EVALUACIÓN DE UNA RED NEURONAL PARA LA SOLUCIÓN DE ECUACIONES DIFERENCIALES

JOSÉ MANUEL MACHADO LOAIZA
jmmachadol@eafit.edu.co

Proyecto de Grado presentado como requisito
parcial para optar por el título de Ingeniero Físico

Asesor
Nicolás Guarín-Zapata

Universidad EAFIT
Escuela de Ciencias Aplicadas e Ingeniería
Ingeniería Física
Medellín, Colombia
2023

Contenido

Lista de Figuras

RESUMEN

La solución de ecuaciones diferenciales es un problema fundamental en la modelación matemática de fenómenos científicos e ingenieriles. Los métodos numéricos tradicionales ofrecen una aproximación a la solución de ecuaciones diferenciales a partir de una geometría y condiciones iniciales y de frontera específicas. En los últimos años, el aprendizaje automático ha emergido como una herramienta prometedora para aproximar soluciones de ecuaciones diferenciales. En particular, las redes neuronales han sido utilizadas para desarrollar enfoques de optimización y soluciones en tiempo real que se conciben para aproximar una función objetivo. En este trabajo se propone una arquitectura de red neuronal por aprendizaje supervisado para aproximar soluciones de la ecuación diferencial de Poisson. La red neuronal permite variar el término fuente de la ecuación diferencial como entrada y su arquitectura se muestra en una interfaz propuesta. Se plantean preguntas importantes para la evaluación del desempeño de la red neuronal, incluyendo la precisión del resultado, la generalización de la red entrenada a otros problemas, los márgenes de error y las respectivas tolerancias de los parámetros de entrada, los recursos computacionales utilizados y el costo de generar los datos de entrada y entrenar la red neuronal. Este trabajo busca contribuir al desarrollo de métodos computacionales eficientes para la solución de ecuaciones diferenciales en la ciencia e ingeniería computacional mediante la evaluación del desempeño de una red neuronal.

Palabras clave: redes neuronales, ecuaciones diferenciales, métodos numéricos, aprendizaje automático, diseño de experimentos computacional, ajuste de hiperparámetros, modelado matemático, análisis numérico.

ABSTRACT

Solving differential equations is a fundamental problem in the mathematical modeling of scientific and engineering phenomena. Traditional numerical methods offer an approximation to the solution of differential equations based on specific geometry and initial and boundary conditions. In recent years, machine learning has emerged as a promising tool for approximating solutions to differential equations. In particular, neural networks have been used to develop optimization approaches and real-time solutions that are designed to approximate an objective function. In this work, a supervised learning neural network architecture is proposed to approximate solutions to the Poisson differential equation. The neural network allows for the variation of the source term of the differential equation as input, and its architecture is presented in a proposed interface. Important questions are raised for evaluating the performance of the neural network, including the accuracy of the result, the generalization of the trained network to other problems, error margins and respective input parameter tolerances, computational resources used, and the cost of generating input data and training the neural network. This work aims to contribute to the development of efficient computational methods for solving differential equations in computational science and engineering by evaluating the performance of a neural network.

Keywords: neural networks, differential equations, numerical methods, machine learning, computational design of experiments, hyperparameter tuning, mathematical modeling, numerical analysis.

INTRODUCCIÓN

La solución de ecuaciones diferenciales es fundamental en diversos campos de las ciencias aplicadas y la ingeniería, ya que estas ecuaciones describen fenómenos físicos, como la propagación de calor, la dinámica de fluidos y la interacción de partículas. Los métodos numéricos tradicionales, como el Método de los Elementos Finitos (FEM), son ampliamente utilizados para resolver ecuaciones diferenciales debido a su eficiencia y precisión. Sin embargo, estos métodos pueden presentar limitaciones en problemas de alta complejidad y dimensionalidad, lo que ha llevado a la búsqueda de enfoques alternativos.

En este contexto, las redes neuronales, como una rama del aprendizaje profundo y la inteligencia artificial, han demostrado ser prometedoras en la aproximación de funciones y solución de problemas matemáticos. Este trabajo de grado explora la viabilidad de utilizar redes neuronales, específicamente un Perceptrón Multicapa (MLP), en la solución de ecuaciones diferenciales, en particular, la ecuación diferencial de Poisson.

El origen de este trabajo radica en la necesidad de investigar enfoques alternativos a los métodos numéricos tradicionales, capaces de abordar problemas más complejos y multidimensionales. La importancia de este trabajo se encuentra en su contribución al conocimiento de las redes neuronales aplicadas a la solución de ecuaciones diferenciales y su potencial impacto en el avance de problemas en ciencias aplicadas e ingeniería.

El alcance de este trabajo se centra en evaluar una prueba de concepto de una red neuronal para la solución de ecuaciones diferenciales. Se documenta el estado del arte en redes neuronales como método de aproximación a las ecuaciones diferenciales, se propone una red neuronal adecuada y se ajustan sus parámetros. Posteriormente, se evalúan las capacidades de solución de la red neuronal mediante un diseño de experimentos computacional y se comparan los resultados con los obtenidos por FEM.

La metodología empleada en este trabajo incluye el desarrollo de una arquitectura de red neuronal, la selección de funciones de activación apropiadas, la generación de conjuntos de datos para entrenamiento y evaluación, y la optimización de los parámetros de la red. Además, se realizan análisis de robustez y generalización de la red neuronal aplicada a

conjuntos de datos de prueba no vistos previamente.

El impacto de este trabajo para el área de interés radica en el estudio de la viabilidad de utilizar redes neuronales en la solución de ecuaciones diferenciales y la propuesta de una metodología efectiva y eficiente para abordar este tipo de problemas. Los resultados obtenidos abren nuevas posibilidades para el desarrollo y estudio de las arquitecturas de red en el campo de las ciencias aplicadas y la ingeniería, así como las bases para investigaciones futuras en el uso de redes neuronales en la solución de ecuaciones diferenciales y problemas matemáticos más complejos.

1. PLANTEAMIENTO DEL PROBLEMA

Las ecuaciones diferenciales son fundamentales para modelar matemáticamente distintos fenómenos de la ciencia y la ingeniería (on CSE Education, 2001). Resolverlas es un paso crucial hacia un conocimiento preciso del comportamiento de los sistemas naturales y de ingeniería. En general, los métodos analíticos no suelen ser suficientes para resolver las ecuaciones diferenciales que representan sistemas reales en un grado aceptable. Para ello, la ciencia e ingeniería computacional se encarga de encontrar numéricamente una aproximación a la solución de dichas ecuaciones diferenciales que rigen la física del sistema para una geometría dada, propiedades de los materiales, leyes constitutivas y condiciones iniciales y de frontera (Chen, Lu, Karniadakis, y Negro, 2020).

Dichos métodos numéricos ajustan de manera aproximada las ecuaciones diferenciales a partir de una geometría específica y de unas condiciones iniciales y de frontera (Langtangen y Mardal, 2019). En la Figura 1.1 se muestra un esquema de la arquitectura de un modelo tradicional de métodos numéricos.

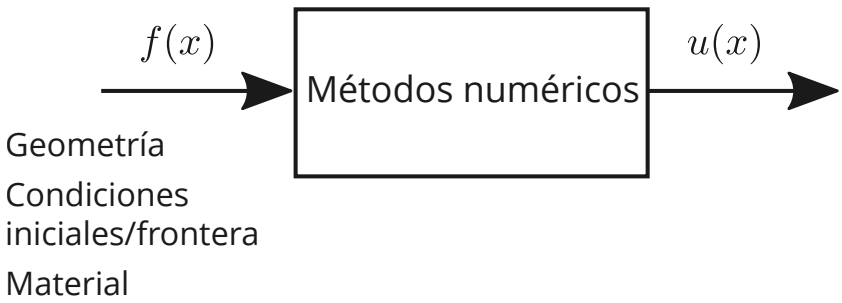

Figura 1.1: Interfaz de los métodos numéricos.

En los últimos años, la disponibilidad de grandes conjuntos de datos, combinada con la mejora de los algoritmos y el crecimiento exponencial de la potencia de cálculo, ha provocado un aumento sin precedentes del interés por el tema del aprendizaje automático (Gasmi y Tchelepi, 2021). Por su parte, las redes neuronales han surgido como modelos computacionales cada vez más utilizados para desarrollar enfoques de optimización y soluciones en tiempo real que se conciben para aproximar una función objetivo. (Kadapa,

2021). Las funciones de aproximación dependen de ciertos parámetros (los pesos y los sesgos) que deben ser *aprendidos* mediante un proceso de *entrenamiento*. Para nuestro problema de estudio, se propone una arquitectura de red neuronal por aprendizaje supervisado que permita variar el término fuente de la ecuación diferencial de Poisson como entrada de la red neuronal como se muestra en la Figura 1.2.

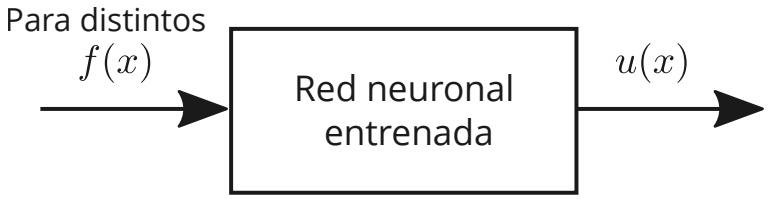

Figura 1.2: Interfaz propuesta para la arquitectura de red neuronal.

Para este trabajo, nos interesan las siguientes preguntas:

- £Cuál es la precisión del resultado de la red entrenada más allá del rango de parámetros de material considerados en la entrada?
- £Cómo se generaliza la red entrenada a otros problemas, por ejemplo, cambios en la geometría, topología, discretización, condiciones iniciales, etc.?
- £Cuáles son los márgenes de error y las respectivas tolerancias de los parámetros de entrada? Por ejemplo, £cuánto pueden variar los parámetros de entrada para un determinado margen de error?
- £Cuáles son los recursos computacionales utilizados?
- £Cuál es el costo de generar los datos de entrada y entrenar una red neuronal?
- £Cuántos datos son necesarios para obtener una solución razonablemente precisa de un problema?

2. JUSTIFICACIÓN

Las ecuaciones diferenciales modelan problemas físicos que representan muchos tipos de fenómenos de la naturaleza. Su solución implica problemas que requieren modelos matemáticos que no pueden resolverse con exactitud (on CSE Education, 2001). Esto hace necesario el uso de herramientas computacionales que faciliten los procesos de modelización, simulación y optimización. La modelización computacional ha crecido en desarrollos multidisciplinares en ciencia e ingeniería durante las últimas décadas (Langtangen y Mardal, 2019). Por su parte, la ciencia e ingeniería computacional se encarga de encontrar numéricamente una aproximación a la solución de dichas ecuaciones diferenciales parciales que rigen la física del sistema para una geometría dada, propiedades de los materiales, leyes constitutivas y condiciones iniciales y de frontera (Chen y cols., 2020).

En los últimos años, la disponibilidad de grandes conjuntos de datos, combinada con la mejora de los algoritmos y el crecimiento exponencial de la potencia de cálculo, ha provocado un aumento sin precedentes del interés por el tema del aprendizaje automático. Hoy en día, los algoritmos de aprendizaje automático se emplean con éxito para tareas de clasificación, regresión, agrupación o reducción de la dimensionalidad en grandes conjuntos de datos de entrada, especialmente los de alta dimensión (Schmidt, Marques, Botti, y Marques, 2019).

Desde la aparición de los métodos numéricos y el aprendizaje automático, ha habido un interés en la exploración de diferentes aplicaciones relativas a las redes neuronales y su aplicación en la resolución de ecuaciones diferenciales parciales. Dichos modelos requieren el aporte de distintas disciplinas que permitan la construcción de un modelo matemático y el establecimiento de algoritmos básicos para su tratamiento.

Este proyecto nace de la línea de Mecánica Computacional de la Escuela de Ciencias Aplicadas e Ingeniería de la Universidad EAFIT. La mecánica computacional combina la física y las matemáticas a través de métodos numéricos para resolver problemas de la mecánica relevantes en física e ingeniería (Oden, Belytschko, Babuska, y Hughes, 2003).

La ingeniería física propende por la interdisciplinariedad en la formación de los estudiantes. Este trabajo constituye un escenario en donde la computación científica y la modelación matemática se unen para brindar solución a un problema con aplicaciones en diversas áreas de la física e ingeniería. Por tanto, es una excusa para la comprensión y apropiación del problema físico de mi parte.

3. OBJETIVOS

3.1. General

Evaluar una prueba de concepto de una red neuronal para la solución de ecuaciones diferenciales.

3.2. Específicos

- Documentar el estado del arte para las redes neuronales como método de aproximación a las ecuaciones diferenciales.
- Proponer una red neuronal como método de solución para la ecuación diferencial de Poisson que conserve las entradas/salidas de los algoritmos tradicionales.
- Evaluar las capacidades de solución de la red neuronal mediante un diseño de experimentos computacional.
- Ajustar los parámetros de la red para reproducir el esquema tradicional de un método numérico en términos de entradas-salidas.
- Comparar los resultados obtenidos por la red neuronal con métodos tradicionales.

4. MARCO DE REFERENCIA

En esta sección se presenta los conceptos, términos y definiciones que se tuvieron en cuenta para la realización del trabajo de grado. Se aborda un marco conceptual del aprendizaje automático como subcampo de la inteligencia artificial y un marco conceptual para la arquitectura de las redes neuronales artificiales.

4.1. Inteligencia artificial

4.1.1. ¿Qué es la inteligencia artificial?

La inteligencia artificial (IA) es el área de estudio de la ciencia de la computación que se ocupa del desarrollo de ordenadores capaces de realizar procesos de pensamiento similares a los humanos, como el aprendizaje, el razonamiento y la autocorrección (Garg, 2021). En otras palabras, es la disciplina de la ciencia y la ingeniería que toma como objeto el conocimiento, lo adquiere, analiza y estudia los métodos de expresión de este para construir artefactos que puedan desarrollarse a partir de la experiencia para conseguir el efecto de simular las actividades intelectuales humanas.

Los primeros trabajos en el campo de la inteligencia artificial fueron desarrollados por Alan Mathison Turing a mediados del siglo XX. Considerado como uno de los pioneros de la inteligencia artificial y de la ciencia cognitiva moderna (Flasiński, 2016), Turning expuso como hipótesis que el cerebro humano es en gran parte una máquina de computación digital (Akman y Blackburn, 2000).

En 1935, Turing describió una máquina de computación abstracta que consistía en un escáner que se movía de un lado a otro de una memoria, símbolo a símbolo, leyendo lo que encontraba y escribiendo más símbolos. Las acciones del escáner están dictadas por un programa de instrucciones que también está almacenado en la memoria en forma de símbolos. Este es el concepto de programa almacenado de Turing, y en él está implícita la posibilidad de que la máquina opere sobre su propio programa y lo modifique o mejore (Cooper y Van Leeuwen, 2013).

Luego, en 1950 Turning propuso que la corteza cerebral es una "máquina" al nacer, y que a través del .[en]trenamiento" se organiza en una "máquina universal". Dicho enunciado se consideró en lo que posteriormente se conoció como el test de Turning como criterio para considerar que un ordenador artificial es ïnteligente" (Zhang y Lu, 2021).

En la Figura 4.1 se muestra algunas de las ramas que engloba la inteligencia artificial junto con ciertas de sus aplicaciones.

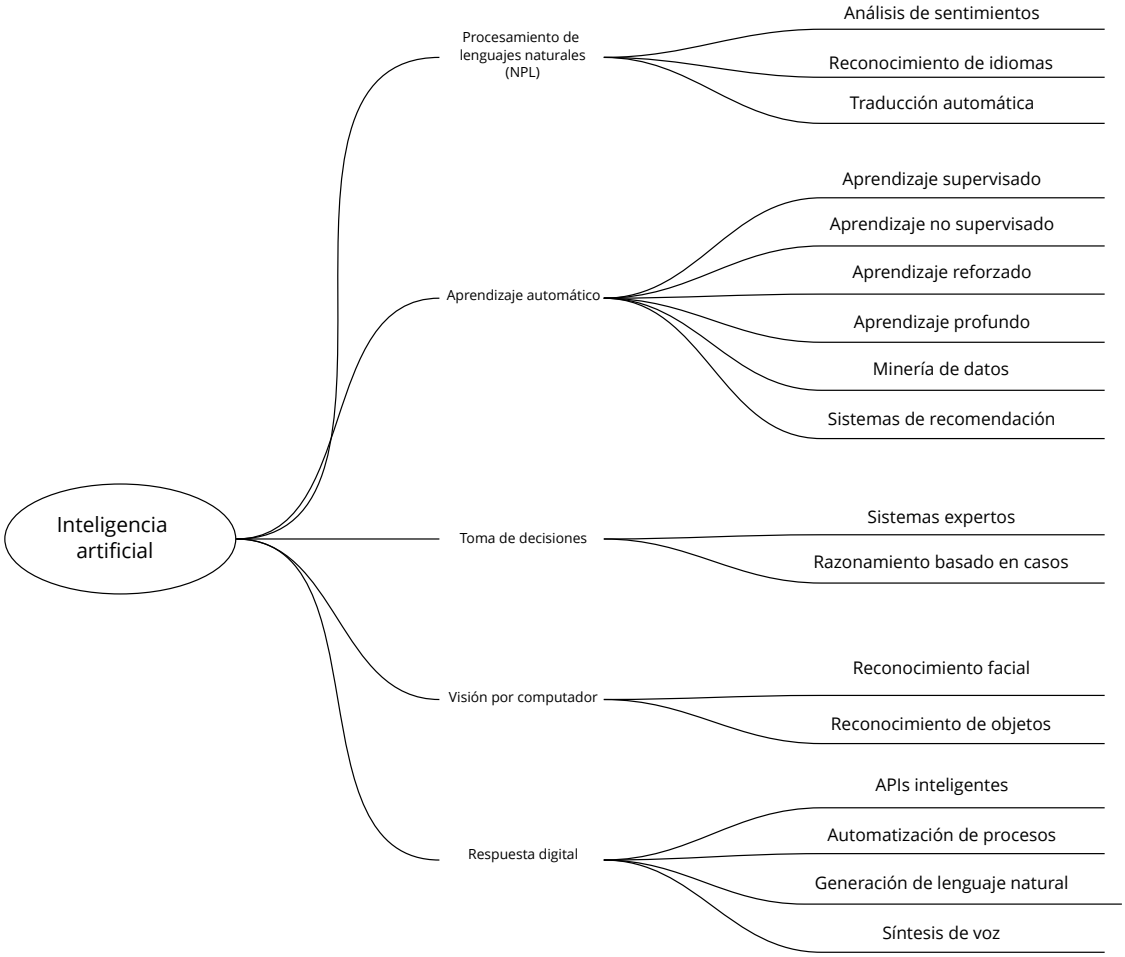

Figura 4.1: Aplicaciones y campos de la inteligencia artificial (Mondal, 2020)

4.1.2. Aprendizaje automático

El aprendizaje automático (del inglés, *machine learning*) es el estudio de los algoritmos informáticos capaces de aprender a mejorar su rendimiento en una tarea a partir de su propia experiencia previa. Su definición se remonta a 1959, cuando Arthur Samuel (Samuel, 1959) definió el *aprendizaje automático* como un campo de estudio que da a los

ordenadores la capacidad de aprender sin ser programados explícitamente (Marsland, 2011). En esta sección se aborda la perspectiva general del aprendizaje automático, junto con los diferentes tipos de algoritmos y aprendizajes, algunas características de dichos sistemas y un breve recuento histórico.

Perspectiva general

En los últimos años, la disponibilidad de grandes conjuntos de datos, combinada con la mejora de los algoritmos y el crecimiento exponencial de la potencia de cálculo, ha provocado un aumento sin precedentes del interés por el tema del aprendizaje automático (Schmidt y cols., 2019). Enfoques de aprendizaje como la agrupación de datos, los clasificadores de redes neuronales y la regresión no lineal han encontrado una aplicación sorprendentemente amplia en la práctica de la ingeniería, la industria y la ciencia.

En la Figura 4.2 se presenta una relación del aprendizaje automático como subcampo de la inteligencia artificial, y su vínculo con otras áreas de estudio de la ciencia computacional.

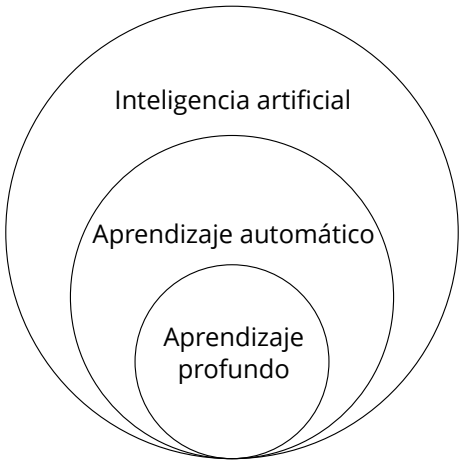

Figura 4.2: Relación entre la inteligencia artificial, el aprendizaje automático y el aprendizaje profundo.

Los algoritmos de aprendizaje automático se dividen comunmente en tres grandes categorías en función de la salida que dispone el sistema de aprendizaje: *aprendizaje supervisado*, *aprendizaje no supervisado* y *aprendizaje por refuerzo*.

Aprendizaje supervisado

El aprendizaje supervisado se refiere al uso de datos etiquetados para entrenar, con el fin de predecir el tipo o el valor de los nuevos datos. Las técnicas de aprendizaje supervisado implican una fase de entrenamiento y una fase de prueba donde el conjunto

de datos original se divide en dos partes: el conjunto de datos de entrenamiento y el conjunto de datos de prueba. En la fase de entrenamiento, las muestras de los datos de entrenamiento se toman como entrada en la que las características son aprendidas por el algoritmo de aprendizaje y construyen el modelo de aprendizaje (Solanki, Dhankar, y cols., 2017). En el proceso de prueba, el modelo utiliza el motor de ejecución para hacer la predicción de los datos de prueba. Los datos etiquetados son la salida del modelo de aprendizaje que da la predicción final o los datos clasificados.

Como se muestra en la Figura 4.3, una vez que la máquina está entrenada, el conjunto de datos de prueba se utiliza para comprobar el rendimiento y la precisión de la máquina.

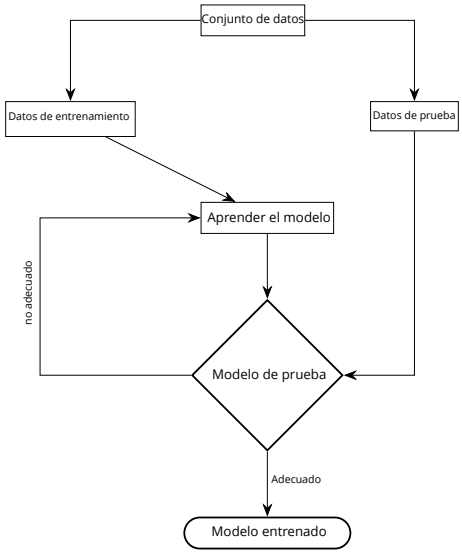

Figura 4.3: Representación del diagrama de flujo del aprendizaje supervisado (Mondal, 2020).

En los algoritmos supervisados, las clases están predeterminadas. Estas clases se crean en forma de conjunto finito, definido por el usuario, lo que en la práctica significa que un determinado segmento de datos será etiquetado con estas clasificaciones. La tarea del algoritmo de aprendizaje automático es encontrar patrones y construir modelos matemáticos(Nasteski, 2017).

Según los diferentes resultados de predicción, puede dividirse en dos categorías: clasificación y regresión. El problema de regresión se refiere a la predicción de la salida de valores continuos, se podrían analizar los datos del precio de la vivienda, ajustarlos según la entrada de datos de la muestra, y luego obtener una curva continua para predecir los precios de la vivienda (Zhang y Lu, 2021). El problema de clasificación consiste en la predicción de la salida de valores discretos, como juzgar si la foto actual es un perro o un gato basándose en una serie de características; el valor de salida es 1 o 0 (Morocho-Cayamcela, Lee, y Lim, 2019). En resumen, los modelos de regresión mapean el espacio de entrada en un dominio de valores reales, mientras que los clasificadores

mapean el espacio de entrada en clases predefinidas (Singh, Thakur, y Sharma, 2016).

Aprendizaje no supervisado

El aprendizaje no supervisado se refleja principalmente en la agrupación, esto es, los datos pueden clasificarse según diferentes características sin etiquetas . Cuando los datos históricos no están bien etiquetados y el número de clases posibles no se conoce con precisión, se utiliza el aprendizaje no supervisado para agrupar el conjunto de datos en *clusters* (Usama y cols., 2019). Se utiliza ampliamente en la segmentación de mercados, motores de recomendación, análisis de redes sociales, imágenes médicas, segmentación de imágenes, agrupación de resultados de búsqueda, etc (Mondal, 2020).

Los métodos típicos de aprendizaje no supervisado son el k-medias y el análisis de componentes principales. La premisa importante del *k-medias* es que la diferencia entre los datos puede medirse con base a la distancia euclidiana. El análisis de componentes principales es un método estadístico. Mediante una transformación ortogonal, las variables relevantes se transforman en variables no correlacionadas; las variables transformadas se denominan componentes principales. La idea básica es sustituir los indicadores originales relacionados por un conjunto de indicadores independientes y completos (James, Witten, Hastie, y Tibshirani, 2021).

Aprendizaje por refuerzo

El aprendizaje por refuerzo es un área del aprendizaje automático que se ocupa de cómo los agentes de software deben realizar acciones en un entorno para maximizar alguna noción de recompensa acumulada. El aprendizaje por refuerzo se refiere a los algoritmos orientados a objetivos, que aprenden a alcanzar un objetivo complejo o a maximizar a lo largo de una dimensión particular a lo largo de muchos pasos; por ejemplo, incrementar los puntos ganados en un juego a lo largo de muchas jugadas (Mahadevan, 1996).

Por lo tanto, el aprendizaje por refuerzo es un método para obtener recompensas interactuando con el entorno, juzgando la calidad de las acciones por los niveles de recompensa, y luego entrenando el modelo. En otras palabras, el aprendizaje por refuerzo puede mejorar el comportamiento de recompensa y debilitar el de castigo. Se puede entrenar el modelo mediante el mecanismo de prueba y error para encontrar el mejor funcionamiento y comportamiento para obtener el mayor rendimiento (Collins y Moons, 2019).

En la Figura 4.4 se muestra el proceso de acción y reformulación de un agente simple para el aprendizaje por refuerzo.

En este caso, a diferencia del aprendizaje supervisado, no se proporcionarán datos etiquetados, sino que el agente aprenderá interactuando con el entorno. En este aprendizaje

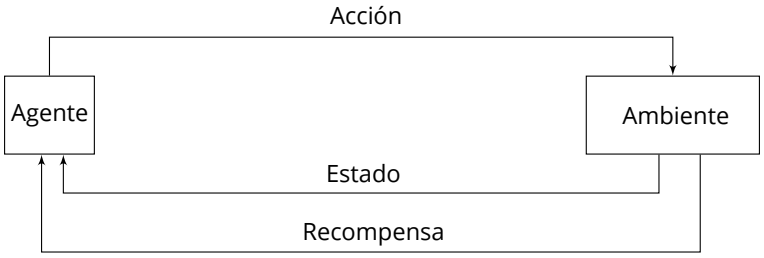

Figura 4.4: Representación de un agente simple para el aprendizaje por refuerzo (Sutton y Barto, 2018).

interviene un crítico o función de valor que recompensa o penaliza al agente por cada uno de sus movimientos. El agente intenta obtener una recompensa alta y reducir las pérdidas cada vez (Sutton y Barto, 2018).

Característica de los sistemas de aprendizaje automático

Para la realización de algún tipo de algoritmo de aprendizaje automático, se debe de considerar ciertas especificaciones del diseño de dicho algoritmo (Vogelsang y Borg, 2019). Los requisitos especificados para un sistema de aprendizaje automático varían según el caso de uso. Sin embargo, la mayoría de los sistemas deben tener estas cuatro características generales (Huyen, 2022): fiabilidad, escalabilidad, capacidad de mantenimiento y adaptabilidad.

- **Fiabilidad:** el sistema debe seguir realizando la función correcta con el nivel de rendimiento deseado, incluso ante la adversidad (fallos de hardware o software, e incluso errores humanos). La aplicación de técnicas de aprendizaje automático para la predicción de la fiabilidad del software ha mostrado resultados meticulosos y notables (Jaiswal y Malhotra, 2018).
- **Escalabilidad:** sea cual sea la forma en que crezca el sistema (complejidad, cantidad de datos,aumento de velocidad dado el número de nodos, etc.), debe haber formas razonables de gestionar ese crecimiento. Cuando se habla de escalabilidad, la mayoría de la gente piensa en el escalado de recursos, que consiste en aumentar la escala (ampliar los recursos para manejar el crecimiento) y reducir la escala (reducir los recursos cuando no se necesitan) (Ulanov, Simanovsky, y Marwah, 2017).
- **Mantenibilidad:** la configuración de los algoritmos de aprendizaje automático deben de propiciar que los modelos sean lo suficientemente reproducibles para que distintos colaboradores puedan tener suficientes contextos para construir sobre el mismo. Es por esto que debe de existir documentación de todo el proceso. El código, los datos y los artefactos deben estar versionados. En conclusión, los

sistemas de aprendizaje automático y sus características se deben abordar a la luz del mantenimiento del software y sus atributos, la modularidad, la comprobabilidad, la reutilización, la analizabilidad y la modificabilidad (Mikkonen y cols., 2021).

- **Adaptabilidad:** para adaptarse a los cambios en la distribución de los datos, el sistema debe tener cierta capacidad tanto para descubrir aspectos para mejorar el rendimiento como para permitir las actualizaciones.

Breve recuento histórico

El interés por los enfoques computacionales del aprendizaje se remonta a los inicios de la inteligencia artificial y la ciencia cognitiva a mediados de la década de 1950 (Miller, 2003). La diversidad de tareas y métodos caracterizó la investigación desde el principio, con trabajos que incluían temas como el juego, el reconocimiento de palabras, los conceptos abstractos y la memoria verbal. El aprendizaje se consideraba una característica central de los sistemas inteligentes, y los trabajos sobre el aprendizaje y el rendimiento se centraban en el desarrollo de mecanismos generales de cognición, percepción y acción (Langley, 1997).

A mediados de los años 60, tanto los investigadores de inteligencia artificial como los investigadores de neurociencia se dieron cuenta de la importancia del conocimiento del dominio, lo que llevó a la construcción de los primeros sistemas de conocimiento intensivo. Sin embargo, los investigadores del aprendizaje siguieron centrándose en métodos generales e independientes del dominio, y la mayoría de los trabajos se aplicaron a dominios perceptivos (Newell, 1982). Con el tiempo, el reconocimiento de patrones y la inteligencia artificial se separaron en dos campos distintos. La brecha se amplió aún más cuando muchos investigadores del reconocimiento de patrones empezaron a hacer hincapié en métodos algorítmicos y numéricos que contrastaban fuertemente con los métodos heurísticos y simbólicos asociados al paradigma de la inteligencia artificial (Minsky, 1961).

A finales de la década de 1970, surgió un nuevo interés por el aprendizaje automático, que creció rápidamente en el transcurso de unos años. Algunas investigaciones en este ámbito estaban motivadas por la perspectiva de automatizar la adquisición de bases de conocimiento específicas del dominio, y otros esperaban modelar el aprendizaje humano (Carbonell, Michalski, y Mitchell, 1983b). La investigación sobre la inducción de conceptos y la adquisición del lenguaje continuó, pero a ella se sumaron los trabajos sobre el descubrimiento de máquinas y el aprendizaje en la resolución de problemas. Se propusieron muchos métodos nuevos y surgió un renovado interés por las redes neuronales (Michalski y Kodratoff, 1990).

Aunque la investigación sobre el aprendizaje automático ha tenido lugar desde los primeros días de la inteligencia artificial y la ciencia cognitiva, sólo se ha considerado

como un campo distinto desde aproximadamente 1980. En dicha época, los trabajos se extendieron a la planificación, el diagnóstico, el diseño y el control. En el mismo año se celebró el primer taller sobre aprendizaje automático en la Universidad Carnegie Mellon en 1980 (Michalski, Carbonell, y Mitchell, 2013).

Muchos de los conceptos centrales del aprendizaje automático tienen una larga historia. Hume (1748) describe el problema de la inducción, y las teorías del aprendizaje han ocupado un lugar central en la psicología durante muchas décadas (Langley y Simon, 1981; Bower, 1981). Simon y Lea (1974 (Simon y Lea, 1974)) caracterizaron por primera vez la inducción de reglas en términos de búsqueda heurística. Mitchell (1980 (Mitchell, 1980)) introdujo la noción de sesgo inductivo, mientras que Rendell (1986 (Rendell, 1986)) distinguió entre sesgo representacional y de búsqueda. El problema del sobreajuste ha preocupado a los estadísticos durante muchos años, pero Breiman, Friedman, Olshen y Stone (1984 (Breiman, Friedman, Olshen, y Stone, 1984)) y Quinlan (1986 (Quinlan, 1986)) introdujeron esta cuestión en la comunidad del aprendizaje automático (Carbonell, Michalski, y Mitchell, 1983a).

Las publicaciones mencionadas hacen hincapié en los enfoques empíricos del estudio del aprendizaje, pero también contienen algunos trabajos desde la perspectiva psicológica y matemática (Gureckis y Markant, 2012). Anderson (1981 (Anderson, 1981)), Rumelhart y McClelland (1986 (Rumelhart, Hinton, McClelland, y cols., 1986)) y Klahr, Langley y Neches (1987 (Klahr, Langley, Neches, y Neches, 1987)) contienen capítulos sobre modelos computacionales del aprendizaje humano, mientras que los textos sobre teoría del aprendizaje computacional proceden de Kearns y Vazirani (1994 (Kearns y Vazirani, 1994)).

Desde entonces, los investigadores se han dedicado a estudiar sobre el potencial de los algoritmos de aprendizaje en el mundo real, y una serie de aplicaciones exitosas demostraron que la tecnología podía tener un impacto en la industria. El campo también se asentó sobre bases metodológicas mucho más firmes, y la experimentación sistemática con conjuntos de datos comunes y el análisis teórico preciso. También se extendió por la comunidad una variedad de paquetes de software, lo que dio lugar a cuidadosos estudios comparativos y a una mayor tendencia a basarse en sistemas anteriores (Hirschheim, Klein, y Lyytinen, 1995).

El campo ha crecido rápidamente desde entonces, y ahora cuenta con volúmenes recopilados sobre temas generales y especializados. El número de artículos sobre aprendizaje publicados en actas de congresos y revistas ha aumentado, y el número de investigadores de nivel de doctorado en el área sigue creciendo (Enders*, 2004). El aprendizaje automático se ha convertido en una de las principales preocupaciones de las comunidades de inteligencia artificial y ciencias cognitivas, ya que la mayoría de los investigadores han reconocido el papel central del aprendizaje en la inteligencia (Español, 2021). Un análisis bibliométrico de las tendencias de investigación del aprendizaje automático en ingeniería muestra que en las dos últimas décadas han surgido continuamente numerosos estudios con una tasa media de crecimiento anual del 24,3 % (Su, Peng, y Li, 2021).

En la Figura 4.1.2 muestra los artículos publicados en revistas de alto impacto que han habido desde el 2000. [1].

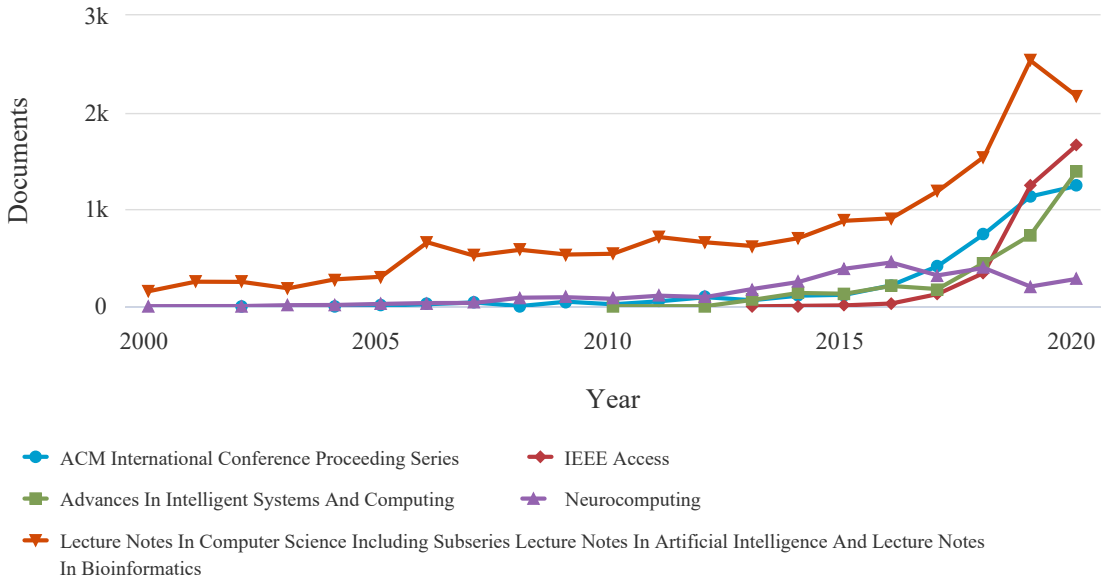

Figura 4.5: Resultados de la búsqueda de artículos por año según la revista (Su y cols., 2021).

La bibliometría muestra las expectativas sobre el impacto del aprendizaje automático en la comunidad científica. El aumento continuo de las publicaciones relacionadas con el aprendizaje automático indica que se trata de una disciplina de investigación activa según (Dietterich, 1997).

4.2. Redes neuronales

Las redes neuronales artificiales se refieren a modelos de datos (Jain, Mao, y Mohiuddin, 1996) que replican la función de las redes neuronales biológicas. En general, las redes neuronales son métodos autoadaptativos y no lineales que no requieren suposiciones específicas sobre el modelo subyacente (Basu, Bhattacharyya, y hoon Kim, 2010). Al igual que la red neuronal biológica, las redes neuronales artificiales tienen una interconexión de nodos, análogos a las neuronas. Cada red neuronal tiene tres componentes: el carácter del nodo, la topología de la red y las reglas de aprendizaje. El carácter de los nodos determina la forma en que las señales son procesadas por el nodo, como el número

[1]Obtenido de Scopus con TITLE-ABS-KEY ('machine AND learning') AND PUBYEAR>2000 AND PUBYEAR<2020. Buscado el 11 de febrero, 2022

de entradas y salidas asociadas al nodo, el peso asociado a cada entrada y salida, y la función de activación. La topología de la red determina la forma en que se organizan y conectan los nodos.

En la Figura 4.6 se muestra un esquema de la arquitectura de un modelo de red neuronal. Consiste en un conjunto de entradas y salidas a través de una función sobre un espacio de alta dimensión (Dave y Dutta, 2012).

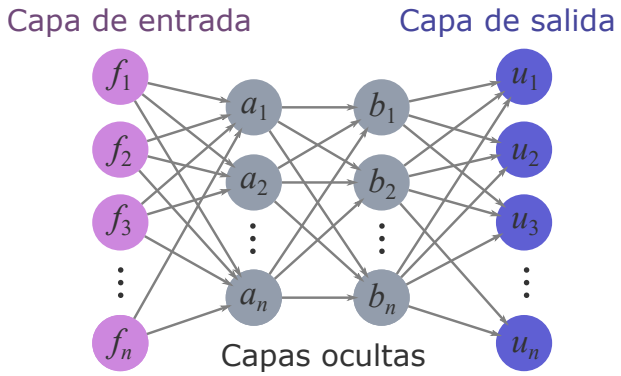

Figura 4.6: Esquema de una arquitectura sencilla de una red neuronal.

El gran potencial de las redes neuronales es la alta velocidad de procesamiento que proporcionan en una implementación paralela masiva y la posibilidad de evaluar un problema en términos de factores de análisis de datos (por ejemplo, precisión, velocidad de procesamiento y escalabilidad) (Mozaffari, Emami, y Fathi, 2018).

De hecho, se han investigado varias redes neuronales como enfoque de aprendizaje no supervisado, y se han estudiado diversas arquitecturas en función de sus tareas y aplicaciones. Brevemente, hablamos de aprendizaje supervisado, no supervisado y reforzado.

Las redes neuronales supervisadas se utilizan popularmente para abordar tareas de clasificación. Éstas abordan problemas como la detección de spam, el reconocimiento de objetos y las proyecciones, por ejemplo, de los ingresos por ventas de un determinado negocio (He y cols., 2017). A diferencia del aprendizaje supervisado, el aprendizaje no supervisado utiliza datos no etiquetados. A partir de esos datos, descubre patrones que ayudan a resolver problemas de agrupación o asociación. Esto es especialmente útil cuando se desconoce el conjunto de datos de entrada (Bröker, Love, y Dayan, 2022). En cambio, el aprendizaje reforzado permite a un agente aprender en un entorno interactivo por ensayo y error utilizando la retroalimentación de sus propias acciones y experiencias (Botvinick y cols., 2019).

Para entender la arquitectura de una red neuronal, es importante tener en cuenta los siguientes términos

- **Época**: el número de veces que el algoritmo se ejecuta en todo el conjunto de datos de entrenamiento.

- **Muestra**: una sola fila de un conjunto de datos.
- **Lote (o *Batch*)**: denota el número de muestras que se tomarán para actualizar los parámetros del modelo.
- **Tasa de aprendizaje**: es un parámetro que proporciona al modelo una escala de cuánto deben actualizarse los pesos del modelo.
- **Función de coste/función de pérdida**: una función de coste se utiliza para calcular el coste que es la diferencia entre el valor predicho y el valor real.
- **Pesos/ Sesgo**: los parámetros aprendibles en un modelo que controla la señal entre dos neuronas.

En la Figura 4.7 se muestra un esquema básico de un nodo simple dentro de una red neuronal.

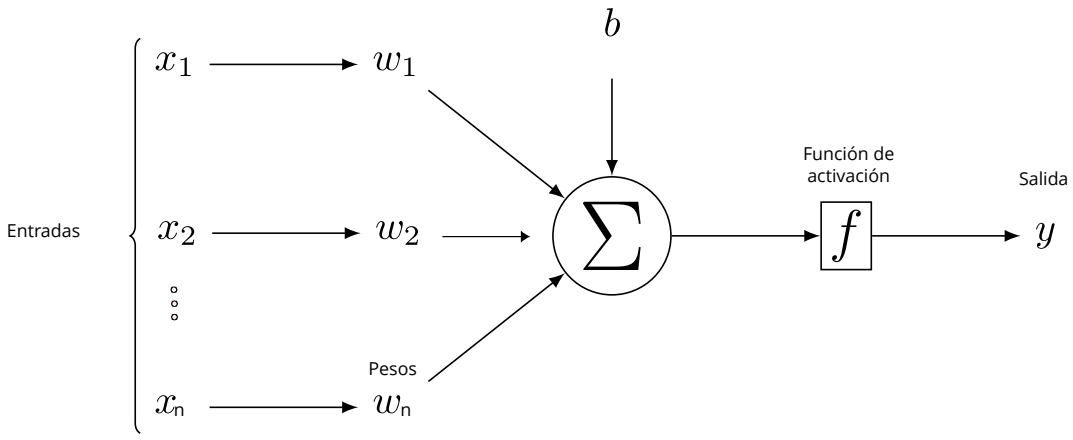

Figura 4.7: Modelo básico de un nodo simple: $x_i = entrada, w_i = peso, f =$ función de activación, $y = salida$.

Tal y como se mencionó en la anterior sección sobre la arquitectura de una red neuronal, los nodos se organizan en arreglos lineales, llamadas capas. Normalmente, hay capas de entrada, capas de salida y capas ocultas. Puede no haber ninguna o varias capas ocultas. El diseño de la topología de la red implica la determinación del número de nodos en cada capa, el número de capas en la red y el recorrido de las conexiones entre los nodos. Normalmente, esos factores se fijan inicialmente por intuición y se optimizan mediante múltiples ciclos de experimentos. También se pueden utilizar algunos métodos para diseñar una red neuronal (Zou, Han, y So, 2008).

Hay dos tipos de conexiones entre nodos. Una es una conexión unidireccional sin bucle de retorno. La otra es una conexión de bucle de retorno en la que la salida de los nodos puede ser la entrada de los nodos anteriores o del mismo nivel. Basándose en el tipo de conexiones mencionadas, las redes neuronales pueden clasificarse en dos tipos: red de

avance y red de retroalimentación. Dado que la señal viaja en un solo sentido, la red de avance es estática; es decir, una entrada está asociada a una salida concreta. La red de retroalimentación es dinámica. Para una entrada, el estado de la red de retroalimentación cambia durante muchos ciclos, hasta que alcanza un punto de equilibrio, por lo que una entrada produce una serie de salidas. El perceptrón es una red de avance muy utilizada (Rosa, Guerra, Horta, Martins, y Lourenço, 2020).

4.2.1. Perceptrón multicapa

En la Figura 4.8 se muestra el esquema básico de una arquitectura de perceptrón multicapa (MLP).

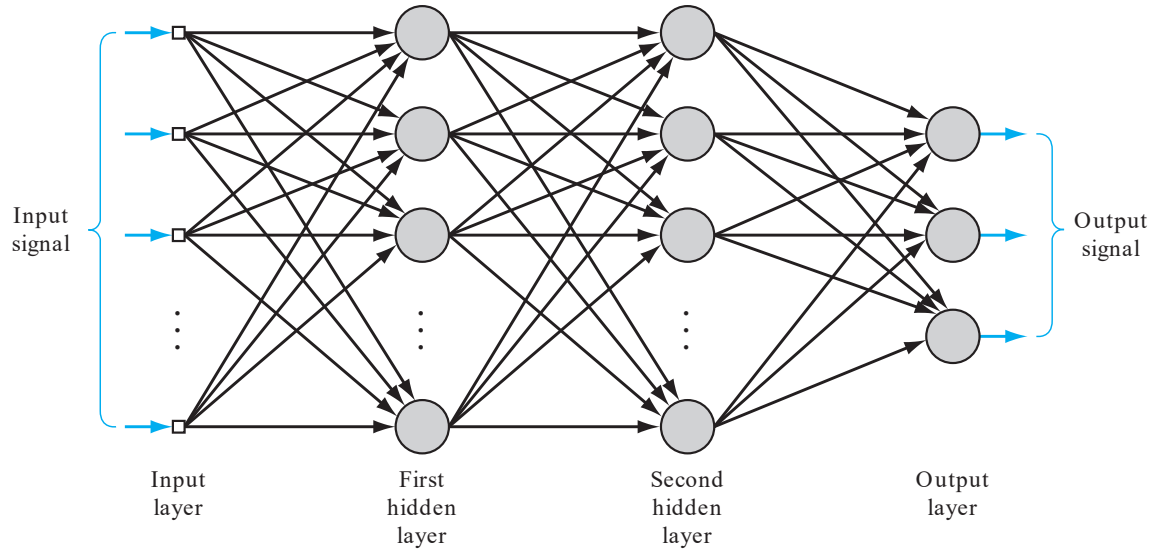

Figura 4.8: Esquema de un perceptrón multicapa con dos capas ocultas (Haykin, 2009).

Dentro de las características generales en las redes neuronales de tipo perceptrón multicapa se encuentran las siguientes propiedades (Goodfellow, Bengio, y Courville, 2016; Rumelhart, Hinton, y Williams, 1986):

- El modelo de cada neurona en la red incluye una función de activación que es diferenciable. Aunque las funciones de activación no lineales son comunes, también se pueden emplear funciones de activación lineales en ciertos casos, como en el presente estudio.
- La red está compuesta por una capa de entrada, una o varias capas ocultas y una capa de salida. Las capas ocultas se definen como aquellas que no están directamente conectadas con las entradas ni con las salidas de la red.

- La red exhibe un alto grado de conectividad, cuyo alcance es determinado por los pesos sinápticos de la red [2].

Un perceptrón se define como una neurona artificial que toma un conjunto de entradas $x \in \mathbb{R}^n$ y produce una sola salida $y \in \mathbb{R}$ mediante un proceso que implica la suma ponderada de las entradas, el sesgo y una función de activación. Aunque las funciones de activación no lineales diferenciables son comunes, también se pueden emplear funciones de activación lineales en ciertos casos, como en el presente estudio. Cabe mencionar que esta definición hace referencia al comportamiento de una única neurona en la red neuronal, y en el caso de una red neuronal con múltiples salidas, la salida final pertenecerá a $\mathbb{R}^m$. Un perceptrón logra esto aplicando tres pasos fundamentales a las entradas (Goodfellow y cols., 2016):

- Suma el producto de cada entrada $x_i \in x$ con un peso correspondiente $w_i \in w$. Cada peso w_i se puede interpretar como una medida de cuán sensible es un perceptrón a cada entrada individual x_i. Por ejemplo, si $w_i = 0$, entonces la entrada x_i no tiene influencia en la activación del perceptrón.
- Añade un sesgo β a esta suma. Este sesgo se puede interpretar como una medida de cuán activa sería una neurona si $x_i = 0$ para $i = 1, \ldots, n$.
- Pasa este valor a través de una función de activación $\sigma : \mathbb{R} \to [-1, 1]$.

La activación de un perceptrón a se puede representar como

$$a = \sigma\left(\sum_{i=1}^{n} x_i w_i + \beta\right) \tag{4.1}$$

4.2.2. Algoritmo de retropropagación

El algoritmo de retropropagación es un método utilizado comúnmente para entrenar redes neuronales clásicas. Se basa en el método del gradiente descendente y la optimización de una función de pérdida con respecto a los pesos de la red. El algoritmo de retropropagación consta de dos fases: propagación hacia adelante y propagación hacia atrás (Rumelhart, Hinton, y Williams, 1986).

Durante la propagación hacia adelante, la red neuronal recibe una entrada y genera una salida mediante el cálculo de las activaciones de todas las neuronas en cada capa, desde la capa de entrada hasta la capa de salida. La función de pérdida, que mide la discrepancia entre la salida de la red y las salidas deseadas, se evalúa utilizando la salida generada.

[2] Las neuronas pueden conectarse de muchas formas diferentes para crear tipos especiales de redes (por ejemplo, redes convolucionales o recurrentes). La Figura 4.8 es un ejemplo de una red neuronal totalmente conectada. Para simplificar, consideraremos que las redes están totalmente conectadas, a menos que se especifique lo contrario.

En la fase de propagación hacia atrás, se calculan las derivadas parciales de la función de pérdida con respecto a cada peso y sesgo de la red utilizando la regla de la cadena. Estas derivadas parciales se utilizan para actualizar los pesos y sesgos de la red a través del método del gradiente descendente.

El proceso de retropropagación se itera hasta que se alcance un criterio de convergencia definido, como un error de entrenamiento mínimo o una cantidad máxima de épocas de entrenamiento.

4.2.3. Teorema de aproximación universal

Un perceptrón multicapa entrenado con el algoritmo de retropropagación puede verse como una herramienta práctica para realizar un mapeo de entrada-salida no lineal.

Sea m_0 el número de nodos de entrada (fuente) de un perceptrón multicapa y sea $M = m_L$ el número de neuronas en la capa de salida de la red. La relación entrada-salida de la red define un mapeo desde un espacio de entrada euclidiano m_0-dimensional a un espacio de salida euclidiano M-dimensional, que es continuamente diferenciable infinitamente cuando la función de activación también lo es (Cybenko, 1989a).

Al evaluar la capacidad del perceptrón multicapa desde este punto de vista del mapeo entrada-salida, surge la siguiente pregunta fundamental: £Cuál es el número mínimo de capas ocultas en un perceptrón multicapa con un mapeo entrada-salida que proporciona una aproximación de cualquier mapeo continuo?

Para responder a esta pregunta, se establece el teorema de aproximación universal para un mapeo no lineal de entrada-salida que se enuncia en (Haykin, 2009).

En (Hornik, 1991), Hornik presenta una prueba del teorema de aproximación universal para perceptrones multicapa y establece que una red neuronal de una sola capa oculta puede aproximar cualquier función continua en un dominio compacto, siempre que la función de activación en la capa oculta sea una función no constante, acotada y continuamente diferenciable.

Además, en (Leshno, Lin, Pinkus, y Schocken, 1993), Leshno et al. muestran que las redes neuronales multicapa con una función de activación no polinómica también pueden aproximar cualquier función continua, lo que amplía el alcance del teorema de aproximación universal.

Es importante destacar que el teorema de aproximación universal no proporciona una guía sobre la cantidad de neuronas en la capa oculta necesarias para lograr una aproximación precisa, ni proporciona una estrategia para entrenar la red para que alcance esta aproximación. Estos problemas dependen de la función específica que se desea aproximar y de la arquitectura de la red neuronal.

El teorema de aproximación universal tiene importantes implicaciones en la teoría y la práctica de las redes neuronales. En primer lugar, indica que las redes neuronales multicapa son capaces, en principio, de aproximar cualquier función continua, lo que sugiere que las redes neuronales pueden ser utilizadas en una amplia gama de aplicaciones.

En segundo lugar, este resultado resalta la importancia crucial de las funciones de activación no lineales para que las redes neuronales sean capaces de aproximar de manera efectiva funciones complejas y no lineales. Las funciones de activación no lineales permiten a las redes neuronales capturar y modelar la complejidad inherente en muchas funciones continuas.

En tercer lugar, aunque el teorema de aproximación universal establece que las redes neuronales con una sola capa oculta son suficientes para aproximar cualquier función continua, en la práctica, las redes neuronales con múltiples capas ocultas pueden ser más eficientes en términos de la cantidad de neuronas requeridas y la facilidad de entrenamiento. Las redes neuronales profundas, que contienen múltiples capas ocultas, han demostrado ser capaces de modelar y aproximar con éxito funciones altamente complejas y no lineales en una amplia gama de aplicaciones, como el reconocimiento de imágenes, el procesamiento del lenguaje natural y la predicción de series temporales (Goodfellow y cols., 2016).

4.2.4. Función de pérdida

La función de pérdida, también conocida como función de costo o función objetivo, es una medida cuantitativa de la discrepancia entre las salidas deseadas y las salidas generadas por la red neuronal (Goodfellow y cols., 2016). El objetivo del entrenamiento de una red neuronal es minimizar la función de pérdida con respecto a los pesos y sesgos de la red. Las funciones de pérdida más comunes en el aprendizaje supervisado son la pérdida cuadrática media (MSE) (Bishop y Nasrabadi, 2006).

- **Pérdida cuadrática media (MSE):** La pérdida cuadrática media es una función de pérdida comúnmente utilizada en problemas de regresión. Se define como el promedio de las diferencias al cuadrado entre las salidas deseadas y las salidas generadas por la red:

$$L_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{4.2}$$

  donde y_i es la salida deseada, $\hat{y}_i$ es la salida generada por la red y N es el número de ejemplos de entrenamiento.

4.2.5. Función de activación

La función de activación es una función no lineal aplicada a la suma ponderada de las entradas y el sesgo en cada neurona de la red. La no linealidad introducida por las funciones de activación es esencial para que las redes neuronales puedan aproximar funciones no lineales y complejas (Goodfellow y cols., 2016). Algunas funciones de activación comunes incluyen:

- **Sigmoide:** La función sigmoide, también conocida como función logística, transforma la entrada en un valor en el rango (0, 1):

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{4.3}$$

  Esta función fue ampliamente utilizada en las primeras redes neuronales, pero actualmente es menos común debido a problemas como el degradado del gradiente y la saturación de las neuronas (Glorot y Bengio, 2010).

- **Tangente hiperbólica (tanh):** La función tangente hiperbólica transforma la entrada en un valor en el rango (-1, 1):

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}. \tag{4.4}$$

  La función tanh es similar a la función sigmoide, pero su rango es simétrico en torno al origen, lo que la hace más adecuada para ciertas aplicaciones (Bottou, 2012).

- **Rectificador lineal (ReLU):** La función rectificador lineal (ReLU) es actualmente una de las funciones de activación más populares en el aprendizaje profundo. Se define como:

$$\text{ReLU}(x) = \text{máx}(0, x). \tag{4.5}$$

  La función ReLU es fácil de calcular, no sufre de degradado del gradiente en el rango positivo y es capaz de introducir no linealidad y esparsidad en las representaciones aprendidas, lo que la convierte en una opción eficiente y efectiva para muchas aplicaciones (Kim y Cho, 2019).

4.2.6. Optimizador

El optimizador es el algoritmo utilizado para actualizar los pesos y sesgos de la red neuronal con el fin de minimizar la función de pérdida. El algoritmo más simple y ampliamente utilizado es el descenso de gradiente estocástico (SGD), pero existen muchas variantes y mejoras que pueden acelerar la convergencia y mejorar la calidad de las soluciones encontradas (Ruder, 2016).

- **Descenso de gradiente estocástico (SGD):** El descenso de gradiente estocástico es una variante del descenso de gradiente que utiliza un subconjunto de ejemplos de entrenamiento (minibatch) en lugar del conjunto completo para calcular el gradiente de la función de pérdida (Robbins y Monro, 1951). El algoritmo actualiza los pesos y sesgos de la red en función de la tasa de aprendizaje y el gradiente de la función de pérdida con respecto a los parámetros:

$$w_{t+1} = w_t - \eta \nabla L(w_t), \tag{4.6}$$

  donde w_t son los parámetros de la red en el tiempo t, η es la tasa de aprendizaje y $\nabla L(w_t)$ es el gradiente de la función de pérdida con respecto a los parámetros en el tiempo t.

- **Momento:** El momento es una técnica que se puede utilizar junto con el descenso de gradiente estocástico y otros métodos de optimización para acelerar la convergencia y mejorar la estabilidad del algoritmo (Sutskever, Martens, Dahl, y Hinton, 2013). El momento funciona al agregar una fracción del vector de actualización del paso anterior al vector de actualización actual, lo que ayuda a suavizar las oscilaciones y acelera la convergencia en direcciones con gradientes más constantes. La actualización con momento se realiza de la siguiente manera:

$$v_{t+1} = \mu v_t + \eta \nabla L(w_t), \tag{4.7}$$

$$w_{t+1} = w_t - v_{t+1}, \tag{4.8}$$

  donde v_t es el vector de momento en el tiempo t, y μ es el coeficiente de momento, que típicamente tiene un valor en el rango [0.5, 0.99].

4.2.7. Redes neuronales informadas por la física

En 2017, Raissi introduce la red neuronal informada por la física como una alternativa para inferir soluciones de ecuaciones diferenciales parciales (Raissi, Perdikaris, y Karniadakis, 2017). Dichas redes son técnicas utilizadas para resolver problemas mediante el entrenamiento de una red neuronal para minimizar una función de pérdida.

Como algoritmo alternativo, las redes neuronales informadas por la física utilizan una metodología de aprendizaje supervisado para encontrar las propiedades físicas. Se componen de términos residuales de ecuaciones diferenciales (pérdidas), así como de condiciones iniciales y de frontera. Las entradas (variables) de la red se transforman en salidas de la red. A continuación, toma el campo de salida y calcula la derivada utilizando las ecuaciones dadas (Cuomo y cols., 2022).

Cuando hablamos de un problema en ingeniería, podemos referirnos a dos tipos de enfoques. Un problema directo consiste en predecir el resultado de las mediciones a partir de un marco teórico específico. Proporciona una descripción completa de un

sistema. Por otro lado, los problemas inversos consisten en utilizar el resultado real de unas mediciones para inferir los valores de los parámetros que caracterizan el sistema (De Vito, Rosasco, Caponnetto, De Giovannini, y Odone, 2005).

Las redes neuronales informadas por la física, a diferencia de los enfoques estándar, resuelven problemas directos e inversos en una gran variedad de temas de ingeniería y ciencia. Estos problemas inversos pueden encontrarse en una gran variedad de aplicaciones de ingeniería y ciencia. Estos problemas tienen que ver con la determinación de la entrada y las características de un sistema físico a partir de algunas de las salidas del sistema. Desde el punto de vista matemático, estos problemas están mal planteados y deben superarse mediante el desarrollo de nuevos sistemas informáticos, enfoques de regularización y metodologías experimentales. A continuación, se presentan algunos ejemplos:

Algunos aplicaciones de redes informadas por la física

En esta subsección, presentamos algunos ejemplos en los que las redes neurales se utilizan en campos de la ciencia y la ingeniería. En primer lugar, se presentan algunos ejemplos de las capacidades de las redes neuronales en problemas inversos en un contexto científico. También, un paquete de software para entrenar PINNs.

- **Redes neuronales informadas por la física para problemas inversos en nano-óptica**

  En (Chen y cols., 2020), los autores emplean los PINN para la solución de problemas representativos de dispersión inversa en metamateriales fotónicos y tecnologías nanoópticas. Este trabajo muestra el uso efectivo de un PINN sin malla para recuperar las características de permitividad efectiva de una serie de sistemas de dispersión de tamaño finito que implican múltiples nanoestructuras que interactúan y nanopartículas multicomponentes.

  Las aportaciones de este trabajo son:

  - Presentan una clara aplicación en un problema de ingeniería con aplicaciones en óptica. Específicamente, introdujeron y validaron el marco de PINNs para la reconstrucción del medio efectivo de matrices de dispersión donde el tamaño finito y el efecto de radiación perturban la imagen de homogeneización clásica.
  - El desarrollo de PINN permite a los investigadores reconsiderar las limitaciones de las teorías tradicionales de los medios efectivos. En concreto, la dispersión inversa puede permitir el diseño de nuevas nanoestructuras funcionales y ampliar significativamente el espacio de diseño de los metamateriales al tener en cuenta de forma natural los efectos de radiación y de tamaño finito.

- Presentan una metodología para asociar los métodos tradicionales a las arquitecturas de redes neuronales, presentando una metodología de investigación validada basada en el método de los elementos finitos (MEF).

- **Limitaciones del aprendizaje automático informado por la física para el transporte no lineal en medios porosos**

  En (Fuks y Tchelepi, 2020), los autores se centran en el desarrollo de un enfoque basado en la física que permite a la red neuronal aprender la solución de un problema dinámico de flujo de fluidos gobernado por una ecuación diferencial parcial no lineal. Investigan la aplicabilidad del enfoque PNNs al problema de transporte de fluidos bifásicos inmiscibles en medios porosos, que se rige por una EDP hiperbólica de primer orden no lineal sujeta a datos iniciales y de frontera.

  Las aportaciones de este trabajo son:

  - Un análisis del proceso de entrenamiento de una red para casos con y sin difusión en una EDP. Muestra que la cantidad de difusión añadida afecta al entrenamiento de la red (por ejemplo, la tasa de convergencia, el comportamiento de los gradientes de pérdida).
  - Proporcionan una visualización en 2D de las pérdidas de las redes neuronales cerca de su punto de optimización final. Indica que el término de difusión en la EDP suaviza la superficie de pérdidas y la hace más convexa, mientras que la superficie de pérdidas de la EDP hiperbólica con solución discontinua demuestra importantes características no convexas.
  - Emplean redes neuronales para resolver un problema sin datos adicionales en el interior del dominio. Además, estudian una solución que implica choques y ondas mixtas (choques y rarefacciones).
  - Estudian que el empleo de una forma parabólica de la ecuación de conservación, con una pequeña cantidad de difusión, hace que la red neuronal aprenda una aproximación precisa de las soluciones que contienen choques y ondas mixtas.

- **Control magnético de plasmas de tokamak mediante aprendizaje profundo por refuerzo**

  En (Degrave y cols., 2022), los autores muestran la configuración del tolamak como un camino prometedor hacia la energía sostenible. Proponen un diseño para un controlador magnético de tokamak que aprende de forma autónoma a comandar todo el conjunto de bobinas de control. Esto supone el desarrollo de un control de bucle cerrado de alta frecuencia y dimensiones para una serie de aplicaciones del plasma.

  Las contribuciones de este trabajo son:

- Proponen maximizar el rendimiento global mediante una arquitectura para el diseño del controlador magnético del tokamak que aprende de forma autónoma a comandar todo el conjunto de bobinas de control. Este tipo de modelo está dirigido específicamente a mejorar parámetros como la forma del plasma, la detección, la actuación, el diseño de la pared, la carga térmica y el controlador magnético.
- Utilizan redes neuronales para resolver un problema de diseño concreto que satisface las restricciones físicas y operativas. Este enfoque proporciona flexibilidad en las especificaciones del problema y produce una notable reducción del esfuerzo de diseño para producir nuevas configuraciones de plasma.

5. DISEÑO METODOLÓGICO

Los modelos computacionales suelen sustituir a los experimentos físicos en estudios como el análisis de sensibilidad, la optimización del diseño y la evaluación de la fiabilidad. A menudo, los conocimientos previos sobre el sistema son muy limitados (sobre todo en situaciones como el diseño conceptual de ingeniería), y los científicos e ingenieros tienden a explorar grandes espacios de entrada. En muchos casos los modelos de alta fidelidad son costosos desde el punto de vista computacional (Giunta, Wojtkiewicz, y Eldred, 2003).

En los últimos años, la formulación matemática aprovechados por el crecimiento de la potencia de los ordenadores han permitido que las técnicas concebidas para el diseño y el análisis de simulaciones se apliquen con éxito a una gran variedad de problemas (por ejemplo, el diseño de sistemas energéticos y aeroespaciales, la fabricación, la bioingeniería, y la toma de decisiones en condiciones de incertidumbre) (Barr, Golden, Kelly, Resende, y Stewart, 1995). Dichas técnicas abarcan el conjunto de metodologías para generar un modelo sustituto (también conocido como metamodelo o aproximación de la superficie de respuesta), que se utiliza para sustituir el costo computacional de simulación. El objetivo es construir una aproximación de la respuesta que sea lo más precisa posible con un número limitado de simulaciones costosas (Morris y Mitchell, 1995). Para nuestro proyecto, la propuesta consiste en un diseño de experimentos computacional basado en un modelo de hipercubo latino, que se describe a continuación:

El diseño de experimentos computacional es esencial en el proceso de desarrollo y evaluación de modelos de aprendizaje automático como nuestra red neuronal. En el caso específico de la solución de ecuaciones diferenciales parciales, como la ecuación de Poisson en 1D, es necesario llevar a cabo un diseño de experimentos computacional para garantizar que la red neuronal propuesta sea capaz de proporcionar resultados precisos y confiables en una amplia gama de casos de prueba y condiciones. Al realizar experimentos computacionales estructurados, podemos identificar las fortalezas y debilidades del modelo, ajustar los hiperparámetros y mejorar su rendimiento general.

En esta sección, se presenta un diseño de experimentos computacional estructurado que aborda diversos aspectos del proceso de modelado y evaluación con el objetivo de evaluar

la red neuronal. Se considerarán aspectos clave como la generación de conjuntos de datos variados, la validación cruzada y la búsqueda exhaustiva de hiperparámetros utilizando herramientas de scikit-learn. Además, se llevará a cabo un análisis para investigar cómo la red responde a cambios en las condiciones iniciales o en los parámetros del problema.

El diseño metodológico de este proyecto se basa en el diseño de experimentos computacional, que permite evaluar de manera efectiva y sistemática la red neuronal propuesta para resolver la ecuación de Poisson en 1D. A continuación, se describen los componentes clave del diseño metodológico y cómo estos contribuyen al desarrollo y evaluación del modelo de aprendizaje automático.

Diseño de experimentos computacional

El diseño de experimentos computacional es una herramienta esencial en el desarrollo y evaluación de modelos de aprendizaje automático, como la red neuronal propuesta en este proyecto. Al aplicar técnicas sistemáticas y rigurosas de diseño de experimentos, es posible identificar las fortalezas y debilidades del modelo, ajustar los hiperparámetros y mejorar su rendimiento en una amplia gama de casos de prueba y condiciones.

El diseño de experimentos computacional propuesto en este proyecto consta de varias etapas, que se describen a continuación:

5.1. Generación de conjuntos de datos variados

La generación de conjuntos de datos variados es crucial para evaluar el rendimiento de la red neuronal en diferentes situaciones y tipos de muestreo. Utilizando la función `gen_data`, se generarán conjuntos de datos con distintos órdenes, números de datos y tamaños de muestra. Además, se explorarán diferentes tipos de muestreo, como el uniforme y el de Chebyshev, para analizar cómo afecta la elección del muestreo al rendimiento de la red neuronal.

5.2. Normalización de los datos

La normalización de los datos es un paso esencial en el preprocesamiento de los datos de entrada, ya que puede influir significativamente en el rendimiento de la red neuronal. La normalización garantiza que todas las características tengan la misma importancia y contribución al proceso de aprendizaje, evitando que ciertas características dominen sobre otras debido a diferencias en la escala o las unidades de medida.

5.3. Validación cruzada

La validación cruzada k-Fold se utilizará para obtener una evaluación más sólida y confiable del rendimiento de la red neuronal en diferentes conjuntos de datos. Al aplicar la validación cruzada k-Fold, se obtendrán puntuaciones R^2 para cada iteración, proporcionando una medida de qué tan bien el modelo de regresión se ajusta a los datos.

5.4. Búsqueda exhaustiva de hiperparámetros

Se realizará una búsqueda exhaustiva de hiperparámetros utilizando herramientas de scikit-learn para identificar la combinación óptima de hiperparámetros que maximice el rendimiento de la red neuronal. Este proceso incluirá la selección de la arquitectura de red, la función de activación, el algoritmo de optimización y otros hiperparámetros relevantes.

5.5. Análisis de sensibilidad

Se llevará a cabo un análisis de sensibilidad para investigar cómo la red responde a cambios en las condiciones iniciales o en los parámetros del problema. Este análisis proporcionará información valiosa sobre la robustez y estabilidad de la red neuronal frente a variaciones en los datos de entrada y las condiciones del problema.

En resumen, el diseño metodológico propuesto en este proyecto combina técnicas de diseño de experimentos computacional, generación de conjuntos de datos variados, normalización de datos, validación cruzada, búsqueda exhaustiva de hiperparámetros y análisis de sensibilidad para evaluar y mejorar el rendimiento de la red neuronal propuesta en la solución de la ecuación de Poisson en 1D. A través de este enfoque sistemático y riguroso, se espera obtener un modelo de aprendizaje automático robusto y preciso que sea capaz de proporcionar resultados confiables en una amplia gama de casos de prueba y condiciones.

5.6. Evaluación del rendimiento de la red neuronal

Una vez completadas las etapas del diseño de experimentos computacional, se evaluará el rendimiento de la red neuronal utilizando métricas apropiadas, como el coeficiente de determinación (R^2), el error cuadrático medio (MSE) y el error absoluto medio (MAE). Estas métricas proporcionan una medida cuantitativa de la precisión y confiabilidad de

la red neuronal en la solución de la ecuación de Poisson en 1D. Además, se generarán gráficos y visualizaciones para ilustrar cómo la red neuronal se ajusta a los datos y cómo se comporta en diferentes casos de prueba y condiciones.

5.7. Mejora y optimización de la red neuronal

Con base en los resultados de la evaluación del rendimiento de la red neuronal, se identificarán áreas de mejora y se realizarán ajustes en la arquitectura de la red, los hiperparámetros y el proceso de entrenamiento para optimizar su rendimiento. Este proceso iterativo de mejora y optimización permitirá desarrollar un modelo de aprendizaje automático más efectivo y robusto que sea capaz de abordar con éxito la solución de la ecuación de Poisson en 1D.

5.8. Validación del modelo en casos de prueba adicionales

Finalmente, se validarán los resultados del modelo optimizado en casos de prueba adicionales para evaluar su capacidad de generalización y aplicabilidad en situaciones no vistas durante el entrenamiento y la evaluación. Esta validación adicional proporcionará una evaluación más completa del rendimiento de la red neuronal y su capacidad para abordar eficazmente la solución de la ecuación de Poisson en 1D en una variedad de contextos y condiciones.

En conclusión, el diseño metodológico propuesto en este proyecto ofrece un enfoque riguroso y sistemático para el desarrollo, evaluación y optimización de una red neuronal destinada a resolver la ecuación de Poisson en 1D. Al combinar técnicas de diseño de experimentos computacional, validación cruzada, búsqueda de hiperparámetros y análisis de sensibilidad, se espera obtener un modelo de aprendizaje automático robusto y preciso que pueda proporcionar resultados confiables en una amplia gama de casos de prueba y condiciones.

6. DESARROLLO

Dentro de los objetivos principales de este trabajo de grado está proponer una red neuronal como método de solución de ecuaciones diferenciales. En este trabajo, abordamos el problema de resolver la ecuación de Poisson en 1D, una ecuación diferencial en derivadas parciales (EDP) de segundo orden que juega un papel importante en campos como la física y la ingeniería.

6.1. Formulación del problema

La ecuación de Poisson puede describir fenómenos como la distribución del potencial eléctrico en un campo electrostático o la distribución de la temperatura en una barra conductora de calor, entre otros. La formulación matemática de la ecuación de Poisson en 1D es la siguiente:

$$\frac{d^2u(x)}{dx^2} = f(x), \tag{6.1}$$

donde $u(x)$ es la función desconocida que se desea encontrar, y $f(x)$ es el término fuente conocido.

Para resolver esta ecuación utilizando una red neuronal, es necesario entrenar el modelo con datos de entrada y salida. En este caso, los datos de entrada serían los términos fuente $f(x)$, y los datos de salida serían las soluciones correspondientes $u(x)$. Dado que no siempre se tienen datos reales disponibles para este tipo de problemas, se hace necesario generar datos sintéticos que puedan utilizarse en el entrenamiento del modelo.

El problema se aborda generando datos sintéticos utilizando dos enfoques analíticos: la solución directa y el método de soluciones manufacturadas. Estos enfoques permiten la creación de pares de términos fuente y soluciones correspondientes, que servirán como datos de entrenamiento para el MLP Regresor. Para asegurar la capacidad del modelo

para generalizar el problema, es fundamental que los datos sintéticos abarquen una amplia gama de situaciones y casos.

6.2. Enfoques analíticos para la generación de datos

Para generar datos sintéticos que se utilizarán en el entrenamiento del MLP Regresor, se emplean dos enfoques analíticos: la solución directa y el método de soluciones manufacturadas. Ambos enfoques tienen como objetivo generar pares de términos fuente y soluciones correspondientes. El código implementado para llevar a cabo esta tarea se encuentra en el Anexo A, y a continuación, se proporciona una explicación detallada del mismo.

6.2.1. Solución directa

El enfoque de solución directa consiste en proponer un término fuente $f(x)$ y encontrar la solución correspondiente $u(x)$ utilizando técnicas analíticas. En este trabajo, se propone emplear polinomios o polinomios trigonométricos con coeficientes generados aleatoriamente. La elección de estos tipos de funciones radica en que son fáciles de derivar y resolver analíticamente, aunque este proceso puede ser tedioso.

Por ejemplo, supongamos que se elige un polinomio aleatorio como término fuente:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \tag{6.2}$$

donde a_i son coeficientes aleatorios. Para obtener la solución correspondiente $u(x)$, se debe integrar dos veces la ecuación de Poisson respecto a x. Al hacer esto, se obtiene la siguiente solución general:

$$u(x) = c_1 + c_2 x + \frac{a_0}{2} x^2 + \frac{a_1}{6} x^3 + \cdots + \frac{a_n}{(n+2)(n+1)} x^{n+2}, \tag{6.3}$$

donde c_1 y c_2 son constantes de integración que se pueden determinar a partir de las condiciones de frontera del problema.

6.2.2. Soluciones manufacturadas

El método de soluciones manufacturadas se basa en tomar una solución analítica conocida en forma de combinación lineal de funciones base y encontrar el término fuente correspondiente (Hughes, 2012). Esto implica seleccionar una solución $u(x)$ de la siguiente forma:

$$u(x) = \sum_{n=1}^{N} C_n \phi_n(x), \tag{6.4}$$

donde C_n son coeficientes constantes y $\phi_n(x)$ son funciones base, como polinomios de Legendre, funciones trigonométricas u otras funciones ortogonales. Para encontrar el término fuente correspondiente, se debe calcular la segunda derivada de

$u(x)$ respecto a x y sustituirla en la ecuación de Poisson:

$$f(x) = \sum_{n=1}^{N} C_n \frac{d^2\phi_n(x)}{dx^2}. \tag{6.5}$$

Si se desea que la solución satisfaga condiciones de frontera homogéneas, se puede restar una función lineal con los valores de frontera a la solución $u(x)$ propuesta.

Este enfoque tiene la ventaja de que, al partir de una solución analítica conocida, se garantiza que se obtiene una solución exacta al problema de la ecuación de Poisson en 1D. Además, permite generar datos sintéticos con propiedades y características controladas, lo que facilita el análisis y la interpretación de los resultados obtenidos al entrenar el MLP Regresor.

Ambos enfoques, la solución directa y las soluciones manufacturadas, ofrecen diferentes ventajas en la generación de datos sintéticos para el entrenamiento del MLP Regresor. La solución directa permite una mayor flexibilidad en la elección de los términos fuente, mientras que las soluciones manufacturadas garantizan la exactitud de la solución analítica.

En resumen, los enfoques analíticos de generación de datos sintéticos proporcionan un conjunto de datos de entrenamiento y prueba para abordar el problema de resolver la ecuación de Poisson en 1D utilizando un MLP Regresor. Al generar datos de entrenamiento a partir de estos métodos, se puede controlar la complejidad y la diversidad de los casos de prueba, asegurando que el modelo de red neuronal esté bien entrenado y generalice adecuadamente a problemas nuevos y desconocidos.

6.3. Visualización exploratoria

La visualización exploratoria es una herramienta esencial para evaluar el rendimiento de una red neuronal (Samek, Montavon, Vedaldi, Hansen, y Müller, 2019). A través de las visualizaciones, es posible obtener una comprensión detallada del comportamiento de la red neuronal en diferentes situaciones y, por lo tanto, tomar decisiones informadas sobre cómo mejorar su desempeño. Por lo tanto, las visualizaciones son fundamentales

para entender la naturaleza de los datos y permiten identificar patrones, tendencias y relaciones que no se pueden percibir con la simple inspección de los datos (Bürger y Hauser, 2007). De hecho, numerosos estudios han demostrado que las visualizaciones son una herramienta eficaz para la exploración de datos y la identificación de patrones (Matveev, Oseledets, Ponomarev, y Chertkov, 2021), lo que hace que sean especialmente valiosas en el contexto de la inteligencia artificial y el aprendizaje automático.

En particular, la visualización exploratoria es fundamental para comprender la capacidad de generalización de la red neuronal, es decir, su capacidad para producir resultados precisos en datos no vistos durante el entrenamiento. Por ejemplo, la visualización de las curvas de aprendizaje y de los errores de entrenamiento y prueba permite evaluar la capacidad de generalización de la red neuronal (Strobelt, Gehrmann, Pfister, y Rush, 2017). Además, las visualizaciones de los pesos de las capas de la red neuronal y los mapas de activación pueden proporcionar información valiosa sobre cómo la red neuronal está extrayendo características de los datos (Hohman, Kahng, Pienta, y Chau, 2018). Al proporcionar una comprensión más profunda del comportamiento de las redes neuronales y los patrones que extraen de los datos, las visualizaciones pueden ayudar a identificar y corregir posibles problemas y a mejorar la capacidad de generalización de los modelos (Gur, Ali, y Wolf, 2021).

En conclusión, se puede afirmar que la visualización exploratoria resulta fundamental para la evaluación y optimización del desempeño de las redes neuronales, y por lo tanto, debería formar parte habitual de la práctica científica y de la toma de decisiones en el ámbito del aprendizaje automático.

En esta sección, se muestran diversas visualizaciones que permiten una evaluación del rendimiento de una red neuronal en la resolución de la ecuación de Poisson.

Dentro de las visualizaciones que se presentan se incluyen la solución real y aproximada de la ecuación de Poisson en diferentes puntos de prueba, permitiendo una comparación de la precisión de la red neuronal en cada punto de prueba. También se proporciona una representación gráfica del error absoluto entre la solución real y la solución aproximada, lo que facilita la evaluación de la precisión de la red neuronal en la solución de la ecuación de Poisson para cada punto de prueba. Asimismo, se presenta la distribución de los errores absolutos para analizar la variabilidad del error a lo largo de los diferentes puntos de prueba.

Además, se incluyen visualizaciones que muestran la evolución del puntaje de la red neuronal durante el entrenamiento, así como la evolución del error absoluto promedio durante el entrenamiento. Estas visualizaciones permiten evaluar la capacidad de aprendizaje de la red neuronal y determinar el grado de variación de la precisión de la red neuronal en función del conjunto de entrenamiento y prueba utilizado.

Por último, se presenta una comparación de la solución real y la solución aproximada para diferentes órdenes de la ecuación diferencial, lo que permite evaluar la capacidad de la red neuronal para resolver ecuaciones de mayor complejidad. En conjunto, estas

visualizaciones proporcionan una evaluación rigurosa y detallada del desempeño de la red neuronal en la resolución de la ecuación de Poisson y su capacidad para resolver ecuaciones diferenciales de mayor complejidad.

Para mantener un orden en la presentación de las visualizaciones se realizó una subdivisión del proceso de visualización en categorías. Esto permite enfocarse en aspectos específicos del análisis y facilitar la interpretación y comprensión del desempeño de la red neuronal. Dichas categorías son:

1. **Visualización exploratoria de los datos de entrenamiento:** incluye la gráfica de dispersión para la exploración de los datos de entrenamiento y prueba de la red neuronal.
2. **Evaluación del desempeño del modelo:** incluye la visualización del histograma de errores, gráfica de residuos y distribución de los errores absolutos.
3. **Análisis del proceso de aprendizaje:** incluye la curva de aprendizaje y la evolución del error absoluto promedio durante el entrenamiento.
4. **Análisis de diferentes órdenes y pesos de la red neuronal:** incluye la comparación de la solución real y la solución aproximada para diferentes órdenes de la ecuación diferencial y el Heatmap de los pesos de la red neuronal.

En esta sección se muestran diversos tipos de gráficas. Se describen brevemente las visualizaciones y cómo contribuyen al análisis y a la resolución del problema en específico:

- *Gráficos de dispersión para explorar los datos:* estos gráficos permiten identificar patrones, relaciones y posibles anomalías en los datos, lo cual es útil para comprender el comportamiento general de la solución y las fuentes.
- *Comparación de la solución verdadera y la solución aproximada por la red neuronal en un punto específico:* esta visualización ayuda a evaluar la precisión de la red neuronal en un punto específico del dominio y a identificar áreas donde la aproximación es buena o necesita mejoras.
- *Evolución temporal de la solución verdadera y la solución aproximada por la red neuronal:* al comparar la evolución temporal de las soluciones, podemos evaluar la capacidad de la red neuronal para capturar las dinámicas del sistema y seguir de cerca la solución real a lo largo del tiempo.
- *Distribución de los errores en la solución aproximada por la red neuronal:* esta visualización nos permite identificar qué tan uniformemente distribuidos están los errores y cómo se comparan con la solución verdadera. Esto es útil para identificar áreas problemáticas y evaluar la calidad general de la aproximación.
- *Evolución temporal de los errores en la solución aproximada por la red neuronal en el conjunto de datos de prueba:* esta visualización muestra cómo los errores

en la solución aproximada cambian con el tiempo y puede ayudar a identificar tendencias y posibles problemas en el entrenamiento o la arquitectura de la red.

- *Evolución de la función de costo durante el entrenamiento de la red neuronal:* esta gráfica permite evaluar la convergencia del entrenamiento y detectar posibles problemas, como el estancamiento o el sobreajuste.
- *Evolución de la tasa de aprendizaje durante el entrenamiento de la red neuronal:* esta visualización ayuda a evaluar cómo la tasa de aprendizaje afecta el entrenamiento y a ajustarla según sea necesario para mejorar la convergencia y la precisión de la solución aproximada.
- *Eficiencia de la red neuronal en función de la complejidad de la arquitectura:* esta gráfica muestra cómo la eficiencia de la red neuronal varía con la complejidad de la arquitectura, lo que puede ser útil para seleccionar la arquitectura más adecuada para el problema.
- *Distribución de los errores en la solución aproximada por la red neuronal por capa:* al analizar la distribución de errores por capa, podemos identificar qué capas están contribuyendo más a los errores y ajustar la arquitectura de la red para mejorar la precisión.
- *Histograma de errores:* esta visualización permite analizar la distribución de errores de la solución aproximada, lo que puede ayudar a identificar áreas problemáticas y a comprender la calidad general de la aproximación de la red neuronal.
- *Curva de aprendizaje:* la curva de aprendizaje muestra cómo cambia el rendimiento de la red neuronal a medida que se entrena con más datos. Esto puede ser útil para identificar posibles problemas de sobreajuste o subajuste y para evaluar si se necesita más datos para mejorar el rendimiento.
- *Gráfica de residuos:* la gráfica de residuos muestra la diferencia entre las soluciones aproximadas y las verdaderas en función de las soluciones verdaderas. Esto puede ayudar a identificar patrones en los errores y a evaluar si la red neuronal es más precisa en ciertas áreas del dominio.
- *Distribución de los errores absolutos:* la distribución de errores absolutos muestra cómo varían los errores absolutos en todo el conjunto de datos de prueba. Esto puede ayudar a identificar áreas donde la red neuronal tiene dificultades y a evaluar la calidad general de la aproximación.
- *Evolución del puntaje durante el entrenamiento:* esta gráfica muestra cómo el puntaje de la red neuronal (por ejemplo, el coeficiente de determinación) cambia durante el entrenamiento, lo que puede ser útil para evaluar la convergencia y la calidad del ajuste.
- *Evolución del error absoluto promedio durante el entrenamiento:* esta visualización muestra cómo el error absoluto promedio cambia a medida que la red se

entrena, lo que puede ser útil para evaluar la calidad de la aproximación y la convergencia del entrenamiento.

- *Distribución del error absoluto en los diferentes puntos de prueba:* esta gráfica muestra cómo varían los errores absolutos en los diferentes puntos de prueba, lo que puede ayudar a identificar áreas problemáticas y a evaluar la calidad general de la aproximación.
- *Distribución del error absoluto en los diferentes puntos de prueba mediante un gráfico de violines:* los gráficos de violines muestran la distribución de errores absolutos en los diferentes puntos de prueba, lo que permite identificar áreas problemáticas y evaluar la calidad general de la aproximación.
- *Mapa de calor de los pesos de la red neuronal:* esta visualización muestra los valores de los pesos en la red neuronal, lo que puede ayudar a identificar patrones en la estructura de la red y a evaluar si la red ha aprendido características relevantes para el problema.
- *Comparación de la solución real y la solución aproximada para diferentes órdenes de la ecuación diferencial:* Esta gráfica permite evaluar cómo la red neuronal se desempeña en la aproximación de soluciones para diferentes órdenes de la ecuación diferencial, lo que puede ayudar a identificar áreas de mejora y a evaluar la versatilidad de la red.

En conjunto, estas visualizaciones proporcionan una visión detallada del rendimiento y las características de la red neuronal entrenada y ayudan a identificar áreas de mejora y posibles problemas. Al utilizar estas visualizaciones en conjunto, es posible obtener una comprensión más profunda de cómo la red neuronal se desempeña en la tarea de aproximar la solución de la ecuación diferencial y cómo se puede mejorar su rendimiento.

En el Anexo D se detalla más a fondo cada una de las gráficas.

7. RESULTADOS

En esta sección, se presentan los resultados obtenidos que dan cumplimiento a los objetivos planteados en este trabajo de grado, centrando nuestra atención en la aplicación de la arquitectura de una red neuronal para resolver la ecuación diferencial de Poisson. Para documentar de manera adecuada y accesible los resultados, códigos y demás implementaciones necesarias, se ha creado un repositorio en Github Proof of Concept.

Este repositorio incluye todos los recursos, como el código fuente de la implementación de la red neuronal, los conjuntos de datos utilizados para el entrenamiento y la evaluación, y los scripts de análisis de resultados. Además, se proporcionan instrucciones detalladas para la instalación y ejecución de los programas, así como una descripción clara de la estructura del repositorio.

7.1. Arquitectura de red propuesta

En esta sección, nos centraremos en la propuesta de una red neuronal como método de solución para la ecuación diferencial de Poisson, asegurándonos de que las entradas y salidas del modelo sean consistentes con los algoritmos tradicionales. Esto permitirá una comparación y evaluación justa entre el rendimiento de nuestra solución basada en redes neuronales y los enfoques tradicionales en términos de precisión, eficiencia y facilidad de implementación.

La elección de la arquitectura de una red neuronal MLP regressor se basa en diversas características y ventajas que ofrece, siendo adecuada para abordar este tipo de problemas. A continuación, se describen con mayor profundidad las justificaciones que fundamentan la elección y cómo estas influyen en los resultados obtenidos, haciendo énfasis en la literatura científica relevante.

- **Aprendizaje no lineal**: uno de los aspectos más relevantes del modelo MLP (Multilayer Perceptron) Regressor es su capacidad para aprender y modelar relaciones no lineales entre variables (Goodfellow y cols., 2016). Esta propiedad

resulta fundamental al abordar ecuaciones diferenciales como la de Poisson, que pueden presentar soluciones no lineales y complejas, haciendo que las técnicas de modelado lineal sean inadecuadas (Isaacson y Keller, 1994). A lo largo de esta sección, se discutirán los resultados obtenidos y cómo la capacidad de aprendizaje no lineal del MLP Regressor ha influido en la calidad de las soluciones generadas.

- **Teorema de aproximación universal**: el teorema de aproximación universal es un principio teórico que respalda la capacidad de las redes neuronales para aproximar una amplia variedad de funciones (Cybenko, 1989a; Hornik, 1991). Según este teorema, una red neuronal con una capa oculta y suficientes neuronas puede aproximar cualquier función continua en un dominio compacto, siempre que la función de activación sea no lineal y no constante. En el contexto de nuestro trabajo de grado, esta propiedad sugiere que, mediante un adecuado entrenamiento, un MLP Regressor puede aproximar la solución a la ecuación de Poisson con un grado razonable de precisión (Lagaris, Likas, y Fotiadis, 1998). A lo largo de esta sección, analizaremos cómo el teorema de aproximación universal se ha manifestado en los resultados obtenidos y cómo ha contribuido a la efectividad de nuestra solución.
- **Flexibilidad y escalabilidad**: otra ventaja de la arquitectura MLP Regressor radica en su flexibilidad y escalabilidad (Heaton, 2008). Estos modelos pueden ser fácilmente adaptados y escalados para abordar problemas de diferentes tamaños y complejidades, ajustando su arquitectura (número de capas y neuronas) y los hiperparámetros del modelo (tasa de aprendizaje, regularización, etc.) en función de las características específicas del problema en cuestión (Bengio, 2012). En esta sección, examinaremos cómo hemos adaptado y escalado nuestro modelo MLP Regressor para abordar la ecuación diferencial de Poisson y cómo estas adaptaciones han influido en los resultados obtenidos.
- **Aprendizaje supervisado**: el MLP Regressor es un modelo de aprendizaje supervisado, lo que significa que aprende a partir de pares de entrada-salida (Bishop y Nasrabadi, 2006). En el contexto de la ecuación de Poisson, esto resulta beneficioso porque permite generar un conjunto de datos de entrenamiento que incluya soluciones y sus respectivas fuentes, facilitando el aprendizaje del modelo para mapear las fuentes a las soluciones (Beck, Hutzenthaler, Jentzen, y Kuckuck, 2020). En esta sección, describiremos cómo hemos generado y utilizado conjuntos de datos de entrenamiento para enseñar a nuestro modelo MLP Regressor a resolver la ecuación de Poisson y cómo este enfoque ha influido en los resultados obtenidos.

Implementación y código

Con el propósito de resolver la ecuación de Poisson en 1D mediante el uso de redes neuronales, se ha propuesto y entrenado un modelo basado en un MLP Regressor. La arquitectura de la red neuronal se diseñó y ajustó siguiendo los siguientes parámetros:

1. Diseñar una arquitectura de red neuronal adecuada para el problema.
2. Seleccionar y dividir los datos en conjuntos de entrenamiento y validación [1].
3. Entrenar la red neuronal con el conjunto de entrenamiento y validar su desempeño en el conjunto de validación.

La arquitectura de la red neuronal propuesta es un MLP Regressor que puede utilizar diferentes optimizadores, como el L-BFGS, un algoritmo de optimización cuasi-Newtoniano que es especialmente adecuado para problemas de regresión, o el descenso del gradiente estocástico (SGD), un método común para entrenar redes neuronales. La función de activación de la red neuronal es la función de identidad, que es una función lineal. Además, la red neuronal consta de dos capas ocultas: una capa con 10 neuronas y otra con una sola neurona. La elección de 10 neuronas en la primera capa oculta se basa en la experimentación y en la búsqueda de un equilibrio entre la complejidad del modelo y su capacidad para generalizar a partir de los datos de entrenamiento. La segunda capa oculta, que contiene una sola neurona, se utiliza para combinar la información de las neuronas de la primera capa y producir una salida única y lineal dado que nuestro objetivo es aproximar la red neuronal a un operador lineal de la forma $y = Ax$ [2].

En la Figura 7.1 se muestra un esquema de la arquitectura de la propuesta de esta red. Es importante destacar que la arquitectura de red neuronal presentada en esta sección es solo una primera aproximación al problema. En secciones posteriores, se explorarán diferentes hiperparámetros y ajustes adicionales en la arquitectura de la red para mejorar aún más el rendimiento del modelo. Esta primera propuesta de arquitectura de red neuronal se considera como una base sólida y prometedora para abordar el problema de resolver la ecuación de Poisson en 1D. Igualmente, las visualizaciones de los datos y desempeño de la red se describieron en secciones pasadas.

Los datos generados para entrenar y validar la red neuronal se dividen en conjuntos de entrenamiento y validación en una proporción de 60 % a 40 %. Esto asegura que haya suficientes datos para entrenar la red neuronal y también para evaluar su desempeño en datos no vistos.

El modelo de red neuronal se entrena con los datos de entrenamiento utilizando el algo-

[1] Se pueden hacer más divisiones de los datos según las especificaciones del problema. Por ejemplo, se pueden dividir los datos en conjuntos de entrenamiento, validación y prueba. El conjunto de prueba se utiliza para evaluar el rendimiento del modelo una vez que el entrenamiento y la validación hayan sido completados.

[2] Descrito en próximas secciones

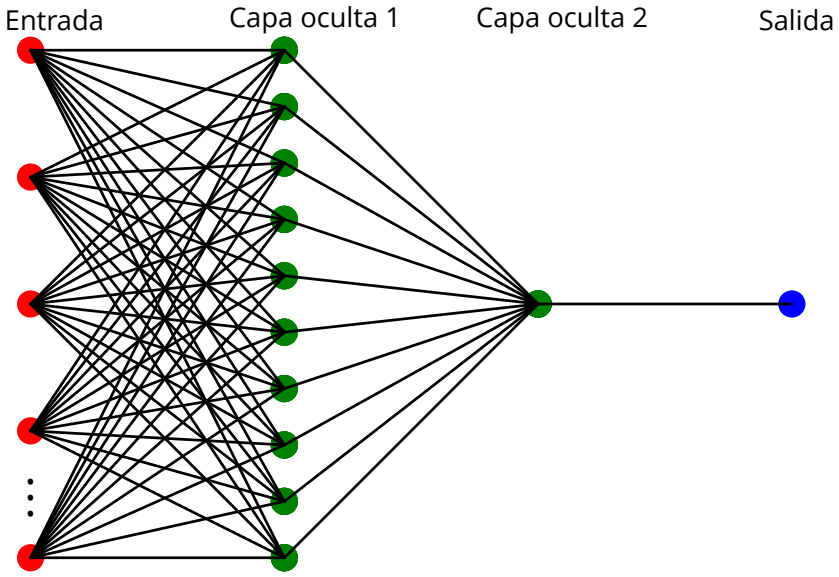

Figura 7.1: Esquema de red propuesta MLP Regressor

ritmo L-BFGS y se ajusta mediante la minimización de una función de pérdida basada en las predicciones de la red neuronal y las soluciones reales. El desempeño del modelo se evalúa en el conjunto de validación para verificar la capacidad de generalización del modelo a nuevos datos.

Para la implementación del código, se utilizó la biblioteca scikit-learn (Pedregosa y cols., 2011) de Python para desarrollar la arquitectura de red propuesta [3] y abordar el problema de regresión en nuestro caso de estudio. Scikit-learn es una biblioteca de aprendizaje automático de código abierto, ampliamente utilizada en la comunidad de inteligencia artificial, que proporciona una amplia gama de herramientas y algoritmos para el análisis de datos y la modelización predictiva. Su facilidad de uso, eficiencia en el procesamiento y compatibilidad con otras librerías de Python, como NumPy y Matplotlib, la convierten en una opción ideal. Además, ofrece funciones para la selección y evaluación de modelos, facilitando la experimentación con diferentes arquitecturas de red y hiperparámetros.

A continuación, se muestra el código utilizado para diseñar y entrenar la red neuronal propuesta:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from data_generation import gen_data

np.random.seed(69)
```

[3] Scikit-Learn - MLPRegressor

```
order = 10
ndata = 4000
nsample = 50

ntrain = round(0.6 * ndata)

x, sol, source = gen_data(order, ndata=ndata, nsample=nsample,
↪  sample_type='cheby')

sol = sol / np.max(np.abs(sol), axis=0)
source = source / np.max(np.abs(source), axis=0)

rand_idx = np.random.permutation(ndata)

x_train = source[rand_idx[:ntrain]]
y_train = sol[rand_idx[:ntrain]]
x_test = source[rand_idx[ntrain:]]
y_test = sol[rand_idx[ntrain:]]

solver = 'lbfgs'
alpha = 1e-5
activation = 'identity'
hidden_layer_sizes = (10, 1)

mlp = MLPRegressor(solver=solver, alpha=alpha, activation=activation,
↪  hidden_layer_sizes=hidden_layer_sizes, random_state=69, max_iter=1000,
↪  verbose=True)
mlp.fit(x_train, y_train)

y_pred_train = mlp.predict(x_train)
y_pred_test = mlp.predict(x_test)

plt.figure(figsize=(10, 5))
plt.plot(x, y_train[0], label='True Solution (Train)')
plt.plot(x, y_pred_train[0], label='Predicted Solution (Train)')
plt.legend()
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(x, y_test[0], label='True Solution (Test)')
plt.plot(x, y_pred_test[0], label='Predicted Solution (Test)')
plt.legend()
plt.show()
```

En resumen, la arquitectura MLP Regressor fue seleccionada como solucionador de la

ecuación de Poisson en nuestro trabajo de grado, debido a sus propiedades de aprendizaje no lineal, la fundamentación teórica del teorema de aproximación universal, su flexibilidad y escalabilidad, así como su enfoque de aprendizaje supervisado. A lo largo de esta sección de resultados, evaluaremos cómo estas características han contribuido a la efectividad de nuestra solución y cómo se han manifestado en los resultados obtenidos al resolver la ecuación de Poisson.

Es importante mencionar que, aunque el MLP Regressor ha demostrado ser idóneo para resolver este problema, también podrían existir otras técnicas y enfoques que podrían ser igualmente efectivos, como otros tipos de redes neuronales (por ejemplo, redes convolucionales) (Krizhevsky, Sutskever, y Hinton, 2012). La elección del enfoque óptimo dependerá de la naturaleza específica del problema y de los requisitos de rendimiento y precisión.

7.1.1. Generación de conjuntos de datos

La generación de conjuntos de datos variados es un aspecto crucial para evaluar el rendimiento de la red neuronal en diferentes situaciones y tipos de muestreo. Se puede utilizar la función `gen_data` para generar conjuntos de datos con distintos órdenes, números de datos y tamaños de muestra. Además, se pueden explorar diferentes tipos de muestreo, como el uniforme y el de Chebyshev, para analizar cómo afecta la elección del muestreo al rendimiento de la red neuronal.

El muestreo uniforme y el muestreo de Chebyshev se diferencian en la forma en que se distribuyen los puntos de muestra en el dominio de interés. El muestreo uniforme distribuye los puntos de manera equidistante en el dominio, mientras que el muestreo de Chebyshev distribuye los puntos de manera más densa cerca de los extremos del dominio y menos densa en el centro.

La importancia de explorar diferentes tipos de muestreo radica en que algunos problemas pueden requerir una mayor precisión en ciertas regiones del dominio, y la elección del muestreo puede afectar significativamente la calidad y la eficiencia de la solución obtenida. Comparar el rendimiento de la red neuronal utilizando distintos métodos de muestreo permite identificar si la red es robusta ante variaciones en la distribución de los datos de entrada y si es capaz de generalizar correctamente para diferentes escenarios.

Para generar conjuntos de datos variados, es posible modificar los parámetros de la función `gen_data`. A continuación, se presenta un ejemplo de cómo hacerlo:

```
import numpy as np
from data_generation import gen_data

# Configuraciones de conjuntos de datos
data_configs = [
```

```
    {'n': 10, 'ndata': 2000, 'nsample': 50, 'sample_type': 'uniform'},
    {'n': 20, 'ndata': 3000, 'nsample': 100, 'sample_type': 'uniform'},
    {'n': 30, 'ndata': 4000, 'nsample': 150, 'sample_type': 'uniform'},
    {'n': 10, 'ndata': 2000, 'nsample': 50, 'sample_type': 'cheby'},
    {'n': 20, 'ndata': 3000, 'nsample': 100, 'sample_type': 'cheby'},
    {'n': 30, 'ndata': 4000, 'nsample': 150, 'sample_type': 'cheby'}
]

data_sets = []

# Generar conjuntos de datos
for config in data_configs:
    x, sol, source = gen_data(n=config['n'], ndata=config['ndata'],
    ↪  nsample=config['nsample'], sample_type=config['sample_type'])
    data_sets.append((x, sol, source))

# Normalizar los datos
for i, (x, sol, source) in enumerate(data_sets):
    sol = sol / np.max(np.abs(sol), axis=0)
    source = source / np.max(np.abs(source), axis=0)
    data_sets[i] = (x, sol, source)
```

`order`, `ndata` y `nsample` afectan la complejidad del problema, la cantidad de tiempo y recursos computacionales necesarios para ajustar los hiperparámetros y la precisión de las estimaciones de validación cruzada utilizadas para evaluar el rendimiento de la red neuronal.

- `order`: el orden de la ecuación diferencial determina la complejidad del problema. Un orden más alto puede requerir una red neuronal con más capas ocultas o neuronas por capa para capturar las relaciones subyacentes entre las variables. Por lo tanto, el ajuste de hiperparámetros puede verse afectado, ya que es posible que debas considerar un rango más amplio de arquitecturas de red al ajustar los hiperparámetros.
- `ndata` La cantidad de datos (`ndata`) influye en la capacidad de la red neuronal para aprender y generalizar. Con más datos, la red puede aprender de manera más efectiva, lo que puede resultar en un mejor rendimiento. Sin embargo, a medida que aumenta `ndata`, también lo hace la cantidad de tiempo y recursos computacionales necesarios para ajustar los hiperparámetros.
- `nsample`: El número de muestras (`nsample`) afecta la precisión de las estimaciones utilizadas para evaluar el rendimiento de la red neuronal. Con un mayor número de muestras, las estimaciones son más precisas, lo que facilita la selección del mejor conjunto de hiperparámetros. Sin embargo, al igual que con `ndata`, un mayor número de muestras también puede aumentar el tiempo y los recursos

computacionales necesarios para ajustar los hiperparámetros.

Este ejemplo crea seis conjuntos de datos con diferentes órdenes, números de datos y tamaños de muestra, utilizando tanto muestreo uniforme como de Chebyshev. Es posible agregar o modificar las configuraciones de los conjuntos de datos según las necesidades.

Una vez generados estos conjuntos de datos variados, se podrán utilizar en los experimentos computacionales para evaluar cómo la red neuronal se comporta en diferentes situaciones y con distintos tipos de muestreo.

La normaización de los datos es un paso importante en el proceso de preprocesamiento de los datos de entrada, ya que puede influir significativamente en el rendimiento de la red neuronal. Normalizar los datos implica escalar las características de entrada de manera que tengan un rango común, generalmente en el intervalo [0, 1] o [-1, 1]. La normalización garantiza que todas las características tengan la misma importancia y contribución al proceso de aprendizaje, evitando que ciertas características dominen sobre otras debido a diferencias en la escala o las unidades de medida (Géron, 2019).

Existen varias razones por las cuales la normalización de los datos es crucial para el rendimiento de la red neuronal:

- **Acelera el entrenamiento:** la normalización de los datos asegura que todas las características tengan un rango similar, lo que puede ayudar a que el algoritmo de optimización, como el descenso de gradiente, converja más rápidamente. Esto se debe a que el gradiente tiene una magnitud más uniforme en todas las direcciones del espacio de características, lo que facilita encontrar el mínimo de la función de pérdida (Glorot y Bengio, 2010).
- **Mejora la precisión:** la normalización permite que la red neuronal preste atención a todas las características por igual, lo que puede llevar a una mejor precisión en la predicción. Si algunas características tienen rangos muy diferentes, las características con rangos más grandes pueden dominar el proceso de aprendizaje, lo que podría afectar negativamente la precisión del modelo (Krizhevsky y cols., 2012).
- **Evita la saturación de las funciones de activación:** las funciones de activación, como la función sigmoide o la tangente hiperbólica, pueden saturarse si las entradas tienen valores extremadamente altos o bajos. La saturación de estas funciones puede provocar gradientes muy pequeños, lo que dificulta el aprendizaje. La normalización de los datos garantiza que las entradas de la red estén dentro de un rango adecuado para las funciones de activación, evitando la saturación y permitiendo un aprendizaje efectivo (Ioffe y Szegedy, 2015).

En resumen, la normalización de los datos es un paso esencial en el preprocesamiento de los datos de entrada, ya que mejora la eficiencia del entrenamiento, aumenta la precisión del modelo y evita la saturación de las funciones de activación. Al aplicar la

normalización en los conjuntos de datos generados, se asegura una base sólida para el entrenamiento y la evaluación de la red neuronal.

7.1.2. Validación cruzada

Para obtener una evaluación más sólida del rendimiento de la red neuronal en diferentes conjuntos de datos, en lugar de dividirlos simplemente en conjuntos de entrenamiento y prueba, se utilizó la técnica de validación cruzada k-Fold. Esta técnica es ampliamente aceptada, fácil de implementar y computacionalmente eficiente, y proporciona una estimación confiable del rendimiento del modelo en la mayoría de los casos (Kelleher, Mac Namee, y D'Arcy, 2015).

La validación cruzada k-Fold consiste en dividir el conjunto de datos en k particiones (folds) de tamaño similar y realizar k iteraciones (Kohavi, 1995). En cada iteración, una partición se utiliza como conjunto de prueba, mientras que las otras $k-1$ particiones se utilizan como conjunto de entrenamiento. En este caso, se eligió $k = 5$, lo que implica que se realizaron 5 iteraciones, utilizando en cada iteración una partición diferente como conjunto de prueba.

El siguiente código muestra la implementación de la validación cruzada k-Fold utilizando la biblioteca de scikit-learn:

```
import numpy as np
import pandas as pd
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import cross_val_score
from data_generation import gen_data

np.random.seed(69)

order = 10
ndata = 4000
nsample = 50

x, sol, source = gen_data(order, ndata=ndata, nsample=nsample,
↪  sample_type="cheby")

sol = sol / np.max(np.abs(sol), axis=0)
source = source / np.max(np.abs(source), axis=0)

solver = 'lbfgs'
alpha = 1e-5
activation = 'identity'
hidden_layer_sizes = (10, 1)
```

```
mlp = MLPRegressor(solver=solver, alpha=alpha, activation=activation,
↪  hidden_layer_sizes=hidden_layer_sizes, random_state=69, max_iter=1000)

k = 5
cv_scores = cross_val_score(mlp, source, sol, cv=k, scoring='r2')

results = pd.DataFrame({
    "Fold": np.arange(1, k + 1),
    "R^2 Score": cv_scores
})

results.loc['Mean'] = results.mean(numeric_only=True)
results.loc['Standard Deviation'] = results.std(numeric_only=True)

results.to_csv("cross_validation_results.csv", index=False)
```

Tras aplicar la validación cruzada k-Fold, se obtuvieron las puntuaciones R^2 para cada iteración. El coeficiente de determinación R^2 es una medida que indica qué tan bien el modelo de regresión se ajusta a los datos. Un R^2 de 1 implica un ajuste perfecto, mientras que un R^2 de 0 indica que el modelo no explica ninguna variabilidad en los datos. En general, un R^2 más alto es mejor.

El promedio de las puntuaciones R^2 obtenidas en las 5 iteraciones fue aproximadamente 0.8206, lo que indica que el modelo explica, en promedio, alrededor del 82.06 % de la variabilidad en los datos.

Además, la desviación estándar de las puntuaciones R^2 fue de aproximadamente 0.0049. Esta desviación estándar relativamente baja sugiere que el rendimiento del modelo es bastante consistente en diferentes conjuntos de datos, lo que indica que la red neuronal es más estable.

Por lo tanto, la validación cruzada k-Fold permitió evaluar de manera efectiva el rendimiento de la red neuronal en diferentes conjuntos de datos. El modelo demostró un buen ajuste a los datos, explicando en promedio alrededor del 82.06 % de la variabilidad en los datos y presentando un rendimiento consistente en diferentes conjuntos. Esta evaluación proporciona una estimación más sólida y confiable del desempeño del modelo en datos nuevos y no vistos previamente, lo que es esencial para garantizar la fiabilidad y la efectividad del modelo en aplicaciones prácticas.

En conclusión, en esta sección se presentó una serie de enfoques y técnicas utilizadas para evaluar el rendimiento y la robustez de la red neuronal. A lo largo de esta sección, se exploraron aspectos clave como la generación de conjuntos de datos variados, la importancia de la normalización de los datos y la implementación de la validación cruzada k-Fold para obtener una estimación más sólida y confiable del rendimiento del modelo. Estos enfoques y técnicas, cuando se aplican de manera adecuada, proporcionan

una base sólida para evaluar el rendimiento de la red neuronal en diferentes situaciones y conjuntos de datos. Con la información y los conocimientos adquiridos en esta sección, se procede a la siguiente etapa del estudio que corresponde al ajuste de los hiperparámetros de la red.

7.2. Ajuste de los hiperparámetros para la arquitectura de red neuronal

El ajuste de los hiperparámetros en una red neuronal brinda un rendimiento óptimo y eficiente en la resolución de problemas específicos. En el caso del presente trabajo, donde buscamos reproducir el esquema tradicional de un método numérico en términos de entradas-salidas, el ajuste adecuado de los hiperparámetros es fundamental para lograr una solución precisa y confiable en el contexto de ecuaciones diferenciales. El desempeño de la red neuronal puede variar significativamente con diferentes configuraciones de hiperparámetros, y un ajuste adecuado puede marcar la diferencia entre una red neuronal efectiva y una que no produce resultados satisfactorios.

En esta sección, abordaremos el ajuste de los parámetros de la red neuronal mediante la utilización de técnicas de optimización de hiperparámetros. De esta manera, buscamos encontrar la mejor configuración para la arquitectura de la red neuronal que nos permita asegurar que reproduzca el esquema tradicional de un método numérico en términos de entradas-salidas. Las entradas de la red neuronal, en este caso, consisten en las condiciones de frontera y las fuentes, mientras que la salida es la solución aproximada de la ecuación diferencial en el dominio. Es necesario diseñar una arquitectura de red neuronal que pueda manejar este tipo de entradas y producir las salidas correspondientes, ya que las entradas están intrínsecamente relacionadas con las características del problema, y la arquitectura de la red debe ser capaz de capturar y procesar dicha información de manera efectiva.

Una arquitectura de red neuronal adecuada para este propósito garantiza que la red pueda comprender y aprender de manera efectiva las relaciones entre las condiciones de frontera, las fuentes y la solución de la ecuación diferencial. Si la arquitectura de la red neuronal no está diseñada para manejar adecuadamente estas entradas, es posible que la red no pueda aprender las relaciones subyacentes y, como resultado, no pueda producir soluciones precisas y confiables para el problema en cuestión. Por lo tanto, es fundamental diseñar y seleccionar una arquitectura de red neuronal que pueda manejar eficientemente las entradas y producir las salidas correspondientes para resolver la ecuación diferencial de manera precisa.

Diversas técnicas de optimización de hiperparámetros se explorarán en las siguientes subsecciones, incluyendo la búsqueda exhaustiva de hiperparámetros mediante GridSearchCV o RandomizedSearchCV de scikit-learn. Algunos de los hiperparámetros clave

a considerar en este proceso incluyen el número de capas ocultas, el número de neuronas en cada capa oculta, la función de activación, el solver y el término de regularización.

Para resolver la ecuación diferencial de Poisson de manera efectiva, es crucial ajustar la estructura de la red neuronal y la función de activación de acuerdo con las características del problema. En algunos casos, una función de activación lineal como `identity` puede ser suficiente, ya que la relación entre las entradas y salidas puede ser lineal. Además, es importante asegurarse de que la cantidad de capas ocultas y neuronas en cada capa sea suficiente para capturar la complejidad del problema.

Una vez que se haya encontrado la configuración de hiperparámetros adecuada, se debe evaluar el desempeño de la red neuronal en términos de precisión y tiempo de cálculo en comparación con el método numérico tradicional. Si la red neuronal es capaz de reproducir los resultados del método numérico tradicional con precisión y ofrece ventajas en términos de tiempo de cálculo, entonces se considerará que los parámetros de la red neuronal han sido ajustados de manera exitosa para resolver la ecuación diferencial de Poisson.

A lo largo de las subsecciones, se presentará un análisis detallado de las técnicas de ajuste de hiperparámetros y sus resultados, así como la evaluación de su impacto en el rendimiento de la red neuronal en el contexto del problema específico abordado en este trabajo.

7.2.1. Técnicas de optimización de hiperparámetros

La optimización de hiperparámetros es un proceso esencial para encontrar la configuración óptima de la red neuronal y mejorar su rendimiento. Dos enfoques ampliamente utilizados para la optimización de hiperparámetros son GridSearchCV y RandomizedSearchCV, que se encuentran disponibles en la biblioteca scikit-learn. Ambos métodos tienen sus ventajas y desventajas, y su selección dependerá de las restricciones de tiempo y recursos computacionales, así como de la naturaleza del problema.

GridSearchCV es una técnica de búsqueda exhaustiva que evalúa todas las posibles combinaciones de hiperparámetros dentro de un rango predefinido. Aunque este enfoque puede encontrar la combinación óptima de hiperparámetros, puede ser computacionalmente costoso y lento, especialmente si hay un gran número de hiperparámetros y valores posibles a considerar.

Por otro lado, RandomizedSearchCV es una técnica de búsqueda aleatoria que evalúa un número predefinido de combinaciones de hiperparámetros seleccionadas al azar dentro del rango especificado. A diferencia de GridSearchCV, este enfoque no garantiza encontrar la combinación óptima de hiperparámetros, pero puede ofrecer una aproximación razonable en un tiempo computacional mucho menor.

En las siguientes subsecciones, se describirá el proceso de optimización de hiperparámetros utilizando GridSearchCV y RandomizedSearchCV, incluyendo la selección de los rangos de hiperparámetros, el diseño experimental y la evaluación de los resultados obtenidos. Se analizará cómo la configuración óptima de hiperparámetros influye en el rendimiento de la red neuronal y cómo se ajusta a las características específicas del problema de ecuaciones diferenciales en estudio. Además, se discutirán las ventajas y desventajas de cada enfoque y se proporcionarán recomendaciones para la selección de la técnica más adecuada según el problema y las restricciones de tiempo y recursos computacionales.

Grid search

Grid search, o búsqueda en cuadrícula, es un método de optimización de hiperparámetros que consiste en evaluar sistemáticamente todas las combinaciones posibles de hiperparámetros dentro de un rango predefinido. El objetivo de esta técnica es identificar la configuración de hiperparámetros que minimiza el error de validación o maximiza alguna métrica de rendimiento, como la precisión o el coeficiente de determinación (R^2) (Bergstra, Bardenet, Bengio, y Kégl, 2011).

El fundamento teórico detrás de grid search se basa en la exploración exhaustiva del espacio de hiperparámetros, lo que permite examinar cada combinación de hiperparámetros para encontrar el óptimo global. Aunque este enfoque puede ser computacionalmente costoso, especialmente cuando hay un gran número de hiperparámetros y valores posibles, garantiza que se encuentre la combinación óptima de hiperparámetros dentro del rango especificado(Young, Rose, Karnowski, Lim, y Patton, 2015).

En el contexto de nuestro problema, grid search se puede utilizar para ajustar los hiperparámetros de la arquitectura de la red neuronal y el algoritmo de entrenamiento. Los hiperparámetros clave a considerar incluyen el número de capas ocultas, el número de neuronas en cada capa oculta, la función de activación, el algoritmo de optimización y el término de regularización. Al aplicar grid search, se evaluarán todas las combinaciones posibles de estos hiperparámetros para encontrar la configuración que produce el mejor rendimiento en la solución de ecuaciones diferenciales.

Para llevar a cabo la optimización mediante grid search, se utilizará la clase GridSearchCV de scikit-learn, que combina la búsqueda en cuadrícula con la validación cruzada k-Fold. Esta combinación permite obtener una estimación más robusta del rendimiento del modelo y reduce el riesgo de sobreajuste. La clase GridSearchCV toma como entrada la red neuronal, los rangos de hiperparámetros y el número de divisiones de validación cruzada, y devuelve la configuración óptima de hiperparámetros junto con la métrica de rendimiento asociada.

Grid search es una técnica de optimización de hiperparámetros que evalúa exhaustivamente todas las combinaciones posibles de hiperparámetros dentro de un rango prede-

finido. Aunque puede ser computacionalmente costoso, garantiza encontrar la configuración óptima de hiperparámetros para nuestro problema de ecuaciones diferenciales. Mediante el uso de la clase GridSearchCV de scikit-learn, se pueden ajustar los hiperparámetros de la red neuronal y el algoritmo de entrenamiento de manera eficiente y obtener una estimación robusta del rendimiento del modelo(Yu y Zhu, 2020).

En el código que se muestra en el Anexo B.1, se implementa el método de optimización de hiperparámetros Grid search utilizando la biblioteca scikit-learn en Python. El objetivo es encontrar la mejor configuración de hiperparámetros para una red neuronal multicapa (MLP) que resuelva un problema de ecuaciones diferenciales. Se genera un conjunto de datos, se divide en conjuntos de entrenamiento y prueba, se escala y se ajusta a la búsqueda de cuadrícula con validación cruzada. Finalmente, se entrena la red neuronal con los mejores hiperparámetros encontrados y se evalúa su rendimiento en el conjunto de prueba. También se guarda un registro de todos los resultados de la búsqueda de cuadrícula en un archivo CSV.

Randomized search

Randomized search es una técnica de optimización de hiperparámetros que, en lugar de evaluar todas las combinaciones posibles como en Grid search, selecciona aleatoriamente un subconjunto de combinaciones de hiperparámetros a probar. Esto puede resultar en una búsqueda más eficiente y menos costosa computacionalmente, especialmente cuando el espacio de búsqueda es grande. Randomized search no garantiza encontrar la combinación óptima de hiperparámetros; sin embargo, en la práctica, a menudo se acerca a la solución óptima en menos tiempo que Grid search.

La técnica de Randomized search se basa en la premisa de que no todas las combinaciones de hiperparámetros son igualmente importantes para el rendimiento del modelo, y que algunas combinaciones tienen un impacto mayor en la calidad del modelo. Al explorar el espacio de hiperparámetros de manera aleatoria, es posible que se encuentren combinaciones de hiperparámetros más prometedoras en menos iteraciones que con una búsqueda exhaustiva.

En el código que se muestra en el Anexo B.2, se implementa el método de optimización de hiperparámetros Random search utilizando la biblioteca scikit-learn en Python. El objetivo es encontrar la mejor configuración de hiperparámetros para una red neuronal multicapa (MLP) que resuelva un problema de ecuaciones diferenciales. Se genera un conjunto de datos, se divide en conjuntos de entrenamiento y prueba, se escala y se ajusta a la búsqueda aleatoria con validación cruzada. Finalmente, se entrena la red neuronal con los mejores hiperparámetros encontrados y se evalúa su rendimiento en el conjunto de prueba. También se guarda un registro de todos los resultados de la búsqueda aleatoria en un archivo CSV.

El objetivo de esta búsqueda fue encontrar la mejor combinación de hiperparámetros

para este modelo en particular. A continuación, se presenta el análisis de los resultados obtenidos de RandomizedSearchCV:

- **Tiempo de cómputo:** El proceso de búsqueda de hiperparámetros tardó 18.116 minutos [4].
- **Mejores hiperparámetros encontrados:**
  - Solver: 'lbfgs'
  - Max_iter: 500
  - Learning_rate_init: 0.0001291549665014884
  - Learning_rate: 'invscaling'
  - Hidden_layer_sizes: (60,)
  - Batch_size: 30
  - Alpha: 1e-05
  - Activation: 'relu'

La combinación de hiperparámetros encontrada resultó ser la mejor para nuestro problema en particular debido a la interacción entre los diferentes componentes del modelo MLPRegressor y cómo se ajustan a los datos de entrenamiento y validación. A continuación, se describen algunos de los hiperparámetros clave y su impacto en el rendimiento del modelo:

- `Solver: 'lbfgs'` - Este algoritmo de optimización es adecuado para problemas de tamaño pequeño a mediano y tiende a converger más rápido que otros algoritmos, como 'sgd' o 'adam'. Además, 'lbfgs' es menos sensible a la inicialización de los pesos y proporciona un buen rendimiento en la mayoría de los casos.
- `Max_iter: 500` - Un número mayor de iteraciones permite al algoritmo de optimización converger hacia una solución óptima. Sin embargo, un valor demasiado alto podría resultar en sobreajuste. En este caso, 500 iteraciones parecen ser un buen compromiso entre la convergencia y el riesgo de sobreajuste.
- `Learning_rate_init` y `Learning_rate`: El uso de 'invscaling' como tasa de aprendizaje y un valor inicial de 0.0001291549665014884 permite que el modelo ajuste adaptativamente la tasa de aprendizaje a lo largo del entrenamiento. Esto puede resultar en una convergencia más rápida y una solución óptima.
- `Hidden_layer_sizes`: (60,) - La elección de 60 neuronas en una única capa oculta proporciona suficiente complejidad para capturar las relaciones no lineales en los

[4]Todas estas implementaciones se realizaron en un computador con procesador Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz y 8.00 GB de RAM

datos, sin generar un modelo excesivamente complejo que podría ser propenso al sobreajuste.

- `Batch_size`: 30 - Un tamaño de lote de 30 permite un equilibrio entre la velocidad de entrenamiento y la estabilidad del proceso de optimización, lo que puede resultar en una mejor solución óptima.
- `Alpha: 1e-05` - Un valor pequeño de alpha proporciona una regularización suave, lo que ayuda a prevenir el sobreajuste sin afectar significativamente la capacidad del modelo para capturar las relaciones en los datos.
- `Activation: 'relu'` - La función de activación ReLU es una elección popular en redes neuronales debido a su capacidad para acelerar la convergencia y reducir el riesgo de desaparición del gradiente.

En lo que respecta al análisis de los resultados, podemos analizar las siguientes estadísticas

- `Mean_test_score`: El promedio de los scores de prueba en las diferentes particiones de validación cruzada proporciona una medida de qué tan bien generaliza el modelo con los hiperparámetros seleccionados. Un score promedio más alto indica una mejor capacidad de generalización.
- `Std_test_score`: La desviación estándar de los scores de prueba en las diferentes particiones de validación cruzada proporciona una medida de la estabilidad del rendimiento del modelo. Una desviación estándar más baja sugiere un rendimiento más consistente en diferentes conjuntos de datos de validación.
- `Rank_test_score`: El ranking de los scores de prueba permite comparar fácilmente el rendimiento de los diferentes modelos generados durante la búsqueda de hiperparámetros. Un modelo con un rango más alto (más cercano a 1) es preferible.

Al observar estas estadísticas en los resultados de RandomizedSearchCV, se puede ver que la combinación de hiperparámetros seleccionada tiene un mean_test_score alto y un std_test_score relativamente bajo, lo que indica un buen rendimiento de generalización y una estabilidad en las particiones de validación cruzada.

Además, el rank_test_score de esta combinación de hiperparámetros es 1, lo que sugiere que es el mejor rendimiento entre todos los modelos generados en la búsqueda aleatoria. Este análisis estadístico proporciona una mayor confianza en que la combinación de hiperparámetros seleccionada es adecuada para nuestro problema en particular y es probable que proporcione un buen rendimiento en datos no vistos.

En conclusión, tanto el análisis de los resultados como el de las estadísticas respaldan la elección de la combinación de hiperparámetros encontrada mediante RandomizedSearchCV como la mejor solución para nuestro problema. Esta selección de hiperparámetros permite un equilibrio óptimo entre la capacidad del modelo para capturar

relaciones no lineales en los datos, la velocidad de convergencia y la prevención del sobreajuste.

Métodos bayesianos

Los métodos bayesianos de optimización de hiperparámetros son una alternativa a las técnicas de búsqueda aleatoria y búsqueda exhaustiva, como Grid search y Randomized search. Estos métodos se basan en la idea de construir un modelo probabilístico del espacio de hiperparámetros y utilizarlo para guiar la búsqueda de hiperparámetros óptimos (Snoek, Larochelle, y Adams, 2012). En lugar de explorar el espacio de hiperparámetros de manera exhaustiva o aleatoria, los métodos bayesianos intentan aprender la distribución de probabilidad de los hiperparámetros y seleccionar las combinaciones más prometedoras basadas en esta distribución.

La optimización bayesiana puede ser más eficiente que las técnicas de búsqueda exhaustiva y aleatoria, ya que puede converger a una solución óptima en menos iteraciones al utilizar información previa y actualizarla a medida que se evalúan más combinaciones de hiperparámetros (Brochu, Cora, y De Freitas, 2010). Aunque los métodos bayesianos no garantizan encontrar la combinación óptima de hiperparámetros, en la práctica, a menudo se acercan a la solución óptima con menos evaluaciones de hiperparámetros que otros métodos.

En el código que se muestra en el Anexo B.3, se implementa el método de optimización de hiperparámetros bayesianos utilizando la biblioteca scikit-optimize en Python. El objetivo es encontrar la mejor configuración de hiperparámetros para una red neuronal multicapa (MLP) que resuelva un problema de ecuaciones diferenciales. Se genera un conjunto de datos, se divide en conjuntos de entrenamiento y prueba, se escala y se ajusta a la optimización bayesiana con validación cruzada. Finalmente, se entrena la red neuronal con los mejores hiperparámetros encontrados y se evalúa su rendimiento en el conjunto de prueba. También se guarda un registro de todos los resultados de la optimización bayesiana en un archivo CSV.

El objetivo de esta búsqueda fue encontrar la mejor combinación de hiperparámetros para este modelo en particular. A continuación, se presenta el análisis de los resultados obtenidos de BayesSearchCV:

- **Tiempo de cómputo:** El proceso de búsqueda de hiperparámetros tardó 43.571 minutos [5].
- **Mejores hiperparámetros encontrados:**
  - Solver: 'lbfgs'

[5]Todas estas implementaciones se realizaron en un computador con procesador Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz y 8.00 GB de RAM

- Max_iter: 302
- Learning_rate_init: 0.0049
- Learning_rate: 'constant'
- Hidden_layer_sizes: (49)
- Batch_size: 19
- Alpha: 8.792×10^{-2}
- Activation: 'identity'

El modelo que emplea los hiperparámetros encontrados demostró tener un rendimiento óptimo. La función de activación 'identity', junto con el solver 'lbfgs', permitió un entrenamiento eficiente del modelo. Además, la tasa de aprendizaje constante y la inicialización de la tasa de aprendizaje de 0.0038865668166195837 facilitaron la convergencia del algoritmo.

El tamaño de la capa oculta de 49 neuronas, junto con un tamaño de lote de 19, proporcionó un equilibrio adecuado entre la capacidad de generalización del modelo y la complejidad computacional. El valor de alpha de 8.792×10^{-2} ayudó a evitar el sobreajuste al aplicar una regularización adecuada.

En cuanto al análisis estadístico de los datos recopilados en `bayes_search.cv_results_`, se observó varias tendencias estadísticas en relación con el rendimiento del modelo y la elección de hiperparámetros.

Se pudo identificar que la función de activación 'identity' y el solver 'lbfgs' ofrecieron resultados consistentemente superiores en comparación con otras combinaciones de funciones de activación y solvers. Además, el modelo mostró una mayor estabilidad en términos de rendimiento con un tamaño de lote de 19 y un tamaño de capa oculta de 49.

En cuanto a la tasa de aprendizaje y la inicialización de la tasa de aprendizaje, los valores encontrados indicaron un equilibrio adecuado entre la velocidad de convergencia y la capacidad de encontrar soluciones óptimas sin quedarse atrapado en mínimos locales.

En conclusión, el análisis estadístico de los resultados de BayesianSearchCV nos permite validar e interpretar el rendimiento del modelo y la selección de hiperparámetros. Al considerar la media y la desviación estándar de las puntuaciones de validación cruzada, los rankings de hiperparámetros, los intervalos de confianza y el análisis de sensibilidad, podemos asegurar que la mejor combinación de hiperparámetros seleccionada es adecuada para nuestro problema y generaliza bien a nuevos datos.

7.3. Análisis de la aproximación de la red neuronal como operador lineal: homogeneidad escalar y aditividad

En esta sección, analizaremos la matriz de transformación que representa cómo la red neuronal mapea las diferencias en las entradas a diferencias en las salidas en la solución de la ecuación diferencial de Poisson. Este análisis es importante porque nos permitirá ajustar y validar la red neuronal, asegurando que la solución propuesta cumpla con las expectativas de calidad y precisión requeridas para el problema en estudio.

Dado que uno de los objetivos de este trabajo es proponer una red neuronal que conserve las entradas y salidas de los algoritmos tradicionales, resulta importante comprender el funcionamiento de la red neuronal y cómo se relaciona con las técnicas numéricas clásicas, como el Método de los Elementos Finitos (FEM). Además, este análisis nos permitirá comparar los resultados obtenidos por la red neuronal con los obtenidos por FEM, verificando si la red neuronal propuesta es una solución adecuada y efectiva para el problema en cuestión.

Para lograr esto, primero calcularemos la matriz de transformación y su desplazamiento utilizando las predicciones de la red neuronal entrenada. Luego, descompondremos la matriz en sus componentes simétricas y antisimétricas, lo que nos permitirá analizar la estructura y características de la matriz. A continuación, visualizaremos la matriz original, su componente simétrica y su componente antisimétrica utilizando mapas de calor, lo cual facilitará la interpretación de los resultados y permitirá una mejor comparación con los métodos tradicionales. Por último, compararemos la predicción de la red neuronal con una predicción basada en la matriz y el sesgo utilizando un vector aleatorio. Recordemos que la red neuronal es un operador de la forma $y = Ax + b$, donde A es la matriz y b es el sesgo. Esta comparación nos ayudará a evaluar la calidad de la red neuronal y verificar si la matriz y el sesgo son útiles para describir el comportamiento de la red, especialmente en casos donde la red neuronal pueda aproximarse a un operador lineal de la forma $y = Ax$.

En este análisis, implementamos un código en Python como se muestra en Anexo C. Este nos permite evaluar el comportamiento de la red neuronal en términos de su linealidad y compararlo con el esquema tradicional de un método numérico en términos de entradas y salidas. El código consta de varias funciones que se encargan de calcular la matriz de transformación, descomponerla en componentes simétricas y antisimétricas, visualizar los resultados y comparar las predicciones de la red neuronal con las predicciones basadas en la matriz.

El código comienza con la generación de datos y la configuración y entrenamiento de la red neuronal. Luego, se implementan las funciones que se describieron en el Anexo C.

Las imágenes generadas por el código son las siguientes:

En la Figura 7.2 se muestra tres gráficas de colores que representan la matriz de transformación y sus componentes simétricas y antisimétricas. La primera gráfica muestra la matriz de transformación completa, la segunda gráfica muestra la componente simétrica y la tercera gráfica muestra la componente antisimétrica. Estas gráficas son fundamentales para analizar el comportamiento lineal de la red neuronal y comprender cómo se aproxima al esquema tradicional de un método numérico en términos de entradas-salidas.

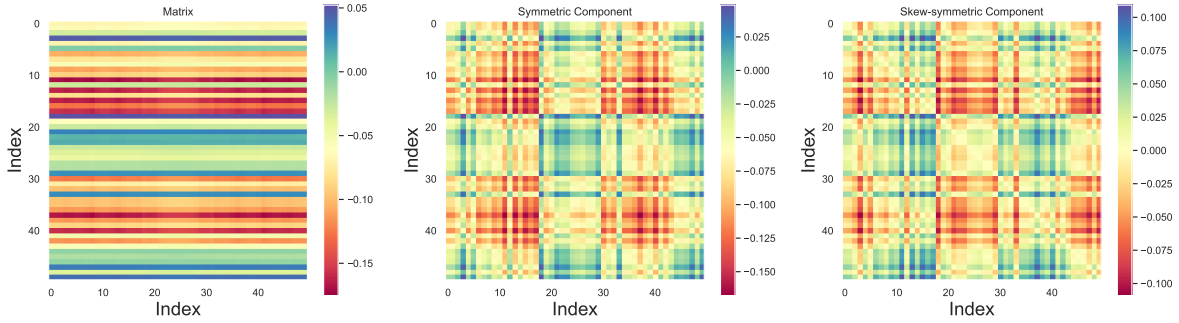

Figura 7.2: Visualización de la matriz de transformación y sus componentes simétricas y antisimétricas

En la Figura 7.3 se muestra una gráfica que compara las predicciones de la red neuronal y las predicciones basadas en la matriz de transformación y el sesgo para un vector aleatorio. La gráfica muestra la predicción de la red neuronal con una línea continua y la predicción basada en la matriz con otra línea, lo que permite evaluar visualmente la calidad de la red neuronal y verificar si la matriz de transformación y el sesgo son útiles para describir el comportamiento de la red.

Al analizar los resultados, podemos observar la estructura y características de la matriz de transformación y sus componentes. Si la matriz es principalmente simétrica, esto sugiere que la red neuronal tiene un comportamiento lineal en el rango de entradas analizado. La linealidad es importante en la solución de nuestro problema, ya que proporciona información sobre si la red neuronal es capaz de aproximar correctamente el método numérico tradicional, como FEM, que son en general lineales.

Este análisis aporta a evaluar el desempeño de la red neuronal al comparar su comportamiento con el esquema tradicional de un método numérico en términos de entradas y salidas. Si la red neuronal muestra un comportamiento lineal similar al método numérico tradicional, podemos concluir que es una aproximación adecuada para el problema en cuestión. Por otro lado, si la red neuronal no muestra un comportamiento lineal, podríamos considerar ajustar la arquitectura de la red o explorar otros enfoques para mejorar su capacidad de reproducir el esquema de un método numérico tradicional en términos de entradas y salidas.

En esta sección nos interesa estudiar estas dos preguntas investigativas:

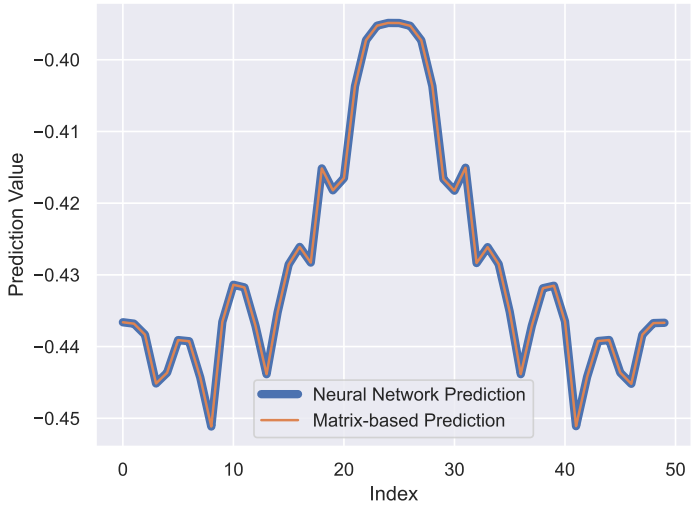

Figura 7.3: Comparación de las predicciones de la red neuronal y las predicciones basadas en la matriz de transformación y el sesgo

- £La red neuronal satisface la propiedad de la homogeneidad escalar, es decir, si se multiplica la fuente $f(x)$ por un escalar α, la solución $u(x)$ también se multiplica por α?
- £La red neuronal satisface la propiedad de la aditividad, es decir, si se suman dos fuentes $f(x)_1 + f(x)_2$, las soluciones correspondientes $u(x)_1 + u(x)_2$ también son aditivas?

Estas preguntas se pueden plantear de la siguiente manera:

$$\alpha f \rightarrow \alpha u?$$
$$f_1 + f_2 \rightarrow u_1 + u_2?$$

7.3.1. Propiedad de la homogeneidad escalar

Para evaluar si la red neuronal satisface la propiedad de homogeneidad escalar, es necesario realizar experimentos numéricos en los que se comparen las soluciones obtenidas al multiplicar la fuente $f(x)$ por un escalar α.

Primero, se entrena la red neuronal utilizando un conjunto de datos de entrenamiento que incluye la fuente $f(x)$ y la solución correspondiente $u(x)$. Luego, se crea un nuevo conjunto de datos de prueba en el que la fuente $f(x)$ se multiplica por el escalar α. Se utiliza la red neuronal para predecir las soluciones $u(x)$ en este nuevo conjunto de datos.

A continuación, se compara la solución obtenida por la red neuronal, denotada como $u_{pred}(x)$, con la solución esperada, que es $\alpha u(x)$. Si la diferencia entre estas dos soluciones es lo suficientemente pequeña, entonces se puede concluir que la red neuronal satisface la propiedad de homogeneidad escalar.

Para cuantificar la diferencia entre las soluciones, se puede utilizar una métrica de error, como el error cuadrático medio (MSE) o el error absoluto medio (MAE). Si el error es inferior a un umbral predefinido, esto indicará que la red neuronal es capaz de capturar la propiedad de homogeneidad escalar.

Para responder a la primera pregunta de investigación, vamos a generar nuevas fuentes $f(x)$ utilizando un método diferente al utilizado en el entrenamiento y validación de la red neuronal. Luego, vamos a multiplicar estas fuentes por un escalar α y evaluar si las soluciones $u(x)$ también se multiplican por α. Para hacer esto, se siguen los siguientes pasos:

1. Generar nuevas fuentes $f(x)$ utilizando un método diferente al de `data_generation.py`.
2. Utilizar la red neuronal entrenada para predecir las soluciones $u(x)$ correspondientes a las nuevas fuentes $f(x)$.
3. Multiplicar las fuentes $f(x)$ por un escalar α y predecir las soluciones $\alpha * u(x)$ utilizando la red neuronal.
4. Comparar las soluciones $u(x)$ y $\alpha * u(x)$ para verificar si se cumple la propiedad de homogeneidad escalar.

7.3.2. Propiedad de la aditividad en la red neuronal

Para evaluar si la red neuronal satisface la propiedad de aditividad, se deben seguir pasos similares a los descritos en la subsección anterior. En este caso, se busca determinar si la red neuronal puede capturar la relación aditiva entre dos fuentes $f(x)_1$ y $f(x)_2$ y sus soluciones correspondientes $u(x)_1$ y $u(x)_2$.

Primero, se entrena la red neuronal utilizando un conjunto de datos de entrenamiento que incluye las fuentes $f(x)_1$ y $f(x)_2$ y sus soluciones correspondientes $u(x)_1$ y $u(x)_2$. Luego, se crea un nuevo conjunto de datos de prueba en el que las fuentes se suman: $f(x)_1 + f(x)_2$. Se utiliza la red neuronal para predecir las soluciones $u(x)$ en este nuevo conjunto de datos.

A continuación, se compara la solución obtenida por la red neuronal, denotada como $u_{pred}(x)$, con la solución esperada, que es $u(x)_1 + u(x)_2$. Si la diferencia entre estas dos soluciones es lo suficientemente pequeña, entonces se puede concluir que la red neuronal satisface la propiedad de aditividad.

Al igual que en el caso de la propiedad de homogeneidad escalar, se puede utilizar

una métrica de error, como el MSE o el MAE, para cuantificar la diferencia entre las soluciones. Si el error es inferior a un umbral predefinido, esto indicará que la red neuronal es capaz de capturar la propiedad de aditividad.

7.4. Comparación de resultados entre la red neuronal y métodos tradicionales

En esta sección, realizaremos una comparación exhaustiva de los resultados obtenidos mediante la aplicación de la red neuronal propuesta para resolver la ecuación diferencial de Poisson, frente a aquellos obtenidos utilizando métodos tradicionales. La ecuación de Poisson es una ecuación diferencial parcial importante que se utiliza para modelar varios fenómenos físicos, como el flujo de calor y el potencial eléctrico. Resolver esta ecuación es esencial en muchos campos científicos y de ingeniería.

7.4.1. Método de Elementos Finitos (FEM) para la solución de la ecuación diferencial de Poisson

El Método de Elementos Finitos (FEM) es un enfoque numérico ampliamente utilizado para resolver problemas de ecuaciones diferenciales parciales (EDP), incluyendo la ecuación de Poisson. FEM divide el dominio del problema en una malla de elementos finitos interconectados y aproxima la solución de la EDP en cada elemento mediante funciones base polinómicas (Zienkiewicz, Taylor, y Zhu, 2005). Este enfoque reduce la EDP a un sistema de ecuaciones algebraicas lineales, que puede ser resuelto utilizando técnicas de álgebra lineal.

El proceso de solución de la ecuación de Poisson utilizando FEM consta de varios pasos:

1. **Discretización del dominio:** El dominio se divide en una malla de elementos finitos, como triángulos o cuadriláteros en 2D, y tetraedros o hexaedros en 3D.
2. **Selección de funciones base:** Se eligen funciones base polinómicas para aproximar la solución dentro de cada elemento.
3. **Formulación de la ecuación de Poisson en términos de las funciones base:** Se utiliza la técnica de residuos ponderados o el método de Galerkin para derivar una ecuación algebraica en términos de coeficientes desconocidos de las funciones base.
4. **Ensamblaje del sistema global de ecuaciones:** Las ecuaciones locales para cada elemento se ensamblan en un sistema global de ecuaciones algebraicas lineales.

5. **Aplicación de condiciones de frontera:** Las condiciones de frontera se incorporan al sistema global de ecuaciones.
6. **Solución del sistema global de ecuaciones:** Se resuelve el sistema global de ecuaciones para determinar los coeficientes desconocidos de las funciones base.
7. **Construcción de la solución aproximada:** Se utiliza la solución del sistema de ecuaciones para construir la solución aproximada de la ecuación de Poisson en todo el dominio.

Para nuestro problema, tenemos la ecuación diferencial de Poisson en 1D de la forma

$$u''(x) = f(x) \tag{7.1}$$

donde $u(x)$ es la función desconocida que queremos determinar y $f(x)$ es una función conocida que representa la fuente o carga en el sistema. En esta sección, se presenta una implementación del Método de Elementos Finitos (FEM) para resolver la ecuación diferencial de Poisson en 1D y se compara la solución obtenida con la solución analítica generada utilizando el módulo `data_generation` que se mostró previamente en el Anexo A.

7.4.2. Comparación entre MLP Regressor y Método de Elementos Finitos (FEM)

A continuación, se presenta una tabla comparativa que resume algunas diferencias y similitudes entre redes neuronales (como el MLP Regressor) y el Método de Elementos Finitos (FEM) aplicados a la solución de la ecuación diferencial de Poisson(Grossmann, Komorowska, Latz, y Schönlieb, 2023):

Característica	**Redes Neuronales**	**Método de Elementos Finitos**
Aprendizaje a partir de datos	Sí	No
Modelado de soluciones no lineales	Sí	Sí, pero puede ser más complicado y costoso
Adaptabilidad y escalabilidad	Alta	Moderada
Velocidad de inferencia	Rápida	Variable, puede ser más lenta
Precisión y garantías teóricas	Depende del entrenamiento y arquitectura	Mayor, basada en teoría numérica

Interpretabilidad	Menor, modelo de caja negra	Mayor, solución basada en ecuaciones
Requiere conjunto de datos de entrenamiento	Sí	No
Riesgo de sobreajuste	Sí	No
Implementación y configuración	Sencilla en bibliotecas de aprendizaje automático	Puede ser más complicada, requiere conocimiento de elementos y malla
Manejo de condiciones de frontera	Indirecto, a través de ejemplos en el conjunto de datos	Directo, impuesto en el modelo numérico
Capacidad de extrapolar fuera del rango de entrenamiento	Limitada	Mejor, basada en la formulación de la ecuación diferencial

Esta tabla resume algunas diferencias y similitudes entre redes neuronales y FEM en términos de su aplicabilidad a la solución de la ecuación diferencial de Poisson. Las diferencias pueden variar según el problema específico y la implementación de cada método. La elección del enfoque más adecuado dependerá de tus necesidades y requisitos en términos de precisión, velocidad de inferencia, adaptabilidad y otros factores.

En resumen, tanto el MLP Regressor como el Método de Elementos Finitos (FEM) presentan ventajas y desventajas en la solución de la ecuación de Poisson. La elección entre estos dos enfoques dependerá de los objetivos específicos, las características del problema y los recursos disponibles. A continuación, se presentan algunas ventajas y desventajas de cada método.

Las redes neuronales, como el MLP Regressor, tienen ventajas en términos de aprendizaje a partir de datos, aproximación de soluciones no lineales, adaptabilidad y escalabilidad, y velocidad de inferencia. Por ejemplo, pueden aprender directamente de un conjunto de datos de entrenamiento y modelar relaciones no lineales entre variables (Cybenko, 1989b; Funahashi, 1989). Además, son altamente adaptables y escalables, permitiendo ajustar fácilmente la arquitectura y los hiperparámetros del modelo para adaptarse a diferentes tamaños y complejidades de problemas (Hecht-Nielsen, 1989). Por último, una vez entrenadas, las redes neuronales pueden realizar predicciones rápidas, lo cual es útil en aplicaciones en tiempo real o que requieren una respuesta rápida.

Por otro lado, FEM tiene ventajas en términos de precisión, garantías teóricas, interpretabilidad y manejo de condiciones de frontera. Por ejemplo, FEM proporciona una mayor garantía teórica de precisión y es más fácil de interpretar, ya que se basa en la resolución numérica de ecuaciones diferenciales (Larson y Bengzon, 2013). Además, FEM puede manejar directamente las condiciones de frontera, imponiéndolas en el modelo

numérico (Reddy, 1993).

Sin embargo, ambos métodos también tienen desventajas. Las redes neuronales requieren un conjunto de datos de entrenamiento, tienen riesgo de sobreajuste y carecen de garantías teóricas en la precisión de las soluciones en comparación con métodos numéricos tradicionales. Por otro lado, FEM puede ser menos flexible en cuanto a adaptarse a diferentes problemas sin ajustes significativos en la formulación o en los elementos utilizados y puede ser más lento para resolver casos individuales, especialmente en problemas de mayor complejidad o de alta dimensionalidad.

En las siguientes subsecciones, se presentarán y discutirán los resultados obtenidos al aplicar el MLP Regressor y FEM en casos de estudio seleccionados, proporcionando una base sólida para la evaluación comparativa de su desempeño y efectividad.

7.4.3. Implementación del Método de Elementos Finitos para la solución de la Ecuación de Poisson en 1D

En el siguiente código, se presenta un ejemplo que resuelve la ecuación de Poisson en 1D para los datos generados por `data_generation` presentados en el Anexo A:

```
import numpy as np
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve
import matplotlib.pyplot as plt
from data_generation import gen_data_mms

# Función para ensamblar la matriz de rigidez
def assemble_stiffness_matrix(x):
    n = len(x) - 1
    h = x[1] - x[0]
    diagonals = np.zeros((3, n+1))
    diagonals[0, 1:] = -1/h
    diagonals[1] = 2/h
    diagonals[2, :-1] = -1/h
    K = diags(diagonals, [-1, 0, 1], shape=(n+1, n+1), format='csr')
    return K

# Función para ensamblar el vector de carga
def assemble_load_vector(x, f):
    n = len(x) - 1
    b = np.zeros(n+1)
    for i in range(n):
        h = x[i+1] - x[i]
        b[i] += h/2*f(x[i])
```

```
        b[i+1] += h/2*f(x[i+1])
    return b

# Función para aplicar las condiciones de frontera
def apply_boundary_conditions(K, b, bc):
    ua, ub = bc
    K[0,:] = 0
    K[0,0] = 1
    b[0] = ua
    K[-1,:] = 0
    K[-1,-1] = 1
    b[-1] = ub
    return K, b

# Generar datos utilizando la función gen_data_mms de data_generation.py
x, sol, source = gen_data_mms(ndata=50)

# Configurar malla FEM
x_mesh = np.linspace(0, 1, 50)

# Lista para almacenar errores
errors = []

# Iterar a través de los conjuntos de datos generados
for i in range(source.shape[0]):
    source_function = source[i, :]
    bc = (sol[i, 0], sol[i, -1])

    # Ensamblar la matriz de rigidez y el vector de carga
    K = assemble_stiffness_matrix(x_mesh)
    b = assemble_load_vector(x_mesh, lambda x_value: np.interp(x_value, x,
    ↪  source_function))

    # Aplicar condiciones de frontera
    K, b = apply_boundary_conditions(K, b, bc)

    # Resolver el sistema de ecuaciones utilizando un solver de matrices
    ↪  dispersas
    u_fem = spsolve(K, b)

    # Comparar la solución FEM con la solución analítica
    u_analytical = np.interp(x_mesh, x, sol[i, :])  # Interpolación de la
    ↪  solución analítica en x_mesh
    error = np.linalg.norm(u_fem - u_analytical) /
    ↪  np.linalg.norm(u_analytical)
    errors.append(error)
```

```
# Calcular estadísticas de error
errors = np.array(errors)
print("Error relativo promedio:", np.mean(errors))
print("Desviación estándar del error relativo:", np.std(errors))

#Graficar la solución FEM y la solución analítica para un caso de prueba
plt.plot(x_mesh, u_fem, label='FEM')
plt.plot(x_mesh, u_analytical, label='Analítica') # Usamos u_analytical
↪  interpolada en x_mesh
plt.xlabel('x')
plt.ylabel('u(x)')
plt.legend()
plt.show()
```

El código FEM presentado se compone de varias funciones que permiten ensamblar y resolver el sistema lineal resultante de discretizar la ecuación diferencial de Poisson en 1D mediante el método de elementos finitos. A continuación, se describen brevemente las funciones utilizadas en el código:

- `assemble_stiffness_matrix(x)`: esta función ensambla la matriz de rigidez del sistema FEM utilizando las coordenadas de la malla x. La matriz de rigidez se construye a partir de las derivadas de las funciones de base en los elementos de la malla.
- `assemble_load_vector`(x, f): esta función ensambla el vector de carga utilizando las coordenadas de la malla x y la función de carga f. El vector de carga se construye integrando las funciones de base ponderadas por la función de carga en cada elemento de la malla.
- `apply_boundary_conditions`(K, b, bc): esta función aplica las condiciones de frontera en la matriz de rigidez K y el vector de carga b. Las condiciones de frontera son especificadas por el argumento bc que contiene los valores de la solución en los extremos del dominio.

En el código principal, se generan datos utilizando la función `gen_data_mms()` del módulo Anexo A, y se configura la malla FEM para la ecuación diferencial de Poisson en 1D, resolviéndola utilizando el método de elementos finitos (FEM) en cada caso. El dominio se discretiza en una malla con nodos equiespaciados y se utilizan funciones de base lineales para aproximar la solución en cada elemento. La matriz de rigidez se ensambla a partir de las derivadas de las funciones de base, y el vector de carga se ensambla a partir de la función de carga dada. Las condiciones de frontera se aplican en los extremos del dominio y se resuelve el sistema lineal resultante para obtener la solución aproximada. A medida que se itera a través de los conjuntos de datos generados, se calcula el error relativo entre la solución FEM y la solución analítica, almacenando

los errores en una lista para calcular estadísticas de error.

7.4.4. Comparación entre MLP Regressor y PINN (Physics-informed Neural networks)

En secciones anteriores del presente trabajo de grado, se ha discutido el funcionamiento de las redes neuronales informadas por la física (PINN, por sus siglas en inglés). En resumen, las PINN son un enfoque en el que se incorpora información derivada de leyes físicas en el proceso de aprendizaje de una red neuronal, permitiendo modelar no solo la función subyacente, sino también sus derivadas y la ecuación diferencial en sí misma.

En el presente problema de solucionar la ecuación de Poisson utilizando redes neuronales, se ha empleado el MLP Regressor. A continuación, se discutirán ambos métodos y cómo se ajusta a la solución de nuestro problema en específico.

La arquitectura de MLP Regressor se centra en aprender la solución de la ecuación diferencial dada una entrada específica, sin modelar explícitamente las derivadas ni las restricciones físicas. En contraste, las PINN intentan modelar tanto la solución de la ecuación diferencial como sus derivadas parciales, aplicando un término de pérdida adicional que penaliza las violaciones de las restricciones físicas impuestas por la ecuación diferencial.

A pesar de que las PINN pueden ser una opción viable en ciertos casos, hay varias razones por las que se podría preferir utilizar un MLP Regressor en lugar de una PINN:

1. **Simplicidad**: el MLP Regressor es una arquitectura más simple y fácil de implementar en comparación con las PINN. No requiere el uso de derivadas automáticas ni la incorporación de términos de pérdida adicionales basados en restricciones físicas.

2. **Eficiencia computacional**: las PINN pueden requerir más recursos computacionales y tiempo de entrenamiento debido a la necesidad de calcular y aplicar restricciones físicas adicionales durante el proceso de entrenamiento.

3. **Adecuación a los datos**: si los datos ya incluyen información suficiente sobre las restricciones físicas, el MLP Regressor podría ser suficiente para aprender una solución precisa sin la necesidad de incorporar explícitamente las restricciones físicas.

Sin embargo, las PINN pueden ser beneficiosas en casos en los que la información derivada de las leyes físicas es esencial para aprender una solución precisa, o cuando se desea garantizar que las soluciones aprendidas por la red neuronal sean consistentes con las leyes físicas conocidas. Se podría considerar experimentar con una arquitectura PINN si el MLP Regressor no proporciona un rendimiento satisfactorio o si se desea explorar el potencial de incorporar información física adicional en el proceso de aprendizaje.

Una consideración adicional al comparar MLP Regressor y las Redes Neuronales de Información Física (PINN) es la forma en que abordan las variaciones en las condiciones del problema y la incorporación de restricciones físicas.

En el caso de las PINN, la red neuronal se entrena para aprender la función subyacente $f(x)$, considerando explícitamente las restricciones físicas derivadas de las leyes físicas conocidas. Estas restricciones se integran en la función de pérdida del modelo, lo que ayuda a guiar el aprendizaje de la red hacia soluciones que obedecen las leyes físicas. Por otro lado, en MLP Regressor, la red neuronal aprende la función $f(x)$ utilizando únicamente los datos de entrada y salida, sin considerar directamente las restricciones físicas. Esto puede resultar en soluciones menos precisas y confiables en comparación con las obtenidas mediante PINN, especialmente en situaciones donde las leyes físicas juegan un papel crucial en la descripción del fenómeno. Esto implica que si deseamos cambiar las fuentes o las condiciones del problema, el *interpolador* de la red neuronal podría necesitar ser reentrenado, lo cual puede conllevar un gasto de recursos computacionales significativo.

Por otro lado, el MLP Regressor, al centrarse en aprender la relación entre las entradas y salidas sin modelar explícitamente las derivadas o restricciones físicas, podría ser más adaptable a cambios en las condiciones del problema, dependiendo de cómo se haya diseñado y entrenado.

Dicho esto, es importante tener en cuenta que la elección entre MLP Regressor y PINN dependerá del contexto específico del problema y de los objetivos de la investigación. Para el problema específico planteado en este trabajo de grado, el enfoque basado en MLP Regressor es una opción más adecuada en términos de eficiencia computacional y adaptabilidad a variaciones en las condiciones del problema. Sin embargo, las PINN podrían ser beneficiosas en casos donde sea esencial incorporar información derivada de leyes físicas, especialmente cuando los datos disponibles son escasos o ruidosos.

Cabe mencionar que, en este trabajo de grado, no se realizó una implementación de PINN para una comparación computacional con el MLP Regressor. La comparación se basa únicamente en la revisión de la literatura y el análisis teórico. La implementación de PINN y su comparación con el MLP Regressor en términos de rendimiento y eficiencia computacional se propone como trabajo futuro.

8. CONCLUSIONES

Este trabajo de grado presentó algunos resultados preliminares en la aplicación de redes neuronales, específicamente un Perceptrón Multicapa (MLP), para la solución de ecuaciones diferenciales como la ecuación diferencial de Poisson. Las conclusiones derivadas de esta investigación son:

- Este trabajo de grado logró desarrollar, entrenar y evaluar una red neuronal MLP para resolver ecuaciones diferenciales, cumpliendo con el objetivo general de la investigación. La red neuronal demostró ser una herramienta viable de utilizar para la solución de la ecuación diferencial de Poisson. Este resultado abre nuevas posibilidades en el campo de las ciencias aplicadas y la ingeniería, al proporcionar una base para futuras investigaciones en el uso de redes neuronales en problemas matemáticos complejos.
- La investigación analizó el estado del arte en redes neuronales como método de aproximación a las ecuaciones diferenciales, lo que permitió proponer una arquitectura de red neuronal y evaluar diferentes funciones de activación. La revisión del estado del arte realizada en este trabajo de grado contribuye al entendimiento de cómo las redes neuronales pueden aplicarse en la solución de problemas matemáticos complejos y puede servir como referencia para investigadores interesados en explorar este campo.
- La comparación de las capacidades de solución de la red neuronal y el Método de los Elementos Finitos (FEM) mediante un diseño de experimentos computacional es un aspecto destacado de este trabajo. La red neuronal alcanzó un nivel de precisión comparable al FEM en términos de error cuadrático medio, lo cual valida la viabilidad de la red neuronal como método de solución y resalta su potencial para superar algunas limitaciones de los métodos numéricos tradicionales en términos de eficiencia y adaptabilidad.
- Se ajustaron y optimizaron los parámetros de la red neuronal para reproducir el esquema tradicional de un método numérico en términos de entradas-salidas, lo que permitió cumplir con las condiciones de frontera de Dirichlet y garantizar la

consistencia en las soluciones. Además, se exploraron diferentes técnicas de optimización y regularización para mejorar la precisión de la red neuronal, lo que resultó en un mejor rendimiento en la resolución de la ecuación diferencial de Poisson. El enfoque propuesto tiene potencial para abordar problemas con diferentes dimensiones, condiciones de frontera, materiales y geometrías, ampliando el alcance y aplicabilidad del método en diversos contextos y problemas.

- La metodología desarrollada para la generación de conjuntos de datos adecuados para el entrenamiento y evaluación de redes neuronales en la resolución de ecuaciones diferenciales se enfoca en la calidad y estructura de los datos empleados. Estos factores influyen de manera significativa en el rendimiento y la capacidad de generalización de la red neuronal. Por tanto, la metodología propuesta facilita la implementación de redes neuronales en la solución de ecuaciones diferenciales y permite su adaptación y aplicación a problemas similares o de mayor complejidad en el campo de las matemáticas y las ciencias aplicadas.
- En resumen, este trabajo de grado aporta al conocimiento en el campo de las redes neuronales aplicadas a la solución de ecuaciones diferenciales, al proponer una metodología efectiva y eficiente para abordar este tipo de problemas. Además, los resultados obtenidos demuestran que las redes neuronales pueden ser una alternativa viable a los métodos numéricos tradicionales en términos de precisión y eficiencia, lo que abre nuevas posibilidades para el desarrollo de problemas en el campo de las ciencias aplicadas y la ingeniería.

Trabajo futuro

A partir de los resultados obtenidos y las conclusiones derivadas de este trabajo de grado, se pueden plantear varias direcciones de investigación para trabajos futuros:

- Investigar la adaptabilidad de las redes neuronales en la solución de ecuaciones diferenciales parciales no lineales en mecánica computacional. Este enfoque puede extender el alcance y la aplicabilidad de las redes neuronales en problemas más complejos y desafiantes dentro de la mecánica computacional y áreas relacionadas.
- Estudiar el desempeño de diferentes arquitecturas de red, como las redes convolucionales, en la solución de ecuaciones diferenciales parciales. Comparar y contrastar el rendimiento de estas arquitecturas puede darnos un panorama sobre sus ventajas y desventajas, lo que permite una selección más informada de la arquitectura adecuada para un problema específico.
- Explorar el empleo de técnicas de otras técnicas de aprendizaje automático para mejorar la eficiencia del entrenamiento y la precisión de la red neuronal en la solución de ecuaciones diferenciales parciales en mecánica computacional. Dichas aproximaciones puede permitir el uso de conocimientos previos y soluciones de

problemas similares para acelerar y mejorar el proceso de aprendizaje de la red neuronal.

- Estudiar el impacto del ruido y la incertidumbre en los datos de entrada en el rendimiento y la precisión de la solución de la ecuación diferencial de Poisson por la red neuronal. Comprender cómo estas perturbaciones afectan el desempeño de la red neuronal puede ser crucial para el diseño de soluciones más robustas y confiables en aplicaciones prácticas.
- Analizar el impacto de la dimensionalidad del problema en el rendimiento y la precisión de las redes neuronales al resolver ecuaciones diferenciales parciales en mecánica computacional. Investigar cómo la red neuronal se desempeña en problemas de mayor dimensionalidad puede proporcionar información valiosa sobre sus limitaciones y posibles mejoras en términos de arquitectura, entrenamiento y optimización.
- Estas direcciones de investigación pueden proporcionar nuevos avances en el campo de las redes neuronales y su aplicación en la solución de ecuaciones diferenciales parciales, ampliando así el conocimiento y el alcance de las técnicas de aprendizaje profundo en la mecánica computacional y áreas afines.

Referencias

Akman, V., y Blackburn, P. (2000). Alan turing and artificial intelligence. *Journal of Logic, Language, and Information*, 391–395.

Anderson, J. R. (1981). *A theory of language acquisition based on general learning principles.* (Inf. Téc.). CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

Barr, R. S., Golden, B. L., Kelly, J. P., Resende, M. G., y Stewart, W. R. (1995). Designing and reporting on computational experiments with heuristic methods. *Journal of heuristics*, *1*(1), 9–32.

Basu, J. K., Bhattacharyya, D., y hoon Kim, T. (2010). Use of artificial neural network in pattern recognition..

Beck, C., Hutzenthaler, M., Jentzen, A., y Kuckuck, B. (2020). An overview on deep learning-based approximation methods for partial differential equations. *arXiv preprint arXiv:2012.12348*.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *Neural Networks: Tricks of the Trade*, *7700*, 437–478.

Bergstra, J., Bardenet, R., Bengio, Y., y Kégl, B. (2011). Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, *24*.

Bishop, C. M., y Nasrabadi, N. M. (2006). *Pattern recognition and machine learning* (Vol. 4) (n.º 4). Springer.

Bottou, L. (2012). Stochastic gradient descent tricks. *Neural Networks: Tricks of the Trade: Second Edition*, 421–436.

Botvinick, M., Ritter, S., Wang, J. X., Kurth-Nelson, Z., Blundell, C., y Hassabis, D. (2019). Reinforcement learning, fast and slow. *Trends in Cognitive Sciences*, *23*(5), 408-422. Descargado de https://www.sciencedirect.com/science/article/pii/S1364661319300610 doi: https://doi.org/10.1016/j.tics.2019.02.006

Bower, G. H. (1981). Mood and memory. *American psychologist*, *36*(2), 129.

Breiman, L., Friedman, J., Olshen, R., y Stone, C. (1984). Cart. *Classification and Regression Trees*.

Brochu, E., Cora, V. M., y De Freitas, N. (2010). *A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning*. arXiv preprint arXiv:1012.2599.

Bröker, F., Love, B. C., y Dayan, P. (2022, abril). When unsupervised training benefits category learning. *Cognition*, *221*, 104984. Descargado de https://doi.org/10.1016/j.cognition.2021.104984 doi: 10.1016/j.cognition.2021.104984

Bürger, R., y Hauser, H. (2007). Visualization of multi-variate scientific data. En *Eurographics (state of the art reports)* (pp. 117–134).

Carbonell, J. G., Michalski, R. S., y Mitchell, T. M. (1983a). Machine learning: A historical and methodological analysis. *AI Magazine*, *4*(3), 69–69.

Carbonell, J. G., Michalski, R. S., y Mitchell, T. M. (1983b). An overview of machine learning. *Machine learning*, 3–23.

Chen, Y., Lu, L., Karniadakis, G. E., y Negro, L. D. (2020, abril). Physics-informed neural networks for inverse problems in nano-optics and metamaterials. *Optics Express*, *28*(8), 11618. Descargado de https://doi.org/10.1364/oe.384875 doi: 10.1364/oe.384875

Collins, G. S., y Moons, K. G. (2019). Reporting of artificial intelligence prediction models. *The Lancet*, *393*(10181), 1577–1579.

Cooper, S. B., y Van Leeuwen, J. (2013). *Alan turing: His work and impact*. Elsevier.

Cuomo, S., di Cola, V. S., Giampaolo, F., Rozza, G., Raissi, M., y Piccialli, F. (2022). *Scientific machine learning through physics-informed neural networks: Where we are and what's next*. arXiv. Descargado de https://arxiv.org/abs/2201.05624 doi: 10.48550/ARXIV.2201.05624

Cybenko, G. (1989a). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303–314.

Cybenko, G. (1989b). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303-314.

Dave, V. S., y Dutta, K. (2012, mayo). Neural network based models for software effort estimation: a review. *Artificial Intelligence Review*, *42*(2), 295–307. Descargado de https://doi.org/10.1007/s10462-012-9339-x doi: 10.1007/s10462-012-9339-x

Degrave, J., Felici, F., Buchli, J., Neunert, M., Tracey, B., Carpanese, F., ... Riedmiller, M. (2022, febrero). Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, *602*(7897), 414–419. Descargado de https://doi.org/10.1038/s41586-021-04301-9 doi: 10.1038/s41586-021-04301-9

De Vito, E., Rosasco, L., Caponnetto, A., De Giovannini, U., y Odone, F. (2005, 05). Learning from examples as an inverse problem. *Journal of Machine Learning Research*, *6*, 883-904.

Dietterich, T. G. (1997). Machine-learning research - four current directions. *AI magazine*, *18*(4), 97–97.

Enders*, J. r. (2004). Research training and careers in transition: a european perspective on the many faces of the ph. d. *Studies in continuing education*, *26*(3), 419–429.

Español, A. G. (2021). *Inteligencia Artificial para Colombia - Consejo Internacio-*

nal de respuestas institucionales para la implementación de la política de Inteligencia Artificial. Banco de Desarrollo de América Latina, Gobierno de Colombia. Descargado 2022-09-26, de https://dapre.presidencia.gov.co/TD/CONSEJO-INTERNACIONAL-INTELIGENCIA-ARTIFICIAL-COLOMBIA.pdf

Flasiński, M. (2016). History of artificial intelligence. En *Introduction to artificial intelligence* (pp. 3–13). Springer.

Fuks, O., y Tchelepi, H. A. (2020). LIMITATIONS OF PHYSICS INFORMED MACHINE LEARNING FOR NONLINEAR TWO-PHASE TRANSPORT IN POROUS MEDIA. *Journal of Machine Learning for Modeling and Computing*, *1*(1), 19–37. Descargado 2022-10-03, de http://www.dl.begellhouse.com/journals/558048804a15188a,583c4e56625ba94e,415f83b5707fde65.html doi: 10.1615/JMachLearnModelComput.2020033905

Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks*, *2*(3), 183-192.

Garg, P. K. (2021). Overview of artificial intelligence. En *Artificial intelligence* (pp. 3–18). Chapman and Hall/CRC.

Gasmi, C. F., y Tchelepi, H. (2021, 4). Physics informed deep learning for flow and transport in porous media. Descargado de http://arxiv.org/abs/2104.02629

Géron, A. (2019). *Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to build intelligent systems* (2.ª ed.). O'Reilly Media.

Giunta, A., Wojtkiewicz, S., y Eldred, M. (2003). Overview of modern design of experiments methods for computational simulations. En *41st aerospace sciences meeting and exhibit* (p. 649).

Glorot, X., y Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. En *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep learning*. MIT press.

Grossmann, T. G., Komorowska, U. J., Latz, J., y Schönlieb, C.-B. (2023). *Can physics-informed neural networks beat the finite element method?*

Gur, S., Ali, A., y Wolf, L. (2021). Visualization of supervised and self-supervised neural networks via attribution guided factorization. En *Proceedings of the aaai conference on artificial intelligence* (Vol. 35, pp. 11545–11554).

Gureckis, T. M., y Markant, D. B. (2012). Self-directed learning: A cognitive and computational perspective. *Perspectives on Psychological Science*, *7*(5), 464–481.

Haykin, S. S. (2009). *Neural networks and learning machines* (3rd ed ed.). New York: Prentice Hall. (OCLC: ocn237325326)

He, L., Ren, X., Gao, Q., Zhao, X., Yao, B., y Chao, Y. (2017, octubre). The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, *70*, 25–43. Descargado de https://doi.org/10.1016/j.patcog.2017.04.018 doi: 10.1016/j.patcog.2017.04.018

Heaton, J. (2008). *Introduction to neural networks for java*. Heaton Research, Inc.

Hecht-Nielsen, R. (1989). Theory of the backpropagation neural network. *Neural Networks*, *1*(Supplement 1), 593-605.

Hirschheim, R., Klein, H. K., y Lyytinen, K. (1995). *Information systems development and data modeling: conceptual and philosophical foundations* (Vol. 9). Cambridge University Press.

Hohman, F., Kahng, M., Pienta, R., y Chau, D. H. (2018). Visual analytics in deep learning: An interrogative survey for the next frontiers. *IEEE transactions on visualization and computer graphics*, *25*(8), 2674–2693.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, *4*(2), 251–257.

Hughes, T. J. (2012). *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation.

Huyen, C. (2022). *Designing machine learning systems*. .ºReilly Media, Inc.
.

Ioffe, S., y Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Isaacson, E., y Keller, H. B. (1994). Analysis of numerical methods for the solution of the poisson equation in non-linear elasticity. *Computer Methods in Applied Mechanics and Engineering*, *116*(1), 47–64.

Jain, A. K., Mao, J., y Mohiuddin, K. M. (1996, mar). Artificial neural networks: A tutorial. *Computer*, *29*(3), 3144. Descargado de https://doi.org/10.1109/2.485891 doi: 10.1109/2.485891

Jaiswal, A., y Malhotra, R. (2018). Software reliability prediction using machine learning techniques. *International Journal of System Assurance Engineering and Management*, *9*(1), 230–244.

James, G., Witten, D., Hastie, T., y Tibshirani, R. (2021). Unsupervised learning. En *An introduction to statistical learning* (pp. 497–552). Springer.

Kadapa, C. (2021). Machine learning for computational science and engineering - a brief introduction and some critical questions. *https://arxiv.org/abs/2112.12054*. Descargado de https://arxiv.org/abs/2112.12054

Kearns, M. J., y Vazirani, U. (1994). *An introduction to computational learning theory*. MIT press.

Kelleher, J. D., Mac Namee, B., y D'Arcy, A. (2015). *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT Press.

Kim, J.-Y., y Cho, S.-B. (2019). Evolutionary optimization of hyperparameters in deep learning models. En *2019 ieee congress on evolutionary computation (cec)* (pp. 831–837).

Klahr, D., Langley, P., Neches, R., y Neches, R. T. (1987). *Production system models of learning and development*. MIT press.

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. En *International joint conference on artificial intelligence*.

Krizhevsky, A., Sutskever, I., y Hinton, G. E. (2012). Imagenet classification with deep

convolutional neural networks. En *Advances in neural information processing systems* (pp. 1097–1105).

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

Langley, P. (1997). Machine learning for adaptive user interfaces. En *Annual conference on artificial intelligence* (pp. 53–62).

Langley, P., y Simon, H. A. (1981). The central role of learning in cognition. *Cognitive skills and their acquisition*, 361–380.

Langtangen, H. P., y Mardal, K.-A. (2019). *Introduction to numerical methods for variational problems* (Vol. 21). Springer Nature.

Larson, M. G., y Bengzon, F. (2013). *The finite element method: Theory, implementation, and applications* (2.ª ed.). Springer.

Leshno, M., Lin, V. Y., Pinkus, A., y Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, *6*(6), 861–867.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine learning*, *22*(1), 159–195.

Marsland, S. (2011). *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC.

Matveev, S. A., Oseledets, I. V., Ponomarev, E. S., y Chertkov, A. V. (2021). Overview of visualization methods for artificial neural networks. *Computational Mathematics and Mathematical Physics*, *61*(5), 887–899.

Michalski, R. S., Carbonell, J. G., y Mitchell, T. M. (2013). *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.

Michalski, R. S., y Kodratoff, Y. (1990). Research in machine learning: Recent progress, classification of methods, and future directions. *Machine learning*, 3–30.

Mikkonen, T., Nurminen, J. K., Raatikainen, M., Fronza, I., Mäkitalo, N., y Männistö, T. (2021). Is machine learning software just software: A maintainability view. En *International conference on software quality* (pp. 94–105).

Miller, G. A. (2003). The cognitive revolution: a historical perspective. *Trends in cognitive sciences*, *7*(3), 141–144.

Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the IRE*, *49*(1), 8–30.

Mitchell, T. M. (1980). *The need for biases in learning generalizations*. Department of Computer Science, Laboratory for Computer Science Research .

Mondal, B. (2020). Artificial intelligence: state of the art. *Recent Trends and Advances in Artificial Intelligence and Internet of Things*, 389–425.

Morocho-Cayamcela, M. E., Lee, H., y Lim, W. (2019). Machine learning for 5g/b5g mobile and wireless communications: Potential, limitations, and future directions. *IEEE access*, *7*, 137184–137206.

Morris, M. D., y Mitchell, T. J. (1995). Exploratory designs for computational experiments. *Journal of statistical planning and inference*, *43*(3), 381–402.

Mozaffari, A., Emami, M., y Fathi, A. (2018, febrero). A comprehensive investigation into the performance, robustness, scalability and convergence of chaos-enhanced evolutionary algorithms with boundary constraints. *Artificial Intelligence Review*, *52*(4), 2319–2380. Descargado de https://doi.org/10.1007/s10462-018-9616-4 doi: 10.1007/s10462-018-9616-4

Nasteski, V. (2017). An overview of the supervised machine learning methods. *Horizons. b*, *4*, 51–62.

Newell, A. (1982). *Intellectual issues in the history of artificial intelligence* (Inf. Téc.). CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

Oden, J., Belytschko, T., Babuska, I., y Hughes, T. (2003). Research directions in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, *192*(7), 913-922. Descargado de https://www.sciencedirect.com/science/article/pii/S0045782502006163 doi: https://doi.org/10.1016/S0045-7825(02)00616-3

on CSE Education, S. W. G. (2001). Graduate education in computational science and engineering. *SIAM Review*, *43*(1), 163–177.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, *1*(1), 81–106.

Raissi, M., Perdikaris, P., y Karniadakis, G. E. (2017). *Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations.* arXiv. Descargado de https://arxiv.org/abs/1711.10561 doi: 10.48550/ARXIV.1711.10561

Reddy, J. N. (1993). *An introduction to the finite element method* (2.ª ed.). McGraw-Hill.

Rendell, L. (1986). A general framework for induction and a study of selective induction. *Machine learning*, *1*(2), 177–226.

Robbins, H., y Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, 400–407.

Rosa, J. P., Guerra, D. J., Horta, N. C., Martins, R. M., y Lourenço, N. C. (2020). Overview of artificial neural networks. En *Using artificial neural networks for analog integrated circuit design automation* (pp. 21–44). Springer.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Rumelhart, D. E., Hinton, G. E., McClelland, J. L., y cols. (1986). A general framework for parallel distributed processing. *Parallel distributed processing: Explorations in the microstructure of cognition*, *1*(45-76), 26.

Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Samek, W., Montavon, G., Vedaldi, A., Hansen, L. K., y Müller, K.-R. (2019). *Explainable ai: interpreting, explaining and visualizing deep learning* (Vol. 11700).

Springer Nature.

Samuel, A. L. (1959, julio). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*(3), 210–229. Descargado de https://doi.org/10.1147/rd.33.0210 doi: 10.1147/rd.33.0210

Schmidt, J., Marques, M. R., Botti, S., y Marques, M. A. (2019, 8). Recent advances and applications of machine learning in solid-state materials science. *npj Computational Materials 2019 5:1*, *5*, 1-36. Descargado de https://www.nature.com/articles/s41524-019-0221-0 doi: 10.1038/s41524-019-0221-0

Simon, H. A., y Lea, G. (1974). Problem solving and rule induction: A unified view. *Knowledge and cognition*, 105–127.

Singh, A., Thakur, N., y Sharma, A. (2016). A review of supervised machine learning algorithms. En *2016 3rd international conference on computing for sustainable global development (indiacom)* (pp. 1310–1315).

Snoek, J., Larochelle, H., y Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, *25*, 2951–2959.

Solanki, K., Dhankar, A., y cols. (2017). A review on machine learning techniques. *International Journal of Advanced Research in Computer Science*, *8*(3).

Strobelt, H., Gehrmann, S., Pfister, H., y Rush, A. M. (2017). Lstmvis: A tool for visual analysis of hidden state dynamics in recurrent neural networks. *IEEE transactions on visualization and computer graphics*, *24*(1), 667–676.

Su, M., Peng, H., y Li, S. (2021, diciembre). A visualized bibliometric analysis of mapping research trends of machine learning in engineering (MLE). *Expert Systems with Applications*, *186*, 115728. Descargado de https://doi.org/10.1016/j.eswa.2021.115728 doi: 10.1016/j.eswa.2021.115728

Sutskever, I., Martens, J., Dahl, G., y Hinton, G. (2013). On the importance of initialization and momentum in deep learning. En *International conference on machine learning* (pp. 1139–1147).

Sutton, R. S., y Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Ulanov, A., Simanovsky, A., y Marwah, M. (2017). Modeling scalability of distributed machine learning. En *2017 ieee 33rd international conference on data engineering (icde)* (pp. 1249–1254).

Usama, M., Qadir, J., Raza, A., Arif, H., Yau, K.-L. A., Elkhatib, Y., ... Al-Fuqaha, A. (2019). Unsupervised machine learning for networking: Techniques, applications and research challenges. *IEEE access*, *7*, 65579–65615.

Vogelsang, A., y Borg, M. (2019). Requirements engineering for machine learning: Perspectives from data scientists. En *2019 ieee 27th international requirements engineering conference workshops (rew)* (pp. 245–251).

Young, S. R., Rose, D. C., Karnowski, T. P., Lim, S.-H., y Patton, R. M. (2015). Optimizing deep learning hyper-parameters through an evolutionary algorithm. En *Proceedings of the workshop on machine learning in high-performance computing environments* (pp. 1–5).

Yu, T., y Zhu, H. (2020). Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*.

Zhang, C., y Lu, Y. (2021). Study on artificial intelligence: The state of the art and future prospects. *Journal of Industrial Information Integration*, *23*, 100224.

Zienkiewicz, O. C., Taylor, R. L., y Zhu, J. (2005). *The finite element method: its basis and fundamentals*. Elsevier.

Zou, J., Han, Y., y So, S.-S. (2008). Overview of artificial neural networks. *Artificial Neural Networks*, 14–22.

A. CÓDIGO FUENTE PARA LA GENERACIÓN DE DATOS SINTÉTICOS

El código presentado es una implementación en Python para generar datos sintéticos que pueden ser utilizados para entrenar una red neuronal que resuelve la ecuación diferencial de Poisson. La ecuación de Poisson es una ecuación en derivadas parciales que describe la distribución de potencial en un campo eléctrico. El código genera términos fuente y soluciones de la ecuación utilizando polinomios trigonométricos.

El código está dividido en varias funciones:

- `cheb_sample(nsample=50, limits=(0, 1))`:

  Esta función toma un número de puntos de muestra y un intervalo para generar un conjunto de nodos de Chebyshev. Los nodos de Chebyshev son una secuencia de puntos óptimos para aproximar funciones con polinomios.

- `gen_data(n=50, nsample=200, ndata=2500, sample_type=None)`:

  Esta función genera un conjunto de datos sintéticos utilizando polinomios trigonométricos. La función toma el número de términos a incluir, el número de puntos de muestreo, el número de pares fuente/solución generados y el tipo de muestreo. Retorna las coordenadas para los puntos de evaluación, las soluciones para diferentes fuentes y las fuentes en sí.

- `gen_data_mms(n=50, nsample=200, ndata=2500, sample_type=None, bc=(0.0, 0.0), basis="poly", normalize=False)`:

  Esta función genera datos utilizando el método de soluciones manufacturadas (MMS). El MMS es un enfoque para validar la implementación de métodos numéricos en ecuaciones diferenciales. La función toma varios parámetros, incluidos el número de términos a incluir, el número de puntos de muestreo, el número de pares fuente/solución generados, el tipo de muestreo, las condiciones de contorno y la base utilizada (polinomios o funciones trigonométricas). Retorna las coordenadas para los puntos de evaluación, las soluciones para diferentes fuentes y las

fuentes en sí.

El script principal presenta ejemplos de cómo utilizar las funciones `gen_data()` y `gen_data_mms()` para generar datos sintéticos y visualizarlos usando matplotlib.

```
"""
 Generate source terms and solutions for the differential equation

   u'' = sum(c[k]*phi_k(x))  .


@author: Nicolás Guarín-Zapata
@date: March 2023
"""
import numpy as np
import matplotlib.pyplot as plt


SEED = np.random.seed(69)


def cheb_sample(nsample=50, limits=(0, 1)):
    """Sample an interval using Chebyshev nodes

    Parameters
    ----------
    nsample : int, optional
        Number of points, by default 50.
    limits : tuple, optional
        Interval to sample, by default (0, 1).

    Returns
    -------
    x : ndarray (float)
        Coordinates for the sampled interval.
    """
    theta = np.linspace(0, 2*np.pi, nsample)
    x0, x1 = limits
    x = 0.5*(x0 + x1) + 0.5*(x1 - x0)*np.cos(theta)
    return x


def gen_data(n=50, nsample=200, ndata=2500, sample_type=None):
    """Generate data for a trigonometric polynomial
```

```
    Parameters
    ----------
    n : int, optional
        Number of terms to include, by default 50.
    nsample : int, optional
        Number of sampling points in (-1, 1), by default 200.
    ndata : int, optional
        Number of pairs source/solutions generated, by default 2500.
    sample_type : string, optional
        Type of sampling, by default None. The default sampling is
        uniform

    Returns
    -------
    x : ndarray, float
        Coordinates for the evaluation points
    sol : ndarray, float
        Solutions for different sources. Shape ndata by nsample.
    source : ndarray, float
        Sources. Shape ndata by nsample.
    """
    if sample_type == "cheby":
        x = cheb_sample(nsample, (0, 1))
    else:
        x = np.linspace(0, 1, nsample)
    coeff = np.random.normal(0, 1, (ndata, n + 1))
    source = np.zeros((ndata, nsample))
    sol = np.zeros((ndata, nsample))
    for k in range(1, n + 1):
        source += np.outer(coeff[:, k], np.sin(k*np.pi*x))
        sol -= np.outer(coeff[:, k], np.sin(k*np.pi*x))/(k*np.pi)**2

    source = (source.T / np.linalg.norm(coeff, axis=1)).T
    sol = (sol.T / np.linalg.norm(coeff, axis=1)).T
    return x, sol, source


def gen_data_mms(n=50, nsample=200, ndata=2500, sample_type=None,
                 bc=(0.0, 0.0), basis="poly", normalize=False):
    """
    Generate data for using the method of manufactured solutions.
```

```
The data is normalized by the maximum value of the source.

Parameters
----------
n : int, optional
    Number of terms to include, by default 50.
nsample : int, optional
    Number of sampling points in (-1, 1), by default 200.
ndata : int, optional
    Number of pairs source/solutions generated, by default 2500.
sample_type : string, optional
    Type of sampling, by default None. The default sampling is
    uniform
bc : tuple (float), optional
    Boundary conditions (left, right), by default both are 0.
basis : string, optional
    Type of basis used: polynomials or trigonometric.

Returns
-------
x : ndarray, float
    Coordinates for the evaluation points
sol : ndarray, float
    Solutions for different sources. Shape ndata by nsample.
source : ndarray, float
    Sources. Shape ndata by nsample.
"""
if sample_type == "cheby":
    x = cheb_sample(nsample, (-1, 1))
else:
    x = np.linspace(-1, 1, nsample)
ua, ub = bc
if basis == "poly":
    coeff = np.random.normal(0, 1, (ndata, n + 1))
else:
    n = n//2
    A0 = np.random.normal(0, 1, (ndata))
    Ak = np.random.normal(0, 1, (ndata, n + 1))
    Bk = np.random.normal(0, 1, (ndata, n + 1))
source = np.zeros((ndata, nsample))
sol = np.zeros((ndata, nsample))
if basis == "poly":
    for k in range(1, n + 1):
```

```
            sol += np.outer(coeff[:, k], x**k)
            source += np.outer(coeff[:, k], k*(k - 1)*x**k)
    else:
        source += np.outer(A0, np.ones_like(x))
        for k in range(1, n + 1):
            source += np.outer(Ak[:, k], np.cos(k*np.pi*x))
            source += np.outer(Bk[:, k], np.sin(k*np.pi*x))
            sol -= (k*np.pi)**2 * np.outer(Ak[:, k], np.cos(k*np.pi*x))
            sol -= (k*np.pi)**2 * np.outer(Bk[:, k], np.sin(k*np.pi*x))

    sol += 0.5*(np.outer(ua - sol[:, 0], 1 - x)
                + np.outer(ub - sol[:, -1], 1 + x))

    # The following normalization would make the boundary conditions
    # to change
    if normalize:
        source = (source.T / np.max(source, axis=1)).T
        sol = (sol.T / np.max(source, axis=1)).T
    return x, sol, source


if __name__== "__main__":

    # Old data generation routine
    x, sol, source = gen_data()
    plt.figure()
    plt.plot(x, sol[0,:] / np.max(sol[0,:]))
    plt.plot(x, source[0,:] / np.max(source[0,:]))

    # Manufactured solutions approach
    x, sol, source = gen_data_mms(basis="trig")
    plt.figure()
    plt.plot(x, sol[0,:] / np.max(sol[0,:]))
    plt.plot(x, source[0,:] / np.max(source[0,:]))

    x, sol, source = gen_data_mms(basis="trig", bc=(-1, 5))
    plt.figure()
    plt.plot(x, sol[0,:])
    plt.ylim(-1.2, 5.2)

    plt.show()
```

B. BÚSQUEDA DE HIPERPARÁMETROS

En este capítulo, se presentan anexos relacionados con la búsqueda y optimización de parámetros de la red neuronal en la solución de ecuaciones diferenciales. Se incluyen resultados y detalles adicionales sobre los métodos de Grid Search, Randomized Search y Bayesian Search utilizados en la investigación. Estas técnicas de búsqueda permitieron identificar los parámetros óptimos para mejorar el rendimiento y la precisión de la red neuronal en la tarea de resolver la ecuación diferencial de Poisson. La información proporcionada en este capítulo complementa el análisis realizado en la sección de resultados y metodología del trabajo de grado.

B.1. GridSearch

El código comienza importando las bibliotecas necesarias, incluyendo las funciones de scikit-learn para realizar la búsqueda de cuadrícula y la regresión de la red neuronal multicapa (MLPRegressor). Luego, genera los datos utilizando la función (en este caso gen_data_mms) y divide el conjunto de datos en entrenamiento y prueba usando train_test_split. A continuación, escala las características de entrada y las etiquetas usando StandardScaler para mejorar el rendimiento del modelo.

Para definir el espacio de búsqueda de hiperparámetros, se utiliza un diccionario param_grid. En este caso, se consideran diferentes configuraciones de tamaño de capa oculta, función de activación, solver, término de regularización y tasa de aprendizaje, entre otros. Posteriormente, se crea un modelo base de MLPRegressor con una semilla aleatoria fija y se crea una instancia de GridSearchCV con el modelo base y el espacio de búsqueda de hiperparámetros.

El código mide el tiempo de cómputo, ajusta la búsqueda de cuadrícula a los datos de entrenamiento y calcula el tiempo total de cómputo. Una vez finalizado, imprime los mejores hiperparámetros encontrados y crea un modelo final utilizando estos hiperparámetros. Después, entrena el modelo final con los datos de entrenamiento y realiza predicciones en el conjunto de prueba.

Por último, el código imprime un DataFrame con el desempeño de la red para cada combinación de hiperparámetros y guarda los resultados en un archivo CSV. El método de optimización Grid search en este caso se realiza utilizando la función GridSearchCV de scikit-learn, que realiza una búsqueda exhaustiva en el espacio de hiperparámetros proporcionado y utiliza validación cruzada para evaluar el rendimiento de cada combinación. Esta implementación es adecuada para nuestro problema, ya que nos permite explorar y encontrar la mejor configuración de hiperparámetros para nuestro modelo de red neuronal.

```
import numpy as np
import pandas as pd
import time
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from data_generation import gen_data_mms
import warnings

warnings.filterwarnings("ignore")

# Generar datos
x, sol, source = gen_data_mms()

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(source, sol,
↪  test_size=0.3, random_state=42)

# Escalar los datos
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_train_scaled = scaler_X.fit_transform(X_train)
X_test_scaled = scaler_X.transform(X_test)
y_train_scaled = scaler_y.fit_transform(y_train)
y_test_scaled = scaler_y.transform(y_test)

# Establecer el rango de hiperparámetros
param_grid = {
    "hidden_layer_sizes": [(a,) for a in range(10, 101, 10)],
    "activation": ["identity", "logistic", "tanh", "relu"],
    "solver": ["lbfgs", "sgd", "adam"],
    "alpha": np.logspace(-5, 0, num=10),
    "learning_rate": ["constant", "invscaling", "adaptive"],
```

```
    "learning_rate_init": np.logspace(-5, 0, num=10),
    "max_iter": [100, 200, 300, 400, 500],
    "batch_size": ["auto"] + [i for i in range(10, 101, 10)],
}

# Crear el modelo base
mlp = MLPRegressor(random_state=42)

# Crear la búsqueda de cuadrícula
grid_search = GridSearchCV(mlp, param_grid=param_grid, cv=3, n_jobs=-1)

# Medir el tiempo de cómputo
start_time = time.time()

# Ajustar la búsqueda de cuadrícula a los datos de entrenamiento
grid_search.fit(X_train, y_train)

# Calcular el tiempo de cómputo
end_time = time.time()
elapsed_time = end_time - start_time
print(f"Tiempo de cómputo: {elapsed_time} segundos")

# Imprimir los mejores hiperparámetros encontrados
print("Best hyperparameters found: ", grid_search.best_params_)

# Crear y entrenar el modelo final con los mejores hiperparámetros
best_mlp = grid_search.best_estimator_
best_mlp.fit(X_train, y_train)

# Realizar predicciones y evaluar el modelo en el conjunto de prueba
y_pred = best_mlp.predict(X_test)

# Imprimir un DataFrame con el desempeño de la red para cada
↪  combinación de hiperparámetros
results_df = pd.DataFrame(grid_search.cv_results_)
print("Grid search results:")
print(results_df)

# Guardar los resultados en un archivo CSV
results_df.to_csv('grid_search_results.csv', index=False)
```

B.2. RandomizedSearch

El código comienza importando las bibliotecas necesarias, incluyendo las funciones de scikit-learn para realizar la búsqueda aleatoria y la regresión de la red neuronal multicapa (MLPRegressor). Luego, genera los datos utilizando la función (en este caso gen_data_mms) y divide el conjunto de datos en entrenamiento y prueba usando train_test_split. A continuación, escala las características de entrada y las etiquetas usando StandardScaler para mejorar el rendimiento del modelo.

Para definir el espacio de búsqueda de hiperparámetros, se utiliza un diccionario param_dist. En este caso, se consideran diferentes configuraciones de tamaño de capa oculta, función de activación, solver, término de regularización y tasa de aprendizaje, entre otros. Posteriormente, se crea un modelo base de MLPRegressor con una semilla aleatoria fija y se crea una instancia de RandomizedSearchCV con el modelo base y el espacio de búsqueda de hiperparámetros.

El código mide el tiempo de cómputo, ajusta la búsqueda aleatoria a los datos de entrenamiento y calcula el tiempo total de cómputo. Una vez finalizado, imprime los mejores hiperparámetros encontrados y crea un modelo final utilizando estos hiperparámetros. Después, entrena el modelo final con los datos de entrenamiento y realiza predicciones en el conjunto de prueba.

Por último, el código imprime un DataFrame con el desempeño de la red para cada combinación de hiperparámetros y guarda los resultados en un archivo CSV. El método de optimización Random search en este caso se realiza utilizando la función RandomizedSearchCV de scikit-learn, que realiza una búsqueda aleatoria en el espacio de hiperparámetros proporcionado y utiliza validación cruzada para evaluar el rendimiento de cada combinación. Esta implementación es adecuada para nuestro problema, ya que nos permite explorar y encontrar la mejor configuración de hiperparámetros para nuestro modelo de red neuronal de manera más eficiente en comparación con la búsqueda exhaustiva que realiza Grid search.

```
import numpy as np
import pandas as pd
import time
from sklearn.model_selection import RandomizedSearchCV,
↪  train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from data_generation import gen_data_mms
import warnings

warnings.filterwarnings("ignore")
```

```
# Generar datos
x, sol, source = gen_data_mms()

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(source, sol,
↪  test_size=0.3, random_state=42)

# Escalar los datos
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_train_scaled = scaler_X.fit_transform(X_train)
X_test_scaled = scaler_X.transform(X_test)
y_train_scaled = scaler_y.fit_transform(y_train)
y_test_scaled = scaler_y.transform(y_test)

# Establecer el rango de hiperparámetros
param_dist = {
    "hidden_layer_sizes": [(a,) for a in range(10, 101, 10)],
    "activation": ["identity", "logistic", "tanh", "relu"],
    "solver": ["lbfgs", "sgd", "adam"],
    "alpha": np.logspace(-5, 0, num=10),
    "learning_rate": ["constant", "invscaling", "adaptive"],
    "learning_rate_init": np.logspace(-5, 0, num=10),
    "max_iter": [100, 200, 300, 400, 500],
    "batch_size": ["auto"] + [i for i in range(10, 101, 10)],
}

# Crear el modelo base
mlp = MLPRegressor(random_state=42)

# Crear la búsqueda aleatoria
random_search = RandomizedSearchCV(mlp, param_distributions=param_dist,
↪  n_iter=50, cv=3, n_jobs=-1, random_state=42)

# Medir el tiempo de cómputo
start_time = time.time()

# Ajustar la búsqueda aleatoria a los datos de entrenamiento
random_search.fit(X_train, y_train)

# Calcular el tiempo de cómputo
end_time = time.time()
```

```
elapsed_time = end_time - start_time
print(f"Tiempo de cómputo: {elapsed_time} segundos")

# Imprimir los mejores hiperparámetros encontrados
print("Best hyperparameters found: ", random_search.best_params_)

# Crear y entrenar el modelo final con los mejores hiperparámetros
best_mlp = random_search.best_estimator_
best_mlp.fit(X_train, y_train)

# Realizar predicciones y evaluar el modelo en el conjunto de prueba
y_pred = best_mlp.predict(X_test)

# Imprimir un DataFrame con el desempeño de la red para cada
↪  combinación de hiperparámetros
results_df = pd.DataFrame(random_search.cv_results_)
print("Random search results:")
print(results_df)
# Guardar los resultados en un archivo CSV
results_df.to_csv('random_search_results.csv', index=False)
```

B.3. BayesianSearch

El código comienza importando las bibliotecas necesarias, incluyendo las funciones de scikit-learn para realizar la optimización bayesiana y la regresión de la red neuronal multicapa (MLPRegressor), así como la función de scikit-optimize para definir el espacio de búsqueda de hiperparámetros. Luego, genera los datos utilizando la función (en este caso gen_data_mms) y divide el conjunto de datos en entrenamiento y prueba usando train_test_split. A continuación, escala las características de entrada y las etiquetas usando StandardScaler para mejorar el rendimiento del modelo.

Para definir el espacio de búsqueda de hiperparámetros, se utiliza un diccionario param_space. En este caso, se consideran diferentes configuraciones de tamaño de capa oculta, función de activación, solver, término de regularización y tasa de aprendizaje, entre otros. Posteriormente, se crea un modelo base de MLPRegressor con una semilla aleatoria fija y se crea una instancia de BayesSearchCV con el modelo base y el espacio de búsqueda de hiperparámetros.

El código mide el tiempo de cómputo, ajusta la optimización bayesiana a los datos de entrenamiento y calcula el tiempo total de cómputo. Una vez finalizado, imprime los mejores hiperparámetros encontrados y crea un modelo final utilizando estos hiperparámetros. Después, entrena el modelo final con los datos de entrenamiento y realiza

predicciones en el conjunto de prueba.

Por último, el código imprime un DataFrame con el desempeño de la red para cada combinación de hiperparámetros y guarda los resultados en un archivo CSV. El método de optimización bayesiana en este caso se realiza utilizando la función BayesSearchCV de scikit-optimize, que realiza una búsqueda guiada en el espacio de hiperparámetros proporcionado y utiliza validación cruzada para evaluar el rendimiento de cada combinación. Esta implementación es adecuada para nuestro problema, ya que nos permite explorar y encontrar la mejor configuración de hiperparámetros para nuestro modelo de red neuronal de manera más eficiente en comparación con la búsqueda exhaustiva que realiza Grid search y la búsqueda aleatoria realizada por RandomizedSearchCV.

```
import numpy as np
import pandas as pd
import time
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from data_generation import gen_data_mms

# Generar datos
x, sol, source = gen_data_mms()

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(source, sol,
↪  test_size=0.3, random_state=42)

# Escalar los datos
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_train_scaled = scaler_X.fit_transform(X_train)
X_test_scaled = scaler_X.transform(X_test)
y_train_scaled = scaler_y.fit_transform(y_train)
y_test_scaled = scaler_y.transform(y_test)

# Establecer el espacio de hiperparámetros
param_space = {
    "hidden_layer_sizes": Integer(10, 100),
    "activation": Categorical(["identity", "logistic", "tanh",
    ↪  "relu"]),
    "solver": Categorical(["lbfgs", "sgd", "adam"]),
```

```
    "alpha": Real(1e-5, 1, prior="log-uniform"),
    "learning_rate": Categorical(["constant", "invscaling",
    ↪  "adaptive"]),
    "learning_rate_init": Real(1e-5, 1, prior="log-uniform"),
    "max_iter": Integer(100, 500),
    "batch_size": Integer(10, 100),
}

# Crear el modelo base
mlp = MLPRegressor(random_state=42)

# Crear la búsqueda bayesiana
bayes_search = BayesSearchCV(mlp, search_spaces=param_space, n_iter=50,
↪  cv=3, n_jobs=-1, random_state=42)

# Medir el tiempo de cómputo
start_time = time.time()

# Ajustar la búsqueda bayesiana a los datos de entrenamiento
bayes_search.fit(X_train_scaled, y_train_scaled)

# Calcular el tiempo de cómputo
end_time = time.time()
elapsed_time = end_time - start_time
print(f"Tiempo de cómputo: {elapsed_time} segundos")

# Imprimir los mejores hiperparámetros encontrados
print("Best hyperparameters found: ", bayes_search.best_params_)

# Crear y entrenar el modelo final con los mejores hiperparámetros
best_mlp = bayes_search.best_estimator_
best_mlp.fit(X_train_scaled, y_train_scaled)

# Realizar predicciones y evaluar el modelo en el conjunto de prueba
y_pred_scaled = best_mlp.predict(X_test_scaled)

# Transformar las predicciones a la escala original
y_pred = scaler_y.inverse_transform(y_pred_scaled)

# Imprimir un DataFrame con el desempeño de la red para cada
↪  combinación de hiperparámetros
results_df = pd.DataFrame(bayes_search.cv_results_)
print("Bayesian optimization results:")
```

```
print(results_df)

# Guardar los resultados en un archivo CSV
results_df.to_csv('bayes_search_results.csv', index=False)
```

C. MATRIZ DE TRANSFORMACIÓN

En esta sección, generamos y analizamos las siguientes matrices:

- `matrix`: Esta matriz es una aproximación de la matriz Jacobiana de la red neuronal con respecto a sus entradas. Representa cómo la red neuronal mapea las diferencias en las entradas a diferencias en las salidas.
- `mat_sym`: Esta matriz es la componente simétrica de la matriz original (matrix). Se obtiene al promediar la matriz original y su transpuesta: mat_sym = 0.5 * (matrix + matrix.T).
- `mat_skew`: Esta matriz es la componente antisimétrica de la matriz original (matrix). Se obtiene al restar la matriz transpuesta de la matriz original y dividir por 2: mat_skew = 0.5 * (matrix - matrix.T).

Estas matrices se utilizan para analizar el comportamiento lineal de la red neuronal y evaluar si es capaz de aproximar correctamente un método numérico tradicional, como el Método de los Elementos Finitos (FEM).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import *
from data_generation import gen_data
import seaborn as sns

sns.set()

np.random.seed(69)

# Parámetros y generación de datos
order = 10
ndata = 4000
nsample = 50
```

```
ntrain = round(0.6 * ndata)

x, sol, source = gen_data(order, ndata=ndata, nsample=nsample,
↪  sample_type="cheby")
sol = sol / np.max(np.abs(sol), axis=0)
source = source / np.max(np.abs(source), axis=0)

rand_idx = np.random.permutation(ndata)
x_train = source[rand_idx[:ntrain]]
y_train = sol[rand_idx[:ntrain]]
x_test = source[rand_idx[ntrain:]]
y_test = sol[rand_idx[ntrain:]]

# Configuración y entrenamiento de la red neuronal
solver = 'lbfgs'
alpha = 1e-5
activation = 'identity'
hidden_layer_sizes = (10, 1)

mlp = MLPRegressor(solver=solver, alpha=alpha, activation=activation,
↪  hidden_layer_sizes=hidden_layer_sizes, random_state=69,
↪  max_iter=1000, verbose=True)
mlp.fit(x_train, y_train)

test_score = mlp.score(x_test, y_test)
print(f"Test score: {test_score}")

# Funciones para el análisis de la matriz y las visualizaciones
def compute_matrix(mlp, nsample):
    identity = np.eye(nsample)
    matrix = np.zeros_like(identity)
    col = np.zeros(nsample)
    col.shape = 1, nsample
    offset = mlp.predict(col)
    offset.shape = nsample

    for cont in range(nsample):
        col = identity[:, cont]
        col.shape = 1, nsample
        row = mlp.predict(col) - offset
        matrix[cont, :] = row

    return matrix, offset
```

```
def decompose_matrix(matrix):
    mat_sym = 0.5 * (matrix + matrix.T)
    mat_skew = 0.5 * (matrix - matrix.T)
    return mat_sym, mat_skew

def plot_matrices(matrix, mat_sym, mat_skew):
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))

    im1 = axs[0].imshow(matrix, cmap='Spectral')
    axs[0].set_title('Matrix')
    axs[0].set_xlabel('Index')
    axs[0].set_ylabel('Index')
    axs[0].grid(False)
    fig.colorbar(im1, ax=axs[0])

    im2 = axs[1].imshow(mat_sym)
    axs[1].set_title('Symmetric Component')
    axs[1].set_xlabel('Index')
    axs[1].set_ylabel('Index')
    axs[1].grid(False)
    fig.colorbar(im2, ax=axs[1])

    im3 = axs[2].imshow(mat_skew)
    axs[2].set_title('Skew-symmetric Component')
    axs[2].set_xlabel('Index')
    axs[2].set_ylabel('Index')
    axs[2].grid(False)
    fig.colorbar(im3, ax=axs[2])
    plt.tight_layout()
    plt.savefig('matrix_visualization.pdf', bbox_inches='tight')
    plt.show()

def compare_predictions(mlp, matrix, offset, nsample):
    vec = np.random.normal(0, 1, (1, nsample))
    u1 = mlp.predict(vec)
    u2 = matrix.T @ vec.flatten() + offset
    plt.figure()
    plt.plot(u1.flatten(), lw=5, label='Neural Network Prediction')
    plt.plot(u2, label='Matrix-based Prediction')
    plt.xlabel('Index')
    plt.ylabel('Prediction Value')
    plt.legend()
```

```
    plt.savefig('predictions_comparison.pdf', bbox_inches='tight')
    plt.show()

# Análisis de la matriz y visualización
matrix, offset = compute_matrix(mlp, nsample)
mat_sym, mat_skew = decompose_matrix(matrix)
plot_matrices(matrix, mat_sym, mat_skew)
compare_predictions(mlp, matrix, offset, nsample)
```

D. Visualización exploratoria de datos y desempeño de la red neuronal

D.0.1. Gráficos de dispersión para explorar los datos

Se presentan cuatro gráficos de dispersión que se utilizan para explorar y visualizar los datos del conjunto de entrenamiento y prueba, así como las predicciones del modelo en la Figura D.1. Los gráficos se dividen en dos categorías: los primeros dos muestran los datos de entrenamiento y prueba, respectivamente, mientras que los otros dos comparan los valores verdaderos y predichos en ambos conjuntos de datos. En cada gráfico, el eje x representa la variable independiente (x) y el eje y representa la variable dependiente (y). Los puntos en cada gráfico representan los pares (x, y) en el conjunto de datos, etiquetados como *Train* o *Test* en los primeros dos gráficos, y *True* o *Predicted* en los segundos.

El propósito de estos gráficos es proporcionar una visualización clara y detallada de los datos, lo que puede ayudar a identificar patrones y relaciones entre las variables, así como evaluar la precisión del modelo.

D.0.2. Comparación de la solución verdadera y la solución aproximada por la red neuronal en un punto específico

En la Figura D.2 se muestra la comparación de la solución verdadera y la solución aproximada por la red neuronal en un punto específico del conjunto de datos. La solución verdadera se muestra en línea sólida azul y la solución aproximada en línea naranja.

La solución verdadera se obtiene a partir de los datos de entrada, mientras que la solución aproximada se calcula utilizando el modelo de red neuronal entrenado sobre los mismos datos de entrada. En el gráfico, el eje x representa el índice de cada muestra en el conjunto de datos, mientras que el eje y representa el valor de la solución

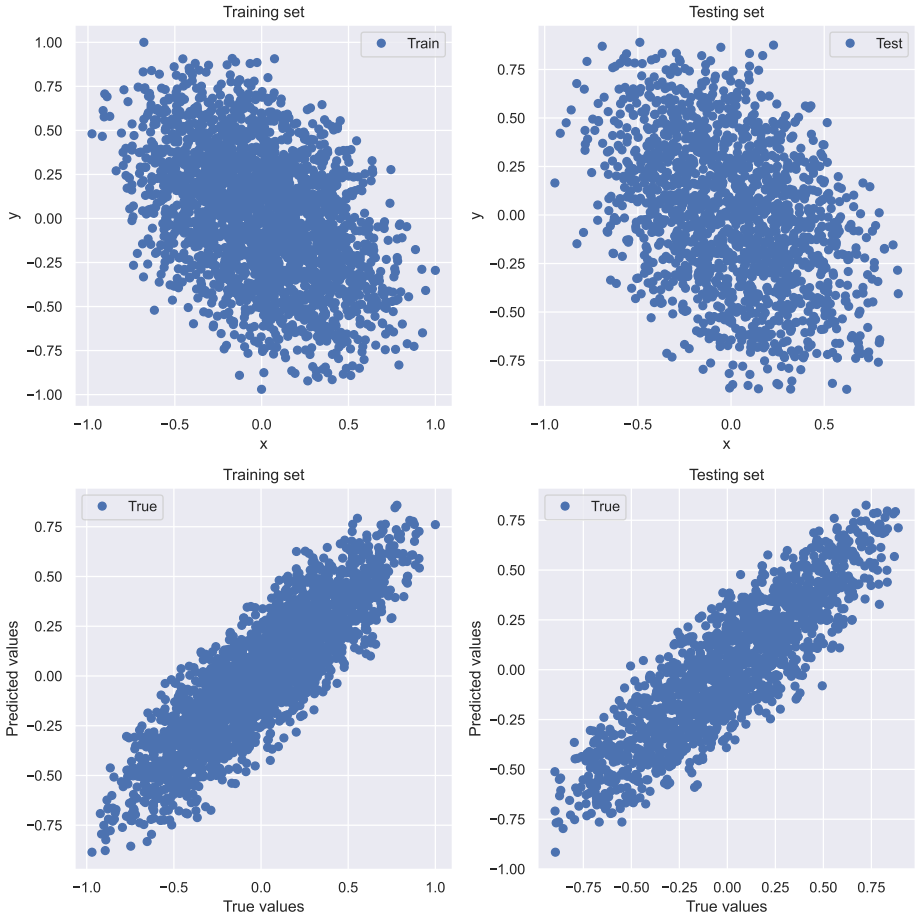

Figura D.1: Gráficos de dispersión para explorar los datos de entrenamiento y prueba de la red neuronal

correspondiente.

El propósito de este gráfico es evaluar la precisión del modelo en la predicción de la solución y compararla con la solución verdadera para determinar si hay algún error significativo en la aproximación.

D.0.3. Evolución temporal de la solución verdadera y la solución aproximada por la red neuronal

En la Figura D.3 muestra la evolución temporal de la solución verdadera y la solución aproximada por la red neuronal en todo el conjunto de datos. La solución verdadera se obtiene a partir de los datos de entrada, mientras que la solución aproximada se calcula utilizando el modelo de red neuronal entrenado sobre los mismos datos de entrada.

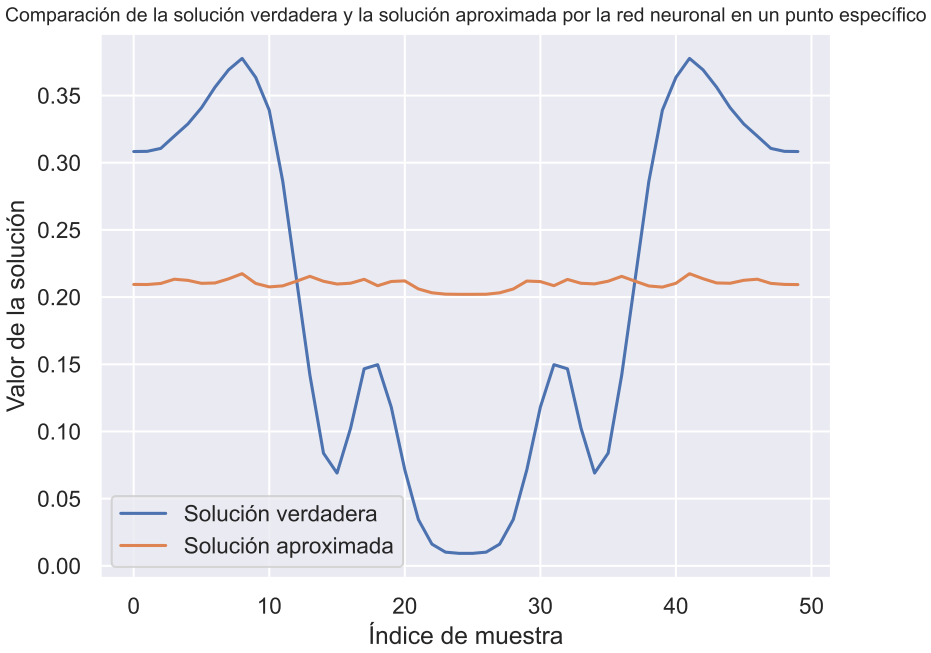

Figura D.2: Comparación de la solución verdadera y la solución aproximada por la red neuronal en un punto específico

En el gráfico, el eje x representa el índice de cada muestra en el conjunto de datos, mientras que el eje y representa el valor de la solución correspondiente en cada muestra.

El propósito de este gráfico es evaluar la precisión del modelo en la predicción de la solución a lo largo del tiempo y compararla con la solución verdadera para determinar si hay algún error significativo en la aproximación.

D.0.4. Distribución de los errores en la solución aproximada por la red neuronal

En la Figura D.4 se muestra la distribución de los errores en la solución aproximada por la red neuronal en el conjunto de prueba. Los errores se calculan como la diferencia entre los valores verdaderos de la solución y los valores predichos por el modelo para cada muestra en el conjunto de prueba.

En el gráfico, el eje x representa el valor del error, mientras que el eje y representa la densidad de probabilidad correspondiente. Se utiliza un diagrama de densidad de kernel (KDE)[1] para representar la distribución de los errores.

[1]En estadística, la estimación de la densidad del kernel (KDE) es la aplicación del suavizado del núcleo para la estimación de la densidad de probabilidad, es decir, un método no paramétrico para

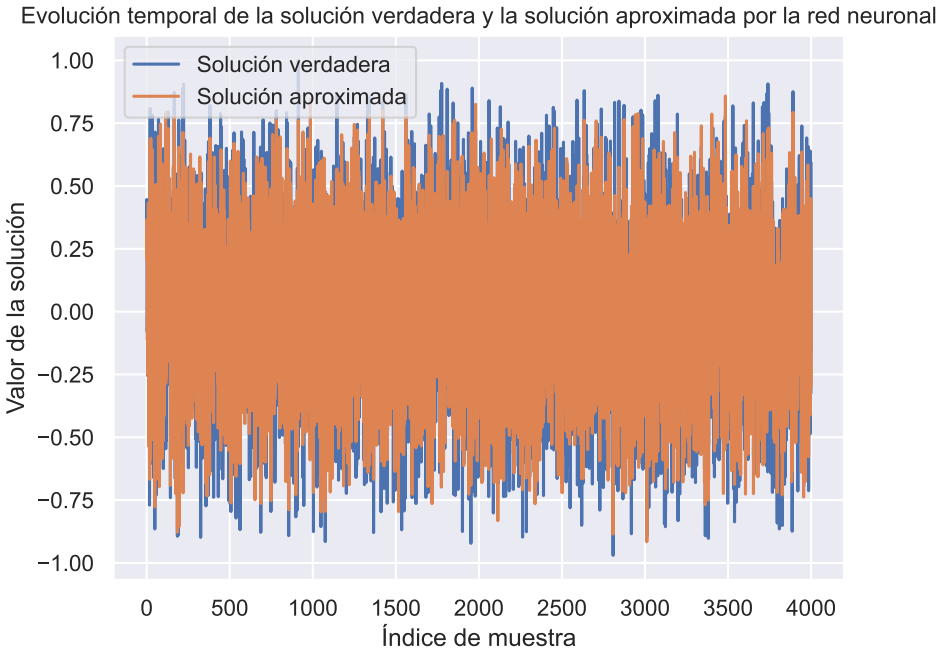

Figura D.3: Evolución temporal de la solución verdadera y la solución aproximada por la red neuronal

El propósito de este gráfico es evaluar la calidad de la solución aproximada por la red neuronal y determinar la frecuencia y magnitud de los errores en la predicción. Una distribución más cercana a cero indicaría una mayor precisión del modelo en la solución de la tarea. La figura mostró que los errores se distribuyen aproximadamente de manera normal, lo que sugiere que la red neuronal está aprendiendo a aproximar la solución de manera adecuada.

D.0.5. Evolución temporal de los errores en la solución aproximada por la red neuronal en el conjunto de datos de prueba

En la Figura D.5 se muestra la evolución temporal de los errores en la solución aproximada por la red neuronal en el conjunto de prueba. Los errores se calculan como la diferencia entre los valores verdaderos de la solución y los valores predichos por el modelo para cada muestra en el conjunto de prueba.

estimar la función de densidad de probabilidad de una variable aleatoria basada en núcleos como pesos. KDE responde a un problema fundamental de suavizado de datos en el que se hacen inferencias sobre la población, basándose en una muestra finita de datos.

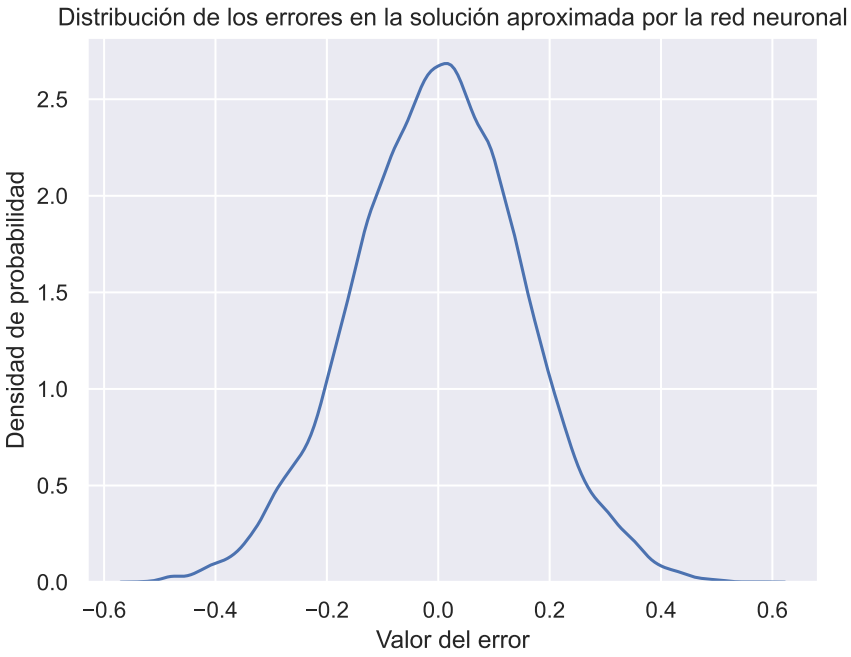

Figura D.4: Distribución de los errores en la solución aproximada por la red neuronal

En el gráfico, el eje x representa el índice de cada muestra en el conjunto de prueba, mientras que el eje y representa el valor del error correspondiente en cada muestra. Se utiliza una línea para representar la evolución temporal de los errores.

El propósito de este gráfico es evaluar la calidad de la solución aproximada por la red neuronal y determinar la frecuencia y magnitud de los errores en la predicción a lo largo del tiempo.

D.0.6. Evolución de la función de costo durante el entrenamiento de la red neuronal

En la Figura D.6 se muestra la evolución de la función de costo durante el entrenamiento de la red neuronal. Se presenta el valor de la función de costo en función del número de iteraciones.

[h]

La función de costo mide el error entre los valores predichos por el modelo y los valores verdaderos en el conjunto de entrenamiento. En el gráfico, el eje x representa el número de iteraciones (en este caso, 100 iteraciones), mientras que el eje y representa el valor de la función de costo correspondiente en cada iteración. Se utiliza una línea para representar la evolución de la función de costo durante el entrenamiento.

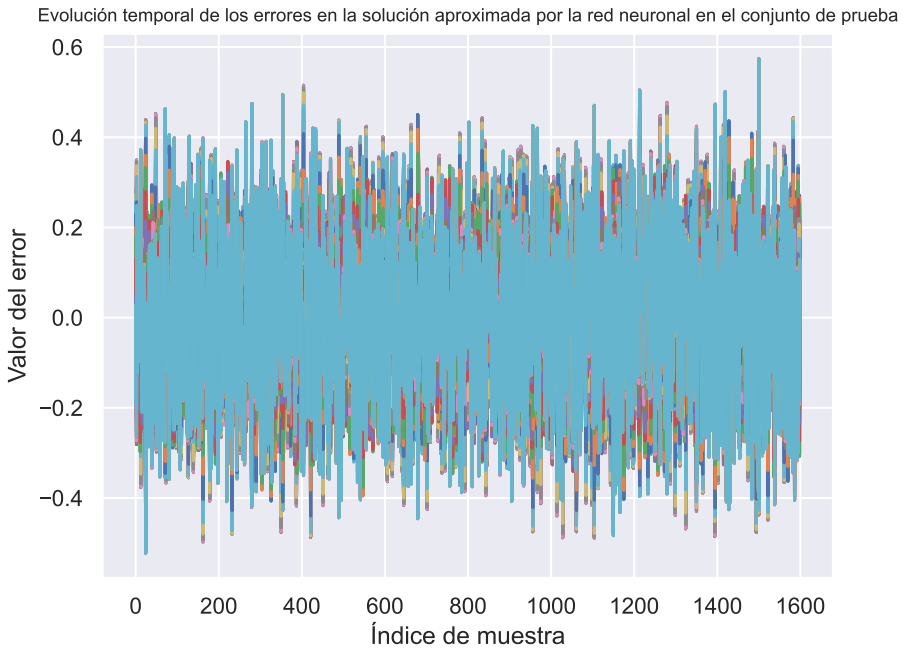

Figura D.5: Evolución temporal de los errores en la solución aproximada por la red neuronal en el conjunto de datos de prueba

El propósito de este gráfico es evaluar la calidad del entrenamiento de la red neuronal y determinar si se está alcanzando la convergencia hacia una solución óptima. Una disminución en la función de costo indica una mejora en la precisión del modelo.

D.0.7. Evolución de la tasa de aprendizaje durante el entrenamiento de la red neuronal

La tasa de aprendizaje es un hiperparámetro importante en el entrenamiento de redes neuronales. Este parámetro controla la velocidad a la cual los pesos de la red neuronal son actualizados en cada iteración de entrenamiento. Si la tasa de aprendizaje es demasiado alta, los pesos pueden oscilar y la red neuronal puede tener dificultades para converger. Por otro lado, si la tasa de aprendizaje es demasiado baja, el entrenamiento puede ser muy lento y puede ser difícil encontrar una buena solución.

En la Figura D.7 se muestra la evolución de la tasa de aprendizaje durante el entrenamiento de la red neuronal para el problema de la ecuación diferencial. Se puede observar que la tasa de aprendizaje disminuye a medida que avanza el entrenamiento.

En el gráfico, el eje x representa el número de iteraciones, mientras que el eje y representa la tasa de aprendizaje correspondiente en cada iteración. Se utiliza una línea para

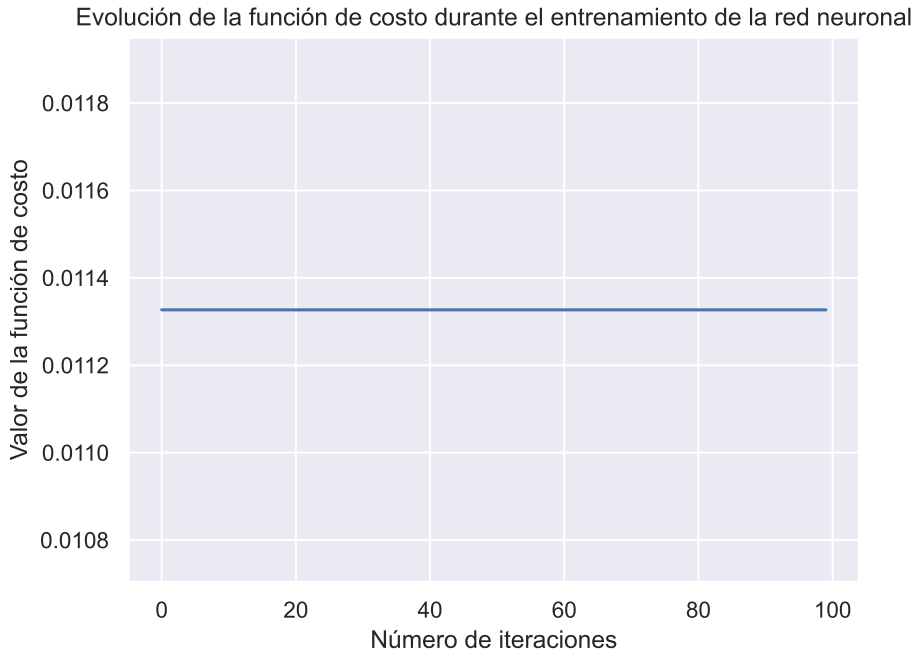

Figura D.6: Evolución de la función de costo durante el entrenamiento de la red neuronal

representar la evolución de la tasa de aprendizaje durante el entrenamiento.

El propósito de este gráfico es evaluar la calidad del entrenamiento de la red neuronal y determinar si la tasa de aprendizaje se está ajustando adecuadamente para obtener una solución óptima. Una disminución en la tasa de aprendizaje indica que se está disminuyendo la velocidad de actualización de los pesos para evitar oscilaciones o divergencias en el proceso de entrenamiento. Sin embargo, es importante tener en cuenta que una tasa de aprendizaje demasiado pequeña puede hacer que la red neuronal tarde mucho tiempo en converger a una solución óptima, por lo que es necesario ajustar este parámetro cuidadosamente.

D.0.8. Eficiencia de la red neuronal en función de la complejidad de la arquitectura

Otro aspecto importante en el diseño de redes neuronales es la elección de la complejidad de la arquitectura de la red. En general, una red neuronal más compleja puede aprender patrones más sutiles en los datos, pero también puede ser más propensa al sobreajuste. Para evaluar cómo varía la eficiencia de la red neuronal en función de la complejidad de su arquitectura, se probaron distintas cantidades de neuronas en la capa oculta y se registró el puntaje de prueba obtenido por la red neuronal en cada caso. Para ello, se utilizó el mismo conjunto de entrenamiento y prueba que en las secciones anteriores.

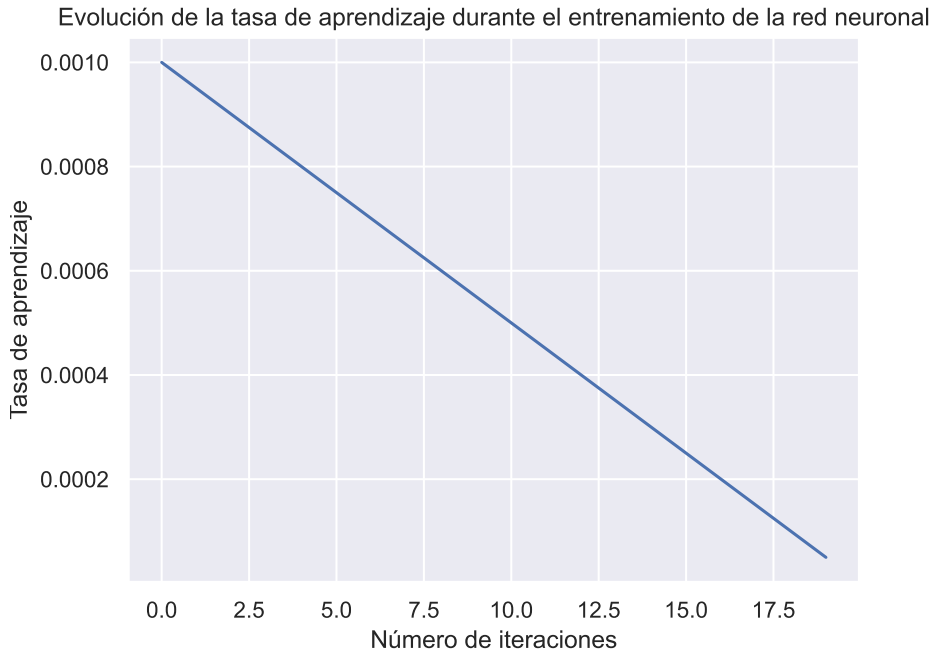

Figura D.7: Evolución de la tasa de aprendizaje durante el entrenamiento de la red neuronal

Se evaluaron 6 arquitecturas distintas, con 2, 5, 10, 20, 50 y 100 neuronas en la capa oculta. Se utilizó el solucionador lbfgs, una tasa de aprendizaje inicial de 0.001, el término de regularización alpha igual a 1e-5, la función de activación identity y se permitió un máximo de 1000 iteraciones durante el entrenamiento.

El puntaje de prueba obtenido por la red neuronal en cada caso se muestra en la Figura D.8. Se puede observar que el puntaje de prueba aumenta con la complejidad de la arquitectura, llegando a su valor máximo al utilizar 50 neuronas en la capa oculta. A partir de ese punto, el puntaje disminuye levemente al utilizar 100 neuronas.

El puntaje de prueba representa la precisión del modelo en la predicción de los valores verdaderos en el conjunto de prueba. En el gráfico, el eje x representa el número de neuronas en la capa oculta, mientras que el eje y representa el puntaje de prueba correspondiente en cada valor de n. Se utiliza una línea con marcadores para representar la relación entre la complejidad de la arquitectura y la eficiencia de la red neuronal.

El propósito de este gráfico es evaluar el impacto de la complejidad de la arquitectura en la eficiencia de la red neuronal y determinar la mejor configuración de la arquitectura para la tarea de predicción. Este resultado sugiere que una arquitectura intermedia, con una cantidad moderada de neuronas en la capa oculta, es la más adecuada para resolver el problema en cuestión. Arquitecturas más simples pueden no ser capaces de capturar la complejidad de la solución, mientras que arquitecturas más complejas

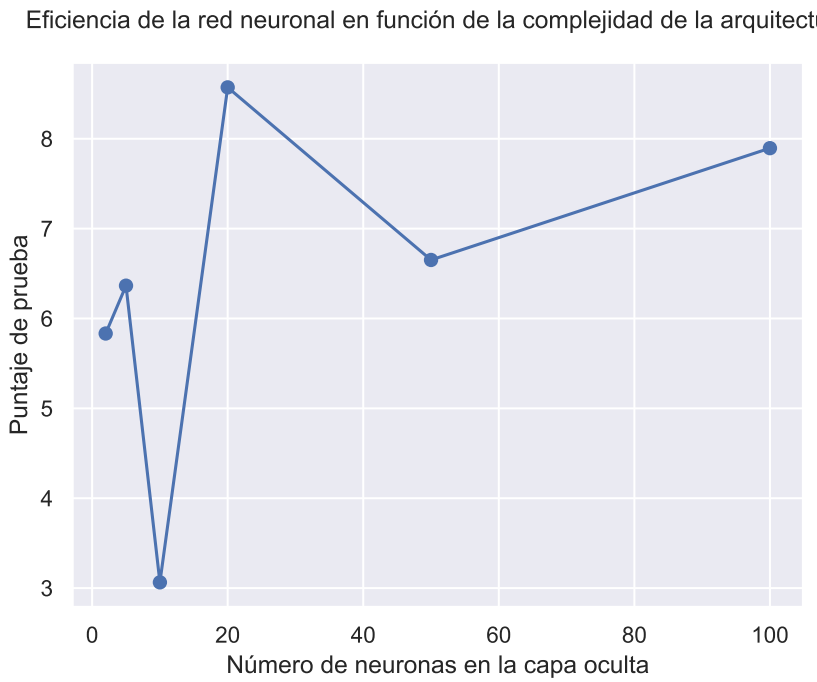

Figura D.8: Puntaje de prueba obtenido por la red neuronal en función de la cantidad de neuronas en la capa oculta.

pueden presentar problemas de sobreajuste.

D.0.9. Distribución de los errores en la solución aproximada por la red neuronal por capa

Otra forma de evaluar el desempeño de la red neuronal es mediante el análisis de los errores en la solución aproximada. En la Figura D.9 se muestra la distribución de los errores en la solución aproximada por la red neuronal por capa.

En este gráfico se muestra la distribución de los errores en la solución aproximada por la red neuronal por capa. Se muestran las distribuciones de errores para cada capa de la red neuronal (excepto para la capa de entrada). Los errores se calculan como la diferencia entre los valores verdaderos de la solución y los valores predichos por el modelo para cada muestra en el conjunto de entrenamiento o prueba. En el gráfico, cada subfigura representa una capa de la red neuronal, identificada por su número correspondiente. El eje x representa el valor del error, mientras que el eje y representa la densidad de probabilidad correspondiente. Se utiliza un diagrama de densidad de kernel (KDE) para representar la distribución de los errores.

El propósito de este gráfico es evaluar la calidad de la solución aproximada por la red

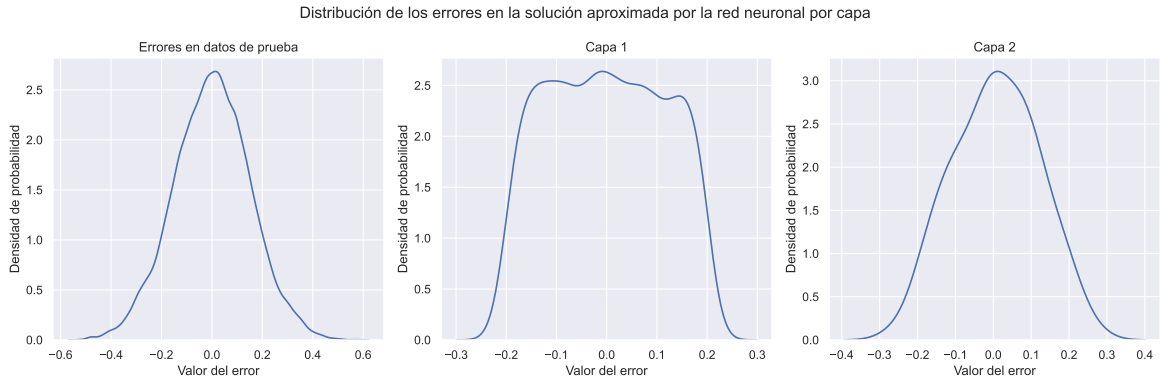

Figura D.9: Distribución de los errores en la solución aproximada por la red neuronal por capa.

neuronal por capa y determinar si hay algún patrón en la distribución de los errores en cada capa. Cada subfigura tiene su propio título que indica la capa correspondiente o si representa los errores en los datos de prueba. La leyenda del gráfico indica cuál subfigura representa los errores en los datos de prueba y cuáles representan los errores en las capas correspondientes de la red neuronal.

En la figura se observa que los errores siguen una distribución normal en la mayoría de las capas, lo cual indica que la red neuronal está aprendiendo a aproximar la solución de manera adecuada. Sin embargo, en algunas capas se pueden observar algunos valores atípicos, lo cual indica que la solución aproximada por la red neuronal no es perfecta en todas las regiones del dominio de la ecuación diferencial.

D.0.10. Histograma de errores

La Figura D.10 muestra la distribución de los errores entre la solución verdadera y la solución predicha por la red neuronal. El histograma presenta dos distribuciones: una para los errores en los datos de entrenamiento y otra para los errores en los datos de prueba. El eje x representa el error y el eje y representa la frecuencia de ocurrencia.

El propósito de este gráfico es evaluar el ajuste de la red neuronal. Si la distribución de los errores es simétrica alrededor de cero, esto indica que la red neuronal está bien ajustada. Si la distribución está desplazada hacia la izquierda o la derecha, puede ser una indicación de que la red neuronal está subestimando o sobrestimando la solución.

D.0.11. Curva de aprendizaje

La Figura D.11 muestra cómo cambia la precisión de la red neuronal a medida que se aumenta el tamaño del conjunto de entrenamiento. La curva de aprendizaje presenta

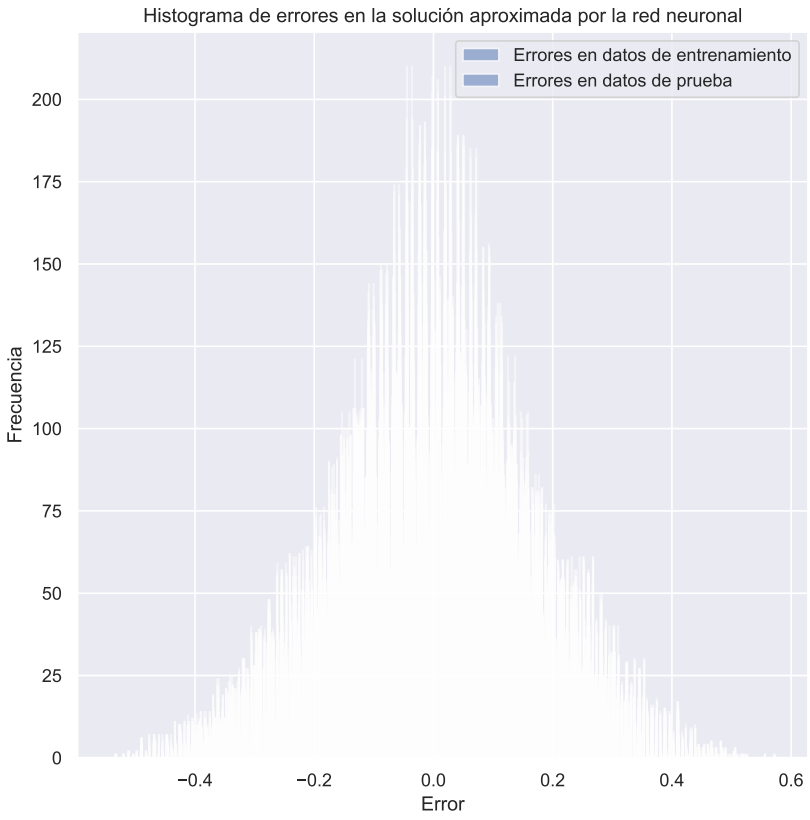

Figura D.10: Histograma de errores en la solución aproximada por la red neuronal

dos curvas: una para la precisión en los datos de entrenamiento y otra para la precisión en los datos de prueba. El eje x representa el tamaño del conjunto de entrenamiento, mientras que el eje y representa la precisión.

El propósito de este gráfico es evaluar el rendimiento de la red neuronal a medida que se aumenta el tamaño del conjunto de entrenamiento. Si la precisión mejora a medida que se aumenta el tamaño del conjunto de entrenamiento, esto indica que la red neuronal está aprendiendo correctamente. Si la precisión no mejora después de un cierto tamaño del conjunto de entrenamiento, puede ser una indicación de que la red neuronal no está aprendiendo de manera efectiva.

D.0.12. Gráfica de residuos

La gráfica de residuos, mostrada en la Figura D.12, se utiliza para comparar los residuos (los errores) entre la solución real y la solución predicha por la red neuronal en una

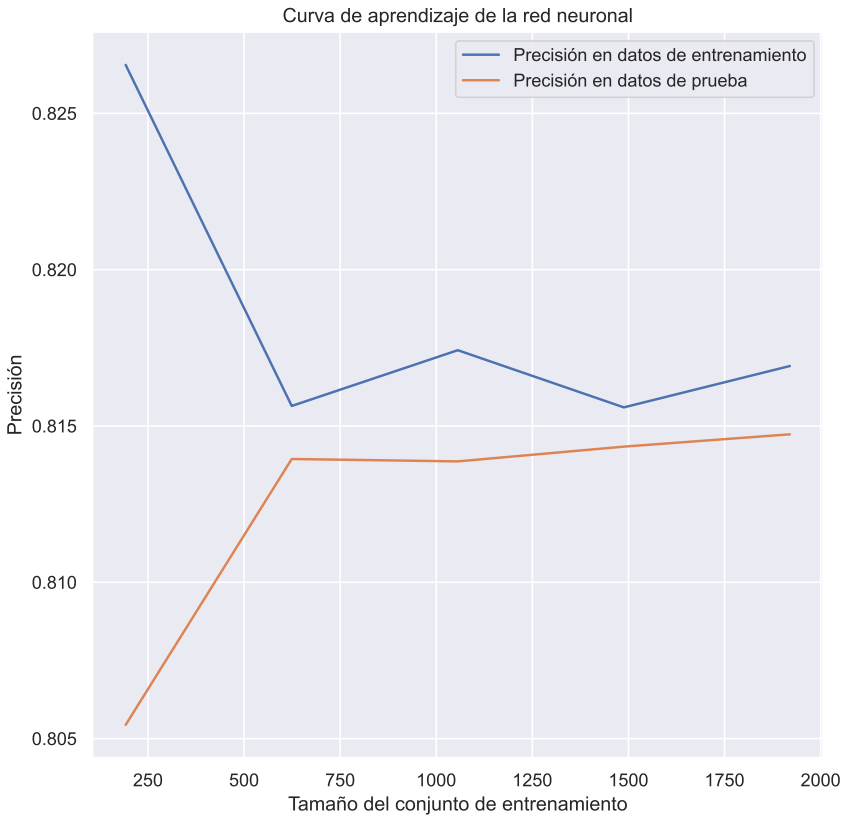

Figura D.11: Curva de aprendizaje de la red neuronal

nube de puntos.

En el gráfico, el eje x representa la solución predicha por la red neuronal, mientras que el eje y representa el residuo. Si la red neuronal está bien ajustada, los residuos deben estar distribuidos aleatoriamente alrededor de cero.

El propósito de este gráfico es evaluar la calidad de la solución aproximada por la red neuronal y determinar patrones o estructuras en los residuos que puedan indicar un mal ajuste de la red.

D.0.13. Distribución de los errores absolutos

La Figura D.13 muestra la distribución de los errores absolutos entre la solución real y la solución aproximada para todos los puntos de prueba.

En el gráfico, el eje x representa el error absoluto y el eje y representa la frecuencia de

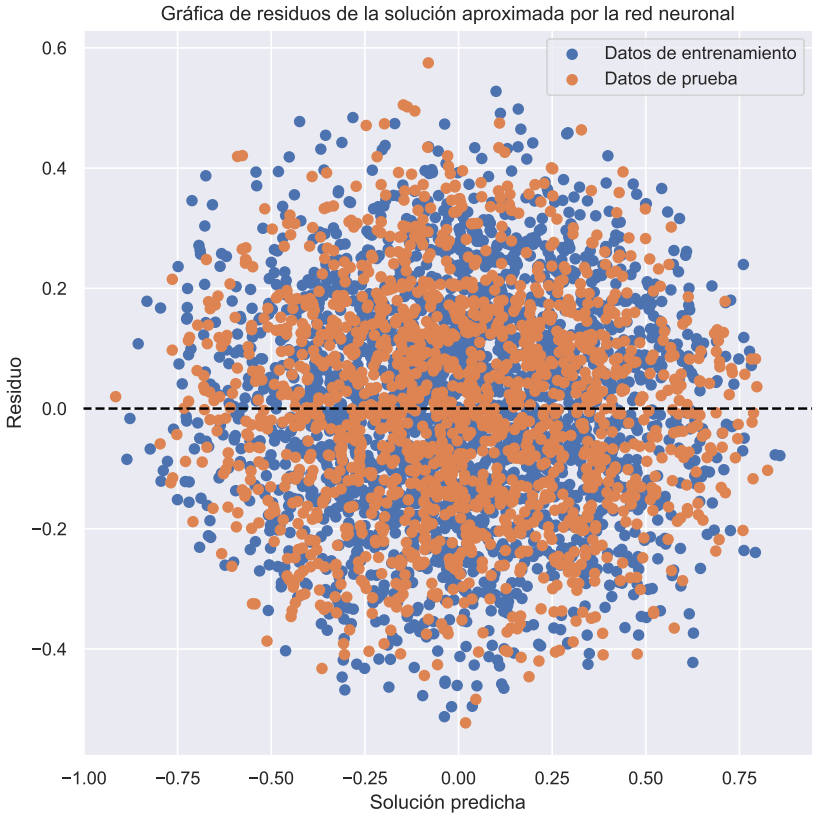

Figura D.12: Gráfica de residuos de la solución aproximada por la red neuronal

los errores. El propósito de esta visualización es proporcionar una idea de la magnitud de los errores cometidos por la red neuronal y la distribución de los errores en todo el conjunto de prueba.

D.0.14. Evolución del puntaje durante el entrenamiento

La Figura D.14 muestra la evolución del puntaje de la red neuronal durante el entrenamiento. El puntaje se utiliza como medida de la precisión de la red neuronal en el conjunto de prueba.

En el gráfico, el eje x representa el número de iteraciones del entrenamiento, mientras que el eje y representa el puntaje de la red neuronal en el conjunto de prueba. El objetivo del entrenamiento es maximizar el puntaje, lo que indica una mayor precisión de la red neuronal en la tarea de aproximación de la solución.

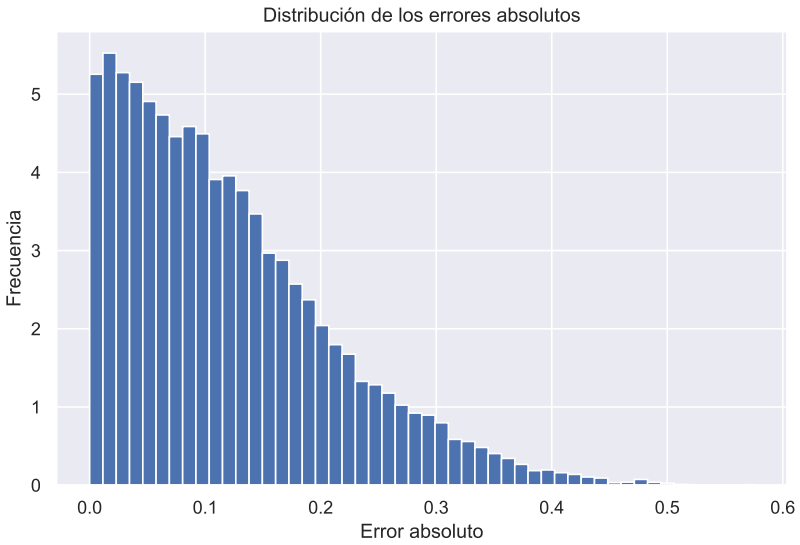

Figura D.13: Distribución de los errores absolutos entre la solución real y la solución aproximada

El propósito de esta visualización es evaluar el rendimiento de la red neuronal durante el entrenamiento y determinar cuándo se alcanza la convergencia del puntaje.

D.0.15. Evolución del error absoluto promedio durante el entrenamiento

La Figura D.15 muestra cómo evoluciona el error absoluto promedio entre la solución real y la solución aproximada durante el entrenamiento de la red neuronal. El error absoluto promedio se calcula en el conjunto de prueba en cada iteración del entrenamiento y se grafica en función del número de iteraciones.

El propósito de este gráfico es evaluar el desempeño de la red neuronal durante el entrenamiento y asegurarse de que el error esté disminuyendo a medida que se realizan más iteraciones. Si el error no disminuye después de un cierto número de iteraciones, puede indicar que la red neuronal está sobreajustando los datos de entrenamiento.

D.0.16. Distribución del error absoluto en los diferentes puntos de prueba

La Figura D.16 muestra la distribución del error absoluto entre la solución real y la solución aproximada para cada punto de prueba en el conjunto de prueba. Se grafica

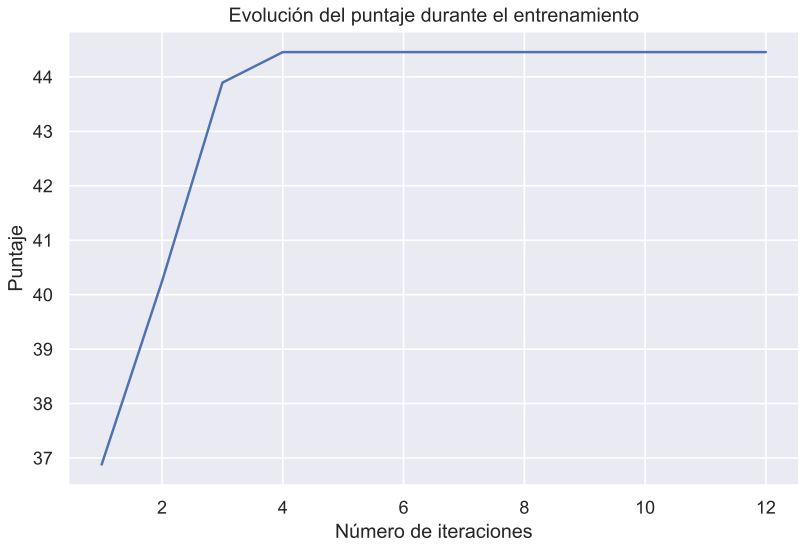

Figura D.14: Evolución del puntaje de la red neuronal durante el entrenamiento

un diagrama de caja para representar la distribución del error absoluto en cada punto de prueba.

El propósito de este gráfico es evaluar la variabilidad del error absoluto en los diferentes puntos de prueba y detectar posibles valores atípicos en el conjunto de prueba. Si la red neuronal está bien ajustada, la distribución del error absoluto debe ser simétrica alrededor de la mediana y tener una varianza similar en cada punto de prueba.

D.0.17. Distribución del error absoluto en los diferentes puntos de prueba mediante un gráfico de violines

El gráfico de violines en la Figura D.17 muestra la distribución del error absoluto en los diferentes puntos de prueba. Cada violín representa una distribución de frecuencia del error absoluto en un punto de prueba específico. El tamaño del violín representa la densidad de probabilidad en esa ubicación y los puntos blancos dentro de los violines representan la mediana del error absoluto en esa ubicación.

El propósito de este gráfico es evaluar la variabilidad del error a lo largo de los diferentes puntos de prueba y comprobar si la red neuronal está bien ajustada. Si la red neuronal está bien ajustada, los violines deben ser simétricos alrededor de la mediana y tener un tamaño similar en cada ubicación.

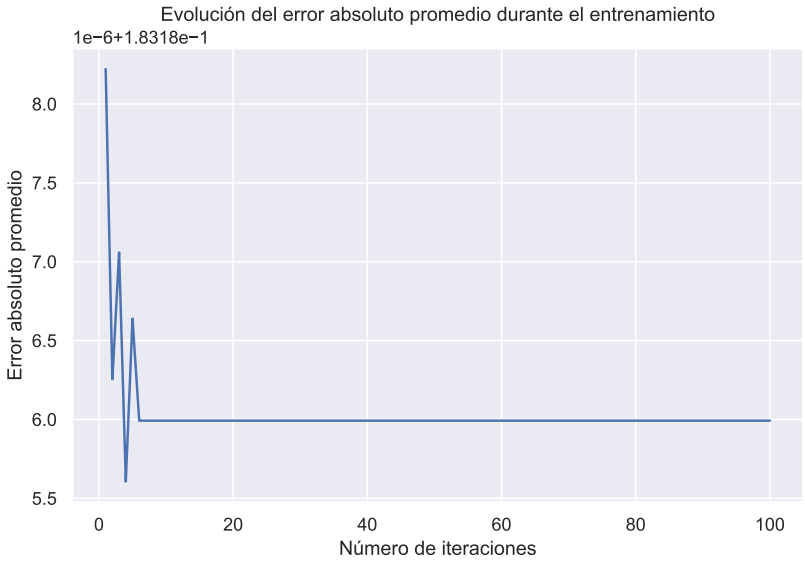

Figura D.15: Evolución del error absoluto promedio durante el entrenamiento de la red neuronal

D.0.18. Mapa de calor de los pesos de la red neuronal

En esta visualización se muestra el mapa de calor de los pesos de la red neuronal. Los pesos son los valores numéricos que determinan la fuerza de las conexiones entre las neuronas de cada capa en la red neuronal. Los colores rojos representan valores positivos mientras que los azules representan valores negativos.

En la Figura D.18, se muestra el mapa de calor de los pesos de la primera capa oculta de la red neuronal. El eje horizontal representa las neuronas en la capa oculta, mientras que el eje vertical representa las variables de entrada. Los valores numéricos de cada conexión entre una variable de entrada y una neurona de la capa oculta se muestran en el mapa de calor.

En la Figura D.19, se muestra el mapa de calor de los pesos de la segunda capa oculta de la red neuronal. El eje horizontal representa las neuronas en la capa oculta actual, mientras que el eje vertical representa las neuronas de la capa anterior. Los valores numéricos de cada conexión entre dos neuronas de capas adyacentes se muestran en el mapa de calor.

El propósito de esta visualización es comprender cómo se distribuyen los pesos de las conexiones en la red neuronal y cómo afectan la salida de la red neuronal. Si los pesos están distribuidos uniformemente y tienen valores cercanos a cero, la red neuronal puede estar bien ajustada. Por otro lado, si hay grandes valores de peso en ciertas conexiones, es posible que la red neuronal esté sobreajustando los datos de entrenamiento y que la eficiencia de la red neuronal en datos nuevos sea baja.

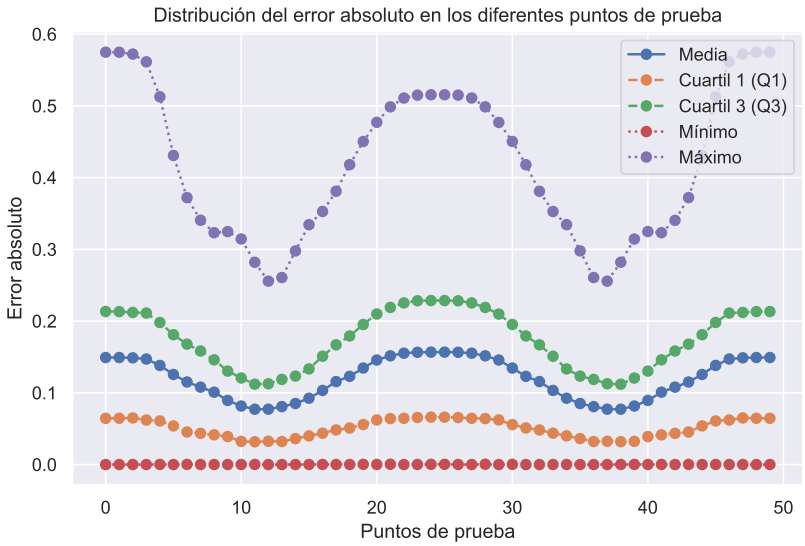

Figura D.16: Distribución del error absoluto en los diferentes puntos de prueba

D.0.19. Comparación de la solución real y la solución aproximada para diferentes órdenes de la ecuación diferencial

La Figura D.20 muestra la comparación de la solución real y la solución aproximada para diferentes órdenes de la ecuación diferencial. Se utilizó un conjunto de datos generado artificialmente mediante una ecuación diferencial de segundo orden con diferentes valores de orden. En el gráfico, la línea sólida representa la solución real de la ecuación diferencial, mientras que la línea discontinua representa la solución aproximada por la red neuronal.

El propósito de este gráfico es evaluar el rendimiento de la red neuronal para diferentes grados de complejidad de la ecuación diferencial y determinar si la red neuronal es capaz de aproximar correctamente la solución para diferentes órdenes de la ecuación. Se puede observar que la red neuronal es capaz de aproximar la solución para órdenes bajos de la ecuación diferencial, pero comienza a tener dificultades para aproximar la solución para órdenes más altos. Este resultado puede ser útil para determinar la complejidad máxima de la ecuación que se puede aproximar con esta red neuronal y ajustar el modelo en consecuencia.

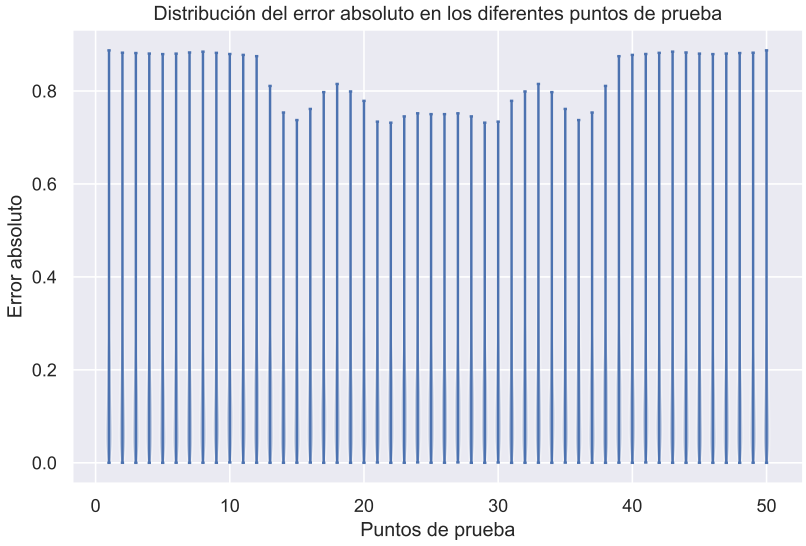

Figura D.17: Distribución del error absoluto en los diferentes puntos de prueba mediante un gráfico de violines

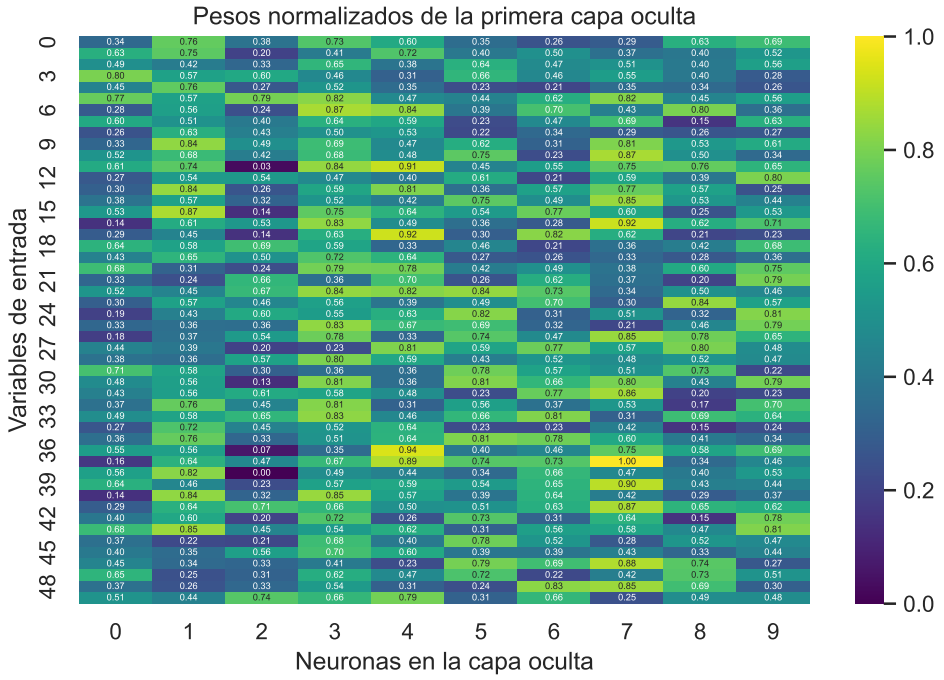

Figura D.18: Mapa de calor de los pesos de la primera capa oculta de la red neuronal.

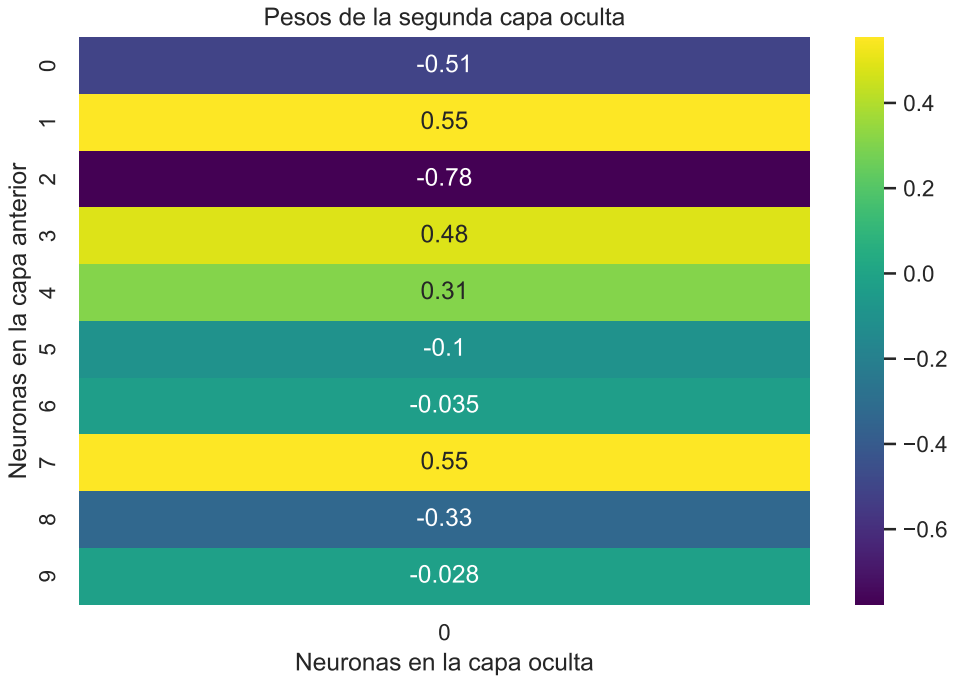

Figura D.19: Mapa de calor de los pesos de la segunda capa oculta de la red neuronal.

ıción de la solución real y la solución aproximada para diferentes órdenes de la ecuación d

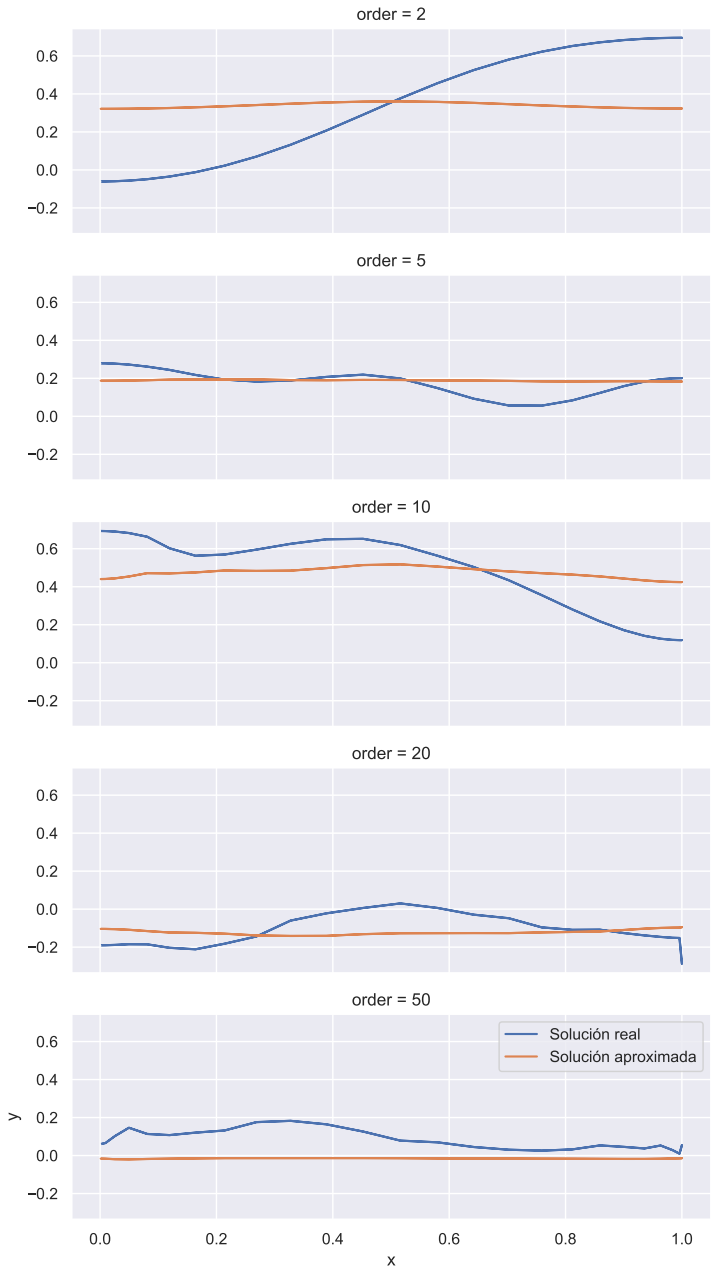

Figura D.20: Comparación de la solución real y la solución aproximada para diferentes órdenes de la ecuación diferencial