



Departamento de Matemáticas, Facultad de Ciencias  
Universidad Autónoma de Madrid

# Redes Neuronales: Aproximación de EDPs

TRABAJO DE FIN DE GRADO

Grado en Matemáticas

*Autor:* Luis Hebrero Garicano

*Tutor:* Julia Novo

Curso 2024-2025



## Resumen

Las redes neuronales son un modelo matemático inspirado en el funcionamiento cerebral que, en esencia, se utiliza para encontrar funciones: funciones que clasifican datos, que predicen valores o incluso que anticipan la siguiente palabra en una frase. En este trabajo, se estudiará cómo se puede aplicar esta capacidad de aproximación de funciones para resolver ecuaciones en derivadas parciales.

Se analizan distintas estrategias para construir redes neuronales capaces de resolver este tipo de ecuaciones, centrándonos especialmente en los métodos conocidos como Physics-Informed Neural Networks (PINNs) y en el enfoque Deep Ritz. A través de ejemplos concretos, se estudia cómo estas redes pueden entrenarse para minimizar funcionales asociados al problema original.

El objetivo principal es evaluar hasta qué punto estas herramientas pueden competir con los métodos numéricos tradicionales. A lo largo del trabajo, se contrastan las ventajas teóricas que ofrecen las redes neuronales —como la posibilidad de operar sin una malla explícita o adaptarse a geometrías complejas— con las limitaciones prácticas que surgen durante su implementación.

## Abstract

Neural networks are a mathematical model inspired by the functioning of the brain and are essentially used to approximate functions—functions that classify data, predict values, or even anticipate the next word in a sentence. This thesis explores how this function approximation capability can be applied to solving partial differential equations (PDEs).

Various strategies are analyzed for building neural networks capable of solving such equations, with a particular focus on methods known as Physics-Informed Neural Networks (PINNs) and the Deep Ritz approach. Through concrete examples, the study examines how these networks can be trained to minimize functions associated with the original problem.

The main objective is to assess to what extent these tools can compete with traditional numerical methods. Throughout the work, the theoretical advantages offered by neural networks—such as the ability to operate without an explicit mesh or to adapt to complex geometries—are compared with the practical limitations encountered during their implementation.



# Índice general

---

<b>1</b>	<b>Introducción y preliminares</b>	<b>1</b>
1.1	Introducción a las redes neuronales . . . . .	1
1.1.1	Comentarios sobre la función sigmoide . . . . .	3
1.2	Conceptos preliminares sobre las ecuaciones en derivadas parciales . .	5
1.3	Métodos numéricos tradicionales: el método de los elementos finitos . .	10
<b>2</b>	<b>Aproximación de EDPs mediante redes neuronales</b>	<b>15</b>
2.1	PINNs: Physics-Informed Neural Networks . . . . .	15
2.1.1	Introducción . . . . .	15
2.1.2	Formulación de las PINNs . . . . .	16
2.2	El “Deep Ritz Method” . . . . .	22
2.2.1	El problema elíptico autoadjunto . . . . .	22
2.2.2	Formulación del método “Deep Ritz” . . . . .	24
<b>3</b>	<b>Conclusiones</b>	<b>31</b>
<b>A</b>	<b>Código en Matlab para la red neuronal de clasificación</b>	<b>35</b>
<b>B</b>	<b>Código en Python para el método de los elementos finitos en un cuadrado</b>	<b>39</b>
<b>C</b>	<b>Código en Python para construir PINNs con la librería DeepXDE</b>	<b>41</b>
<b>D</b>	<b>Código en Python método DeepRitz</b>	<b>43</b>



# CAPÍTULO 1

## Introducción y preliminares

---

Para poder entender las aproximaciones a las EDPs mediante redes neuronales, es necesario tener un conocimiento previo de las redes neuronales y de las ecuaciones en derivadas parciales. En este capítulo, se introducirán los conceptos básicos de ambos temas, así como las herramientas matemáticas necesarias para comprender el resto del trabajo.

### 1.1. Introducción a las redes neuronales

Una red neuronal, de forma abstracta, es simplemente una función que toma una entrada y produce una salida. Es decir, una red neuronal, es una función  $F$  que toma un vector de entrada  $x$  y produce un vector de salida  $y$ , siendo  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ . La red neuronal se compone de una serie de capas, cada una de las cuales está formada por un conjunto de neuronas. Cada neurona de una capa recibe una serie de entradas, las procesa y produce una salida. La salida de cada neurona se calcula mediante una función de activación, que puede ser de distintos tipos, como la función sigmoide, la función tangente hiperbólica o la función ReLU. Para entender este concepto nos vamos a centrar en el caso concreto de la red neuronal de la Figura 1.1.

Como se ve en la Figura 1.1, la entrada en nuestra función está representada por los dos círculos de la izquierda, que representan los valores de entrada  $x_1$  y  $x_2$ , siendo así la entrada  $x \in \mathbb{R}^2$ . Estos valores se multiplican por unos pesos ( $W^{[2]}$ ) y se suman a un sesgo  $b^{[2]}$ . La salida de esta neurona se calcula mediante una función de activación.



Figura 1.1: Esquema de una red neuronal con 4 capas.

Así, los valores que “llegan” a la segunda capa de nuestra red neuronal serán de la forma

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^2,$$

Siendo  $W^{[2]} \in \mathbb{R}^{2 \times 2}$  y el vector  $b^{[2]} \in \mathbb{R}^2$ . A partir de aquí, se repite el proceso para cada capa de la red neuronal, hasta llegar a la capa de salida, que nos dará el valor de salida de nuestra red neuronal. De forma visual, se pueden interpretar las líneas de la Figura 1.1 como los pesos por los que se va multiplicando.

En la tercera capa de la red neuronal, vemos que los valores que llegan de la capa 2, estos  $\sigma(W^{[2]}x + b^{[2]})$ , pertenecen a  $\mathbb{R}^2$ . De este modo, como tenemos 3 neuronas en la tercera capa, para obtener un valor perteneciente a  $\mathbb{R}^3$ , necesitamos una matriz  $W^{[3]} \in \mathbb{R}^{3 \times 2}$  y un vector  $b^{[3]} \in \mathbb{R}^3$ . Así, el valor de nuestra red neuronal en la tercera capa será

$$\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) \in \mathbb{R}^3.$$

Finalmente, la capa de salida recibirá de la tercera capa un vector perteneciente a  $\mathbb{R}^3$ , por lo que necesitaremos una matriz  $W^{[4]} \in \mathbb{R}^{3 \times 3}$  y un vector  $b^{[4]} \in \mathbb{R}^3$ . Así, el valor de salida de nuestra red neuronal, esa  $F$  de la que habíamos hablado al principio, será

$$(1.1) \quad F(x) = \sigma(W^{[4]}\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]}) \in \mathbb{R}^3.$$

En general, una red neuronal se puede representar como una composición de funciones, donde cada función es una capa de la red neuronal.

Nuestra intención con este tipo de funciones es ir variando los valores de las matrices  $W$ , también conocidos como pesos, y los vectores  $b$  también conocidos como sesgos, para que la salida de nuestra red neuronal se acerque lo máximo posible a la salida deseada.

Para entender esto, vamos a utilizar la red neuronal de la Figura 1.1 para resolver un problema concreto muy sencillo de clasificación. Supongamos que tenemos una serie de puntos en el plano, de tres tipos distintos, como los de la Figura 1.2, y queremos clasificarlos en tres grupos, los puntos de tipo azul, rojo y verde.

De este modo, nuestra red neuronal recibirá como entrada un punto del plano, y nos dirá a qué categoría pertenece, devolviendo  $(1, 0, 0)^T$  si es de la categoría azul,  $(0, 1, 0)^T$  si es de la categoría roja y  $(0, 0, 1)^T$  si es de la categoría amarilla.

Lo siguiente que queremos hacer será entrenar la red neuronal, es decir, ajustar los pesos y sesgos de la red neuronal para que la salida se acerque lo máximo posible a la salida deseada. Es decir, que cuando se introduzca un punto de un tipo concreto, la salida lo asigne a la categoría adecuada.

Designamos a  $y(x)$  como la salida deseada de nuestra red neuronal, y a  $F(x)$  como la salida real. Así, el error vendrá dado en función de los pesos y sesgos de la siguiente forma

$$\mathcal{L}(W^{[2]}, W^{[3]}, W^{[4]}, b^{[2]}, b^{[3]}, b^{[4]}) = \frac{1}{2} \sum_{x \in X} \|y(x) - F(x)\|^2,$$





Figura 1.2: Puntos en el plano que marcan las tres categorías

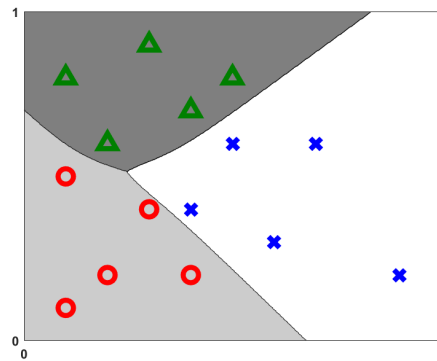


Figura 1.3: Puntos en el plano que marcan las tres categorías

donde  $X$  es el conjunto de puntos que tenemos para entrenar la red neuronal, en nuestro caso, serán 15. Esta función es conocida como función de coste.

Así, lo que queremos hacer es minimizar esta función, es decir, encontrar los pesos que minimicen la función  $\mathcal{L}$ . Para ello, se utilizan algoritmos de optimización, como el descenso del gradiente. Este proceso es conocido como el entrenamiento de la red neuronal. Si se logra con éxito, la red neuronal será capaz de clasificar correctamente los puntos en el plano. En este caso concreto, al entrenar la red neuronal, se obtiene la clasificación de la Figura 1.3, en la que simplemente hemos aplicado nuestra función para cada punto del plano y lo hemos sombreado de acuerdo a la clasificación que se le ha dado. El código que se ha utilizado para entrenar la red neuronal y obtener la clasificación de la Figura 1.3 es el que se encuentra en el Anexo A

### 1.1.1. Comentarios sobre la función sigmoide

Como hemos visto con el ejemplo anterior, las redes neuronales se pueden utilizar para generar funciones. A su vez, estas funciones se utilizan para resolver problemas de forma aproximada. En el ejemplo de la Figura 1.3, la función red neuronal clasi-

fica cualquier parte del plano en una de las tres categorías sombreadas a partir del entrenamiento. Dicho entrenamiento fija los valores de los pesos y sesgos, y por tanto define la función red neuronal (en el ejemplo define la función (1.1)).

Como las redes neuronales se utilizan para aproximar problemas de tipos muy distintos, una de las características deseables es que sean capaces de aproximar el conjunto de funciones "mayor posible".

Para poder lograr este objetivo, se necesitan las funciones de activación. Como ejercicio, supongamos que en nuestra ecuación (1.1) en lugar de utilizar la función sigmoide, no utilizamos ninguna función de activación, es decir, que nuestra red neuronal es simplemente una composición de funciones lineales. En este caso, nuestra red tendría la siguiente forma

$$F(x) = W^{[4]}(W^{[3]}(W^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]} \in \mathbb{R}^3.$$

Claramente, una función lineal definida de forma global puede estar muy lejos de aproximar bien funciones generales. De este modo, parece por tanto razonable utilizar funciones de activación no lineales. La función sigmoide es una de las más utilizadas, pero existen otras como la función tangente hiperbólica o la función ReLU. La función sigmoide se define como:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Esta función tiene la ventaja de que su derivada es fácil de calcular, y es precisamente esta derivada la que se utiliza en el algoritmo de entrenamiento de la red neuronal.

Un resultado relevante para justificar el uso de funciones de activación no lineales es el siguiente propuesto por Pinkus [9, Theorem 3.1]. Este resultado es para redes neuronales de una sola capa.

**Teorema 1.1** (Pinkus). *Sea  $\sigma \in C(\mathbb{R})$  y sea  $\mathcal{M}(\sigma) = \text{span}\{\sigma(w \cdot x + b) : b \in \mathbb{R}, w \in \mathbb{R}^n\}$ , se cumple que:*

*Para cualquier  $f \in C(\mathbb{R}^n)$ , cualquier conjunto compacto  $K \in \mathbb{R}^n$  y cualquier  $\epsilon > 0$ , existe una función  $g \in \mathcal{M}(\sigma)$  tal que  $\max_{x \in K} |f(x) - g(x)| < \epsilon$  si y solo si  $\sigma$  no es una función polinómica.*

De forma más intuitiva, cualquier función  $f \in C(\mathbb{R}^n)$ , se puede aproximar tan bien como se quiera con una red neuronal de una sola capa si y solo si la función de activación no es polinómica.

Este resultado nos aporta una intuición de por qué las funciones de activación no lineales son necesarias para aproximar funciones de forma general pues, de no ser funciones no lineales, habría muchas funciones que no podríamos aproximar.

## 1.2. Conceptos preliminares sobre las ecuaciones en derivadas parciales

En este trabajo nos vamos a centrar en las ecuaciones en derivadas parciales elípticas, definidas con la siguiente forma general.

**Definición 1.2.** Dado un conjunto  $\Omega$ , acotado y abierto en  $\mathbb{R}^n$ , decimos que una ecuación en derivadas parciales es elíptica si es de la siguiente forma:

$$(1.2) \quad - \sum_{i,j=1}^n \frac{\partial}{\partial x_j} \left( a_{ij}(x) \frac{\partial u}{\partial x_i} \right) + \sum_{i=1}^n b_i(x) \frac{\partial u}{\partial x_i} + c(x)u = f(x), \quad x \in \Omega.$$

Donde los coeficientes  $a_{ij}(x)$ ,  $b_i(x)$ ,  $c(x)$  y  $f$  son funciones que satisfacen las siguientes condiciones

$$(1.3) \quad a_{ij} \in C^1(\overline{\Omega}), \quad i, j = 1, \dots, n$$

$$(1.4) \quad b_i, c \in C(\overline{\Omega}), \quad i = 1, \dots, n$$

$$(1.5) \quad c \in C(\overline{\Omega}),$$

$$(1.6) \quad f \in C(\overline{\Omega})$$

y además cumple la condición de elipticidad uniforme, es decir

$$(1.7) \quad \sum_{i,j=1}^n a_{ij}(x) \xi_i \xi_j \geq \tilde{c} \sum_{i=1}^n \xi_i^2, \quad \forall \xi = (\xi_1, \dots, \xi_n) \in \mathbb{R}^n, \quad x \in \overline{\Omega},$$

donde  $\tilde{c}$  es una constante positiva independiente de  $x$  y  $\xi$ .

Concretamente, a lo largo del trabajo, nos centraremos en problemas de condición de frontera de Dirichlet, es decir, problemas en los que se cumple que

$$(1.8) \quad u(x) = g(x), \quad \forall x \in \partial\Omega.$$

En la formulación que hemos dado de las ecuaciones en derivadas parciales elípticas, nos estábamos refiriendo a su formulación en forma fuerte. No obstante, en muchos casos, no es posible encontrar una solución en forma fuerte. En estos casos, se recurre a la formulación débil de la ecuación, que permite encontrar una solución en un espacio de funciones más amplio. Para entender la formulación débil de una ecuación en derivadas parciales, es necesario introducir el concepto de derivada débil.

**Definición 1.3.** Supongamos que  $u$  es localmente integrable en  $\Omega$ . Supongamos que también existe una función  $w_\alpha$ , localmente integrable en  $\Omega$ , tal que

$$\int_{\Omega} w_\alpha(x) v(x) dx = (-1)^{|\alpha|} \int_{\Omega} u(x) D^\alpha v(x) dx, \quad \forall v \in C_0^\infty(\Omega),$$

entonces decimos que  $w_\alpha$  es la derivada débil de  $u$  de orden  $|\alpha| = \alpha_1 + \dots + \alpha_n$  y escribimos  $w_\alpha = D^\alpha u$ . Cuando existe la derivada débil, es única. En lo sucesivo utilizaremos la misma notación para las derivadas débiles y fuertes.

También es necesario definir el espacio de Sobolev  $H^1(\Omega)$ .

**Definición 1.4.** Sea  $\Omega$  un conjunto abierto de  $\mathbb{R}^n$ . Definimos el espacio de Sobolev  $H^1(\Omega)$  como el conjunto de funciones en el siguiente conjunto

$$(1.9) \quad H^1(\Omega) = \{u \in L^2(\Omega) : \frac{\partial u}{\partial x_i} \in L^2(\Omega), i = 1, \dots, n\}.$$

En este espacio, se define la siguiente norma

$$(1.10) \quad \|u\|_{H^1(\Omega)} = \left( \|u\|_{L^2(\Omega)}^2 + \sum_{i=1}^n \left\| \frac{\partial u}{\partial x_i} \right\|_{L^2(\Omega)}^2 \right)^{1/2}.$$

En ambas definiciones las derivadas parciales se entienden en sentido débil.

Definimos además el espacio  $H_0^1(\Omega)$ . Este espacio es el cierre de las funciones de  $C_0^\infty(\Omega)$  en la norma de  $H^1(\Omega)$ . Es decir,  $H_0^1(\Omega)$  es el conjunto de funciones  $u \in H^1(\Omega)$  que se obtienen como límite en  $H^1(\Omega)$  de una serie de funciones  $\{u_m\}_{m=1}^\infty$  todas ellas en  $C_0^\infty(\Omega)$ . Si  $\partial\Omega$  es suficientemente regular,  $H_0^1(\Omega)$  es el siguiente conjunto:

$$(1.11) \quad H_0^1(\Omega) = \{u \in H^1(\Omega) : u = 0 \text{ en } \partial\Omega\}.$$

Con esto, podemos definir la solución débil de una ecuación en derivadas parciales elíptica de la siguiente forma.

**Definición 1.5.** Dadas las funciones  $a_{ij} \in L^\infty(\Omega)$ ,  $i, j = 1, \dots, n$ ,  $b_i \in L^\infty(\Omega)$ ,  $i = 1, \dots, n$ ,  $c \in L^\infty(\Omega)$ , y la función  $f \in L^2(\Omega)$ . Decimos que la función  $u \in H_0^1(\Omega)$  es una solución débil de la ecuación (1.2) con condición de contorno Dirichlet homogénea, es decir,  $g(x) = 0$  si se cumple que,

$$(1.12) \quad \sum_{i,j=1}^n \int_{\Omega} a_{ij}(x) \frac{\partial u}{\partial x_i} \frac{\partial \varphi}{\partial x_j} dx + \sum_{i=1}^n \int_{\Omega} b_i(x) \frac{\partial u}{\partial x_i} \varphi dx + \int_{\Omega} c(x) u \varphi dx = \int_{\Omega} f(x) \varphi(x) dx, \quad \forall \varphi \in H_0^1(\Omega),$$

donde todas las derivadas parciales en la ecuación (1.12) se entienden en sentido débil.

Si  $u$  es una solución clásica de (1.2), entonces también es una solución débil de (1.12). Sin embargo, lo contrario no es cierto: una solución débil puede no ser lo suficientemente regular para ser una solución clásica. En particular, se demostrará

que la ecuación (1.2) tiene una solución débil única  $u \in H_0^1(\Omega)$ . Para simplificar la notación, se introduce la forma bilineal:

$$(1.13) \quad \begin{aligned} a(u, \varphi) = & \sum_{i,j=1}^n \int_{\Omega} a_{ij}(x) \frac{\partial u}{\partial x_i} \frac{\partial \varphi}{\partial x_j} dx + \sum_{i=1}^n \int_{\Omega} b_i(x) \frac{\partial u}{\partial x_i} \varphi dx \\ & + \int_{\Omega} c(x) u \varphi dx, \end{aligned}$$

y el funcional lineal,

$$(1.14) \quad l(\varphi) = \int_{\Omega} f(x) \varphi(x) dx.$$

De este modo, nuestro problema elíptico en forma débil se puede expresar de la siguiente manera: Encontrar  $u \in H_0^1(\Omega)$  tal que

$$(1.15) \quad a(u, \varphi) = l(\varphi) \quad \forall \varphi \in H_0^1(\Omega).$$

Para demostrar la existencia y unicidad de solución débil, se aplica el teorema de Lax-Milgram.

**Teorema 1.6** (Lax-Milgram). *Sea  $V$  un espacio de Hilbert con una norma asociada  $\|\cdot\|_V$ , y sea  $a : V \times V \rightarrow \mathbb{R}$  una forma bilineal y  $l : V \rightarrow \mathbb{R}$  un funcional lineal. Supongamos que se cumplen las siguientes condiciones*

- (a)  $\exists c_0 > 0 \quad \forall v \in V \quad a(v, v) \geq c_0 \|v\|^2,$
- (b)  $\exists c_1 > 0 \quad \forall v, w \in V \quad |a(w, v)| \leq c_1 \|w\| \|v\|.$

Sea  $l(\cdot)$  un funcional lineal en  $V$  tal que:

- (c)  $\exists c_2 > 0 \quad \forall v \in V \quad |l(v)| \leq c_2 \|v\|.$

Entonces, existe una única solución débil  $u \in V$  tal que

$$a(u, v) = l(v) \quad \forall v \in V.$$

Con este teorema se puede demostrar que una ecuación elíptica con condiciones de contorno Dirichlet homogéneas tiene solución única en  $H_0^1(\Omega)$ . Para ello se puede aplicar el teorema de Lax-Milgram en el espacio  $V = H_0^1(\Omega)$  con la norma  $\|\cdot\|_{H^1(\Omega)}$  y las formas bilineal y lineal  $a$  y  $l$  definidas anteriormente. Para que se cumplan las condiciones (a, b, c) del teorema, además de la condición de elipticidad uniforme es necesario requerir que

$$(1.16) \quad c(x) - \frac{1}{2} \sum_{i=1}^n \frac{\partial b_i}{\partial x_i} \geq 0, \quad x \in \overline{\Omega}.$$

De esta manera, se demuestran las condiciones (a), (b) y (c) del teorema de Lax-Milgram como sigue.

**Demostración 1.7.** Empezamos por la condición (c).

(c) La aplicación  $v \mapsto l(v)$  es lineal: de hecho, para cualesquiera  $\alpha, \beta \in \mathbb{R}$ ,

$$\begin{aligned} l(\alpha v_1 + \beta v_2) &= \int_{\Omega} f(x)(\alpha v_1(x) + \beta v_2(x)) dx \\ &= \alpha \int_{\Omega} f(x)v_1(x) dx + \beta \int_{\Omega} f(x)v_2(x) dx \\ &= \alpha l(v_1) + \beta l(v_2), \quad v_1, v_2 \in H_0^1(\Omega). \end{aligned}$$

Por lo tanto,  $l(\cdot)$  es una funcional lineal en  $H_0^1(\Omega)$ . Además, por la desigualdad de Cauchy-Schwarz,

$$\begin{aligned} |l(v)| &= \left| \int_{\Omega} f(x)v(x) dx \right| \leq \left( \int_{\Omega} |f(x)|^2 dx \right)^{1/2} \left( \int_{\Omega} |v(x)|^2 dx \right)^{1/2} \\ &= \|f\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} \leq \|f\|_{L^2(\Omega)} \|v\|_{H^1(\Omega)}, \end{aligned}$$

para todo  $v \in H_0^1(\Omega)$ , donde hemos usado la desigualdad obvia  $\|v\|_{L^2(\Omega)} \leq \|v\|_{H^1(\Omega)}$ . Tomando  $c_2 = \|f\|_{L^2(\Omega)}$ , obtenemos la cota requerida.

(b) A continuación, verificamos (b). Para cualquier  $w \in H_0^1(\Omega)$ , la aplicación  $v \mapsto a(v, w)$  es lineal. De manera similar, para cualquier  $v \in H_0^1(\Omega)$ , la aplicación  $w \mapsto a(v, w)$  es lineal. Por lo tanto,  $a(\cdot, \cdot)$  es una funcional bilineal en  $H_0^1(\Omega) \times H_0^1(\Omega)$ . Aplicando la desigualdad de Cauchy-Schwarz, deducimos que

$$\begin{aligned} |a(w, v)| &\leq \sum_{i,j=1}^n \max_{x \in \Omega} |a_{ij}(x)| \left| \int_{\Omega} \frac{\partial w}{\partial x_i} \frac{\partial v}{\partial x_j} dx \right| \\ &\quad + \sum_{i=1}^n \max_{x \in \Omega} |b_i(x)| \left| \int_{\Omega} \frac{\partial w}{\partial x_i} v dx \right| \\ &\quad + \max_{x \in \Omega} |c(x)| \left| \int_{\Omega} w(x)v(x) dx \right| \\ &\leq \hat{c} \left\{ \left( \sum_{i,j=1}^n \int_{\Omega} \left| \frac{\partial w}{\partial x_i} \right|^2 dx \right)^{1/2} \left( \int_{\Omega} \left| \frac{\partial v}{\partial x_j} \right|^2 dx \right)^{1/2} \right. \\ &\quad + \sum_{i=1}^n \left( \int_{\Omega} \left| \frac{\partial w}{\partial x_i} \right|^2 dx \right)^{1/2} \left( \int_{\Omega} |v|^2 dx \right)^{1/2} \\ &\quad \left. + \left( \int_{\Omega} |w|^2 dx \right)^{1/2} \left( \int_{\Omega} |v|^2 dx \right)^{1/2} \right\} \\ &\leq \hat{c} \left\{ \left( \int_{\Omega} |w|^2 dx \right)^{1/2} + \sum_{i=1}^n \left( \int_{\Omega} \left| \frac{\partial w}{\partial x_i} \right|^2 dx \right)^{1/2} \right\} \\ &\quad \times \left\{ \left( \int_{\Omega} |v|^2 dx \right)^{1/2} + \sum_{j=1}^n \left( \int_{\Omega} \left| \frac{\partial v}{\partial x_j} \right|^2 dx \right)^{1/2} \right\}. \end{aligned}$$

donde

$$\hat{c} = \max \left\{ \max_{1 \leq i, j \leq n} \max_{x \in \Omega} |a_{ij}(x)|, \max_{1 \leq i \leq n} \max_{x \in \Omega} |b_i(x)|, \max_{x \in \Omega} |c(x)| \right\}.$$

Si acotamos aún más el lado derecho de la última desigualdad, deducimos que

$$|a(w, v)| \leq (n+1)\hat{c} \left\{ \left( \int_{\Omega} |w|^2 dx + \sum_{i=1}^n \int_{\Omega} \left| \frac{\partial w}{\partial x_i} \right|^2 dx \right)^{1/2} \right. \\ \left. \times \left( \int_{\Omega} |v|^2 dx + \sum_{j=1}^n \int_{\Omega} \left| \frac{\partial v}{\partial x_j} \right|^2 dx \right)^{1/2} \right\},$$

por lo que, tomando  $c_1 = (n+1)\hat{c}$ , obtenemos la desigualdad de (b).

- (a) Por último, demostramos la condición (a). Aquí utilizaremos la condición extra requerida en (1.16) y la condición de elipticidad uniforme (1.7). Usando (1.7) y la desigualdad de Cauchy-Schwarz,

$$a(v, v) \geq \tilde{c} \sum_{i=1}^n \int_{\Omega} \left| \frac{\partial v}{\partial x_i} \right|^2 dx + \sum_{i=1}^n \int_{\Omega} b_i(x) \frac{1}{2} \frac{\partial}{\partial x_i} (v^2) dx + \int_{\Omega} c(x) |v|^2 dx,$$

donde escribimos  $\frac{\partial v}{\partial x_i} \cdot v$  como  $\frac{1}{2} \frac{\partial}{\partial x_i} (v^2)$ . Integrando por partes el segundo término en el lado derecho, obtenemos

$$a(v, v) \geq \tilde{c} \sum_{i=1}^n \int_{\Omega} \left| \frac{\partial v}{\partial x_i} \right|^2 dx + \int_{\Omega} \left( c(x) - \frac{1}{2} \sum_{i=1}^n \frac{\partial b_i}{\partial x_i} \right) |v|^2 dx.$$

Asumiendo que se cumple (1.16) llegamos a la desigualdad

$$(1.17) \quad a(v, v) \geq \tilde{c} \sum_{i=1}^n \int_{\Omega} \left| \frac{\partial v}{\partial x_i} \right|^2 dx.$$

Mediante la desigualdad de Poincaré-Friedrichs, el lado derecho puede acotarse aún más para obtener

$$(1.18) \quad a(v, v) \geq \frac{\tilde{c}}{c_{\star}} \int_{\Omega} |v|^2 dx.$$

Juntando (1.17) y (1.18), obtenemos

$$(1.19) \quad a(v, v) \geq c_0 \left( \int_{\Omega} |v|^2 dx + \sum_{i=1}^n \int_{\Omega} \left| \frac{\partial v}{\partial x_i} \right|^2 dx \right),$$

donde  $c_0 = \tilde{c}/(1 + c_{\star})$ , y por tanto queda demostrado (a).

Se comprueban así las condiciones (a), (b) y (c) del teorema de Lax-Milgram y concluimos que existe una única solución débil en  $H_0^1(\Omega)$  para la ecuación (1.15).

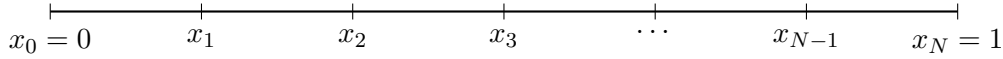


Figura 1.4: División de  $\Omega = (0, 1)$  en  $N$  elementos finitos

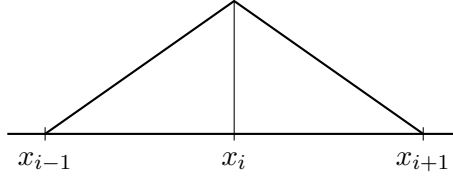


Figura 1.5: Polinomio  $\phi_i$

### 1.3. Métodos numéricos tradicionales: el método de los elementos finitos

El método de los elementos finitos (FEM por su siglas en inglés) es una técnica numérica para resolver ecuaciones en derivadas parciales. Este método es ampliamente utilizado en ingeniería y ciencias aplicadas porque permite aproximar soluciones en dominios complejos.

Para un problema elíptico con condiciones de contorno Dirichlet homogéneas, se parte de la ecuación en su formulación débil. Como hemos visto en el apartado anterior, este problema se puede formular de forma simplificada como encontrar  $u \in H_0^1(\Omega)$  tal que

$$a(u, \varphi) = l(\varphi) \quad \forall \varphi \in H_0^1(\Omega).$$

Por lo visto en la Sección 1.2, esta ecuación tiene solución única en  $H_0^1(\Omega)$  si se cumple la condición (1.16). De este modo, para que la formulación del problema sea válida, asumiremos que se cumple esta condición.

Lo siguiente que se hace es dividir el dominio  $\Omega$  en un conjunto de subdominios más pequeños, llamados elementos finitos. Por ejemplo, si estamos trabajando con un problema en un dominio unidimensional, con  $\Omega = (0, 1)$ , dividimos  $\bar{\Omega} = [0, 1]$  en  $N$  subintervalos  $[x_i, x_{i+1}]$ ,  $i = 1, \dots, N - 1$ , con  $x_i = ih$ , obteniendo la división de la Figura 1.4.

A esta subdivisión, se le asocia una base de polinomios a trozos. Reemplazando así el subespacio de funciones que habíamos llamado  $H_0^1(\Omega)$  por un subespacio de dimensión finita  $V_h$  formado por polinomios a trozos de grado fijo. Siguiendo nuestro ejemplo de antes, la base de polinomios que vamos a utilizar será la de los polinomios como los vistos en la Figura 1.5, definidos de la siguiente forma.

$$\phi_i(x) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}}, & \text{si } x \in [x_{i-1}, x_i], \\ \frac{x_{i+1} - x}{x_{i+1} - x_i}, & \text{si } x \in [x_i, x_{i+1}], \\ 0, & \text{en otro caso.} \end{cases}$$



En este ejemplo, nuestro espacio de funciones será  $V_h = \text{span}\{\phi_1, \dots, \phi_{N-1}\}$ , donde  $N$  es el número de elementos finitos en los que hemos dividido el dominio. Con esta definición es claro que todos los  $\phi_i \in H_0^1(\Omega)$  pues son funciones derivables en sentido débil que valen 0 en la frontera de  $\Omega = (0, 1)$ . De este modo, al buscar soluciones en este nuevo espacio de funciones, nuestra ecuación diferencial elíptica se convertiría en encontrar  $u_h \in V_h$  tal que

$$a(u_h, v_h) = l(v_h) \quad \forall v_h \in V_h.$$

Nótese que las condiciones de contorno van implícitas en la definición de  $V_h$  pues todos los  $\phi_i \in H_0^1(\Omega)$ .

Con todo, de forma intuitiva, lo que se hace es que en vez de buscar una solución en un espacio de funciones de dimensión infinita, como es  $H_0^1(\Omega)$ , se busca una solución en un espacio de funciones de dimensión finita, como es  $V_h$ . Para poder hacer eso, que nos simplifica mucho el problema, lo que hemos hecho es que nuestro dominio lo hemos dividido en subdominios más pequeños a partir de los cuales hemos construido nuestro nuevo espacio de funciones.

Ahora, encontrar una solución se convierte en encontrar los coeficientes en la base de polinomios a trozos, es decir, los  $U_i$  en la siguiente ecuación

$$(1.20) \quad u_h = \sum_{i=1}^{N-1} U_i \phi_i.$$

Por tanto, resolver la ecuación diferencial se convierte en encontrar  $U = (U_1, \dots, U_{N-1}) \in \mathbb{R}^{N-1}$  que cumplen:

$$(1.21) \quad \sum_{i=1}^{N-1} a(\phi_i, \phi_j) U_i = l(\phi_j) \quad \forall j = 1, \dots, N-1.$$

Esto se traduce en un sistema lineal de la forma  $AU = b$ , donde  $A$  es la matriz de rigidez (con entradas  $A_{ij} = a(\phi_i, \phi_j)$ ), y  $b$  es el vector de términos fuente (con entradas  $b_j = l(\phi_j)$ ). A este nuevo sistema lineal se le pueden aplicar métodos numéricos tradicionales para resolverlo, como la factorización LU, encontrando así los coeficientes  $U$  que nos dan  $u_h$ , la aproximación de  $u$ .

Este proceso se puede generalizar a problemas en más dimensiones, donde se dividen los dominios en elementos finitos de mayor dimensión, y se construye una base de polinomios a trozos en cada uno de estos elementos. Por ejemplo, en dos dimensiones, un elemento finito sería un triángulo, y la base de polinomios a trozos sería una base de funciones que valen 1 en uno de los nodos de la partición, 0 en el resto de nodos y además son lineales en cada uno de los triángulos.

**Ejemplo 1.8.** Veamos ahora un ejemplo concreto de la aplicación del FEM a través de la ecuación de Poisson en dos dimensiones. El objetivo es encontrar un

campo escalar  $u$  sobre un dominio  $\Omega \subset \mathbb{R}^2$  que satisfaga:

$$(1.22) \quad \begin{aligned} -\Delta u &= f & \text{en } \Omega, \\ u &= 0 & \text{en } \partial\Omega, \end{aligned}$$

donde  $f$  es un término fuente y la condición de contorno es de Dirichlet homogénea. Para encontrar una solución, construimos la forma débil de la ecuación. Para ello, tomamos  $v \in H_0^1(\Omega)$  y multiplicamos ambos lados por  $v$  e integramos sobre  $\Omega$ .

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} f v \, dx.$$

Utilizando la identidad  $\text{div}(v\nabla u) = (\nabla u)(\nabla v) + v\Delta u$  obtenemos:

$$\int_{\Omega} \nabla u \nabla v - \int_{\Omega} \text{div}(v\nabla u) = \int_{\Omega} f v$$

Por el teorema de Green, podemos escribir (si  $\partial\Omega$  es suficientemente regular):

$$\int_{\Omega} \nabla u \nabla v - \int_{\partial\Omega} (v\nabla u) \cdot \mathbf{n} \, dS = \int_{\Omega} f v$$

Finalmente, dado que  $v = 0$  en  $\partial\Omega$ , concluimos:

$$\int_{\Omega} \nabla u \nabla v = \int_{\Omega} f v$$

La forma fuerte del problema se convierte en:

$$(1.23) \quad a(u, v) = l(v) \quad \forall v \in H_0^1(\Omega),$$

donde:

- $a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx$
- $l(v) = \int_{\Omega} f v \, dx$

Para aplicar el FEM, nos interesa encontrar una solución aproximada en un espacio de funciones de dimensión finita. Para ello, dividimos el dominio  $\Omega$  en un conjunto de elementos finitos y construimos una base de polinomios a trozos en cada uno de estos elementos.

Antes de seguir avanzando, para este ejemplo específico, tomaremos un dominio cuadrado  $\Omega = (0, 1)^2$  y  $f(x, y) = 1$ . Bajo estas condiciones sabemos que existe una solución única en  $H_0^1(\Omega)$  para el problema (1.23).

A continuación, usamos el paquete de Python Scipy para construir la malla de elementos finitos y la base de polinomios necesarias para resolver el problema (código comentado en el Anexo B). El resultado obtenido se muestra en la Figura

1.6, donde la malla de elementos finitos está compuesta por 50 nodos en cada lado.

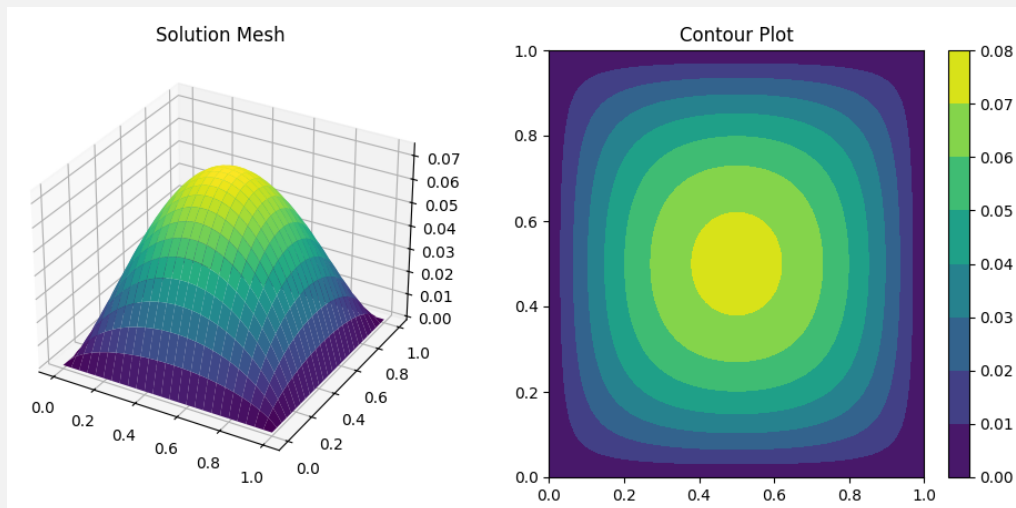


Figura 1.6: Solución de la ecuación de Poisson



## CAPÍTULO 2

# Aproximación de EDPs mediante redes neuronales

---

### 2.1. PINNs: Physics-Informed Neural Networks

#### 2.1.1. Introducción

Las PINNs, o Physics-Informed Neural Networks, son una técnica que combina la resolución de ecuaciones en derivadas parciales con redes neuronales. La idea es que, en vez de resolver la ecuación diferencial directamente o con métodos numéricos tradicionales, se entrena una red neuronal para que aproxime la solución de la ecuación.

Como habíamos comentado en la Sección 1.1, las redes neuronales son capaces de aproximar cualquier función, por lo que, en teoría, pueden aproximar cualquier solución de una ecuación diferencial. De este modo, las PINNs van a aprovechar esto, definiendo la aproximación de nuestra solución  $u$ , como la salida de la función red neuronal  $\hat{u}(x; \theta)$ , donde  $\theta$  hace referencia a los pesos y sesgos vistos en la Sección 1.1. Así, la función en la Figura 2.1 sería la aproximación de la solución de la ecuación diferencial.



Figura 2.1: Esquema de una PINN

### 2.1.2. Formulación de las PINNs

Consideramos una EDP de la forma vista en la Sección 1.2, es decir, una EDP elíptica sujeta a condiciones de contorno Dirichlet. De modo que la ecuación a resolver es la siguiente:

$$(2.1) \quad \begin{cases} -\sum_{i,j=1}^n \frac{\partial}{\partial x_j} \left( a_{ij}(\mathbf{x}) \frac{\partial u}{\partial x_i} \right) + \sum_{i=1}^n b_i(\mathbf{x}) \frac{\partial u}{\partial x_i} + c(\mathbf{x})u = f(\mathbf{x}), & \mathbf{x} \in \Omega, \\ u(\mathbf{x}) = g(\mathbf{x}), & \mathbf{x} \in \partial\Omega. \end{cases}$$

donde  $\Omega \subset \mathbb{R}^n$  es un dominio abierto y acotado y las funciones  $a_{ij}(\mathbf{x})$ ,  $b_i(\mathbf{x})$ ,  $c(\mathbf{x})$ ,  $f(\mathbf{x})$  y  $g(\mathbf{x})$  son conocidas. La incógnita es la función  $u : \Omega \rightarrow \mathbb{R}$  que satisface la ecuación diferencial y las condiciones de contorno. Para poder construir y ajustar los pesos de una red neuronal que aproxime  $u$  es esencial tener una función de coste, es decir, una función que nos indique la calidad de nuestra aproximación.

Con las PINNs nosotros queremos encontrar una función  $\hat{u}$  que minimice el residuo en  $\Omega$  y que cumpla las condiciones de contorno en  $\partial\Omega$ . Para ello, lo que queremos es la  $\hat{u}$  que haga que las siguientes expresiones sean mínimas:

$$\begin{aligned} & \int_{\Omega} \left( -\sum_{i,j=1}^n \frac{\partial}{\partial x_j} \left( a_{ij}(\mathbf{x}) \frac{\partial \hat{u}}{\partial x_i} \right) + \sum_{i=1}^n b_i(\mathbf{x}) \frac{\partial \hat{u}}{\partial x_i} + c(\mathbf{x})\hat{u} - f(\mathbf{x}) \right)^2, \\ & \int_{\partial\Omega} (\hat{u}(\mathbf{x}) - g(\mathbf{x}))^2. \end{aligned}$$

Para poder calcular estas integrales, se utiliza una regla de cuadratura que las aproxima mediante sumas ponderadas de los valores de las funciones en un conjunto discreto de puntos. En este contexto, los puntos de evaluación se denominan puntos de colocación o collocation points. Para la primera expresión, los puntos de colocación que se usarán los denominaremos  $\mathcal{T}_f$ , mientras que para la segunda los denominaremos  $\mathcal{T}_b$ .

Por tanto, en el caso de las PINNs, la función de coste se compone de la suma ponderada de dos términos. El primero garantiza que la red neuronal aproxime la solución de la ecuación diferencial en un conjunto discreto de puntos. El segundo asegura que dicha aproximación cumpla las condiciones de contorno de Dirichlet en otro conjunto finito de puntos.

$$(2.2) \quad \mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b),$$

donde  $w_f$  y  $w_b$  son pesos que se ajustan para dar más importancia a un término u otro y

$$\begin{aligned}
 \mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) &= \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} \left| - \sum_{i,j=1}^n \frac{\partial}{\partial x_j} \left( a_{ij}(\mathbf{x}) \frac{\partial \hat{u}}{\partial x_i} \right) \right. \\
 (2.3) \quad &\quad \left. + \sum_{i=1}^n b_i(\mathbf{x}) \frac{\partial \hat{u}}{\partial x_i} + c(\mathbf{x}) \hat{u} - f(\mathbf{x}) \right|^2, \\
 \mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) &= \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_b} |\hat{u}(\mathbf{x}) - g(\mathbf{x})|^2.
 \end{aligned}$$

Como hemos indicado, aquí  $\mathcal{T}_f$  y  $\mathcal{T}_b$  representan conjuntos de puntos en el dominio  $\Omega$  y en la frontera  $\partial\Omega$ , respectivamente. De este modo, las funciones,  $\mathcal{L}_f$  y  $\mathcal{L}_b$ , representan la aproximación al residuo fuerte, y la condición de contorno, por una regla de cuadratura.

Existe mucha literatura sobre como tomar los puntos de colocación pues, al igual que en FEM la malla impacta el resultado, en las PINNs, el conjunto  $\mathcal{T}$  determina como de bien nuestra red neuronal se ajusta a la solución [8], [1], [7], [10], [4].

Es importante destacar que con esta formulación, se impone que el residuo en forma fuerte de la ecuación diferencial sea cero en los puntos de colocación, lo cual, veremos más adelante que puede ser una fuente de errores.

Dada la función de coste y la distribución de los puntos de colocación, el siguiente paso es el entrenamiento. Esto se reduce a resolver un problema de optimización: encontrar los parámetros  $\boldsymbol{\theta}$  que minimizan la función de coste  $\mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$ . Para ello, es habitual emplear métodos de optimización basados en gradientes, como Adam o L-BFGS.

Una ventaja fundamental del uso de redes neuronales en este contexto es la diferenciación automática, una técnica numérica muy potente que facilita la obtención de las derivadas de  $\hat{u}$  con respecto a sus entradas de forma exacta y eficiente. A continuación, vamos a ver dos ejemplos para ver como se aplican las PINNs a problemas concretos.

**Ejemplo 2.1.** Consideramos la ecuación elíptica unidimensional:

$$-u_{xx} = \pi^2 \sin(\pi x), \quad x \in [-1, 1],$$

con las condiciones de contorno de Dirichlet

$$u(-1) = 0, \quad u(1) = 0.$$

En esta ecuación la solución exacta es conocida  $u(x) = \sin(\pi x)$ . Para resolver este problema con una PINN, nos asistimos de la librería de python DeepXDE [5], la cual tiene implementada la funcionalidad necesaria para entrenar una red neuronal que aproxime EDPs. Así, comenzamos definiendo el intervalo en el que se encuentra el dominio y los puntos de colocación

```
1 geom = dde.geometry.Interval(-1, 1)
```

definimos la EDP

```
1 def pde(x, y):
2     dy_xx = dde.grad.hessian(y, x)
3     return -dy_xx - np.pi ** 2 * tf.sin(np.pi * x)
```

indicamos las condiciones de contorno de Dirichlet

```
1 bc = dde.icbc.DirichletBC(geom, func, boundary)
```

y, con todo, se define el problema de EDPs

```
1 data = dde.data.PDE(geom, pde, bc, 16, 2, solution=func,
    num_test=100)
```

Gracias a la librería DeepXDE, podemos entrenar la red neuronal con el siguiente código

```
1 Losshistory, train_state = model.train(
2     iterations=10000, callbacks=[checkpointer, movie]
3 )
```

Por detrás, esta se encargará de definir la función de coste asociada, que en este caso será

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b),$$

donde

$$\mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} |\hat{u}_{xx} + \pi^2 \sin(\pi x)|^2,$$

$$\mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_b} |\hat{u}(\mathbf{x})|^2,$$

por los argumentos con los que se han inicializado las funciones,  $\mathcal{T}_f$  serán 16 puntos aleatorios en el intervalo  $[-1, 1]$  y  $\mathcal{T}_b = \{0, 1\}$ . Con todo, el resultado de ejecutar este código será el de la Figura 2.2.



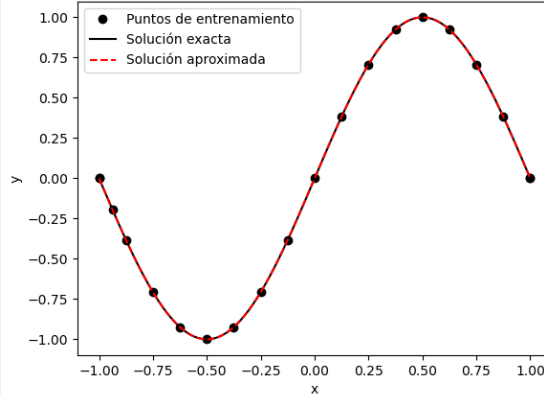


Figura 2.2: Solución de la ecuación de Poisson

Aunque en este ejemplo sencillo las PINNs funcionan muy bien, estas pueden presentar errores elevados incluso en sistemas relativamente simples. A continuación, vemos uno de los ejemplos que presentan en Luo et al. en [6] en los que las PINNs fallan.

**Ejemplo 2.2.** Consideramos la siguiente ecuación en un dominio unidimensional:

$$(2.4) \quad \begin{cases} Lu = -D_x(AD_x u) = f & \text{en } \Omega = (-1, 1), \\ u = 0 & \text{en } \partial\Omega = \{-1, 1\}, \end{cases}$$

donde la función coeficiente  $A$  y  $f$  son ambas funciones continuas a trozos y se expresan como

$$(2.5) \quad A(x) = \begin{cases} \frac{1}{2}, & x \in (-1, 0), \\ 1, & x \in [0, 1), \end{cases} \quad f(x) = \begin{cases} 0, & x \in (-1, 0), \\ -2, & x \in [0, 1). \end{cases}$$

Claramente, no existe una solución fuerte. Sin embargo la solución débil  $u \in H^1(-1, 1)$  para esta ecuación es

$$(2.6) \quad u(x) = \begin{cases} -\frac{2}{3}x - \frac{2}{3}, & x \in (-1, 0), \\ x^2 - \frac{1}{3}x - \frac{2}{3}, & x \in [0, 1). \end{cases}$$

En la serie de experimentos numéricos, utilizamos una red neuronal con la siguiente configuración 1-256-256-256-1. Así, la función de coste vendrá dada por la siguiente función.

$$\mathcal{L}(\theta; \mathcal{T}) = w_f \mathcal{L}_f(\theta; \mathcal{T}_f) + w_b \mathcal{L}_b(\theta; \mathcal{T}_b),$$

donde

$$\mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} |D_x(A(x)D_x w(x)) + f(x)|^2,$$

$$\mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_b} |\hat{u}(\mathbf{x})|^2,$$

En este caso,  $\mathcal{T}_b = \{-1, 1\}$  y  $\mathcal{T}_f$  es un conjunto de 1000 puntos muestreados uniformemente en el intervalo  $(-1, 1)$ . Es decir,  $|\mathcal{T}_f| = 1000$  y  $|\mathcal{T}_b| = 2$ .

Bajo esta configuración, obtenemos la función de red  $\hat{u}(\mathbf{x}; \theta)$  numéricamente a través del método PINN y la comparamos con la solución débil en la Figura 2.3.

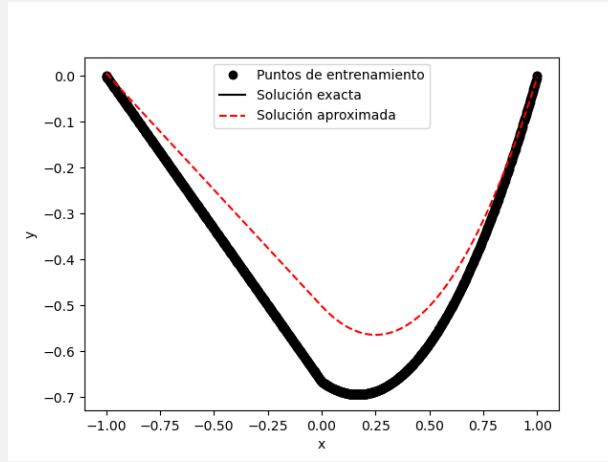


Figura 2.3: Solución de la ecuación

Evidentemente, el método no logra obtener la solución exacta  $u$ . La diferencia entre  $u$  y  $\hat{u}(\mathbf{x}; \theta)$  es del mismo orden de magnitud que  $u$  en sí. En otras palabras, cuanto mayor es el valor absoluto de la solución, mayor es la desviación de la aproximación. Además, al analizar la evolución del error en las gráficas de entrenamiento y test, observamos que la diferencia entre la solución real y la aproximación, medida con la norma  $L^2$ , no disminuye con el número de iteraciones (Error relativo  $L^2$  en el gráfico).

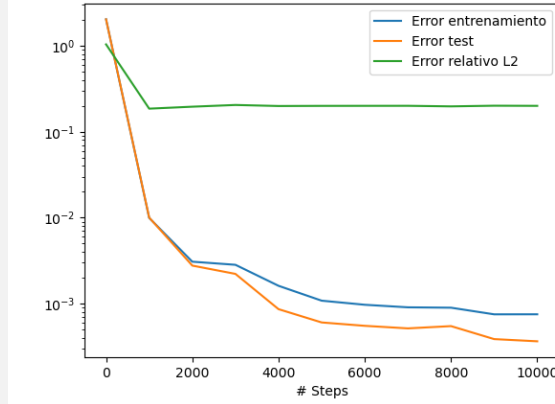


Figura 2.4: Error de la red neuronal

En esta gráfica, las curvas *Error entrenamiento* y *Error test* representan el error de la aproximación en los puntos de entrenamiento y test, respectivamente. Los puntos de entrenamiento corresponden al conjunto de colocación utilizado para aproximar la integral, mientras que los puntos de test pertenecen a un subconjunto del dominio donde se evalúa la aproximación. Por otro lado, el error relativo  $L^2$  cuantifica la diferencia entre la solución aproximada por la red neuronal y la solución real, integrando dicha diferencia sobre el dominio  $(-1, 1)$ . El código utilizado para este ejemplo se puede encontrar en el Anexo C.

Como hemos visto, las PINNs presentan errores significativos en ciertos problemas, lo que puede atribuirse a diversos factores. Uno de los principales desafíos de las PINNs es la falta de garantía de unicidad en la solución. A diferencia de los métodos numéricos tradicionales, las PINNs resuelven problemas de optimización no convexos, los cuales, por naturaleza, no aseguran una solución única. Esto implica que la red neuronal no tiene garantía de converger a la solución correcta y, de hecho, en muchos casos no lo hace. Es más, al igual que ocurre con otros algoritmos de inteligencia artificial, la red puede converger con la misma certeza (o error) hacia una solución incorrecta que hacia la solución real.

Otro obstáculo importante es la ausencia de una justificación teórica que determine cuáles son los hiperparámetros óptimos. Este problema añade incertidumbre al proceso de ajuste de la red, lo que complica la obtención de resultados precisos. Finalmente, otro aspecto problemático de las PINNs es que aproximan el residuo de forma fuerte. Esto significa que, si no existe una solución en el sentido fuerte (como ocurre en el ejemplo mostrado), la red neuronal no puede aproximar de forma adecuada la solución débil.

De estos problemas, los dos primeros son característicos del uso de redes neuronales, mientras que el tercero es específico del método. Con esto en mente, exploraremos un enfoque alternativo que considere la forma débil en la definición del residuo, lo que podría permitir superar esta limitación.

## 2.2. El “Deep Ritz Method”

El método “Deep Ritz” es una técnica numérica basada en las redes neuronales, que busca encontrar una solución a una ecuación en derivadas parciales utilizando su forma débil. La formulación de este método es más parecida al método de elementos finitos. La principal diferencia con el método de elementos finitos es el espacio donde se busca el mínimo. Nos restringimos a problemas autoadjuntos en los que la forma débil es equivalente a minimizar un funcional.

### 2.2.1. El problema elíptico autoadjunto

El problema elíptico autoadjunto permite caracterizar las soluciones de las EDP elípticas como el mínimo de un funcional. Esto conecta de manera lógica con el enfoque de redes neuronales para aproximar soluciones, donde la función de coste de la red neuronal se puede definir como esta función. Al minimizarla, se obtiene una solución aproximada del problema original. No obstante, esto lo veremos más adelante en detalle. De momento, consideramos una EDP de la forma vista en la Sección 1.2, es decir, una EDP elíptica sujeta a condiciones de contorno Dirichlet:

$$(2.7) \quad \begin{cases} -\sum_{i,j=1}^n \frac{\partial}{\partial x_j} \left( a_{ij}(\mathbf{x}) \frac{\partial u}{\partial x_i} \right) + \sum_{i=1}^n b_i(\mathbf{x}) \frac{\partial u}{\partial x_i} + c(\mathbf{x})u = f(\mathbf{x}), & \mathbf{x} \in \Omega, \\ u(\mathbf{x}) = g(\mathbf{x}), & \mathbf{x} \in \partial\Omega. \end{cases}$$

Se define el problema elíptico autoadjunto de la siguiente forma.

**Definición 2.3.** Dado un conjunto  $\Omega$ , acotado y abierto en  $\mathbb{R}^n$ , decimos que una ecuación en derivadas parciales elíptica es autoadjunta, si se cumplen las siguientes condiciones de simetría en los coeficientes.

$$(2.8) \quad \begin{aligned} a_{ij} &= a_{ji}, & i, j &= 1, \dots, n, \\ b_i &= 0, & i &= 1, \dots, n, \end{aligned}$$

Bajo estas condiciones, nuestro problema se puede formular de la siguiente forma.

$$(2.9) \quad \begin{cases} -\sum_{i,j=1}^n \frac{\partial}{\partial x_j} \left( a_{ij}(x) \frac{\partial u}{\partial x_i} \right) + c(x)u = f(x), & x \in \Omega, \\ u(x) = 0, & x \in \partial\Omega. \end{cases}$$

Por lo visto en la Sección 1.2, el problema (2.9) se puede reescribir en forma débil de la siguiente forma: encontrar  $u \in H_0^1(\Omega)$  tal que

$$(2.10) \quad a(u, v) = l(v) \quad \forall v \in H_0^1(\Omega).$$

Donde, al ser un problema autoadjunto, el funcional  $a(\cdot, \cdot)$  es simétrico

$$a(u, w) = a(w, u) \quad \forall u, w \in H_0^1(\Omega).$$

De ahora en adelante vamos a asumir que (2.9) cumple la condición de elipticidad uniforme y además se cumple que

$$c(x) - \frac{1}{2} \sum_{i=1}^n \frac{\partial b_i}{\partial x_i} \geq 0, \quad x \in \overline{\Omega},$$

por lo que, tal como se estableció en la Sección 1.2, el problema (2.9) admite una única solución débil. La solución débil de (2.10) no solo es única, sino que también puede caracterizarse como el mínimo de un funcional. Esto es así por el siguiente resultado.

**Lema 2.4.** *Definimos el funcional cuadrático  $J : H_0^1(\Omega) \rightarrow \mathbb{R}$  como*

$$J(u) = \frac{1}{2}a(v, v) - l(v), \quad u \in H_0^1(\Omega),$$

*de modo que las siguientes afirmaciones son equivalentes:*

1. *Encontrar el único  $u \in H_0^1(\Omega)$  tal que  $a(u, v) = l(v) \quad \forall v \in H_0^1(\Omega)$*
2. *Encontrar el único  $u \in H_0^1(\Omega)$  tal que  $J(u) \leq J(v) \quad \forall v \in H_0^1(\Omega)$*

**Demostración 2.5.** Sea  $u$  la única solución débil de (2.10) en  $H_0^1(\Omega)$  y, para  $v \in H_0^1(\Omega)$ , consideremos  $J(v) - J(u)$ :

$$\begin{aligned} J(v) - J(u) &= \frac{1}{2}a(v, v) - l(v) - \frac{1}{2}a(u, u) + l(u) \\ &= \frac{1}{2}a(v, v) - \frac{1}{2}a(u, u) - l(v - u) \\ &= \frac{1}{2}a(v, v) - \frac{1}{2}a(u, u) - a(u, v - u) \\ &= \frac{1}{2} [a(v, v) - 2a(u, v) + a(u, u)] \\ &= \frac{1}{2} [a(v, v) - a(u, v) - a(v, u) + a(u, u)] \\ &= \frac{1}{2}a(v - u, v - u). \end{aligned}$$

Por lo tanto,

$$J(v) - J(u) = \frac{1}{2}a(v - u, v - u).$$

Debido a (1.19),

$$a(v - u, v - u) \geq c_0 \|v - u\|_{H_0^1(\Omega)}^2,$$

donde  $c_0$  es una constante positiva. Así,

$$(2.11) \quad J(v) - J(u) \geq \frac{c_0}{2} \|v - u\|_{H_0^1(\Omega)}^2 \quad \forall v \in H_0^1(\Omega)$$

y, en consecuencia,

$$(2.12) \quad J(v) \geq J(u) \quad \forall v \in H_0^1(\Omega)$$

es decir,  $u$  minimiza  $J(\cdot)$  sobre  $H_0^1(\Omega)$ . De hecho,  $\tilde{u}$  es el único mínimo de  $J(\cdot)$  en  $H_0^1(\Omega)$ . En efecto, si  $\tilde{u}$  también minimiza  $J(\cdot)$  en  $H_0^1(\Omega)$ , entonces

$$(2.13) \quad J(v) \geq J(\tilde{u}) \quad \forall v \in H_0^1(\Omega).$$

Tomando  $v = \tilde{u}$  en (2.12) y  $v = u$  en (2.13), deducimos que

$$J(u) = J(\tilde{u});$$

pero entonces, en virtud de (2.11),

$$\|\tilde{u} - u\|_{H_0^1(\Omega)} = 0,$$

y, por lo tanto,  $u = \tilde{u}$ .

Para la implicación en sentido contrario, es fácil demostrar que  $J(\cdot)$  es convexa, es decir:

$$J((1 - \theta)v + \theta w) \leq (1 - \theta)J(v) + \theta J(w) \quad \forall \theta \in [0, 1], \quad \forall v, w \in H_0^1(\Omega).$$

Esto se deduce de la identidad

$$(1 - \theta)J(v) + \theta J(w) = J((1 - \theta)v + \theta w) + \frac{1}{2}\theta(1 - \theta)a(v - w, v - w)$$

y del hecho de que  $a(v - w, v - w) \geq 0$ . Además, si  $u$  minimiza  $J(\cdot)$ , entonces  $J(\cdot)$  tiene un punto estacionario en  $u$ ; es decir:

$$J'(u)v := \lim_{\lambda \rightarrow 0} \frac{J(u + \lambda v) - J(u)}{\lambda} = 0$$

para todo  $v \in H_0^1(\Omega)$ . Dado que

$$\frac{J(u + \lambda v) - J(u)}{\lambda} = a(u, v) - l(v) + \frac{\lambda}{2}a(v, v),$$

deducimos que si  $u$  minimiza  $J(\cdot)$  entonces

$$\lim_{\lambda \rightarrow 0} [a(u, v) - l(v) + \frac{\lambda}{2}a(v, v)] = a(u, v) - l(v) = 0 \quad \forall v \in H_0^1(\Omega),$$

lo que demuestra el resultado.  $\square$

### 2.2.2. Formulación del método “Deep Ritz”

El método Deep Ritz, propuesto por E y Yu en [2], se basa en minimizar el funcional  $J(u)$  (definido en el Lema 2.4) representando la solución desconocida mediante una red neuronal  $\hat{u}_\theta : \mathbb{R}^d \rightarrow \mathbb{R}$ , en lugar de emplear funciones base del método de los elementos finitos. Es decir, en lugar de aproximar la función  $u \in H_0^1(\Omega)$  que minimiza  $J(u)$  dentro del espacio de elementos finitos  $V_h$ , se busca una aproximación en el espacio de funciones red neuronal. Aquí,  $\theta \in \mathbb{R}^{\#\mathcal{N}}$  denota el conjunto de parámetros de la red, donde  $\#\mathcal{N}$  es el número total de parámetros libres.

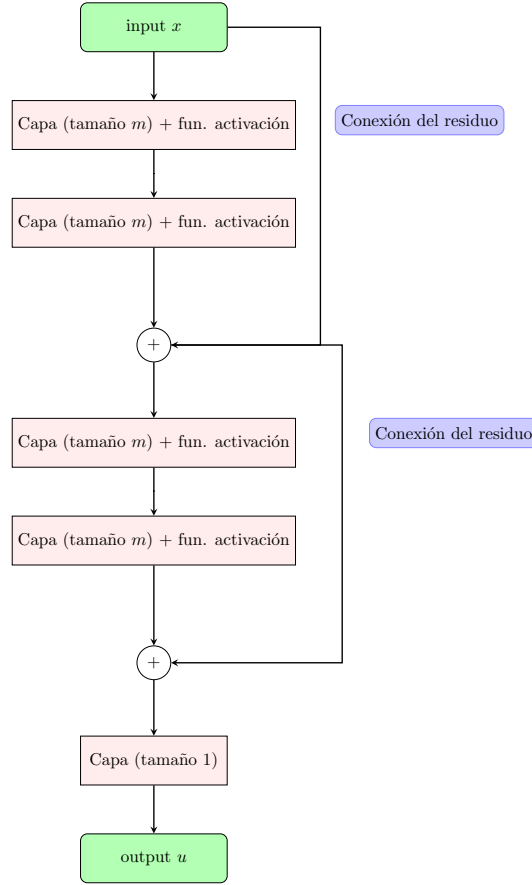


Figura 2.5: La figura muestra una red con dos bloques y una capa lineal de salida. Cada bloque consiste en dos capas completamente conectadas y una conexión de salto.

El marco teórico del método Deep Ritz requiere que la red neuronal sea diferenciable en el sentido débil. No obstante, E y Yu señalan que el uso de funciones de activación diferenciables mejora los resultados prácticos. La Figura 2.5 muestra la arquitectura de la red neuronal profunda utilizada en [2].

Dada una arquitectura de red  $\mathcal{N}$ , el espacio de aproximación  $V_{\mathcal{N}}$  utilizado en el método Deep Ritz se define como:

$$V_{\mathcal{N}} := \{\hat{u}_{\theta} : \mathbb{R}^d \rightarrow \mathbb{R} \mid \theta \in \mathbb{R}^{\#\mathcal{N}}\}.$$

Mientras que  $\#\mathcal{N}$  sea finito y las funciones de activación sean diferenciables, se cumple que  $V_{\mathcal{N}} \subset H^1(\Omega)$ . Sin embargo, en general, para  $u_{\mathcal{N}} \in V_{\mathcal{N}}$  no se garantiza que  $u_{\mathcal{N}} = 0$  en  $\partial\Omega$ , por lo que  $V_{\mathcal{N}} \not\subset V = H_0^1(\Omega)$ . Además, es importante notar que  $V_{\mathcal{N}}$  no es un espacio vectorial, ya que para  $v_1, v_2 \in V_{\mathcal{N}}$  no necesariamente se cumple que  $v_1 + v_2 \in V_{\mathcal{N}}$ .

Para abordar esta limitación, se introduce el funcional de energía penalizado:

$$J_\lambda(v) := J(v) + \frac{\lambda}{2} |v|_{\partial\Omega}^2,$$

donde  $\lambda \in \mathbb{R}^+$  es un parámetro de penalización y  $|\cdot|_{\partial\Omega}$  denota la norma  $L^2$  en la frontera del dominio. El término de penalización adicional impone que  $v$  sea próximo a cero en  $\partial\Omega$ .

Esta formulación permite el uso de redes neuronales en la aproximación de ecuaciones diferenciales parciales sin necesidad de definir funciones de base explícitas, facilitando la resolución de problemas de altas dimensiones donde los métodos clásicos pueden volverse ineficientes. No obstante, igual que en el caso de las PINN el problema de la formulación residía en que se aproximaba el residuo de forma fuerte, en el método Deep Ritz, el problema radica en que se minimiza el funcional  $J_\lambda$  en lugar de  $J$ , es decir, se altera el problema para imponer las condiciones de contorno, variando así la solución. Para ver esto de forma más clara, vamos a aplicar el método Deep Ritz a la ecuación de Poisson en una dimensión.

**Ejemplo 2.6.** Consideremos la ecuación de Poisson en una dimensión, con  $f = 1$  y  $\Omega = (0, 1)$

$$(2.14) \quad -u'' = 1, \quad u(0) = u(1) = 0,$$

cuya solución exacta es

$$u(x) = -x^2/2 + x/2.$$

En forma débil, la ecuación (2.14) se puede escribir como: encontrar  $u \in H^1(0, 1)$  tal que:

$$\int_0^1 u'v' = \int_0^1 v \quad \forall v \in H^1(0, 1).$$

en cuyo caso, la forma bilineal  $a$  y lineal  $l$  asociadas son:

$$\begin{aligned} a(u, v) &= \int_0^1 u'v', \\ l(v) &= \int_0^1 v. \end{aligned}$$

Por lo tanto, el funcional  $J(u)$  asociado a este problema es:

$$J(u) = \frac{1}{2} \int_0^1 (u')^2 - \int_0^1 u.$$



Con esta información podemos plantear el método de Ritz para este problema como encontrar  $\hat{u}_\theta \in V_{\mathcal{N}}$  tal que:

$$\begin{aligned} & \arg \min_{\{\hat{u}_\theta: \theta \in \mathbb{R}^{\#\mathcal{N}}\}} \left( \frac{1}{2} \int_{\Omega} \left( (\hat{u}'_\theta)^2 - \int_{\Omega} \hat{u}_\theta + \lambda(\hat{u}_\theta(0)^2 + \hat{u}_\theta(1)^2) \right) \right) = \\ & \arg \min_{\{\hat{u}_\theta: \theta \in \mathbb{R}^{\#\mathcal{N}}\}} \left( \frac{1}{2} \int_0^1 \left( (\hat{u}'_\theta)^2 - \int_0^1 \hat{u}_\theta + \lambda(\hat{u}_\theta(0)^2 + \hat{u}_\theta(1)^2) \right) \right), \end{aligned}$$

donde  $\hat{u}_\theta$  es una función de una red neuronal. Así, el funcional que estamos minimizando,  $J_\lambda: H^1(0,1) \rightarrow \mathbb{R}$  se define como:

$$J_\lambda(u) = \frac{1}{2} \int_0^1 \left( (u')^2 - \int_0^1 u + \lambda(u(0)^2 + u(1)^2) \right).$$

Si buscamos el mínimo de  $J_\lambda$ , vamos a encontrar que  $u$  verifica la siguiente ecuación:

$$J'_\lambda(u)v := \lim_{\beta \rightarrow 0} \frac{J_\lambda(u + \beta v) - J_\lambda(u)}{\beta} = 0, \quad \forall v \in H^1(0,1).$$

Es fácil ver que  $u \in H^1(0,1)$  verifica

$$\int_0^1 u'v' - \int_0^1 v + 2\lambda u(0)v(0) + 2\lambda u(1)v(1) = 0, \quad \forall v \in H^1(0,1).$$

Consideramos ahora el siguiente problema: encontrar  $u \in C^2(0,1)$  tal que

$$-u'' = f, \quad u'(0) = 2\lambda u(0), \quad u'(1) = -2\lambda u(1),$$

que tiene como solución para  $f = 1$

$$u_\lambda = -x^2/2 + x/2 + 1/(4\lambda).$$

La solución débil de este problema es: encontrar  $u \in H^1(0,1)$  tal que

$$\int_0^1 u'v' + 2\lambda u(1)v(1) + 2\lambda u(0)v(0) = \int_0^1 v, \quad \forall v \in H^1(0,1).$$

O lo que es lo mismo, encontrar  $u \in H^1(0,1)$  tal que

$$\int_0^1 u'v' + 2\lambda v(1)u(1) + 2\lambda v(0)u(0) = \int_0^1 v, \quad \forall v \in H^1(0,1).$$

Con este ejemplo, nos damos cuenta de que el método Deep Ritz, en su formulación utilizada aquí, no aproxima exactamente el problema original, sino que en su lugar resuelve una ecuación con condiciones de frontera modificadas. Esto introduce un sesgo en la solución obtenida, lo que implica que el método no es exacto para el problema original de Poisson con condiciones de Dirichlet homogéneas. Siendo más específicos, el método Deep Ritz estaría aproximando la solución  $u_\lambda(x) = -x^2/2 + x/2 + 1/(4\lambda)$

en lugar de la solución exacta  $u(x) = -x^2/2 + x/2$ , lo que implica un error de  $O(1/\lambda)$  en la solución obtenida.

A continuación, vamos a plantear la ecuación de Poisson en un dominio cuadrado y en una  $L$ , para lo cual he tenido que programar el método Deep Ritz en Python usando PyTorch. El objetivo con este ejemplo es entender cómo se comporta el método Deep Ritz ante los distintos valores que puede tomar el parámetro de penalización  $\lambda$ .

**Ejemplo 2.7.** Consideramos el siguiente problema de Poisson en un cuadrado unitario  $\Omega = (0, 1)^2$  con condiciones de contorno Dirichlet homogéneas:

$$(2.15) \quad \begin{cases} -\Delta u = 1, & \text{en } \Omega, \\ u = 0, & \text{en } \partial\Omega. \end{cases}$$

Para aplicar el método, lo primero que necesitamos es definir el funcional  $J_\lambda(u)$  asociado a este problema. Para ello, partimos de la forma débil del mismo, ya obtenida en el Ejemplo 1.8. Sea  $v$  una función del espacio de funciones  $H_0^1(\Omega)$ , definido como:

$$H_0^1(\Omega) = \{v \in H^1(\Omega) \mid v = 0 \text{ en } \partial\Omega\},$$

la forma débil del problema (2.15) consiste en encontrar  $u \in H_0^1(\Omega)$  tal que:

$$\int_{\Omega} \nabla u \nabla v \, dx = \int_{\Omega} v \, dx, \quad \forall v \in H_0^1(\Omega).$$

El funcional de energía asociado se define entonces como:

$$J(u) = \frac{1}{2} \int_{\Omega} |\nabla u|^2 \, dx - \int_{\Omega} u \, dx.$$

Así, el funcional  $J_\lambda(u)$  se define como:

$$J_\lambda(u) = \frac{1}{2} \int_{\Omega} |\nabla u|^2 - \int_{\Omega} u + \lambda \int_{\partial\Omega} u^2.$$

Para resolver este problema, implementé el método Deep Ritz en Python utilizando la biblioteca PyTorch. La red neuronal empleada tiene una arquitectura 2-64-64-1. La Figura 2.6 muestra la solución obtenida con  $\lambda = 500$ , mientras que la Figura 2.7 presenta la diferencia con la solución obtenida mediante el método de los elementos finitos en el Ejemplo 1.8, donde se utilizó una malla cuadrada con 50 nodos por lado. En dicho ejemplo, el error cometido respecto a la solución con una malla de 100 nodos por lado fue de  $3,12 \cdot 10^{-5}$ , medido en norma  $L^2$ . En cambio, la solución proporcionada por el método Deep Ritz muestra diferencias notables respecto a la obtenida por elementos finitos. De hecho, a pesar de corresponder a uno de los mejores resultados que se pueden alcanzar con este enfoque para este problema concreto, la solución sigue alejada de la real.

Siguiendo la intuición del caso unidimensional, al reducir  $\lambda$  a  $10^{-5}$ , esperamos una aproximación mucho peor. Y, efectivamente, la solución resultante no se parece en nada a la real. La Figura 2.8 muestra este resultado.

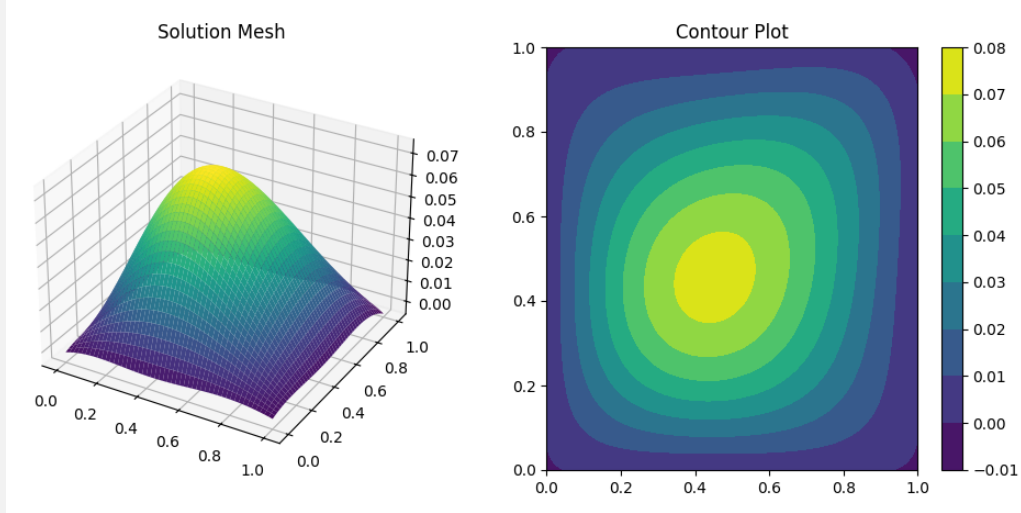


Figura 2.6: Solución de la ecuación de Poisson en un cuadrado unitario con  $\lambda = 500$ .

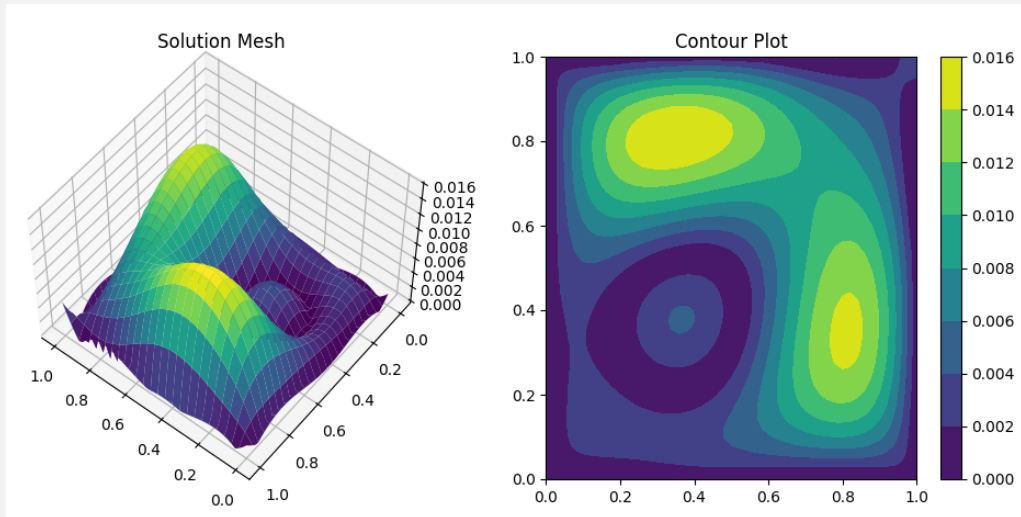


Figura 2.7: Diferencia entre FEM y Deep Ritz con  $\lambda = 500$ .

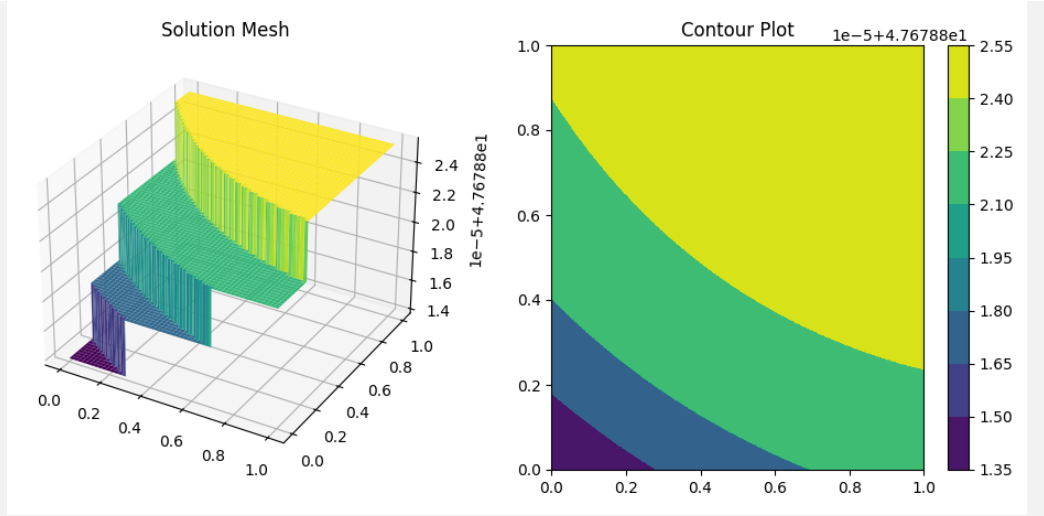


Figura 2.8: Solución de la ecuación de Poisson en un cuadrado unitario con  $\lambda = 10^{-5}$ .

El código que se ha utilizado para implementar el método Deep Ritz en este problema se puede encontrar en el Anexo [D](#).

El método Deep Ritz ofrece una formulación efectiva para superar las limitaciones de las PINN tradicionales, que imponen el residuo en forma fuerte. Sin embargo, como se ha ilustrado en los ejemplos, la manera en que este método maneja las condiciones de contorno no es del todo óptima. En particular, el uso de un término de penalización para forzar el cumplimiento de las condiciones de frontera introduce una modificación en el problema original, lo que puede generar soluciones sesgadas, especialmente cuando el parámetro de penalización  $\lambda$  no es suficientemente grande.

En comparación con el método de elementos finitos (FEM), el Deep Ritz tiene la ventaja de no requerir la construcción explícita de funciones base, lo que resulta especialmente útil en problemas de alta dimensionalidad. No obstante, esta misma característica es también una limitación, ya que, en el FEM, las funciones base garantizan automáticamente el cumplimiento de las condiciones de contorno, mientras que en el método Deep Ritz esto debe ser impuesto externamente, con los problemas asociados que hemos analizado.

## CAPÍTULO 3

# Conclusiones

---

El uso de redes neuronales para la aproximación de ecuaciones en derivadas parciales es un campo con un gran potencial teórico. Su capacidad de aproximar funciones complejas y de operar en dimensiones altas sin la necesidad de una malla explícita las hace atractivas en escenarios donde los métodos numéricos tradicionales pueden resultar costosos o impracticables. No obstante, en la práctica, su aplicación a problemas relativamente simples muestra importantes deficiencias en comparación con métodos bien establecidos como el método de los elementos finitos (FEM).

Los enfoques basados en redes neuronales, como las PINNs (Physics-Informed Neural Networks), presentan limitaciones importantes al forzar el cumplimiento de la ecuación diferencial en su forma fuerte. Esto implica que, en situaciones donde no existe una solución en este sentido, la red neuronal no puede aproximar correctamente la solución débil. Además, la falta de garantías de convergencia y la ausencia de una base matemática sólida para la selección de hiperparámetros complican su aplicación efectiva.

Por otro lado, el método Deep Ritz plantea una alternativa más alineada con la formulación variacional de los problemas elípticos, minimizando un funcional en lugar de imponer restricciones puntuales. Sin embargo, el uso de penalizaciones para imponer condiciones de contorno introduce modificaciones en el problema original, lo que afecta la precisión de la solución obtenida. A pesar de ello, este enfoque podría ser más viable en problemas de alta dimensionalidad, donde los métodos tradicionales pierden eficiencia.

En resumen, aunque las redes neuronales ofrecen herramientas novedosas para la resolución de ecuaciones diferenciales, su aplicación en contextos prácticos aún enfrenta numerosos desafíos. Su rendimiento es inferior al de métodos numéricos establecidos en escenarios convencionales, y la interpretabilidad de los resultados sigue siendo limitada. La dificultad para diagnosticar fallos y la falta de garantías teóricas claras refuerzan la idea de que, por el momento, estas técnicas deben verse como complementarias a los métodos clásicos, en lugar de como un reemplazo directo.



# Bibliografía

---

- [1] Yuri Aikawa, Naonori Ueda y Toshiyuki Tanaka. “Improving the efficiency of training physics-informed neural networks using active learning”. En: *New Generation Computing* (2024), págs. 1-22.
- [2] Weinan E y Bing Yu. “The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems”. En: *Communications in Mathematics and Statistics* 6.1 (2018), págs. 1-12. URL: <https://doi.org/10.48550/arXiv.1710.00211>.
- [3] Catherine F. Higham y Desmond J. Higham. “Deep Learning: An Introduction for Applied Mathematicians”. En: *SIAM Review* 61.4 (2019), págs. 860-891. DOI: [10.1137/18M1165748](https://doi.org/10.1137/18M1165748). eprint: <https://doi.org/10.1137/18M1165748>. URL: <https://doi.org/10.1137/18M1165748>.
- [4] Jie Hou, Ying Li y Shihui Ying. “Enhancing PINNs for solving PDEs via adaptive collocation point movement and adaptive loss weighting”. En: *Nonlinear Dynamics* 111.16 (2023), págs. 15233-15261.
- [5] Lu Lu et al. “DeepXDE: A deep learning library for solving differential equations”. En: *SIAM Review* 63.1 (2021), págs. 208-228. DOI: [10.1137/19M1274067](https://doi.org/10.1137/19M1274067).
- [6] Tao Luo y Qixuan Zhou. *On Residual Minimization for PDEs: Failure of PINN, Modified Equation, and Implicit Bias*. 2023. arXiv: [2310.18201](https://arxiv.org/abs/2310.18201) [math.AP]. URL: <https://arxiv.org/abs/2310.18201>.
- [7] Takashi Matsubara y Takaharu Yaguchi. *Good Lattice Training: Physics-Informed Neural Networks Accelerated by Number Theory*. 2023. arXiv: [2307.13869](https://arxiv.org/abs/2307.13869) [cs.LG]. URL: <https://arxiv.org/abs/2307.13869>.
- [8] Marcus Münzer y Chris Bard. *A Curriculum-Training-Based Strategy for Distributing Collocation Points during Physics-Informed Neural Network Training*. 2022. arXiv: [2211.11396](https://arxiv.org/abs/2211.11396) [cs.LG]. URL: <https://arxiv.org/abs/2211.11396>.
- [9] Allan Pinkus. “Approximation theory of the MLP model in neural networks”. En: *Acta numerica* 8 (1999), págs. 143-195.
- [10] Shashank Subramanian et al. *Adaptive Self-supervision Algorithms for Physics-informed Neural Networks*. 2022. arXiv: [2207.04084](https://arxiv.org/abs/2207.04084) [cs.LG]. URL: <https://arxiv.org/abs/2207.04084>.





## APÉNDICE A

# Código en Matlab para la red neuronal de clasificación

---

El código aquí escrito está basado en el propuesto por Catherine et al. en [3] pero extendiéndolo para incluir más puntos de entrenamiento de distinto tipo.

Lo primero que se hace es declarar cada punto de entrenamiento y su respectiva etiqueta. En este caso, las etiquetas son vectores de 3 componentes, donde cada componente representa la probabilidad de que el punto pertenezca a una de las tres clases.

```
1 function pclassifier_more_points
2
3 % xcoords, ycoords, targets
4 x1 = [0.1,0.3,0.1,0.2,0.4,0.6,0.5,0.9,0.4,0.7,0.3,0.4,0.2,0.1,0.5,
      0.9,0.8];
5 x2 = [0.1,0.4,0.5,0.2,0.2,0.3,0.6,0.2,0.4,0.6,0.9,0.7,0.6,0.8,0.8,
      0.6,0.1];
6 y = [ones(1,5) zeros(1,11) 1; zeros(1,5) ones(1,5) zeros(1,7); zeros
      (1,10) ones(1,6) 0];
```

A continuación, visualizamos los puntos creados. En este caso, la primera clase esta representada por círculos rojos, la segunda por cruces azules y la tercera por triángulos verdes.

```
1 figure(1)
2 clf
3 a1 = subplot(1,1,1);
4 plot(x1(1:5),x2(1:5),'ro','MarkerSize',12,'LineWidth',4)
5 hold on
6 plot(x1(17),x2(17),'ro','MarkerSize',12,'LineWidth',4)
7 plot(x1(6:10),x2(6:10),'bx','MarkerSize',12,'LineWidth',4)
8 plot(x1(11:16),x2(11:16),'gv','MarkerSize',12,'LineWidth',4)
9 a1.XTick = [0 1];
10 a1.YTick = [0 1];
11 a1.FontWeight = 'Bold';
12 a1.FontSize = 10;
13 xlim([0,1])
14 ylim([0,1])
15
16 print -dpng pic_more_points.png
```

Es siguiente paso es crear las matrices de pesos y sesgos que van a definir la red neuronal. En este caso, la red tiene 2 capas ocultas y una capa de salida. La primera capa oculta tiene 2 neuronas, la segunda 3 y la capa de salida 3 neuronas.

```

1
2 rng(5000);
3 W2 = 0.5*randn(2,2);
4 W3 = 0.5*randn(3,2);
5 W4 = 0.5*randn(3,3);
6 b2 = 0.5*randn(2,1);
7 b3 = 0.5*randn(3,1);
8 b4 = 0.5*randn(3,1);

```

Finalmente, queda el paso de entrenamiento de la red neuronal. Este sigue el algoritmo de retropropagación, donde se calcula el gradiente del coste y se actualizan los pesos y sesgos de la red.

```

1 eta = 0.05;
2 Niter = 1e6;
3 savecost = zeros(Niter,1);
4 for counter = 1:Niter
5     k = randi(17);
6     x = [x1(k); x2(k)];
7     % Forward pass
8     a2 = activate(x,W2,b2);
9     a3 = activate(a2,W3,b3);
10    a4 = activate(a3,W4,b4);
11    % Backward pass
12    delta4 = a4.*(1-a4).*(a4-y(:,k));
13    delta3 = a3.*(1-a3).*(W4'*delta4);
14    delta2 = a2.*(1-a2).*(W3'*delta3);
15    % Gradient step
16    W2 = W2 - eta*delta2*x';
17    W3 = W3 - eta*delta3*a2';
18    W4 = W4 - eta*delta4*a3';
19    b2 = b2 - eta*delta2;
20    b3 = b3 - eta*delta3;
21    b4 = b4 - eta*delta4;
22    % Monitor progress
23    newcost = cost(W2,W3,W4,b2,b3,b4); % display cost to screen
24    savecost(counter) = newcost;
25 end

```

Acabado el entrenamiento, el paso final es visualizar los resultados, para ver que regiones asigna la red neuronal a cada una de las clases existentes.

```

1 figure(2)
2 clf
3 semilogy(1:1e4:Niter,savecost(1:1e4:Niter),'b-','LineWidth',2)
4 xlabel('Iteration Number')
5 ylabel('Value of cost function')
6 set(gca,'FontWeight','Bold','FontSize',10)
7 print -dpng pic_cost_more_points.png
8
9 %%%%%%%%%%% Display shaded and unshaded regions
10 N = 500;
11 Dx = 1/N;

```

```

12 Dy = 1/N;
13 xvals = 0:Dx:1;
14 yvals = 0:Dy:1;
15 for k1 = 1:N+1
16     xk = xvals(k1);
17     for k2 = 1:N+1
18         yk = yvals(k2);
19         xy = [xk;yk];
20         a2 = activate(xy,W2,b2);
21         a3 = activate(a2,W3,b3);
22         a4 = activate(a3,W4,b4);
23         Aval(k2,k1) = a4(1);
24         Bval(k2,k1) = a4(2);
25         Cval(k2,k1) = a4(3);
26     end
27 end
28 [X,Y] = meshgrid(xvals,yvals);
29
30 figure(3)
31 clf
32 a2 = subplot(1,1,1);
33
34 Mval = ones(size(Aval));
35
36 Mval(Cval >= Aval & Cval >= Bval) = 2;
37 Mval(Aval >= Bval & Aval >= Cval) = 1;
38 Mval(Bval >= Aval & Bval >= Cval) = 0;
39
40 contourf(X, Y, Mval, [-1 0.75 1.5 2.5]);
41
42 hold on
43
44 colormap([1 1 1; 0.8 0.8 0.8; 0.5 0.5 0.5]);
45
46 plot(x1(1:5),x2(1:5),'ro','MarkerSize',12,'LineWidth',4)
47 plot(x1(17),x2(17),'ro','MarkerSize',12,'LineWidth',4)
48 plot(x1(6:10),x2(6:10),'bx','MarkerSize',12,'LineWidth',4)
49 plot(x1(11:16),x2(11:16),'yv','MarkerSize',12,'LineWidth',4)
50 a2.XTick = [0 1];
51 a2.YTick = [0 1];
52 a2.FontWeight = 'Bold';
53 a2.FontSize = 10;
54 xlim([0,1])
55 ylim([0,1])
56
57 print -dpng classifier_back_more_points.png
58
59 function costval = cost(W2,W3,W4,b2,b3,b4)
60
61     costvec = zeros(10,1);
62     for i = 1:10
63         x = [x1(i);x2(i)];
64         a2 = activate(x,W2,b2);
65         a3 = activate(a2,W3,b3);
66         a4 = activate(a3,W4,b4);
67         costvec(i) = norm(y(:,i) - a4,2);
68     end

```

```
69     costval = norm(costvec,2)^2;  
70     end % of nested function  
71  
72 end
```

## APÉNDICE B

# Código en Python para el método de los elementos finitos en un cuadrado

---

El primer paso consiste en definir la malla sobre la que se realizará la aproximación. Se elige un número de puntos en cada dirección, denotado como  $n$ , y se calcula el tamaño del paso  $h = 1/n$ . Esto permite dividir el dominio  $\Omega = [0, 1] \times [0, 1]$  en una malla uniforme.

```
1 import numpy as np
2 import scipy.sparse as sp
3 import scipy.sparse.linalg as spla
4 import matplotlib.pyplot as plt
5
6 n = 50
7 h = 1 / n
```

El siguiente paso es definir la matriz de coeficientes del sistema lineal, que representa la discretización del operador diferencial en la malla. Se utiliza una combinación de matrices tridiagonales y productos de Kronecker para construir la estructura del problema.

```
1 e = np.ones(n-1)
2 B = sp.diags([-e, 4*e, -e], [-1, 0, 1], shape=(n-1, n-1))
3 I = sp.eye(n-1)
4 I1 = sp.diags([-e, -e], [-1, 1], shape=(n-1, n-1))
5 A = sp.kron(I, B) + sp.kron(I1, I)
6 A /= h**2
```

Se define el término independiente del sistema, que en este caso es un vector lleno de unos en todo el dominio, excepto en la frontera. Esto refleja que la solución aproximada debe tomar el valor 1 en cada punto interior de la malla y 0 en la frontera.

```
1 f = np.ones((n-1)**2)
```

Para obtener la solución, se resuelve el sistema lineal  $Ay = f$  utilizando la función `spsolve` de la librería `scipy`. El resultado se almacena en la variable  $y$ .

```
1 y = spla.spsolve(A, f)
```

Finalmente, la solución se reorganiza en una matriz para representarla gráficamente. Se muestran dos visualizaciones: una malla tridimensional y un gráfico de contorno, que permiten interpretar mejor la distribución de valores obtenidos.

```
1 val = np.zeros((n-1, n-1))
2 for i in range(n-1):
3     for j in range(n-1):
4         val[i, j] = y[j + (n-1) * i]
5
6 valnn = np.zeros((n+1, n+1))
7 valnn[1:n, 1:n] = val
8
9 xx = np.linspace(0, 1, n+1)
10 yy = xx
11 X, Y = np.meshgrid(xx, yy)
12
13 fig = plt.figure(figsize=(12, 5))
14
15 ax1 = fig.add_subplot(121, projection='3d')
16 ax1.plot_surface(X, Y, valnn, cmap='viridis')
17 ax1.set_title('Solution Mesh')
18
19 ax2 = fig.add_subplot(122)
20 c = ax2.contourf(X, Y, valnn, cmap='viridis')
21 plt.colorbar(c, ax=ax2)
22 ax2.set_title('Contour Plot')
23
24 plt.show()
```

## APÉNDICE C

# Código en Python para construir PINNs con la librería DeepXDE

---

El siguiente código es el utilizado en el ejemplo 2.2. Lo primero que se hace es importar las librerías necesarias para el funcionamiento del código. En este caso, se utilizan las librerías de DeepXDE para la implementación de la red neuronal y Matplotlib para la visualización. Además, se define la ecuación en derivadas parciales (EDP) que se va a resolver.

```
1 import deepxde as dde
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import tensorflow as tf
5
6 def pde(x, u):
7     A = 0.5 * tf.cast(tf.less(x, 0), tf.float32) + 1 * tf.cast(tf.
8         greater_equal(x, 0), tf.float32)
9     du_x = dde.grad.jacobian(u, x)
10    dA_du_x = dde.grad.jacobian(A * du_x, x)
11    f = 0 * tf.cast(tf.less(x, 0), tf.float32) + (-2) * tf.cast(tf.
12        greater_equal(x, 0), tf.float32)
13    return -dA_du_x - f
```

A continuación se define la solución débil de la EDP, que se utilizará como referencia para evaluar el rendimiento de la red neuronal. Además, se definen las condiciones de frontera. En este caso, se utiliza una condición de Dirichlet en la frontera del dominio.

```
1 def weak_solution(x):
2     return (-2/3 * x - 2/3) * (x < 0) + (x**2 - (1/3) * x - (2/3)) * (x
3         >= 0)
4
5 def boundary(x, on_boundary):
6     return on_boundary
7
8 geom = dde.geometry.Interval(-1, 1)
9 bc = dde.DirichletBC(geom, weak_solution, boundary)
```

Seguidamente, definimos el objeto red neuronal que se va a utilizar para aproximar la solución de la EDP así como el método de optimización.

```
1 data = dde.data.PDE(geom, pde, bc, num_domain=1000, num_boundary=2,  
    num_test=100, solution=weak_solution)  
2  
3 net = dde.maps.FNN([1] + [256] * 3 + [1], "tanh", "Glorot normal")  
4  
5 model = dde.Model(data, net)  
6  
7 model.compile("adam", lr=0.001, metrics=["l2 relative error"])
```

Con todos los parámetros del problema definidos, nos apoyamos en el método `train` de la librería DeepXDE para entrenar la red neuronal.

```
1 losshistory, train_state = model.train(iterations=10000)
```

Finalmente, se guarda el historial de pérdidas y se visualiza la solución aproximada. Se generan 1000 puntos uniformemente distribuidos en el intervalo  $[-1, 1]$  y se evalúa la red neuronal en esos puntos. Luego, se grafican los resultados.

```
1 dde.saveplot(losshistory, train_state, issave=True, isplot=True)  
2  
3 x = geom.uniform_points(1000, True)  
4 y = model.predict(x, operator=pde)  
5 plt.figure()  
6 plt.plot(x, y)  
7 plt.xlabel("x")  
8 plt.ylabel("Residuo de la EDP")  
9 plt.show()
```



## APÉNDICE D

# Código en Python método DeepRitz

---

Este código implementa una red neuronal para resolver ecuaciones en derivadas parciales mediante el método de Ritz. Se describe la construcción del modelo, el proceso de entrenamiento y la evaluación de la solución.

Se importan las bibliotecas necesarias para realizar gráficas, implementar redes neuronales en PyTorch y realizar operaciones matemáticas.

```
1 import numpy as np
2 import math, torch, generateData, time
3 import torch.nn.functional as F
4 from torch.optim.lr_scheduler import MultiStepLR, StepLR
5 import torch.nn as nn
6 import matplotlib.pyplot as plt
7 import sys, os
8 import writeSolution
9 import numpy as np
10 import scipy.sparse as sp
11 import scipy.sparse.linalg as spla
12 import matplotlib.pyplot as plt
```

Se define la clase RitzNet, que implementa una red neuronal profunda para aproximar soluciones.

```
1 class RitzNet(torch.nn.Module):
2     def __init__(self, params):
3         super(RitzNet, self).__init__()
4         self.params = params
5         self.linearIn = nn.Linear(self.params["d"], self.params["width"
6 ])
7         self.linear = nn.ModuleList()
8         for _ in range(params["depth"]):
9             self.linear.append(nn.Linear(self.params["width"], self.
10 params["width"]))
11
12         self.linearOut = nn.Linear(self.params["width"], self.params["
13 dd"])
14
15     def forward(self, x):
```

```

13     x = torch.tanh(self.linearIn(x))
14     for layer in self.linear:
15         x_temp = torch.tanh(layer(x))
16         x = x_temp
17
18     return self.linearOut(x)

```

Se define el proceso de entrenamiento de la red neuronal mediante el método de optimización de Adam.

```

1 def train(model, device, params, optimizer, scheduler):
2     ratio = (4*2.0+2*math.pi*0.3)/(2.0*2.0-math.pi*0.3**2)
3     model.train()
4
5     data1 = torch.from_numpy(generateData.sampleFromSquare(1, params["
6 bodyBatch"])).float().to(device)
7     data2 = torch.from_numpy(generateData.sampleFromSquareBoundary(1,
8 params["bdryBatch"])).float().to(device)
9     x_shift = torch.from_numpy(np.array([params["diff"], 0.0])).float().
10 to(device)
11     y_shift = torch.from_numpy(np.array([0.0, params["diff"]])).float().
12 to(device)
13     data1_x_shift = data1+x_shift
14     data1_y_shift = data1+y_shift
15
16     loss_history = []
17
18     for step in range(params["trainStep"]-params["preStep"]):
19         output1 = model(data1)
20         output1_x_shift = model(data1_x_shift)
21         output1_y_shift = model(data1_y_shift)
22
23         dfdx = (output1_x_shift-output1)/params["diff"]
24         dfdy = (output1_y_shift-output1)/params["diff"]
25
26         model.zero_grad()
27
28         fTerm = ffun(data1).to(device)
29         loss1 = torch.mean(0.5*(dfdx*dfdx+dfdy*dfdy)-fTerm*output1)
30
31         output2 = model(data2)
32         target2 = exact(data2)
33         loss2 = torch.mean((output2-target2)*(output2-target2) * params
34 ["penalty"] * ratio)
35         loss = loss1+loss2
36         loss_history.append(loss.item())
37
38         if step%params["writeStep"] == params["writeStep"]-1:
39             with torch.no_grad():
40                 target = exact(data1)
41                 error = errorFun(output1, target, params)
42
43                 print("Loss at Step %s is %s"%(step+params["preStep"
44 ]+1, loss.item()))
45                 file = open("lossData.txt", "a")

```

```

42         file.write(str(step+params["preStep"]+1)+" "+str(error)+"\n
43     ")
44     if step%params["sampleStep"] == params["sampleStep"]-1:
45         data1 = torch.from_numpy(generateData.sampleFromSquare(1,
46             params["bodyBatch"])).float().to(device)
47         data2 = torch.from_numpy(generateData.
48             sampleFromSquareBoundary(1, params["bdryBatch"])).float().to(device
49     )
50
51     data1_x_shift = data1+x_shift
52     data1_y_shift = data1+y_shift
53
54     if 10*(step+1)%params["trainStep"] == 0:
55         print("%s%% finished..."%(100*(step+1)//params["trainStep"
56     ]))
57
58     loss.backward()
59
60     optimizer.step()
61     scheduler.step()
62
63     return loss_history

```

Se incluyen funciones para calcular el error y visualizar la evolución de la pérdida durante el entrenamiento.

```

1 def errorFun(output, target, params):
2     error = output - target
3     error = math.sqrt(torch.mean(error * error))
4     ref = math.sqrt(torch.mean(target * target))
5
6     if ref == 0:
7         return error
8
9     return error / ref
10
11 def test(model, device, params):
12     numQuad = params["numQuad"]
13
14     data = torch.from_numpy(generateData.sampleFromSquare(1,numQuad)).
15     float().to(device)
16     output = model(data)
17     target = exact(data).to(device)
18
19     error = output - target
20     error = math.sqrt(torch.mean(error * error))
21     ref = math.sqrt(torch.mean(target * target))
22
23     if ref == 0:
24         return error
25
26     return error / ref
27
28 def ffun(data):
29     return torch.ones([data.shape[0],1],dtype=torch.float)

```

```

30 def exact(data):
31     return torch.zeros((data.shape[0], 1), dtype=torch.float, device=
        data.device)
32
33
34 def count_parameters(model):
35     return sum(p.numel() for p in model.parameters())
36
37 def plot_loss(loss_history):
38     plt.figure(figsize=(8, 6))
39     plt.plot(range(len(loss_history)), loss_history, linestyle='-')
40     plt.xlabel("Iterations (every {} steps)".format(500))
41     plt.ylabel("Loss")
42     plt.yscale("log")
43     plt.title("Training Loss Over Time (Log Scale)")
44     plt.grid(True, which="both", linestyle="--", linewidth=0.5)
45     plt.show()

```

La función `principal` define los parámetros, inicializa la red, ejecuta el entrenamiento y evaluación y grafica los resultados. Los parámetros utilizados tienen el siguiente significado:

- `d`: Dimensionalidad del problema (2 para el cuadrado).
- `dd`: Dimensionalidad de la salida (1 para un escalar).
- `bodyBatch`: Tamaño del lote para el cuerpo.
- `bdryBatch`: Tamaño del lote para la frontera.
- `lr`: Tasa de aprendizaje.
- `width`: Ancho de la red neuronal.
- `depth`: Profundidad de la red neuronal.
- `numQuad`: Número de puntos de colocación.
- `trainStep`: Número total de pasos de entrenamiento.
- `penalty`: Parámetro de penalización.
- `preStep`: Pasos previos al entrenamiento principal.
- `diff`: Diferencia utilizada en el cálculo del gradiente.

```

1 def main():
2
3     device = torch.device("cuda:0" if torch.cuda.is_available() else "
        cpu")
4
5     params = dict()
6     params["d"] = 2
7     params["dd"] = 1
8     params["bodyBatch"] = 1024

```

---

```

 9     params["bdryBatch"] = 1024
10     params["lr"] = 0.0001
11     params["width"] = 64
12     params["depth"] = 4
13     params["numQuad"] = 40000
14     params["trainStep"] = int(1e4)
15     params["penalty"] = 500
16     params["preStep"] = 0
17     params["diff"] = 0.001
18     params["writeStep"] = 500
19     params["sampleStep"] = 10
20     params["step_size"] = 5000
21     params["gamma"] = 0.3
22     params["decay"] = 0.00001
23
24     startTime = time.time()
25     model = RitzNet(params).to(device)
26     print("Generating network costs %s seconds."%(time.time()-startTime))
27
28     preOptimizer = torch.optim.Adam(model.parameters(),lr=params["preLr"])
29     optimizer = torch.optim.Adam(model.parameters(),lr=params["lr"],
30     weight_decay=params["decay"])
31     scheduler = StepLR(optimizer,step_size=params["step_size"],gamma=
32     params["gamma"])
33
34     startTime = time.time()
35     loss_history = train(model,device,params,optimizer,scheduler)
36     print("Training costs %s seconds."%(time.time()-startTime))
37
38     model.eval()
39     testError = test(model,device,params)
40     print("The test error (of the last model) is %s."%testError)
41     print("The number of parameters is %s,"%count_parameters(model))
42
43     torch.save(model.state_dict(), "last_model.pt")
44
45     plot_loss(loss_history)
46
47     pltResult(model, device, 500)
48
49     valn = femPoissonSquare(50)
50
51     compare_fem_model(50, model, valn)

```