j=1}^{N_f}$, ubicados en el dominio Ω (incluyendo la frontera), donde se evalúa el residuo de la ecuación diferencial.

Vista previa 2026

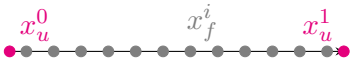


Figura 4.23 En este caso tenemos $N_u = 2$ y $N_f = 12$ en una distribución uniforme.

Esta es una diferencia fundamental respecto a los métodos numéricos clásicos: no es necesario contar con una malla estructurada ni con información de vecindad entre los puntos. La red neuronal puede entrenarse a partir de conjuntos de puntos no ordenados, siempre que estén correctamente asociados a la información que representan (valores de frontera o evaluación del residuo).

De manera análoga al problema de ajuste de curvas, el proceso de entrenamiento combina ambos conjuntos de puntos. La selección de los puntos de colocación juega un papel crucial, ya que de ella depende qué tan bien se incorpora la información del modelo físico en todo el dominio; por esta razón, es común emplear muestreos aleatorios o distribuciones uniformes para mejorar la capacidad de generalización de la red.

Con estos elementos, el entrenamiento se formula como un problema de optimización en el que se busca que la red satisfaga simultáneamente las condiciones de frontera y la ecuación diferencial en el dominio. Esto se logra mediante una función de pérdida compuesta por dos términos: uno que mide el error en los puntos de entrenamiento y otro que cuantifica el residuo en los puntos de colocación, equilibrando así la información de datos y la del modelo físico, como se verá en la siguiente subsección.

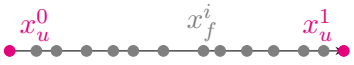


Figura 4.24 En este caso tenemos $N_u = 2$ y $N_f = 12$ en una distribución aleatoria.

4.4.4 Definir la función de pérdida

Una vez definidos los conjuntos de puntos de entrenamiento y de colocación, el siguiente paso consiste en establecer el criterio mediante el cual se ajustarán los parámetros de la red neuronal. En una PINN, el entrenamiento busca determinar los pesos y sesgos de manera que la aproximación $\tilde{u}$ satisfaga simultáneamente las condiciones de frontera y la ecuación diferencial en el dominio.

DEFINICIÓN

Se define una función de pérdida total como la suma de dos contribuciones:

$$E(W,b) = E_u(W,b) + E_f(W,b), \tag{4.63}$$

donde el primer término mide el error asociado a las condiciones de frontera y el segundo cuantifica el incumplimiento de la ecuación diferencial.

En el área de teoría de decisiones se abordan las funciones de perdida como la elección de medición de la información. Lo que en si mismo es una decisión a partir de las propiedades de cada una de las funciones.

Figura 4.25 Comentario sobre funciones de perdida.

El error sobre los puntos de entrenamiento se define como

$$E_u(W,b) = \frac{1}{N_u}\sum_{i=1}^{N_u}\left|\tilde{u}(x_u^i;W,b) - \hat{u}_i\right|^2, \tag{4.64}$$

el cual corresponde al error cuadrático medio entre la salida de la red y los valores conocidos en la frontera.

Vista previa 2026

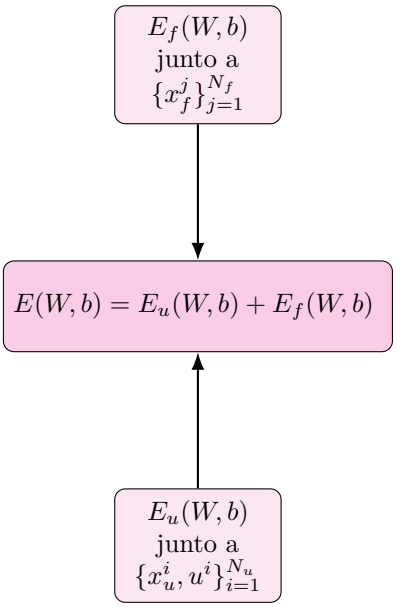


Figura 4.26 Diagrama de la composición del error total, considerando la condición de frontera tipo Dirichlet.

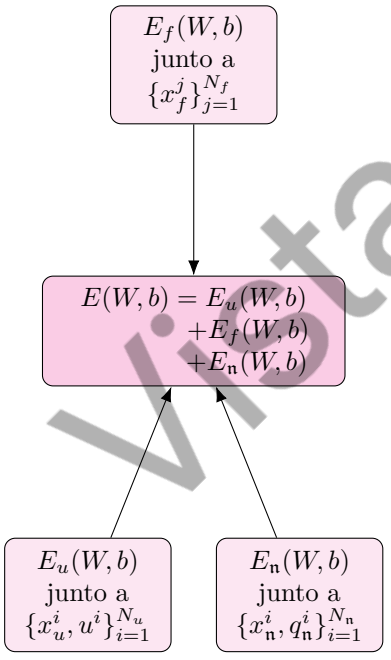


Figura 4.27 Diagrama de la composición del error total, considerando condiciones de frontera tipo Dirichlet y Neumann.

Por otro lado, el error asociado a la ecuación diferencial se define mediante la aproximación del residuo $\widetilde{f}$ introducido previamente en (4.62):

$$E_f(W,b) = \frac{1}{N_f}\sum_{j=1}^{N_f}\left|\widetilde{f}(x_f^j; W, b)\right|^2, \tag{4.65}$$

ya que, por construcción, para la solución exacta debe cumplirse f(x)=0, y por tanto se espera que $\widetilde{f}^j = \widetilde{f}(x_f^j; W, b) \approx 0$. Por tanto, minimizar E_f equivale a forzar que la aproximación neuronal satisfaga la ecuación diferencial en los puntos de colocación.

De manera análoga, esta formulación puede extenderse para incorporar condiciones de frontera de tipo Neumann, las cuales imponen información sobre la derivada de la solución en lugar de su valor. En este caso, se introduce un término adicional en la función de pérdida que penaliza el error entre la derivada aproximada por la red y los valores prescritos en la frontera:

$$E_{\mathfrak{n}}(W,b) = \frac{1}{N_g}\sum_{i=1}^{N_g}\left|\frac{d\widetilde{u}}{dx}(x_{\mathfrak{n}}^i; W, b) - q_{\mathfrak{n}}^i)\right|^2, \tag{4.66}$$

donde $\{(x_{\mathfrak{n}}^i, q_{\mathfrak{n}}^i)\}_{i=1}^{N_{\mathfrak{n}}}$ corresponde al conjunto donde se conocen valores de la derivada. Por lo tanto, la función de pérdida total se redefine como

$$E(W,b) = E_u(W,b) + E_f(W,b) + E_{\mathfrak{n}}(W,b), \tag{4.67}$$

permitiendo incorporar simultáneamente condiciones de Dirichlet, restricciones físicas y condiciones de Neumann dentro del proceso de entrenamiento.

Así, la función de pérdida total combina todas las fuentes de información complementarias: por un lado, los datos disponibles en la frontera y, por otro, las restricciones físicas impuestas por el modelo matemático. Esta formulación constituye el núcleo de las PINNs, pues permite entrenar la red incluso cuando la cantidad de datos observados es limitada. A continuación, se presenta el proceso de entrenamiento de la red y los métodos de optimización utilizados para minimizar la función de pérdida.

4.4.5 Entrenar la red

El proceso de entrenamiento de una PINN es análogo al utilizado en problemas de ajuste de curvas, con la diferencia principal en la construcción de la función de pérdida, la cual incorpora tanto información proveniente de datos como restricciones físicas del problema. Debido a esta estructura, el entrenamiento requiere estrategias de optimización más cuidadosas para lograr una convergencia adecuada.

Vista previa 2026



Figura 4.28 Un *buen* entrenamiento de la red producirá mejores soluciones numéricas.

La minimización de (4.63) se realiza mediante algoritmos de optimización basados en gradiente. Entre los métodos más comunes se encuentran el descenso de gradiente estocástico (*stochastic gradient descent*, SGD) y Adam (*Adaptive Moment Estimation*). Aunque SGD constituye una alternativa robusta y ampliamente utilizada por su simplicidad y eficiencia computacional, en este trabajo se emplea Adam, ya que su capacidad para adaptar dinámicamente la tasa de aprendizaje de cada parámetro favorece una convergencia más estable y rápida en problemas donde la función de pérdida presenta múltiples escalas, como ocurre frecuentemente en PINNs.

La elección de los hiperparámetros, tales como el número de épocas y la tasa de aprendizaje, resulta determinante en el desempeño del entrenamiento. En particular, dado que la función de pérdida total está compuesta por varios términos, es común que durante distintas etapas del entrenamiento alguno de ellos domine temporalmente sobre los demás. Por esta razón, suelen requerirse más épocas que en problemas tradicionales de ajuste de curvas, con el fin de alcanzar un balance adecuado entre los distintos componentes de la pérdida.

Asimismo, es habitual implementar estrategias de disminución progresiva de la tasa de aprendizaje para reducir las oscilaciones del error y refinar la convergencia en etapas avanzadas del entrenamiento. Un ejemplo de este tipo de estrategia se presenta en la secciones posteriores.

4.4.6 Evaluar el método

Un entrenamiento adecuado produce una red neuronal capaz de generalizar el comportamiento de la solución más allá de los puntos utilizados durante el entrenamiento. Para evaluar esta capacidad, se considera el siguiente conjunto de datos:

- Puntos de prueba $\{x_p^k\}_{k=1}^{N_p}$, conformado por puntos con valores conocidos u_p^k que no fueron utilizados durante el entrenamiento.



Figura 4.29 Una vez que la red se entrenó es hora de ponerla a prueba.

Aprovechando la naturaleza continua del modelo, es común utilizar un conjunto de prueba más denso que el conjunto de entrenamiento, es decir $N_p > N_f$, lo que permite evaluar la capacidad de interpolación de la red. En problemas de ecuaciones diferenciales ordinarias, las condiciones de frontera suelen ser escasas y se emplean completamente en el entrenamiento, por lo que no forman parte del conjunto de prueba.

Una práctica habitual consiste en comparar gráficamente la solución aproximada con la solución analítica para evaluar visualmente la calidad del ajuste. Posteriormente, puede realizarse un análisis más detallado mediante métricas como el error absoluto o relativo, con el objetivo de identificar regiones de mayor y menor precisión.

Vista previa 2026

Para la implementación de la función de costos es importante usar las funciones y métodos de `torch` para que el código se ejecute eficientemente.

Figura 4.30 Uso de `Pytorch`.

4.5 IMPLEMENTACIÓN EN PYTHON

En esta sección se presenta la implementación completa en *Python* utilizando *PyTorch* de una red neuronal multicapa con dos capas ocultas de cinco neuronas cada una, empleada para resolver la ecuación diferencial (4.47). La estructura del programa sigue el mismo orden en que fueron introducidos los conceptos fundamentales a lo largo del capítulo. Para ello, el código se ha dividido en dos partes. La primera contiene los elementos básicos del modelo, que corresponden esencialmente a los mismos componentes utilizados en el capítulo anterior. Por su parte, la segunda se centra en la incorporación de la ecuación diferencial al modelo, lo que constituye la principal diferencia respecto al problema estudiado previamente.

```python
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
# ========================================================
# 1. Puntos de entrenamiento y colocacion
xI, xF = 0, 1
uI, uF = 1, np.exp(1)
x_bc = torch.tensor([[xI], [xF]], dtype=torch.float32)
u_bc = torch.tensor([[uI], [uF]], dtype=torch.float32)
Nf = 30
x_train  = np.linspace(xI, xF, Nf).reshape(-1, 1)
x_tensor = torch.tensor(x_train, dtype=torch.float32)

# ========================================================
# 2. Red neuronal multicapa
Ne = 10
class SimpleMLP(nn.Module):
  def __init__(self):
    super().__init__()
    self.hidden1 = nn.Linear(1, Ne)
    self.hidden2 = nn.Linear(Ne, Ne)
    self.output = nn.Linear(Ne, 1)
  def forward(self, x):
    x = torch.tanh(self.hidden1(x))
    x = torch.tanh(self.hidden2(x))
    return self.output(x)
model = SimpleMLP()
```

Código 4.9 Implementación en `PyTorch` para entrenar una red neuronal profunda que aproxima la solución de una ED en 1D. Parte 1



Figura 4.31 La clave de una buena implementación es el orden y coherencia del proceso.

Asimismo, se han identificado y definido explícitamente tres parámetros principales involucrados en el entrenamiento y evaluación del modelo, los cuales serán modificados sistemáticamente en los experimentos numéricos: el número de puntos de colocación (`Nf`), el número de neuronas en cada capa oculta (`Ne`) y el número de puntos utilizados para la evaluación del modelo (`Np`).

Las librerías básicas empleadas son `numpy`, para cómputo numérico; `matplotlib.pyplot`, para visualización y generación de gráficas; y `torch` junto con `torch.nn`, para la implementación de la arquitectura neuronal y los procesos de optimización.

Vista previa 2026

Al implementar las PINNs con PyTorch, conveniente aprovechar la librería nativa, ya que simplifica la construcción y entrenamiento de los modelos. Entre sus principales ventajas destacan:

- Automatización del cálculo de derivadas mediante diferenciación automática.
- Integración eficiente de operaciones tensoriales y algebra lineal.
- Optimización numérica mediante algoritmos de entrenamiento ya implementados.
- Compatibilidad con aceleración en GPU para reducir tiempos de cómputo.

Figura 4.32 Pytorch es un entorno que ya tiene implementado las redes neuronales.

Notas

Nf = 30
Ne = 10
Np = 100
2 capas ocultas

Figura 4.33 Parámetros de la red neuronal.

```python
# ========================================================
# 3.A Termino fuente de la ED
def rhs(x):
  return torch.exp(x)*(torch.cos(2*torch.pi*x)
                   - 4*torch.pi*torch.sin(2*torch.pi*x)
                   - (4*torch.pi**2)*torch.cos(2*torch.pi*x))

# ========================================================
# 3.B PDE con derivadas automaticas en PyTorch
def pde(x, model):
    x      = x.clone().detach().requires_grad_(True)
    u_pred = model(x)
    kappa = 4 * torch.pi**2
    u_x = torch.autograd.grad(
        u_pred, x,
        grad_outputs=torch.ones_like(u_pred),
        create_graph=True
    )[0]  # Primera derivada
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        create_graph=True
    )[0]  # Segunda derivada
    return u_xx + kappa * u_pred - rhs(x)

# ========================================================
# 3.C Funcion de perdida total (dominio + frontera)
def loss_fn(model, x, x_bc, y_bc):
  res      = pde(x, model)  # Evaluar puntos de colocacion
  loss_pde = torch.mean(res ** 2)
  u_bc_pred = model(x_bc) # Evaluar puntos de frontera
  loss_bc   = torch.mean((u_bc - u_bc_pred) ** 2)
  return loss_pde + loss_bc

# ========================================================
# 3.D Optimizador y entrenamiento
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
for epoch in range(8000):
  optimizer.zero_grad()
  loss = loss_fn(model, x_tensor, x_bc, y_bc)
  loss.backward()
  optimizer.step()
  if epoch % 500 == 0 or epoch == 7999:
    print(f"Epoch {epoch}: loss = {loss.item():.6f}")

# ========================================================
# 4. Datos de prueba
Np  = 100
x_p = torch.tensor(np.linspace(xI, xF, Np),dtype=torch.float32)
u_p_pred = model(x_p.view(-1,1)).detach().numpy()

# ========================================================
# 5. Solucion analitica y grafica de resultados
def u_exact(x):
  return torch.exp(x)*torch.cos(2*torch.pi*x)
u_p = u_exact(x_p).detach().numpy()

# ========================================================
# 6. Grafica de resultados
plt.plot(x_p, u_p, label="u(x)")
plt.plot(x_p, u_p_pred, '--', label="Red neuronal")
plt.xlabel('x'); plt.ylabel('u')
plt.legend()
plt.show()
```

Código 4.10 Implementación en PyTorch para entrenar una red neuronal profunda que aproxima la solución de una ED en 1D. Parte 2.

Vista previa 2026



Figura 4.34 `PyTorch` para usar diferenciación automática construye el grafo del modelo. Eso es lo que representa `create_graph`.

Además de las librerías de `torch`, optimizadas para su integración en *Python*, existen alternativas como `TensorFlow`, que permiten desarrollar rutinas equivalentes bajo una lógica similar, pero con una sintaxis y herramientas propias. En los ejemplos presentados se empleará `torch`, salvo indicación contraria.

En las siguientes secciones se describen e identifican claramente cada una de las etapas que conforman las PINNs, estableciendo su correspondencia con los distintos componentes del programa y facilitando la comprensión de su implementación computacional.

4.5.1 Recopilación de datos

El programa inicia con la recopilación de los datos necesarios para el entrenamiento del modelo; este proceso se muestra en el Código 4.11. En primer lugar, se definen las condiciones de frontera, donde `xI` y `xF` representan los extremos del intervalo $[0, 1]$, mientras que `uI` y `uF` corresponden a los valores de la solución en dichos puntos. Posteriormente, estos datos se almacenan en formato tensorial: `x_bc` contiene los puntos de frontera y `u_bc` los valores asociados de la función. Para los puntos de colocación, se genera un muestreo equidistante de `N = 30` puntos en el intervalo $[0, 1]$, denotados por `x_train`, los cuales también se convierten a formato tensorial en `x_tensor`.

```
xI, xF = 0, 1
uI, uF = 1, np.exp(1)
x_bc = torch.tensor([[xI], [xF]], dtype=torch.float32)
u_bc = torch.tensor([[uI], [uF]], dtype=torch.float32)
Nf = 30
x_train  = np.linspace(xI, xF, Nf).reshape(-1, 1)
x_tensor = torch.tensor(x_train, dtype=torch.float32)
```

Código 4.11 Implementación de paso de recopilación de datos.

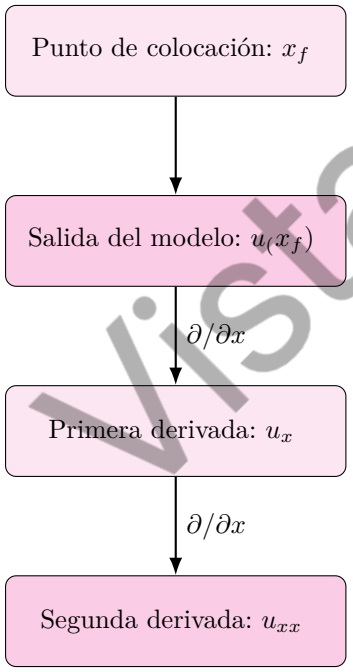


Figura 4.35 Derivadas sucesivas mediante diferenciación automática en `PyTorch`.

4.5.2 Construcción del modelo

Se define la arquitectura de una red neuronal multicapa mediante la clase `SimpleMLP`, mostrada en el Código 4.12. La red consta de dos capas ocultas con `Ne = 10` neuronas cada una. Primero, se inicializan las variables `self.hidden1` y `self.hidden2`, que representan, respectivamente, una conexión de entrada hacia 10 neuronas y una conexión entre dos capas de 10 neuronas. Ambas capas se construyen mediante el método `nn.Linear`, el cual implementa una transformación lineal densa entre capas.

Posteriormente, la función `forward` define el proceso de propagación hacia adelante de la red a través de sus distintas capas. En esta etapa, se establece la función de activación aplicada después de cada transformación lineal, utilizando la instrucción `torch.tanh(self.hidden(x))`. Finalmente, fuera de la clase `SimpleMLP`, se instancia el modelo y se almacena en la variable `model`.

Vista previa 2026



Figura 4.36 La estructura del código facilita su comprensión y modificación, permitiendo realizar diversos experimentos numéricos mediante ajustes sencillos en los parámetros y componentes del modelo.

```
Ne = 10
class SimpleMLP(nn.Module):
  def __init__(self):
    super().__init__()
    self.hidden1 = nn.Linear(1, Ne)
    self.hidden2 = nn.Linear(Ne, Ne)
    self.output = nn.Linear(Ne, 1)
  def forward(self, x):
    x = torch.tanh(self.hidden1(x))
    x = torch.tanh(self.hidden2(x))
    return self.output(x)
model = SimpleMLP()
```

Código 4.12 Implementación del paso construcción del modelo

4.5.3 Función de pérdida

Una vez definida la arquitectura de la red neuronal, el siguiente paso consiste en establecer el criterio de entrenamiento del modelo. Para ello, se define una función de pérdida que cuantifica el error asociado al cumplimiento de las condiciones de frontera y al residual de la ecuación diferencial.

Para construir esta función, primero se calculan las derivadas involucradas en la ecuación diferencial. La primera derivada se obtiene a partir de la variable independiente `x` y de la salida del modelo, almacenada en `u_pred`, guardándose el resultado en `u_x`. De manera análoga, la segunda derivada se calcula usando `x` y `u_x`, almacenándose en `u_xx`. Con estos valores, se determina el residual como la diferencia entre la segunda derivada y el término fuente evaluados en los mismos puntos.

```
def pde(x, model):
    x      = x.clone().detach().requires_grad_(True)
    u_pred = model(x)
    kappa = 4 * torch.pi**2
    u_x = torch.autograd.grad(
        u_pred, x,
        grad_outputs=torch.ones_like(u_pred),
        create_graph=True
    )[0]  # Primera derivada
    u_xx = torch.autograd.grad(
        u_x, x,
        grad_outputs=torch.ones_like(u_x),
        create_graph=True
    )[0]  # Segunda derivada
    return u_xx + kappa * u_pred - rhs(x)
```

Código 4.13 Implementación del paso de definir la función de perdida.

Posteriormente, se define la función de pérdida `loss_fn` como la suma del error cuadrático medio asociado al residual en los puntos de colocación y al cumplimiento de las condiciones de frontera, de acuerdo con (4.63). Para ello, se evalúa la función `pde` sobre los puntos de colocación y el modelo sobre los puntos de frontera. Finalmente, ambos términos se suman para obtener el valor total de la pérdida.

Notas

Épocas: 8000
Tasa de aprendizaje: `lr = 0.01`

Figura 4.37 Hiperparámetros del entrenamiento.

Vista previa 2026

```
Epoch    0: loss = 2566.1
Epoch  500: loss = 0.6143
Epoch 1000: loss = 0.0222
Epoch 1500: loss = 0.0125
Epoch 2000: loss = 0.0233
Epoch 2500: loss = 0.0027
Epoch 3000: loss = 0.3361
Epoch 3500: loss = 0.0212
Epoch 4000: loss = 0.1151
Epoch 4500: loss = 0.6686
Epoch 5000: loss = 0.0062
Epoch 5500: loss = 0.0621
Epoch 6000: loss = 0.0494
Epoch 6500: loss = 0.0024
Epoch 7000: loss = 0.1055
Epoch 7500: loss = 0.0213
Epoch 7999: loss = 0.0264
```

Figura 4.38 Ejemplo del impresión de la evolución de la función de pérdida del código.

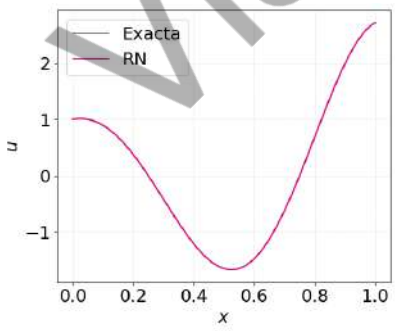


Figura 4.39 Grafica que resultante e del código.

```python
def loss_fn(model, x, x_bc, y_bc):
  res      = pde(x, model)  # Evaluar puntos de colocacion
  loss_pde = torch.mean(res ** 2)
  u_bc_pred = model(x_bc) # Evaluar puntos de frontera
  loss_bc   = torch.mean((u_bc - u_bc_pred) ** 2)
  return loss_pde + loss_bc
```

Código 4.14 Implementación del paso de definir la función de perdida.

4.5.4 Optimizador y entrenamiento

Para el entrenamiento el modelo utiliza el optimizador `Adam` con una tasa de aprendizaje de `lr=0.01`. Para el entrenamiento, se implementa un ciclo `for` que ejecuta de manera iterativa el proceso de actualización de parámetros. Se consideran 8000 épocas. En cada iteración, primero se reinician los gradientes acumulados mediante `optimizer.zero_grad()`. Posteriormente, se calcula el valor de la pérdida utilizando la función `loss_fn`, se obtienen los gradientes correspondientes con el método `backward()`, y finalmente se actualizan los parámetros de la red mediante `optimizer.step()`. Adicionalmente, cada 500 épocas se imprime el valor actual de la pérdida.

```python
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
for epoch in range(8000):
  optimizer.zero_grad()
  loss = loss_fn(model, x_tensor, x_bc, y_bc)
  loss.backward()
  optimizer.step()
  if epoch % 500 == 0 or epoch == 7999:
    print(f"Epoch {epoch}: loss = {loss.item():.6f}")
```

Código 4.15 Implementación del paso de entrenamiento de la red.

4.5.5 Evaluación de los datos de prueba

Una vez finalizado el entrenamiento, se procede a evaluar la PINN. En el código 4.16 se genera un conjunto de datos de prueba compuesto por `Np = 100` puntos equidistantes en el intervalo $[0, 1]$. Posteriormente, el modelo se evalúa sobre estos puntos para obtener la aproximación numérica, almacenada en la variable `u_p_pred`. Con el fin de verificar la precisión de la solución aproximada, se define la función `u_exact`, correspondiente a la solución analítica del problema, y se calculan sus valores sobre el mismo conjunto de prueba, almacenándose en `u_p`. Adicionalmente, se grafican en una misma figura los conjuntos de datos `u_p` y `u_p_pred`, lo que permite comparar la aproximación obtenida por la red con la solución exacta.

```python
Np  = 100
x_p = torch.tensor(np.linspace(xI, xF, Np),dtype=torch.float32)
u_p_pred = model(x_p.view(-1,1)).detach().numpy()
```

Código 4.16 Implementación del paso de evaluación del modelo.

Vista previa 2026

"Nuestro mensaje aquí, [...], es que los métodos clásicos, [...], pueden coexistir en armonía con las redes neuronales profundas y ofrecer una intuición invaluable en la construcción de algoritmos predictivos estructurados"(Rassi et al, 2019, p.20)

Figura 4.40 Nota sobre los algoritmos clásicos.

4.6 RESULTADOS NUMÉRICOS USANDO LAS PINNS

En esta sección se presentan y analizan los resultados numéricos obtenidos para la ecuación (4.47). A menos que se indique lo contrario, se considera el problema definido por las ecuaciones (4.52)-(4.54), con $\kappa = 0$. El análisis se organiza en las siguientes etapas:

- Exploración inicial del método. Se examina el comportamiento de las PINNs mediante la visualización de las soluciones aproximadas, con el propósito de evaluar cualitativamente su capacidad para reproducir la solución exacta.
- Comparación con métodos numéricos clásicos. Se contrastan los resultados obtenidos mediante PINNs con los proporcionados por el método de diferencias finitas, identificando las principales ventajas, limitaciones y diferencias entre ambos enfoques.
- Análisis de convergencia del error. Se estudia la precisión de las aproximaciones numéricas y su comportamiento al modificar los conjuntos de puntos utilizados durante el entrenamiento, con el fin de evaluar su influencia en la calidad de las soluciones.
- Estudio de una arquitectura neuronal sencilla. Finalmente, se analiza de manera preliminar el desempeño de una red neuronal con una sola capa oculta, examinando su capacidad para aproximar la solución de la ecuación diferencial y las limitaciones asociadas con esta arquitectura.

En conjunto, estos experimentos permitirán comprender mejor el comportamiento numérico de las PINNs, así como identificar algunos de los factores que influyen en su precisión y desempeño al resolver ecuaciones diferenciales.

4.6.1 Exploración inicial del método

Como punto de partida, se considera una arquitectura compuesta por dos capas ocultas con 10 neuronas en cada una. Con esta configuración inicial, se realiza una exploración elemental de la implementación presentada en la Sección 4.5. Para la primera simulación, el entrenamiento se lleva a cabo utilizando 30 puntos de colocación.

Al presentar resultados, es fundamental destacar su relevancia. Una adecuada discusión de resultados debe responder, para cada figura o tabla, las siguientes preguntas:

- ¿Qué muestra?
- ¿Cómo se interpreta?
- ¿Por qué es importante?

Figura 4.41 Consejos para presentar comparaciones

La Figura 4.42 presenta tres ejecuciones independientes del código. En la parte superior se muestra la solución exacta de la ED y la aproximación obtenida mediante la PINN. Por su parte, en la sección inferior se incluye la gráfica del error absoluto entre la aproximación y la solución exacta, lo cual permite visualizar con mayor claridad la precisión alcanzada por el modelo. Notar que que la primera simulación no logró capturar adecuadamente el comportamiento de la solución del problema, mientras que la segunda y la tercera sí consiguieron aproximarla de manera satisfactoria.

Vista previa 2026

Lo anterior muestra que, aun manteniendo constantes los puntos de entrenamiento, la arquitectura del modelo y los hiperparámetros, es normal que la calidad del ajuste varíe entre ejecuciones debido a la naturaleza aleatoria del entrenamiento. En particular, los pesos y sesgos iniciales son asignados aleatoriamente cada vez que se genera una nueva red neuronal. Ante esta variabilidad, surge la siguiente pregunta: ¿cómo evaluar de manera objetiva el desempeño del método? Para ello, en las siguientes secciones se realiza un análisi más profundo de los resultados.

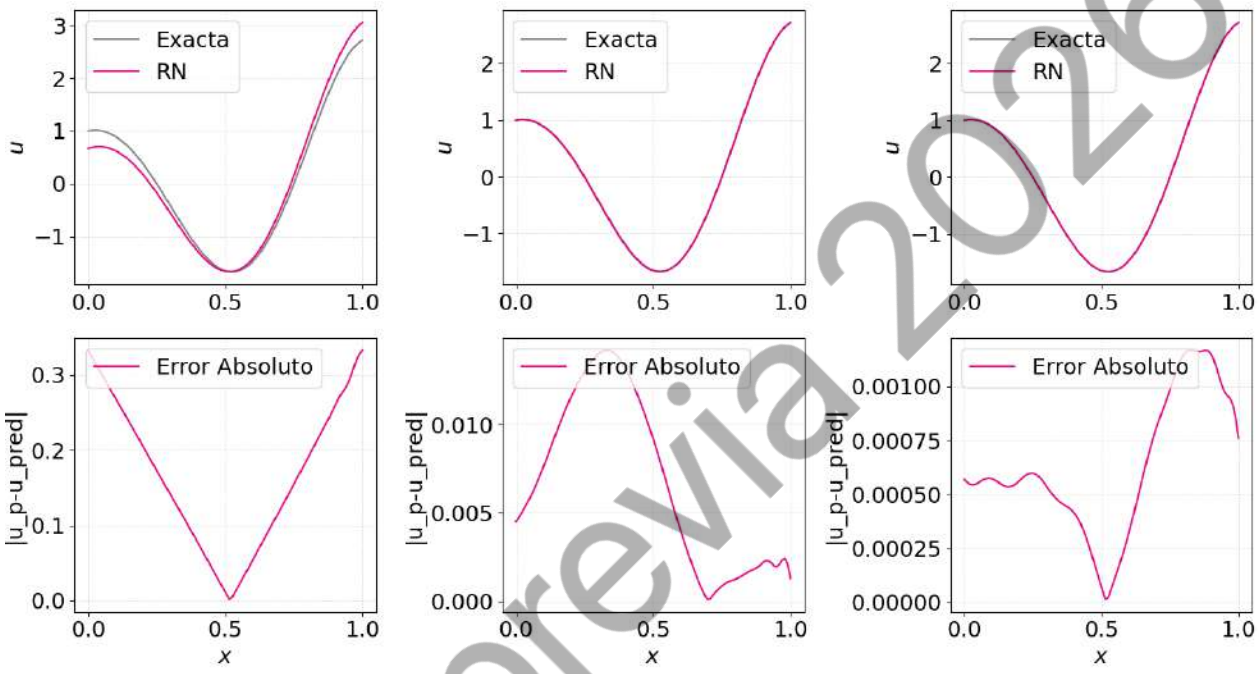


Figura 4.42 Resultados obtenidos mediante PINNs usando dos capas ocultas con 10 neuronas en cada una y 30 puntos de colocación. En la parte superior se presenta la comparación entre la solución exacta y la aproximación de la red neuronal, mientras que en la parte inferior se muestra la gráfica del error absoluto.

Vista previa 2026

4.6.2 Comparación con diferencias finitas

Como se observó en la subsección anterior, las aproximaciones obtenidas mediante PINNs pueden variar entre ejecuciones debido a la naturaleza aleatoria del entrenamiento. Esta característica contrasta con métodos numéricos clásicos, como diferencias finitas, cuya implementación para un mismo problema produce siempre el mismo resultado.

La Figura 4.43 compara los errores absolutos obtenidos mediante PINNs y diferencias finitas evaluados sobre una malla equidistante de 100 divisiones. En este caso, ambos métodos aproximan correctamente la solución exacta y alcanzan errores máximos del orden de 10^{-3}. No obstante, se observan diferencias en los errores absolutos. Mientras que en DF el error tiene valores igual a cero en la frontera, en el caso de las redes neuronales el error presenta un comportamiento oscilante y puede variar entre ejecuciones.

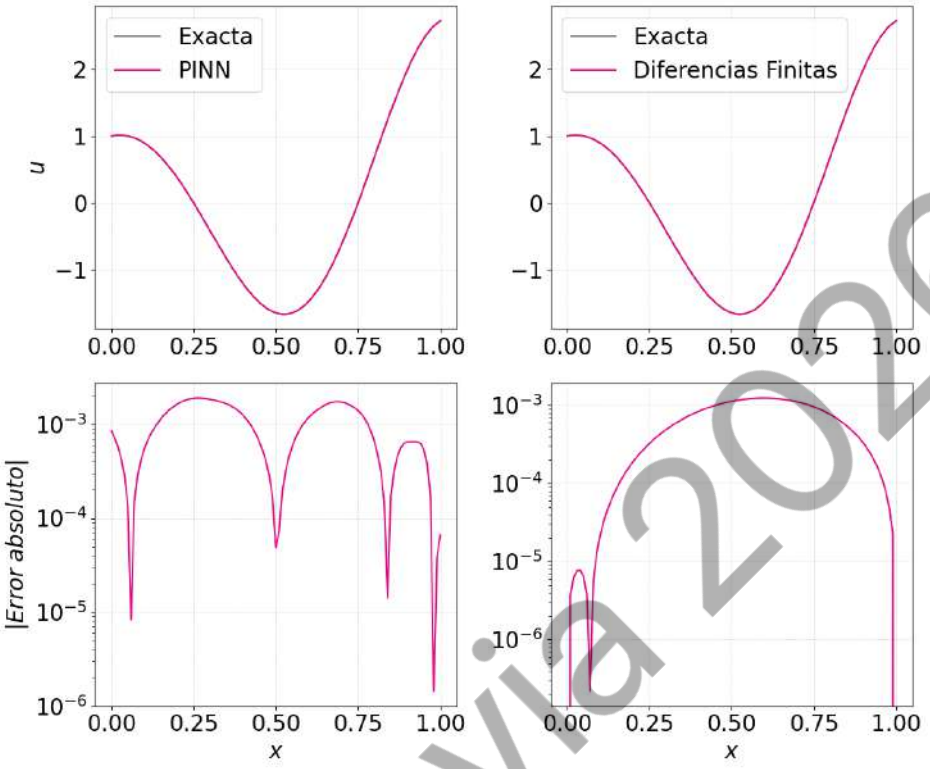


Figura 4.43 Comparación de resultados entre el método PINNs (izquierda) y diferencias finitas (derecha).



Figura 4.44 Una aproximación visual adecuada no siempre refleja la precisión real del método.

Por otra parte, para establecer una comparación bajo condiciones equivalentes, en ambos casos se emplearon puntos ordenados. Sin embargo, esta restricción no es necesaria para las PINNs. Las redes neuronales construyen un modelo continuo a partir de un conjunto de datos, por lo que los puntos de entrenamiento pueden seleccionarse de manera arbitraria. Esto constituye una ventaja importante, ya que permite utilizar datos aleatorios sin afectar el proceso de aproximación.

Otra característica relevante del método de DF es la posibilidad de controlar el error mediante el tamaño de paso h. En particular, en la Sección 4.3 se verificó que el método es de segundo orden de convergencia. En contraste, para redes neuronales no existe un control análogo del error asociado directamente a parámetros discretos, debido a la naturaleza aleatoria del entrenamiento.

En el siguiente experimento se realizan 100 entrenamientos variando únicamente la cantidad de puntos de colocación, considerando valores de $N_f = 10, 20, 40, 80, 160$ con puntos equidistantes, de manera análoga a diferencias finitas. Debido a la naturaleza aleatoria del entrenamiento, en lugar de estimar un orden numérico de convergencia, en la Tabla 4.3 se reportan únicamente los errores mínimos obtenidos en norma L_2 y L_∞, evaluados sobre el conjunto de datos de entrenamiento.

Vista previa 2026



Figura 4.45 Dependiendo del problema se elige la métodología.

Tabla 4.3 Errores absolutos observados del método de diferencias finitas y PINNs bajo diferente número de puntos de colocación.

N	DF: norma L_2	PINN: L_2 Mínimo	DF: norma L_∞	PINN: L_∞ Mínimo
10	7.67×10^{-2}	3.32×10^{-7}	1.26×10^{-1}	1.63×10^{-3}
20	1.89×10^{-2}	5.56×10^{-8}	3.10×10^{-2}	4.12×10^{-4}
40	4.71×10^{-3}	1.53×10^{-8}	7.73×10^{-3}	2.57×10^{-4}
80	1.17×10^{-3}	1.23×10^{-8}	1.93×10^{-3}	2.34×10^{-4}
160	2.94×10^{-4}	2.31×10^{-8}	4.82×10^{-4}	3.44×10^{-4}

Los resultados muestran que una buena aproximación de la PINN puede alcanzarse incluso utilizando conjuntos reducidos de puntos de entrenamiento. Sin embargo, en aquellos casos donde la red neuronal no logra aproximar adecuadamente la solución del problema, los errores máximos alcanzan valores de 19.329 y 6.932 en las normas L_2 y L_∞, respectivamente. Estos resultados ponen de manifiesto la considerable variabilidad entre distintas ejecuciones, incluso cuando se utiliza una misma configuración. Asimismo, se aprecia que incrementar el número de puntos de colocación no garantiza una mejora en la calidad de la aproximación.

Finalmente, es importante señalar que esta metodología basada en redes neuronales no debe entenderse como un reemplazo total de los métodos clásicos, sino como una herramienta complementaria dentro del conjunto de técnicas disponibles para aproximar soluciones de ecuaciones diferenciales.

Cuadro comparativo endtre DFs y PINNs

En este cuadro se encuentran las ventajas y desventajas discutidas de cada metodología.

Ventajas de PINNs

- No requiere mallado estructurado.
- Produce una solución continua.
- Implementación casi universal.

Desventajas de PINNs

- Entrenamiento costoso.
- Sensible a hiperpárametros y puntos de colocación.
- No existe una teoría completa de estabilidad y error comparable con métodos clásicos.

Ventajas de Diferencias Finitas

- Análisis teórico solido.
- Implementación computacionamente eficiente.
- Control del error explícito.

Desventajas de Diferencias Finitas

- Solución solo en la malla.
- Mejor precisión necesita refinamiento de malla.
- Menos flexible al integrar datos experimentales.

Vista previa 2026



Figura 4.46 El entrenar es un juego, donde hay altos y bajos.

4.6.3 Análisis de la convergencia del error

Una vez analizados los resultados obtenidos directamente de la implementación y realizada la comparación con diferencias finitas, resulta natural profundizar en el comportamiento interno del entrenamiento. En particular, para las redes neuronales informadas por la física, la evolución de la función de pérdida permite estudiar cómo el modelo reduce progresivamente el error durante el proceso de optimización. Recordar que el error total se define como la suma de sus componentes, tal como se muestra en la ecuación (4.63).

Con este propósito, en la Figura 4.47 se presenta un experimento donde se analiza la evolución de la función de pérdida. En el lado izquierdo se muestra la historia del error en escala logarítmica, lo que permite visualizar con mayor claridad su decrecimiento y el orden de magnitud alcanzado. En particular, se representan los tres términos considerados durante el entrenamiento: en rosa el error total, en gris claro el error asociado a los datos de entrenamiento y en gris oscuro el error correspondiente a los puntos de colocación. Por su parte, en el lado derecho se presenta el error absoluto final obtenido por el modelo.

En estas gráficas se observan cambios de tendencia en la evolución de los errores los cuales son asociados a la contribución relativa de cada término dentro del error total. Para este experimento, el error asociado a los puntos de colocación es el que más contribuye al error final, mientras que el correspondiente a las condiciones de frontera resulta aproximadamente cuatro órdenes de magnitud menor. Asimismo, en los resultados se aprecia que el error de colocación presenta mayor dificultad para ser reducido durante el entrenamiento.

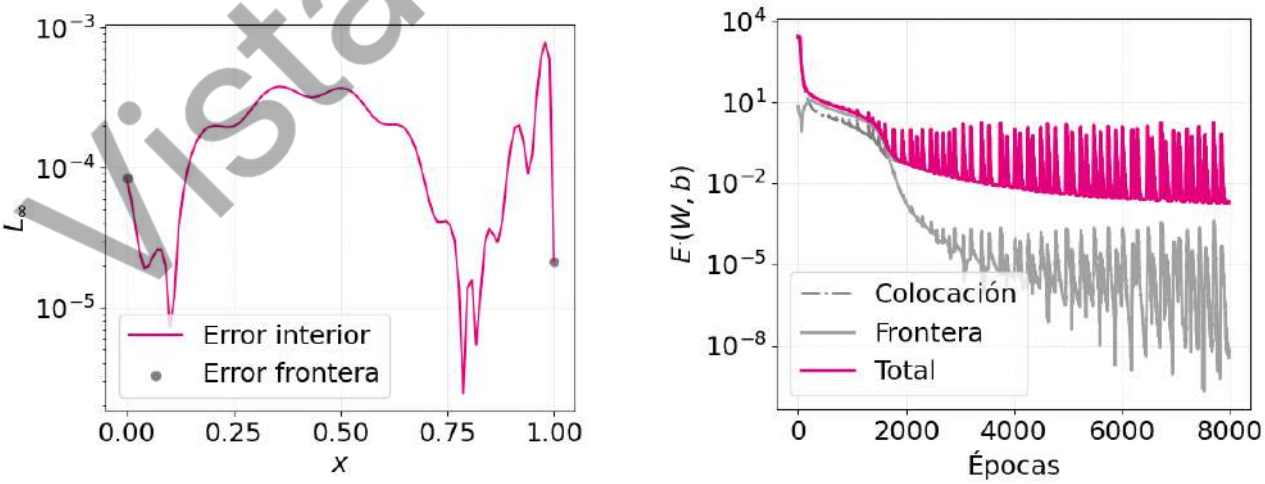


Figura 4.47 Error de entrenamiento: error absoluto final sobre los puntos de prueba (derecha) y evolución del error cuadrático medio a lo largo de las épocas de entrenamiento.

Vista previa 2026

```
==========================
    ERRORES FINALES
==========================
Error total     : 1.97e-03
Error colocación: 1.97e-03
Error frontera  : 7.32e-09
==========================
```

Figura 4.48 Errores de ejecución sin reducción de tasa de aprendizaje.

```
==========================
    ERRORES FINALES
==========================
Error total     : 3.36e-03
Error colocación: 3.36e-03
Error frontera  : 1.56e-09
==========================
```

Figura 4.49 Errores de ejecución con reducción de tasa de aprendizaje.

A partir de las 1500 épocas se observa un comportamiento oscilante en el error, fenómeno inherente a los procesos iterativos de optimización. Para mitigar este efecto, se puede usar una reducción de la tasa de aprendizaje (*learning rate decay*). En este caso, se implementa una estrategia de reducción por pasos, donde cada 1000 épocas la tasa de aprendizaje se multiplica por 0.9. La implementación correspondiente se muestra en el Código 4.17 dentro de la sección del optimizador, mientras que el proceso de entrenamiento modificado se presenta en el Código 4.18.

```
# ============================================================
# Optimizador
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# ============================================================
# Esquema de reduccion de tasa por paso
scheduler = torch.optim.lr_scheduler.StepLR(
  optimizer,          # Optimizador
  step_size=1000,     # Epocas de paso
  gamma=0.9           # Tasa de reduccion
  )
```

Código 4.17 Implemenatición de esquema redución de tasa por paso.

```
optimizer.step()
scheduler.step() # <--------------- Nuevo
```

Código 4.18 Implemenatición de esquema redución de tasa por paso.

Con esta modificación al código original, se obtiene el error de entrenamiento mostrado en la Figura 4.50, sin modificar los demás hiperparámetros. Los resultados muestran una reducción considerable de las oscilaciones y errores finales de magnitud similar a los obtenidos previamente.

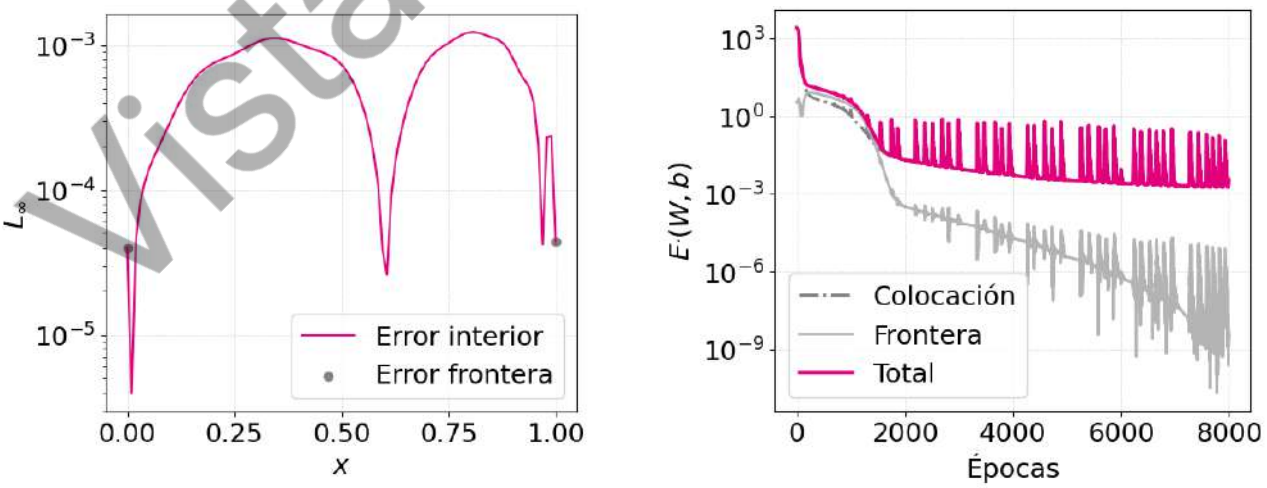


Figura 4.50 Error de entrenamiento con reducción en la tasa de aprendizaje: error absoluto final sobre los puntos de prueba (derecha) y el error cuadrático medio sobre cada época.

Vista previa 2026



Figura 4.51 En experimentos con un gran número de simulaciones, es útil recurrir a representaciones alternativas para analizar los datos de manera global. Por ejemplo, las gráficas de barras permiten comparar el desempeño de distintas configuraciones de la arquitectura o hiperparámetros.

4.6.4 Análisis con otras arquitecturas

En las secciones anteriores se analizó el desempeño de la arquitectura propuesta para resolver la ecuación (4.47). Sin embargo, dado el carácter estocástico del entrenamiento en redes neuronales, resulta conveniente comparar distintas configuraciones para evaluar la estabilidad y consistencia de los resultados obtenidos. En esta sección se estudia el comportamiento de otras arquitecturas aplicadas al mismo problema.

Al comparar arquitecturas, se consideran tres criterios principales: un alto porcentaje de éxito, una cantidad reducida de épocas de entrenamiento y una estructura sencilla. El porcentaje de éxito permite medir la capacidad del modelo para aproximar la solución de manera robusta frente a la variabilidad inherente al método, mientras que el número de épocas y la complejidad de la arquitectura de la red están asociados al costo computacional del entrenamiento.

En la Figura 4.52 se muestran los porcentajes de éxito obtenidos en el siguiente experimento. Se evaluaron redes con una, dos y tres capas ocultas, considerando $N_e = 5$, 10 y 15 neuronas por capa. Para todas las configuraciones se fijaron 30 puntos de colocación, una tasa de aprendizaje de $\eta = 0.01$ y 8000 épocas de entrenamiento. Se considera una ejecución exitosa cuando el error máximo absoluto entre la aproximación y la solución exacta es menor que 10^{-2}. Para cada arquitectura se realizaron 100 ejecuciones independientes.

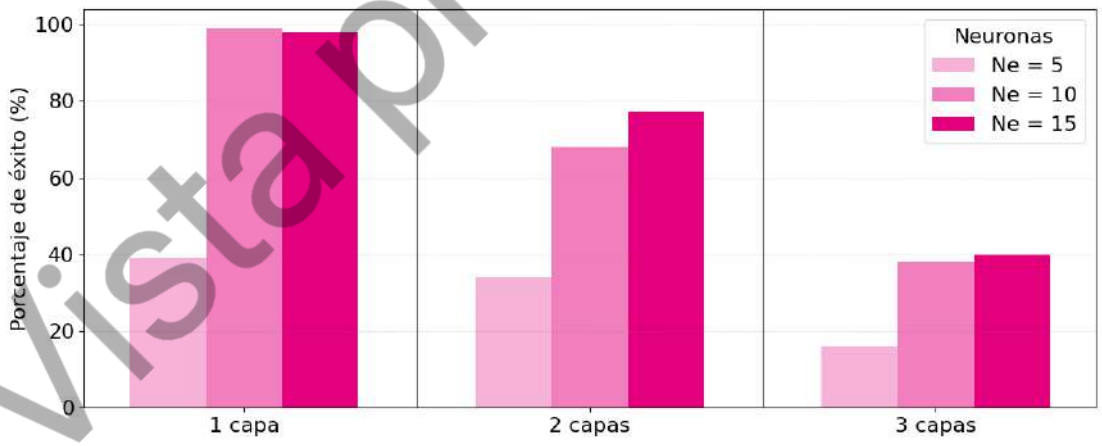


Figura 4.52 Gráficas de barras de los porcentajes de éxito de cada arquitectura.

Los mejores resultados se obtuvieron con arquitecturas de una sola capa oculta y $N_e = 10$ y $N_e = 15$ neuronas, alcanzando porcentajes de éxito del 99 % y 98 %, respectivamente. Dado que ambas configuraciones presentan desempeños similares, se optó por la arquitectura de una sola capa oculta con 10 neuronas, debido a su menor complejidad y a que ofrece un rendimiento ligeramente superior.

Vista previa 2026

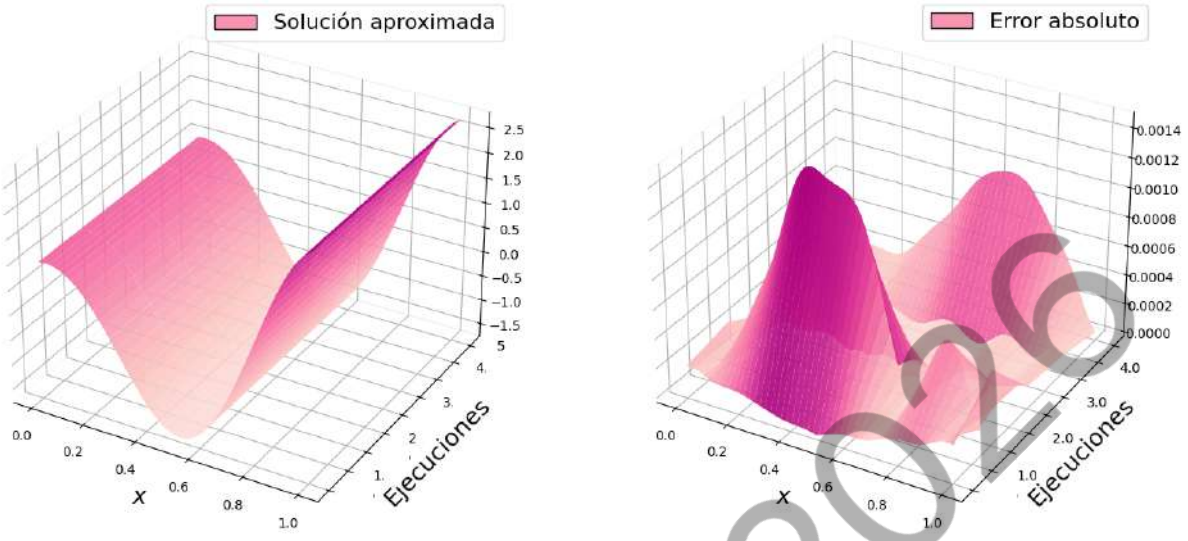


Figura 4.53 Gráfica de cinco ejecuciones del modelo con una capa de 10 neuronas y los correspondientes errores absolutos de las ejecuciones.

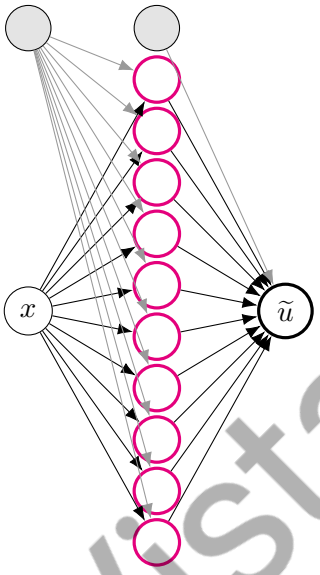


Figura 4.54 La red de una sola capa oculta con 10 neuronas muestra resultados con mayor grado de éxito.

En la Figura 4.53 se presentan los resultados obtenidos en cinco ejecuciones independientes utilizando una arquitectura de una sola capa oculta con 10 neuronas. En el panel izquierdo se muestran las soluciones aproximadas correspondientes a cada ejecución, las cuales describen curvas cercanas a la solución exacta. Por su parte, el panel derecho presenta los errores absolutos asociados a cada ejecución, lo que permite apreciar con mayor claridad las diferencias entre las aproximaciones obtenidas. Cabe destacar que, en dos de las ejecuciones, los errores absolutos se mantienen por debajo de 2×10^{-4} en todo el dominio, lo que evidencia la capacidad de esta arquitectura relativamente simple para generar aproximaciones de alta precisión.

4.6.5 Caso de un oscilador armónico

A partir del análisis preliminar realizado en la sección anterior, se observó que, entre las arquitecturas evaluadas, una red de una sola capa presentó el mejor desempeño para el problema considerado. Sin embargo, este resultado no implica que una arquitectura tan sencilla sea siempre suficiente para resolver cualquier ecuación diferencial.

Surge entonces la siguiente pregunta: ¿puede una red de baja complejidad mantener un buen desempeño en problemas con dinámicas distintas? Para explorar esta cuestión, en esta sección se estudia la ecuación de un oscilador armónico simple, para la cual se dispone de una solución analítica conocida. Sobre este problema se realiza una exploración directa de la metodología propuesta.

Vista previa 2026

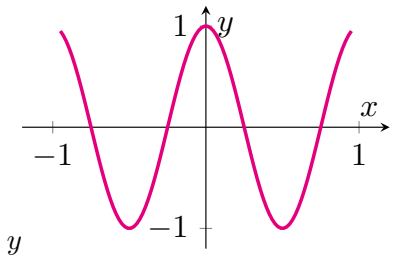


Figura 4.55 Solución analítica del oscilador armónico simple dada por la ecuación diferencial 4.68.

OSCILADOR ARMÓNICO SIMPLE

Consideremos la ecuación diferencial correspondiente al oscilador armónico simple:

$$\frac{d^2u}{dx^2}(x) + 4\pi^2 u(x) = 0, \quad x \in (-1, 1), \tag{4.68}$$

La ecuación se complementa con condiciones de frontera de Dirichlet en ambos extremos del dominio:

$$u(-1) = 1, \qquad u(1) = 1.$$

Bajo estas condiciones, la solución analítica está dada por

$$u(x) = \cos(2\pi x).$$

Nótese que la ecuación (4.68) corresponde a un caso particular de la formulación general presentada en (4.47), al considerar $\kappa = 4\pi^2$ y $g(x) = 0$. Tanto el valor κ como la función g se incorporan directamente en la función de pérdida programada en el Código 4.14.

Las soluciones asociadas al oscilador armónico representan un reto para las redes neuronales informadas por la física debido a su comportamiento oscilatorio y a la presencia de componentes de alta frecuencia. En particular, la solución analítica del problema exhibe oscilaciones rápidas dentro del dominio, lo que exige a la red capturar variaciones locales de manera precisa.

Se sabe que las redes neuronales con funciones de activación suaves, así como aquellas basadas en ReLU, tienden a aprender primero las componentes de baja frecuencia y posteriormente las de mayor frecuencia. Este comportamiento se conoce como sesgo espectral (*spectral bias*) [61].

Los métodos clásicos tienden a amortiguar y desfasar componentes oscilatorias de alta frecuencia; al fenómeno de dispersión y disipación numérica explica cómo privilegian dinámicas suaves, de forma análoga al sesgo espectral que favorece primero bajas frecuencias. [62].

Figura 4.56 Fenómeno equivalente en métodos clásicos.

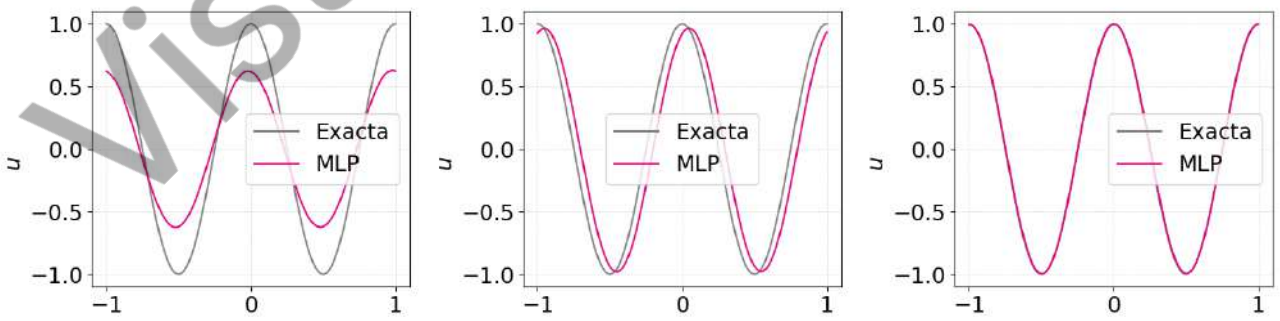


Figura 4.57 Gráficas de aproximación con redes neuronales de una capa (izquierda), dos capas (centro) y 3 capas (derecha) con 30 neuronas en cada capa.

Vista previa 2026



Figura 4.58 A veces se requieren mas capas para una tarea.

En la Figura 4.57 se muestran tres ejemplos de entrenamiento utilizando arquitecturas con una, dos y tres capas ocultas, respectivamente, cada una con 30 neuronas por capa y función de activación tanh. Para todos los casos se emplearon 100 puntos de colocación, 10000 épocas de entrenamiento y una tasa de aprendizaje de 0.001. Se observa que la arquitectura de una sola capa no logra capturar completamente el comportamiento oscilatorio de la solución, presentando dificultades para reproducir su frecuencia de manera adecuada. En contraste, las arquitecturas de dos y tres capas muestran una mejor aproximación, logrando representar con mayor precisión tanto la amplitud como la frecuencia de la solución analítica.

Motivados por este comportamiento, se realiza un análisis más sistemático del efecto de la arquitectura sobre el desempeño del modelo. Para ello, se consideran redes con una, dos y tres capas ocultas, combinadas con 10, 20 y 30 neuronas por capa. En todos los experimentos se utilizan 100 puntos de colocación, 10000 épocas y una tasa de aprendizaje de 0.001. Con el fin de evaluar la estabilidad y robustez del entrenamiento, para cada arquitectura se entrenan 100 modelos independientes. Se considera que un entrenamiento es exitoso si el error máximo absoluto obtenido es menor a 10^{-1}. Adicionalmente, se implementa un esquema de reducción de la tasa de aprendizaje, multiplicándola por un factor de 0.5 cada 2000 épocas, con el objetivo de mitigar las oscilaciones observadas en el error durante el entrenamiento.

Los resultados del experimento se presentan en la Figura 4.59, donde se compara el porcentaje de éxito alcanzado por cada arquitectura. Se observa que las redes de una sola capa muestran un desempeño deficiente al aproximar la solución analítica. En contraste, las arquitecturas multicapa presentan una mayor tasa de éxito, destacando la configuración de 2 capas ocultas con 30 neuronas por capa como la de mejor desempeño dentro del conjunto evaluado.

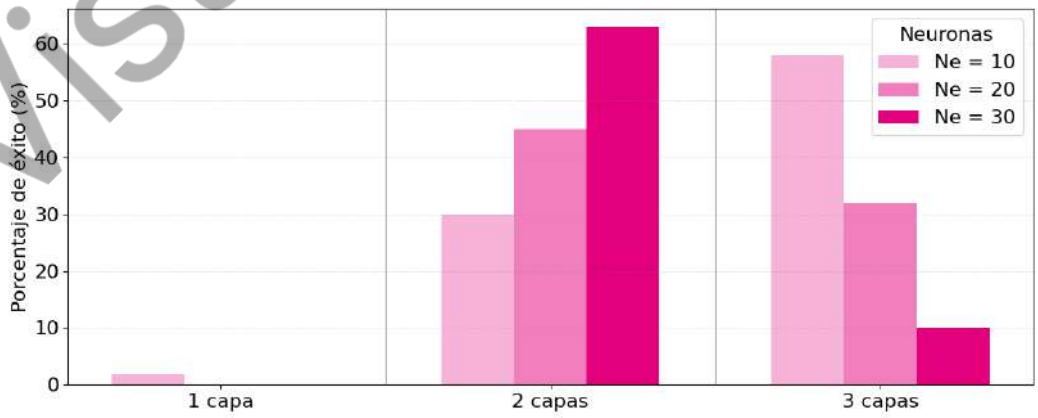


Figura 4.59 Gráficas de barras de los porcentajes de éxito de cada arquitectura.

Vista previa 2026

RESUMEN

Este capítulo introduce la metodología de redes neuronales informadas por la física tomando como punto de partida el caso más sencillo: ecuaciones diferenciales unidimensionales. Primero, se presenta un método numérico clásico con el fin de establecer una referencia para comparar los resultados obtenidos y contrastarlos con las PINNs.

En cuanto al marco teórico, se construye una base conceptual que permite al lector identificar los elementos fundamentales de la metodología, así como algunos de los fenómenos característicos que surgen durante el entrenamiento. Asimismo, se incorpora el ejemplo del oscilador armónico, el cual amplía la perspectiva sobre la variabilidad del desempeño del método ante problemas con comportamiento oscilatorio.

Una de las principales características de las redes neuronales informadas por la física es su alta flexibilidad, ya que permiten incorporar modificaciones en la función de pérdida, estrategias de muestreo, arquitecturas y esquemas de entrenamiento. Esta versatilidad brinda al usuario una gran libertad para adaptar el método a distintos tipos de problemas.

Finalmente, se invita al lector a experimentar con los ejercicios propuestos en esta sección, diseñados para reforzar conceptos y explorar fenómenos que se comprenden mejor mediante la práctica. Como se verá en el siguiente capítulo, la estructura general del método se mantiene para problemas en dimensiones superiores, siendo necesario únicamente ajustar la capa de entrada al número de variables involucradas.

PARÁMETROS CLAVE

Al emplear PINNs para aproximar soluciones de ecuaciones diferenciales, es importante considerar los siguientes parámetros clave:

- **Arquitectura de la red:**
  - Número de capas ocultas.
  - Número de neuronas por capa.
  - Función de activación.
- **Datos de entrenamiento:**
  - Cantidad de datos disponibles.
  - Puntos de colocación.
- **Inicialización de pesos:**
  - Método utilizado.
- **Función de pérdida:**
  - Error cuadrático medio de puntos de colocación.
  - Error cuadrático medio de puntos de frontera.
- **Algoritmo de optimización:**
  - Tipo de optimizador.
  - Número total de épocas.
  - Tasa de aprendizaje.
  - Esquema de reducción de tasa.
- **Evaluación del modelo:**
  - Comparar con la solución analítica.
  - Comparar con un método clásico.
  - Visualizar el error.
  - Usar de datos de prueba.

Vista previa 2026

EJERCICIOS

Realiza los siguientes ejercicios con atención, asegurándote de comprender cada uno de los pasos y de aplicar correctamente los conceptos estudiados. Dedica el tiempo necesario para resolverlos con cuidado y, en caso de surgir dudas, consulta las notas correspondientes para reforzar tu aprendizaje. En el libro de soluciones se pueden encontrar las respuestas correspondientes, así como los códigos utilizados. Sin embargo, se recomienda recurrir a este material únicamente para verificar los resultados obtenidos o en caso de presentar dificultades durante la resolución de los mismos.

Ejercicio 4.1: Análisis del número de puntos de colocación. En nuestro primer ejercicio exploraremos el número de puntos de colocación aleatorios y equidistantes del problema actualmente planteado.

- Modifica el número de puntos de colocación equidistantes y analiza los resultados obtenidos con $N_f = 10$, 20 y 40.
- Repite el experimento utilizando puntos de colocación generados aleatoriamente con `np.random.uniform`.
- Para cada caso, grafica la solución PINN, la solución exacta y el error absoluto. Comenta las diferencias observadas.

Ejercicio 4.2: Análisis de la arquitectura de la red neuronal. Ahora analizaremos la influencia de la arquitectura de la red neuronal del mismo ejemplo proporcionado en esta sección.

- Manteniendo fijo el número de puntos de colocación, entrena redes neuronales con tres capas ocultas usando 5, 10 y 20 neuronas por capa.
- Repite el experimento utilizando cinco capas ocultas con 10 neuronas cada una.
- Para cada configuración, reporta la norma del error L_2 y discute cómo afecta la arquitectura a la precisión y generalización de la red.

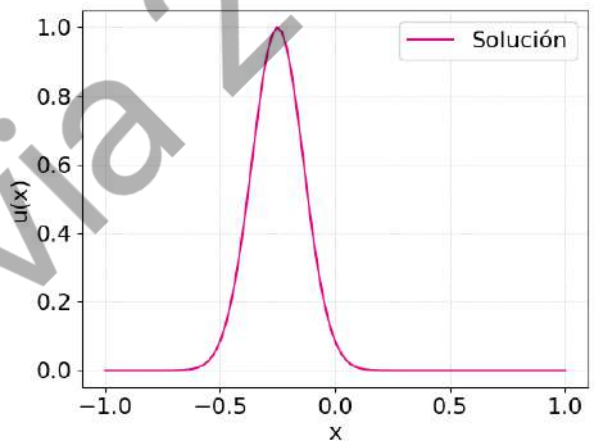
Ejercicio 4.3: Análisis de otras funciones como término fuente de la ED. Consideremos la siguiente función de lado derecho

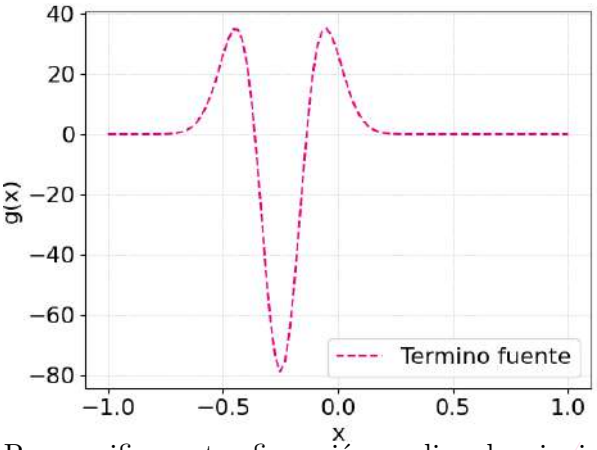
$$g(x) = \left[-8\pi^2\left(1 - 8\pi^2\left(x + 1/4\right)^2\right)\right] e^{-4\pi^2(x+1/4)^2}, \tag{4.1}$$

y solución exacta del problema de Poisson unidimensional:

$$u(x) = e^{-4\pi^2(x+1/4)^2}, \tag{4.2}$$

junto con condiciones de frontera de Dirichlet tomadas de la solución exacta. Ambas funciones son suaves (infinitamente diferenciables). Dada la suavidad de las funciones u y g, se espera que el análisis mediante redes neuronales sea similar al planteado con el ejemplo anterior.





Para verificar esta afirmación, realizar las siguientes actividades:

- Implementar el problema de Poisson unidimensional utilizando la metodología de PINNs, incorporando el nuevo término fuente y las condiciones de frontera correspondientes.

Vista previa 2026

- Proponer una arquitectura de red neuronal y mantenerla fija durante todos los experimentos numéricos.
- Realizar diferentes simulaciones variando únicamente el número de puntos de colocación, distribuidos uniformemente en el intervalo $[-1, 1]$.
- Comparar las soluciones aproximadas obtenidas con la solución exacta, utilizando representaciones gráficas que permitan visualizar las diferencias.
- Calcular y analizar el error absoluto para cada simulación, identificando cómo cambia al incrementar el número de puntos de colocación.
- Reportar los resultados numéricos y discutir si el comportamiento observado es similar al del ejemplo anterior, considerando la precisión de las aproximaciones y la influencia de la discretización.

Ejercicio 4.4: PINNs para la ecuación de Sturm-Liouville. Consideremos la ecuación de Sturm-Liouville para una cuerda vibrante fija en dos extremos dada por:

$$\frac{d^2u}{dx^2}(x) + k^2u(x) = e^x(\cos(kx) - 2k\sin(kx)), \tag{4.3}$$

donde $x \in (0, 2\pi)$ y $u = u(x)$ es la función de posición y k es un número de onda positivo que determina la frecuencia espacial de las oscilaciones. Para un valor alto de k, la solución presenta muchas oscilaciones en el intervalo espacial considerado. La ecuación es complementada con las siguientes condiciones de frontera homogéneas de tipo Dirichlet: $u(0) = 1$ y $u(2\pi) = e^{2\pi}$. La solución de este problema es $u(x) = e^x\cos(kx)$. Analizar este problema con la metodología de redes neuronales multicapa usando $k = 5$.

Ejercicio 4.5: PINNs para un oscilador armónico amortiguado. Analizar con la metodología de redes neuronales profundas la siguiente ecuación diferencial ordinaria de segundo orden que modela el comportamiento de un oscilador armónico amortiguado en estado estacionario:

$$-\frac{d^2u}{dx^2}(x) + 2\frac{du}{dx}(x) + u(x) = \cos(x), \tag{4.4}$$

donde $x \in (0, 1)$ y complementada con las siguientes condiciones de frontera homogéneas de tipo Dirichlet: $u(0) = 0$ y $u(1) = 0$. Aquí $u(x)$ representa el desplazamiento (o posición) del oscilador armónico amortiguado en función de la posición espacial x dentro del intervalo $[0, 1]$. La solución exacta está dada por $u(x) = Ae^{(1+\sqrt{2})x} + Be^{(1-\sqrt{2})x} + \frac{1}{4}(\sin(x) + \cos(x))$, donde A, B satisfacen el sistema

$$\begin{aligned} A + B &= -\tfrac{1}{4} \\ Ae^{1+\sqrt{2}} + Be^{1-\sqrt{2}} &= -\tfrac{1}{4}(\sin(1) + \cos(1)) \end{aligned} \tag{4.5}$$

Ejercicio 4.6: Análisis de ED más complejas. Analizar con la metodología de redes neuronales multicapa la ecuación:

$$\frac{d^2u}{dx^2} - \frac{du}{dx} + 4\pi^2u(x) = -2\pi e^x\sin(2\pi x), \tag{4.6}$$

donde $x \in (0, 1)$ y con condiciones de frontera $u(0) = 1$ y $u(1) = e$. Esta ecuación tiene solución $u(x) = e^x\cos(2\pi x)$.

Nótese que esta ecuación tiene la misma solución que el ejemplo propuesto del capítulo. En una primera instancia verifique si una arquitectura de una capa es suficiente y en caso que no lo sea, busque una configuración que alcance consistentemente un error menor a 0.01.

Ejercicio 4.7: PINNs para ecuaciones diferenciales ordinarias. Para describir la metodología para ecuaciones diferenciales ordinarias de primer orden se realizan pequeñas modificaciones al programa. Implemente el método para resolver la ecuación diferencial

$$u'(x) = \alpha u(x), \quad x \in (0, 1) \tag{4.7}$$

con condición inicial $u(0) = 1$ y $\alpha = 1$. Utilice una red de una capa con 5 neuronas entrenada con 2000 épocas y una tasa de aprendizaje de 0.01. Considere 20 puntos de colocación equidistantes y 100 datos de prueba. Realice lo siguiente y ejecute múltiples veces el algoritmo.

- Determine el error máximo absoluto con respecto al modelo, ¿Cuál es el error L_∞ máximo y mínimo alcanzado?
- Realice el mismo experimento con $\alpha = 2$. ¿Como son los resultados en este caso? Reporte el error Ł$_\infty$ máximo y mínimo.

Vista previa 2026

- Experimente con $\alpha = 3$. ¿Cómo son las aproximaciones en este caso? ¿El método sigue convergiendo adecuadamente?
- Aumente la cantidad de épocas para el caso con $\alpha = 3$. ¿Hay alguna una mejora? Reporte el error $Ł_\infty$ máximo y mínimo.
- Disminuya la tasa de aprendizaje para el mismo caso. Reporte el error $Ł_\infty$ máximo y mínimo.

Ejercicio 4.8: En un laboratorio están trabajando con un cultivo de bacterias. El crecimiento poblacional lo modelan con la ecuación logística

$$u'(t) = ru(t)\left(1 - \frac{u(t)}{K}\right) \tag{4.8}$$

donde u es la población bacteriana en densidad óptica (OD_{600}). Consideran la capacidad de carga fija de 1.5 y una tasa de crecimiento de 0.4. La población inicial fue de 1.0 OD_{600}, aproxime el crecimiento de la población en un periodo de 4 horas.

Aproxime el crecimiento de la población con una red de una capa con 10 neuronas. Considere 50 puntos de colocación y entrene con 3000 épocas y tasa de aprendizaje de 0.001. Evalué el error máximo absoluto de la aproximación con la solución analítica que esta dada por

$$u(t) = \frac{ku(0)}{u(0) + (K - u(0))e^{-rt}} \tag{4.9}$$



Ejercicio 4.9: En el laboratorio esta como pasante Juanito. A él le dejan de encargo realizar el experimento de crecimiento de bacterias y medir la capacidad de carga 50 veces durante el experimento. Janito es algo despistado y tomó la medida cada vez que se acordaba. Además, el aparato que se uso para medir presenta un pequeño error de calibración. Con los datos recabados le piden a Juanito aproximar el crecimiento poblacional, y no sabe que hacer.

Su amigo Jhon le dice que implemente la metodología de PINNs para resolver su problema. Considere una red de una capa con 15 neuronas y que la entrene con 3000 épocas y tasa de aprendizaje de 0.001.

Para simular los datos recabados de Janito considere 50 puntos de colocación uniformemente distribuidos. Además, agregue un ruido gaussiano a los datos de la carga centrado en el valor $K = 1.5$. Analice el error máximo absoluto con respecto a la solución analítica considerando $K = 1.5$. ¿Qué puede concluir de este ejemplo? ¿Por qué la metodología de PINNs ayuda a Juanito?

Nota: Asegúrese que para el entrenamiento se usa los datos con ruido. Modifique la función `pde` como sea más conveniente.

```python
#0. Parametros de la ecuacion
K  = 1.5   # Capacidad de carga

# Muestreo de datos
sample = 50

#1.2 Puntos de colocaccion
t = np.random.uniform(0, 4, sample)
x_tensor = torch.tensor(t.reshape(-1,1),
    dtype=torch.float32)

# Ruido
s = 0.1 # Amplitud del ruido
noise = s*torch.randn(x_tensor.shape)
# Datos de carga con ruido
K_tensor = K*torch.ones_like(x_tensor) +
    noise
```

Código 4.1 Implementación de los valores extraidos de Juanito.

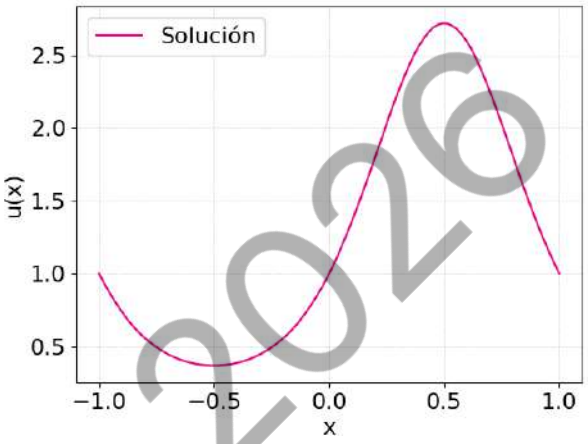
Ejercicio 4.10: Consideremos la ecuación diferencial no lineal de segundo orden

$$u''(x) - \pi cos(\pi x)u'(x) + \pi^2 \sin(\pi x)u(x) = 0 \tag{4.10}$$

Vista previa 2026

la cual con las condiciones de frontera $u(-1) = 1$, $u(1) = 1$ tiene como solución $u(x) = e^{\sin(\pi x)}$. Realiza los siguiente:

- Implementa el algoritmo para PINNs de este problema. Utiliza las funciones de `PyTorch` para las funciones elementales necesarias.
- Entrena redes de 1, 2 y 3 capas con 20 neuronas en cada capa con 4000 épocas y tasa de aprendizaje de 0.001. Considera 300 puntos de colocación y un conjunto de prueba de 500 puntos. ¿Hay diferencias notorias de las aproximaciones?
- Realiza 10 entrenamientos de cada arquitectura y reporta los errores L_2 y L_∞ máximo y mínimo en cada caso. ¿Cual arquitectura es la que mejor se ajusta al problema, considerando cercanía de la solución y tamaño?



Vista previa 2026

CAPÍTULO 5

REDES NEURONALES INFORMADAS POR LA FÍSICA PARA ECUACIONES EN 2D

Ahora nos enfocaremos en la resolución de *Ecuaciones Diferenciales Parciales* (EDP) en dos dimensiones (2D) mediante una red neuronal multicapa, dentro del enfoque de de las *redes neuronales informadas por la física*. Esta metodología permite incorporar, de manera simultánea, tanto la información proveniente de datos como las leyes físicas que gobiernan el fenómeno bajo estudio, lo que la convierte en una herramienta especialmente atractiva, incluso en escenarios donde los datos disponibles son escasos, irregulares o se encuentran distribuidos de forma no uniforme.

A partir de este punto, se presentarán los elementos fundamentales necesarios para formular y resolver problemas en 2D mediante redes neuronales, incluyendo la construcción de la arquitectura, la definición de la función de pérdida a partir de la ecuación diferencial y sus condiciones asociadas, así como el proceso de entrenamiento y ajuste de parámetros. Finalmente, se estudiarán algunos ejemplos representativos que permitirán ilustrar la implementación práctica de esta técnica y consolidar los conceptos desarrollados a lo largo del capítulo.

Vista previa 2026

OBJETIVOS

Al finalizar este capítulo, el lector será capaz de:

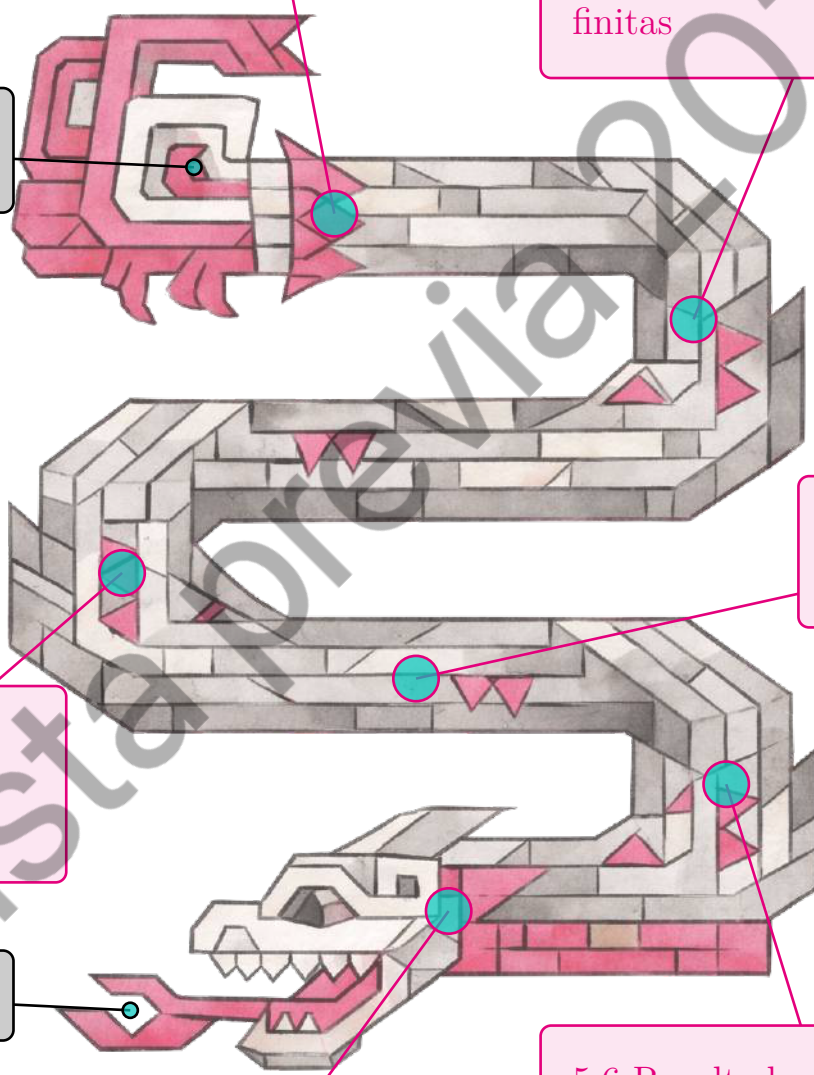
- Comprender la formulación general de problemas elípticos en dos dimensiones y su importancia en la modelación de fenómenos físicos.
- Introducir el enfoque de PINNs para aproximar la solución de EDP incorporando información física directamente en el entrenamiento.
- Describir la construcción de una red neuronal multicapa para aproximar soluciones de EDP en dos variables espaciales.
- Analizar ejemplos representativos de ecuaciones elípticas en 2D con el fin de ilustrar la implementación práctica de la metodología y evaluar su desempeño en la aproximación de soluciones.

CONTENIDO

5.2 Ecuaciones elípticas en 2D

5.3 El método de diferencias finitas

5.1 Introducción



5.5 El programa en Python

5.4 Aproximaciones con redes neuronales

Resumen

5.6 Resultados en dominios rectangulares

5.7 Resultados en dominios irregulares

Vista previa 2026



Figura 5.1 Un pequeño paso para el hombre, un gran paso para la humanidad. Así quedó grabada en la historia la voz de Neil Armstrong el 20 de julio de 1969 durante la misión Apolo 11.



Figura 5.2 La ley de Gauss para la gravedad establece que $\nabla \cdot \mathbf{g} = -4\pi G\rho$, lo que conecta de manera local la geometría del campo con la distribución de masa.

5.1 INTRODUCCIÓN

Un pequeño paso para el hombre, un gran paso para la humanidad. Con estas palabras quedó marcada en la historia la voz de Neil Armstrong el 20 de julio de 1969 durante la misión Apolo 11. Sin embargo, la posibilidad de llevar una nave hasta la Luna no apareció de manera repentina; fue el resultado de siglos de desarrollo científico y matemático. Gran parte de este camino comenzó con las ideas de Isaac Newton, quien formuló las leyes fundamentales de la gravitación y sentó las bases para comprender la interacción entre la Tierra, la Luna y los cuerpos celestes mediante relaciones matemáticas precisas.

A partir de estas ideas surge de manera natural el concepto de campo gravitacional. En lugar de describir únicamente la fuerza entre dos objetos, la teoría permite estudiar cómo la gravedad actúa en cada punto del espacio. Esta descripción conduce a una formulación matemática mucho más profunda basada en campos y potenciales. En el contexto de la gravitación newtoniana, el campo gravitacional satisface una ley de conservación expresada mediante ecuaciones diferenciales. En particular, la ley de Gauss para la gravedad establece que

$$\nabla \cdot \mathbf{g} = -4\pi G\rho,$$

donde $\mathbf{g}$ representa el campo gravitacional, $\nabla\cdot$ denota el operador divergencia, G es la constante universal de gravitación y ρ corresponde a la densidad de masa del sistema. Esta ecuación relaciona localmente la geometría del campo gravitacional con la distribución de masa presente en el espacio.

Aquí aparece una idea fundamental. Cuando el campo gravitacional es irrotacional, puede expresarse como el gradiente negativo de un potencial escalar ϕ, es decir, $\mathbf{g} = -\nabla\phi$. Sustituyendo esta relación en la ley de Gauss se obtiene

$$\nabla^2\phi = 4\pi G\rho, \qquad \text{donde} \qquad \nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2}$$

es el operador Laplaciano. Esta ecuación, conocida como ecuación de Poisson, constituye uno de los ejemplos más importantes y clásicos de las ecuaciones diferenciales parciales elípticas.

Aquí está el punto central: las ecuaciones elípticas no son un capricho matemático elegante, son el lenguaje natural de los estados de equilibrio. Aparecen cada vez que un sistema alcanza una configuración estable: en la distribución del potencial gravitacional, en campos eléctricos, en la difusión estacionaria de calor, entre muchos otros fenómenos. En pocas palabras, describen cómo se organizan los sistemas cuando han dejado de evolucionar en el tiempo.

Vista previa 2026



Figura 5.3 Uno de los fenómenos físicos clásicos que puede modelarse mediante ecuaciones elípticas en 2D es la distribución estacionaria de temperatura en placas metálicas, donde el calor alcanza un equilibrio y deja de variar con respecto al tiempo.

Desde el punto de vista matemático, las EDPs elípticas poseen propiedades notables: regularidad de sus soluciones, fuerte dependencia de las condiciones de frontera y una estructura variacional que permite interpretarlas como problemas de minimización de energía. Estas características no solo garantizan que los modelos sean bien planteados, sino que también facilitan su análisis teórico y aproximación numérica.

Desde el punto de vista físico y computacional, su importancia es aún más contundente. Resolver la ecuación de Poisson equivale, en esencia, a reconstruir un campo completo (gravitacional, eléctrico o térmico) a partir de su fuente. Esto es precisamente lo que permitió modelar trayectorias orbitales, diseñar misiones espaciales y comprender sistemas complejos con un alto grado de precisión.

Así, lo que comenzó como una pregunta sobre *por qué cae una manzana* terminó siendo una teoría capaz de describir el movimiento de planetas, el diseño de trayectorias espaciales y la simulación de fenómenos físicos complejos. En este contexto, las ecuaciones diferenciales parciales elípticas emergen como una pieza fundamental: el puente entre la física, la matemática y la computación.

Este capítulo se sitúa precisamente en ese cruce. Aquí no solo se estudiará la ecuación de Poisson como modelo gravitacional, sino también como representante de una clase mucho más amplia de problemas cuya relevancia atraviesa diversas disciplinas. Porque, al final, entender estas ecuaciones no es solo resolver un problema matemático: *es aprender a describir cómo el universo alcanza el equilibrio.*

Vista previa 2025



Figura 5.4 Una parte fundamental en la resolución de cualquier problema consiste en la obtención del modelo matemático que describe adecuadamente el fenómeno físico bajo estudio.



Figura 5.5 Las placas metálicas pueden presentar diferentes formas geométricas, comenzando con configuraciones simples como dominios rectangulares.

5.2 ECUACIONES ELÍPTICAS EN 2D

Con el propósito de ilustrar la implementación práctica de la metodología basada en redes neuronales y evaluar su desempeño en la aproximación de soluciones, consideraremos una clase de ecuaciones elípticas en dos dimensiones descritas a continuación.

ECUACIÓN DIFERENCIAL ELÍPTICA EN 2D

Consideremos ecuaciones elípticas de segundo orden en dos dimensiones de la forma

$$\nabla \cdot (\beta(\boldsymbol{x})\nabla u(\boldsymbol{x})) + \kappa(\boldsymbol{x})u(\boldsymbol{x}) = g(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Omega \subset \mathbb{R}^2, \quad (5.69)$$

donde $\boldsymbol{x} = (x, y) \in \Omega$, $u = u(x, y)$ es la función desconocida que se desea aproximar, $\beta(x, y)$ representa un coeficiente de difusión o conductividad espacialmente variable, $\kappa(x, y)$ es un término de reacción o absorción, y $g(x, y)$ corresponde al término fuente.

Supondremos que las funciones β, κ y g son conocidas y suficientemente suaves, de manera que el problema se encuentre bien planteado desde el punto de vista analítico. En particular, para garantizar el carácter elíptico del operador diferencial, se asume que $\beta(x, y) > 0$ en todo punto del dominio Ω.

La ecuación (5.69) aparece en una gran variedad de contextos físicos e ingenieriles, como la conducción de calor en estado estacionario, procesos de difusión, electrostática y mecánica de medios continuos, entre otros. En este sentido, la solución buscada $u(x, y)$ describe típicamente una magnitud física en equilibrio, cuya distribución espacial está determinada tanto por las propiedades del medio como por las condiciones impuestas en la frontera del dominio.

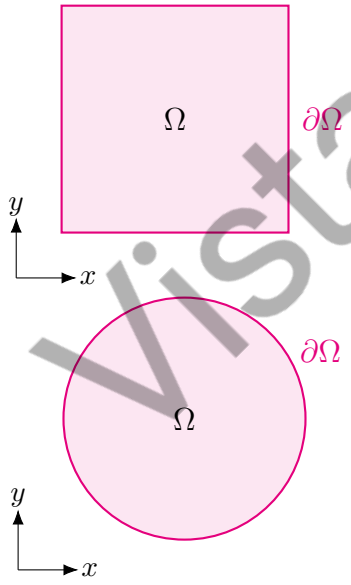


Figura 5.6 Ejemplos de dominios con geometría regular empleados para el problema elíptico en $\mathbb{R}^2$.

5.2.1 Condiciones de frontera

Para que el problema quede completamente determinado, no basta con especificar únicamente la ecuación diferencial en el interior del dominio; también es necesario imponer información adicional sobre lo que ocurre en su frontera. A esta información se le conoce como *condiciones de frontera*, y su papel es restringir el conjunto de soluciones posibles de acuerdo con el comportamiento físico que se desea modelar.

Denotemos por $\partial\Omega$ la frontera del dominio Ω. En aplicaciones físicas, esta frontera puede representar las paredes de una placa, el contorno de un material o la superficie externa de una región donde ocurre un fenómeno de difusión o transferencia. Dependiendo del problema considerado, la frontera $\partial\Omega$ puede dividirse en distintas porciones, sobre cada una de las cuales se prescribe un tipo específico de condición. A continuación se presentan sus expresiones matemáticas más generales.

Vista previa 2026

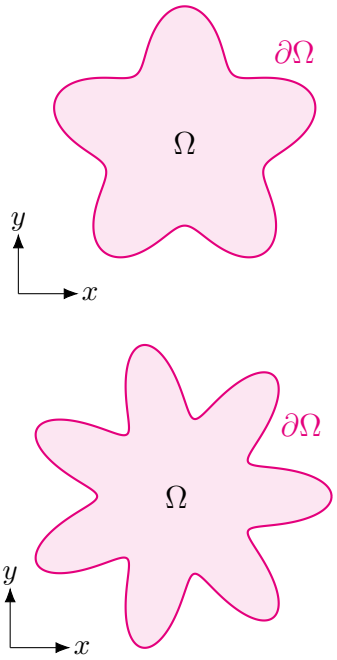


Figura 5.7 Ejemplos de dominios irregulares empleados para el problema elíptico en $\mathbb{R}^2$.

CONDICIONES DE FRONTERA

El problema diferencial parcial (5.69) se considera junto con las condiciones de frontera de tipo Dirichlet, Neumann y Robin dadas por:

$$u(\boldsymbol{x}) = u_D(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Gamma_D, \tag{5.70}$$

$$\beta(\boldsymbol{x})\nabla u(\boldsymbol{x}) \cdot \boldsymbol{n} = q_N(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Gamma_N, \tag{5.71}$$

$$\beta(\boldsymbol{x})\nabla u(\boldsymbol{x}) \cdot \boldsymbol{n} + \alpha(\boldsymbol{x})\, u(\boldsymbol{x}) = h_R(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Gamma_R, \tag{5.72}$$

respectivamente, donde Γ_D, Γ_N y Γ_R son subconjuntos apropiados de $\partial\Omega$, $\boldsymbol{n}$ denota el vector normal unitario exterior a la frontera, y u_D, q_N, h_R y α son funciones conocidas.

- Las condiciones de Dirichlet especifican directamente el valor de la solución sobre la frontera. En términos físicos, esto equivale, por ejemplo, a fijar la temperatura en el borde de una placa o el potencial eléctrico sobre una superficie conductora.
- Las condiciones de Neumann, en cambio, no fijan directamente el valor de la solución, sino su derivada normal a la frontera, es decir, la tasa de cambio de la solución en la dirección perpendicular al borde. Este tipo de condición suele interpretarse como la imposición de un flujo, por ejemplo, el flujo de calor que entra o sale de una superficie.
- Finalmente, las condiciones de Robin combinan información sobre el valor de la solución y su derivada normal. Este tipo de condición aparece con frecuencia en modelos donde existe intercambio con el medio exterior, como en problemas de transferencia de calor por convección donde el comportamiento en la frontera depende tanto de la temperatura como del flujo.

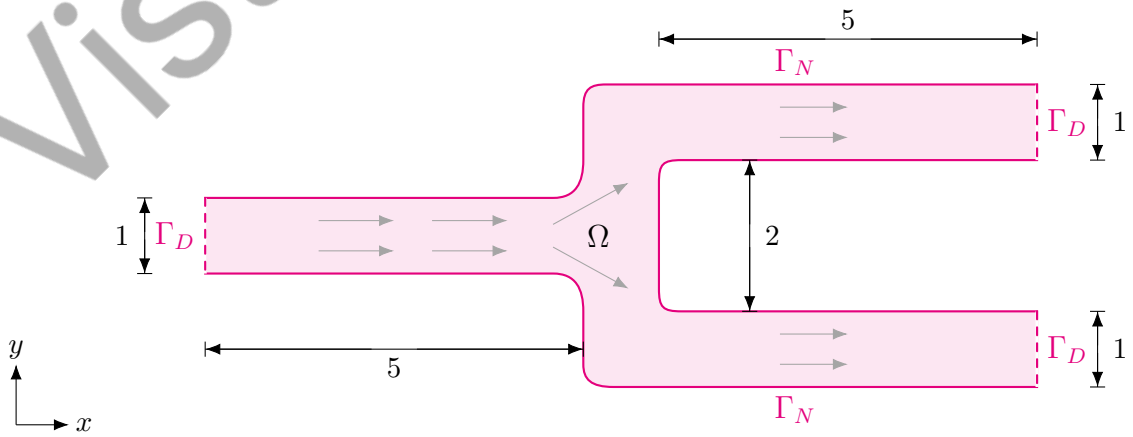


Figura 5.8 Ejemplo de un dominio bidimensional tipo tubería ramificada con condiciones de frontera mixtas.

Vista previa 2026

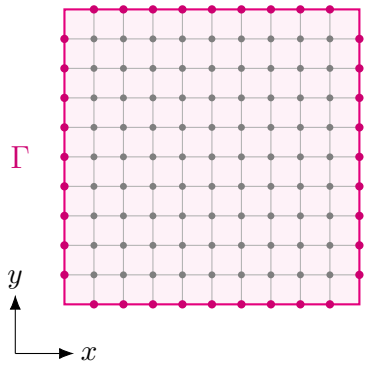


Figura 5.9 Malla uniforme identificando los puntos de la frontera conocidos y puntos interiores donde se aproxima la solución de la EDP.

5.3 EL MÉTODO DE DIFERENCIAS FINITAS

Antes de introducir la metodología basada en redes neuronales, presentaremos el método de diferencias finitas, un método numérico clásico para la resolución de ecuaciones elípticas bidimensionales en dominios rectangulares. Al igual que en el caso unidimensional, la idea fundamental de este método consiste en reemplazar las segundas derivadas presentes en la EDP por aproximaciones discretas evaluadas sobre una malla uniforme, obteniendo así un sistema de ecuaciones. En particular, consideraremos el siguiente ejemplo.

EL PROBLEMA DE POISSON

Como ejemplo, procedemos a resolver la ecuación de Poisson bidimensional con solución suave y condiciones de frontera de Dirichlet de la siguiente forma

$$\begin{aligned} u_{xx} + u_{yy} &= g(x,y), \quad (x,y) \in \Omega, \\ u(x,y) &= u_D(x,y), \quad (x,y) \in \partial\Omega, \end{aligned} \tag{5.73}$$

donde el lado derecho de la ecuación está dado por

$$u(x,y) = -2\pi^2 \operatorname{sen}(\pi x)\operatorname{sen}(\pi y) \tag{5.74}$$

y u_D se calcula a partir de la solución analítica dada por

$$u(x,y) = \operatorname{sen}(x)\operatorname{sen}(y). \tag{5.75}$$

En este ejemplo el dominio está dado por $\Omega = (-1,1)\times(-1,1)$.

Ahora dividimos cada lado del cuadrado en $N+1$ puntos equidistantes, de modo que el tamaño de paso sea la misma constante en ambas direcciones, $h = 2/N$. Así, las segundas derivadas parciales en cada punto $\boldsymbol{x}_{ij} = (x_i, y_j)$ se aproximan mediante las fórmulas:

$$\frac{\partial^2 u}{\partial x^2}(\boldsymbol{x}_{ij}) \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \quad \frac{\partial^2 u}{\partial y^2}(\boldsymbol{x}_{ij}) \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2},$$

donde $u_{ij} = u(x_i, y_j)$ para cada $i = 2,\ldots,N$ y $j = 2,\ldots,N$. Estas aproximaciones se obtienen a partir del Teorema de Taylor, el cual también indica que el método posee un error de truncamiento de $\mathcal{O}(h^2)$. Nótese que el esquema se aplica únicamente en los puntos interiores, ya que los valores en la frontera son conocidos.

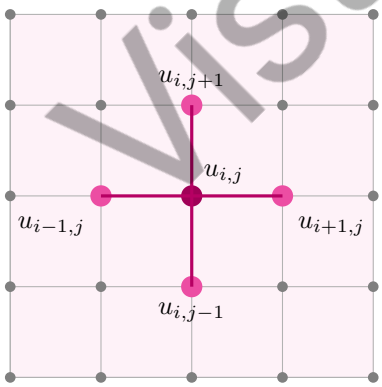


Figura 5.10 Esquema de diferencias finitas para la ecuación de Poisson bidimensional.

Sustituyendo ambas aproximaciones, se obtiene

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = h^2 g(x_i, y_j). \tag{5.76}$$

Aplicando esta expresión en cada punto interior del dominio, se obtiene un sistema de ecuaciones lineales en el que las condiciones de frontera de Dirichlet se incorporan de manera directa, cerrando así el problema.

Vista previa 2026



Figura 5.11 El programa del método de diferencias finitas consiste de un sistema de ecuaciones lineales compuesta por el esquema numérico aplicado a cada punto del dominio.

Para mayores detalles sobre esta metodología, puede consultarse el libro de Burden [60]. A continuación, se presenta la implementación de este método para la aproximación numérica del problema, considerando una malla con $N = 20$ divisiones, lo que corresponde a un tamaño de paso de $h = 0.1$.

```python
import numpy as np
import matplotlib.pyplot as plt

#============================================================
# Solucion exacta y termino fuente
def solucion_exacta(x, y):
    return np.sin(np.pi*x) * np.sin(np.pi*y)
def termino_fuente(x, y):
    return -2.0* (np.pi**2) * np.sin(np.pi*x) * np.sin(np.pi*y)

#============================================================
# Solución de Poisson 2D con condiciones de Dirichlet
def poisson2D(F, G, h):
    N = F.shape[0]
    Ntot = N * N
    A = np.zeros((Ntot, Ntot))
    b = np.zeros(Ntot)
    for i in range(N):
        for j in range(N):
            k = i + j * N
            # Puntos en la frontera
            if i == 0 or i == N-1 or j == 0 or j == N-1:
                A[k, k] = 1.0
                b[k] = G[i, j]
            # Puntos interiores
            else:
                A[k, k]     = -4.0 / h**2
                A[k, k - 1] =  1.0 / h**2
                A[k, k + 1] =  1.0 / h**2
                A[k, k - N] =  1.0 / h**2
                A[k, k + N] =  1.0 / h**2
                b[k] = F[i, j]
    # Solucionador del sistema lineal
    uvec = np.linalg.solve(A, b)
    U = uvec.reshape((N, N), order='F') # Solución en 2D
    return U

#============================================================
# Programa principal

# Dominio
xI, xF, yI, yF = -1.0, 1.0, -1.0, 1.0

# Malla uniforme
N = 20
x = np.linspace(xI, xF, N+1)
y = np.linspace(yI, yF, N+1)
h = x[1] - x[0]
X, Y = np.meshgrid(x, y, indexing='ij')

# Solución exacta y término fuente
Uexa = solucion_exacta(X, Y)
F = termino_fuente(X, Y)

# Solución numérica
U = poisson2D(F, Uexa, h)
```

Código 5.2 Programa usando diferencias finitas para la solución de la ecuación de Poisson en 2D.

Notas

```
E = np.abs(Uexa-U)
s=np.sum(E**2)*h**2
L2 = np.sqrt(s)
LI = np.max(E)
print(f"Linf=LI:.6e")
print(f"L2=L2:.6e")
```

Notas

```
plt.figure()
plt.contourf(x,y,U.T)
plt.colorbar()
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

Figura 5.12 Cálculo de las normas del error y gráficas del programa de diferencias finitas.

Vista previa 2026

DEFINICIÓN
El orden de convergencia de los resultados numéricos se puede calcular mediante la fórmula:

$$\text{Orden} \approx \frac{\log(\|e_1\|/\|e_2\|)}{\log(h_1/h_2)},$$

donde $\|e_1\|$ y $\|e_2\|$ son las normas del error con tamaño de paso h_1 y h_2 respectivamente.

DEFINICIÓN
Sea e_{ij} el valor absoluto de cada punto $\boldsymbol{x}_{ij}$ de la malla. La norma L_∞ se define como

$$\|e\|_\infty = \max_{i,j}(e_{ij}),$$

y la norma L_2 como:

$$\|e\|_2 = \left(\frac{1}{h^2}\sum_{i,j} e_{ij}^2\right)^{1/2}.$$

Figura 5.13 Definición del orden de convergencia numérico y normas del error.

La Figura 5.14 presenta la solución aproximada y el error absoluto obtenidos mediante el método de diferencias finitas usando $N = 20$. Aunque la solución analítica no se muestra aquí, visualmente resulta prácticamente indistinguible de la aproximación numérica. Esto se confirma con los valores del error absoluto, cuyo máximo es menor que 9×10^{-3}. Otra característica esperada es que los errores de mayor magnitud se localicen en el interior del dominio, ya que en la frontera no se realiza ninguna aproximación, sino que se imponen directamente los valores exactos dados por las condiciones de Dirichlet.

Un análisis de la convergencia del error para mallas con $N = 10$, 20, 40, 80, 160 divisiones se presenta en la tabla de abajo. En ella se reportan los errores en las normas L_2 y L_∞ para cada malla, así como el orden numérico estimado entre una ejecución y la anterior. La fórmula empleada para calcular dicho orden se muestra en la Figura 5.13.

Nótese que, a medida que aumenta el número de puntos en la malla, la aproximación numérica se vuelve más precisa. Además, al observar ambas normas del error absoluto, se aprecia que estas se reducen aproximadamente a una cuarta parte cada vez que se duplica N. En consecuencia, y en concordancia con los resultados teóricos, el orden de convergencia obtenido es $\mathcal{O}(h^2)$.

Tabla 5.4 Orden de convergencia del método de diferencias finitas.

N	Norma L_2	Orden	Norma L_∞	Orden
10	3.55×10^{-2}	—	3.03×10^{-2}	—
20	8.26×10^{-3}	2.02	8.26×10^{-3}	1.87
40	2.05×10^{-3}	2.00	2.05×10^{-3}	2.00
80	5.14×10^{-4}	2.00	5.14×10^{-4}	2.00
160	1.28×10^{-4}	2.00	1.28×10^{-4}	2.00

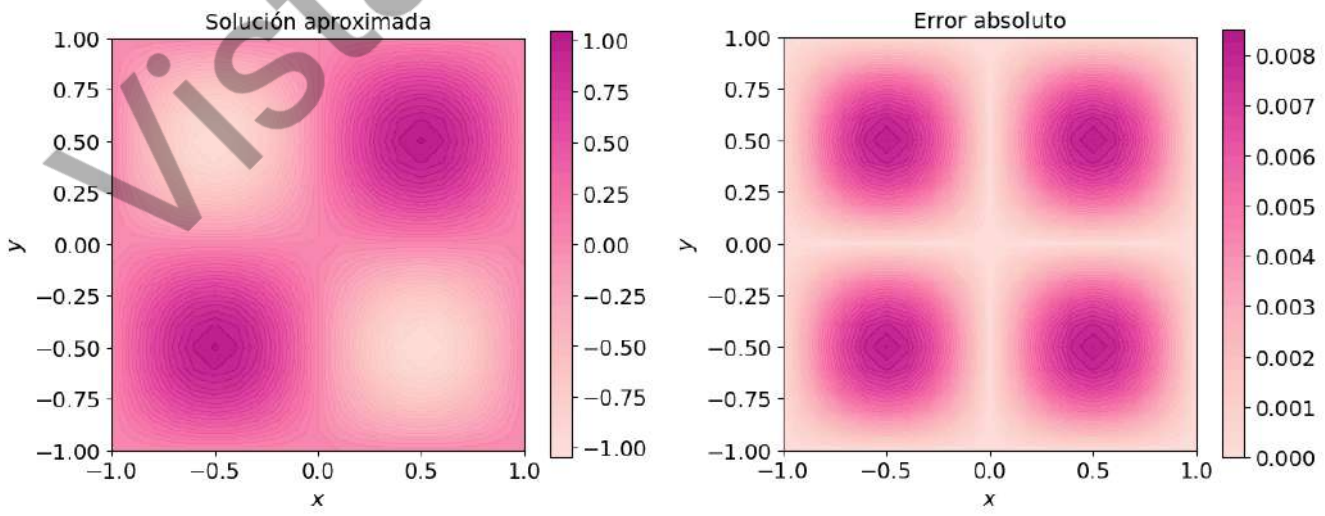


Figura 5.14 Solución aproximada y error absoluto del método de diferencias finitas usando $N = 20$.

Vista previa 2026

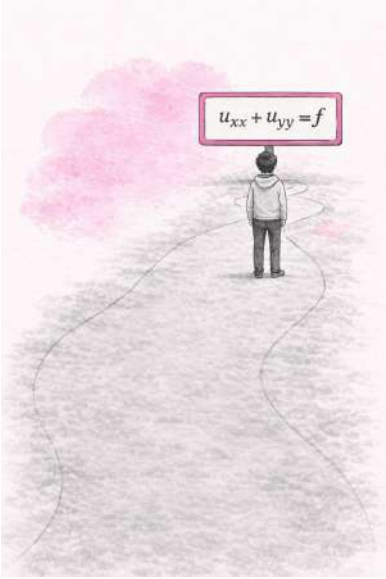


Figura 5.15 Finalmente hemos llegado al punto de resolver EDPs ampliamente usadas para modelar fenómenos físicos mediante redes neuronales.

5.4 APROXIMACIONES CON REDES NEURONALES

Nuestro objetivo es construir una red neuronal multicapa que, dentro del marco de las *Redes Neuronales Informadas por la Física* (*Physics-Informed Neural Networks*, PINNs), permita aproximar la solución $u(x, y)$ de (5.69) a partir de la información física contenida en la propia ecuación diferencial y en las condiciones de frontera asociadas.

El problema considerado guarda una estructura análoga a la estudiada en el caso unidimensional; sin embargo, la extensión a dos dimensiones introduce nuevos elementos que requieren una formulación más general y un tratamiento computacional más cuidadoso. Por esta razón, en las siguientes secciones nos concentraremos en describir cada una de las etapas de la metodología, enfatizando las modificaciones necesarias para abordar problemas bidimensionales. Siguiendo el enfoque adoptado en los capítulos anteriores, la metodología se describirá con base en la Figura 5.16, donde se resume de forma ilustrativa la secuencia de etapas necesarias para plantear, entrenar y validar el modelo de redes neuronales empleado en la resolución del problema.

En primer lugar, se presentará la construcción del modelo basado en redes neuronales, incluyendo la arquitectura de la red, la representación de sus capas, las funciones de activación y los parámetros entrenables (pesos y sesgos). Posteriormente, se discutirán los demás componentes esenciales de la formulación, tales como la selección de puntos de entrenamiento, la construcción de la función de pérdida, el proceso de entrenamiento y, finalmente, la evaluación del modelo obtenido.

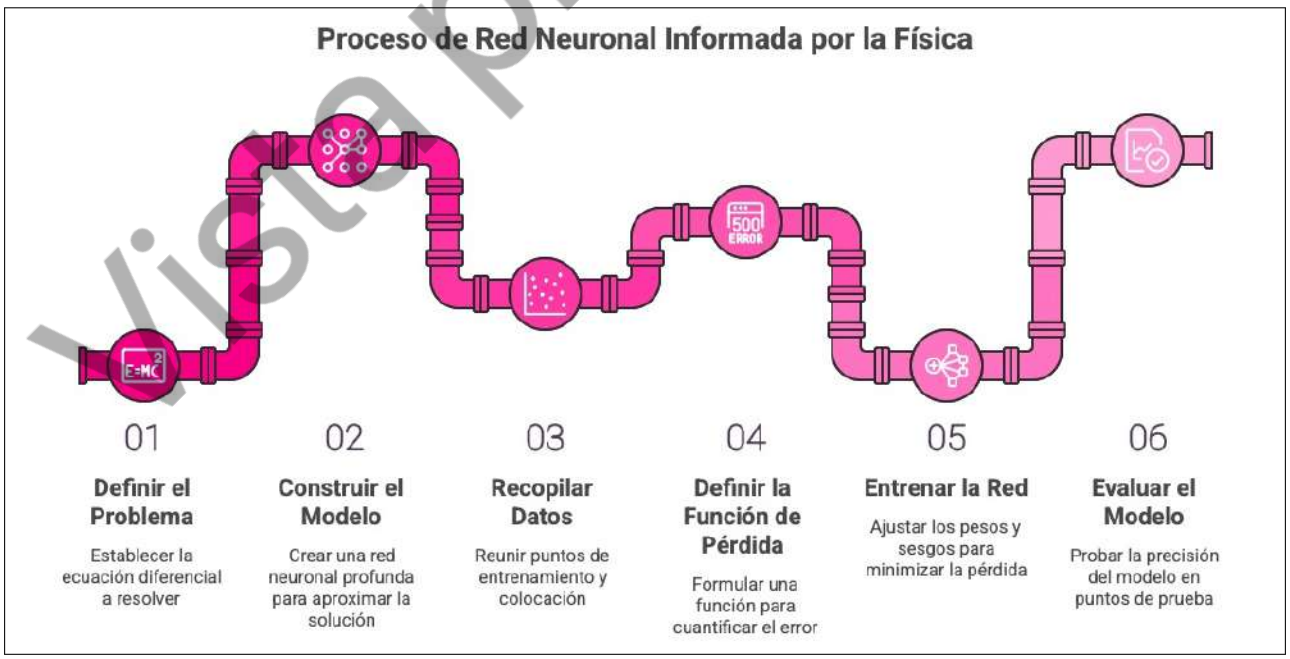


Figura 5.16 Diagrama ilustrativo de los pasos necesarios para resolver EDPs utilizando redes neuronales.

Vista previa 2026



Figura 5.17 A diferencia de los casos previamente estudiados, los problemas bidimensionales suelen requerir arquitecturas neuronales más profundas para obtener mejores aproximaciones.

5.4.1 El modelo basado en redes neuronales

El punto de partida de esta metodología consiste en asumir que todos los términos que definen la EDP son conocidos, mientras que la solución exacta u permanece desconocida y debe ser aproximada.

EL MODELO BASADO EN REDES

Se introduce una red neuronal multicapa como una función aproximante de la forma

$$u(x, y) \approx \widetilde{u}(x, y; W, \boldsymbol{b}), \tag{5.77}$$

donde $\widetilde{u}(x, y; W, \boldsymbol{b})$ representa la salida de la red neuronal evaluada en el punto (x, y), y W y $\boldsymbol{b}$ denotan, respectivamente, el conjunto de pesos y sesgos (*biases*) entrenables del modelo. El objetivo será ajustar dichos parámetros de tal manera que $\widetilde{u}$ satisfaga, en la medida de lo posible, tanto la EDP en el interior del dominio como las condiciones de frontera asociadas.

Aunque en los capítulos anteriores bastaba con emplear redes neuronales relativamente sencillas, compuestas por una o dos capas ocultas, en el caso bidimensional resulta conveniente utilizar arquitecturas más profundas para lograr una mejor capacidad de aproximación. La red neuronal multicapa propuesta consiste de una capa de entrada, una capa de salida y un conjunto de capas ocultas intermedias. Cada capa oculta contiene N_e neuronas, como se ilustra en la Figura 5.18. La primera capa siempre recibe como entrada el punto espacial $\boldsymbol{x} = (x, y) \in \mathbb{R}^2$, mientras que la capa de salida produce el valor aproximado $\widetilde{u}(\boldsymbol{x}) \in \mathbb{R}$.

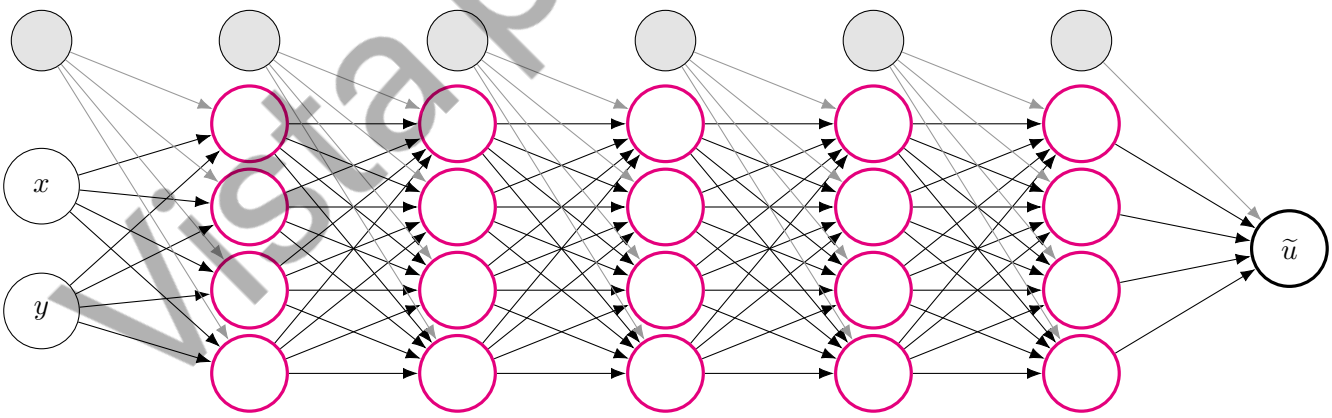


Figura 5.18 Ejemplo de una red neuronal con cinco capas ocultas de cuatro neuronas cada una.

En el ejemplo ilustrado en la Figura 5.18, la red neuronal tiene la configuración

$$\text{capas} = (2, 4, 4, 4, 4, 4, 1),$$

lo que significa que la red recibe dos entradas espaciales, contiene cinco capas ocultas con cuatro neuronas cada una y una única salida escalar.

Vista previa 2026



Figura 5.19 El conocimiento adquirido en problemas más sencillos permite ahora interpretar con mayor facilidad la relación entre la representación gráfica y la formulación matemática de una red neuronal.

A continuación, presentamos la formulación matemática de esta arquitectura de red neuronal.

DEFINICIÓN

Una red neuronal multicapa $\widetilde{u}(\boldsymbol{x}; W, \boldsymbol{b})$ con dos entradas, cinco capas ocultas compuestas cada una por cuatro neuronas y una única salida, se define mediante las siguientes ecuaciones:

$$\begin{aligned}
\boldsymbol{z}^{[1]} &= W^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}, & \boldsymbol{a}^{[1]} &= \phi(\boldsymbol{z}^{[1]}),\\
\boldsymbol{z}^{[2]} &= W^{[2]}\boldsymbol{a}^{[1]} + \boldsymbol{b}^{[2]}, & \boldsymbol{a}^{[2]} &= \phi(\boldsymbol{z}^{[2]}),\\
\boldsymbol{z}^{[3]} &= W^{[3]}\boldsymbol{a}^{[2]} + \boldsymbol{b}^{[3]}, & \boldsymbol{a}^{[3]} &= \phi(\boldsymbol{z}^{[3]}),\\
\boldsymbol{z}^{[4]} &= W^{[4]}\boldsymbol{a}^{[3]} + \boldsymbol{b}^{[4]}, & \boldsymbol{a}^{[4]} &= \phi(\boldsymbol{z}^{[4]}),\\
\boldsymbol{z}^{[5]} &= W^{[5]}\boldsymbol{a}^{[4]} + \boldsymbol{b}^{[5]}, & \boldsymbol{a}^{[5]} &= \phi(\boldsymbol{z}^{[5]}),\\
z^{[6]} &= W^{[6]}\boldsymbol{a}^{[5]} + b^{[6]}, & \widetilde{u} &= z^{[6]}.
\end{aligned}$$

Aquí, $\boldsymbol{x} \in \mathbb{R}^2$ representa el vector de entrada, mientras que $\boldsymbol{z}^{[\ell]}$ y $\boldsymbol{a}^{[\ell]}$ denotan, respectivamente, la preactivación y la activación de la capa oculta ℓ, para $\ell = 1, \ldots, 5$. En particular, se tiene que

$$\boldsymbol{z}^{[\ell]} \in \mathbb{R}^4, \qquad \boldsymbol{a}^{[\ell]} \in \mathbb{R}^4, \qquad \ell = 1, \ldots, 5,$$

mientras que

$$z^{[6]} \in \mathbb{R}, \qquad \widetilde{u} \in \mathbb{R}.$$

Es decir, cada capa oculta contiene cuatro neuronas, y la red produce finalmente un único valor de salida, el cual corresponde a la aproximación de la solución buscada. Además, se tiene:

- $W^{[1]} \in M_{4\times 2}(\mathbb{R})$ es la matriz de pesos que conecta la capa de entrada con la primera capa oculta.
- $\boldsymbol{b}^{[1]} \in \mathbb{R}^4$ es el vector de sesgos asociado a la primera capa oculta.
- Para $\ell = 2, 3, 4, 5$, se tiene que $W^{[\ell]} \in M_{4\times 4}(\mathbb{R})$ es la matriz de pesos que conecta la capa oculta $(\ell - 1)$ con la capa ℓ.
- Para $\ell = 2, 3, 4, 5$, se tiene que $\boldsymbol{b}^{[\ell]} \in \mathbb{R}^4$ es el vector de sesgos asociado a la capa oculta ℓ.
- $W^{[6]} \in M_{1\times 4}(\mathbb{R})$ es la matriz de pesos que conecta la quinta capa oculta con la única neurona de salida.
- $b^{[6]} \in \mathbb{R}$ es el sesgo de la neurona de salida.

En esta formulación, la función ϕ se aplica componente a componente en cada capa oculta, mientras que en la capa de salida no se utiliza función de activación.

Notas

A menos que se indique lo contrario, en este trabajo se utilizará la función tangente hiperbólica, $\phi(z) = \tanh(z)$, como función de activación en las capas ocultas.

Figura 5.20 Función de activación.

Vista previa 2026

Ejemplo.
Si consideramos la ecuación diferencial parcial llamada ecuación de Poisson:

$$u_{xx} + u_{yy} = g$$

Entonces el residuo de la ecuación diferencial está dada por

$$f := u_{xx} + u_{yy} - g.$$

Si u fuera la solución exacta del problema, entonces

$$f(x, y) = 0.$$

Figura 5.21 Ejemplo del residuo de una EDP.

Notemos que, en cada capa de la red neuronal, una vez realizada la transformación lineal correspondiente, se aplica una función de activación ϕ, la cual introduce la no linealidad necesaria para que la red pueda aproximar funciones complejas. Es importante subrayar que esta regla no se aplica en la capa de salida. En dicha capa, la red únicamente realiza la combinación lineal final de las activaciones provenientes de la capa anterior. Esta elección es natural cuando se desea aproximar directamente una cantidad continua, como la solución de una ecuación diferencial parcial.

Por otro lado, la función de activación deben elegirse de manera que la red sea lo suficientemente expresiva para aproximar soluciones complejas, pero también que sus derivadas puedan calcularse de forma eficiente y estable. Así, la función de activación tangente hiperbólica resulta particularmente conveniente porque se trata de una función suave, derivable tantas veces como sea necesario.

Más adelante se mostrará, a través de los experimentos numéricos, que esta función de activación exhibe un comportamiento robusto en los ejemplos considerados para resolver ecuaciones elípticas con dominios en dos dimensiones.

5.4.2 Incorporación de la EDP al modelo

El aspecto fundamental que distingue la resolución de EDP mediante PINNs de un problema convencional de ajuste de datos radica en la manera en que la ecuación diferencial y, por tanto el modelo físico subyacente, se incorpora directamente al proceso de entrenamiento. De forma análoga a lo presentado en el caso unidimensional, la idea central es conceptualmente sencilla, aunque su implementación práctica resulta más elaborada: introducir la EDP a través de un término residual que se evalúa en un conjunto adicional de puntos dentro del dominio.

Así, para incorporar la ecuación diferencial al proceso de entrenamiento, se define una función auxiliar asociada al residuo de la ecuación diferencial, obtenida al sustituir la solución exacta por su aproximación neuronal. Así, se introduce la función

$$f := \nabla \cdot (\beta \nabla u) + \kappa u - g, \tag{5.78}$$

de modo que, si u fuera la solución exacta del problema, entonces se tendría

$$f(x, y) = 0 \quad \text{para todo} \quad (x, y) \in \Omega. \tag{5.79}$$

En el contexto de PINNs, esta idea se traslada a la aproximación neuronal al definir el residual inducido por la red como sigue.

Vista previa 2026

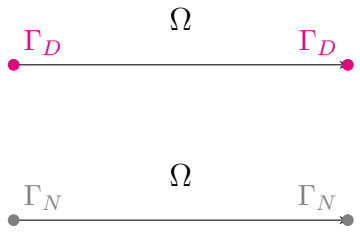


Figura 5.22 En el caso de dominios en 1D, la frontera estaba constituida únicamente por dos puntos, sobre los cuales se pueden imponer distintos tipos de condiciones de frontera.

EL RESIDUO DE LA ECUACIÓN

Para incorporar la ecuación diferencial parcial dentro del proceso de entrenamiento, se introduce una función auxiliar llamada residuo, definida como

$$\widetilde{f}(x, y; W, \boldsymbol{b}) = \nabla \cdot (\beta \nabla \widetilde{u}) + \kappa \widetilde{u} - g. \tag{5.80}$$

Si $\widetilde{u}$ coincidiera exactamente con la solución del problema, entonces se tendría $\widetilde{f}(x, y; W, \boldsymbol{b}) = 0$, para todo $(x, y) \in \Omega$. Por ello, durante el entrenamiento, la red neuronal no solo busca una aproximación de la solución $\widetilde{u}$, sino que también respete la ley física o matemática que modela el fenómeno.

Observe que el operador diferencial presente en la ecuación puede escribirse de manera explícita como

$$\nabla \cdot (\beta \nabla \widetilde{u}) = (\beta \widetilde{u}_x)_x + (\beta \widetilde{u}_y)_y.$$

Por lo tanto, para evaluar el residual asociado al problema, es necesario calcular derivadas parciales de la red neuronal $\widetilde{u}$ con respecto a las variables espaciales.

Un aspecto clave de esta metodología es que las derivadas parciales de $\widetilde{u}$ con respecto a las variables espaciales pueden calcularse de manera exacta a nivel computacional. Esta técnica permite evaluar derivadas de funciones compuestas implementadas como grafos computacionales, sin recurrir a esquemas como diferencias finitas o elemento finito, entre otras. En particular, gracias a la regla de la cadena y al uso de algoritmos como la retropropagación [15, 19, 43], es posible obtener de forma eficiente tanto las derivadas espaciales como los gradientes de la función de pérdida con respecto a los parámetros entrenables W y $\boldsymbol{b}$.

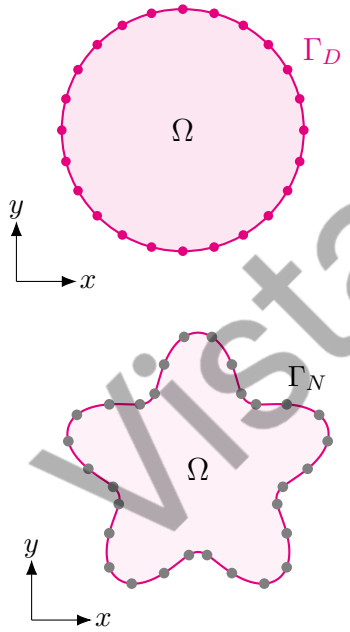


Figura 5.23 Ejemplos de distribución de puntos de entrenamiento sobre la frontera, clasificados según el tipo de condición de frontera impuesta en el problema.

5.4.3 Datos conocidos: los puntos de entrenamiento

Dada la naturaleza de los problemas elípticos, es natural que la información conocida se localice sobre la frontera del dominio, a través de las condiciones de frontera impuestas. En el caso de dominios en una dimensión, la frontera estaba constituida únicamente por dos puntos, sobre los cuales se impusieron condiciones de frontera. En contraste, en el caso bidimensional, la frontera está formada por porciones unidimensionales del contorno del dominio, lo que ofrece múltiples posibilidades para la selección y distribución de los puntos de entrenamiento.

No obstante, a diferencia de muchos métodos numéricos clásicos, en el enfoque basado en redes neuronales no existe una restricción estricta (desde el punto de vista de la implementación) respecto al número de puntos utilizados ni a la forma en que estos se distribuyen sobre la frontera. Esta flexibilidad permite adaptar el conjunto de datos a la geometría del dominio y a las necesidades particulares del problema.

Vista previa 2026



Figura 5.24 En problemas físicos, puede ser difícil obtener mediciones sobre toda la superficie de interés debido a restricciones experimentales o a la complejidad geométrica del dominio.

Naturalmente, es de esperarse que, mientras más abundante y mejor distribuida sea la información disponible, mayor sea la capacidad de la red para reproducir con precisión el comportamiento de la solución en esas regiones, en contraste con aquellas zonas donde se dispone de pocos datos o incluso de ninguno.

Esta flexibilidad en la selección de los puntos de entrenamiento resulta particularmente valiosa en aplicaciones reales, donde la información disponible suele ser limitada, ya sea por el costo asociado a su obtención o por la imposibilidad práctica de contar con mediciones en toda la frontera.

Sean N_D y $N_{\mathcal{N}}$ los números que representan la cantidad de puntos de entrenamiento asociados a condiciones de frontera de tipo Dirichlet y Neumann, respectivamente. Entonces, los conjuntos de puntos de entrenamiento pueden definirse como sigue:

- Los puntos de entrenamiento de tipo de Dirichlet están dados por
$$\left\{(\hat{x}_i^D, \hat{y}_i^D)\right\}_{i=1}^{N_D},$$
los cuales se ubican sobre la porción $\partial\Omega_D$. A cada uno de estos puntos se le asocia un valor conocido de la solución,
$$\left\{\hat{u}_i^D\right\}_{i=1}^{N_D}.$$
- Por otro lado, si se consideran condiciones tipo Neumann, los puntos de entrenamiento están dados por
$$\left\{(\hat{x}_i^{\mathcal{N}}, \hat{y}_i^{\mathcal{N}})\right\}_{i=1}^{N_{\mathcal{N}}},$$
los cuales se ubican en la porción $\partial\Omega_N$. A cada uno de estos puntos se le asocia un valor conocido de la derivada normal de la solución,
$$\left\{\hat{q}_i^{\mathcal{N}}\right\}_{i=1}^{N_{\mathcal{N}}},$$
donde $\hat{q}_i^{\mathcal{N}}$ es el valor prescrito de $\beta(\hat{x}_i^{\mathcal{N}}, \hat{y}_i^{\mathcal{N}})\nabla u(\hat{x}_i^{\mathcal{N}}, \hat{y}_i^{\mathcal{N}})\cdot\boldsymbol{n}$, siendo $\boldsymbol{n}$ el vector normal unitario exterior a la frontera.
- Los puntos de entrenamiento asociados a condiciones de Robin se definen de manera análoga.

En conjunto, estos tres tipos de datos de frontera permiten guiar a la red neuronal para reproducir adecuadamente el comportamiento de la solución sobre $\partial\Omega$, siempre que dicho tipo de condición esté presente en el problema considerado.

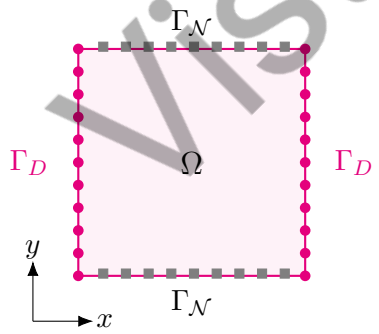


Figura 5.25 Ejemplo de distribución de dos tipos de puntos de entrenamiento sobre la frontera de un mismo problema.

Las Figuras 5.25 y 5.26 muestran dos ejemplos de distribución de puntos de entrenamiento asociados a condiciones de frontera de tipo Dirichlet y Neumann. En ambos casos, se ha considerado una distribución uniforme de los datos conocidos sobre las porciones correspondientes de la frontera.

Vista previa 2026

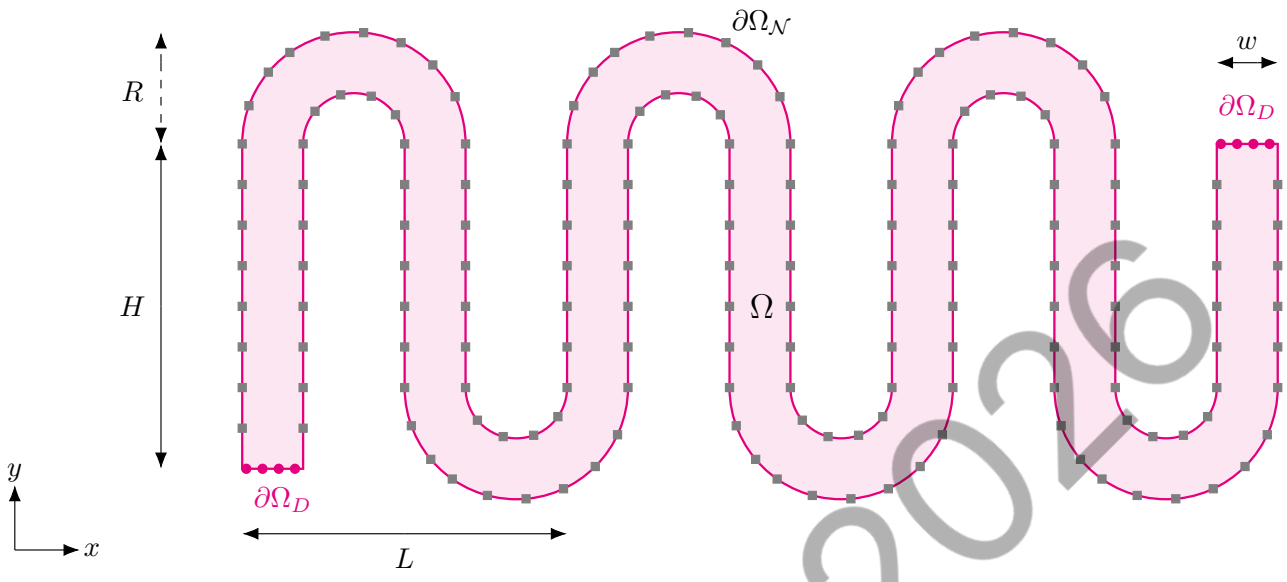


Figura 5.26 Dominio con geometría serpenteante que inspirada en la configuración de una resistencia eléctrica. En este ejemplo se ilustra un problema con fronteras curvas y diferentes tipos de puntos de entrenamiento.

En consecuencia, los pesos y sesgos en (5.77) pueden determinarse a partir de distintos conjuntos finitos de puntos de entrenamiento asociados a condiciones de frontera de tipo Dirichlet, Neumann y/o Robin. Sin embargo, la sola incorporación de estos datos no es suficiente para garantizar que la aproximación satisfaga la EDP en el interior del dominio. Para ello, es necesario introducir un conjunto adicional de puntos, denominados *puntos de colocación*, sobre los cuales se evalúa el residuo (5.80). A continuación, se describe con mayor detalle el papel que desempeñan estos puntos dentro de la formulación.



Figura 5.27 Los puntos de colocación son necesarios para incluir la EDP en el entrenamiento.

5.4.4 Datos conocidos: los puntos de colocación

A diferencia de un problema clásico de ajuste de datos, donde el entrenamiento se apoya exclusivamente en pares de entrada–salida observados, en el contexto de PINNs se introduce una segunda fuente de información: la propia ecuación diferencial. Esta idea da lugar a un ingrediente distintivo y fundamental del método, a saber, los *puntos de colocación*.

DEFINICIÓN
Una de las características más importantes de las PINNs es la incorporación de *puntos de colocación*, es decir, puntos del dominio donde no necesariamente se dispone de datos observados, pero donde sí se conoce que la solución debe satisfacer la ecuación diferencial parcial. De esta manera, la física del problema se incorpora directamente al entrenamiento.

Vista previa 2026

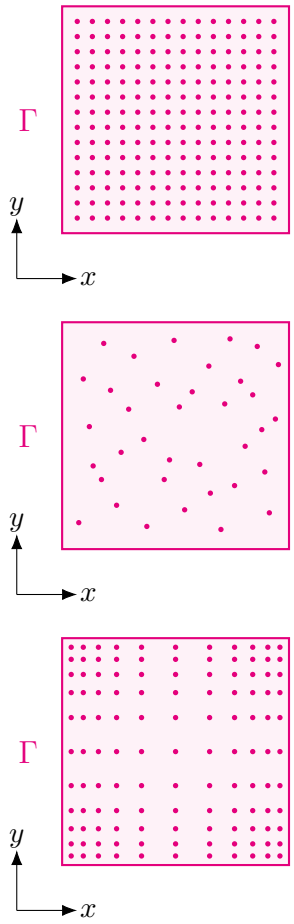


Figura 5.28 Ejemplo de puntos de colocación distribuidos de manera uniforme, aleatoria y adaptativa en un dominio rectangular.

Los puntos de colocación están dados por

$$\left\{(\hat{x}_j^f, \hat{y}_j^f)\right\}_{j=1}^{N_f},$$

los cuales se distribuyen en el interior del dominio Ω y, en algunos casos, también sobre partes de la frontera. Aquí N_f representa el número de puntos de colocación. En estos puntos no se conoce necesariamente el valor exacto de la solución u, pero sí se exige que la ecuación diferencial sea satisfecha. Para ello, se evalúa el residual

$$\left\{\widetilde{f}_j\right\}_{j=1}^{N_f},$$

con el objetivo de que sea cercano a cero en cada uno de dichos puntos.

Es importante señalar que los puntos de colocación desempeñan un papel análogo al mallado utilizado en los métodos numéricos clásicos, ya que permiten imponer la ecuación diferencial en un conjunto discreto de ubicaciones dentro del dominio. Es importante señalar que este conjunto de puntos también puede seleccionarse de manera flexible, ya sea de forma uniforme, aleatoria o siguiendo alguna estrategia adaptativa, dependiendo de la geometría del dominio y de las características del problema.

Es razonable esperar que, cuanto mayor y mejor distribuida sea la información disponible, más precisa será la capacidad de la red para aproximar el comportamiento de la EDP en dichas regiones. Posibles distribuciones de los puntos de colocación se ilustra en la Figura 5.28 para un dominio rectangular. Por otro lado la Figura 5.29 se muestra los puntos de colocación usados para el ejemplo con un dominio inspirado en una resistencia eléctrica presentado en la sección anterior.

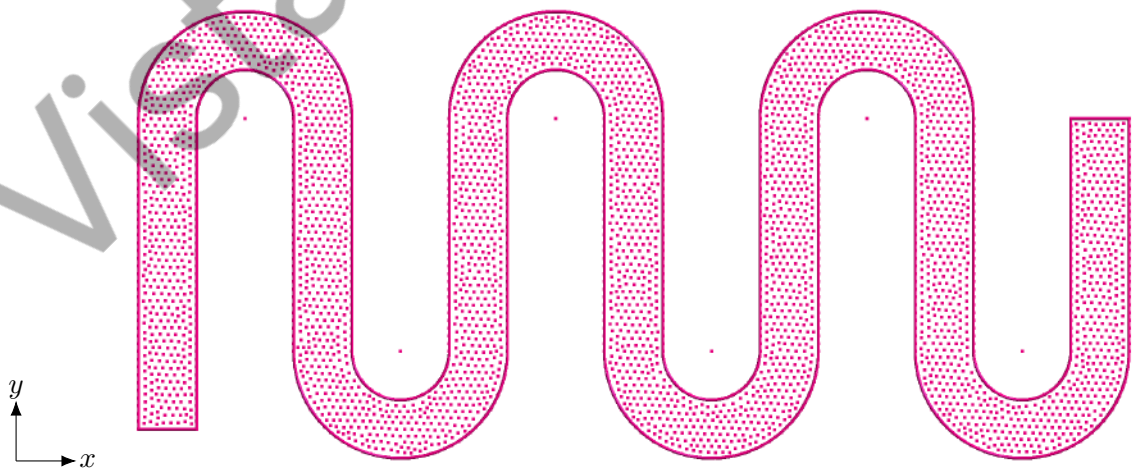


Figura 5.29 Ejemplo de un dominios Ω con una posible configuración del conjunto de puntos de colocación requeridos para entrenar la red neuronal.

Vista previa 2026



Figura 5.30 La generación de puntos de colocación en dominios irregulares puede realizarse de manera eficiente mediante el software de acceso libre *Gmsh*.

Selección de los puntos de colocación

En redes neuronales no existe una restricción rígida respecto al número de puntos de colocación ni a la manera en que estos deben distribuirse en el dominio. Esta flexibilidad constituye una de las principales ventajas del método, ya que permite adaptar la formulación a la geometría del problema. No obstante, dicha libertad también plantea la cuestión de cómo seleccionar estos puntos de forma adecuada y qué configuración es más conveniente.

En los ejemplos numéricos considerados en este trabajo, la generación de los puntos de colocación se realizará atendiendo principalmente a la geometría del dominio y a la manera en que se busca representar la información espacial del problema.

- En dominios rectangulares se dispone de más alternativas. En particular, se considerarán mallados uniformes obtenidos al dividir los ejes x y y en un número determinado de intervalos. A partir de esta discretización, los puntos de colocación podrán tomarse en su totalidad de los vértices o bien seleccionarse parcialmente de manera aleatoria. Asimismo, existen estrategias más elaboradas que permiten generar distribuciones aleatorias con una cobertura espacial más equilibrada, como la técnica del *hipercubo latino*.
- En dominios irregulares, las posibilidades de generación se encuentran más condicionadas por la propia geometría del dominio. En este trabajo, los puntos de colocación se obtendrán a partir de los vértices de una malla triangular previamente construida mediante un software externo, como textcolorpinkMX*Gmsh*. Posteriormente, dichos puntos podrán emplearse de forma completa o seleccionarse aleatoriamente, dependiendo de los requerimientos del experimento numérico correspondiente.

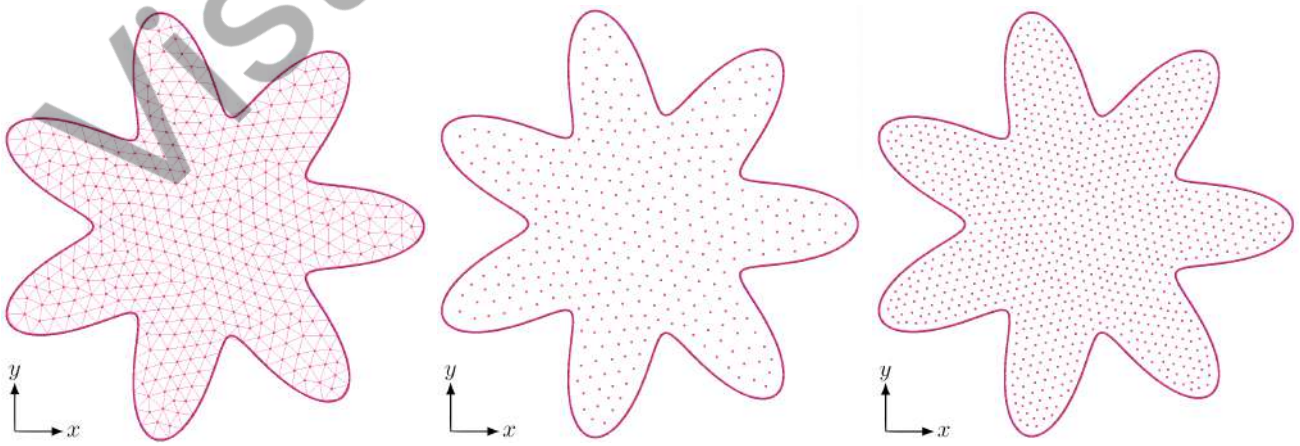


Figura 5.31 Ejemplos de puntos de colocación obtenidos de una malla triangular para un dominio irregular.

Vista previa 2026

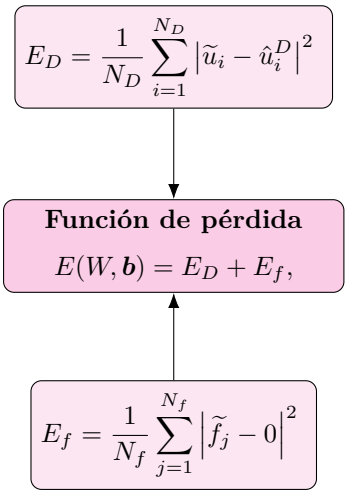


Figura 5.32 Componentes de la función de pérdida para un problema con sólo condiciones de frontera tipo Dirichlet.

5.4.5 La función de pérdida

Una vez definida la red neuronal que aproxima la solución del problema y los datos que disponemos, el siguiente paso consiste en determinar sus parámetros, es decir, los pesos W y los sesgos $\boldsymbol{b}$. Para ello, el entrenamiento se formula como un problema de optimización, cuyo objetivo es encontrar los valores de dichos parámetros que produzcan la mejor aproximación posible de la solución buscada.

En el contexto de las PINNs, esta etapa resulta especialmente importante, ya que el proceso de entrenamiento no solo busca ajustar la red a un conjunto de datos conocidos, sino también forzar que la solución aproximada satisfaga la ecuación diferencial en el dominio. Esto se logra mediante la construcción de una función de pérdida adecuada, la cual cuantifica simultáneamente el error en la frontera y el incumplimiento de la física del problema.

En este trabajo utilizamos nuevamente el *error cuadrático medio* (*Mean Squared Error*, MSE) como medida para el error. Este criterio permite cuantificar, en promedio, la diferencia entre los valores predichos por la red neuronal y los valores de referencia conocidos.

DEFINICIÓN

En su forma más básica, cuando únicamente se consideran condiciones de frontera de Dirichlet y la ecuación diferencial en el interior del dominio, la función de pérdida se define como

$$E(W,\boldsymbol{b}) = E_D + E_f, \tag{5.81}$$

donde E_D representa el error asociado a las condiciones de frontera de Dirichlet y E_f corresponde al error asociado al residuo de la ecuación diferencial.

Notas

La función de pérdida cuantifica simultáneamente el error en la frontera y el incumplimiento de ecuación diferencial parcial y por tanto la física del problema.

Figura 5.33 Función de activación.

De forma explícita, los aportes debidos a la condición de Dirichlet y al residuo físico en la ecuación (5.81) vienen dados por:

- Término de Dirichlet. Si se dispone del conjunto de puntos de entrenamiento tipo Dirichlet con su correspondiente valor conocido:

$$\left\{(\hat{x}_i^D, \hat{y}_i^D, \hat{u}_i^D)\right\}_{i=1}^{N_D},$$

entonces el error se define por

$$E_D = \frac{1}{N_D}\sum_{i=1}^{N_D}\left|\widetilde{u}(\hat{x}_i^D, \hat{y}_i^D; W, \boldsymbol{b}) - \hat{u}_i^D\right|^2. \tag{5.82}$$

Este término penaliza las discrepancias entre la predicción de la red neuronal y los valores conocidos de la solución sobre la frontera.

Vista previa 2026



Figura 5.34 Física de las tortillas: ¿Por qué se inflan? Recordemos que aproximar bien la ecuación no es suficiente para reproducir el fenómeno físico; también es necesario modelar correctamente las condiciones físicas del problema.

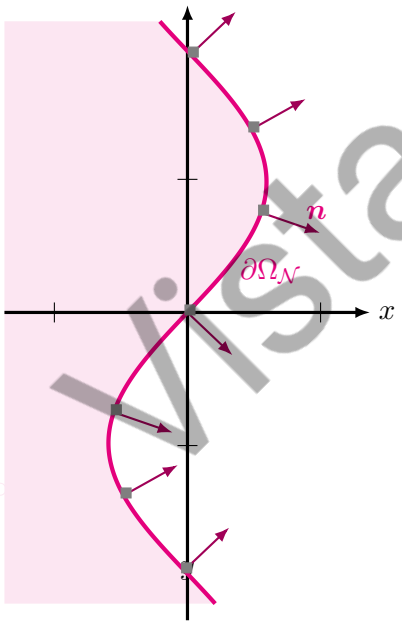


Figura 5.35 La normal sobre la frontera se considera un dato conocido. En dominios irregulares este cálculo puede ser complejo.

Término del residuo físico. Por otra parte, si se consideran los puntos de colocación

$$\left\{(\hat{x}_j^f, \hat{y}_j^f)\right\}_{j=1}^{N_f},$$

entonces el término asociado a la ecuación diferencial se construye como

$$E_f = \frac{1}{N_f}\sum_{j=1}^{N_f}\left|\widetilde{f}(\hat{x}_j^f, \hat{y}_j^f; W, \boldsymbol{b})\right|^2. \tag{5.83}$$

Aquí, $\widetilde{f}$ definido en la ecuación (5.80), representa el residuo de la ecuación diferencial obtenido al sustituir la solución aproximada $\widetilde{u}$ en la ecuación gobernante. En consecuencia, este término obliga a la red neuronal a producir funciones que satisfagan, en la medida de lo posible, la física del problema en el interior del dominio.

Incorporación de otras condiciones de frontera

La formulación anterior puede extenderse de manera natural para incluir otros tipos de condiciones de frontera, como las de Neumann o Robin. En tales casos, basta con añadir a la función de pérdida términos adicionales que penalicen el incumplimiento de dichas condiciones en los puntos de entrenamiento correspondientes.

Por ejemplo, si se dispone de un conjunto de puntos de entrenamiento tipo Neumann

$$\left\{(\hat{x}_i^{\mathcal{N}}, \hat{y}_i^{\mathcal{N}}, \hat{q}_i^{\mathcal{N}})\right\}_{i=1}^{N_{\mathcal{N}}},$$

donde $\hat{q}_i^{\mathcal{N}}$ representa el valor conocido de la derivada normal de la solución sobre la frontera. Entonces la función de pérdida se modifica a

$$E(W, \boldsymbol{b}) = E_D + E_{\mathcal{N}} + E_f, \tag{5.84}$$

donde el término adicional $E_{\mathcal{N}}$ está dado por

$$E_N = \frac{1}{N_{\mathcal{N}}}\sum_{i=1}^{N_{\mathcal{N}}}\left|\beta(\hat{x}_i^{\mathcal{N}}, \hat{y}_i^{\mathcal{N}})\nabla\widetilde{u}(\hat{x}_i^{\mathcal{N}}, \hat{y}_i^{\mathcal{N}}; W, \boldsymbol{b})\cdot\boldsymbol{n} - \hat{q}_i^{\mathcal{N}}\right|^2. \tag{5.85}$$

De manera completamente análoga, las condiciones de Robin pueden incorporarse añadiendo un término adicional que involucre la combinación lineal correspondiente entre la solución y su derivada normal.

Conviene enfatizar que, en la presente formulación, los vectores normales unitarios sobre la frontera se consideran datos conocidos y, en consecuencia, es responsabilidad del usuario proporcionarlos en aquellos puntos donde se prescriban condiciones de tipo Neumann o Robin. Aunque en dominios regulares su determinación suele ser directa, en geometrías irregulares o con fronteras curvas este cálculo puede involucrar consideraciones geométricas adicionales. En este sentido, la especificación adecuada de dichas normales constituye un elemento esencial en la implementación práctica del método.

Vista previa 2026



Figura 5.36 Debido al uso de arquitecturas neuronales más profundas, el entrenamiento adquiere un mayor costo computacional, por lo que es especialmente importante emplear un método de optimización eficiente.

5.4.6 El entrenamiento de la red neuronal

Con los elementos introducidos hasta este punto, puede observarse que el problema de resolver la ecuación diferencial parcial ha sido reformulado como un problema de optimización, en el que se busca minimizar la función de pérdida. En esencia, se trata de encontrar los valores de W y $\boldsymbol{b}$ para los cuales la función aproximante $\widetilde{u}(x, y; W, \boldsymbol{b})$ logre satisfacer simultáneamente la ecuación diferencial en el interior del dominio y las condiciones de frontera asociadas.

EL PROBLEMA DE OPTIMIZACIÓN

El entrenamiento del método basado en PINNs consiste en encontrar el conjunto de parámetros que minimizan la función de pérdida total. Matemáticamente, esto se expresa como

$$\min_{\theta} E(\theta),$$

donde $\theta = [W, \boldsymbol{b}]$ representa el conjunto completo de pesos y sesgos de la red neuronal.

En otras palabras, entrenar la red equivale a resolver un problema de minimización no lineal de alta dimensión, donde la calidad de la solución final depende directamente de qué tan pequeño logre hacerse el valor de la función de pérdida. Existen numerosos algoritmos para abordar este tipo de problemas de optimización. En este capítulo seguimos utilizado el método textcolorpinkMX*Adam*.



Figura 5.37 Principales librerías usadas para entrenar redes neuronales.

El algoritmo comienza con una estimación inicial de los parámetros y genera de manera iterativa una sucesión de aproximaciones con el objetivo de reducir progresivamente el valor de la función de pérdida hasta alcanzar un criterio de convergencia previamente establecido.

En cada iteración, el algoritmo requiere evaluar tanto la función de pérdida como sus gradientes con respecto a los parámetros entrenables. Estos gradientes se obtienen automáticamente mediante el mecanismo de *retropropagación* (*backpropagation*), implementado de manera eficiente en `Pytorch` o `TensorFlow` [44].

Este mismo esquema permite calcular automáticamente las derivadas espaciales necesarias para construir el residuo de la ecuación diferencial, incorporando de manera natural la información física al proceso de entrenamiento. De este modo, la red asociada al residuo, $\widetilde{f}(x, y; W, \boldsymbol{b})$, no se introduce como un modelo adicional independiente, sino que surge implícitamente a partir de $\widetilde{u}$ y de sus derivadas espaciales.

Vista previa 2026

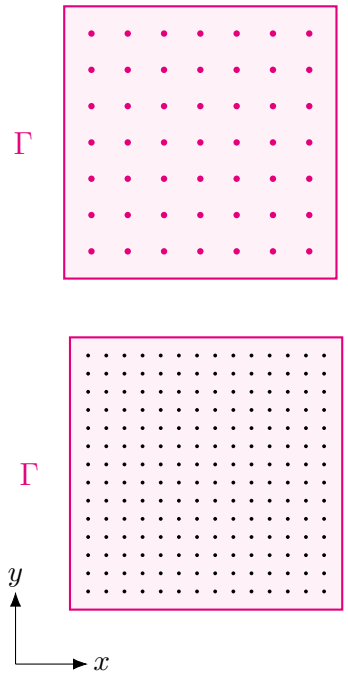


Figura 5.38 Aunque los puntos de prueba (abajo) podrían coincidir con los puntos de colocación (arriba), se recomienda, cuando sea posible, utilizar dos conjuntos distintos.

5.4.7 Evaluación del modelo

Una vez entrenada la red neuronal, la aproximación obtenida no se limita a un conjunto discreto de valores, sino que proporciona una representación funcional de la solución en todo el dominio. En otras palabras, el modelo entrenado define una función aproximante $\widetilde{u}$ la cual puede evaluarse en cualquier punto del dominio de interés.

Con el propósito de analizar la calidad de la aproximación obtenida, se introduce un conjunto adicional de puntos, denominado *conjunto de datos de prueba*, dado por

$$\left\{(\hat{x}_i^P, \hat{y}_i^P)\right\}_{i=1}^{N_P},$$

donde N_P representa el número total de puntos de prueba $\boldsymbol{x}_P$. Estos puntos pueden elegirse arbitrariamente dentro del dominio Ω (y, si se desea, también sobre la frontera), y en general son distintos de los puntos empleados durante el entrenamiento. A cada uno de estos puntos se le puede asociar un valor de referencia

$$\left\{\hat{u}_i^P\right\}_{i=1}^{N_P},$$

el cual proviene, por ejemplo, de una solución exacta conocida, de una solución numérica altamente refinada o de mediciones experimentales confiables. De esta manera, el conjunto de prueba permite comparar la predicción de la red neuronal con una referencia externa y, por tanto, cuantificar la precisión de la solución aproximada.

Es importante subrayar que estos puntos no participan directamente en el proceso de entrenamiento; su función principal es evaluar la capacidad de generalización del modelo, es decir, su habilidad para reproducir adecuadamente la solución en ubicaciones distintas de aquellas utilizadas para ajustar los parámetros de la red.

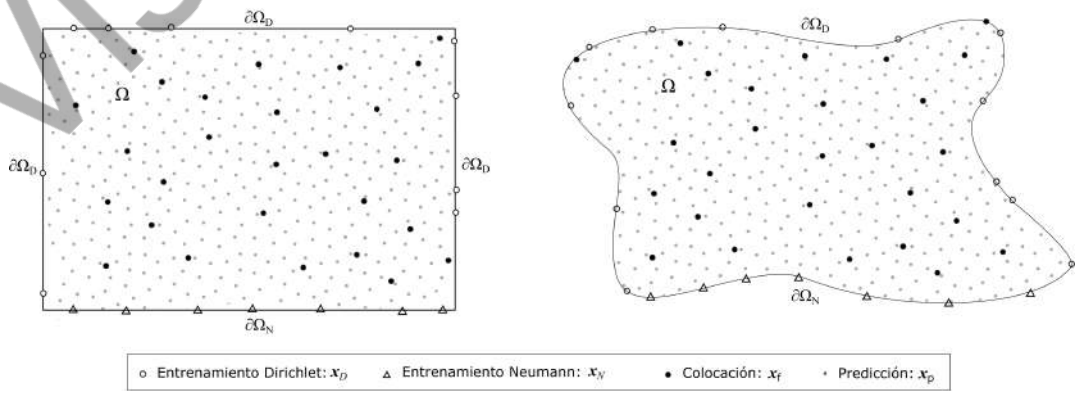


Figura 5.39 Ejemplos de dominios Ω: rectangular e irregular. Ubicación de los tres diferentes tipos de puntos requeridos para las simulaciones con redes neuronales: entrenamiento, colocación y prueba.

Vista previa 2026



Figura 5.40 Es momento de implementar todos los pasos de la metodología basada en redes neuronales usando Python y PyTorch.

5.5 EL PROGRAMA EN PYTHON

En el caso bidimensional es necesario introducir varios componentes del programa que no aparecían en la formulación unidimensional. Entre ellos destacan la definición de los puntos de entrenamiento y de colocación, la construcción de una red neuronal con dos entradas correspondientes a las coordenadas espaciales (x, y), así como la formulación del residual mediante diferenciación automática, la función de pérdida y la posterior evaluación del modelo sobre un conjunto independiente de prueba.

Aunque, en principio, la incorporación de todos estos elementos podría hacer pensar que la implementación en dos dimensiones es sustancialmente más compleja, en realidad gran parte de esa dificultad reside en detalles de programación que no alteran la estructura esencial del método. Por esta razón, se presentará primero un ejemplo base lo más simple posible: la solución de la ecuación de Laplace en un cuadrado unitario, utilizando puntos de entrenamiento, colocación y verificación generados a partir de un mallado uniforme.

Una vez establecido este esquema general, será posible extenderlo de manera natural hacia problemas más complejos o hacia configuraciones más flexibles, como el uso de puntos seleccionados aleatoriamente.

EL PROBLEMA DE LAPLACE

Procedemos a resolver la ecuación de Laplace bidimensional con solución suave y condiciones de frontera de Dirichlet de la siguiente forma

$$\begin{aligned} u_{xx} + u_{yy} &= 0, \quad (x, y) \in \Omega, \\ u(x, y) &= u_D(x, y), \quad (x, y) \in \partial\Omega, \end{aligned} \tag{5.86}$$

donde u_D se calcula a partir de la solución analítica dada por

$$u(x, y) = e^x \cos(y). \tag{5.87}$$

En este ejemplo el dominio está dado por $\Omega = (-1, 1) \times (-1, 1)$.

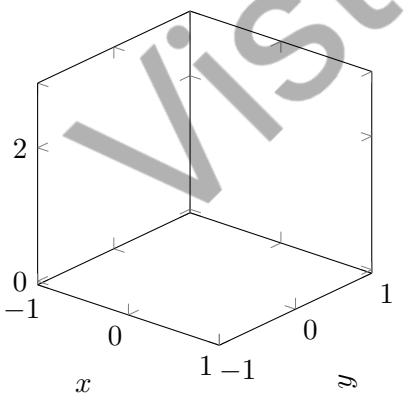


Figura 5.41 Los parámetros del programa son los mismos que del ejemplo calculado manualmente.

Obsérvese que, para este problema, el coeficiente difusivo es constante e igual a 1, mientras que el término de reacción es idénticamente nulo. De igual manera, el término fuente se anula en todo el dominio. En consecuencia, el residuo de la ecuación diferencial se reduce a

$$f := u_{xx} + u_{yy}.$$

En la Figura 5.41 se presenta la solución analítica correspondiente al problema. Se aprecia que dicha solución exhibe un comportamiento regular y suave en todo el dominio, sin la presencia de gradientes pronunciados.

Vista previa 2026

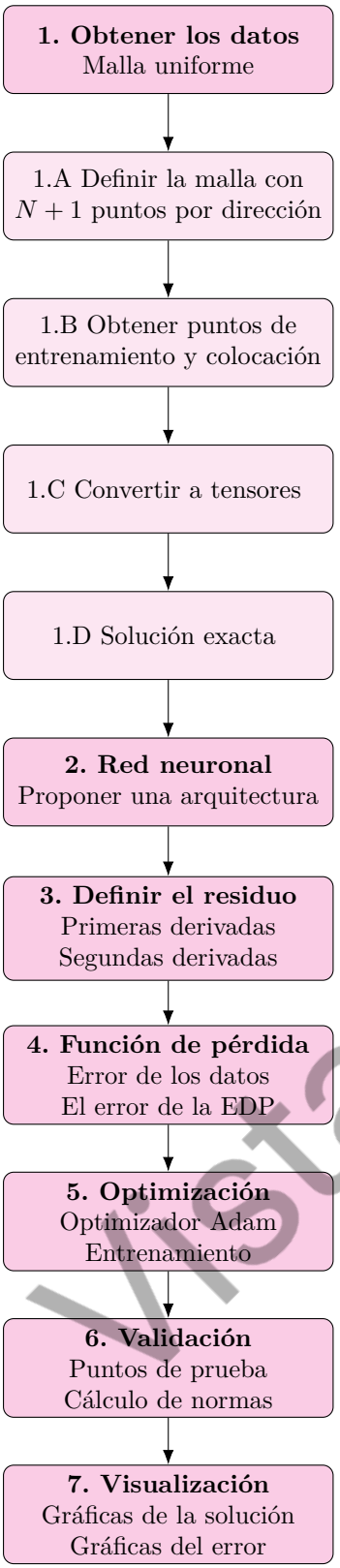


Figura 5.42 Esquema general del código completo.

Es importante señalar que la solución analítica se emplea únicamente para definir las condiciones de frontera de Dirichlet, evitando así su imposición manual. Sin embargo, dicha solución no se utiliza en ningún momento durante el proceso de entrenamiento de la red neuronal para aproximar la solución en los puntos interiores del dominio. Sin embargo, las características de esta solución sugiere que una red neuronal con una arquitectura de complejidad moderada debería ser suficiente para aproximar adecuadamente la solución del problema.

A continuación, se presentará la descripción del programa básico en Python utilizado para resolver el problema de Laplace planteado. Dicha descripción se desarrollará por partes, comenzando con la definición de los puntos de entrenamiento y de colocación.

5.5.1 Código para obtener los datos conocidos

Una de las secciones del código que presenta mayor variación corresponde a la generación de los datos de entrenamiento dado que estos dependen del problema que deseamos resolver. Para la ecuación de Laplace con condiciones de Dirichlet, se ha establecido que los puntos de entrenamiento y colocación se generarán a partir de un mallado uniforme en el dominio cuadrado. Para ello, cada eje coordenado se divide en N intervalos, lo que produce $N+1$ nodos por dirección. De esta manera, se construye una malla uniforme bidimensional con $(N-1)^2$ puntos interiores y $4N$ puntos sobre la frontera. Notar que en nuestro ejemplo usamos $N = 10$.

```
# ========================================================
# 1.A Dominio y malla uniforme
# ========================================================
xI, xF = -1.0, 1.0
yI, yF = -1.0, 1.0

N = 10  # numero de intervalos por eje (N+1 puntos)

x = np.linspace(xI, xF, N+1)
y = np.linspace(yI, yF, N+1)
X, Y = np.meshgrid(x, y)

# Todos los nodos de la malla
XY = np.column_stack((X.ravel(), Y.ravel()))
```

Código 5.3 Generación de la malla uniforme.

En este ejemplo particular, se utilizarán todos los puntos de la frontera como puntos de entrenamiento asociados a las condiciones de Dirichlet, mientras que todos los puntos interiores serán empleados como puntos de colocación. Aún cuando el criterio para definir los puntos ya lo hemos establecido, la forma de implementarlo en un programa puede diferir considerablemente de una persona a otra. Es decir, dos implementaciones distintas pueden conducir exactamente al mismo conjunto de puntos, aunque su estructura y programación sean diferentes.

Vista Previa 2026

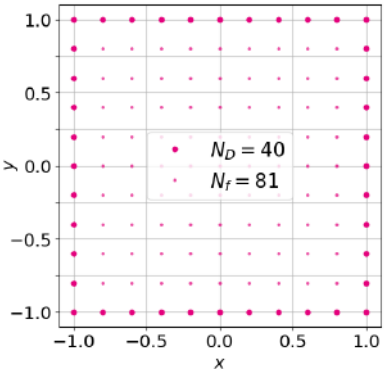


Figura 5.43 Puntos de entrenamiento y colocación para el problema de Laplace usando $N = 10$ divisiones en cada eje coordenado.

El Código 5.4 muestra una estrategia automática para clasificar los nodos de la malla en puntos de frontera y puntos interiores. Para ello, se evalúa si cada punto del arreglo XY satisface alguna de las ecuaciones que definen la frontera del dominio, esto es, $x = x_I$, $x = x_F$, $y = y_I$, $y = y_F$. Con el fin de evitar errores asociados a la representación numérica en punto flotante, esta verificación se realiza mediante una tolerancia `tol`. El resultado es una máscara lógica (`mask_bc`) que permite extraer de forma directa los puntos de frontera, almacenados en `XY_bc_np`, y los puntos interiores, almacenados en `XY_f_np`. Estos últimos se utilizarán posteriormente como puntos de colocación.

```
# ============================================================
# 1.B Separar automaticamente frontera e interior
# ============================================================
tol = 1e-12
mask_bc = (
    (np.abs(XY[:,0] - xI) < tol) |
    (np.abs(XY[:,0] - xF) < tol) |
    (np.abs(XY[:,1] - yI) < tol) |
    (np.abs(XY[:,1] - yF) < tol)
)
XY_bc_np = XY[mask_bc]      # puntos de frontera
XY_f_np  = XY[~mask_bc]     # puntos interiores (colocacion)

# ============================================================
# 1.C Convertir a tensores
# ============================================================
XY_bc = torch.tensor(XY_bc_np, dtype=torch.float32)
XY_f  = torch.tensor(XY_f_np,  dtype=torch.float32)

# ============================================================
# 1.D Solucion exacta (para imponer Dirichlet y verificar)
# ============================================================
def u_exact(x, y):
    return torch.exp(x) * torch.cos(y)

x_bc = XY_bc[:, 0:1]
y_bc = XY_bc[:, 1:2]
u_bc = u_exact(x_bc, y_bc)
```

Código 5.4 Selección de los puntos de entrenamiento y colocación.

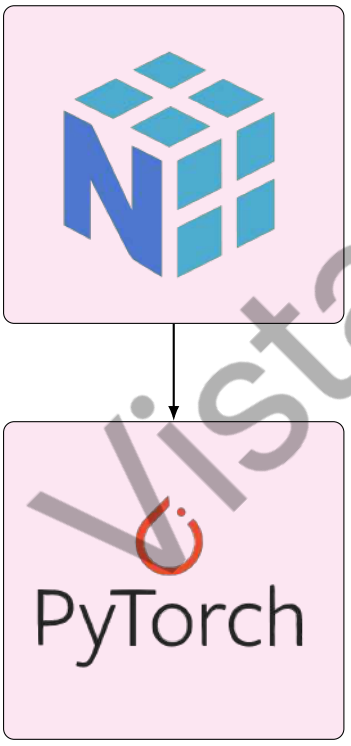


Figura 5.44 Los conjuntos de puntos en formato *NumPy* se deben transformar en tensores de *PyTorch*.

Para completar la preparación de los datos necesarios para el entrenamiento de la red neuronal, en la parte `1.C`, los conjuntos de puntos previamente obtenidos en formato *NumPy* se transforman en tensores de *PyTorch*. Este paso resulta indispensable, ya que el entrenamiento del modelo y el cálculo de derivadas mediante diferenciación automática se realizan directamente sobre este tipo de estructuras.

Posteriormente, en la parte `1.D`, se define la solución analítica mediante la función `u_exact(x,y)`. Como se ha señalado, esta solución no se utiliza para entrenar la red en los puntos interiores del dominio, sino únicamente para imponer las condiciones de frontera de Dirichlet y para verificar la aproximación obtenida. En específico, a partir de los puntos almacenados en `XY_bc`, se extraen sus coordenadas x y y, y se evalúa la solución exacta en dichos puntos y se almacena en el tensor `u_bc`.

Vista previa 2026



Figura 5.45 El código conserva la lógica del caso 1D, que tengamos dos entradas es lo que permite extender al caso 2D.

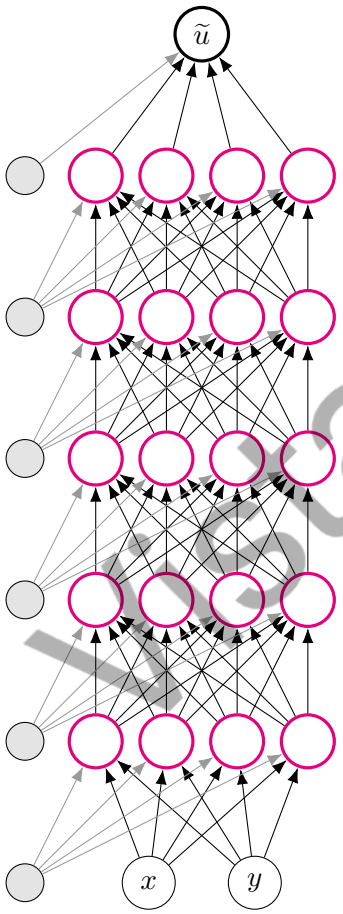


Figura 5.46 Red neuronal con 5 capas ocultas de 4 neuronas cada empleada en el programada.

5.5.2 Código para definir la red neuronal

El Código 5.5 presenta la definición de la red neuronal que se utilizará para aproximar la solución del problema bidimensional. A primera vista, esta arquitectura es muy similar a la empleada en el caso unidimensional, ya que sigue consistiendo en una red neuronal multicapa totalmente conectada, con cinco capas ocultas, una función de activación `tanh` y una única salida escalar.

Sin embargo, la diferencia esencial que indica que ahora se está resolviendo un problema en dos dimensiones se encuentra en la capa de entrada. En particular, la instrucción `nn.Linear(2, Ne)` muestra que la red recibe dos variables de entrada por cada punto de evaluación, correspondientes a las coordenadas espaciales (x, y). Esto contrasta con el caso unidimensional, en el cual la red solo recibe una única coordenada x.

A partir de estas dos coordenadas, la información se propaga a través de las capas ocultas, cada una compuesta por $N_e = 4$ neuronas y activada mediante la función `tanh`. Finalmente, la capa de salida devuelve un único valor escalar, que corresponde a la aproximación de la solución en el punto (x, y).

```
# ============================================================
# 2. Red neuronal
# ============================================================
Ne = 4
class MultiCapa2D(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear( 2, Ne)
        self.hidden2 = nn.Linear(Ne, Ne)
        self.hidden3 = nn.Linear(Ne, Ne)
        self.hidden4 = nn.Linear(Ne, Ne)
        self.hidden5 = nn.Linear(Ne, Ne)
        self.output  = nn.Linear(Ne,  1)
    def forward(self, x):
        x = torch.tanh(self.hidden1(x))
        x = torch.tanh(self.hidden2(x))
        x = torch.tanh(self.hidden3(x))
        x = torch.tanh(self.hidden4(x))
        x = torch.tanh(self.hidden5(x))
        return self.output(x)
model = MultiCapa2D()
```

Código 5.5 Definición de la red neuronal.

5.5.3 Código para obtener el residuo

Recordemos que, para este problema, el residuo asociado a la ecuación diferencial está dado por $f := u_{xx} + u_{yy}$. Por lo tanto, para evaluar qué tan bien la aproximación neuronal satisface la ecuación en un conjunto de puntos interiores, es necesario calcular las derivadas parciales de segundo orden de la salida de la red con respecto a las variables espaciales x y y.

Vista previa 2026

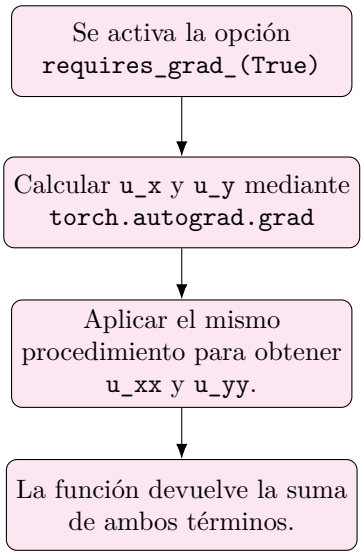


Figura 5.47 Esquema general para obtener el residuo.

El Código 5.6 implementa el cálculo del residuo de la ecuación de Laplace utilizando diferenciación automática en *PyTorch*. Esta es una de las partes más importantes del programa, ya que permite incorporar directamente la EDP dentro del proceso de entrenamiento.

```python
# ============================================================
# 3. Residual de Laplace usando diferenciacion automatica
# ============================================================
def pde(XY, model):
    XY = XY.clone().detach().requires_grad_(True)
    u_pred = model(XY)

    # Primeras derivadas
    grads = torch.autograd.grad(
        u_pred, XY,
        grad_outputs=torch.ones_like(u_pred),
        create_graph=True
    )[0]
    u_x = grads[:, 0:1]
    u_y = grads[:, 1:2]

    # Segundas derivadas
    grads_x = torch.autograd.grad(
        u_x, XY,
        grad_outputs=torch.ones_like(u_x),
        create_graph=True
    )[0]
    grads_y = torch.autograd.grad(
        u_y, XY,
        grad_outputs=torch.ones_like(u_y),
        create_graph=True
    )[0]
    u_xx = grads_x[:, 0:1]
    u_yy = grads_y[:, 1:2]

    return u_xx + u_yy
```

Código 5.6 Implementación del residual de la EDP.

La función `pde(XY, model)` recibe como entrada un conjunto de puntos del dominio, almacenados en `XY`, así como el modelo neuronal ya definido. Entonces el residuo se calcula siguiendo los siguientes pasos:

- En primer lugar, se activa la opción `requires_grad_(True)` para que *PyTorch* pueda calcular derivadas automáticamente. Luego, se evalúa la red neuronal en los puntos de entrada, obteniendo `u_pred`, que representa la aproximación de la solución en cada par ordenado (x, y).
- Luego, con `torch.autograd.grad`, se calculan las derivadas $\widetilde{u}_x$ y $\widetilde{u}_y$, almacenadas en las columnas de `grads`.
- Posteriormente, el mismo procedimiento se aplica a `u_x` y `u_y` para obtener las derivadas de segundo orden, en particular `u_xx` y `u_yy`.
- Finalmente, la función devuelve la suma $\widetilde{u}_{xx} + \widetilde{u}_{yy}$, la cual corresponde precisamente al operador de Laplace aplicado a la aproximación neuronal. Este valor constituye el residuo que será utilizado posteriormente en la función de costo para forzar a la red neuronal a satisfacer la ecuación diferencial en los puntos de colocación.

Notas

La segunda derivada se obtiene aplicando la primera derivada dos veces, es decir:

$$u_{xx} = (u_x)_x,$$

$$u_{yy} = (u_y)_y.$$

Figura 5.48 Cálculo de las derivadas de segundo orden.

Vista previa 2026



Figura 5.49 La programación de la función de pérdida resulta muy directa.

5.5.4 Código para definir la función de pérdida

El Código 5.7 define la función de pérdida que será utilizada durante el entrenamiento de la red neuronal. Esta función desempeña un papel central en el método, ya que establece el criterio cuantitativo que permitirá evaluar qué tan buena es la aproximación generada por la red en cada iteración del proceso de optimización.

```
# ============================================================
# 4. Funcion de perdida
# ============================================================
def loss_fn(model, XY_f, XY_bc, u_bc):
    # Residual en puntos de colocacion
    res = pde(XY_f, model)
    loss_pde = torch.mean(res**2)

    # Condicion de Dirichlet en frontera
    u_bc_pred = model(XY_bc)
    loss_bc = torch.mean((u_bc - u_bc_pred)**2)

    return loss_pde + loss_bc
```

Código 5.7 Implementación de la función de pérdida.

Como estudiamos anteriormente, la función de pérdida se construye de manera que la aproximación neuronal respete la EDP y las condiciones de frontera del problema. En este caso particular, la pérdida está compuesta por dos términos principales.

- El primero de ellos corresponde al error asociado al residuo de la ecuación diferencial en los puntos de colocación. Para ello, se evalúa la función `pde(XY_f, model)`, la cual devuelve el valor del residuo en los puntos interiores del dominio. Posteriormente, se calcula `loss_pde` como el promedio del cuadrado del residuo.

- El segundo término de la pérdida está asociado a las condiciones de frontera de Dirichlet. Para ello, la red neuronal se evalúa en los puntos de frontera `XY_bc`, obteniendo la aproximación `u_bc_pred`. A continuación, esta se compara con los valores exactos `u_bc`, calculando el error cuadrático medio en `loss_bc`.

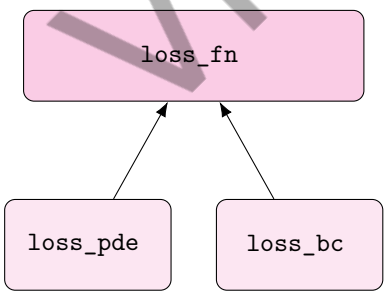


Figura 5.50 Esquema general del programa para la función de pérdida.

Finalmente, la función de pérdida total se define como la suma de ambos aportes lo que implica que, durante el entrenamiento, la red neuronal deberá encontrar un equilibrio entre satisfacer la ecuación diferencial en el interior y cumplir las condiciones de frontera en los bordes del dominio.

5.5.5 Código para el entrenamiento

El Código 5.8 corresponde al proceso de entrenamiento de la red neuronal. En esta etapa, los parámetros del modelo (pesos y sesgos) se ajustan iterativamente con el objetivo de minimizar la función de pérdida previamente definida.

Vista previa 2026

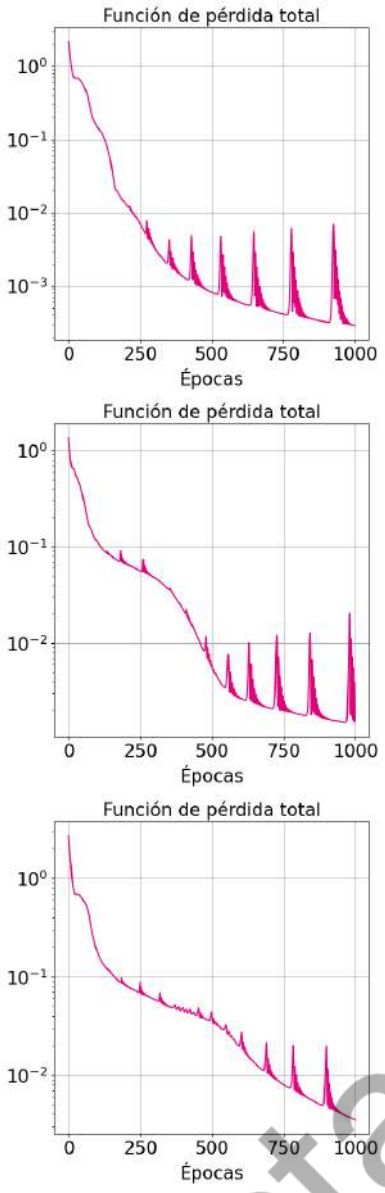


Figura 5.51 Ejemplos del comportamiento de la función de pérdida durante las primeras 1000 épocas para este ejemplo.

Notas

Optimizador: `Adam`
Tasa η: `0.01`,
Épocas: `10000`

Figura 5.52 Hiperparámetros principales para el entrenamiento.

```python
# ============================================================
# 5. Optimizador y entrenamiento
# ============================================================
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

epochs = 10000
loss_history = []

for epoch in range(epochs):
    optimizer.zero_grad()
    loss = loss_fn(model, XY_f, XY_bc, u_bc)
    loss.backward()
    optimizer.step()

    loss_history.append(loss.item())

    if epoch % 500 == 0 or epoch == epochs - 1:
        print(f"Epoch {epoch}: loss = {loss.item():.6e}")
```

Código 5.8 Implementación en *Python* para el ejemplo con tanh.

Primero, se define el optimizador mediante el algoritmo *Adam*, aplicado a los parámetros entrenables del modelo con una tasa de aprendizaje de `0.01`. Este optimizador será el encargado de actualizar los pesos y sesgos de la red en cada iteración del entrenamiento con el propósito de minimizar la función de pérdida.

El entrenamiento se lleva a cabo durante un total de `10000` épocas. En cada iteración del ciclo principal tenemos que:

- Primero se eliminan los gradientes acumulados mediante la instrucción `optimizer.zero_grad()`. Este paso es necesario porque, en *PyTorch*, los gradientes se acumulan por defecto en cada pasada, por lo que deben reiniciarse antes de calcular una nueva actualización.
- Posteriormente, se evalúa la función de pérdida a través de `loss = loss_fn()`, donde se consideran simultáneamente los puntos interiores `XY_f`, los puntos de frontera `XY_bc` y los valores exactos de la solución sobre la frontera `u_bc`.
- A continuación, la instrucción `loss.backward()` calcula automáticamente el gradiente de la función de pérdida con respecto a todos los parámetros entrenables de la red. Una vez obtenidos dichos gradientes, la instrucción `optimizer.step()` actualiza los parámetros del modelo en la dirección que reduce la pérdida.
- Además, en cada época se almacena el valor de la pérdida en la lista `loss_history`, lo que permite posteriormente analizar la evolución del entrenamiento y verificar si el proceso de optimización converge adecuadamente.
- Finalmente, cada 500 épocas (y también en la última iteración) se imprime en pantalla el valor actual de la pérdida, con el propósito de monitorear el progreso del entrenamiento.

Vista previa 2026

Norma infinito discreta.

Sea $u(x, y)$ la solución exacta del problema y $\tilde{u}(x, y)$ la aproximación obtenida por la red neuronal. Si definimos el error puntual como

$$e(x, y) = u(x, y) - \tilde{u}(x, y).$$

Entonces, sobre la malla de validación compuesta por los puntos $\{\boldsymbol{x}_{ij}\}$, donde $\boldsymbol{x}_{ij} = (x_i, y_j)$, pueden definirse la **norma infinito discreta** del error como

$$\|e\|_\infty = \max_{i,j} |u(\boldsymbol{x}_{ij}) - \tilde{u}(\boldsymbol{x}_{ij})|,$$

Esta norma mide la mayor discrepancia puntual entre la solución exacta y la solución aproximada en toda la malla de validación.

Figura 5.53 Definición de la norma infinito (norma máxima).

La norma L^2 discreta.

Por otro lado, si definimos el error absoluto en cada punto $\boldsymbol{x}_{ij}$ como

$$e_{ij} = |u(\boldsymbol{x}_{ij}) - \tilde{u}(\boldsymbol{x}_{ij})|,$$

entonces, la **norma L^2 discreta** del error puede definirse como

$$\|e\|_2 = \left(\frac{1}{N_p} \sum_{i,j} e_{ij}^2 \right)^{1/2}$$

donde N_p representa el número total de puntos de la malla de validación.

Esta norma proporciona una medida global del error, promediando la discrepancia cuadrática en todo el dominio.

Figura 5.54 Definición de la norma L^2 discreta.

5.5.6 Código para la validación del modelo

Para llevar a cabo esta validación, primero se construye una nueva malla uniforme sobre el dominio cuadrado, independiente de la utilizada durante el entrenamiento. El Código 5.9 corresponde a esta etapa de validación del modelo entrenado. Como en este ejemplo se dispone de la solución analítica, es posible evaluar de manera directa la precisión de la aproximación obtenida por la red neuronal.

```
# ============================================================
# 6. Evaluacion sobre una nueva la malla
# ============================================================
Nt = 30
x_test = np.linspace(xI, xF, Nt+1)
y_test = np.linspace(yI, yF, Nt+1)
X_test, Y_test = np.meshgrid(x_test, y_test)
XY_test_np = np.hstack([X_test.reshape(-1, 1), Y_test.reshape
    (-1, 1)])
XY_test = torch.tensor(XY_test_np, dtype=torch.float32)

with torch.no_grad():
    u_pred = model(XY_test).cpu().numpy().reshape(Nt+1, Nt+1)

X_t = torch.tensor(X_test, dtype=torch.float32)
Y_t = torch.tensor(Y_test, dtype=torch.float32)
u_ex = u_exact(X_t, Y_t).cpu().numpy()

# Error y normas
error = u_ex - u_pred
error_abs = np.abs(error)
error_inf = np.max(error_abs)            # Norma infinito
error_l2  = np.sqrt(np.mean(error**2))   # Norma L2 discreta
print("Norma infinito (max) =", error_inf)
print("Norma L2 discreta    =", error_l2)
```

Código 5.9 Validación del modelo

En este caso, cada eje coordenado se divide en `Np=30` intervalos generando un nuevo conjunto de puntos. Una vez definida esta nueva malla, sus coordenadas se reorganizan en el arreglo `XY_test_np` y posteriormente se convierten a un tensor de *PyTorch* mediante `XY_test`. Este tensor se utiliza como entrada del modelo entrenado para obtener la aproximación en los puntos de la malla de validación.

La evaluación de la red se realiza con `with torch.no_grad():`, lo cual indica que en esta etapa ya no es necesario calcular gradientes ni construir grafos computacionales lo que reduce el costo computacional. El resultado se almacena en `u_pred`, reorganizado posteriormente con la misma estructura bidimensional de la malla.

Posteriormente, la solución analítica se evalúa en los mismos puntos de la malla de validación mediante la función `u_exact`. Esto permite construir el arreglo `u_ex`, que contiene los valores exactos de referencia contra los cuales se comparará la aproximación neuronal. Finalmente, se calcula el error absoluto punto a punto y se reportan dos métricas de error: la norma L_∞ y la norma L_2.

Vista previa 2026

Notas

N = 10,
Nd = 40, (4N),
Nf = 81, $(N-1)^2$,
Ne = 5,
Nt = 30,
Np = 961. $(Nt+1)^2$

Figura 5.55 Parámetros principales del programa con una malla de $N+1$ puntos por lado.

5.5.7 Gráfica de los resultados

El programa puede complementarse con diversas representaciones gráficas que permiten visualizar tanto la solución analítica, solución aproximada como el error cometido por la red neuronal. Aunque en esta etapa ya es posible generar una variedad más amplia de análisis, por el momento bastará con considerar únicamente las gráficas de la solución y del error absoluto. Un estudio más detallado de estos resultados se presentará en la siguiente sección.

```
# ============================================================
# 7. Graficas
# ============================================================
plt.figure(figsize=(6,5))
plt.contourf(X_t, Y_t, u_pred, levels=40)
plt.colorbar()
plt.title("Solucion de la red neuronal")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

plt.figure(figsize=(6,5))
plt.contourf(X_t, Y_t, error_abs, levels=40)
plt.colorbar()
plt.title("Error absoluto")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Código 5.10 Visualización de resultados.

Cabe señalar que la construcción de representaciones gráficas admite distintas alternativas de implementación y estilo, dependiendo tanto de los objetivos del análisis como de las preferencias de quien programa. En este sentido, en el Código 5.9 se presentan dos visualizaciones básicas que resultan suficientes para una primera inspección del comportamiento de la solución aproximada.



Figura 5.56 La construcción de representaciones gráficas admite distintas alternativas de implementación y estilo.

5.6 RESULTADOS EN DOMINIOS RECTANGULARES

Dada la flexibilidad de la metodología con redes neuronales, ya con la herramienta planteada, es posible considerar numerosos casos de interés para un estudio más profundo. Sin embargo, como se indicó desde el inicio, el objetivo principal es comprender de manera clara el funcionamiento de esta herramienta y, a partir de ello, despertar la curiosidad y ampliar el panorama para que cada lector pueda abordar sus propios problemas de interés, o incluso otros más complejos que los aquí presentados. En esta sección presentaremos los resultados numéricos correspondientes a varios ejemplos, todos ellos planteados en un dominio rectangular:

- Comenzaremos con el problema de Laplace planteado en el programa de la sección anterior, cuya solución analítica es conocida. En este caso el problema se plantea usando mallas uniformes. Aquí, evaluaremos de manera exhaustiva la capacidad del método en

Vista previa 2026



Figura 5.57 A diferencia de los problemas abordados anteriormente mediante PINNs, en esta sección se evaluará la capacidad de las redes neuronales para aproximar soluciones de EDPs de mayor complejidad.

términos de sus parámetros principales, tales como el número y la distribución de los datos conocidos (puntos de entrenamiento y de colocación). Además, presentaremos un análisis del optimizador considerando el número de épocas y la tasa de aprendizaje. Finalmente, compararemos nuestra metodología con el método clásico de diferencias finitas presentado al inicio de este capítulo.

- Como segundo ejemplo, se abordará nuevamente el problema de Laplace, pero considerando esta vez una distribución aleatoria de los puntos de colocación, en lugar de una malla uniforme. Este cambio permitirá analizar la influencia de la distribución de los puntos en la precisión de las aproximaciones obtenidas. En particular, se estudiará el efecto del número de datos conocidos sobre la precisión del método, prestando especial atención a la arquitectura de la red neuronal y su rendimiento.

- En el tercer ejemplo, se abordará nuevamente la ecuación de Laplace, pero considerando condiciones de frontera mixtas que incorporan condiciones de tipo Neumann. Se describirán las modificaciones necesarias en la implementación del código para incorporar estas condiciones y se analizará su influencia en la precisión de las aproximaciones obtenidas mediante PINNs.

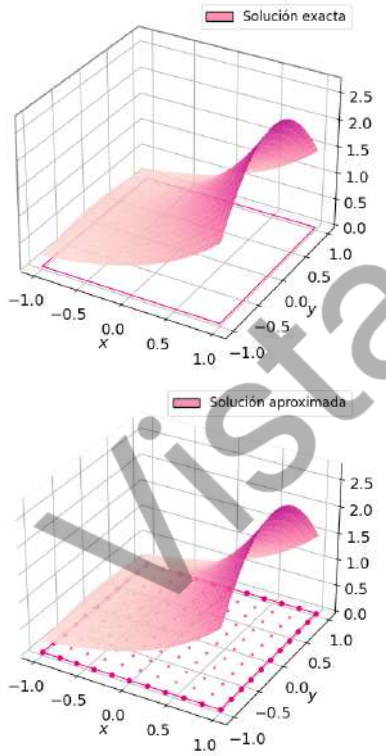


Figura 5.58 Ejemplo 1: solución analítica y aproximada usando $N_D = 40$ y $N_f = 81$ con una malla uniforme.

En lugar de continuar con el estudio de ecuaciones diferenciales parciales de mayor complejidad, en la siguiente sección se abordarán problemas definidos sobre dominios irregulares. Este cambio de enfoque permitirá explorar la capacidad de las PINNs para aproximar soluciones en geometrías más complejas. Sin embargo, con el propósito de ampliar el alcance de los ejemplos estudiados, en la sección de ejercicios se proponen diversos problemas que involucran otras ecuaciones diferenciales parciales. Estos permitirán al lector aplicar los conceptos y procedimientos desarrollados hasta el momento, así como explorar el comportamiento de las PINNs ante diferentes operadores diferenciales y condiciones de frontera.

5.6.1 Ejemplo 1: ecuación de Laplace con mallas uniformes

Para nuestro primer ejemplo, analizaremos las simulaciones obtenidas mediante redes neuronales utilizando el programa descrito detalladamente en la sección anterior.

En este ejemplo estudiaremos el problema de Laplace (5.86) con condiciones de Dirichlet, planteado en la sección anterior, cuya solución analítica es conocida y está dada por (5.87). Por otra parte, los datos conocidos de la red neuronal serán proporcionados en una malla uniforme.

Vista previa 2026

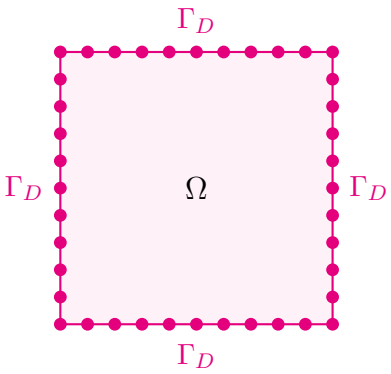


Figura 5.59 Dominio con condiciones de frontera Dirichlet en toda la frontera.

Los resultados numéricos

Para nuestras primeras simulaciones, la solución aproximada se calcula en el dominio $\Omega = [-1, 1] \times [-1, 1]$ usando una malla uniforme de $N+1 \times N+1$ utilizando una malla uniforme de $(N+1) \times (N+1)$ puntos con una subdivisión de $N = 10$ en cada dirección espacial. Los resultados obtenidos mediante la red neuronal se muestran en las Figuras 5.58 y 5.61.

Como puede observarse, incluso utilizando un número relativamente reducido de puntos de entrenamiento y colocación ($N_D = 40$ y $N_f = 81$), la red neuronal es capaz de aproximar con buena precisión la solución del problema. Asimismo, el comportamiento del método resulta estable y robusto frente a esta discretización relativamente simple, característica que analizaremos con mayor detalle en las siguientes simulaciones.

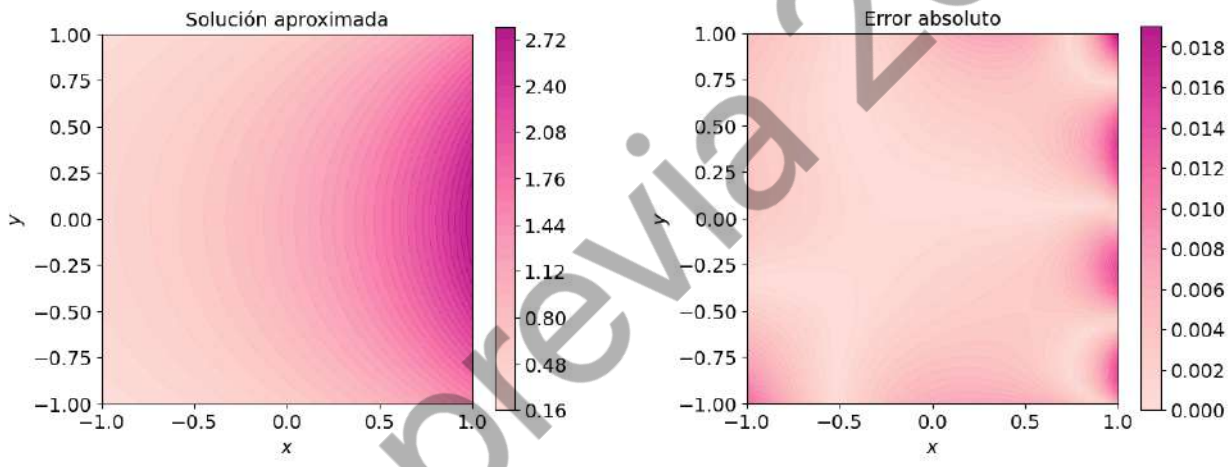


Figura 5.60 Ejemplo1: solución aproximada y error absoluto de la red neuronal para $N = 10$.

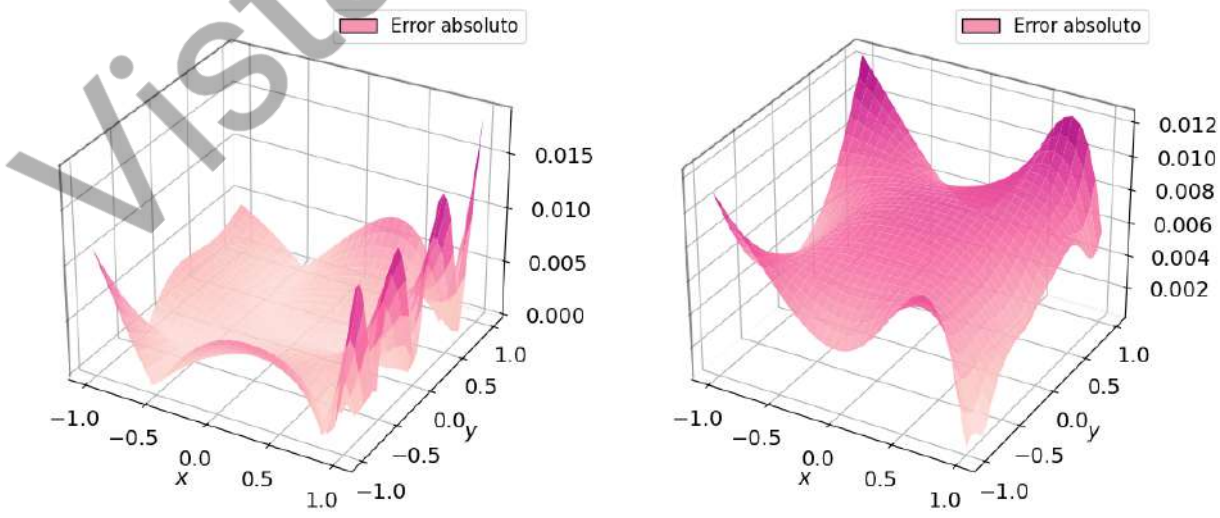


Figura 5.61 Ejemplo 1: error absoluto de la red neuronal usando $N = 10$ para dos simulaciones independientes.

Vista previa 2026

Por otro lado, la Figura 5.62 muestra el comportamiento de la función de pérdida en términos del número de épocas de entrenamiento. En esta gráfica se presentan tanto la pérdida total como las contribuciones individuales asociadas a los puntos de colocación y a los puntos de entrenamiento sobre la frontera.

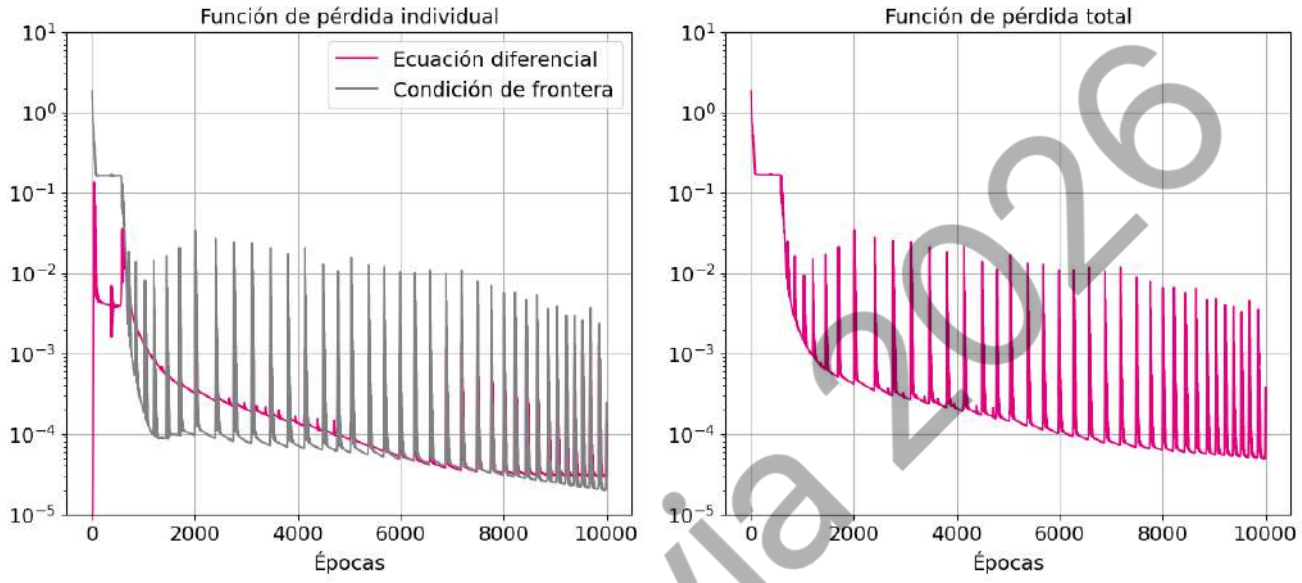


Figura 5.62 Comportamiento de la función de pérdida separando el error proporcionado por los puntos de colocación y los de entrenamiento de la red neuronal usando $N = 20$.

Vista previa 2026

```
Epoch   0: loss = 9.44e-01,
           pde  = 2.50e-05
           bdy  = 9.44e-01

Epoch  500:loss = 9.00e-04
           pde  = 3.79e-04
           bdy  = 5.21e-04

Epoch 1000:loss = 4.52e-04
           pde  = 1.48e-04
           bdy  = 3.03e-04

Epoch 1500:loss = 2.22e-04
           pde  = 7.68e-05
           bdy  = 1.45e-04

Epoch 2000:loss = 3.17e-04
           pde  = 6.67e-05
           bdy  = 2.50e-04
```

Figura 5.63 Ejemplo 1: comportamiento de la función de pérdida en las primeras épocas del entrenamiento.

Puede observarse que, para este ejemplo, ambos términos contribuyen de manera importante al error total, lo cual indica que la red neuronal debe aprender simultáneamente el comportamiento de la solución en el interior del dominio y el cumplimiento de las condiciones de frontera. Asimismo, se aprecia que alrededor de las 2000 épocas la función de pérdida comienza a estabilizarse, sugiriendo que el proceso de entrenamiento ha alcanzado un estado cercano a la convergencia.

Análisis del número de datos conocidos

Tal como analizamos en el caso unidimensional, en la resolución de ecuaciones diferenciales no basta con que la red neuronal proporcione una solución visualmente parecida a la solución analítica; también es importante cuantificar qué tan buena es dicha aproximación y, de ser posible, compararla con la obtenida mediante otras metodologías numéricas.

Así, en este primer caso, evaluaremos de manera exhaustiva la capacidad del método en términos de sus parámetros principales, tales como el número y la distribución de los datos conocidos (puntos de entrenamiento y de colocación), así como la arquitectura de la red neuronal en función del número de capas y neuronas.

Notas

La arquitectura de la red neuronal consiste de 5 capas ocultas de 4 neuronas cada una.

Figura 5.64 Arquitectura de la red neuronal.

El hecho de contar con una malla uniforme nos permite mantener una estructura fija en el número de puntos de colocación y de entrenamiento mediante el parámetro N asociado a la malla. En nuestro análisis, consideraremos todos los puntos interiores como puntos de colocación y todos los puntos de la frontera como puntos de entrenamiento.

La Figura 5.65 muestra la solución para los casos $N = 5$, 10 y 20. Puede observarse que, en las tres simulaciones, la red neuronal produce soluciones estables y con buena precisión. Además, conforme se incrementa la cantidad total de datos empleados, la magnitud del error absoluto tiende a reducirse, mejorando así la calidad de la aproximación obtenida.

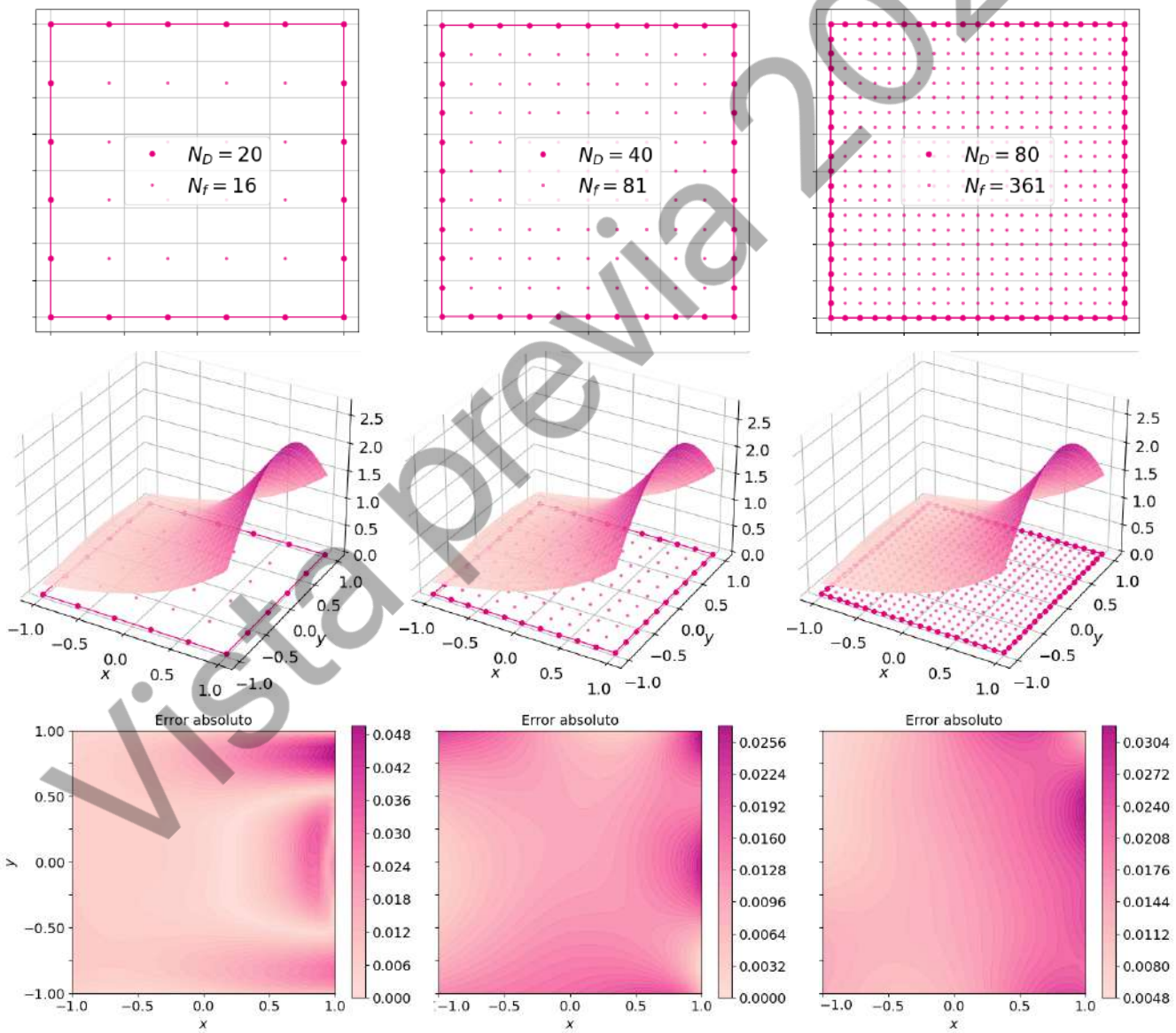


Figura 5.65 Ejemplo 1: muestra de puntos, solución aproximada y error absoluto de la red neuronal multicapa para distintos número de puntos de entrenamiento y colocación elegidos aleatoriamente.

Vista previa 2026



Figura 5.66 Cada metodología tiene sus propias ventajas y desventajas.

N	Norma L_2	Orden
5	8.62×10^{-3}	—
10	2.21×10^{-3}	1.95
20	5.58×10^{-4}	1.99
40	1.39×10^{-4}	1.99
80	3.49×10^{-5}	1.99

N	Norma L_∞	Orden
5	7.80×10^{-3}	—
10	2.07×10^{-3}	1.98
20	5.25×10^{-4}	1.99
40	1.31×10^{-4}	1.99
80	3.29×10^{-5}	1.99

Figura 5.67 Análisis de convergencia usando el método de diferencias finitas para el Ejemplo 1.

También puede notarse que las zonas donde aparecen los errores más grandes no corresponden necesariamente a los puntos donde la solución exacta toma sus valores máximos. En general, las discrepancias más importantes tienden a concentrarse en regiones próximas a la frontera del dominio. Este comportamiento es natural dentro del proceso de entrenamiento de las PINNs, ya que las condiciones de frontera se satisfacen de manera aproximada mediante la minimización de la función de pérdida. En consecuencia, el término asociado a dichas condiciones suele permanecer pequeño pero distinto de cero.

La Tabla 5.5 muestra las normas L_2 y L_∞ para diferentes combinaciones de puntos de entrenamiento y de colocación. En términos generales, la precisión de la aproximación mejora conforme aumenta el número de puntos de entrenamiento N_D. Sin embargo, también puede observarse que, para cantidades grandes de N, la reducción del error comienza a estabilizarse en valores cercanos a 10^{-3}.

Tabla 5.5 Ejemplo 1: norma L_2 y norma L_∞ para diferentes cantidades de datos de entrenamiento y puntos de colocación.

N	N_D	N_f	Norma L_2	Norma L_∞
5	20	16	1.32×10^{-2}	4.92×10^{-2}
10	40	81	2.80×10^{-3}	1.17×10^{-2}
20	80	361	2.52×10^{-3}	1.03×10^{-2}
40	160	1521	1.03×10^{-3}	1.06×10^{-2}
80	320	6241	2.82×10^{-3}	1.59×10^{-3}
160	640	25281	3.01×10^{-3}	1.39×10^{-3}

Si comparamos estos resultados con los obtenidos mediante el método de diferencias finitas utilizando una cantidad similar de datos, puede apreciarse claramente la ventaja en precisión que posee el método numérico clásico sobre la aproximación basada en redes neuronales, como se muestra en las Tablas 5.67. En particular, los errores obtenidos mediante PINNs todavía no alcanzan la precisión correspondiente al caso $N = 20$ del método de diferencias finitas.

Sin embargo, una de las principales ventajas de las PINNs radica en su gran flexibilidad. A diferencia de los métodos tradicionales, las redes neuronales no requieren una estructura fija de malla para funcionar correctamente. Tanto los puntos de entrenamiento asociados a la frontera como los puntos de colocación en el interior del dominio pueden elegirse de manera arbitraria o incluso aleatoria, manteniendo esencialmente la misma estructura del programa. Esta característica resulta especialmente valiosa en problemas donde la geometría del dominio es compleja o donde no se dispone de datos organizados sobre una malla estructurada.

Por esta razón, en el siguiente ejemplo abandonaremos la distribución uniforme de puntos y analizaremos el comportamiento de nuestra metodología utilizando conjuntos de puntos generados aleatoriamente dentro del dominio.

vista previa 2026

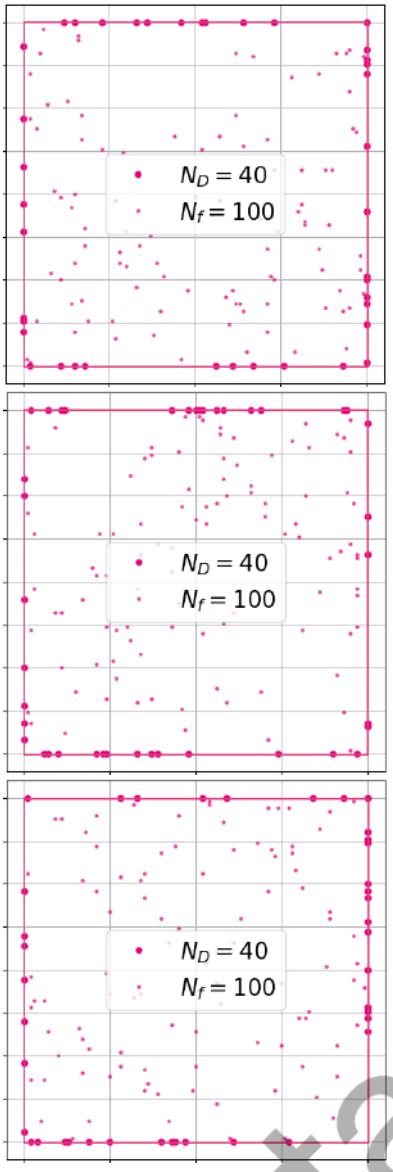


Figura 5.68 Tres ejemplos de los puntos de entrenamiento ($N_D = 40$) y puntos de colocación ($N_f = 100$) obtenidos aleatoriamente usando una distribución uniforme en base a una malla inicial de 100×100 puntos. Notar como en algunos casos inclusive tenemos ausencia de información en las esquinas.

5.6.2 Ejemplo 2: ecuación de Laplace con puntos aleatorios

Una de las grandes ventajas de las metodologías basadas en redes neuronales es la flexibilidad en la selección de puntos dentro del dominio de estudio. En particular, tanto los puntos de entrenamiento asociados a las condiciones de frontera como los puntos de colocación utilizados en el interior del dominio pueden elegirse de manera prácticamente arbitraria. Esta característica representa una diferencia importante con respecto a muchos métodos numéricos clásicos, los cuales suelen depender de mallas estructuradas o distribuciones rígidas de nodos.

Gracias a esta flexibilidad, es posible adaptar la distribución de puntos según las necesidades del problema, concentrando muestras en regiones donde la solución presenta cambios más pronunciados o utilizando estrategias de muestreo aleatorio para mejorar la cobertura del dominio.

En este ejemplo modificaremos nuestro programa reemplazando la malla uniforme utilizada previamente por una selección de puntos generados de manera aleatoria. Seguiremos con nuestras simulaciones considerando el mismo problema de Laplace (5.86) con condiciones de Dirichlet, planteado en el ejemplo anterior. De la misma manera, continuaremos utilizando la misma arquitectura de red neuronal multicapa planteada anteriormente.

La generación de puntos aleatorios

La generación de puntos aleatorios dentro de un dominio rectangular puede parecer, en principio, una tarea sencilla. No obstante, la calidad de las aproximaciones obtenidas mediante nuestra red neuronal depende en gran medida de la forma en que se realice dicho muestreo. En otras palabras, aunque dos conjuntos de puntos puedan contener el mismo número de muestras, la distribución espacial de estos puntos puede influir considerablemente en la precisión de la solución aproximada.

En este ejemplo analizaremos dos estrategias distintas para generar los puntos interiores del dominio.

- La primera consistirá en un muestreo completamente aleatorio, donde los puntos se seleccionan de manera independiente siguiendo una distribución uniforme sobre el dominio rectangular. Este enfoque es sencillo de implementar y computacionalmente económico; sin embargo, presenta algunas limitaciones importantes como se puede ver en la Figura 5.68. Debido al carácter puramente aleatorio de la muestra, pueden aparecer regiones con acumulación excesiva de puntos y otras con escasa representación, lo que ocasiona zonas del dominio pobremente exploradas por la red neuronal y, en consecuencia, una pérdida de precisión en la aproximación.

Vista previa 2026



Figura 5.69 Es momento de retomar algunos conceptos fundamentales de probabilidad y estadística relacionados con la construcción de muestras aleatorias y la manera en que estas pueden influir en la representación adecuada de un dominio.

El siguiente fragmento de código muestra las pequeñas modificaciones realizadas al programa presentado en la sección anterior con el objetivo de incorporar una muestra aleatoria de puntos dentro del dominio y sobre la frontera. La idea principal consiste en partir de los conjuntos completos de puntos interiores y de frontera previamente generados, para posteriormente seleccionar de manera aleatoria únicamente una fracción de ellos que será utilizada durante el entrenamiento de la red neuronal.

Obsérvese que el procedimiento se implementa de forma muy sencilla mediante el comando `np.random.choice`, el cual permite elegir índices aleatorios sin reemplazo a partir de los arreglos `XY_f_np` y `XY_bc_np`. De esta manera, cada punto seleccionado aparece una sola vez en la muestra final, evitando repeticiones innecesarias y produciendo una distribución aleatoria uniforme sobre el conjunto de datos disponible. Finalmente, los puntos seleccionados se convierten a tensores de `PyTorch`, ya que estos serán utilizados directamente durante el proceso de entrenamiento de la red neuronal.

```
# ============================================================
# 1.C Seleccionar puntos aleatorios
# ============================================================
N_f = 100    # Numero de puntos interiores
N_D = 40     # Numero de puntos de frontera

# Seleccion aleatoria SIN reemplazo
idx_f  = np.random.choice(len(XY_f_np),  N_f, replace=False)
idx_bc = np.random.choice(len(XY_bc_np), N_D, replace=False)

# Puntos finales
XY_bc_random = XY_bc_np[idx_bc]
XY_f_random  = XY_f_np[idx_f]

# ============================================================
# 1.D Convertir a tensores
# ============================================================
XY_bc = torch.tensor(XY_bc_random, dtype=torch.float32)
XY_f  = torch.tensor(XY_f_random,  dtype=torch.float32)
```

Código 5.11 Parte del programa que debe ser modificado para incluir una muestra aleatoria de distribución uniforme.

Como segunda alternativa, consideraremos un muestreo basado en la técnica del hipercubo latino (*Latin Hypercube Sampling*, LHS). La idea principal de esta estrategia es distribuir los puntos de manera más uniforme sobre el dominio, evitando que múltiples muestras queden concentradas en la misma región. En términos generales, el método divide cada dirección espacial en subintervalos y selecciona los puntos de forma que no compartan la misma posición relativa dentro de dichos intervalos. Esto permite obtener una cobertura más homogénea del dominio utilizando incluso una cantidad moderada de puntos, ver la Figura 5.70.

Vista previa 2026

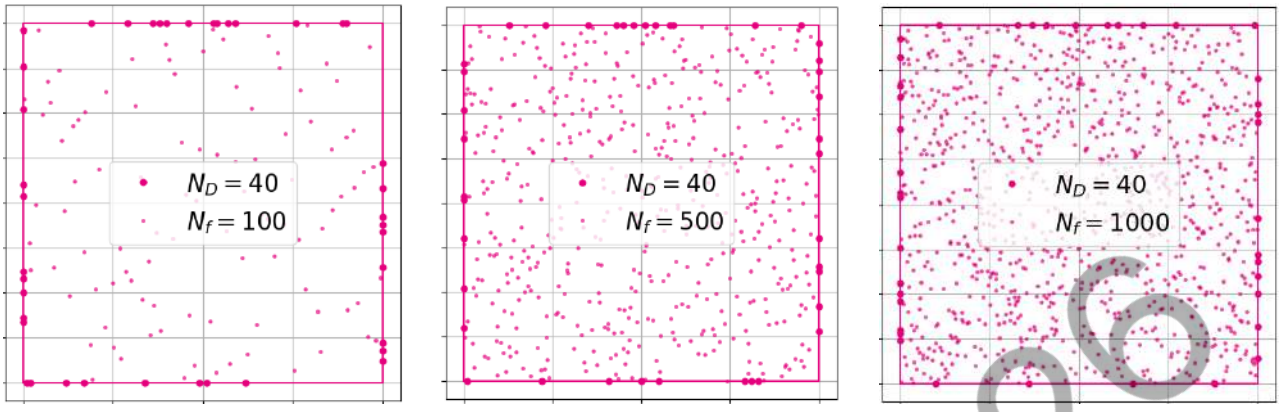


Figura 5.70 Ejemplos de los puntos de entrenamiento ($N_D = 40$) y puntos de colocación ($N_f = 100$, 500 y 1000) obtenidos usando un muestreo basado en la técnica del hipercubo latino.

El Código 5.12 muestra las modificaciones realizadas al programa presentado en la sección anterior con el objetivo de incorporar una muestra aleatoria de puntos interiores utilizando la técnica del hipercubo latino. Existen diversas maneras de implementar esta estrategia de muestreo; en nuestro caso utilizaremos la alternativa disponible en la librería `scipy`, la cual ofrece una implementación sencilla y eficiente.

A diferencia del enfoque presentado anteriormente, en este caso no partimos de un conjunto completo de puntos interiores previamente generado mediante una malla uniforme. En lugar de ello, los puntos son construidos directamente a partir de un modelo aleatorio definido inicialmente sobre el cuadrado unitario $[0,1] \times [0,1]$. Posteriormente, dichos puntos son escalados al tamaño del dominio rectangular deseado mediante una transformación lineal. Notar que desde un punto de vista práctico, mantendremos el dominio original completo, ya que la probabilidad de que un punto generado mediante el muestreo del hipercubo latino caiga exactamente sobre la frontera es despreciable.

Notas

Si se desea garantizar que todos los puntos generados pertenezcan estrictamente al interior del dominio, basta con reducir ligeramente los intervalos utilizados, evitando así que algunos puntos puedan coincidir exactamente con la frontera del rectángulo.

Figura 5.71 Puntos de colocación estrictamente al interior del dominio.

```
from scipy.stats import qmc
...

# ============================================================
# 1.C Seleccionar puntos aleatorios
# ============================================================
N_f = 100

# Muestra usando hipercubo latino
sampler = qmc.LatinHypercube(d=2)

# Puntos en [0,1]^2
sample = sampler.random(n=N_f)

# Escalar al dominio
XY_f_random = qmc.scale(sample, [xI, yI], [xF, yF])
```

Código 5.12 Fragmento del programa que debe ser incorporado para incluir una muestra aleatoria usando la técnica del hipercubo latino.

Vista previa 2026

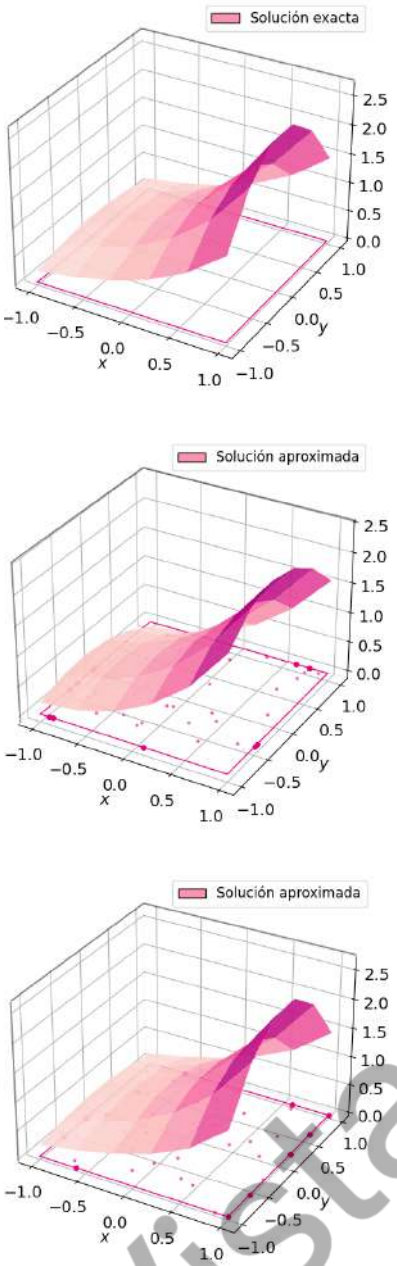


Figura 5.72 Ejemplo 2: solución analítica (arriba), solución predicha usando $N_D = 10$ y $N_f = 25$ con un muestreo uniforme (centro) y la técnica hipercubo latino (abajo). La solución se presenta utilizando la malla rectangular con 6×6 puntos de prueba.

Es importante mencionar que, en este ejemplo, la técnica del hipercubo latino es utilizada únicamente para generar los puntos de colocación en el interior del dominio. Por otro lado, los puntos asociados a las condiciones de frontera de tipo Dirichlet continuarán siendo tomados mediante una distribución uniforme sobre cada lado del dominio. No obstante, una posible extensión consiste en aplicar también un muestreo aleatorio independiente sobre cada segmento de la frontera, dividiendo el contorno en varias regiones y generando muestras aleatorias en cada una de ellas. Esta variante se deja como ejercicio para el lector.

Los resultados numéricos

La solución predicha se calcula en el dominio $\Omega = [-1, 1] \times [-1, 1]$. Inicialmente, se consideran $(N + 1)$ puntos equidistantes sobre cada lado de la frontera, utilizando distintos valores de subdivisión N. A partir de este conjunto de datos, el número total de puntos de entrenamiento asociados a las condiciones de frontera (N_D) se selecciona aleatoriamente. Por otro lado, el número de puntos de colocación (N_f) dentro del dominio se genera mediante dos estrategias distintas de muestreo: una distribución aleatoria uniforme y un muestreo basado en la técnica del Hipercubo Latino, la cual permite obtener una cobertura más homogénea del espacio interior del dominio.

La Figura 5.72 muestra tanto la solución exacta como la aproximada obtenidos mediante la red neuronal utilizando $N_D = 10$ puntos de frontera y $N_f = 25$ puntos de colocación con las dos técnicas de muestreo. Por otro lado la Figura 5.73 muestra el valor absoluto correspondiente a ambos casos.

- Notar que el caso aleatorio no logra reproducir completamente la solución analítica del problema.
- Sin embargo, a pesar del reducido número de datos utilizados con el hipercubo latino, puede observarse que la aproximación obtenida reproduce adecuadamente el comportamiento global de la solución exacta.

Este comportamiento resulta consistente incluso al repetir múltiples simulaciones independientes. En particular, el muestreo basado en hipercubo latino presenta una mayor estabilidad y capacidad de reproducibilidad de la solución aproximada, mientras que el muestreo completamente aleatorio suele producir resultados más irregulares debido a la posible concentración o ausencia de puntos en ciertas regiones del dominio.

Lo anterior pone de manifiesto la importancia que tiene la estrategia de muestreo dentro de los métodos basados en PINNs, ya que la distribución espacial de los puntos de entrenamiento y colocación puede influir significativamente tanto en la estabilidad del entrenamiento como en la calidad de la aproximación obtenida.

Vista previa 2026

Para complementar los resultados numéricos, presentamos en la Figura 5.74 el comportamiento de la función de pérdida utilizando la técnica de muestreo basada en el hipercubo latino para obtener los puntos de colocación. Como era de esperarse, puede observarse que la función de pérdida converge de manera adecuada utilizando un número relativamente reducido de épocas de entrenamiento, permitiendo obtener aproximaciones precisas con un costo computacional moderado.

A partir de este punto, y salvo que se indique lo contrario, todas las simulaciones que requieran un muestreo aleatorio emplearán esta estrategia debido a su mejor distribución espacial de puntos y a la estabilidad observada en los resultados obtenidos.

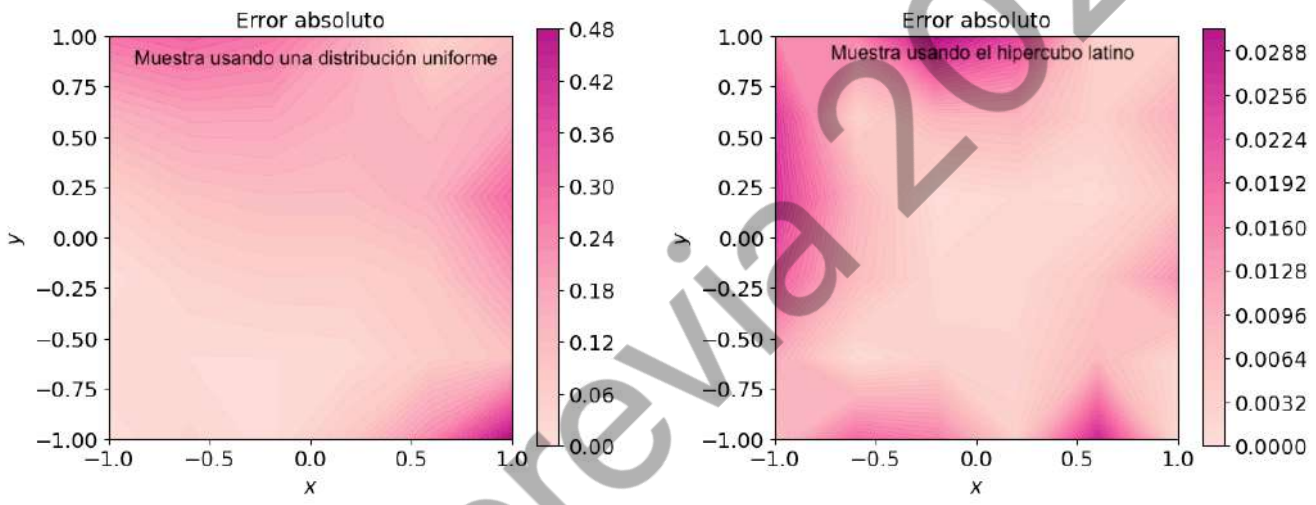


Figura 5.73 Ejemplo 2: Error absoluto usando $N_D = 10$ y $N_f = 25$ con un muestreo con distribución uniforme y la técnica hipercubo latino.

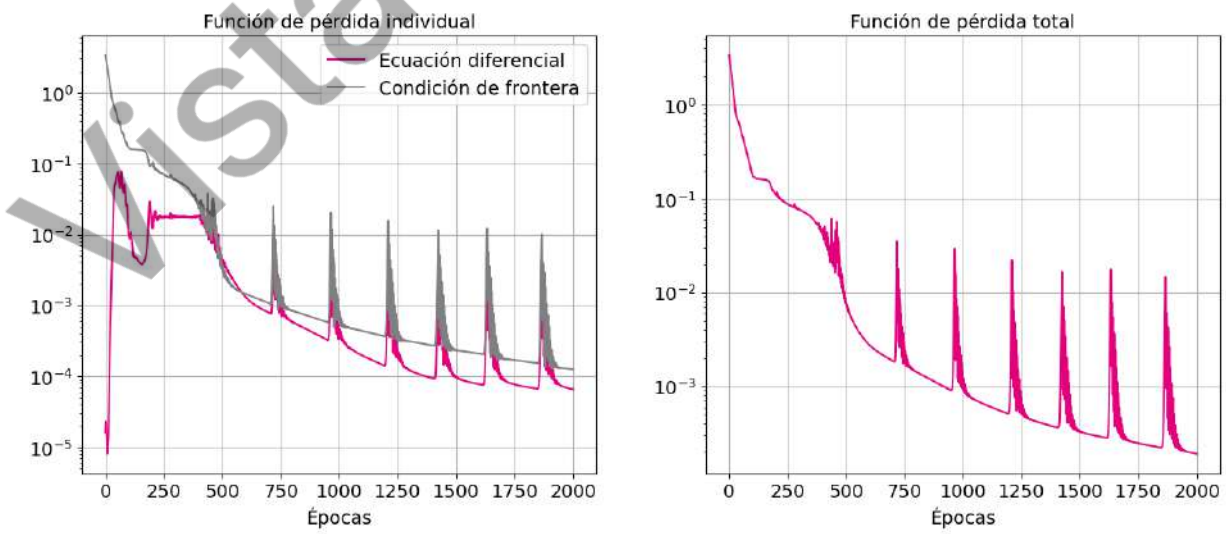


Figura 5.74 Ejemplo 2: Comportamiento de la función de pérdida usando $N_D = 10$ y $N_f = 25$ empleando la técnica hipercubo latino para obtener los puntos de colocación.

Vista previa 2026

```
Epoch   0: loss = 3.36,
           pde  = 1.61e-05
           bdy  = 3.36

Epoch  500:loss = 8.07e-03
           pde  = 4.30e-03
           bdy  = 3.76e-03

Epoch 1000:loss = 9.05e-04
           pde  = 3.57e-04
           bdy  = 5.48e-04

Epoch 1500:loss = 3.28e-04
           pde  = 9.70e-05
           bdy  = 2.31e-04

Epoch 2000:loss = 1.91e-04
           pde  = 6.60e-05
           bdy  = 1.25e-04
```

Figura 5.75 Comportamiento de la función de pérdida en las primeras épocas durante el proceso de entrenamiento.



Figura 5.76 Diferencias entre los resultados numéricos entre las técnicas de muestreo.

Por lo tanto, los resultados numéricos muestran que el método propuesto es capaz de aproximar con gran precisión la solución exacta incluso utilizando un número relativamente pequeño de puntos de entrenamiento y de colocación aleatorios. Este comportamiento es consistente con estudios previos, donde se ha observado que las redes neuronales informadas por la física proporcionan aproximaciones altamente precisas para soluciones suaves de ecuaciones diferenciales parciales [12].

Análisis del número de puntos de prueba

Es importante destacar que los resultados mostrados en las Figuras 5.72 y 5.73 fueron desplegados utilizando una malla rectangular compuesta por 36 nodos, correspondiente a una discretización de tipo 6×6. Esto equivale a considerar $N = 5$ subdivisiones en cada dirección espacial del dominio.

No obstante, una vez concluido el proceso de entrenamiento, la solución predicha por la red neuronal multicapa y las correspondientes medidas de error no deberían depender de la resolución de la malla utilizada para visualizar o evaluar la solución. En efecto, la aproximación obtenida mediante la red neuronal define una función continua $\widetilde{u}(x, y; W, \boldsymbol{b})$, la cual puede evaluarse en cualquier punto del dominio independientemente de la discretización empleada.

La Tabla 5.6 presenta el error en las normas L_2 y L_∞ calculado para la misma solución aproximada utilizando distintas resoluciones de malla N. Los resultados muestran que las diferencias entre los errores obtenidos con distintas discretizaciones son mínimas, lo cual confirma que la precisión de la solución aproximada depende principalmente del entrenamiento de la red neuronal y no de la malla empleada para su evaluación, tal como era de esperarse.

Tabla 5.6 Ejemplo 2: norma L_2 y norma L_∞ del error absoluto entre la solución predicha y la solución exacta para $N_u = 10$ y $N_f = 25$ variando la resolución de la malla usados comop puntos de prueba.

N	5	10	15	20
Norma L_2	2.89e-03	2.69e-03	2.60e-03	2.56e-03
Norma L_∞	1.20e-02	1.44e-02	1.41e-02	1.44e-02

N	25	30	35	40
Norma L_2	2.53×10^{-3}	2.51×10^{-3}	2.49×10^{-3}	2.48×10^{-3}
Norma L_∞	1.43×10^{-2}	1.44×10^{-2}	1.43×10^{-2}	1.44×10^{-2}

Análisis del número de datos conocidos

Ahora, el desempeño de la red neuronal multicapa se analiza considerando diferentes cantidades de puntos de entrenamiento y de colocación. En particular, dichos valores se eligen de acuerdo con las relaciones

Vista previa 2026

Notas

$N_D = 2N$
$N_f = N^2$
$N_p = (N+1)^2$

Figura 5.77 Parámetros.

$N_D = 2N$ y $N_f = N^2$, donde N_D representa el número de puntos asociados a las condiciones de frontera y N_f el número de puntos de colocación en el interior del dominio. Para este análisis usaremos una red neuronal multicapa bastante robusta compuesta por cinco capas ocultas de 20 neuronas cada una.

La Figura 5.78 presenta la distribución del error absoluto para los casos $N = 5$, 10 y 20. En todos los casos se obtienen aproximaciones estables y precisas. Como era de esperarse, el error absoluto disminuye conforme aumenta el número total de datos utilizados durante el entrenamiento.

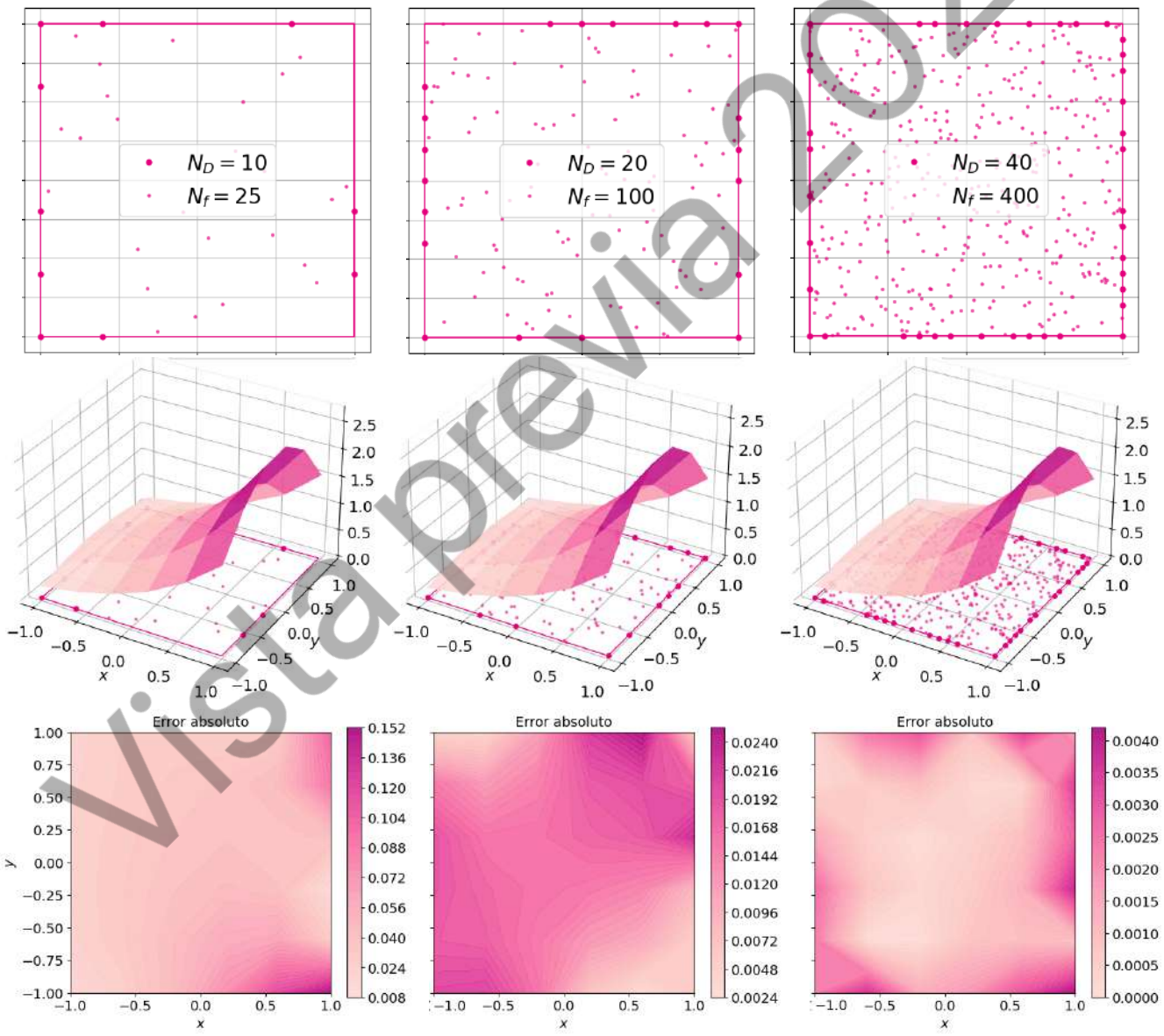


Figura 5.78 Muestra de puntos, solución aproximada y error absoluto de la red neuronal multicapa para distintos número de puntos de entrenamiento y colocación elegidos aleatoriamente.

Vista previa 2026

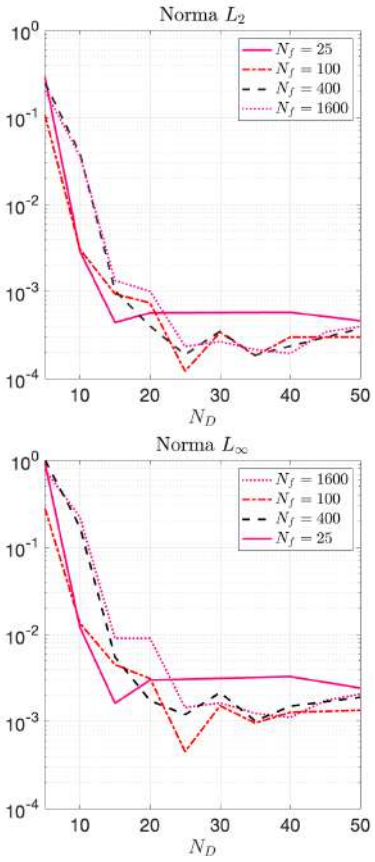


Figura 5.79 Norma L_2 y L_∞ usando diferentes cantidades de puntos de colocación N_f y variando los datos de entrenamiento N_D.

Puede observarse que las regiones con mayor error no coinciden necesariamente con aquellas donde la solución exacta alcanza sus valores máximos. Por el contrario, los errores más significativos tienden a localizarse en diferentes zonas cercanas a la frontera del dominio. Aunque las condiciones de frontera de tipo Dirichlet son impuestas durante el entrenamiento, es natural esperar errores distintos de cero sobre la frontera. Esto se debe a que la solución aproximada se obtiene mediante la minimización de una función de pérdida, por lo que el término asociado a las condiciones de contorno, denotado por E_D, no necesariamente se anula de manera exacta. Además, estos errores también están relacionados con el número limitado de puntos de entrenamiento y con la distribución aleatoria de los mismos en cada simulación.

La Tabla 5.7 muestra las normas L_2 y L_∞ para diferentes combinaciones de puntos de entrenamiento y de colocación. En términos generales, la precisión de la aproximación mejora conforme aumenta el número de puntos de entrenamiento N_D. Sin embargo, también puede observarse que, para cantidades grandes de puntos de colocación N_f, como 400 y 1600, la reducción del error comienza a estabilizarse incluso para valores moderados de N_D, tal como se aprecia en la Figura 5.78.

Por otro lado, cuando se emplea un número reducido de puntos de colocación, se requiere una mayor cantidad de datos de entrenamiento sobre la frontera para alcanzar niveles de precisión comparables. Este comportamiento es consistente con la necesidad de proporcionar suficiente información a la red neuronal para capturar adecuadamente tanto la dinámica interior del problema como las restricciones impuestas en la frontera.

Tabla 5.7 Ejemplo 2: norma L_2 y norma L_∞ del error absoluto entre la solución aproximada y la exacta para diferentes cantidades de datos de entrenamiento en la frontera N_D y puntos de collocación N_f.

	$N_f = 25$		$N_f = 100$	
N_D	Norma L_2	Norma L_∞	Norma L_2	Norma L_∞
5	2.91×10^{-1}	9.42×10^{-1}	1.07×10^{-1}	2.82×10^{-1}
10	2.89×10^{-3}	1.20×10^{-2}	3.05×10^{-3}	1.34×10^{-2}
20	5.68×10^{-4}	2.99×10^{-3}	7.38×10^{-4}	3.11×10^{-3}
40	5.76×10^{-4}	3.28×10^{-3}	2.99×10^{-4}	1.27×10^{-3}
80	2.36×10^{-4}	9.78×10^{-4}	3.04×10^{-4}	1.59×10^{-3}
160	4.80×10^{-4}	3.20×10^{-3}	2.87×10^{-4}	1.58×10^{-3}

	$N_f = 400$		$N_f = 1600$	
N_D	Norma L_2	Norma L_∞	Norma L_2	Norma L_∞
5	2.50×10^{-1}	1.01	2.02×10^{-1}	7.81×10^{-1}
10	3.86×10^{-2}	1.71×10^{-1}	3.62×10^{-2}	2.23×10^{-1}
20	4.01×10^{-4}	1.73×10^{-3}	1.00×10^{-3}	9.06×10^{-3}
40	2.36×10^{-4}	1.49×10^{-3}	1.94×10^{-4}	1.11×10^{-3}
80	3.33×10^{-4}	1.46×10^{-3}	3.78×10^{-4}	1.81×10^{-3}
160	3.05×10^{-4}	1.51×10^{-3}	3.93×10^{-4}	2.37×10^{-3}

Vista previa 2026

Notas

Para analizar el rendimiento se usó un servidor con procesador Intel Xeon E5-2690 a 3.0 GHz, sistema operativo Ubuntu 18.04 de 64 bits, 20 núcleos, 256 GB de memoria RAM y una tarjeta gráfica NVIDIA Tesla K40.

Figura 5.80 Aunque la mayor parte de los experimentos son ejecutados usando Colab, este equipo de cómputo fue empleado para medir el rendimiento de la red neuronal.

Análisis de la arquitectura de la red y el rendimiento

Es importante destacar que el desempeño de una red neuronal no depende únicamente de la cantidad y distribución de los datos de entrenamiento, sino también de la arquitectura seleccionada para la red. En particular, parámetros como el número de capas ocultas, el número de neuronas por capa y la función de activación pueden influir significativamente tanto en la precisión de la solución aproximada como en el costo computacional requerido durante el entrenamiento.

A continuación, analizamos el comportamiento de la red neuronal multicapa considerando diferentes arquitecturas de red, variando tanto el número de capas ocultas como el número de neuronas por capa. En todos los casos se mantiene fija la función de activación tangente hiperbólica. La Figura 5.81 muestra las normas L_2 y L_∞ obtenidos para distintas configuraciones de red, manteniendo fijo el número de puntos de entrenamiento y colocación en $N_D = 40$ y $N_f = 1600$, respectivamente. Los resultados muestran que todas las arquitecturas analizadas producen aproximaciones suficientemente precisas para este problema.

Podemos observar que la precisión de nuestras simulaciones mejora conforme aumentamos el número de neuronas en cada capa. Sin embargo, después de aproximadamente 20 neuronas, las mejoras comienzan a ser cada vez menores. Esto significa que agregar más neuronas no necesariamente produce una diferencia importante en la calidad de la solución. Por otro lado, en este ejemplo el número de capas ocultas no parece afectar demasiado la precisión final. De hecho, redes más profundas no muestran cambios significativos respecto a redes más simples, lo que sugiere que para este problema una arquitectura moderada resulta suficiente para obtener buenas aproximaciones.

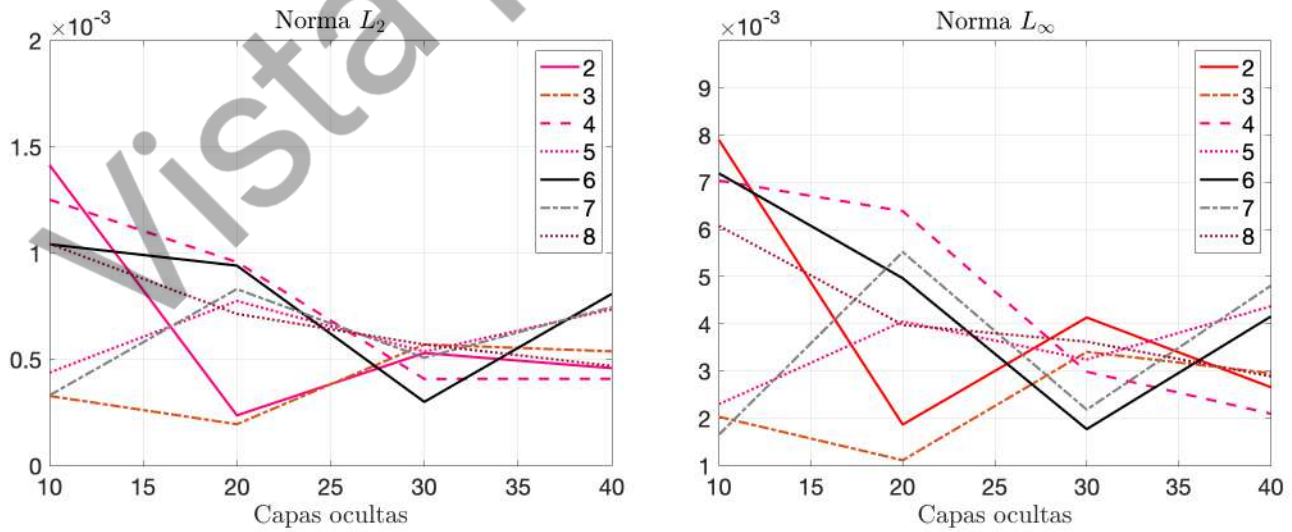


Figura 5.81 Ejemplo 2: análisis de la norma L_2 y L_∞ usando diferentes números de capas ocultas y variando el número de neuronas (N_e). El número de datos conocidos se fija en $N_u = 40$ y $N_f = 1600$, respectivamente.

Vista previa 2026



Figura 5.82 La eficiencia computacional de nuestros algoritmos es un aspecto fundamental, ya que reducir el tiempo de cómputo y el consumo de recursos también contribuye a disminuir el consumo energético.

No obstante, el número de capas sí tiene un impacto considerable en el costo computacional del algoritmo, tal como se muestra en la Tabla 5.8. En particular, se observa que arquitecturas relativamente pequeñas, como aquellas compuestas por dos capas ocultas con 20, 30 o 40 neuronas, no solo producen soluciones más precisas que arquitecturas más profundas con ocho capas y 10 neuronas, sino que además requieren un tiempo de entrenamiento considerablemente menor. De hecho, las simulaciones con dos capas resultan aproximadamente cinco veces más rápidas que las correspondientes arquitecturas profundas analizadas en este ejemplo.

Tabla 5.8 Ejemplo 2: Norma L_2 y tiempo de simulación para diferentes números de capas ocultas y diferentes números de neuronas por capa. Aquí, el número total de puntos de entrenamiento y de colocación se fija en $N_D = 40$ y $N_f = 1600$, respectivamente.

	Neuronas (Norma L_2)			
Capas	10	20	30	40
2	1.41e-03	2.36e-04	5.29e-04	4.58e-04
3	3.27e-04	1.94e-04	5.69e-04	5.37e-04
4	1.25e-03	9.57e-04	4.07e-04	4.07e-04
5	4.37e-04	7.74e-04	5.36e-04	7.34e-04
6	1.41e-03	2.36e-04	5.29e-04	4.58e-04
7	3.27e-04	1.94e-04	5.69e-04	5.37e-04
8	1.25e-03	9.57e-04	4.07e-04	4.07e-04

	Neuronas (Tiempo en segundos)			
Capas	10	20	30	40
2	13.64	6.66	4.76	6.19
3	18.97	6.16	8.28	6.07
4	23.52	9.45	22.68	15.29
5	16.30	19.35	14.60	13.28
6	25.05	11.89	15.01	26.50
7	20.21	21.07	31.91	23.89
8	30.42	30.56	32.81	37.15

Estos resultados sugieren que, para problemas suaves como el considerado en esta sección, aumentar excesivamente la profundidad de la red no necesariamente conduce a mejoras sustanciales en la precisión, mientras que sí incrementa de manera importante el costo computacional asociado al entrenamiento.

Análisis de la precisión

Como se comentó anteriormente, nuestras simulaciones alcanzan un límite natural de precisión. En consecuencia, incrementar el número de puntos de entrenamiento N_D, aumentar el número de puntos de colocación N_f o modificar la arquitectura de la red no produce mejoras significativas después de cierto punto. Es decir, nuestro formulación basada en redes neuronales no tiene la propiedad de *consistencia*.

Vista previa 2026

Esto sugiere que la principal fuente de error ya no está asociada a la cantidad de datos utilizados, sino a otros elementos propios de la aproximación mediante redes neuronales, tales como la construcción de la función de pérdida, la elección de la función de activación o incluso el algoritmo de optimización empleado durante el entrenamiento.

Para tener una referencia más clara sobre la precisión alcanzada por la red neuronal profunda, resulta natural comparar nuestros resultados con métodos numéricos clásicos para ecuaciones diferenciales parciales. En particular, utilizaremos como referencia el esquema estándar de diferencias finitas centrales de segundo orden presentado al inicio de este capítulo.

La Tabla 5.9 muestra la aproximación obtenida mediante diferencias finitas. En las simulaciones realizadas con redes neuronales, los errores mínimos obtenidos se encuentran aproximadamente en el orden de 10^{-4} y 5×10^{-4} utilizando las normas L_2 y L_∞, respectivamente. Entonces Los resultados indican que la precisión alcanzada por la red neuronal es comparable a la obtenida mediante el método de diferencias finitas utilizando una malla aproximada de 441 puntos ($N = 20$).

Tabla 5.9 Ejemplo 2: Análisis de error en un dominio rectangular utilizando un método de diferencias finitas centrales.

Malla			Solución numérica			
N	Pts. total	Pts.frontera	Norma L_2	Orden	Norma L_∞	Orden
5	36	22	3.01e-03	—	7.80e-03	—
10	121	42	8.59e-04	1.81	2.08e-03	1.91
15	256	62	3.98e-04	1.89	9.25e-04	2.00
20	441	82	2.29e-04	1.93	5.25e-04	1.97
25	676	102	1.48e-04	1.94	3.36e-04	2.00
30	961	122	1.04e-04	1.95	2.34e-04	1.99

Así, la solución basada en redes neuronales todavía no alcanza la precisión característica de un método clásico de segundo orden, lo cual representa una desventaja importante. Sin embargo, las redes neuronales presentan ventajas que no están disponibles en muchos métodos numéricos tradicionales. Por ejemplo, el número de datos requeridos sobre la frontera puede ser relativamente pequeño y, además, dichos datos no necesitan encontrarse en posiciones específicas o estructuradas dentro del dominio. Esta flexibilidad en la selección de puntos representa una ventaja considerable frente a métodos estándar como diferencias finitas, donde la construcción de la malla juega un papel fundamental.

Adicionalmente, la metodología basada en redes neuronales profundas puede extenderse de manera natural a dominios irregulares o arbitrarios sin requerir modificaciones importantes en el algoritmo principal, mientras que muchos métodos clásicos necesitan procedimientos adicionales para manejar geometrías complejas.

Vista previa 2026

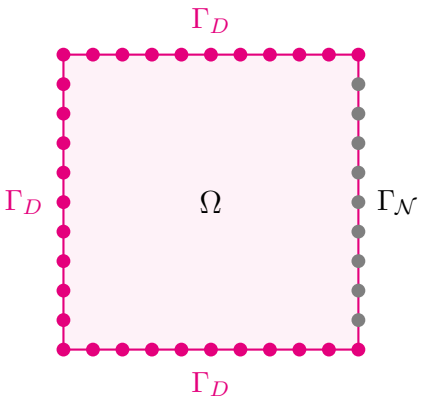


Figura 5.83 Dominio computacional con condiciones de frontera Dirichlet y Neumann empleado en el Ejemplo 3.

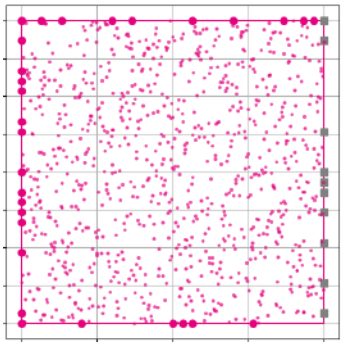


Figura 5.84 Ejemplo de la distribución de los puntos de entrenamiento y colocación seleccionados de forma aleatoria.

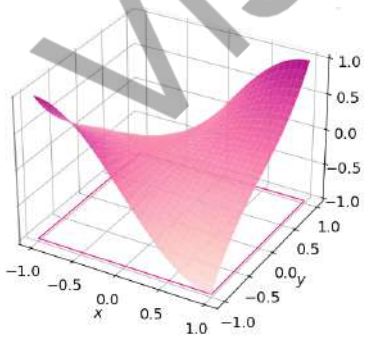


Figura 5.85 Solución exacta del Ejemplo 3.

5.6.3 Ejemplo 3: ecuación de Laplace con condiciones de frontera mixtas

Para nuestro tercer ejemplo incrementaremos ligeramente la complejidad del problema incorporando no solo condiciones de frontera de tipo Dirichlet, sino también condiciones de tipo Neumann. De esta manera, podremos analizar cómo se comporta la metodología cuando diferentes tipos de restricciones deben satisfacerse simultáneamente durante el entrenamiento de la red neuronal.

Cabe destacar que tanto los puntos asociados a las condiciones de Neumann como aquellos correspondientes a las condiciones de Dirichlet, denotados por $N_{\mathcal{N}}$ y N_D, respectivamente, se seleccionan de manera aleatoria a partir de los datos disponibles sobre la frontera. Esto permite mantener la misma flexibilidad en la distribución de puntos observada en el ejemplo anterior.

El problema

En este caso, consideraremos el siguiente problema para la ecuación de Laplace definido en la región $\Omega = [0,1] \times [0,1]$, bajo condiciones de frontera mixtas. El cambio en las dimensiones del dominio se realiza con el propósito de disponer de una solución analítica conocida.

LAPLACE CON CONDICIONES MIXTAS

Consideremos la ecuación de Laplace bidimensional con condiciones de frontera mixtas de la siguiente forma:

$$u_{xx} + u_{yy} = 0, \quad 0 \leq x \leq 1, \tag{5.88}$$

con las condiciones de frontera:

$$\begin{aligned} u(x,0) &= 0, \quad 0 \leq x \leq 1 \\ u(x,1) &= \operatorname{sen}\left(\frac{\pi}{2}x\right), \quad 0 \leq x \leq 1 \\ u(0,y) &= 0, \quad 0 < y < 1, \\ u_x(1,y) &= 0, \quad 0 < y < 1. \end{aligned} \tag{5.89}$$

Notar que, además del *borde Este del cuadrado* ($x = 1$), donde se utilizan condiciones de Neumann, se aplican condiciones de Dirichlet en todas las demás fronteras. La solución exacta del problema es suave y dado por [48]:

$$u(x,y) = \operatorname{senh}\left(\frac{\pi}{2}y\right) \frac{\operatorname{sen}\left(\frac{\pi}{2}x\right)}{\operatorname{senh}\left(\frac{\pi}{2}\right)}. \tag{5.90}$$

Vista previa 2026

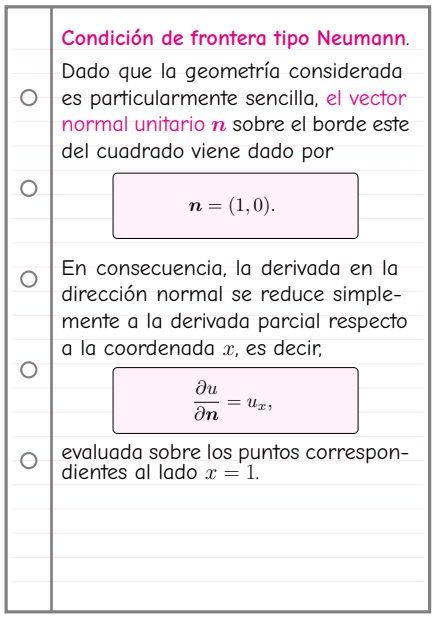


Figura 5.86 Condición de frontera de Neumann en un dominio rectangular.

Modificaciones al código

Las modificaciones necesarias en el programa son relativamente directas. Además de actualizar la solución analítica asociada a este nuevo problema, el cambio principal consiste en modificar la función de pérdida para incorporar el término correspondiente a la condición de frontera de tipo Neumann. En la práctica, esto implica calcular la derivada de la red neuronal con respecto a la coordenada x y evaluarla en los puntos de entrenamiento localizados sobre el borde donde $x = 1$.

El Código 5.13 muestra la estrategia utilizada para clasificar los nodos de la malla en puntos asociados a condiciones de frontera de tipo Dirichlet y puntos correspondientes a condiciones de tipo Neumann. Nótese que esta forma de identificar y separar los nodos de frontera es diferente a la utilizada en los ejemplos anteriores, ya que ahora resulta necesario distinguir explícitamente entre los distintos tipos de condiciones de contorno presentes en el problema.

Esta metodología permite extraer de manera directa los conjuntos de puntos de frontera, almacenándolos en los arreglos `XY_bc_np` y `XYn_bc_np`, correspondientes a las condiciones de Dirichlet y Neumann, respectivamente. Por otro lado, la construcción de los puntos de colocación en el interior del dominio permanece sin modificaciones respecto al procedimiento previamente utilizado.

```python
# ============================
# 1.E Solucion exacta
# ============================
def u_exact(x, y):
    p = torch.pi
    a = torch.sinh(p * y / 2)
    b = torch.sin(p * x / 2)
    d = torch.sinh(p / 2)

    return a * b / d
```

Figura 5.87 Código de la solución exacta del Ejemplo 3.

```python
# ============================================================
# 1.B Frontera Dirichlet y Neumann
# ============================================================

xx1 = np.hstack((X[0:1, :].T,Y[0:1, :].T )) # borde sur
xx2 = np.hstack((X[:, 0:1], Y[:, 0:1]))     # borde oeste
xx3 = np.hstack((X[-1:, :].T,Y[-1:, :].T))  # borde norte
xx4 = np.hstack((X[:, -1:], Y[:, -1:]))     # borde este

XY_bc_np  = np.vstack([xx1, xx2, xx3])
XYn_bc_np = xx4

# ============================================================
# 1.C Muestra aleatoria
# ============================================================

N_D = 30
N_N = 10

idx = np.random.choice(XY_bc_np.shape[0], N_D, replace=False)
XY_bc_random = XY_bc_np[idx, :]

idx = np.random.choice(XYn_bc_np.shape[0], N_N, replace=False)
XYn_bc_random = XYn_bc_np[idx, :]

# ============================================================
# 1.D Convertir a tensores
# ============================================================

XY_bc  = torch.tensor(XY_bc_random,  dtype=torch.float32)
XYn_bc = torch.tensor(XYn_bc_random, dtype=torch.float32)
```

Código 5.13 Selección de los puntos de entrenamiento.

Vista previa 2026

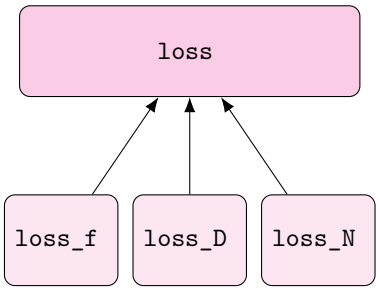


Figura 5.88 Esquema general del programa para la función de pérdida.

Para construir la función de pérdida, primero necesitamos definir los residuales asociados no solo a la ecuación diferencial, sino también a la derivada normal correspondiente a la condición de frontera de tipo Neumann. Para ello utilizaremos la solución aproximada predicha por la red neuronal en cada época de entrenamiento con el fin de calcular las derivadas necesarias mediante diferenciación automática. Posteriormente, dichas derivadas serán combinadas con las componentes conocidas del vector normal sobre la frontera correspondiente. La implementación de esta parte del código puede encontrarse en el Código 5.14.

```
# ============================================================
# 3.B Residual de la normal
# ============================================================
def un(XY, model):
    XY = XY.clone().detach().requires_grad_(True)
    u_pred = model(XY)

    # Primeras derivadas
    grads = torch.autograd.grad(u_pred, XY, grad_outputs=torch.
        ones_like(u_pred), create_graph=True)[0]
    u_x = grads[:, 0:1]
    u_y = grads[:, 1:2]

    # Derivada normal
    n_x = 1.0
    n_y = 0.0
    u_n = n_x * u_x + n_y * u_y

    return u_n
```

Código 5.14 Implementación de la derivada normal.



Figura 5.89 Aunque las modificaciones realizadas al programa original son significativas, su implementación no representa una dificultad adicional considerable.

Por otro lado, el Código 5.15 define la función de pérdida utilizada durante el entrenamiento de la red neuronal. Recordemos que esta función desempeña un papel central dentro de la metodología. En este caso, la función de pérdida contempla simultáneamente tres contribuciones distintas: el residual de la ecuación diferencial en el interior del dominio, las condiciones de frontera de tipo Dirichlet y las de tipo Neumann.

```
# ============================================================
# 4. Funcion de perdida
# ============================================================
def loss_fn(model, XY_f, XY_bc, u_bc, XYn_bc):
    # Residual en puntos de colocacion
    res = pde(XY_f, model)
    loss_f = torch.mean(res**2)
    # Condicion de Neumann en frontera
    res = un(XYn_bc, model)
    loss_N = torch.mean(res**2)
    # Condicion de Dirichlet en frontera
    u_bc_pred = model(XY_bc)
    loss_D = torch.mean((u_bc - u_bc_pred)**2)
    loss_total = loss_f + loss_N + loss_D

    return loss_total, loss_f, loss_D, loss_N
```

Código 5.15 Implementación de la función de pérdida.

Vista previa 2026

Notas

$N_D = 3N$
$N_{\mathcal{N}} = N$
$N_f = N^2$
$N_p = (N+1)^2$

Figura 5.90 Parámetros.

Los resultados numéricos

La Figura 5.91 muestra los resultados obtenidos utilizando $N_D = 3N$ puntos de entrenamiento para las condiciones de Dirichlet y $N_{\mathcal{N}} = N$ puntos asociados a las condiciones de Neumann, considerando los casos $N = 5$, 10 y 20. En todas las simulaciones se utilizaron además $N_f = N^2$ puntos de colocación dentro del dominio y una red neuronal multicapa con 4 capas ocultas y 20 neuronas por capa. Podemos observar que la red neuronal reproduce correctamente la solución exacta incluso cuando el número de datos es relativamente pequeño. Sin embargo, aunque el número total de puntos de entrenamiento y colocación aumenta considerablemente entre los distintos casos, el orden de magnitud del error absoluto permanece prácticamente similar.

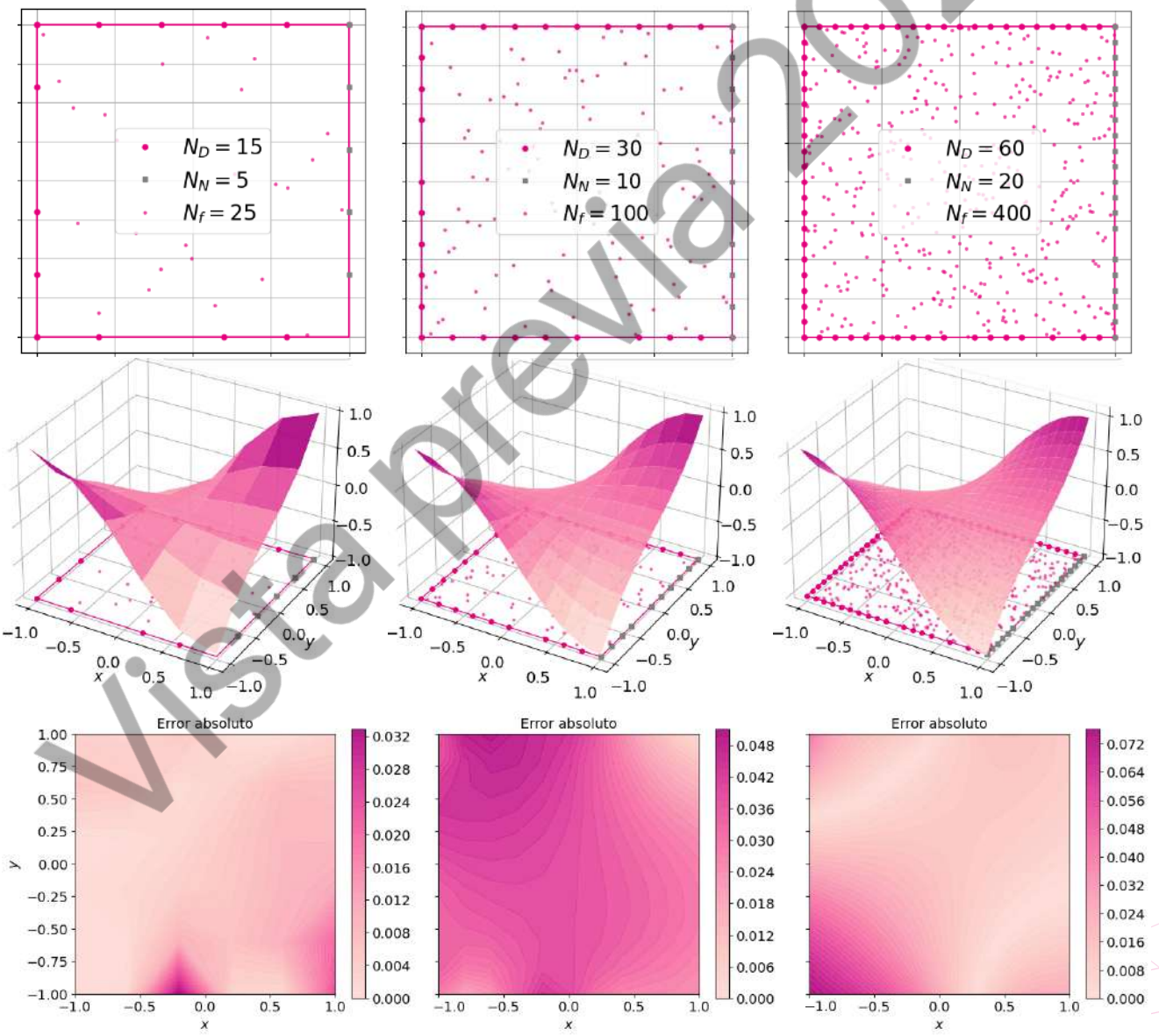


Figura 5.91 Muestra de puntos, solución aproximada y error absoluto de la red neuronal multicapa para distintos número de puntos de entrenamiento y colocación elegidos aleatoriamente.

Vista previa 2026

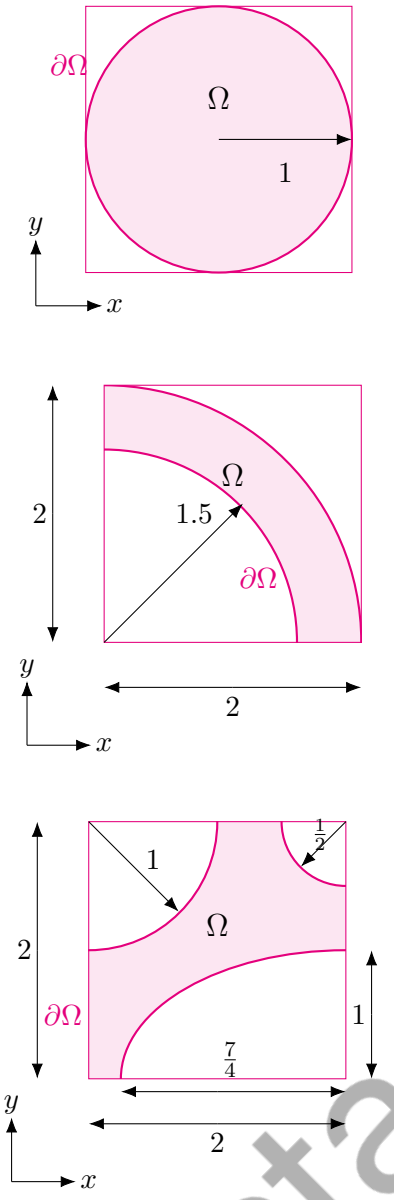


Figura 5.92 Ejemplos de dominios con geometría regular empleados para el problema de Laplace en $\mathbb{R}^2$.

5.7 RESULTADOS EN DOMINIOS IRREGULARES

Una de las principales ventajas de resolver ecuaciones diferenciales parciales mediante redes neuronales radica en la flexibilidad de esta metodología para trabajar con dominios irregulares. La estructura del programa basado en redes neuronales es esencialmente independiente de la geometría del problema; es decir, no incorpora de forma explícita la interpretación espacial de los datos.

SOLUCIÓN EN DOMINIOS IRREGULARES

Las entradas del programa se componen de conjuntos de puntos y valores conocidos, cuya ubicación e interpretación dependen del usuario y no del programa en sí. En consecuencia, para abordar dominios irregulares basta con proporcionar correctamente tanto los puntos de entrenamiento en la frontera como los puntos de colocación en el interior del dominio.

En los siguientes ejemplos se ilustrará la capacidad del método para adaptarse a distintas geometrías.

5.7.1 Ejemplo: ecuación de Laplace en dominios irregulares

Comenzaremos con el problema de Laplace (5.86) con condiciones de Dirichlet, planteado en el programa de la sección anterior, cuya solución analítica es conocida y está dada por (5.87). Sin embargo, ahora consideraremos tres dominios diferentes con geometría no rectangular, como se muestra en la Figura 5.92 y se describe a continuación:

- El primer dominio corresponde a un círculo de radio uno y centro en $(0, 0)$. Este dominio representa el caso más simple dentro de las geometrías no rectangulares, ya que posee una frontera suave y simétrica.
- El segundo ejemplo corresponde a un arco determinado por la intersección de un cuadrado de longitud de lado 2 y dos círculos de radio 2 y 1.5.
- El tercer dominio considera un dominio más arbitrario inspirado en una configuración realista de flujos como ríos, zonas fluviales o estuarios. Los límites del cuadrado de longitud 2 son deformados cortando los lados mediante dos círculos de radio 1 y 1/2 ubicados en las esquinas superior derecha e izquierda, respectivamente, y una elipse de eje mayor y menor igual a 7/4 y 1 en la esquina inferior derecha.

Puntos de entrenamiento y colocación

Los puntos de entrenamiento y colocación están ubicados en los vértices de mallas triangulares. Aunque existen diversas alternativas para

Vista previa 2026



Figura 5.93 Si bien existen múltiples herramientas para la generación de mallas triangulares en dominios irregulares, en este trabajo se opta por `Gmsh`, debido a su facilidad de implementación y que es software libre.

la generación de estas mallas en dominios irregulares, en este trabajo se emplea `Gmsh` debido a su facilidad de implementación y a su disponibilidad como software de acceso libre.

Por ejemplo, el Código 5.16 corresponde al programa que genera una malla triangular en un dominio circular en `Gmsh`. Ahí, se definen los parámetros geométricos (radio, centro y tamaño de malla). A continuación, se construye la geometría mediante tres puntos: el centro del círculo y dos puntos ubicados en los extremos izquierdo y derecho del diámetro horizontal. Con estos puntos se definen dos arcos de circunferencia que, en conjunto, describen completamente la frontera circular. Estos se agrupan en un `Curve Loop` para definir la superficie del dominio. Finalmente, se etiquetan la frontera y el interior, se genera la malla y se exporta en formato `.vtk`.

```
// ========================================
// EJEMPLO CÍRCULO
tm  = 0.1;          // Tamaño de malla
r   = 1.0;          // Radio del circulo
cx  = 0.0;          // Centro del circulo (x)
cy  = 0.0;          // Centro del circulo (y)

// GEOMETRIA
Point(1)  = { cx,   cy, 0.0,tm};
Point(2)  = { cx-r, cy, 0.0,tm};
Point(3)  = { cx+r, cy, 0.0,tm};
Circle(1) = {3,1,2};
Circle(2) = {2,1,3};

// DOMINIO COMPUTACIONAL
Curve Loop(1) = {1,2};     // Cerrando el circulo
Plane Surface(1) = {1};    // Definiendo el dominio

// ETIQUETAS
Physical Surface("Interior") = {1};
Physical Curve("Frontera")  = {1,2};

// MALLA
Mesh 2;
Mesh.SurfaceFaces = 1;
Mesh.Points       = 1;

// GUARDAR
Save "MallaCirculo.vtk";
```

Código 5.16 Programa en `Gmsh` (formato `.geo`) para la generación de una malla en un dominio circular.



Figura 5.94 `Gmsh` ofrece la posibilidad de generar geometrías y mallas tanto mediante una interfaz gráfica como a través de scripts en formato `.geo`.

En particular, para el problema de Laplace, dos mallas de diferente resolución son empleadas de la siguiente manera.

- Se usa una primera malla para fijar el número de puntos de collocación N_f y seleccionar aleatoriamente el número de puntos de entrenamiento tipo Dirichlet N_D. Así, la Malla 1 se seleccionó de manera que la longitud media interior sea cercana a $\Delta = 2/32 = 0.0625$.
- Una segunda malla de mayor resolución se emplea para los puntos de prueba N_p. Así, para la Malla 2, la longitud media interior es $\Delta = 2/64 = 0.03125$.

Vista previa 2026

Notas

No se emplean todos los nodos de la frontera como puntos de entrenamiento; en su lugar, se considera una muestra aleatoria de tamaño N_D extraída de dicho conjunto.

Figura 5.95 Puntos de entrenamiento.

El número de triángulos y nodos de cada malla se presenta en la Tabla 5.10, mientras que sus respectivas representaciones gráficas se muestran en las Figuras 5.96 y 5.97. Se observa que, al emplear mallas no estructuradas, los puntos se distribuyen de manera aproximadamente uniforme en el dominio, sin seguir un patrón regular. En particular, la Malla 1 ilustra la distribución de los puntos de colocación y de entrenamiento utilizados en el proceso de aproximación. Por otro lado, la Malla 2 muestra los puntos de prueba, los cuales son más numerosos y presentan una distribución distinta.

Tabla 5.10 Número de nodos utilizados en los dominios irregulares.

Círculo	Triángulos	Nodos	Nodos internos	Nodos frontera
Malla 1	2149	1126	1025	101
Malla 2	4337	8470	8268	202

Curva	Triángulos	Nodos	Nodos internos	Nodos frontera
Malla 1	774	440	104	336
Malla 2	3129	1672	213	1459

General	Triángulos	Nodos	Nodos internos	Nodos frontera
Malla 1	906	512	396	116
Malla 2	3695	1961	1736	225

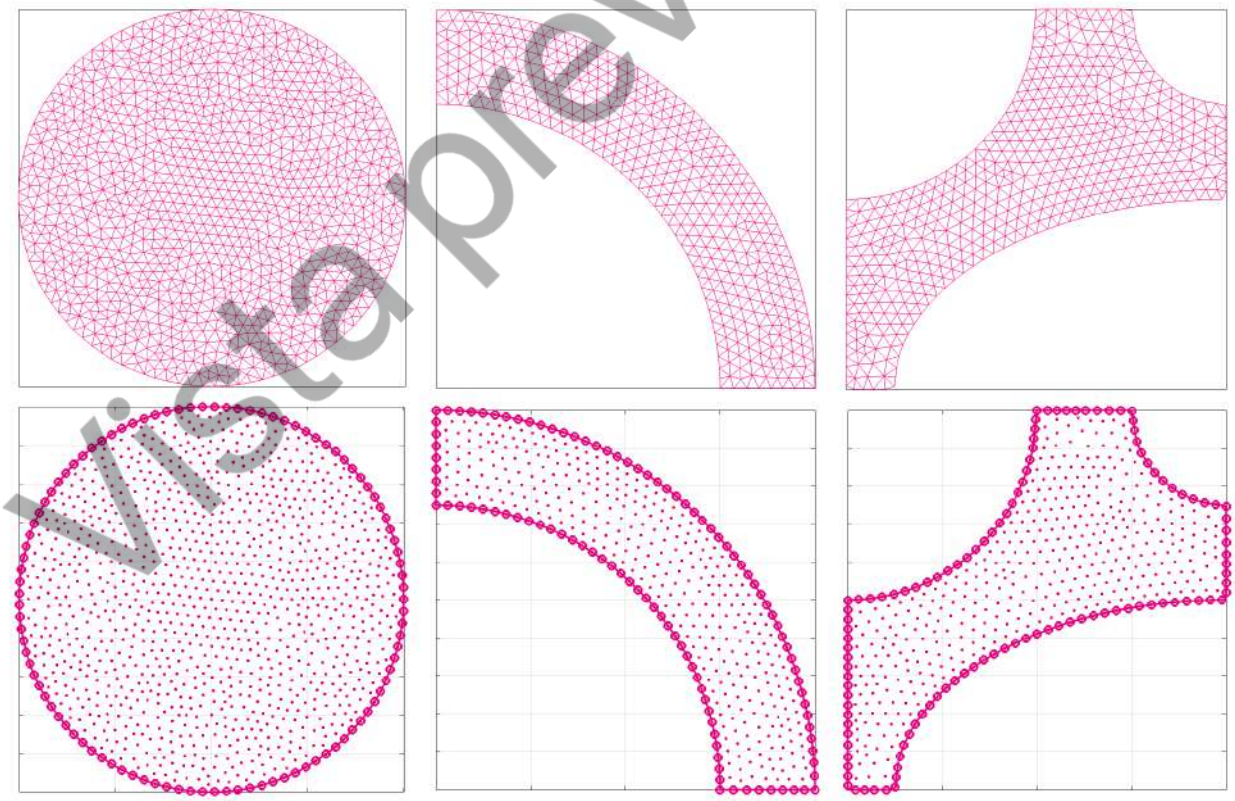


Figura 5.96 Mallas para obtener los puntos de colocación y entrenamiento empleados en nuestro programa.

Vista previa 2026



Figura 5.97 Mallas para obtener los puntos de prueba empleados en nuestro programa.



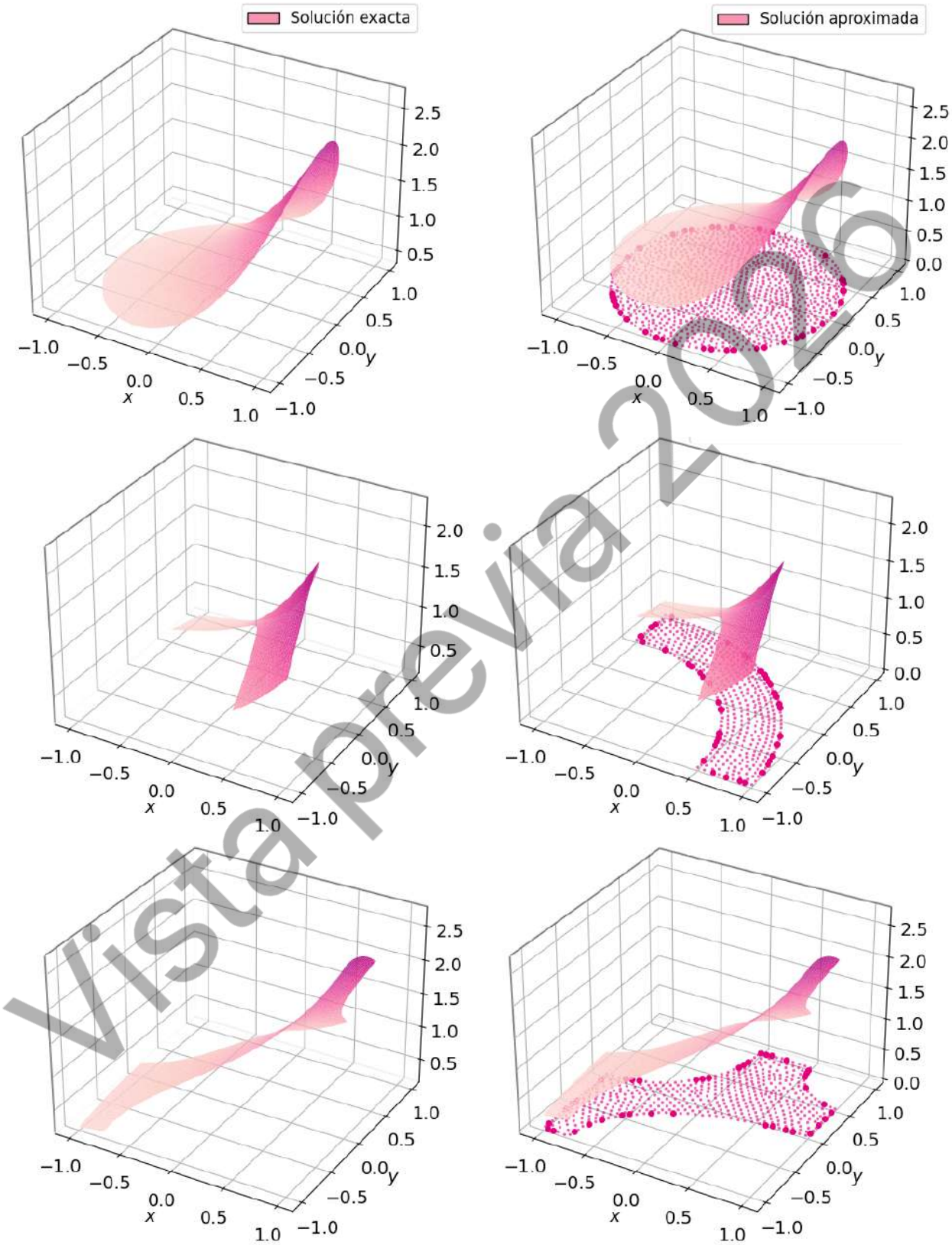
Figura 5.98 Las gráficas de las soluciones y errores en varias dimensiones permiten evaluar de manera rigurosa el desempeño de la metodología.

Resultados numéricos

La Figura 5.99 presenta la solución exacta y la solución aproximada correspondiente a este problema. En todos los casos se emplea un conjunto de $N_u = 40$ puntos de entrenamiento, mientras que los puntos de colocación corresponden a los generados por la Malla 1. La aproximación se obtiene utilizando la misma arquitectura de red neuronal considerada para los dominios rectangulares. Como era de esperarse, los resultados evidencian que el método propuesto es capaz de aproximar con precisión la solución exacta de tal manera que no podemos distinguir diferencia entre ambas gráficas.

Cabe destacar que los errores de mayor magnitud se concentran en regiones cercanas a la frontera, como se observa en la Figura 5.100. Aunque no se presenta aquí, incrementar el número de puntos de colocación no garantiza necesariamente una reducción del error. Mas aún, el comportamiento del error depende no solo de la cantidad de puntos de entrenamiento, sino también de su distribución espacial. Esto se debe a que la geometría arbitraria del dominio desempeña un papel relevante en la calidad de la aproximación. En particular, la ausencia de información de frontera en ciertas regiones tiende a incrementar el error localmente.

Vista previa 2026



Vista previa 2026

Figura 5.99 Solución exacta y aproximada utilizando diferentes dominios. El número de puntos de entrenamiento se fija en $N_u = 40$ y el número de puntos de colocación N_f está dado por los vértices de la Malla 1.

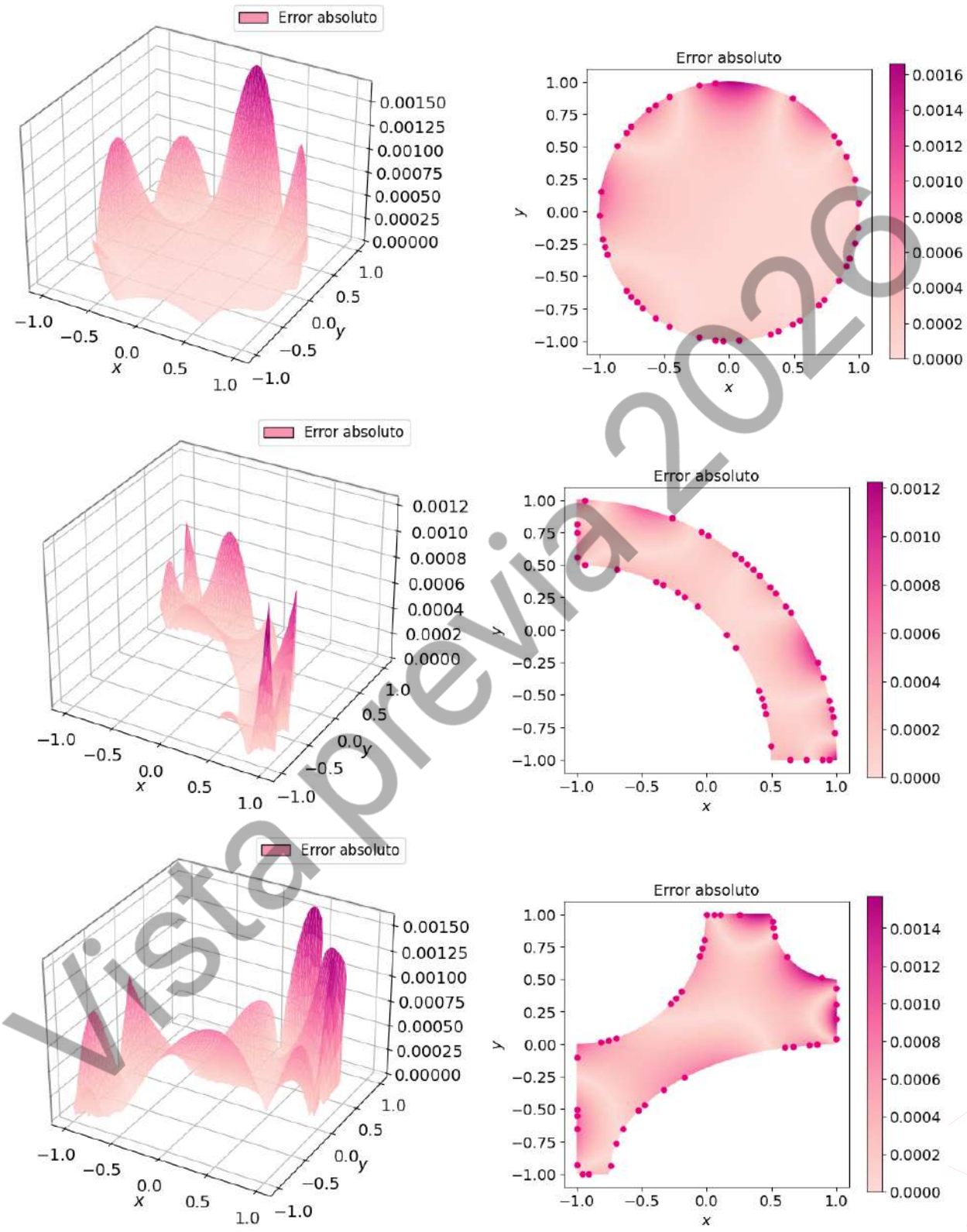


Figura 5.100 Visualización en 3D y 2D del error absoluto utilizando diferentes dominios. El número de puntos de entrenamiento se fija en $N_u = 40$ y el número de puntos de colocación N_f está dado por los vértices de la Malla 1.

Vista previa 2020



Figura 5.101 El análisis del número de puntos de entrenamiento y de su distribución constituye uno de los factores más sensibles en la obtención de soluciones precisas mediante redes neuronales multicapa.

Análisis del número de puntos de entrenamiento

La red neuronal se analiza considerando diferentes cantidades de puntos de entrenamiento. En la Tabla 5.11 se reporta el error en las normas L_2 y L_∞ para los tres casos estudiados. La solución se obtiene manteniendo fijo el número de puntos de colocación N_f, mientras que se varía el número de puntos de entrenamiento N_D. Los resultados muestran que la precisión mejora conforme aumenta N_D, hasta alcanzar un nivel de convergencia. Este comportamiento es consistente con el hecho de que una mayor cantidad de información en la frontera permite capturar con mayor fidelidad la geometría del dominio. Asimismo, se observa que, a medida que la complejidad geométrica del dominio incrementa, se requiere un mayor número de puntos de entrenamiento para lograr soluciones precisas.

Tabla 5.11 Norma L_2 y norma L_∞ entre la solución aproximada y la solución exacta para diferentes cantidades de datos de entrenamiento en la frontera.

	Círculo ($N_f = 1126$)		Curva ($N_f = 440$)		General ($N_f = 512$)	
N_u	L_2	L_∞	L_2	L_∞	L_2	L_∞
10	2.01e-01	8.21e-01	1.02e-02	5.24e-02	7.53e-03	3.38e-02
20	1.05e-04	6.07e-04	2.20e-03	1.06e-02	3.66e-04	1.99e-03
40	2.60e-04	1.65e-03	1.99e-04	1.22e-03	3.78e-04	1.56e-03
60	1.48e-04	7.49e-04	2.52e-04	1.37e-03	1.12e-04	7.21e-04
80	3.03e-04	1.62e-03	3.55e-04	1.62e-03	1.07e-04	4.62e-04
100	1.93e-04	9.95e-04	2.96e-04	9.87e-04	1.35e-04	6.21e-04

Finalmente, es importante destacar que la precisión máxima alcanzada es comparable en todos los dominios considerados, incluido el dominio rectangular, como se muestra en la Figura 5.102. Esto sugiere que, al proporcionar suficiente información a través de los puntos de entrenamiento y colocación, la red neuronal multicapa es capaz de aproximar la solución con un nivel de precisión similar, independientemente de la geometría del dominio.

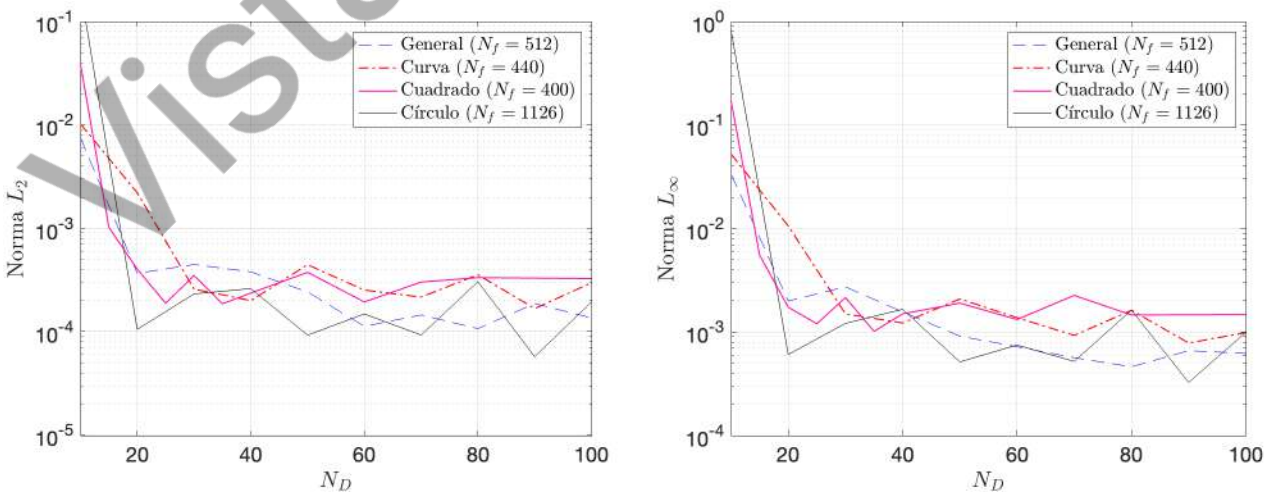


Figura 5.102 Error en norma L_2 y L_∞ utilizando diferentes cantidades de puntos de colocación N_f en distintos dominios y variando los datos de entrenamiento N_u.

Vista previa 2026

DEFINICIÓN
El orden de convergencia de los resultados numéricos se puede calcular mediante la fórmula:

$$\text{Orden} \approx \frac{\log(\|e_1\|/\|e_2\|)}{\log(\Delta_1/\Delta_2)},$$

donde $\|e_1\|$ y $\|e_2\|$ son las normas del error con distancias de longitudes medias Δ_1 y Δ_2 respectivamente.

Figura 5.103 Definición del orden de convergencia numérico para mallas no estructuradas.

Análisis de la precisión

Como se ha observado en los resultados anteriores, la solución aproximada presenta un límite de precisión. En consecuencia, incrementar el número de puntos de entrenamiento N_D o de colocación N_f, así como modificar la arquitectura de la red, no conduce a mejoras significativas en la exactitud de la aproximación. Esto sugiere que la principal fuente de error radica en otros componentes del modelo, tales como la formulación de la función de pérdida, la elección de la función de activación o el algoritmo de optimización empleado.

Con el fin de contar con una referencia sobre la precisión de la aproximación obtenida mediante redes neuronales multicapa, se comparan los resultados con soluciones numéricas calculadas mediante un esquema estándar, ampliamente utilizado y bien documentado, como el método de elemento finito (*Finite Element Method*, FEM). Para el caso de estudio, se considera únicamente el dominio circular y la solución numérica se obtiene mediante elementos lineales, siguiendo la implementación presentada por Alberty et al. [47].

Por un lado, en las simulaciones realizadas sobre el dominio circular, los errores mínimos alcanzados se sitúan del orden de 10^{-4} y 5×10^{-4} en las normas L_2 y L_∞, respectivamente, como se muestra en la Tabla 5.11. Por otro lado, la Tabla 5.12 presenta los errores correspondientes obtenidos mediante el método de elementos finitos, utilizando mallas triangulares con distintas longitudes medias interiores (Δ).

Tabla 5.12 Análisis del error usando FEM para el dominio circular.

Δ	Elementos	Nodos	Norma L_2	Orden	Norma L_∞	Orden
2/16	561	307	3.10e-04	—	1.39e-03	—
2/32	2149	1126	8.16e-05	1.92	4.29e-04	1.69
2/64	8470	4337	1.91e-05	2.09	1.42e-04	1.59
2/128	33339	16872	5.24e-06	1.87	4.20e-05	1.76

Nótese que la precisión alcanzada por la formulación basada en PINNs es comparable a la obtenida mediante el método de elementos finitos utilizando la Malla 1. Sin embargo, a diferencia de este último, la solución proporcionada por la red neuronal no satisface una propiedad de consistencia en el sentido clásico. En consecuencia, los errores no disminuyen de manera sistemática al incrementar el número de puntos, lo cual constituye una limitación de esta metodología.

No obstante, es importante destacar que los datos utilizados en el enfoque basado en PINNs no requieren una discretización triangular estructurada ni uniformemente distribuida en todo el dominio, como ocurre en el método de elemento finito. Esta característica representa una ventaja significativa, particularmente en problemas con geometrías arbitrarias, donde la generación de mallas de buena calidad puede resultar costosa o compleja.

Notas
Aunque los resultados no se presentan aquí, se obtiene conclusiones similares al comparar errores entre la formulación con redes neuronales y elemento finito para los otros dominios irregulares.

Figura 5.104 Análisis de la precisión con otros dominios.

Vista previa 2026

RESUMEN

Este capítulo extiende la metodología de redes neuronales informadas por la física (PINNs) al estudio de ecuaciones diferenciales parciales elípticas en dos dimensiones, tomando como referencia la ecuación de Laplace.

A partir de los conceptos desarrollados en el capítulo anterior, se presentan las modificaciones necesarias para adaptar la implementación del método a problemas bidimensionales, prestando especial atención a la formulación de la función de pérdida, la distribución de los puntos de colocación y la incorporación de las condiciones de frontera.

En primer lugar, se estudian problemas definidos sobre dominios regulares, comenzando con una distribución uniforme de los puntos de colocación y extendiendo posteriormente el análisis a distribuciones aleatorias. A través de estos ejemplos, se evalúa la influencia del número y la distribución de los puntos sobre la precisión de las aproximaciones obtenidas. Asimismo, se analiza el efecto de diferentes arquitecturas neuronales, considerando tanto su capacidad de aproximación como la variabilidad de los resultados entre ejecuciones independientes.

Posteriormente, se incorporan condiciones de frontera mixtas, incluyendo condiciones de tipo Neumann, con el propósito de mostrar la flexibilidad de las PINNs para abordar diferentes formulaciones de problemas elípticos. Se describen las modificaciones necesarias en la implementación y se estudia cómo estas condiciones influyen en la precisión del método. Finalmente, se consideran problemas definidos sobre dominios irregulares, lo que permite explorar las ventajas y dificultades de aplicar esta metodología a geometrías de mayor complejidad.

A lo largo del capítulo se pone especial énfasis en el análisis numérico de los resultados, mostrando que la precisión de las PINNs no depende únicamente de la complejidad de la arquitectura neuronal, sino también de factores como la selección de los puntos de colocación, la formulación de la función de pérdida y el proceso de entrenamiento. Por último, se proponen ejercicios que involucran otras ecuaciones diferenciales parciales, con el objetivo de que el lector consolide los conocimientos adquiridos y explore las posibilidades y limitaciones de esta metodología en problemas bidimensionales.

Vista previa 2026

PARÁMETROS CLAVE

Al emplear PINNs para problemas elípticos, es importante considerar los parámetros clave:

- **Arquitectura de la red:**
  - Número de capas ocultas.
  - Número de neuronas por capa.
  - Tipo de función de activación.
- **Datos de entrenamiento:**
  - Cantidad de puntos de entrenamiento y de colocación.
  - Distribución de los datos.
- **Función de pérdida (costo):**
  - El error cuadrático medio.
  - Inclusión de la EDP y CF Neumann.
- **Algoritmo de optimización:**
  - Adam.
  - Número de épocas.
  - Tasa de aprendizaje.

EJERCICIOS

Realiza los siguientes ejercicios con atención, asegurándote de comprender cada uno de los pasos y de aplicar correctamente los conceptos estudiados. Dedica el tiempo necesario para resolverlos con cuidado y, en caso de surgir dudas, consulta las notas correspondientes para reforzar tu aprendizaje.

Ejercicio 5.1: Ecuación de Poisson. Considere la siguiente ecuación de Poisson con condiciones de frontera de Dirichlet homogéneas:

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = g, & (x,y) \in \Omega, \\ u(x,y) = 0, & (x,y) \in \partial\Omega, \end{cases} \tag{5.1}$$

donde $\Omega = [-1,1] \times [-1,1]$. Este problema corresponde a una ecuación de difusión con coeficiente constante $\beta = 1$. La función $g(x,y)$ se determina considerando que la solución exacta del problema está dada por

$$u(x,y) = \sin(\pi x)\sin(\pi y). \tag{5.2}$$

Utilizando la metodología de redes neuronales informadas por la física (PINNs), realice las siguientes actividades:

- Determine analíticamente la función $g(x,y)$ sustituyendo la solución exacta en la ecuación de Poisson. Verifique además que la solución propuesta satisface las condiciones de frontera.
- Modifique el código desarrollado anteriormente para incorporar el término fuente $g(x,y)$. Formule la función de pérdida considerando tanto el residuo de la ecuación diferencial como las condiciones de frontera.
- Implemente una PINN utilizando una arquitectura de cuatro capas ocultas. Considere inicialmente $N_f = 1600$ puntos de colocación en el interior del dominio y $N_u = 40$ puntos sobre la frontera, distribuidos aleatoriamente.
- Compare gráficamente la solución exacta y la aproximación obtenida mediante la PINN. Represente también el error absoluto sobre el dominio y calcule los errores en ambas normas.
- Analice la influencia del número de puntos de colocación N_f sobre la precisión del método. Mantenga fijo el número de puntos de frontera y considere diferentes valores de N_f.

Ejercicio 5.2: Ecuación elíptica con coeficientes variables. Considere la siguiente ecuación diferencial parcial elíptica con coeficientes variables y condiciones de frontera de Dirichlet homogéneas:

$$\begin{cases} \dfrac{\partial}{\partial x}\left(\beta \dfrac{\partial u}{\partial x}\right) + \dfrac{\partial}{\partial y}\left(\beta \dfrac{\partial u}{\partial y}\right) + \kappa u = g, & (x,y) \in \Omega, \\ u(x,y) = 0, & (x,y) \in \partial\Omega, \end{cases} \tag{5.3}$$

donde $\Omega = [-1,1] \times [-1,1]$ y los coeficientes están dados por

$$\beta(x,y) = \sin(x+y), \quad \kappa(x,y) = -(x^2+y^2). \tag{5.4}$$

La función $g(x,y)$ se determina considerando que la solución exacta del problema es

$$u(x,y) = \sin(\pi x)\sin(\pi y). \tag{5.5}$$

Utilizando la metodología de redes neuronales informadas por la física (PINNs), realice las siguientes actividades:

- Determine analíticamente la función $g(x,y)$ sustituyendo la solución exacta en la ecuación diferencial. Verifique que se satisfacen las condiciones de frontera.
- Modifique el código desarrollado anteriormente para incorporar los coeficientes variables $\beta(x,y)$ y $\kappa(x,y)$. Formule la función de pérdida correspondiente, considerando tanto el residuo de la ecuación diferencial como las condiciones de frontera.
- Implemente una PINN utilizando una arquitectura de cuatro capas ocultas. Considere inicialmente $N_f = 1600$ puntos de colocación en el interior del dominio y $N_u = 40$ puntos sobre la frontera, distribuidos aleatoriamente.
- Compare gráficamente la solución exacta y la aproximación obtenida mediante la PINN. Represente también el error absoluto sobre el dominio y calcule los errores en las normas L_2 y L_∞.

Vista previa 2026

- Analice la influencia del número de puntos de frontera sobre la precisión del método. Para ello, mantenga fijo $N_f = 1600$ y considere diferentes valores de N_u. Determine si incrementar el número de puntos de frontera mejora la aproximación obtenida.
- Realice al menos cinco ejecuciones independientes utilizando la misma configuración. Compare los errores obtenidos y analice la variabilidad del método, prestando especial atención a las regiones donde se presentan los errores más elevados.

Vista previa 2026

Bibliografía

[1] Chavlis, S., & Poirazi, P. (2021). Drawing inspiration from biological dendrites to empower artificial neural networks. Current opinion in neurobiology, 70, 1-10.

[2] Rosenblatt, F. The perceptron: A probalistic model for information storage and organization in the brain. Psycological Review, (1958) (65), 386-408.

[3] Kingma D. P., Ba J. ADAM: A method for stochastic optimization. arXiv.1412.6980 (2017)

[4] I. J. Goodfellow, Y. Bengio and A. Courville, Deep Learning, *MIT Press*, 2016.

[5] G. Shrivastava, S. Karmakar, M. K. Kowar, P. Guhathakurta, Application of artificial neural networks in weather forecasting: a comprehensive literature review, *International Journal of Computer Applications*, 51(18) (2012).

[6] Q. Zhong, Y. Liu, X. Ao, B. Hu, J. Feng, J. Tang , Q. He, Financial Defaulter Detection on Online Credit Payment via Multi-view Attributed Heterogeneous Information Network, *In Proceedings of The Web Conference 2020*, (2020), 785–795.

[7] M. Lam, Neural network techniques for financial performance prediction: integrating fundamental and technical analysis, *Decision support systems*, 37(4) (2004), 567–58.

[8] G. Youyang, COVID-19 Projections Using Machine Learning, 2020. Available from: https://covid19-projections.com/about.

[9] J. Han, A. Jentzen, E. Weinan, Solving high-dimensional partial differential equations using deep learning, *Proceedings of the National Academy of Sciences*, 115(34) (2018), 8505–8510.

[10] J. Sirignano, K. Spiliopoulos, DGM: A deep learning algorithm for solving partial differential equations, *Journal of computational physics*, 375 (2018), 1339–1364.

[11] K. Xu, E. Darve, The neural network approach to inverse problems in differential equations, preprint, arXiv:1901.07758.

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

[13] C. Michoski, M. Milosavljevic, T. Oliver, D. Hatch, Solving irregular and data-enriched differential equations using deep neural networks, preprint, arXiv:1905.04351.

[14] K. Gurney, An introduction to neural networks, CRC press, 1997.

[15] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, Automatic differentiation in machine learning: a survey, preprint, arXiv:1502.05767.

Vista previa 2026

[16] S. Marsland, Machine learning: an algorithmic perspective, CRC press, (2015).

[17] S. Pattanayak, Pro deep learning with tensorflow: A mathematical approach to advanced artificial intelligence in python, New York, NY, USA. Apress, (2017).

[18] G. Zaccone, R. Karim, Deep learning with tensorFlow: Explore neural networks and build intelligent systems with python, Packt Publishing Ltd, (2018).

[19] M. A. Nielsen, Neural networks and deep learning, Determination press San Francisco, CA., 2015.

[20] Z. Mao, A. D. Jagtap, G. E. Karniadakis, Physics-informed neural networks for high-speed flows, *Computer Methods in Applied Mechanics and Engineering*, 360 (2020), 112789.

[21] L. Sun, H. Gao, S. Pan, J. X. Wang, Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data. *Computer Methods in Applied Mechanics and Engineering*, 361 (2020),112732.

[22] M. Raissi, A. Yazdani, G. E. Karniadakis, Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations, *Science*, 367(6481) (2020), 1026–1030.

[23] J. Berg, K. Nyström, A unified deep artificial neural network approach to partial differential equations in complex geometries, *Neurocomputing*, 317 (2018), 28–41.

[24] E. Weinan, B. Yu, The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems, *Communications in Mathematics and Statistics*, 6(1) (2018), 1–12.

[25] C. Anitescu, E. Atroshchenko, N. Alajlan, T. Rabczuk, Artificial neural network methods for the solution of second order boundary value problems, *Computers, Materials and Continua*, 59(1) (2019), 345–359.

[26] L. Lu, X. Meng, Z. Mao, G. E. Karniadakis, DeepXDE: A deep learning library for solving differential equations, preprint, arXiv:1907.04502.

[27] A. Koryagin, R. Khudorozkov, S. Tsimfer, PyDEns: A python framework for solving differential equations with neural networks, *arXiv preprint arXiv*, (2019).

[28] J. Han, M. Nica, A.R. Stinchcombe, A derivative-free method for solving elliptic partial differential equations with deep neural networks, preprint, arXiv:2001.06145.

[29] Y. Zang, G. Bao, X. Ye, H. Zhou, Weak adversarial networks for high-dimensional partial differential equations, *Journal of Computational Physics*, (2020),109409.

[30] C. S. Peskin, Flow patterns around heart valves: a numerical method, *J Comput Phys*, 10(2) (1972), 252–271.

[31] C. S. Peskin, Numerical analysis of blood flow in the heart, *J Comput Phys*, 25(3) (1977), 220–252.

[32] Y. Kim, C. S. Peskin, Penalty immersed boundary method for an elastic boundary with mass, *Physics of Fluids*, 19(5) (2007), 053103.

[33] Y. Mori, C. S. Peskin, Implicit second-order immersed boundary methods with boundary mass, *Computer methods in applied mechanics and engineering*, 197(25-28) (2008), 2049–2067.

[34] Y. Kim, C. S. Peskin, 2-D parachute simulation by the immersed boundary method, *SIAM Journal on Scientific Computing*, 28(6) (2006), 2294–2312.

Vista previa 2026

[35] S. Lim, A. Ferent, X. S. Wang, C. S. Peskin, Dynamics of a closed rod with twist and bend in fluid, *SIAM Journal on Scientific Computing*, 31(1) (2008), 273–302.

[36] P. J. Atzberger, C. S. Peskin, A Brownian dynamics model of kinesin in three dimensions incorporating the force-extension profile of the coiled-coil cargo tether, *Bulletin of mathematical biology*, 68(1) (2006), 131.

[37] P. J. Atzberger, P. R. Kramer, C. S. Peskin, Stochastic immersed boundary method incorporating thermal fluctuations, *Proceedings in Applied Mathematics and Mechanics*, 7(1) (2007), 1121401–1121402.

[38] T. G. Fai, B. E. Griffith,Y. Mori, C. S. Peskin, Immersed boundary method for variable viscosity and variable density problems using fast constant-coefficient linear solvers I: Numerical method and results, *SIAM Journal on Scientific Computing*, 35(5) (2013), B1132–B1161.

[39] C. S. Peskin, The immersed boundary method, *Acta Numerica*, 11 (2002), 1–39.

[40] R. Mittal, G. Iaccarino, Immersed boundary methods, *Annual Review of Fluid Mechanics*, 37 (2005), 239–261.

[41] R. J. Leveque, Z. Li, The immersed interface method for elliptic equations with discontinuous coefficients and singular sources, *SIAM Journal on Numerical Analysis*, 31(4) (1994), 1019–1044.

[42] C. Basdevant, M. Deville, P. Haldenwang, J. Lacroix, J. Ouazzani, R. Peyret, P. Orlandi, A. Patera, Spectral and finite difference solutions of the Burgers equation, *Computers & fluids*, 14 (1986), 23–41.

[43] D. A. Fournier, H. J. Skaug, J. Ancheta, J. Ianelli, A. Magnusson, M. N. Maunder, A. Nielsen, J. Sibert, Ad model builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models, *Optimization Methods & Software*, 27(2) (2012), 233–249.

[44] M. Abadi, et al, Tensorflow: Large-scale machine learning on heterogeneous distributed systems, preprint, arXiv:1603.04467.

[45] W. Qi, M. Yue, Z. Kun, T. Yingjie, A Comprehensive Survey of Loss Functions in Machine Learning, *Annals of Data Science*, 1 (2020), 1–26.

[46] F. Zhengqing, L. Goulin, G. Lanlan, Sequential quadratic programming method for nonlinear least squares estimation and its application, *Mathematical Problems in Engineering*, (2019).

[47] J. Alberty, C. Carstensen, S. A. Funken, Remarks around 50 lines of Matlab: short finite element implementation. *Numerical algorithms*, 20(2-3) (1999), 117-137.

[48] F. Civan, C. M. Sliepcevich, Solution of the Poisson equation by differential quadrature. *International journal for numerical methods in engineering*, 19(5) (1983), 711–724.

[49] J. Hua, J. Lou, Numerical simulation of bubble rising in viscous liquid. *Journal of Computational Physics*, (2007) 222(2), 769–795.

[50] M. Uh, S. Xu, The immersed interface method for simulating two-fluid flows. *Numerical Mathematics: Theory, Methods and Applications*, (2014) 7(4), 447–472.

[51] R. Byrd, P. Lu, J. Nocedal, C. Zhu, A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, (1995) 16(5), 1190–1208.

Vista previa 2026

[52] C. B. Moler. Numerical computing with MATLAB. Society for Industrial and Applied Mathematics (2004).

[53] M. Uh Zapata, R. Itzá Balam. High-order implicit finite difference schemes for the two-dimensional Poisson equation. Applied Mathematics and Computation, (2017) 309, 222–244.

[54] R. Itzá Balam, R., M. Uh Zapata, J. Montalvo-Urquizo. Global second-order immersed interface Poisson solver using zero-order local truncation errors. Boletín de la Sociedad Matemática Mexicana, (2024), 30(3), 83.

[55] R. Itzá Balam, M. Zapata, J. Montalvo-Urquizo (2023). A new sixth-order immersed interface method for solving Poisson equations with straight interfaces. Métodos numéricos para cálculo y diseño en ingeniería: Revista internacional, (2023) 39(4), 1-20.

[56] Boyer, C. B. (2012). The history of the calculus and its conceptual development. Courier Corporation.

[57] Zill, D. G., Edwards Jr, C. H., & Penney, D. E. (2022). Ecuaciones diferenciales con aplicaciones de modelado. Ed. Thomson Learnig.

[58] Lemarié-Rieusset, P. G. (2016). The Navier–Stokes problem in the 21st century. CRC Press.

[59] Hampel, F. R., Ronchetti, E. M., Rousseeuw, P. J., & Stahel, W. A. (1986). *Robust statistics: The approach based on influence functions*. Wiley.

[60] Burden, R. L., & Faires, J. D. (2010). Numerical analysis. Cengage Learning.

[61] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F.A. Hamprecht, Y. Bengio, A. Courville On the expectral bias of neural networks, preprint arXiv:1806.08734. (2019)

[62] Gottlieb, D., Orszag, S. A., & Mathematics, S. F. I. A. A. (1977). Numerical analysis of spectral methods theory and applications. https://doi.org/10.1137/1.9781611970425

[63] Gauze, G. F. (1934). The struggle for existence. https://doi.org/10.5962/bhl.title.4489

Vista previa 2026

Miguel Uh Zapata
Reymundo Itzá Balam



Todos los derechos reservados por
los autores.

Vista previa 2026