Universidad Autónoma de Chiapas

Facultad de Ciencias en Física y Matemáticas

Estimación de la temperatura con la ecuación del Bio-Calor usando DeepONet

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN FÍSICA

PRESENTA:

FRANCISCO DAMIÁN ESCOBAR CANDELARIA X200032

Director:

Dr. Yofre Hernán García Gómez

Tuxtla Gutiérrez, Chiapas; Octubre de 2025

Dedicatoria

A mis padres y hermanos,
por su apoyo incondicional y absoluta fe en mí. Esta tesis es mi manera de agradecerles su amor, paciencia y sacrificio.

Agradecimientos

En primer lugar, deseo agradecer de todo corazón a mi director de tesis, el Dr. Yofre Hernán García Gómez. Gracias por su apoyo y fe, no solo en el presente trabajo sino también en mí. El tiempo que ha invertido en mi persona ha sido un regalo de lo más valioso; gracias a su guía y mentoría he logrado adquirir conocimientos que nunca habría imaginado alcanzar por mi cuenta. Esta tesis es el fruto directo de su apoyo como tutor y de la colaboración entre la escuela de matemáticas y la de física, por lo que estoy profundamente agradecido.

Asimismo, quiero expresar mi profundo agradecimiento a mis estimados profesores y profesoras, pues todos y cada uno de ellos aportaron a mi formación académica con pasión y dedicación genuinas, a veces yendo más allá del deber para explicar un tema o un ejercicio con tal de transmitir su vasto conocimiento. Además, quiero agradecer al personal administrativo y de la biblioteca, quienes, además de desempeñar con excelencia su labor, me brindaron su amistad; eso fue un maravilloso regalo que atesoraré en mi corazón.

Adicionalmente, mi gratitud se extiende a mis maravillosos y amados amigos Wilson, Noé, Tass, Wicho, Kevin, Yahir, Lalo y Javier. Gracias por su apoyo incondicional ante todas las adversidades y problemas; su presencia fue fundamental en esta etapa de mi vida. Agradezco cada risa, momento, conversación y hasta discusión compartida con ustedes. Todos y cada uno me inspiraron, alentaron y enseñaron a crecer no solo como estudiante, sino como persona; me mostraron su determinación y voluntad para no rendirse, pero también que está bien tomar un descanso y compartir aquello que nos aqueja con nuestros seres queridos. Son verdaderamente los mejores amigos que pude desear y su contribución a este logro es imposible de omitir.

Por último, pero no menos importante, expreso mi profundo agradecimiento a mis padres, Zoila y Eczar; y a mis hermanos, Eczar y María. Su amor y apoyo incondicionales me han llevado hasta aquí, a una meta que creí tan lejos y que hoy veo realizada. Son incontables los sacrificios que han realizado en pro de mi bienestar, tanto físico como mental, cargando con pesos que en cierto momento me resultaban imposibles de sobrellevar. No tengo palabras para describir la gratitud que siento hacia ustedes; siempre estaré orgulloso de ser llamado su hijo y su hermano. Los amo con todo mi corazón.

A todos ustedes, mi más sincero agradecimiento. Este logro no habría sido posible sin su apoyo, generosidad y aliento constante.

Tabla de contenidos

Resumen

Asesor: Dr. Yofre Hernán García Gómez

Este trabajo explora el uso de DeepONet para resolver ecuaciones diferenciales parciales (EDPs), aplicándola a la estimación de temperatura en tejidos biológicos mediante la ecuación del bio-calor, en contextos clínicos como la hipertermia oncológica.

Se comparó el desempeño de DeepONet frente a un método numérico clásico, como Crank-Nicolson, evaluando precisión mediante métricas de error. La red neuronal fue entrenada para resolver la EDP del Bio-Calor simplificandola de manera que no se tuviera en cuenta la fuente metabólica de calor Q.

Los resultados muestran que DeepONet puede aproximar la solución eficazmente en distintos tiempos, con ventajas de generalización respecto a redes PINN convencionales, posicionándose como una herramienta prometedora en el modelado térmico biomédico.

1. Introducción

El uso de redes neuronales en la resolución de ecuaciones diferenciales parciales (EDPs) ha ganado relevancia en la última década gracias al desarrollo de técnicas que integran principios físicos en el entrenamiento de modelos. Este enfoque, conocido como redes neuronales informadas por la física (PINNs), ha demostrado ser especialmente útil en situaciones donde la disponibilidad de datos es limitada y donde las leyes físicas subyacentes pueden ser incorporadas como restricciones en la función de pérdida (George Em Karniadakis 2021). En este trabajo se explora una variante más reciente: DeepONet, una arquitectura diseñada para aprender operadores funcionales, y su aplicación en la estimación de temperatura en tejidos biológicos mediante la ecuación del Bio-Calor(Lu, Meng, et al. 2021).

La ecuación del Bio-Calor fue propuesta por Pennes en 1948 con el objetivo de modelar la transferencia de calor en tejidos vivos, considerando los efectos de conducción térmica, metabolismo y perfusión sanguínea (Pennes 1948). Este modelo ha sido ampliamente utilizado en aplicaciones clínicas como la hipertermia terapéutica, una técnica que consiste en elevar localmente la temperatura del tejido para mejorar la eficacia de tratamientos oncológicos (Instituto Nacional del Cáncer 2021). Sin embargo, debido a la complejidad de las condiciones fisiológicas y a las propiedades variables de los tejidos, su resolución analítica es inviable, y las aproximaciones numéricas, como el método de Crank-Nicolson, se vuelven indispensables.

En este contexto, surge la oportunidad de aplicar DeepONet como una alternativa innovadora. A diferencia de una PINN tradicional, que se entrena para resolver una instancia específica de una EDP, DeepONet aproxima un operador que puede generalizar a nuevas condiciones de frontera o iniciales sin requerir reentrenamiento (Lu, Meng, et al. 2021). Esta característica resulta de gran valor en aplicaciones médicas donde las condiciones pueden variar entre pacientes o incluso durante un mismo procedimiento. Además, el modelo puede ser entrenado sobre una base de soluciones simuladas, lo que reduce la necesidad de datos experimentales, difíciles y costosos de obtener en contextos clínicos (George Em Karniadakis 2021).

El presente trabajo tiene como objetivo comparar la precisión y eficiencia de DeepONet con el método numérico clásico de Crank-Nicolson en la estimación de temperatura sobre un dominio bidimensional. Para ello, se implementó un modelo basado en la versión adimensionalizada de la ecuación del Bio-Calor, y se utilizó la biblioteca DeepXDE para su entrenamiento (Lu, Meng, et al. 2021). Posteriormente, se evaluaron métricas como el error medio absoluto (MAE) y el error máximo absoluto (MaxAE), y se analizaron

las predicciones visualmente frente a referencias obtenidas por Alessio Borgi (2023), con resultados prometedores.

La combinación de eficiencia, capacidad de generalización y adecuación a condiciones reales posiciona a DeepONet como una alternativa poderosa frente a métodos clásicos. Esta tesis busca sentar las bases para diversificar su uso en escenarios variados e incentivar su uso en la solución de operadores. Así, el trabajo contribuye a la creciente tendencia de aplicar inteligencia artificial en el ámbito médico con fundamentos sólidos en física matemática.

2. Objetivos

2.1. Objetivo general

Comparar la aproximación numérica de la solución de una EDP obtenida del uso de una red neuronal con arquitectura DeepONet, con aproximaciones numéricas obtenidas de métodos numéricos clásicos, en el contexto de un estudio médico basado en la EDP del Bio-Calor, utilizando métricas de error relevantes para evaluar su desempeño.

2.2. Objetivos específicos

1. Comprender y adaptar el uso de las PINNs para la resolución de PDEs y ODEs, así como sus aplicaciones multidisciplinarias.
2. Explorar el uso de DeepONet como alternativa a las PINNs clásicas y determinar tanto ventajas como desventajas de su implementación.
3. Evaluar y contrastar la eficacia de la arquitectura de red neuronal artificial DeepONet con un método numérico de referencia, como el método de Crank Nickolson.
4. Enmarcar las ventajas/desventajas al implementar un modelo de red neuronal en un lenguaje de programación como Python.

Part I.

Preliminares

3. Ecuaciones diferenciales parciales

Las EDPs, al igual que las ecuaciones diferenciales ordinarias (EDOs), se clasifican en lineales y no lineales. De forma análoga a una EDO lineal, la variable dependiente y sus derivadas parciales en una EDP lineal se elevan únicamente a la primera potencia (Zill y Cullen 2008).

3.1. Ecuación diferencial parcial lineal

Si dejamos que u denote la variable dependiente y que x e y representen las variables independientes, entonces la forma general de una **ecuación diferencial parcial lineal de segundo orden** está dada por:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x\,\partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu = G, \tag{3.1}$$

donde los coeficientes $A, B, C, \ldots, G$ son funciones de x e y. Cuando $G(x, y) = 0$, la Ecuación 3.1 se denomina **homogénea**; de lo contrario, es **no homogénea**. Por ejemplo, las ecuaciones lineales:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{y} \quad \frac{\partial^2 u}{\partial x^2} - \frac{\partial u}{\partial y} = xy$$

son homogénea y no homogénea, respectivamente.

3.2. Solución de una EDP

Una **solución** de una ecuación diferencial parcial es una función $u(x, y)$ de dos variables independientes que posee todas las derivadas parciales que aparecen en la ecuación y que satisface dicha ecuación en alguna región del plano xy.

No es lo habitual examinar los procedimientos para encontrar **soluciones generales** de ecuaciones diferenciales parciales lineales. No solo porque suele ser difícil obtener una solución general de una EDP lineal de segundo orden, sino que una solución general no es tan útil en aplicaciones prácticas. Por lo tanto, el enfoque común es el de encontrar

soluciones particulares de las EDPs lineales más importantes, sin olvidar que a cada solución particular le pertenecen un conjunto de condiciones iniciales y de frontera.

3.3. Separación de variables

Dentro del banco de métodos para encontrar soluciones particulares de una EDP lineal, uno de los más comunes se llama **método de separación de variables**. En este método buscamos una solución particular de la forma de un producto de una función de x y una función de y:

$$u(x, y) = X(x)Y(y).$$

Bajo ciertas condiciones, esta suposición permite reducir una EDP lineal en dos variables a dos ecuaciones diferenciales ordinarias (ODEs). Para este fin, observamos que:

$$\frac{\partial u}{\partial x} = X'Y, \quad \frac{\partial u}{\partial y} = XY', \quad \frac{\partial^2 u}{\partial x^2} = X''Y, \quad \frac{\partial^2 u}{\partial y^2} = XY'',$$

donde las comillas denotan derivación ordinaria.

3.4. Principio de superposición

Teorema 3.1. *Si $u_1, u_2, \dots, u_k$ son soluciones de una ecuación diferencial parcial lineal homogénea, entonces la combinación lineal*

$$u = c_1u_1 + c_2u_2 + \cdots + c_ku_k$$

donde las $c_i, i = 1, 2, \dots, k$ son constantes. Es también una solución.

El teorema 3.1 se puede entender como: *siempre que tengamos un conjunto infinito de soluciones $u_1, u_2, u_3, \dots$ de una ecuación lineal homogénea, podemos construir otra solución u mediante la serie infinita*:

$$u = \sum_{k=1}^{\infty} c_k u_k,$$

donde las constantes c_i, con $i = 1, 2, \dots$, son coeficientes.

3.5. Clasificación de ecuaciones

Una ecuación diferencial parcial lineal de segundo orden con dos variables independientes y coeficientes constantes puede clasificarse en uno de tres tipos. Esta clasificación depende únicamente de los coeficientes de las derivadas de segundo orden. Por supuesto, asumimos que al menos uno de los coeficientes A, B o C es distinto de cero.

Definición 3.1. La ecuación diferencial parcial lineal de segundo orden

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x\,\partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu = 0,$$

donde A, B, C, D, F son constantes reales, se dice que es:

- **Hiperbólica** si $B^2 - 4AC > 0$,
- **Parabólica** si $B^2 - 4AC = 0$,
- **Elíptica** si $B^2 - 4AC < 0$.

4. Problemas de valores en la frontera

Si, por ejemplo, $u(x,t)$ es una solución de una EDP, donde x representa una dimensión espacial y t representa el tiempo, entonces es posible prescribir el valor de u, o $\frac{\partial u}{\partial x}$, o una combinación lineal de u y $\frac{\partial u}{\partial x}$ en un valor x especificado, así como prescribir u y $\frac{\partial u}{\partial t}$ en un instante dado t (normalmente, $t = 0$). En otras palabras, un *problema de valores en la frontera* puede consistir en una EDP, junto con *condiciones de frontera* y *condiciones iniciales* (Zill y Cullen 2008).

4.1. Ecuaciones clásicas

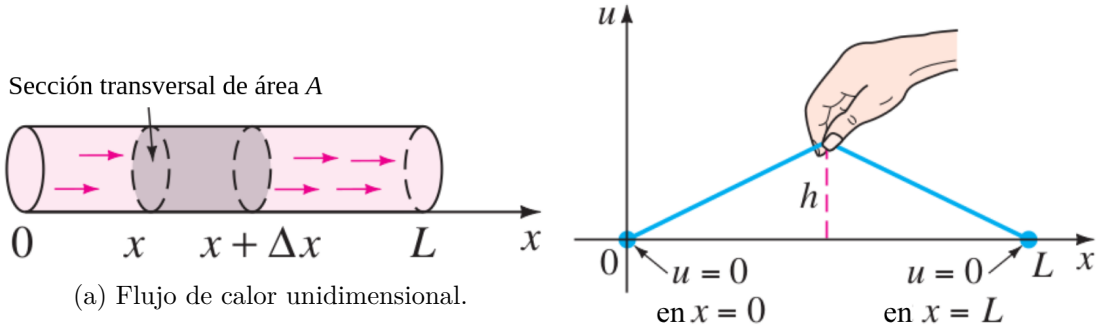
Aplicar el método de separación de variables para encontrar soluciones en forma de producto es muy común con las siguientes *ecuaciones clásicas* de la física matemática:

$$k\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}, \quad k > 0 \tag{4.1}$$

$$\alpha^2\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2} \tag{4.2}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{4.3}$$

o variantes ligeras de estas ecuaciones. Las EDPs 4.1, 4.2 y 4.3 se conocen, respectivamente, como la *ecuación del calor unidimensional*, la *ecuación de onda unidimensional* y la *forma bidimensional de la ecuación de Laplace*. El término "unidimensional" en el caso de las ecuaciones 4.1 y 4.2 se refiere al hecho de que x denota una variable espacial, mientras que t representa el tiempo; "bidimensional" en 4.3 significa que tanto x como y son variables espaciales. Si se compara 4.1-4.3 con la forma lineal en la Definición 3.1 (donde t juega el papel del símbolo y), se observa que la ecuación del calor 4.1 es parabólica, la ecuación de onda 4.2 es hiperbólica y la ecuación de Laplace 4.3 es elíptica.



(a) Flujo de calor unidimensional.

(b) Cuerda tensada.

Figura 4.1.: Aplicaciones de las ecuaciones 4.1 y 4.2 (Zill y Cullen 2008).

4.2. Condiciones iniciales

Dado que las soluciones de las ecuaciones 4.1 y 4.2 dependen del tiempo t, es posible especificar lo que ocurre en $t = 0$; es decir, establecer **condiciones iniciales (CI)**. Si $f(x)$ representa la distribución inicial de temperatura en la varilla mostrada en la Figura 4.1a, entonces una solución $u(x,t)$ de 4.1 debe satisfacer la condición inicial única $u(x,0) = f(x), \quad 0 < x < L.$

Por otro lado, para una cuerda vibrante podemos especificar tanto su desplazamiento inicial (o forma) $f(x)$ como su velocidad inicial $g(x)$. En términos matemáticos, buscamos una función $u(x,t)$ que satisfaga 4.2 y las dos condiciones iniciales:

$$u(x,0) = f(x), \quad \left.\frac{\partial u}{\partial t}\right|_{t=0} = g(x), \quad 0 < x < L. \tag{4.4}$$

Por ejemplo, la cuerda podría ser tensada, como se muestra en la Figura 4.1b, o liberada desde el reposo ($g(x) = 0$).

4.3. Condiciones de frontera

La cuerda en la Figura 4.1b está fija al eje x en $x = 0$ y $x = L$ para todos los tiempos. Ésto se interpreta a través de dos **condiciones de frontera (CF):**

$$u(0,t) = 0, \quad u(L,t) = 0, \quad t > 0.$$

En éste contexto la función f en la Ec 4.4 es continua y, en consecuencia, $f(0) = 0$ y $f(L) = 0$. En general, existen tres tipos de condiciones de frontera asociadas con las ecuaciones 4.1, 4.2 y 4.3. En la frontera es posible especificar los valores de *una* de las siguientes:

$$\text{(i)} \quad u, \qquad \text{(ii)} \quad \frac{\partial u}{\partial n}, \qquad \text{o} \qquad \text{(iii)} \quad \frac{\partial u}{\partial n} + hu, \quad \text{con } h \text{ constante.}$$

Aquí $\frac{\partial u}{\partial n}$ denota la derivada normal de u (la derivada de u en dirección perpendicular a la frontera). Una condición de frontera del primer tipo (i) es llamada **condición de Dirichlet**; una condición de frontera del segundo tipo (ii) es llamada **condición de Neumann**; y una condición de frontera del tercer tipo (iii) es conocida como **condición de Robin**. Por ejemplo, para $t > 0$ una condición típica al extremo derecho de la varilla de la Figura 4.1a puede ser:

$$\text{(i)}' \quad u(L,t) = u_0, \text{ con } u_0 \text{ constante}$$

$$\text{(ii)}' \quad \left.\frac{\partial u}{\partial x}\right|_{x=L} = 0$$

$$\text{(iii)}' \quad \left.\frac{\partial u}{\partial x}\right|_{x=L} = -h(u(L,t) - u_m), \text{ con } h > 0 \text{ y } u_m \text{ constantes}$$

La condición (i)' simplemente establece que el límite $x = L$ se mantiene, por algún medio, a una temperatura constante u_0 durante todo el tiempo $t > 0$. La condición (ii)' indica que el contorno $x = L$ está *aislado*. Según la ley empírica de la transferencia de calor, el flujo de calor a través del borde (es decir, la cantidad de calor por unidad de área por unidad de tiempo conducida a través la frontera) es proporcional al valor de la derivada normal $\frac{\partial u}{\partial n}$ de la temperatura u. Por lo tanto, cuando el límite $x = L$ está aislado térmicamente, no fluye calor hacia dentro ni hacia fuera de la varilla, por lo que

$$\left.\frac{\partial u}{\partial x}\right|_{x=L} = 0.$$

Es posible interpretar (iii)' como que el calor se pierde del extremo derecho de la varilla al estar en contacto con un medio, como el aire o el agua, que se mantiene a temperatura constante. Según la ley de enfriamiento de Newton, el flujo de calor hacia afuera de la varilla es proporcional a la diferencia entre la temperatura $u(L,t)$ en la frontera y la temperatura u_m del medio circundante. Se observa que si se pierde calor por el extremo izquierdo de la varilla, la condición de contorno es

$$\left.\frac{\partial u}{\partial x}\right|_{x=0} = h(u(0,t) - u_m).$$

El cambio de signo respecto de (iii)' corresponde con el supuesto de que la varilla está a una temperatura más alta que el medio que rodea los extremos, de modo que $u(0,t) > u_m$

y $u(L,t) > u_m$. Para $x = 0$ y $x = L$, las pendientes $u_x(0,t)$ y $u_x(L,t)$ deben ser positivas y negativas, respectivamente.

Por supuesto, en los extremos de la varilla se pueden especificar diferentes condiciones al mismo tiempo. Por ejemplo, podríamos tener

$$\left.\frac{\partial u}{\partial x}\right|_{x=0} = 0 \quad \text{y} \qquad u(L,t) = u_0, \quad t > 0.$$

5. Problemas de valor inicial

Las ecuaciones diferenciales son utilizadas para modelar problemas en ciencia e ingeniería que implican el cambio de una variable con respecto a otra. La mayoría de estos problemas requieren la solución de un problema de valor inicial, es decir, la solución de una ecuación diferencial que satisface una condición inicial dada.

En situaciones reales comunes, la ecuación diferencial que modela el problema es demasiado compleja para resolverse con exactitud, y se adopta uno de dos enfoques para aproximar la solución. El primer enfoque consiste en modificar el problema simplificando la ecuación diferencial a una que pueda resolverse con exactitud y luego utilizar la solución de la ecuación simplificada para aproximar la solución del problema original. El otro enfoque utiliza métodos para aproximar la solución del problema original. Este es el enfoque más común porque los métodos de aproximación proporcionan resultados más precisos e información de error realista (Burden y Faires 2010).

Ejemplo

El movimiento de un péndulo oscilante bajo ciertas suposiciones se describe mediante la ecuación diferencial de segundo orden:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin\theta = 0,$$

donde L es la longitud del péndulo, $g \approx 9.81\frac{m}{s^2}$ es la constante gravitacional terrestre y θ es el ángulo que forma el péndulo con la vertical. Si, además, especificamos la posición del péndulo al inicio del movimiento, $\theta(t_0) = \theta_0$, y su velocidad en ese punto, $\theta'(t_0) = \theta_0'$. Tenemos un *problema de valor inicial.*

Para dar una idea más clara acerca de los problemas de valor inicial Burden y Faires (2010) brinda las siguientes definiciones y teoremas:

Definición 5.1. Se dice que una función $f(t, y)$ satisface una **Condición de Lipschitz** en la variable y en un conjunto $D \subset \mathbb{R}^2$ si existe una constante $L > 0$ tal que

$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|$$

donde $f(t, y_1)$ y $f(t, y_2)$ están en D. La constante L es llamada **constante de Lipschitz** para f.

Definición 5.2. Se dice que un conjunto $D \subset \mathbb{R}^2$ es **convexo** si para cualesquiera $f(t, y_1), f(t, y_2) \in D$, entonces $((1-\lambda)t_1 + \lambda t_2, (1-\lambda)y_1 + \lambda y_2)$ también pertenece a D para cada $\lambda \in [0, 1]$.

En términos geométricos, la Definición 5.2 establece que un conjunto es convexo siempre que, para cualesquiera dos puntos dentro del conjunto, todo el segmento recto entre ellos también pertenezca al conjunto Figura 5.1.

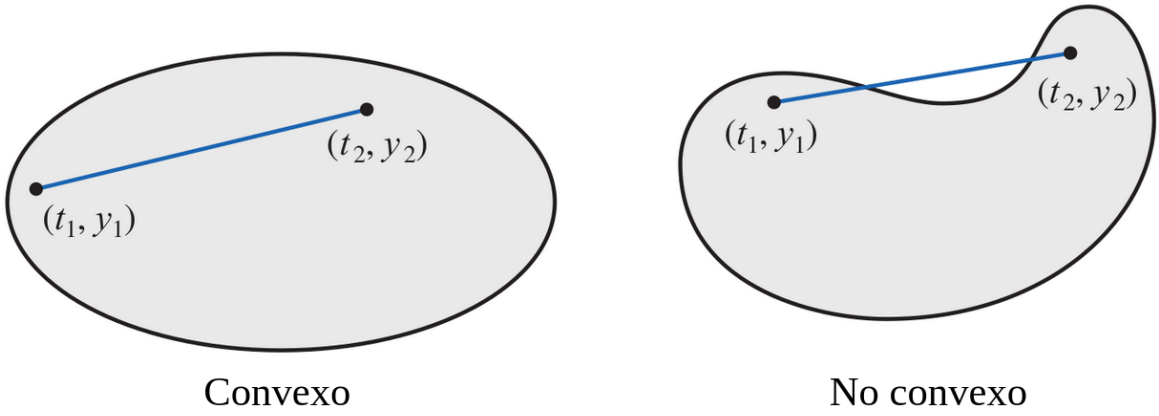


Figura 5.1.: Ejemplo geométrico de un conjunto *convexo* y *no convexo* (Burden y Faires 2010).

Teorema 5.1. *Supongamos que $f(t, y)$ está definida en un conjunto convexo $D \in \mathbb{R}^2$. Si existe una constante $L > 0$ con*

$$\left|\frac{\partial f}{\partial y}(t, y)\right| \leq L, \quad \textit{para todo} \quad (t, y) \in D, \tag{5.1}$$

entonces f satisface una condición de Lipschitz en D en la variable y con una constante de Lipschitz L.

Como se mostrará en el siguiente teorema, suele ser de gran interés determinar si la función involucrada en un problema de valor inicial satisface una condición de Lipschitz en su segunda variable, y la condición 5.1 suele ser más fácil de aplicar que la definición. Cabe destacar, sin embargo, que el Teorema 5.1 solo proporciona condiciones suficientes para que se cumpla una condición de Lipschitz.

Teorema 5.2. *Supóngase que $D = \{(t, y) | \ a \leq t \leq b, \ -\infty < y < \infty\}$ y que $f(t, y)$ es continua en D. Si f satisface una condición de Lipschitz en D en la variable y, entonces el problema del valor inicial*

$$y'(t) = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha,$$

tiene una solución única $y(t)$ para $a \leq t \leq b$.

Ejemplo

Use el Teorema 5.2 para mostrar que hay una única solución al problema de valor inicial:

$$y'(t) = 1 + t\sin(ty), \quad 0 \leq t \leq 2, \quad y(0) = 0.$$

Solución: Manteniendo a t constante y usando el *Teorema de valor medio* a la función

$$f(t, y) = 1 + t\sin(ty),$$

notamos que cuando $y_1 < y_2$, un número ξ existe en (y_1, y_2) tal que:

$$\frac{f(t, y_2) - f(t, y_1)}{y_2 - y_1} = \frac{\partial}{\partial y} f(t, \xi) = t^2 \cos(\xi t).$$

De este modo:

$$|f(t, y_2) - f(t, y_1)| = |y_2 - y_1||t^2 \cos(\xi t)| \leq 4|y_2 - y_1|,$$

y f satisface una condición de Lipschitz en la variable y con constante de Lipschitz $L = 4$. Además, $f(t, y)$ es continua cuando $0 \leq t \leq 2$ y $-\infty < y < \infty$, por lo que el Teorema 5.2 implica que existe una solución única para este problema de valor inicial.

5.1. Problemas bien planteados

Ahora que hemos abordado, hasta cierto punto, la cuestión de cuándo los problemas de valor inicial tienen soluciones únicas, podemos pasar a la segunda consideración importante: cuándo aproximar la solución de un problema de valor inicial. Los problemas de valor inicial obtenidos mediante la observación de fenómenos físicos generalmente solo se aproximan a la situación real, por lo que necesitamos saber si pequeños cambios en el planteamiento del problema introducen cambios correspondientemente pequeños en la solución (Burden y Faires 2010).

A continuación se presentan otras definiciones así como teoremas que brindarán un conocimiento más sólido acerca de los problemas bien planteados. Se usará como referencia a Burden y Faires (2010).

Definición 5.3. Se dice que el problema de valor inicial

$$\frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha, \tag{5.2}$$

es un **problema bien planteado** si:

- Existe una única solución $y(t)$ para el problema, y
- Existen constantes $\varepsilon_0 > 0$ y $k > 0$ tales que para cualquier ε, con $\varepsilon_0 > \varepsilon > 0$, siempre que $\delta(t)$ sea continua con $|\delta(t)| < \varepsilon$ para todo t en $[a, b]$, y cuando $|\delta_0| < \varepsilon$, el problema del valor inicial

$$\frac{dz}{dt} = f(t, z) + \delta(t), \quad a \leq t \leq b, \quad z(a) = \alpha + \delta_0 \tag{5.3}$$

tenga una única solución $z(t)$ que satisface:

$$|z(t) - y(t)| < k\varepsilon \quad \forall t \in [a, b],$$

donde k es conocida como *constante de estabilidad.*

El problema especificado por la Ecuación 5.3 se denomina **problema perturbado** asociado al problema original Ecuación 5.2. Se asume la posibilidad de que se introduzca un error en el planteamiento de la ecuación diferencial, así como la presencia de un error δ_0 en la condición inicial.

Los métodos numéricos siempre se centrarán en la solución de un problema perturbado, ya que cualquier error de redondeo introducido en la representación perturba el problema original. A menos que el problema original esté bien planteado, hay pocas razones para esperar que la solución numérica de un problema perturbado se aproxime con precisión a la solución del problema original.

Teorema 5.3. *Supongamos* $D = \{(t, y) | \ a \leq t \leq b, \ -\infty < y < \infty\}$. *Si* f *es continua y satisface una condición de Lipschitz en la variable* y *en el conjunto* D, *entonces el problema de valor inicial*

$$\frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha$$

es bien planteado.

Ejemplo

Demostrar que el problema de valor inicial

$$\frac{dy}{dt} = y - t^2 + 1, \quad 0 \leq t \leq 2, \quad y(0) = 0.5,$$

está bien planteado en el dominio $D = \{(t, y) \mid \ 0 \leq t \leq 2 \quad \text{y} \quad -\infty < y < \infty\}$.
Solución: Dado que

$$\left|\frac{\partial(y - t^2 + 1)}{\partial y}\right| = |1| = 1, \tag{5.4}$$

el Teorema 5.1 implica que la función $f(t, y) = y - t^2 + 1$ satisface una condición

de Lipschitz en y sobre D con constante de Lipschitz igual a 1. Además, como f es continua en D, el Teorema 5.3 garantiza que el problema está bien planteado.
A modo de ilustración, consideremos ahora la solución del problema perturbado:

$$\frac{dz}{dt} = z - t^2 + 1 + \delta, \quad 0 \leq t \leq 2, \quad z(0) = 0.5 + \delta_0, \tag{5.5}$$

donde δ y δ_0 son constantes pequeñas, las soluciones respectivas de las ecuaciones 5.4 y 5.5 son:

$$y(t) = (t+1)^2 - 0.5e^t$$

$$z(t) = (t+1)^2 + (\delta + \delta_0 - 0.5)e^t - \delta$$

Sea ε un número positivo. Si $|\delta| < \varepsilon$ y $|\delta_0| < \varepsilon$, entonces

$$|y(t) - z(t)| = |(\delta + \delta_0)e^t - \delta| \leq |\delta + \delta_0|e^2 + |\delta| \leq (2e^2 + 1)\varepsilon,$$

para todo t. Esta desigualdad demuestra que 5.4 está bien planteado, con una constante de estabilidad $k = 2e^2 + 1$ para cualquier $\varepsilon > 0$.

6. Método de Crank Nicolson

El algoritmo introducido por **J. Crank** y **P. Nicolson** (*Crank-Nicolson*) en 1947 representa un esquema numérico ampliamente utilizado para resolver ecuaciones diferenciales parciales de tipo parabólico, como la ecuación de calor. Para comprender su fundamento, resulta ilustrativo considerar el problema de determinar la evolución temporal de la temperatura en una varilla metálica. Dado que la temperatura en cada punto varía de manera continua, es necesario discretizar el problema para su tratamiento computacional, lo que implica representar la varilla mediante un conjunto discreto de puntos y el tiempo mediante una secuencia de pasos finitos (Zill y Cullen 2008).

En este marco, las estrategias de solución numérica se clasifican principalmente en dos categorías. Por un lado, los métodos explícitos calculan el estado futuro del sistema a partir exclusivamente de información del estado presente, lo que los hace conceptualmente sencillos y computacionalmente eficientes por paso de tiempo. Sin embargo, presentan una limitación significativa: su estabilidad depende críticamente del tamaño del paso temporal. Si este paso excede un umbral crítico, la solución numérica puede volverse inestable, manifestando oscilaciones no físicas que divergen hacia infinito (Burden y Faires 2010).

Por otro lado, los métodos implícitos superan esta restricción de estabilidad al establecer una dependencia entre el estado futuro de un punto y el de sus vecinos en el mismo instante futuro. Esta característica garantiza estabilidad incondicional para una gama más amplia de parámetros, pero conlleva una mayor complejidad computacional, ya que requiere resolver un sistema de ecuaciones acoplado en cada paso de tiempo (Burden y Faires 2010).

El método de Crank-Nicolson surge como un esquema híbrido que sintetiza las ventajas de ambos enfoques. Se fundamenta en promediar la discretización espacial de la ecuación diferencial entre el instante de tiempo actual (n) y el futuro ($n+1$). Esta estrategia de promediado le confiere dos propiedades clave (Zill y Cullen 2008):

1. **Estabilidad Incondicional:** A diferencia de los métodos explícitos, el esquema de Crank-Nicolson permanece estable para cualquier tamaño de paso temporal, lo que permite simulaciones más rápidas sin riesgo de divergencia.
2. **Precisión de Segundo Orden:** Al ser un método de segundo orden en tiempo, el error de truncamiento local se reduce más ráticamente al disminuir el paso temporal, lo que se traduce en una mayor precisión global de la solución numérica comparado con métodos de primer orden.

En esencia, el algoritmo consiste en sustituir la segunda derivada parcial en $c\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}$ por el promedio de dos cocientes de diferencias centrales, uno evaluado en t y el otro en $t+k$:

$$\begin{aligned} &\frac{c}{2}\left[\frac{u(x+h,t)-2u(x,t)+u(x-h,t)}{h^2}\right]+ \\ &+\frac{c}{2}\left[\frac{u(x+h,t+k)-2u(x,t+k)+u(x-h,t+k)}{h^2}\right] \\ &\quad=\frac{1}{k}[u(x,t+k)-u(x,t)] \end{aligned} \tag{6.1}$$

si se define $\lambda = \frac{ck}{h^2}$ y

$$\begin{aligned} u(x+h,t)=u_{i+1,j}, \quad u(x,t)=u_{ij}, \quad u(x-h,t)=u_{i-1,j}, \\ u(x+h,t+k)=u_{i+1,j+1}, \quad u(x,t+k)=u_{i,j+1}, \quad u(x-h,t+k)=u_{i-1,j+1}, \end{aligned}$$

es posible reescribir a la Ec. 6.1 como:

$$-u_{i-1,j+1}+\alpha u_{i,j+1}-u_{i+1,j+1}=u_{i+1,j}-\beta u_{ij}+u_{i-1,j}, \tag{6.2}$$

donde $\alpha = 2(1+\frac{1}{\lambda})$, $\beta = 2(1-\frac{1}{\lambda})$, $j = 0,1,\ldots,m-1$, $i = 0,1,\ldots,n-1$.

Para cada elección de j, la ecuación diferencial Ec. 6.2 para $i = 0,1,\ldots,n-1$ da $n-1$ ecuaciones en $n-1$ incógnitas $u_{i,j+1}$. Debido a las condiciones de contorno preestablecidas, los valores de $u_{i,j+1}$ se conocen para $i = 0$ y para $i = n$. Por ejemplo, en el caso $n = 4$, el sistema de ecuaciones para determinar los valores aproximados de u en la línea de tiempo $(j+1)$ es:

$$\begin{aligned} -u_{0,j+1}+\alpha u_{1,j+1}-u_{2,j+1} &= u_{2,j}-\beta u_{1,j}+u_{0,j} \\ -u_{1,j+1}+\alpha u_{2,j+1}-u_{3,j+1} &= u_{3,j}-\beta u_{2,j}+u_{1,j} \\ -u_{2,j+1}+\alpha u_{3,j+1}-u_{4,j+1} &= u_{4,j}-\beta u_{3,j}+u_{2,j} \end{aligned}$$

reordenando se llega a

$$\begin{array}{ccccccc} \alpha u_{1,j+1} & - & u_{2,j+1} & & & = & b_1 \\ -u_{1,j+1} & + & \alpha u_{2,j+1} & - & u_{3,j+1} & = & b_2 \\ & - & u_{2,j+1} & + & \alpha u_{3,j+1} & = & b_3 \end{array} \tag{6.3}$$

donde

$$\begin{aligned} b_1 &= u_{2,j}-\beta u_{1,j}+u_{0,j}+u_{0,j+1}, \\ b_2 &= u_{3,j}-\beta u_{2,j}+u_{1,j}, \\ b_3 &= u_{4,j}-\beta u_{3,j}+u_{2,j}+u_{4,j+1}. \end{aligned}$$

En general, si utilizamos la ecuación diferencial Ec. 6.2 para determinar valores de u en la línea de tiempo $(j-1)$, es necesario resolver un sistema lineal $\mathbf{AX} = \mathbf{B}$, donde la matriz de coeficientes $\mathbf{A}$ es una **matriz tridiagonal**,

$$A = \begin{pmatrix} \alpha & -1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ -1 & \alpha & -1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & -1 & \alpha & -1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -1 & \alpha & -1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & \alpha & -1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & -1 & \alpha \end{pmatrix},$$

y las componentes de la matriz columna $\mathbf{B}$ son

$$\begin{aligned} b_1 &= u_{2,j} - \beta u_{1,j} + u_{0,j} + u_{0,j+1}, \\ b_2 &= u_{3,j} - \beta u_{2,j} + u_{1,j}, \\ b_3 &= u_{4,j} - \beta u_{3,j} + u_{2,j}, \\ &\vdots \\ b_{n-1} &= u_{n,j} - \beta u_{n-1,j} + u_{n-2,j} + u_{n,j+1}. \end{aligned}$$

Part II.

Redes neuronales

7. Physics Informed Neural Networks (PINNs)

Las Physics-Informed Neural Networks (PINNs) son un enfoque innovador que combina redes neuronales con ecuaciones diferenciales gobernantes para resolver problemas complejos de física (Blechschmidt y Ernst 2021). A diferencia de métodos tradicionales, las PINNs incorporan directamente las ecuaciones físicas en su función de pérdida mediante diferenciación automática, lo que permite minimizar simultáneamente el error en los datos y el residual de las PDEs (George Em Karniadakis 2021). Esta característica las hace particularmente valiosas en escenarios con datos limitados, donde el conocimiento físico actúa como un regularizador efectivo. La capacidad de aproximación de las PINNs se fundamenta en el teorema de aproximación universal de las redes neuronales, adaptado para incorporar restricciones físicas a través de términos de penalización en la función de optimización (George Em Karniadakis 2021).

como ejemplo, se considera la **ecuación de Burgers para viscocidad**:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

con una condición inicial adecuada y condiciones de contorno de Dirichlet. En la Figura 7.1, la red izquierda (*physics-uninformed*) representa el sustituto de la solución de EDP $u(x,t)$, mientras que la red derecha (*physics-informed*) describe el residuo de EDP $\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0$. La función de pérdida incluye una pérdida supervisada de las mediciones de datos de u de las condiciones iniciales y de contorno, y una pérdida no supervisada de EDP:

$$\mathcal{L} = w_{\text{data}}\mathcal{L}_{\text{data}} + w_{\text{PDE}}\mathcal{L}_{\text{PDE}} \tag{7.1}$$

donde:

$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}} \sum_{i=1}^{N_{\text{data}}} (u(x_i, t_i) - u_i)^2$$

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_{\text{PDE}}} \sum_{j=1}^{N_{\text{PDE}}} \left(\frac{\partial u}{\partial t}(x_j, t_j) + u \frac{\partial u}{\partial x}(x_j, t_j) - \nu \frac{\partial^2 u}{\partial x^2}(x_j, t_j) \right)^2$$

Aquí, (x_i, t_i) representan puntos donde se conocen valores de la solución y (x_j, t_j) son puntos interiores del dominio. Los pesos w_{data} y w_{PDE} equilibran la contribución de cada término. La red se entrena minimizando $\mathcal{L}$ usando optimizadores como Adam o L-BFGS hasta alcanzar un umbral ε (George Em Karniadakis 2021).

Este enfoque permite resolver EDPs (clásicas, fraccionarias o estocásticas) sin mallas, en dominios complejos o con datos escasos y ruidosos, siendo una herramienta flexible y poderosa para la modelación científica.

7.1. Algoritmos de optimización

Un algoritmo de optimización busca minimizar o maximizar una función objetivo ajustando sus parámetros de manera iterativa. Son esenciales en el entrenamiento de redes neuronales y otros modelos de aprendizaje automático (Kingma y Ba 2014).

7.1.1. ADAM

Adaptive Moment Estimation (*ADAM*) combina estimaciones de primer y segundo momento del gradiente para adaptar las tasas de aprendizaje por parámetro. Utiliza promedios móviles exponenciales de gradientes y gradientes al cuadrado, corregidos por bias, lo que lo hace eficiente en problemas con gradientes ruidosos o dispersos. Es robusto y requiere poco ajuste hiperparamétrico (Kingma y Ba 2014).

7.1.2. L-BFGS

Limited-memory BFGS (*L-BFGS*) es un método quasi-Newton que aproxima la inversa del Hessiano usando un historial limitado de gradientes y actualizaciones de parámetros. Evita el costo computacional de almacenar matrices completas, lo que lo hace viable para problemas de alta dimensionalidad. Es especialmente útil en optimización batch o con gradientes estables (Goldfarb, Ren, y Bahamou 2016).

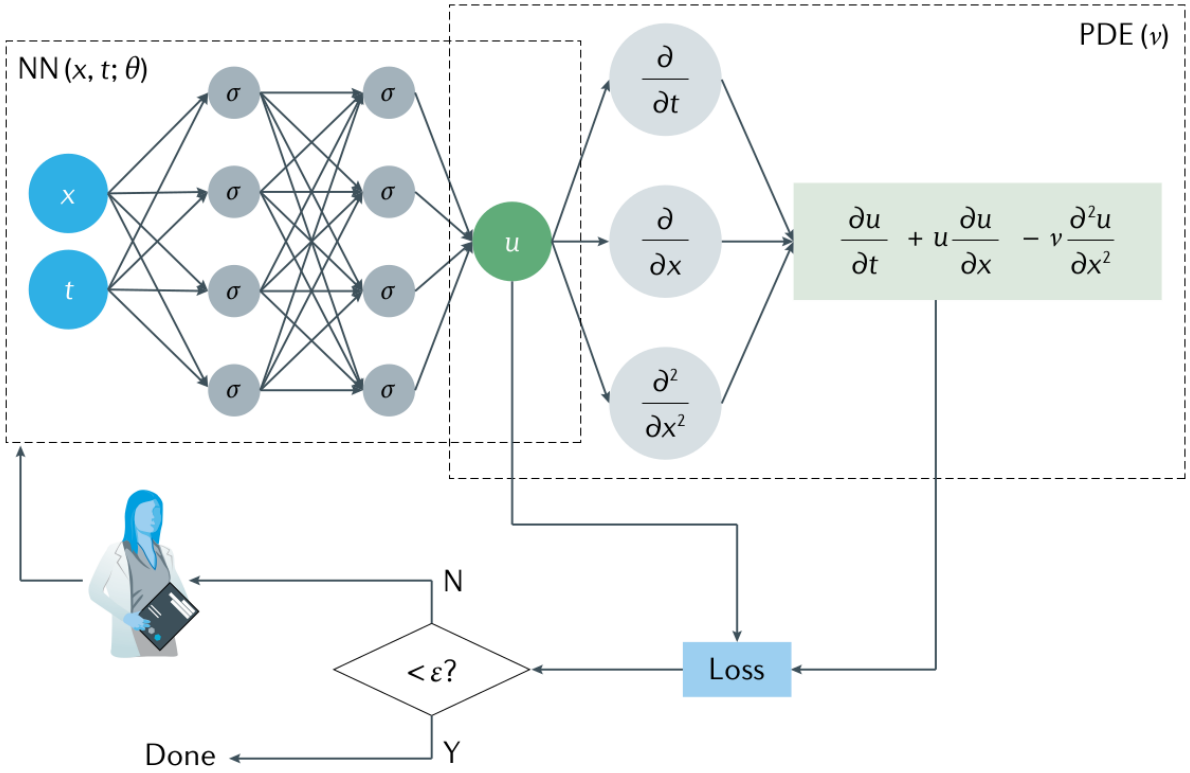


Figura 7.1.: **El algoritmo de una PINN.** Se construye una red neuronal (NN) $u(x, t; \theta)$ donde θ representa el conjunto de pesos entrenables w y sesgos b, y σ representa una función de activación no lineal. Especifique los datos de medición x_i, t_i, u_i para u y los puntos residuales x_j, t_j para la EDP. Se especifica la pérdida $\mathcal{L}$ en la Ecuación 7.1 sumando las pérdidas ponderadas de los datos y la EDP. Entrene la NN para encontrar los mejores parámetros θ^* minimizando la pérdida $\mathcal{L}$ (George Em Karniadakis 2021).

7.2. Deepxde

DeepXDE es una biblioteca en Python de aprendizaje profundo diseñada para resolver ecuaciones diferenciales, incluyendo ecuaciones diferenciales parciales (PDEs), ecuaciones integro-diferenciales (IDEs) y ecuaciones diferenciales estocásticas (SDEs), utilizando redes neuronales informadas por la física (PINNs). Combina técnicas de aprendizaje automático con principios físicos al incorporar las ecuaciones diferenciales directamente en la función de pérdida de la red neuronal, aprovechando la diferenciación automática para calcular derivadas de manera precisa y eficiente (Lu, Meng, et al. 2021).

7.3. Ejemplo de resolución de la ecuación de Burger 1D con deepxde

Dada la ecuación:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2}, \quad x \in [-1, 1],\ t \in [0, 1],$$

con la condición de frontera de Dirichlet y condición inicial:

$$u(-1, t) = u(1, t) = 0, \quad u(x, 0) = -\sin(\pi x).$$

```
import deepxde as dde
import numpy as np

# Definir una función para cargar los datos
def gen_testdata():
    data = np.load("data/Burgers.npz")
    t, x, exact = data["t"], data["x"], data["usol"].T
    xx, tt = np.meshgrid(x, t)
    X = np.vstack((np.ravel(xx), np.ravel(tt))).T
    y = exact.flatten()[:, None]
    return X, y

# Definir la PDE
def pde(x, y):
    dy_x = dde.grad.jacobian(y, x, i=0, j=0)
    dy_t = dde.grad.jacobian(y, x, i=0, j=1)
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)
    return dy_t + y * dy_x - 0.01 / np.pi * dy_xx

# Definir los dominios espacial, temporal y juntarlos
geom = dde.geometry.Interval(-1, 1)
timedomain = dde.geometry.TimeDomain(0, 0.99)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)

# Definir la condición de frontera
bc = dde.icbc.DirichletBC(
    geomtime,
    lambda x: 0,
```

```
    lambda _, on_boundary: on_boundary)

# Definir la condición inicial
ic = dde.icbc.IC(
    geomtime,
    lambda x: -np.sin(np.pi * x[:, 0:1]),
    lambda _, on_initial: on_initial
)

# Definir la cantidad de puntos en el dominio
data = dde.data.TimePDE(
    geomtime, pde, [bc, ic],
    num_domain=2540,
    num_boundary=80,
    num_initial=160,
    num_test=300
)

# Definir la arquitectura de la red, así como
# su función de activación y el inicializador
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh", "Glorot normal")
model = dde.Model(data, net)

# Compilar el modelo y entrenarlo
model.compile("adam", lr=1e-3)
losshistory, train_state = model.train(iterations=3000)
model.compile("L-BFGS")
losshistory, train_state = model.train()
#dde.saveplot(losshistory, train_state, issave=False, isplot=True)
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

X, y_true = gen_testdata()
y_pred = model.predict(X)
f = model.predict(X, operator=pde)

# Extraer componentes de X
x_coords = X[:, 0]  # coordenadas x (espacio)
time = X[:, 1]      # coordenadas t (tiempo)

# Crear figura con dos subgráficos 3D
fig = plt.figure(figsize=(14, 8), constrained_layout=True)
ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122, projection='3d')

# Calcular límites comunes para los ejes z
z_min = min(y_true.min(), y_pred.min())
z_max = max(y_true.max(), y_pred.max())

# Gráfico 1: Valores reales
sc1 = ax1.scatter(x_coords, time, y_true, c=y_true,
                  cmap='viridis', marker='o', vmin=z_min, vmax=z_max)
ax1.set_xlabel('Posición (x)')
ax1.set_ylabel('Tiempo (t)')
ax1.set_zlabel('u(x,t)')
ax1.set_title('Valores reales de u(x,t)')
ax1.set_zlim([z_min, z_max])
ax1.set_box_aspect(None, zoom=0.9)
```

```
# Gráfico 2: Valores predichos
sc2 = ax2.scatter(x_coords, time, y_pred, c=y_pred,
                  cmap='viridis', marker='^', vmin=z_min, vmax=z_max)
ax2.set_xlabel('Posición (x)')
ax2.set_ylabel('Tiempo (t)')
ax2.set_zlabel('u(x,t)')
ax2.set_title('Valores predichos de u(x,t)')
ax2.set_zlim([z_min, z_max])
ax2.set_box_aspect(None, zoom=0.75)

cbar = fig.colorbar(sc1, ax=(ax1,ax2),
                    shrink=0.9, aspect=90,
                    pad=0.1, orientation='horizontal')
cbar.set_label('Magnitud de u(x,t)')

plt.show()

print("Error relativo L2:", dde.metrics.l2_relative_error(y_true, y_pred))
```

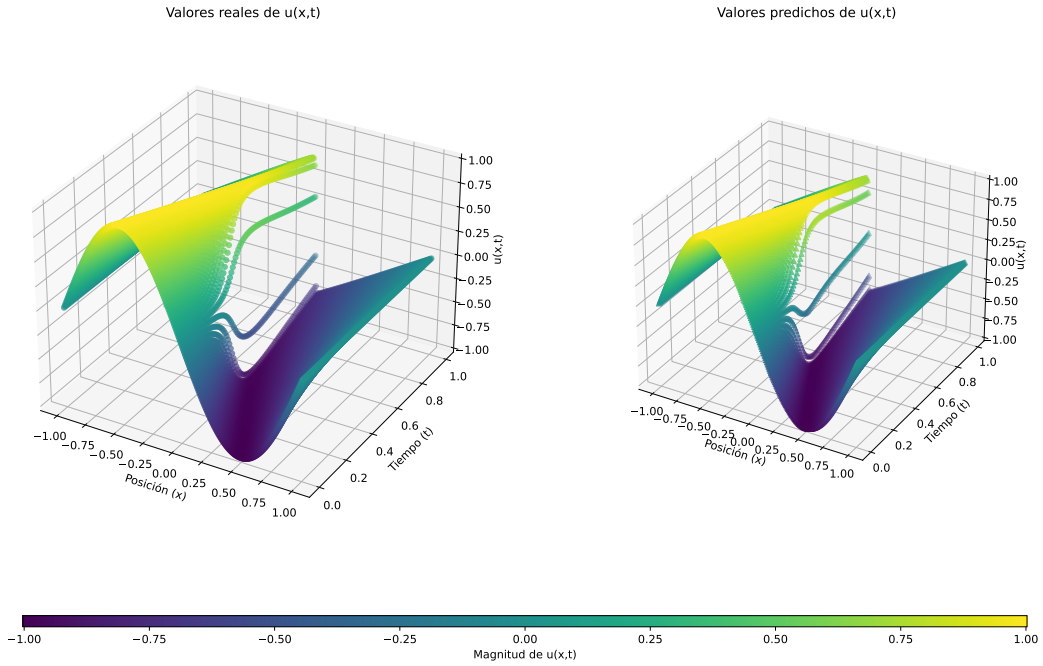


Figura 7.2.: Comparación entre solución real y predicción de la red neuronal para la ecuación de Burger 1D. Debido a su naturaleza unidimensional, es posible plasmar en un eje al tiempo (t) y representar a la función a lo largo de éste como una serie de *fotos* para un instante t dado.

```
Error relativo L2: 0.03250306476857025
```

7.4. Comparación con Redes Neuronales Tradicionales

Mientras que las redes neuronales tradicionales dependen exclusivamente de grandes volúmenes de datos etiquetados para su entrenamiento (George Em Karniadakis 2021), las PINNs integran el conocimiento físico como parte esencial de su arquitectura (Blechschmidt y Ernst 2021). Esta diferencia clave permite a las PINNs generar soluciones físicamente consistentes incluso con datos escasos, evitando el sobreajuste común en enfoques puramente basados en datos. Otra ventaja significativa de las PINNs es su naturaleza mesh-free, que contrasta con los métodos numéricos tradicionales como FEM (*Finite Element Method*) o FDM (*Finite Difference Method*) que requieren discretización espacial. Sin embargo, el entrenamiento de PINNs puede ser más desafiante debido a la necesidad de optimizar múltiples objetivos simultáneamente (ajuste a datos y cumplimiento de leyes físicas) (Blechschmidt y Ernst 2021; George Em Karniadakis 2021).

8. DeepONet

DeepONet (Deep Operator Network) es una arquitectura de red neuronal profunda diseñada para aprender operadores no lineales que mapean funciones de entrada a funciones de salida. A diferencia de las redes convencionales que aprenden funciones escalares, DeepONet se enfoca en representar operadores completos, como soluciones de ecuaciones diferenciales, a partir de datos observados o simulaciones numéricas (Lu, Jin, et al. 2021).

8.1. Arquitectura

La arquitectura de DeepONet está compuesta por dos redes principales: la red de *branch* y la red de *trunk*. La red *branch* procesa las evaluaciones discretas de la función de entrada (por ejemplo, condiciones iniciales o de frontera), mientras que la red *trunk* recibe como entrada los puntos del dominio donde se desea evaluar la función de salida. La salida final se obtiene mediante el producto punto de los vectores generados por ambas redes, lo que permite representar operadores complejos con alta generalización a nuevos datos (Lu, Jin, et al. 2021).

8.2. Ejemplo de resolución de un operador usando DeepONet

Se resolverá el operador

$$G : f \to u$$

para el problema unidimensional de Poisson:

$$u''(x) = f(x), \quad x \in [0, 1]$$

con la condición de frontera de Dirichlet

$$u(0) = u(1) = 0$$

dónde el término f representa a una función continua arbitraria.

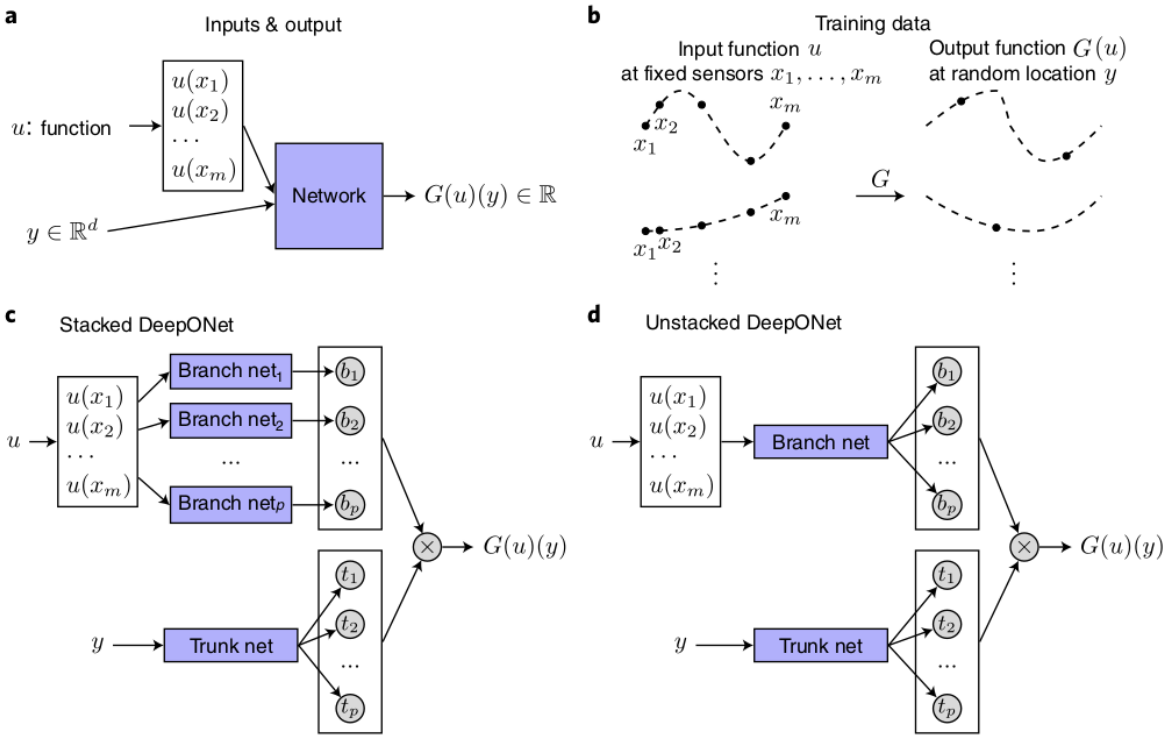


Figura 8.1.: **Ilustraciones del planteamiento del problema y arquitectura DeepONet que conducen a una buena generalización.** **a)** Para que la red aprenda un operador $G : u \to G(u)$ se necesita la entrada $[u(x_1), u(x_2), ..., u(x_m)]$ y la entrada y. **b)** Ilustración de los datos de entrenamiento. Para cada función de entrada u, se requiere el mismo número de evaluaciones en los mismos sensores dispersos $x_1, x_2, ..., x_m$. Sin embargo, no se impone ninguna restricción sobre el número ni las ubicaciones para la evaluación de las funciones de salida. **c)** La DeepONet *stacked* se inspira en el **Teorema de aproximación universal para operadores** y consta de una red *Trunk* y p redes *Branch* apiladas. La red cuya construcción se inspira en el mismo teorema es una DeepONet *stacked* formada al elegir la red *Trunk* como una red de una capa de ancho p y cada red *Branch* como una red de una capa oculta de ancho n. **d)** La red DeepONet *unstacked* se inspira en el **Teorema general de aproximación universal para operadores** y consta de una red *Trunk* y una red *Branch*. Una red DeepONet *unstacked* puede considerarse como una red DeepONet *stacked*, en la que todas las redes *Branch* comparten el mismo conjunto de parámetros (Lu, Jin, et al. 2021).

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

# Seed
dde.config.set_random_seed(123)

# Poisson equation: -u_xx = f
def equation(x, y, f):
    dy_xx = dde.grad.hessian(y, x)
    return -dy_xx - f

# Domain is interval [0, 1]
geom = dde.geometry.Interval(0, 1)

# Zero Dirichlet BC
def u_boundary(_):
    return 0

def boundary(_, on_boundary):
    return on_boundary

bc = dde.icbc.DirichletBC(geom, u_boundary, boundary)

# Define PDE
pde = dde.data.PDE(geom, equation, bc, num_domain=100, num_boundary=2)

# Function space for f(x) are polynomials
degree = 3
space = dde.data.PowerSeries(N=degree + 1)

# Choose evaluation points
num_eval_points = 10
evaluation_points = geom.uniform_points(num_eval_points, boundary=True)

# Define PDE operator
pde_op = dde.data.PDEOperatorCartesianProd(
    pde,
    space,
    evaluation_points,
    num_function=100,
    num_test=20
)

# Setup DeepONet
dim_x = 1
p = 32
net = dde.nn.DeepONetCartesianProd(
    [num_eval_points, 32, p],
    [dim_x, 32, p],
    activation="tanh",
    kernel_initializer="Glorot normal",
)

# Define and train model
model = dde.Model(pde_op, net)
dde.optimizers.set_LBFGS_options(maxiter=1000)
model.compile("L-BFGS")
model.train()

# Plot realisations of f(x)
```

```
n = 3
features = space.random(n)
fx = space.eval_batch(features, evaluation_points)

x = geom.uniform_points(100, boundary=True)
y = model.predict((fx, x))
```

```
# Setup figure
fig = plt.figure(figsize=(7, 8))
plt.subplot(2, 1, 1)
plt.title("Ecuación de Poisson: término f(x) y solución u(x)")
plt.ylabel("f(x)")
z = np.zeros_like(x)
plt.plot(x, z, "k-", alpha=0.1)

# Plot source term f(x)
for i in range(n):
    plt.plot(evaluation_points, fx[i], "--")

# Plot solution u(x)
plt.subplot(2, 1, 2)
plt.ylabel("u(x)")
plt.plot(x, z, "k-", alpha=0.1)
for i in range(n):
    plt.plot(x, y[i], "-")
plt.xlabel("x")

plt.show()
```

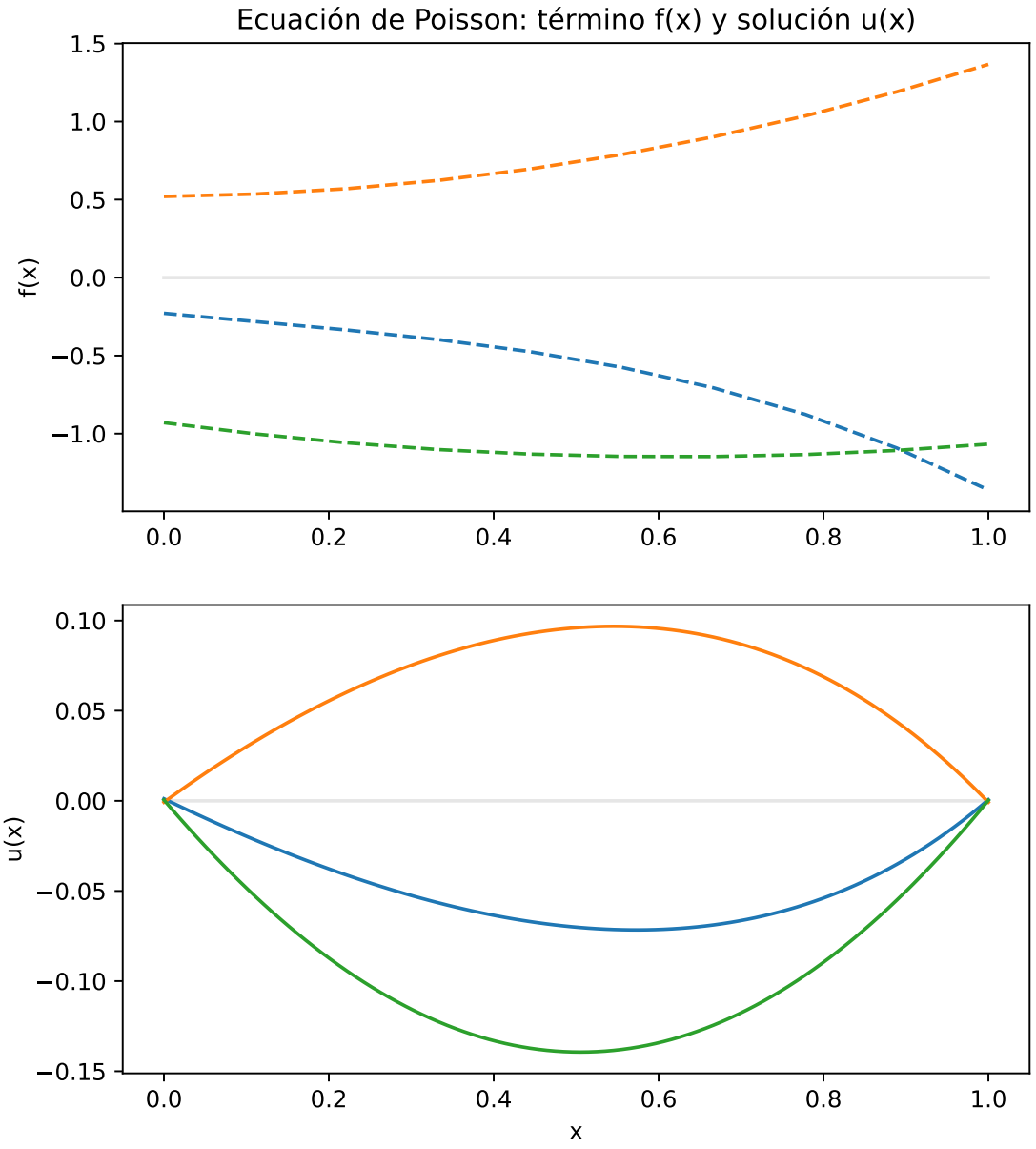


Figura 8.2.: Soluciones halladas por la red neuronal, en la parte superior las funciones arbitrarias $f(x)$, mientras que en la parte inferior está su solución $u(x)$, el color representa la relación *término-solución*.

8.3. Comparación con una PINN

En contraste con una red PINN convencional (Physics-Informed Neural Network), que resuelve una instancia específica de una ecuación diferencial para un conjunto dado de condiciones, DeepONet aproxima el operador general que resuelve varias instancias a la vez. Mientras que una PINN debe ser reentrenada para cada nuevo problema, DeepONet, una vez entrenado, puede predecir soluciones rápidamente para múltiples condiciones nuevas. Esto lo hace especialmente eficiente en aplicaciones donde se requiere realizar inferencias repetidas, como en control o diseño inverso (Kumar et al. 2024).

Part III.

Ecuación del Bio-Calor

La ecuación del bio-calor, formulada por Pennes (1948), surgió de su estudio pionero *"Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm"*. Publicado en el *Journal of Applied Physiology*, este trabajo fue el primero en cuantificar la interacción entre la temperatura arterial y tisular en humanos. Pennes combinó principios termodinámicos con mediciones experimentales en el antebrazo, estableciendo un modelo matemático que relacionaba el flujo sanguíneo, la producción metabólica de calor y la conducción térmica en tejidos.

Experimento

Durante su estudio, Pennes diseñó un experimento riguroso para medir la temperatura interna del antebrazo humano. Utilizó termopares tipo "Y" insertados transversalmente en la musculatura del antebrazo mediante una aguja estéril, como se ilustra en la Figura 8.3. Esta configuración permitía capturar un perfil térmico a lo largo del eje transversal, minimizando interferencias derivadas del contacto externo o la conducción axial no deseada.

La técnica experimental buscó máxima precisión geométrica y térmica: los termopares eran fijados con tensión controlada mediante un sistema mecánico que aseguraba trayectorias rectas y repetibles dentro del tejido. La inserción se realizaba con anestesia tópica mínima y bajo condiciones ambientales estables, lo cual garantizaba que los gradientes de temperatura registrados fueran atribuibles principalmente al metabolismo local y al efecto del flujo sanguíneo arterial.

Trascendencia

El modelo de Pennes simplificó la complejidad biológica al asumir un flujo sanguíneo uniforme y una transferencia de calor proporcional a la diferencia entre la temperatura arterial y la tisular. Aunque posteriores investigaciones refinaron sus supuestos, su ecuación sigue siendo un referente en bioingeniería térmica. Su trabajo no solo sentó las bases para aplicaciones clínicas, como la hipertermia oncológica, sino que también inspiró avances en el estudio de la termorregulación humana y el diseño de dispositivos médicos.

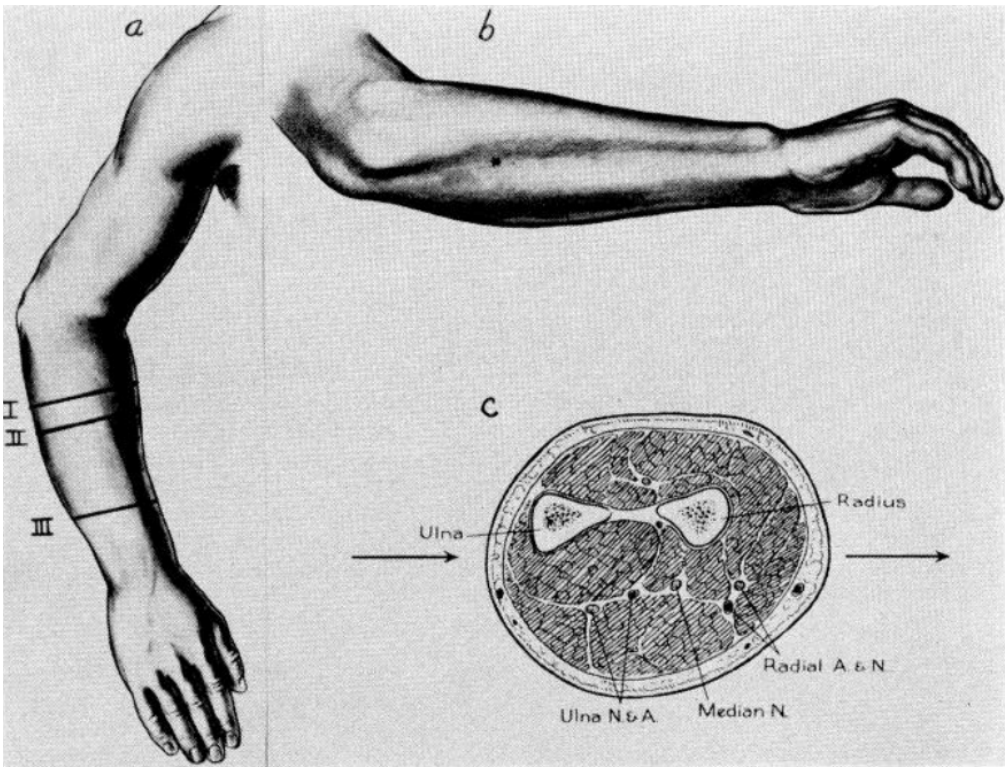


Figura 8.3.: **a)** Posición del brazo derecho (vista superior). La linea horizontal II indica el nivel de la figura c). **b)** Posición del brazo derecho (vista lateral). **c)**Sección transversal anatómica del antebrazo en el nivel II (Pennes 1948).

9. Forma de la ecuación

La ecuación diferencial de bio-calor de Pennes (1948) modela la transferencia de calor en tejidos biológicos, integrando efectos de conducción, perfusión sanguínea y metabolismo. Su forma general es:

$$\rho c \frac{\partial T}{\partial t} = k_{\text{eff}} \frac{\partial^2 T}{\partial x^2} - \rho_b c_b \omega_b (T - T_a) + \mathcal{Q}, \quad x \in \Omega,\, t \in [0, t_f] \tag{9.1}$$

Tabla 9.1.: Tabla de nomenclaturas de la Ecuación 9.1.

Símbolo	Descripción	Unidades
T	Temperatura del tejido	°C
ρ	Densidad del tejido	$\frac{kg}{m^3}$
c	Calor específico del tejido	$\frac{J}{kg°C}$
k_{eff}	Conductividad térmica	$\frac{W}{m°C}$
ρ_b	Densidad de la sangre	$\frac{kg}{m^3}$
c_b	Calor específico de la sangre	$\frac{J}{kg°C}$
ω_b	Tasa de perfusión sanguínea	$1/s$
T_a	Temperatura arterial	°C
$\mathcal{Q} = q_m + q_p$	Fuente de calor	$\frac{W}{m^3}$
q_m	Metabolismo	$\frac{W}{m^3}$
q_p	Externa	$\frac{W}{m^3}$

9.1. Versión reducida (adimensionalizada)

Mediante escalamiento:

$$T' = T - T_a \quad \theta = \frac{T'}{T_M - T_a} \quad X = \frac{x}{L_0} \quad \tau = \frac{t}{t_f} \tag{9.2}$$

Tabla 9.2.: Tabla de nomenclatura de las relaciones para escalamiento.

Símbolo	Descripción	Unidades
L_0	Longitud característica del dominio	m
t_f	Tiempo final de simulación	s

la Ecuación 9.1 se convierte en:

$$\partial_\tau \theta = a_1 \partial_{XX} \theta - a_2 W \theta + a_3$$

para una dimensión espacial; para el caso dos dimensional se tiene:

$$\partial_\tau \theta = a_1 \nabla^2 \theta - a_2 W \theta + a_3 \tag{9.3}$$

Parámetros adimensionales:
- $a_1 = \frac{t_f}{\alpha L_0^2}$ (difusividad térmica $\alpha = \frac{k_{\text{eff}}}{\rho c}$).
- $a_2 = \frac{t_f c_b}{\rho c}$.
- $a_3 = \frac{t_f \mathcal{Q}}{\rho c (T_M - T_a)}$.
- $W = \rho_b \omega_b$: Tasa volumétrica de perfusión ($\text{kg/m}^3 \cdot \text{s}$).

Cabe decir que se modeló el caso más sencillo, que es asumiendo la fuente de calor $\mathcal{Q} = 0$.

9.2. Condiciones de uso adecuadas

1. **Tejidos homogéneos**: Aproximación válida para regiones con propiedades térmicas uniformes.

2. **Perfusión sanguínea constante**: Supone flujo sanguíneo estable en el dominio.

3. **Aplicaciones clínicas**: Hipertermia, crioterapia y modelado térmico en terapias oncológicas.

9.3. Solución analítica

Consideremos la ecuación diferencial parcial 9.3 sin el término a_3,

$$\partial_\tau \theta(x, y, \tau) = a_1 \nabla^2 (x, y, \tau) - a_2 W \theta(x, y, \tau), \tag{9.4}$$

donde a_1 y a_2 son parámetros positivos adimensionales, W es una constante asociada al término de disipación. El dominio de estudio corresponde al cuadrado $[0,1] \times [0,1]$ en el espacio y al intervalo $[0,1]$ en el tiempo adimensional τ.

Las condiciones de frontera establecidas son mixtas:

- En $y = 0$ es $y = 1$ se imponen condiciones de tipo Neumann, es decir,

$$\partial_y \theta(x, 0, \tau) = 0, \quad \partial_y \theta(x, 1, \tau) = 0, \quad \tau \geq 0.$$

- En $x = 0$ se prescribe una condición de tipo Dirichlet:

$$\theta(0, y, \tau) = 0, \quad \tau \geq 0.$$

- En $x = 1$ se fija una condición de tipo Neumann no homogénea, con dependencia lineal en el tiempo:

$$\partial_x \theta(1, y, \tau) = \tau, \quad \tau \geq 0.$$

Además, se establece la condición inicial

$$\theta(x, y, 0) = 0, \quad (x, y) \in [0,1] \times [0,1].$$

9.3.1. Reducción del problema

Obsérvese que la ecuación es independiente de la variable y, y que las condiciones de frontera en esa dirección son homogéneas (Neumann). Por tanto, la solución puede asumirse también independiente de y, reduciendo el problema a una dimensión espacial. En consecuencia, se escribe

$$\theta(x, y, \tau) \equiv \theta(x, \tau).$$

La ecuación 9.4 se reduce a resolver

$$\partial_\tau \theta(x, \tau) = a_1 \frac{\partial^2 \theta}{\partial x^2}(x, \tau) - a_2 W \theta(x, \tau), \tag{9.5}$$

con las condiciones de frontera

$$\theta(0, \tau) = 0, \quad \partial_x \theta(1, \tau) = \tau,$$

y la condición inicial

$$\theta(x, 0) = 0.$$

9.3.2. Método de solución

Para resolver este problema, se aplica la técnica de separación en solución **particular + solución homogénea**. Se introduce la transformación

$$\theta(x,\tau) = u(x,\tau) + \varphi(x,\tau),$$

donde $\varphi(x,\tau)$ se escoge de manera que satisfaga la condición de frontera no homogénea en $x = 1$. Una elección natural es

$$\varphi(x,\tau) = x\tau,$$

pues se cumple

$$\partial_x \varphi(1,\tau) = \tau, \quad \varphi(0,\tau) = 0.$$

De este modo, la función $u(x,\tau)$ obedece condiciones homogéneas:

$$u(0,\tau) = 0, \quad \partial_x u(1,\tau) = 0,$$

y al sustituir en la ecuación 9.5, se obtiene para $u(x,\tau)$:

$$\partial_\tau u = a_1 \frac{\partial^2 u}{\partial x^2} - a_2 W u - a_2 W x\tau - x.$$

9.3.3. Solución mediante series

Se plantea una expansión de $u(x,\tau)$ en términos de los autovalores y autofunciones del problema de Sturm–Liouville asociado:

$$u(x,\tau) = \sum_{m=1}^{\infty} a_m(\tau) sen(\alpha_m x),$$

donde los modos propios satisfacen

$$\alpha_m = \frac{(2m-1)\pi}{2}.$$

La evolución temporal de los coeficientes $a_m(\tau)$ resulta

$$a_m(\tau) = c_m \frac{1 - e^{-\sigma_m \tau}}{\sigma_m} + d_m \frac{\sigma_m \tau - 1 + e^{-\sigma_m \tau}}{\sigma_m^2},$$

con

$$\sigma_m = a_1 \alpha_m^2 + a_2 W, \quad c_m = -\frac{2(-1)^{m-1}}{\alpha_m^2}, \quad d_m = -\frac{2a_2 W (-1)^{m-1}}{\alpha_m^2}.$$

Finalmente la solución aproximada truncada a M modos se expresa como

$$u(x,\tau) \approx x\tau + \sum_{m=1}^{M} a_m(\tau) sen(\alpha_m x). \tag{9.6}$$

9.3.4. Solución truncada codificada

Basado la Ecuación 9.6 se codificó la solución analítica aproximada y se guardó en un dataframe para los tiempos de interés.

Listado 9.1 Guardado de los datos de la solución analítica (Parte 1).

```
import numpy as np
import pandas as pd

# ---------- Parámetros físicos ----------
p = 1050     # densidad
c = 3639     # calor específico
keff = 5     # conductividad efectiva
tf = 1800    # tiempo característico
L0 = 0.05    # longitud característica
cb = 3825    # perfusión
Q = 0        # fuente (no usada aquí)

# Parámetro auxiliar
alpha_phys = p * c / keff

# ---------- Coeficientes adimensionales ----------
a1 = tf / (alpha_phys * L0**2)
a2 = tf * cb / (p * c)

# ---------- Parámetros de la serie analítica ----------
W = 1.0
M = 60  # número de modos en el truncamiento

m = np.arange(1, M+1)
alpha_m = (2*m - 1) * np.pi / 2.0     #  _m
lambda_m = alpha_m**2
sigma_m = a1 * lambda_m + a2 * W      #  _m

sign = (-1.0)**(m-1)
c_m = -2.0 * sign / (alpha_m**2)
d_m = -2.0 * a2 * W * sign / (alpha_m**2)

# ---------- Mallado en espacio y tiempos ----------
step = 0.04
# 0.00, 0.04, ..., 1.00
grid_vals = np.round(np.arange(0.0, 1.0 + 1e-12, step), 2)
x = grid_vals.copy()
y = grid_vals.copy()
X, Y = np.meshgrid(x, y, indexing='xy')

times = [0.0, 0.25, 0.5, 0.75, 1.0]
```

```
# ---------- Función para calcular  (x, ) ----------
def theta_xt(x_vec, tau_val):
    """Devuelve theta(x, tau) en un vector de x,
    usando la serie truncada."""
    # coeficientes a_m(tau)
    small_mask = np.isclose(sigma_m, 0.0, atol=1e-12)
    a = np.empty_like(sigma_m)
    if np.any(small_mask):
        a[small_mask] = (c_m[small_mask]*tau_val
        + 0.5*d_m[small_mask]*tau_val**2)
    if np.any(~small_mask):
        s = sigma_m[~small_mask]
        cm = c_m[~small_mask]
        dm = d_m[~small_mask]
        a[~small_mask] = (cm*(1.0 - np.exp(-s*tau_val))/s
        + dm*(s*tau_val - 1.0
        + np.exp(-s*tau_val))/s**2)
    # suma modal
    sin_ax = np.sin(np.outer(alpha_m, x_vec))   # (M, Nx)
    theta_series = a.dot(sin_ax)                # (Nx,)
    # añadir  (x, ) = x
    return theta_series + tau_val * x_vec

# ---------- Construcción del DataFrame ----------
results = []
for t_val in times:
    theta_x = theta_xt(x, t_val)         # (Nx,)
    Theta = np.tile(theta_x, (y.size, 1))  # shape (Ny, Nx)
    for xi, yi, thetai in zip(X.ravel(), Y.ravel(), Theta.ravel()):
        results.append([t_val, float(xi), float(yi), float(thetai)])

df = pd.DataFrame(results, columns=["time", "X", "Y", "Theta"])

# ---------- Guardar en CSV ----------
df.to_csv("data/sol_analitica.csv", index=False)
```

10. Otras aplicaciones de la ecuación del bio-calor

- Quintero et al. (2017) desarrollan un modelo basado en ecuaciones diferenciales parciales que integra la ecuación del bio-calor y la ley de Arrhenius para estimar el daño térmico en tratamientos de hipertermia superficial. Utilizan el método de líneas para resolver el sistema y plantean un problema de optimización que busca maximizar el daño al tejido tumoral minimizando el daño colateral. Su trabajo demuestra cómo la modelación matemática puede guiar estrategias terapéuticas más seguras y eficaces.
- Dutta y Rangarajan (2018) presentan una solución analítica cerrada en dos dimensiones para la ecuación del bio-calor, considerando modelos de conducción tanto de tipo Fourier como no-Fourier. Mediante el uso de la transformada de Laplace, analizan la influencia de parámetros fisiológicos como la perfusión sanguínea y el tiempo de relajación térmica sobre la evolución de la temperatura. Su investigación aporta una base teórica sólida para comprender la propagación térmica en tejidos vivos durante la hipertermia terapéutica.
- Yang et al. (2014) propone una estrategia numérica para resolver problemas inversos de conducción térmica en tejidos biológicos multicapa, utilizando un enfoque en diferencias finitas y el concepto de tiempo futuro. El estudio se enfoca en predecir las condiciones de frontera necesarias para generar distribuciones de temperatura deseadas. La implementación de este método permite estimar parámetros relevantes en tiempo real, lo cual resulta esencial para el control térmico preciso en procedimientos médicos no invasivos como la hipertermia localizada.

Part IV.

Estudio de caso

Hipertermia como opción terapéutica complementaria en el manejo de cáncer

La Organización Mundial de la Salud (2022) en su página web define Cáncer como:

«Cáncer» es un término genérico utilizado para designar un amplio grupo de enfermedades que pueden afectar a cualquier parte del organismo; también se habla de «tumores malignos» o «neoplasias malignas». Una característica definitoria del cáncer es la multiplicación rápida de células anormales que se extienden más allá de sus límites habituales y pueden invadir partes adyacentes del cuerpo o propagarse a otros órganos, en un proceso que se denomina «metástasis». La extensión de las metástasis es la principal causa de muerte por la enfermedad.

Por su parte Instituto Nacional del Cáncer (2021) aporta lo siguiente:

Es posible que el cáncer comience en cualquier parte del cuerpo humano, formado por billones de células. En condiciones normales, las células humanas se forman y se multiplican (mediante un proceso que se llama división celular) para formar células nuevas a medida que el cuerpo las necesita. Cuando las células envejecen o se dañan, mueren y las células nuevas las reemplazan. A veces el proceso no sigue este orden y las células anormales o células dañadas se forman y se multiplican cuando no deberían. Estas células tal vez formen tumores, que son bultos de tejido. Los tumores son cancerosos (malignos) o no cancerosos (benignos).

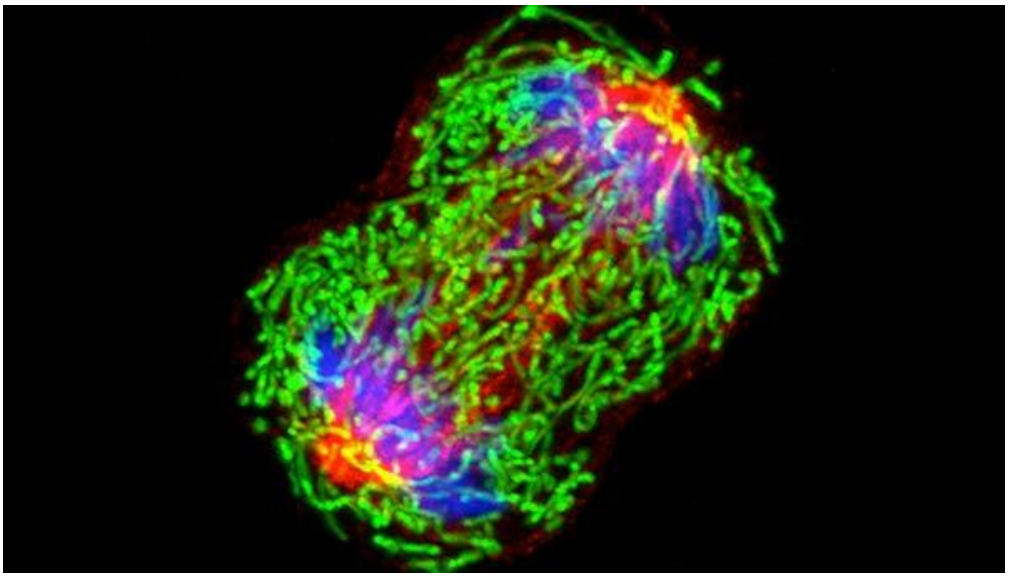


Figura 10.1.: Una célula de cáncer de seno que se multiplica (Instituto Nacional del Cáncer 2021).

Ésta enfermedad es la principal causa de muerte a nivel mundial, solo en 2020 arrebató casi 10 millones de vidas y, según datos de Organización Mundial de la Salud (2022), los cánceres más comunes en 2020 fueron:

- De mama (2.26 millones de casos)
- De pulmón (2.21 millones de casos)
- De colon (1.93 millones de casos)
- De próstata (1.41 millones de casos)
- De piel (distinto del melanoma) (1.20 millones de casos)
- Gástrico (1.09 millones de casos)

Es ante este panorama, distintos tratamientos surgen con el objetivo de erradicar la enfermedad siempre que se tenga una detección oportuna. Uno de dichos tratamientos es la hipertermia, según en el National Cancer Institute (2021), es un método que consiste en calentar el tejido corporal hasta los 39-45 °C para ayudar a erradicar células cancerígenas con pequeñas o nulas lesiones en el tejido sano. La hipertermia también es llamada terapia térmica o termoterapia.

Uno de los principales retos de este tratamiento es la creación de un modelo óptimo que se adecue al comportamiento de la transferencia de calor que se hace a los tejidos con el fin de dañar únicamente el área en el que se encuentran las células cancerígenas, es por ello que los modelos de inteligencia artificial y más precisamente las PINN's Capítulo 7 surgen como posible solución a este reto.

El presente estudio utilizó como punto de partida el trabajo realizado por Alessio Borgi (2023) para modelar el calentamiento del tejido corporal usando la ecuación del Bio-Calor en dos dimensiones.

11. Metodología

En esta sección se describe el enfoque metodológico utilizado para evaluar la efectividad de una PINN utilizando una arquitectura DeepONet con el objetivo de resolver la ecuacion del Bio-Calor. El proceso metodológico se divide en las siguientes etapas:

11.1. Aportaciones del modelo

Ya que se parte del trabajo de Alessio Borgi (2023), se examinó que dos de los puntos a mejorar de la red neuronal que plantearon son:

1. Desarrollar nuevas arquitecturas para la red neuronal y explorar nuevas configuraciones.
2. Combinar las fortalezas de los algoritmos de optimización Adam y L-BFGS Sección 7.1 para mejorar la velocidad de convergencia y la precisión.

Teniendo los anteriores puntos en cuenta, se procedió a abordarlos e implementarlos dentro del diseño del modelo.

11.2. Diseño del modelo

El lenguaje seleccionado fue Python, a su vez el código se basa enteramente en la librería Deepxde creada por Lu, Meng, et al. (2021), la cual está directamente enfocada a resolver ecuaciones diferenciales, se usó además como backend *tensorflow_compat_v1*, siendo su elección debida únicamente a la familiarización previa que se tenía con ella. Finalmente, el entorno donde se programó y optimizó el código fué en *Google Colab* ya que la potencia de cómputo ofrecida por la plataforma era necesaria para ejecutar el modelo.

11.3. Implementación del modelo

La implementación del modelo se llevó a cabo en dos etapas clave: **(1) el desarrollo del código base para resolver la ecuación del Bio-Calor mediante DeepXDE**, y **(2) la optimización sistemática de los hiperparámetros**. Para esta última, se siguieron las recomendaciones del estudio de Alessio Borgi (2023), adaptadas a las

particularidades del problema. Se ajustaron parámetros críticos como el número de épocas de entrenamiento (*iterations*), la tasa de aprendizaje (*learning rate*) así como un decaimiento en el mismo dependiente de la iteración actual (*decay*), la función de activación (*elu*) y el esquema de inicialización de pesos (*Glorot normal*). Estos ajustes se realizaron mediante un proceso iterativo que buscaba minimizar la función de pérdida mientras se mantenía un tiempo de entrenamiento computacionalmente viable.

11.4. Evaluación del modelo

Para validar el desempeño del modelo propuesto, se realizó una evaluación exhaustiva utilizando un *conjunto de datos independiente*, el cual no fue empleado durante las fases de entrenamiento o ajuste de hiperparámetros. Este enfoque garantiza una medición objetiva de la capacidad de generalización del modelo ante datos no vistos.

Las predicciones generadas por el modelo fueron analizadas mediante visualizaciones espaciotemporales, las cuales permiten comparar cualitativamente el comportamiento de las soluciones pronosticadas frente a los rangos físicos y temporales definidos en el problema. En particular, se generaron gráficas de superficies 3D que muestran la evolución de las variables de interés a lo largo del dominio espacial y temporal bajo estudio. Adicionalmente, se incluyeron representaciones de cortes transversales y series temporales en puntos estratégicos para facilitar la interpretación de los resultados.

Cabe destacar que este análisis preliminar se centró en examinar la coherencia física y la estabilidad numérica de las predicciones. Para la evaluación cuantitativa del modelo, se implementó una comparación directa con las soluciones obtenidas mediante el método numérico de **Crank-Nicolson**, resuelto en *Julia* utilizando la librería *DifferentialEquations.jl*.

11.5. Comparación de resultados

Para evaluar el desempeño predictivo del modelo propuesto, se realizaron dos tipos de comparaciones:

1. Una evaluación cualitativa basada en visualizaciones.
2. Un análisis cuantitativo mediante métricas de error estandarizadas.

En primer lugar, se llevó a cabo una comparación visual con los resultados reportados en el trabajo de Alessio Borgi (2023), dado que dicho estudio no incluye datos numéricos tabulados, sino únicamente representaciones gráficas de las soluciones. Esta comparación permitió identificar coincidencias y discrepancias en el comportamiento espaciotemporal de las variables de interés, destacando las fortalezas del modelo propuesto en términos de estabilidad numérica.

En segundo lugar, para una evaluación cuantitativa rigurosa, se compararon las predicciones del modelo con soluciones de referencia generadas mediante el método de Crank-Nicolson, implementado en Julia utilizando la librería DifferentialEquations.jl. La comparación se realizó sobre una malla uniforme de 26×26 puntos en el cuadrado de $[0, 1] \times [0, 1]$, calculando para cada instante de tiempo relevante las siguientes métricas:

- Error Absoluto Medio (MAE).
- Error Absoluto Máximo (MaxAE).
- Error L2 (norma euclidiana).

Estos criterios permitieron cuantificar no solo la precisión global del modelo, sino también sus desviaciones locales más significativas, particularmente en regiones con alta variabilidad espacial. Los resultados detallados de este análisis, junto con una discusión sobre la eficiencia computacional relativa entre ambos métodos, se presentan en la Tabla 14.1 y Tabla 14.2.

11.6. Análisis y conclusión

Finalmente, se realizó un análisis detallado de los resultados obtenidos para extraer conclusiones significativas. Se proporcionaron recomendaciones basadas en los hallazgos del estudio, lo que permitió establecer un marco para interpretaciones analíticas profundas y recomendaciones bien fundamentadas en la sección de conclusiones del estudio.

Este enfoque metodológico proporcionó una base sólida para los resultados obtenidos, asegurando la integridad y la calidad del análisis realizado en el estudio.

12. Predicciones del método numérico

Para validar los resultados del modelo propuesto, se implementó el método de Crank-Nicolson en Julia utilizando la librería *DifferentialEquations.jl*. El método se resolvió sobre una malla refinada de 51×51 puntos para garantizar alta precisión en la solución numérica, calculando las predicciones en los tiempos clave: t = [0.0, 0.25, 0.50, 0.75, 1.0]. En la Figura 12.1 se muestran las cinco gráficas generadas, las cuales ilustran la evolución temporal de la solución en el dominio de estudio.

Listado 12.1 Método de Crank-Nicolson (Parte 1).

```
using DifferentialEquations, LinearAlgebra
using DataFrames, CSV

# --- PARÁMETROS FÍSICOS Y DIMENSIONALES ---------------------------------
p, c  = 1050.0, 3639.0                        # densidad, calor específico
k_eff = 5.0                                   # conductividad
t_f   = 1800.0                                # tiempo final
L     = 0.05                                  # longitud del dominio
c_b   = 3825.0                                # coef. perfusión
Q     = 0.0                                   # fuente térmica
T_M, T_a = 45.0, 37.0                         # temp máxima, temp ambiente

# --- COEFICIENTES ADIMENSIONALES ----------------------------------------
  = p * c / k_eff
a  = t_f / (  * L^2)
a  = t_f * c_b / (p * c)
a  = (t_f * Q) / (p * c * (T_M - T_a))  # aquí Q=0 → a =0

# --- MALLA ESPACIAL -----------------------------------------------------
Nx, Ny = 51, 51
dx, dy = 1.0 / (Nx - 1), 1.0 / (Ny - 1)
x = range(0, 1, length=Nx)
y = range(0, 1, length=Ny)
N = Nx * Ny  # total de puntos

# --- CONDICIÓN INICIAL --------------------------------------------------
u0 = zeros(N)   # coincide con la condición inicial  =0
```

```
# --- SISTEMA DE EDOs DEL PDE --------------------------------------------
function f!(du, u, _, )
    U = reshape(u, Nx, Ny)          # arreglo 2D con los valores actuales
    D = zeros(eltype(U), Nx, Ny)    # preasigno en cero
```

```
    # Enforce Dirichlet en X=0:  (0,y, ) = 0 (mantenemos esa fila = 0)
    U[1, :] .= 0.0

    @inbounds for i in 1:Nx, j in 1:Ny
        # Si estamos en la frontera izquierda X=0 (Dirichlet),
        # la condición fija implica  _    = 0 en esos puntos
        # (se mantiene constante).
        if i == 1
            D[i, j] = 0.0
            continue
        end

        # Derivada segunda en x:
        if i == Nx
            # Neumann con valor no homogéneo en X=1:  _X  (1,y, ) =
            # uso ghost node U_{N+1} tal que (U_{N+1} - U_N)/dx =
            #   → U_{N+1} = U_N +  *dx
            U_ghost = U[Nx, j] +   * dx
            d2x = (U_ghost - 2U[Nx, j] + U[Nx-1, j]) / dx^2
        else
            # interior (o i==2 cuando i-1 existe)
            d2x = (U[i+1, j] - 2U[i, j] + U[i-1, j]) / dx^2
        end

        # Derivada segunda en y (bordes Y=0 y Y=1 son
        # Neumann homogéneos  _Y = 0)
        if j == 1
            # j=1: ghost U_0 = U_2  → segunda derivada =
            # 2*(U[2] - U[1]) / dy^2
            d2y = 2*(U[i, 2] - U[i, 1]) / dy^2
        elseif j == Ny
            # j=Ny: ghost U_{Ny+1} = U_{Ny-1} → segunda derivada =
            # 2*(U[Ny-1] - U[Ny]) / dy^2
            d2y = 2*(U[i, Ny-1] - U[i, Ny]) / dy^2
        else
            d2y = (U[i, j+1] - 2U[i, j] + U[i, j-1]) / dy^2
        end

        # ECUACIÓN:  _    = a  * (d2x + d2y) - a  *   + a
        D[i, j] = a  * (d2x + d2y) - a  * U[i, j] + a
    end

    du .= vec(D)
end
```

```
# --- RESOLVER PDE -------------------------------------------------------
 span = (0.0, 1.0)
prob = ODEProblem(f!, u0,  span)
taus = [0.0, 0.25, 0.5, 0.75, 1.0]
sol = solve(prob, Trapezoid(), dt=8e-4, saveat=taus)

# --- PROCESAR SOLUCIÓN EN GRILLA REDUCIDA -----------------------------
idxs = 1:2:Nx  # índices para submuestreo

# Preasignar vectores para crear el DataFrame
times = Float64[]
Xs = Float64[]
Ys = Float64[]
Thetas = Float64[]
```

```
for   in taus
    θ = reshape(sol( ), Nx, Ny)
    # Asegurarse (por consistencia) que la frontera izquierda permanece 0
    θ[1, :] .= 0.0
    for j in idxs, i in idxs
        push!(times,  )
        push!(Xs, x[i])
        push!(Ys, y[j])
        push!(Thetas, θ[i, j])
    end
end

df = DataFrame(time=times, X=Xs, Y=Ys, Theta=Thetas)

# --- GUARDAR CSV ----------------------------------------------------
ruta = "data"
CSV.write(joinpath(ruta, "crank_nick.csv"), df)
```

```
using DataFrames, CSV, Plots, Statistics
pyplot()

# --- OBTENER VALORES ÚNICOS Y ORDENADOS DE X, Y, TIME ----------------
x_vals = sort(unique(df.X))
y_vals = sort(unique(df.Y))
times = sort(unique(df.time))  # tiempos

Nx, Ny = length(x_vals), length(y_vals)

# --- RECONSTRUIR MATRICES 2D DE THETA PARA CADA TIEMPO ----------------
solutions = []

for t in times
    dft = filter(:time => ==(t), df)

    # Crear matriz vacía
    θ = fill(NaN, Nx, Ny)

    # Llenar la matriz con los valores correspondientes
    for row in eachrow(dft)
        ix = findfirst(==(row.X), x_vals)
        iy = findfirst(==(row.Y), y_vals)
        θ[ix, iy] = row.Theta
    end

    push!(solutions, θ)
end

# --- DETERMINAR ESCALA GLOBAL DE COLORES ------------------------------
zmin = minimum([minimum(u) for u in solutions])
zmax = maximum([maximum(u) for u in solutions])

# --- GRAFICAR EN LAYOUT 3x2 -------------------------------------------
p = plot(layout = (3, 2), size = (800, 900))

for (i, (t, θ)) in enumerate(zip(times, solutions))
    surface!(
        p, y_vals, x_vals, θ;
        camera = (45,30),
```

```
        xlabel = "Y",
        ylabel = "X",
        zlabel = "T   ",
        title = "t = $(t)",
        subplot = i,
        c = :thermal,
        clim = (zmin, zmax),
        legend = false
    )
end

# Eliminar ejes y contenido del subplot 6
plot!(p[6], framestyle = :none,
            grid = false,
            xticks = false,
            yticks = false)
close("all")

display(p)
```

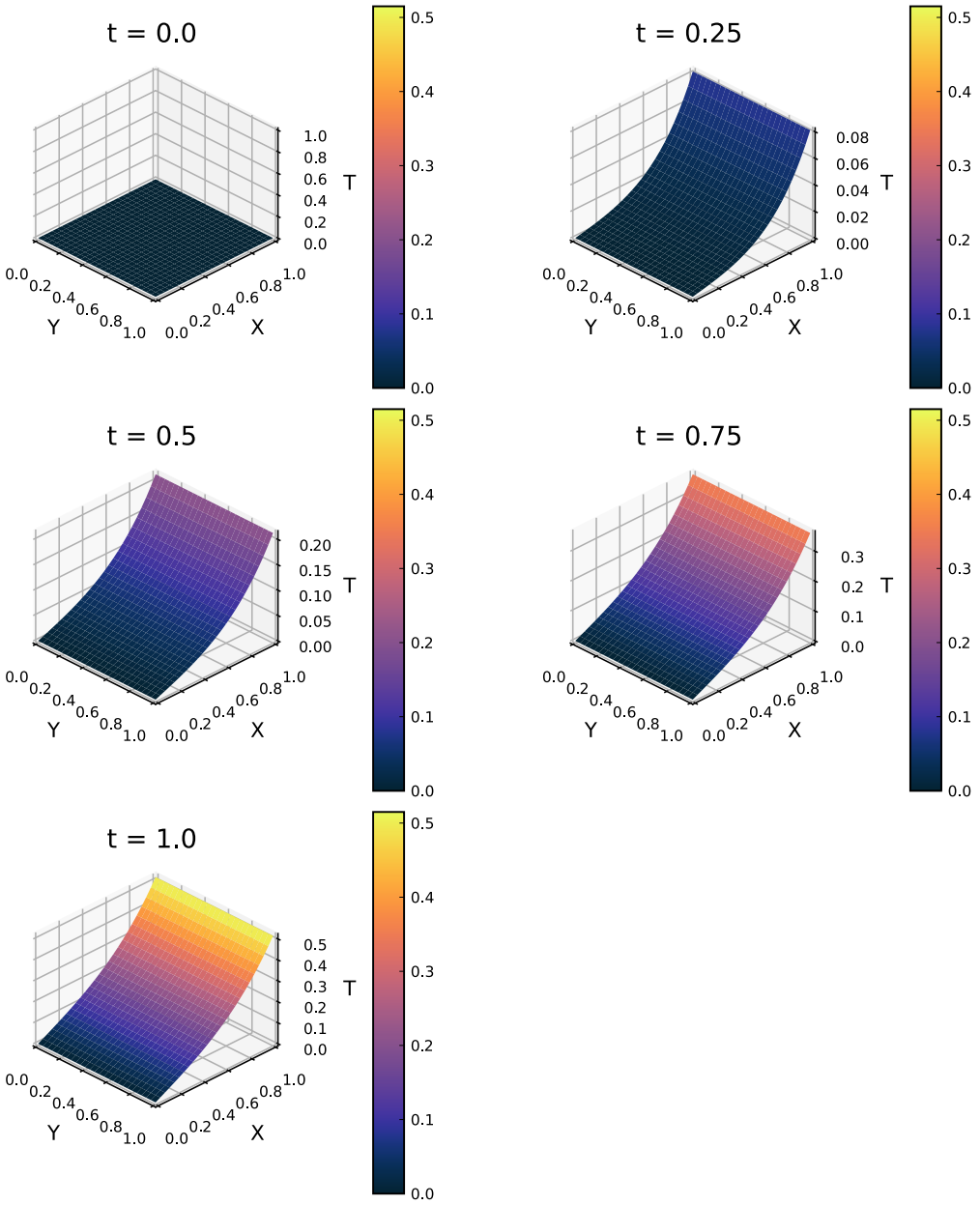


Figura 12.1.: Resultados obtenidos mediante el método numérico de Crank Nickolson, para la temperatura T en función de las coordenadas X e Y, aquí el código de colores representa el gradiente de temperaturas; todas las medidas están adimensionalizadas dadas las ecuaciones 9.2.

12.1. Análisis de sensibilidad

Con el objetivo de optimizar el proceso de comparación cuantitativa con el modelo de redes neuronales, se exportó un subconjunto representativo de los resultados Código 12.1. Aunque la simulación original utilizó una malla de 51×51 puntos, se almacenaron únicamente los valores correspondientes a una grilla de 26×26 puntos. Esta decisión se basó en:

1. Suficiencia estadística: La densidad de puntos conserva los patrones espaciales críticos.
2. Eficiencia computacional: Reduce el tamaño del archivo sin perder información relevante.

Los datos se guardaron en un archivo CSV estructurado con las siguientes columnas:

- Coordenadas espacio-temporales (t, x, y) para cada punto de la grilla 26×26.
- Valores de la solución en los tiempos de interés.

Este archivo permitió calcular de manera estandarizada las métricas de error (MAE, MaxAE, error L2) en la sección de comparación de resultados Sección 14.2.1.

13. Métricas del modelo

Conforme se ha referido previamente, el desarrollo del modelo predictivo se realizó utilizando el framework DeepXDE (versión 1.10.1) con backend de TensorFlow 1.x (configurado mediante tensorflow.compat.v1). Para asegurar reproducibilidad, se fijó la semilla aleatoria en 123 a nivel de DeepXDE, TensorFlow y NumPy. La red neuronal se implementó como un DeepONetCartesianProd con la siguiente estructura especializada:

- Rama (*branch*)
  - Capa de entrada: $(\text{num\_sensors} + 1)^2 = 49$ neuronas.
  - 3 capas ocultas de 20 neuronas cada una.
- Tronco (*trunk*)
  - Capa de entrada: 3 neuronas (coordenadas espaciotemporales x, y, t).
  - Misma configuración de capas ocultas que la rama.
- Hiperparámetros clave
  - Función de activación: ELU (Exponential Linear Unit).
  - Inicialización de pesos: Glorot normal.
  - Optimizador: ADAM con tasa de aprendizaje inicial de 2×10^{-3} y decaimiento exponencial (*decay_rate*=0.05 cada 500 pasos).

```
import deepxde as dde
import numpy as np
import tensorflow as tf

# ------------------------------------------------------------------------
# Constantes y Parametros
# ------------------------------------------------------------------------

# Backend y semilla
dde.backend.set_default_backend("tensorflow.compat.v1")
dde.config.set_random_seed(123)

# Parametros fisicos
p = 1050
c = 3639
keff = 5
final_time = 1800
L0 = 0.05
cb = 3825
Q = 0
TM = 45
```

```
Ta = 37
alpha = p * c / keff

# Coeficientes adimencionales
a1 = final_time / (alpha * L0**2)
a2 = final_time * cb / (p * c)
a3 = (final_time * Q) / (p * c * (TM - Ta))

# Dominio de las fronteras
x_initial, x_boundary = 0.0, 1.0
y_initial, y_boundary = 0.0, 1.0
t_initial, t_final = 0.0, 1.0

# Configuracion del numero de datos
pts_dom = 45
pts_bc = 30
pts_ic = 20
num_test = 25

# Malla de sensores y espacio de funciones
num_sensors = 6
num_function = 25
size_cov_matrix = 50

# Arquitectura de la red
width_net = 20
len_net = 3
AF = "elu"
k_initializer = "Glorot normal"


# Parámetros de entrenamiento
num_iterations = 20000
learning_rate = 2e-3
decay_rate = 0.05
decay_steps = 500

# ---------------------------------------------------------------------------
# Dominio espacial y temporal
# ---------------------------------------------------------------------------

spatial_domain = dde.geometry.Rectangle([x_initial, y_initial],
                                        [x_boundary, y_boundary])
time_domain = dde.geometry.TimeDomain(t_initial, t_final)
geomtime = dde.geometry.GeometryXTime(spatial_domain, time_domain)

# ---------------------------------------------------------------------------
# EDP, CI y CF
# ---------------------------------------------------------------------------

def initial_condition(X):
    X = np.asarray(X)
    if X.ndim == 1:
        return 0.0
    return np.zeros((X.shape[0], 1))

def heat_equation(x, u, coords):
    u_t = dde.grad.jacobian(u, x, i=0, j=2)
    u_xx = dde.grad.hessian(u, x, i=0, j=0)
    u_yy = dde.grad.hessian(u, x, i=1, j=1)
    return u_t - a1*(u_xx + u_yy) + a2*u
```

```
def zero_value(X):
    return 0

def time_value(X):
    return X[:, 2]

def is_on_vertex(x):
    vertices = np.array([[x_initial, y_initial],
                         [x_boundary, y_initial],
                         [x_initial, y_boundary],
                         [x_boundary, y_boundary]])
    return any(np.allclose(x, v) for v in vertices)

def is_initial(X, on_initial):
    return on_initial and np.isclose(X[2], t_initial)

def left_boundary(X, on_boundary):
    spatial = X[0:2]
    t = X[2]
    return (
        on_boundary
        and np.isclose(spatial[0], x_initial)
        and not np.isclose(t, t_initial)
        and not is_on_vertex(spatial)
    )

def right_boundary(X, on_boundary):
    spatial = X[0:2]
    t = X[2]
    return (
        on_boundary
        and np.isclose(spatial[0], x_boundary)
        and not np.isclose(t, t_initial)
        and not is_on_vertex(spatial)
    )

def up_low_boundary(X, on_boundary):
    spatial = X[0:2]
    t = X[2]
    return (on_boundary
    and (np.isclose(spatial[1], y_initial)
    or np.isclose(spatial[1], y_boundary))
    and not np.isclose(t, t_initial)
    and not is_on_vertex(spatial)
    )

# Condiciones iniciales y de frontera
ic = dde.icbc.IC(geomtime, initial_condition, is_initial)
left_bc = dde.icbc.DirichletBC(geomtime,
                               zero_value, left_boundary)
right_bc = dde.icbc.NeumannBC(geomtime,
                              time_value, right_boundary)
up_low_bc = dde.icbc.NeumannBC(geomtime,
                               zero_value, up_low_boundary)

# ------------------------------------------------------------------------
# Construccion de los datos
# ------------------------------------------------------------------------

pde_data = dde.data.TimePDE(
    geomtime,
```

```
    heat_equation,
    [ic, left_bc, right_bc, up_low_bc],
    num_domain=pts_dom,
    num_boundary=pts_bc,
    num_initial=pts_ic
)

# ------------------------------------------------------------------------
# Sensores y espacio de funciones
# ------------------------------------------------------------------------

side = np.linspace(x_initial, x_boundary, num_sensors + 1)
x, y = np.meshgrid(side, side, indexing='xy')
sensor_pts = np.stack([x.ravel(), y.ravel()], axis=1)

fs = dde.data.function_spaces.GRF2D(N=size_cov_matrix,
                                    interp="linear")

data = dde.data.PDEOperatorCartesianProd(
    pde_data,
    fs,
    sensor_pts,
    num_function=num_function,
    function_variables=[0, 1],
    num_test=num_test
)

# ------------------------------------------------------------------------
# Definicion de la red
# ------------------------------------------------------------------------

branch_layers = [(num_sensors + 1)**2] + len_net * [width_net]
trunk_layers = [3] + len_net * [width_net]

net = dde.nn.DeepONetCartesianProd(
    branch_layers,
    trunk_layers,
    activation=AF,
    kernel_initializer=k_initializer
)

# ------------------------------------------------------------------------
# Compilacion y entrenamiento del modelo
# ------------------------------------------------------------------------

model = dde.Model(data, net)
model.compile("adam", lr=learning_rate,
            decay=("inverse time", decay_steps, decay_rate))
losshistory, train_state = model.train(iterations=num_iterations,
display_every=decay_steps)

# Refinamiento con el optimizador L-BFGS
model.compile("L-BFGS")
losshistory, train_state = model.train()
```

13.1. Gráficas de pérdida del modelo

El proceso de entrenamiento del modelo se monitoreó mediante el seguimiento detallado de cinco componentes de pérdida, cada una asociada a restricciones físicas y matemáticas específicas del problema:

1. Pérdida residual de la EDP
    - Función: Mide el cumplimiento de la ecuación de Bio-Calor en el dominio interior.
    - Importancia: Garantiza que la solución aprendida satisfaga la física subyacente.
    - Comportamiento esperado: Debe converger a valores cercanos a cero (típicamente $<$ 1e-3).
2. Pérdida de condición inicial
    - Función: Controla la precisión en t=0.
    - Importancia: Asegura coherencia con el estado inicial del sistema.
    - Patrón típico: Suele ser la primera en converger por su carácter puntual.
3. Pérdida de frontera izquierda (Dirichlet)
    - Función: Evalúa el cumplimiento de condiciones de valor prescrito.
    - Relevancia: Mantiene valores fijos en bordes específicos.
    - Convergencia: Normalmente rápida por ser restrictiva.
4. Pérdida de frontera derecha (Neumann)
    - Función: Verifica gradientes normales en esta frontera
    - Dificultad característica: Puede mostrar oscilaciones iniciales
5. Pérdida de fronteras superior/inferior (Neumann)
    - Función: Controla condiciones de flujo en estos bordes
    - Complejidad: En problemas 2D/3D suele ser la última en estabilizarse

13.1.1. Perdida para el conjunto de entrenamiento

La tendencia decreciente de la pérdida durante el entrenamiento evidencia que el modelo neuronal está aprendiendo progresivamente a satisfacer las restricciones físicas impuestas por la ecuación de Bio-Calor y sus condiciones asociadas. Este comportamiento indica una correcta adaptación de los parámetros de la red, permitiendo reducir consistentemente el error en las distintas componentes de la función objetivo y aproximando con mayor precisión la dinámica térmica en el dominio.

```
import plotly.graph_objects as go

# Nombres de las componentes del loss
loss_labels = [
    "Pérdida residual PDE",
    "Pérdida de condición inicial",
    "Pérdida de frontera izquierda (Dirichlet)",
    "Pérdida de frontera derecha (Neumann)",
    "Pérdida de fronteras superior/inferior (Neumann)"
]

# Extraer pasos y pérdida de entrenamiento
steps = losshistory.steps
train_loss = np.array(losshistory.loss_train)

# Crear figura
fig_train = go.Figure()

for i in range(train_loss.shape[1]):
    fig_train.add_trace(go.Scatter(
        x=steps,
        y=train_loss[:, i],
        mode='lines',
        name=loss_labels[i]
    ))

fig_train.update_layout(
    title="Historial de pérdida en el entrenamiento",
    xaxis=dict(title="Iteración", tickformat=".1e"),
    yaxis=dict(title="Pérdida (log)", type="log", tickformat=".1e"),
    template="plotly_white",
    legend=dict(x=0.99, y=0.99),
    font=dict(size=14)
)
```

13.1.2. Pérdida para el conjunto de prueba

La disminución del error en el conjunto de prueba confirma que el modelo no solo memoriza los datos de entrenamiento, sino que logra generalizar a situaciones no vistas. Esto constituye un resultado favorable, pues asegura que la red neuronal mantiene su capacidad predictiva fuera de los escenarios empleados para el ajuste, garantizando robustez y confiabilidad en aplicaciones biomédicas donde la precisión en la estimación térmica es fundamental.

```
import plotly.graph_objects as go

# Nombres de las componentes del loss
loss_labels = [
    "Pérdida residual PDE",
    "Pérdida de condición inicial",
    "Pérdida de frontera izquierda (Dirichlet)",
    "Pérdida de frontera derecha (Neumann)",
    "Pérdida de fronteras superior/inferior (Neumann)"
```

Figura 13.1.: Gráfica de la perdida en el entrenamiento.

```
]

# Extraer pasos y pérdida de entrenamiento
steps = losshistory.steps
test_loss = np.array(losshistory.loss_test)

# Crear figura
fig_test = go.Figure()

for i in range(test_loss.shape[1]):
    fig_test.add_trace(go.Scatter(
        x=steps,
        y=test_loss[:, i],
        mode='lines',
        name=loss_labels[i]
    ))

fig_test.update_layout(
    title="Historial de pérdida en el conjunto de prueba",
    xaxis=dict(title="Iteración", tickformat=".1e"),
    yaxis=dict(title="Pérdida (log)", type="log", tickformat=".1e"),
    template="plotly_white",
    legend=dict(x=0.99, y=0.99),
    font=dict(size=14)
)
```

13.2. Guardado de datos

Para permitir la comparación cuantitativa con el método de Crank-Nicolson y facilitar la generación de visualizaciones consistentes, se exportaron las predicciones del modelo neuronal en formato CSV. El proceso consistió en:

1. Generación de la malla de evaluación:
    - Dominio espacial: Cuadrado unitario [0,1] × [0,1].
    - Discretización: 26 segmentos equiespaciados en cada eje (x, y).
    - Puntos totales: 676 (26 × 26).
    - Tiempos evaluados: t = [0.0, 0.25, 0.50, 0.75, 1.0].
2. Estructura del archivo:
    - Coordenadas espacio-temporales (t, x, y) para cada punto de la grilla 26×26.
    - Valores de la solución en los tiempos de interés.

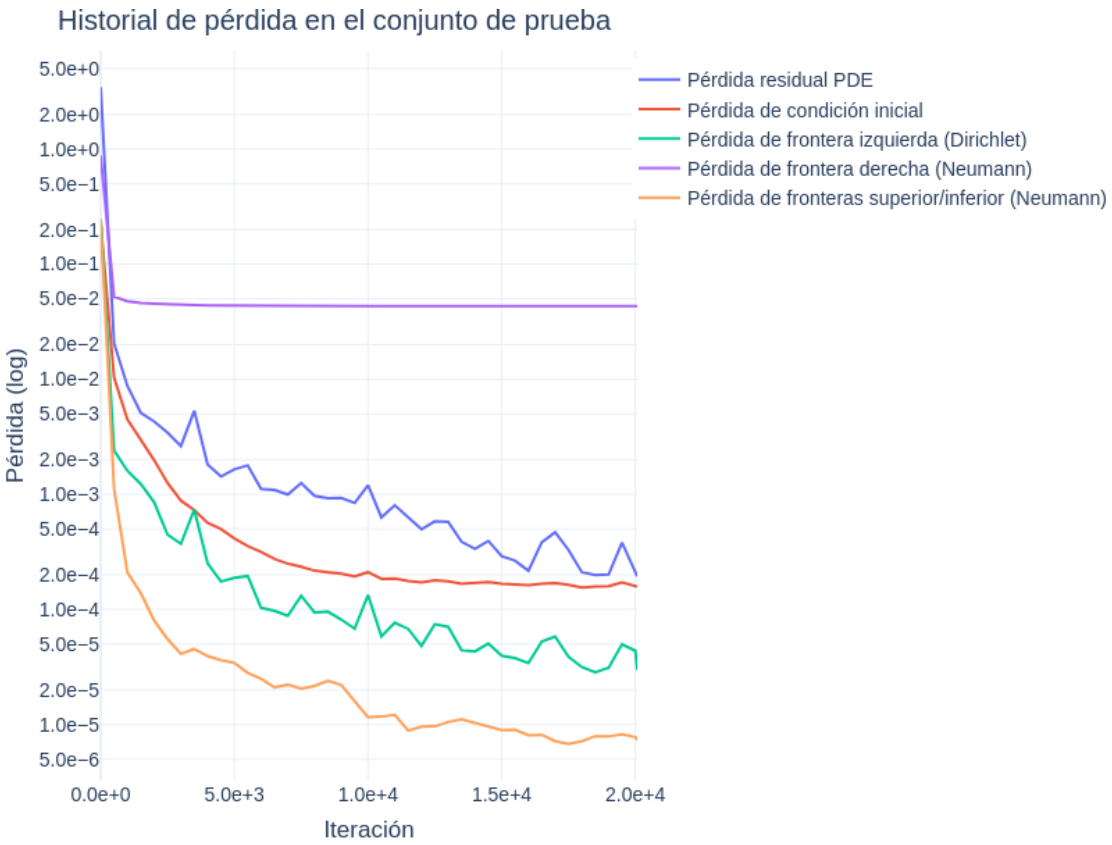


Figura 13.2.: Gráfica de la perdida en el conjunto de prueba.

Listado 13.1 Guardado de los datos de la red neuronal.

```
import pandas as pd
# Lista de tiempos
times = [0.0, 0.25, 0.5, 0.75, 1.0]

# Crear la malla (x, y)
num_points = 26
x = np.linspace(0, 1, num_points)
y = np.linspace(0, 1, num_points)
X, Y = np.meshgrid(x, y)

# Lista para almacenar resultados
results = []

for t_val in times:
    # Crear entrada trunk: (num_points^2, 3)
    points = np.vstack((X.flatten(), Y.flatten(),
                        t_val * np.ones_like(X.flatten()))).T

    # Crear entrada branch: condición inicial constante cero
    branch_input = np.zeros((1, sensor_pts.shape[0]))

    # Predecir
    predicted = model.predict((branch_input, points)).flatten()

    # Agregar los datos al resultado
    for xi, yi, thetai in zip(points[:, 0], points[:, 1], predicted):
        results.append([t_val, xi, yi, thetai])

# Crear el DataFrame
df = pd.DataFrame(results, columns=["time", "X", "Y", "Theta"])

# Obtener la ruta del script actual y guardar el archivo CSV
ruta = r"data/model_DoN.csv"
df.to_csv(ruta, index=False)
```

14. Comparación de resultados

14.1. Comparativa visual de las predicciones

Esta sección presenta un análisis cualitativo de los resultados mediante la comparación directa entre las predicciones del modelo, las soluciones reportadas en el estudio de Alessio Borgi (2023), así como las obtenidas mediante el método de Crank Nicolson y la solución analítica. La visualización paralela permite evaluar:

- Dominio espacial: Cuadrado unitario [0,1] × [0,1] con malla 26×26.
- Escala de colores: Mapa térmico YlGnBu y viridis (consistente en sus respectivos gráficos).

14.1.1. Modelo contra resultados de Alessio Borgi (2023)

El siguiente código grafica las predicciones de la red neuronal DeepONet para la temperatura dadas las coordenadas espaciales en los distintos tiempos de interés, los ejes están adimencionalizados siguiendo las ecuaciones 9.2.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
import pandas as pd
import numpy as np

# Lista de tiempos
times = [0.0, 0.25, 0.5, 0.75, 1.0]

# Cargar el dataframe
df = pd.read_csv(r'data/model_DoN.csv')

# Crear figura con subplots 3D en 1 fila y 5 columnas
fig, axes = plt.subplots(nrows=1, ncols=len(times),
                         figsize=(28, 7),
                         subplot_kw={'projection': '3d'})

plt.subplots_adjust(right=0.8)

# Asumimos que el grid es regular
num_points = int(np.sqrt(df[df["time"] == times[0]].shape[0]))

# Lista para almacenar los objetos surface
surf_list = []
```

```
# Reordenar para graficar
for i, (t_val, ax) in enumerate(zip(times, axes)):
    # Filtrar por tiempo actual
    df_t = df[df["time"] == t_val]

    # Obtener los valores de X, Y, Theta
    X_vals = df_t["X"].values.reshape((num_points, num_points))
    Y_vals = df_t["Y"].values.reshape((num_points, num_points))
    Z_vals = df_t["Theta"].values.reshape((num_points, num_points))

    # Dibujar la superficie
    surf = ax.plot_surface(
        Y_vals, X_vals, Z_vals,
        rstride=1, cstride=1,
        cmap="YlGnBu",
        edgecolor="none",
        antialiased=True
    )
    surf_list.append(surf)

    ax.set_title(f"Time = {t_val:.2f}", pad=10)
    ax.set_xlabel("Y", labelpad=10)
    ax.set_ylabel("X", labelpad=10)
    ax.set_zlabel("T [ ]", labelpad=10, rotation=90)
    ax.set_box_aspect(None, zoom=0.75)

# Añadir barra de color común
cbar = fig.colorbar(surf_list[-1], ax=axes,
                    shrink=0.9, aspect=90,
                    pad=0.1, orientation='horizontal')
cbar.set_label('Temperatura [ ]')

plt.show()
```

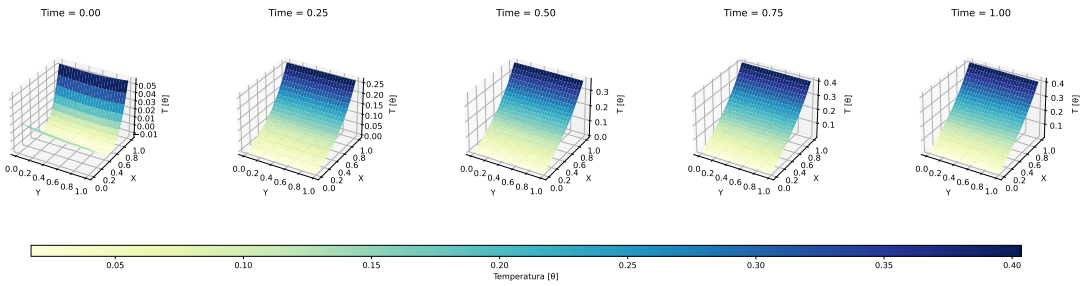


Figura 14.1.: Predicciones de la red neuronal a distintos tiempos.

La comparación visual entre las predicciones del modelo DeepONet Figura 14.1 y los resultados de , Figura 14.2 revela una notable similitud en la evolución temporal y espacial de la temperatura. Ambos modelos capturan la misma tendencia de calentamiento progresivo, con un gradiente térmico que se intensifica cerca de la frontera derecha ($x = 1$), donde se aplica una condición de Neumann no homogénea. Sin embargo, se observa que en $t = 0$, el modelo DeepONet se aproxima con mayor fidelidad al plano

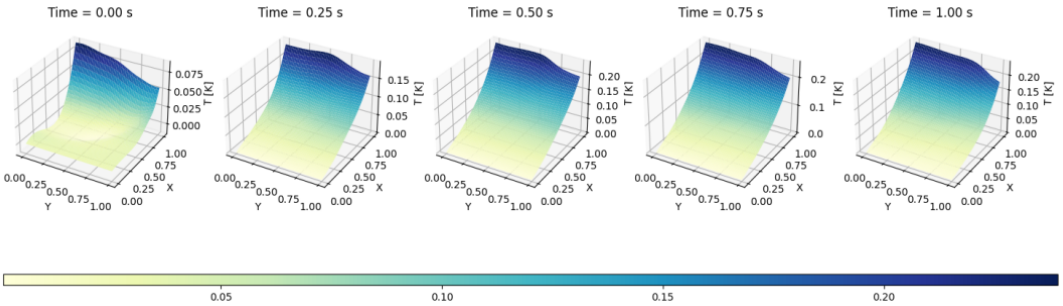


Figura 14.2.: Resultados reportados por Alessio Borgi (2023) en el caso 2D.

$XY = 0$, lo que sugiere una mejor captura de la condición inicial en comparación con el trabajo de referencia.

14.1.2. Modelo contra método numérico

La siguiente gráfica utiliza los archivos csv obtenidos de los códigos 13.1 y 12.1 para motrar una comparativa entre los tiempos de interés, las predicciones del modelo se encuentran en la parte superior mientras que las del método numérico en la parte inferior.

```
crank_nick_data = pd.read_csv(r'data/crank_nick.csv')
model_don_data = pd.read_csv(r'data/model_DoN.csv')

# Determinar los límites comunes para el colorbar
min_temp = min(model_don_data['Theta'].min(),
               crank_nick_data['Theta'].min())
max_temp = max(model_don_data['Theta'].max(),
               crank_nick_data['Theta'].max())

# Crear figura con subplots 3D en 2 filas y 5 columnas
fig = plt.figure(figsize=(22, 12))
axes = []

# Crear los subplots
for i in range(2):  # 2 filas
    for j in range(5):  # 5 columnas
        axes.append(fig.add_subplot(2, 5, i*5 + j + 1, projection='3d'))

axes = np.array(axes).reshape(2, 5)

# Añadir títulos generales para cada fila
fig.text(0.5, 0.92, "Predicciones modelo DON",
         ha='center', va='center', fontsize=14,fontweight='bold')
fig.text(0.5, 0.58, "Predicciones método numérico",
         ha='center', va='center', fontsize=14, fontweight='bold')

# Función para graficar un dataframe en una fila específica
def plot_dataframe(df, row, num_points, cmap="viridis"):
    surf_list = []
    for col, t_val in enumerate(times):
```

```
        ax = axes[row, col]

        # Filtrar por tiempo actual
        df_t = df[df["time"] == t_val]

        # Obtener los valores de X, Y, Theta
        X_vals = df_t["X"].values.reshape((num_points, num_points))
        Y_vals = df_t["Y"].values.reshape((num_points, num_points))
        Z_vals = df_t["Theta"].values.reshape((num_points, num_points))

        # Dibujar la superficie con límites comunes
        surf = ax.plot_surface(
            Y_vals, X_vals, Z_vals,
            rstride=1, cstride=1,
            cmap=cmap,
            edgecolor="none",
            antialiased=True,
            vmin=min_temp,
            vmax=max_temp
        )
        surf_list.append(surf)

        ax.set_title(f"Time = {t_val:.2f}", pad=10)
        ax.set_xlabel("Y", labelpad=10)
        ax.set_ylabel("X", labelpad=10)
        ax.set_zlabel("T [ ]", labelpad=10, rotation=90)
        ax.set_box_aspect(None, zoom=0.75)

    return surf_list

# Asumimos que el grid es regular para ambos dataframes
num_points = int(np.sqrt(
        model_don_data[model_don_data["time"] == times[0]].shape[0]))

# Graficar el primer dataframe en la fila superior
surf_model_don = plot_dataframe(model_don_data, 0, num_points)

# Graficar el segundo dataframe en la fila inferior
surf_crank_nick = plot_dataframe(crank_nick_data, 1, num_points)

# Añadir barra de color común en la parte inferior
cbar = fig.colorbar(surf_crank_nick[-1], ax=axes.ravel().tolist(),
                    use_gridspec=True, orientation='horizontal',
                    pad=0.05, aspect=90, shrink=0.9)
cbar.set_label('Temperatura [ ]', labelpad=10)


plt.show()
```

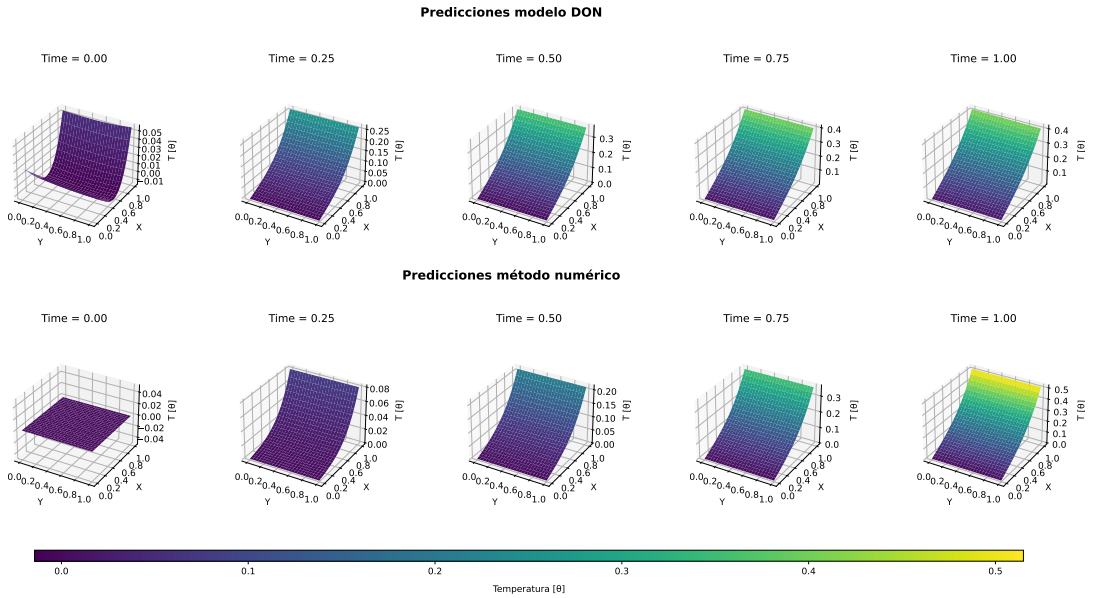


Figura 14.3.: Contraste de las predicciones entre el modelo y el método de Crank Nicolson para cada tiempo. Se aprecia que ambas comparten forma y tendencia, sin embargo a medida que el tiempo se acerca a t=1 los resultados divergen.

Al contrastar las predicciones del modelo con las obtenidas mediante el método de Crank-Nicolson Figura 14.3, se confirma que ambas soluciones comparten la misma estructura general y comportamiento temporal. No obstante, a medida que el tiempo avanza hacia $t = 1$, se aprecia una ligera divergencia en la magnitud de la temperatura, especialmente en la región cercana a $x = 1$, donde el gradiente impuesto introduce mayor sensibilidad numérica.

14.1.3. Modelo contra solución analítica

La siguiente gráfica utiliza los archivos csv obtenidos de los códigos 13.1 y 9.1 para mostrar una comparativa entre los tiempos de interés, las predicciones del modelo se encuentran en la parte superior mientras que las de la solución analítica en la parte inferior.

```
sol_ana_data = pd.read_csv(r'data/sol_analitica.csv')
model_don_data = pd.read_csv(r'data/model_DoN.csv')

# Determinar los límites comunes para el colorbar
min_temp = min(model_don_data['Theta'].min(),
               sol_ana_data['Theta'].min())
max_temp = max(model_don_data['Theta'].max(),
               sol_ana_data['Theta'].max())
```

```python
# Crear figura con subplots 3D en 2 filas y 5 columnas
fig = plt.figure(figsize=(22, 12))
axes = []

# Crear los subplots
for i in range(2):  # 2 filas
    for j in range(5):  # 5 columnas
        axes.append(fig.add_subplot(2, 5, i*5 + j + 1, projection='3d'))

axes = np.array(axes).reshape(2, 5)

# Añadir títulos generales para cada fila
fig.text(0.5, 0.92, "Predicciones modelo DON",
        ha='center', va='center', fontsize=14,fontweight='bold')
fig.text(0.5, 0.58, "Predicciones de la Sol. analítica",
        ha='center', va='center', fontsize=14, fontweight='bold')

# Función para graficar un dataframe en una fila específica
def plot_dataframe(df, row, num_points, cmap="viridis"):
    surf_list = []
    for col, t_val in enumerate(times):
        ax = axes[row, col]

        # Filtrar por tiempo actual
        df_t = df[df["time"] == t_val]

        # Obtener los valores de X, Y, Theta
        X_vals = df_t["X"].values.reshape((num_points, num_points))
        Y_vals = df_t["Y"].values.reshape((num_points, num_points))
        Z_vals = df_t["Theta"].values.reshape((num_points, num_points))

        # Dibujar la superficie con límites comunes
        surf = ax.plot_surface(
            Y_vals, X_vals, Z_vals,
            rstride=1, cstride=1,
            cmap=cmap,
            edgecolor="none",
            antialiased=True,
            vmin=min_temp,
            vmax=max_temp
        )
        surf_list.append(surf)

        ax.set_title(f"Time = {t_val:.2f}", pad=10)
        ax.set_xlabel("Y", labelpad=10)
        ax.set_ylabel("X", labelpad=10)
        ax.set_zlabel("T [ ]", labelpad=10, rotation=90)
        ax.set_box_aspect(None, zoom=0.75)

    return surf_list

# Asumimos que el grid es regular para ambos dataframes
num_points = int(np.sqrt(
        model_don_data[model_don_data["time"] == times[0]].shape[0]))

# Graficar el primer dataframe en la fila superior
surf_model_don = plot_dataframe(model_don_data, 0, num_points)

# Graficar el segundo dataframe en la fila inferior
surf_crank_nick = plot_dataframe(sol_ana_data, 1, num_points)

# Añadir barra de color común en la parte inferior
```

```
cbar = fig.colorbar(surf_crank_nick[-1], ax=axes.ravel().tolist(),
                    use_gridspec=True, orientation='horizontal',
                    pad=0.05, aspect=90, shrink=0.9)
cbar.set_label('Temperatura [ ]', labelpad=10)


plt.show()
```

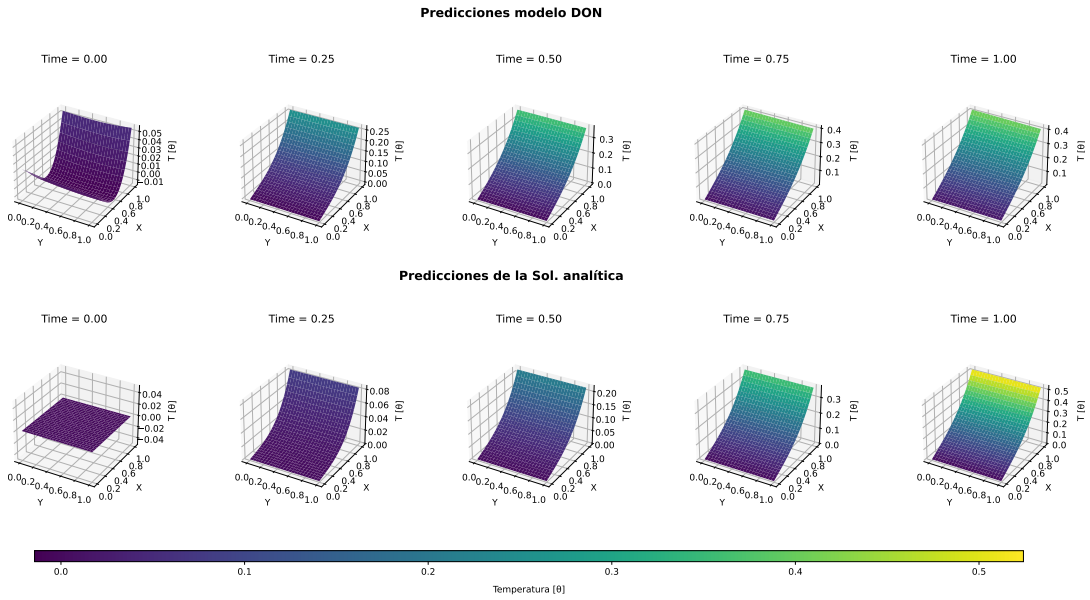


Figura 14.4.: Contraste de las predicciones entre el modelo y la solución analítica para cada tiempo. Se aprecia que ambas comparten forma y tendencia, sin embargo a medida que el tiempo se acerca a t=1 los resultados divergen.

Finalmente, la comparación con la solución analítica Figura 14.4 refuerza la validez del modelo DeepONet. Aunque la forma general de la solución es consistentemente recuperada, se observa que las discrepancias aumentan levemente con el tiempo, lo cual es esperable dada la naturaleza truncada de la solución analítica y la aproximación inherente de la red neuronal.

14.2. Validaciones cuantitativas

14.2.1. Modelo contra el método de Crank-Nicolson

Para evaluar numéricamente la precisión del modelo DeepONet, se realizó una comparación sistemática con soluciones de referencia generadas mediante el método de Crank-Nicolson. Este enfoque proporciona una métrica objetiva de la exactitud del

modelo, siendo complementado con una serie de gráficos que muestran el error absoluto para cada punto del dominio en los tiempos de interés.

```
# Función para calcular errores
def calculate_errors(true_data, pred_data, times):
    results = []

    for time in times:
        # Filtrar datos por tiempo
        true_subset = true_data[true_data['time'] == time]
        pred_subset = pred_data[pred_data['time'] == time]

        if len(true_subset) == 0 or len(pred_subset) == 0:
            print(f"Advertencia: No hay datos para tiempo t={time}")
            continue

        # Verificar que las dimensiones coincidan
        if len(true_subset) != len(pred_subset):
            print(f"Advertencia:Num de puntos no coincide para t={time}")
            min_len = min(len(true_subset), len(pred_subset))
            true_subset = true_subset.iloc[:min_len]
            pred_subset = pred_subset.iloc[:min_len]

        # Calcular errores para Theta
        theta_true = true_subset['Theta'].values
        theta_pred = pred_subset['Theta'].values

        absolute_error = np.abs(theta_true - theta_pred)
        l2_error = np.sqrt(np.sum((theta_true - theta_pred)**2))

        results.append({
            'time': time,
            'mean_absolute_error': np.mean(absolute_error),
            'max_absolute_error': np.max(absolute_error),
            'l2_error': l2_error
        })

    return pd.DataFrame(results)

# Calcular errores
error_results = calculate_errors(crank_nick_data, model_don_data, times)

# Guardar resultados
error_results.to_csv("data/error_crank_nic.csv", index=False)
```

Tabla 14.1.: Desviación del modelo DeepONet respecto a Crank-Nicolson.

Tiempo	MAE	MaxAE	Error L2
0.000	0.013	0.055	0.457
0.250	0.067	0.182	2.250
0.500	0.070	0.156	2.205
0.750	0.035	0.052	0.992
1.000	0.028	0.107	1.040

Las métricas de error calculadas —Error Absoluto Medio (MAE), Error Absoluto Máximo (MaxAE) y Error L2— confirman el buen desempeño del modelo DeepONet. En la comparación con el método de Crank-Nicolson Tabla 14.1, el MAE se mantuvo entre 0.013 y 0.07, con un valor máximo de 0.182 en el MaxAE. Estos valores reflejan una aproximación satisfactoria, aunque se observa que los errores tienden a aumentar en tiempos intermedios ($t = 0.25$ y $t = 0.5$), posiblemente debido a la mayor complejidad dinámica en esas etapas.

14.2.1.1. Gráficas de error absoluto

```
# Calcular el error absoluto entre los dos dataframes
error_data = model_don_data.copy()
error_data['error'] = np.abs(
                    crank_nick_data['Theta'] - model_don_data['Theta'])

# Crear figura con 3 filas y 2 columnas
fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(22, 8))
axes = axes.ravel()  # Convertir a array 1D para fácil acceso

# Asumir que el grid es regular
num_points = int(np.sqrt(
                    error_data[error_data["time"] == times[0]].shape[0]
                    ))

# Configuración común para los mapas de calor
plot_kwargs = {
    'cmap': 'hot_r',
    'shading': 'auto',
    'vmin': error_data['error'].min(),
    'vmax': error_data['error'].max()
}
# Lista para guardar los gráficos
abs_errors_pc = []

# Crear los subplots
for i, t_val in enumerate(times):
    ax = axes[i]

    # Filtrar por tiempo actual
    df_t = error_data[error_data["time"] == t_val]

    # Obtener valores y reshape
    X_vals = df_t["X"].values.reshape((num_points, num_points))
    Y_vals = df_t["Y"].values.reshape((num_points, num_points))
    error_vals = df_t["error"].values.reshape((num_points, num_points))

    # Crear mapa de calor
    pc = ax.pcolormesh(X_vals, Y_vals, error_vals, **plot_kwargs)

    # Configuración de ejes
    ax.set_title(f"Tiempo = {t_val:.2f}", pad=10)
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_aspect('equal')
```

```
    abs_errors_pc.append(pc)

cbar = fig.colorbar(abs_errors_pc[-1], ax=axes,
                    use_gridspec=True, shrink=0.9,
                    aspect=90, pad=0.1, orientation='horizontal')
cbar.set_label('Error absoluto [ ]')

# Mostrar el gráfico
plt.show()
```

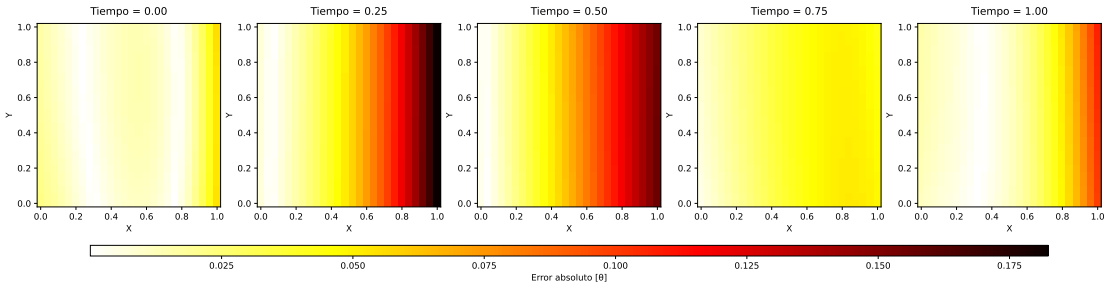


Figura 14.5.: Errores absolutos entre el modelo y el método de Crank Nicolson para cada tiempo.

14.2.2. Modelo contra la solución analítica

De manera análoga a la sección anterior se realizó una comparativa contra la solución analítica (Sección 9.3). De este modo se tiene una visión más completa acerca del rendimiento del modelo.

```
# Función para calcular errores
def calculate_errors(true_data, pred_data, times):
    results = []

    for time in times:
        # Filtrar datos por tiempo
        true_subset = true_data[true_data['time'] == time]
        pred_subset = pred_data[pred_data['time'] == time]

        if len(true_subset) == 0 or len(pred_subset) == 0:
            print(f"Advertencia: No hay datos para tiempo t={time}")
            continue

        # Verificar que las dimensiones coincidan
        if len(true_subset) != len(pred_subset):
            print(f"Advertencia:Num de puntos no coincide para t={time}")
            min_len = min(len(true_subset), len(pred_subset))
            true_subset = true_subset.iloc[:min_len]
            pred_subset = pred_subset.iloc[:min_len]

        # Calcular errores para Theta
        theta_true = true_subset['Theta'].values
```

```
        theta_pred = pred_subset['Theta'].values

        absolute_error = np.abs(theta_true - theta_pred)
        l2_error = np.sqrt(np.sum((theta_true - theta_pred)**2))

        results.append({
            'time': time,
            'mean_absolute_error': np.mean(absolute_error),
            'max_absolute_error': np.max(absolute_error),
            'l2_error': l2_error
        })

    return pd.DataFrame(results)

# Calcular errores
error_results = calculate_errors(sol_ana_data, model_don_data, times)

# Guardar resultados
error_results.to_csv("data/error_ana.csv", index=False)
```

Tabla 14.2.: Error del modelo DeepONet respecto a la solución analítica.

Tiempo	MAE	MaxAE	Error L2
0.000	0.013	0.055	0.457
0.250	0.067	0.180	2.221
0.500	0.068	0.151	2.143
0.750	0.032	0.047	0.905
1.000	0.031	0.116	1.153

Al comparar con la solución analítica Tabla 14.2, los errores son consistentemente bajos, con un MAE máximo de 0.068 y un MaxAE de 0.18. La similitud entre ambas tablas sugiere que el método de Crank-Nicolson y la solución analítica están bien alineados, y que el modelo DeepONet se aproxima a ambos con un nivel de error comparable.

14.2.2.1. Gráficas de error absoluto

```
# Calcular el error absoluto entre los dos dataframes
error_data = model_don_data.copy()
error_data['error'] = np.abs(
                    sol_ana_data['Theta'] - model_don_data['Theta'])

# Crear figura con 3 filas y 2 columnas
fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(22, 8))
axes = axes.ravel()  # Convertir a array 1D para fácil acceso

# Asumir que el grid es regular
num_points = int(np.sqrt(
                    error_data[error_data["time"] == times[0]].shape[0]
```

```
                    ))

# Configuración común para los mapas de calor
plot_kwargs = {
    'cmap': 'hot_r',
    'shading': 'auto',
    'vmin': error_data['error'].min(),
    'vmax': error_data['error'].max()
}
# Lista para guardar los gráficos
abs_errors_pc = []

# Crear los subplots
for i, t_val in enumerate(times):
    ax = axes[i]

    # Filtrar por tiempo actual
    df_t = error_data[error_data["time"] == t_val]

    # Obtener valores y reshape
    X_vals = df_t["X"].values.reshape((num_points, num_points))
    Y_vals = df_t["Y"].values.reshape((num_points, num_points))
    error_vals = df_t["error"].values.reshape((num_points, num_points))

    # Crear mapa de calor
    pc = ax.pcolormesh(X_vals, Y_vals, error_vals, **plot_kwargs)

    # Configuración de ejes
    ax.set_title(f"Tiempo = {t_val:.2f}", pad=10)
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_aspect('equal')

    abs_errors_pc.append(pc)

cbar = fig.colorbar(abs_errors_pc[-1], ax=axes,
                    use_gridspec=True, shrink=0.9,
                    aspect=90, pad=0.1, orientation='horizontal')
cbar.set_label('Error absoluto [ ]')

# Mostrar el gráfico
plt.show()
```

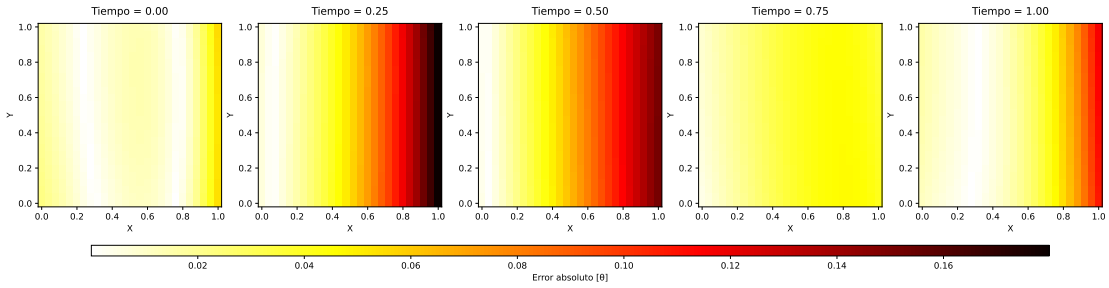


Figura 14.6.: Errores absolutos entre el modelo y la solución analítica para cada tiempo.

Los mapas de error absoluto Figura 14.5 y Figura 14.6 permiten localizar espacialmente

las discrepancias. Se observa que los mayores errores se concentran en la región de $x = 1$, donde la condición de Neumann no homogénea introduce mayores exigencias en la aproximación. Esta distribución del error es coherente con el comportamiento reportado en la literatura para problemas con condiciones de frontera variables en el tiempo.

15. Conclusiones

El presente trabajo ha abordado la complejidad de resolver una ecuación diferencial parcial dependiente del tiempo en dos dimensiones espaciales a través de una red neuronal con la arquitectura DeepONet, asimismo se obtuvieron predicciones para el cuadrado de $[0, 1] \times [0, 1]$. Los hiperparámetros de la red se fueron variando para obtener la mejor configuración, usando como base los resultados obtenidos por Alessio Borgi (2023). Los resultados obtenidos mediante la comparación con el método de Crank Nicolson demostraron que la red neuronal DeepONet se aproxima eficientemente, pues el MAE se mantuvo entre 1.3% y 7%, con un valor máximo de 18.2% en el MaxAE Tabla 14.1. Complementando a los resultados previos, al comparar con la solución analítica, el MAE máximo fué de 6.8% y un MaxAE de 18% Tabla 14.2.

Los errores obtenidos demuestran la eficacia del modelo para converger a la condición inicial, pues tal como se aprecia en las figuras 14.5 y 14.6, a medida que la ecuación evoluciona en el tiempo, las predicciones entre el método de Crank Nicolson y la red neuronal divergen, esto es conforme evoluciona la función, vemos que cada vez se aleja más del valor real. Valdría la pena en otro trabajo comparar la solución con otro método númerico clásico para observar si se presenta el mismo comportamiento.

Lo anterior evidencia el potencial que tiene las PINNs como herramienta auxiliar en la solución de ecuaciones diferenciales parciales, pues solo a través de la definición de la geometria y el espacio temporal (si es necesario) junto con algunos puntos en el dominio y las condiciones iniciales y de frontera probarón predecir de forma muy acertada el conjunto de prueba. Una situación que es común en el ámbito científico, es la de no siempre contar con una base de datos extensa y libre de ruido con la que entrenar a un modelo, lo que le otortga a las PINNs una gran ventaja respecto a los modelos de Deep learning que necesitan una gran cantidad de datos para poder ser entrenados (George Em Karniadakis 2021).

Complementado a las PINNs, que predicen soluciones específicas para condiciones fijas; la arquitectura DeepONet aprende operadores (mapeos entre espacios de funciones) en lugar de solo aproximar funciones, lo que le otorga la capacidad de generalizar a nuevas condiciones iniciales y de frontera sin reentrenamiento, gracias a su estructura de red dual (branch-trunk). Esto lo hace ideal para aplicaciones en tiempo real, como la *hipertermia*, donde las características del problema son suceptibles a cambios, como lo son las propiedades del cuerpo humano que varían en cada paciente. Un problema clave que se encontró es que al tener una estructura más compleja, los tiempos de entrenamiento

respectos a las PINNs son mayores, sin embargo esto se ve bien compensado por su alta capacidad de adaptabilidad a nuevas condiciones ya sean iniciales o de frontera.

La creación de éste tipo de modelos, tanto PINNs clásicas como DeepONets se puede ver obstaculizada por el conocimiento en programación del investigador o estudiante que se plantee programarlos. Si bien, tanto en el ámbito científico como en la programación el pensamiento crítico, seguimiento lógico y abstracción de los problemas son pilares fundamentales; también es necesario familiarizarse con las librerías que implementan éste tipo de modelos, además es bastante recomendado tener una noción básica de como funciona una red neuronal y las partes que la componen. Lo anterior implica una inversión de tiempo y esfuerzo por parte de los interesados, cosa que cuando se lleva a cabo un experiemento o investigación no siempre es posible. Si bien éstas herramientas son bastante fascinantes y con mucho potencial, como cualquier nueva habilidad hay que practicar su uso para obtener resultados que valgan la pena.

Cabe mencionar que las aplicaciones de las PINNs son tan amplias como lo es en sí en campo de las PDEs, si nos centramos en la *hipertermia*, la cual busca elevar la temperatura en tejidos tumorales (39-45°C), nos topamos con que predecir la distribución térmica en tiempo real es un desafío más complejo de lo que parece en un inicio, hay varias formas de caracterizar la temperatura en un cuerpo biológico, sin contar que las múltiples varibales que componen el fenómeno cambian dependiendo del individuo. Por ello, las redes neuronales, especialmente DeepONet por su capacidad de generalización, permiten aproximar la temperatura bajo distintas condiciones, optimizando la dosificación de calor y minimizando daños a tejidos sanos. Esto facilita terapias personalizadas y no invasivas, mejorando la eficacia clínica.

16. Futuros trabajos de investigación

De cara al futuro, es planteable explorar ramas alternas de la metodología usada en ciertos aspectos, como lo son: - Ajustar la configuración del optimizador *L-BFGS* ya sea aumentando o disminuyendo sus iteraciones máximas, su umbral de tolerancia o máximo número de funciones a evaluar.

- Utilizar el módulo de *callbacks* para hacer un *earlyStopping* del modelo para evitar un sobreajuste.
- Utilizar un conjunto de validación con datos reales tomados de una sesión de hipertérmia.
- Utilizar otro método numérico para comparar el modelo, como puede ser diferencias finitas.
- Comparar otras librerías en Python que implementen PINNs como lo son SimNet, PyDEns, NeuroDiffEq o SciANN.
- Comparar con otros lengujes de programación como lo es Julia, librerías que implementen PINNs como lo son NeuralPDE o ADCME.

Referencias

Alessio Borgi, Alessandro De Luca, Eugenio Bugli. 2023. «BioHeat PINNs: Temperature Estimation with Bio-Heat Equation using Physics-Informed Neural Networks». https://github.com/alessioborgi/BioHeat_PINNs/tree/main?tab=readme-ov-file#bioheat-pinns-temperature-estimation-with-bio-heat-equation-using-physics-informed-neural-networks.

Blechschmidt, Jan, y Oliver G. Ernst. 2021. «Three ways to solve partial differential equations with neural networks—A review». *GAMM-Mitteilungen* 44 (2): e202100006. https://doi.org/10.1002/gamm.202100006.

Burden, Richard L., y J. Douglas Faires. 2010. «Numerical Analysis». En *Numerical Analysis*, 9.ª ed., 259-64. Boston, USA: Brooks Cole.

Dutta, Abhijit, y Gopal Rangarajan. 2018. «Diffusion in pharmaceutical systems: modelling and applications». *Journal of Pharmacy and Pharmacology* 70 (5): 581-98. https://doi.org/10.1111/jphp.12885.

George Em Karniadakis, Lu Lu, Ioannis G. Kevrekidis. 2021. «Physics-informed machine learning». *Nature Reviews Physics* 3 (6): 422-40. https://doi.org/10.1038/s42254-021-00314-5.

Goldfarb, Donald, Yi Ren, y Achraf Bahamou. 2016. «Practical Quasi-Newton Methods for Training Deep Neural Networks». *arXiv preprint arXiv:1606.01205*. https://arxiv.org/abs/1606.01205.

Instituto Nacional del Cáncer. 2021. «¿Qué es el cáncer?» https://www.cancer.gov/espanol/cancer/naturaleza/que-es.

Kingma, Diederik P., y Jimmy Ba. 2014. «Adam: A Method for Stochastic Optimization». *arXiv preprint arXiv:1412.6980*. https://arxiv.org/abs/1412.6980.

Kumar, Varun, Somdatta Goswami, Katiana Kontolati, Michael D. Shields, y George Em Karniadakis. 2024. «Synergistic Learning with Multi-Task DeepONet for Efficient PDE Problem Solving». *arXiv preprint arXiv:2408.02198*. https://arxiv.org/abs/2408.02198.

Lu, Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, y George Em Karniadakis. 2021. «Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators». *Nature Machine Intelligence* 3 (3): 218-29. https://doi.org/10.1038/s42256-021-00302-5.

Lu, Lu, Xuhui Meng, Zhiping Mao, y George Em Karniadakis. 2021. «DeepXDE: A deep learning library for solving differential equations». *SIAM Review* 63 (1): 208-28. https://doi.org/10.1137/19M1274067.

National Cancer Institute. 2021. «Hyperthermia to Treat Cancer». https://www.cancer.gov/about-cancer/treatment/types/hyperthermia.

Organización Mundial de la Salud. 2022. «Cáncer». https://www.who.int/es/news-room/fact-sheets/detail/cancer.

Pennes, H. H. 1948. «Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm». *Journal of Applied Physiology* 1 (2): 93-122. https://doi.org/10.1152/jappl.1948.1.2.93.

Quintero, Luis A., Mauricio Peñuela, Armando Zambrano, y Edwin Rodríguez. 2017. «Optimización del proceso de preparación de soluciones madre de antibióticos en un servicio farmacéutico hospitalario». *Revista Cubana de Farmacia* 50 (2): 448-65. https://www.medigraphic.com/cgi-bin/new/resumen.cgi?IDARTICULO=75483.

Yang, Lihong, Xin Wu, Qian Wan, Jian Kong, Rui Liu, y Xiaoxi Liu. 2014. «Pharmaceutical preparation of antibiotics: a review on formulation and technique». *Asian Journal of Pharmaceutical Sciences* 9 (3): 145-53. https://doi.org/10.1016/j.ajps.2014.04.001.

Zill, Dennis G., y Michael R. Cullen. 2008. «Differential Equations with Boundary-Value Problems». En, 7.ª ed., 433-42. Belmont, CA: Cengage Learning.