UNIVERSIDAD DE MÁLAGA

ESCUELA DE INGENIERÍAS INDUSTRIALES

TRABAJO DE FIN DE GRADO

MODELADO DEL CRECIMIENTO DE GRIETAS POR FATIGA EN ALEACIONES METÁLICAS MEDIANTE MODELOS DE APRENDIZAJE AUTOMÁTICO

GRADO EN INGENIERÍA EN DISEÑO INDUSTRIAL Y DESARROLLO DEL PRODUCTO

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE MÁLAGA

Autor: Pablo Caballero de Leiva
Tutor: María Belén Moreno Morales

Departamento:
Ingeniería Civil, de Materiales y Fabricación

Áreas de conocimiento:
Ciencia de los Materiales e Ingeniería Metalúrgica

22 de junio de 2025

Resumen

Este Trabajo Fin de Grado se centra en el estudio del crecimiento de grietas por fatiga en materiales metálicos mediante técnicas de aprendizaje automático, aplicadas a datos experimentales obtenidos en ensayos sobre las aleaciones aluminio 2024-T351 y titanio Ti-6Al-4V. La fatiga constituye una de las principales causas de fallo en componentes sometidos a cargas cíclicas, y su modelado preciso es esencial para garantizar la integridad estructural, especialmente en sectores exigentes como el aeroespacial. Tradicionalmente, este fenómeno se ha abordado mediante formulaciones analíticas como la ley de Paris o la ley de Forman, las cuales requieren calibraciones específicas y presentan limitaciones en entornos experimentales complejos.

Como alternativa, en este trabajo se han implementado y comparado dos tipos de redes neuronales artificiales: una red multicapa (MLP) y una red de base radial (RBFN), con el objetivo de predecir la velocidad de propagación de grietas a partir de variables físicas del ensayo, sin necesidad de recurrir a modelos explícitos. Ambas arquitecturas fueron entrenadas y validadas sobre subconjuntos de datos diferenciados por aleación, espesor y condiciones de carga.

Los resultados obtenidos indican que la red MLP ofrece un mejor equilibrio entre precisión, robustez y capacidad de generalización frente a nuevas condiciones, especialmente en zonas no lineales del dominio. Por su parte, la RBFN presentó un comportamiento competitivo en ciertos escenarios, con un ajuste rápido y eficaz cuando se dispone de una estructura de datos bien definida. En conjunto, este estudio demuestra que el uso de redes neuronales, y particularmente del modelo MLP, constituye una herramienta eficaz y complementaria a las leyes clásicas en la predicción del comportamiento por fatiga, contribuyendo a mejorar la evaluación y el diseño estructural basado en datos experimentales.

Palabras clave

Crecimiento de grietas por fatiga, aprendizaje automático, redes neuronales, perceptrón multicapa, red de base radial, titanio Ti-6Al-4V, aluminio 2024-T351.

Abstract

This Final Degree Project focuses on the study of fatigue crack growth in metallic materials through the application of machine learning techniques to experimental data obtained from tests on aluminum alloy 2024-T351 and titanium alloy Ti-6Al-4V. Fatigue is one of the leading causes of failure in components subjected to cyclic loading, and accurate modeling of this phenomenon is essential for ensuring structural integrity, particularly in critical sectors such as aerospace. Traditionally, fatigue crack growth has been addressed using analytical formulations such as the Paris law or the Forman law, which require specific parameter calibration and present limitations in complex or highly variable experimental conditions.

As an alternative, this work implements and compares two types of artificial neural networks: a multilayer perceptron (MLP) and a radial basis function network (RBFN), aiming to predict crack growth rate based on test conditions without relying on explicit mathematical models. Both architectures were trained and validated on representative subsets of experimental data, categorized by alloy, specimen thickness, and loading configuration.

The results show that the MLP model offers a better balance of accuracy, robustness, and generalization capability under varying conditions, particularly in nonlinear regions of the input domain. The RBFN, meanwhile, demonstrated competitive performance in specific scenarios, providing fast and effective fitting when the data structure is well defined. Overall, this study demonstrates that neural networks—especially the MLP—are effective and complementary tools to classical fatigue laws in predicting fatigue behavior, contributing to improved structural evaluation and data-driven design.

Keywords

Fatigue crack growth, machine learning, neural networks, multilayer perceptron, radial basis function network, titanium Ti-6Al-4V, aluminum 2024-T351.

Índice general

1. Introducción

1.1. Antecedentes

La caracterización del crecimiento de grietas por fatiga continúa siendo un tema de interés en la evaluación de la integridad estructural de componentes mecánicos. Conocer la evolución de una grieta permite trabajar con componentes defectuosos, planificando revisiones periódicas. Desde la década de 1960, se han desarrollado numerosos modelos fenomenológicos para describir la caracterización del crecimiento de grietas por fatiga, como la Ley de Paris, la ecuación de Forman o los modelos de NASGRO, aunque no existe un consenso sobre cuál es el más adecuado. Entre los inconvenientes de estos modelos destacan las inconsistencias dimensionales, el alto número de parámetros y la dificultad de ajuste con datos experimentales. Además, estos modelos a menudo no tienen en cuenta los efectos no lineales como el cierre de grietas, la influencia de la relación de carga o las características microestructurales.

Por otro lado, en los últimos años la inteligencia artificial ha experimentado un gran auge, transformando completamente nuestra sociedad y forma de vivir. Desde la aparición de la inteligencia artificial en 1950, no se había producido un incremento notable en la cantidad de publicaciones al respecto hasta 2013, cuando el número de artículos y citas empieza a crecer significativamente. La reciente popularidad del Aprendizaje automático se debe a la creciente disponibilidad de grandes volúmenes de datos y el aumento de la capacidad de procesamiento computacional. Durante la última década hemos visto como la inteligencia artificial permea en nuestro día a día, desde modelos generativos de lenguaje hasta algoritmos de recomendación en redes sociales.

El aprendizaje automático es un tipo de inteligencia artificial que se enfoca en la mejora de toma de decisiones de las maquinas, haciéndolas capaces de predecir a partir de datos con los que no habían sido entrenadas previamente. Los modelos de aprendizaje automático han demostrado su capacidad de reconocer patrones en grandes volúmenes de datos, siendo capaces de generalizar comportamientos complejos y realizar predicciones precisas en contextos variados.

Debido a la complejidad y no linealidad del crecimiento de grietas por fatiga, el uso de algoritmos de aprendizaje automático ofrece una alternativa muy prometedora frente a los modelos analíticos tradicionales. Usando conjuntos de datos obtenidos experimentalmente, estos algoritmos pueden ajustarse con mayor precisión a los comportamientos reales observados en materiales.

1.2. Objetivos

- **Aplicar modelos de aprendizaje automático al crecimiento de grietas por fatiga.** El objetivo principal es explorar la capacidad predictiva de algoritmos como el Perceptrón Multicapa y la Red de Base Radial para modelar la velocidad de propagación de grietas en aleaciones metálicas a partir de datos experimentales, superando las limitaciones de los modelos tradicionales.
- **Diseñar, entrenar y validar redes neuronales con datos experimentales.** Se busca construir modelos robustos utilizando técnicas de preprocesado, normalización, validación cruzada y ajuste de hiperparámetros para garantizar su fiabilidad y capacidad de generalización.
- **Comparar el rendimiento de los modelos frente a métodos clásicos.** A través de métricas cuantitativas (como el error cuadrático medio y el coeficiente de determinación), se evaluará si los modelos propuestos ofrecen una mejora significativa respecto a formulaciones analíticas como la Ley de Paris.

2. Fatiga

2.1. Introducción

El fenómeno de la fatiga es uno de los principales responsables del fallo en componentes mecánicos sometidos a cargas repetidas. Aunque las tensiones aplicadas no superen el *límite elástico* del material, la acumulación de daño a lo largo del tiempo puede dar lugar a la aparición y progresivo crecimiento de *grietas* que, si no se detectan o controlan, terminan provocando la fractura del componente. En la práctica ingenieril, este tipo de fallo es especialmente problemático porque ocurre sin previo aviso y en muchas ocasiones bajo condiciones de carga que se consideran perfectamente seguras desde un punto de vista estático. Se estima, que entre el 50 % y el 90 % de los fallos en mecanicos están relacionados con fenómenos de [1], lo que pone de manifiesto su importancia tanto desde el punto de vista técnico como económico y de la seguridad.

En general, el proceso de fallo por fatiga se divide en dos etapas bien diferenciadas: la nucleación de la grieta y su posterior propagación. La primera suele tener lugar en zonas concretas del material, normalmente en la superficie, donde existen *concentradores de tensiones*, defectos microestructurales o acabados superficiales inadecuados. La segunda etapa comienza una vez que la grieta ha alcanzado un tamaño tal que su evolución puede ser descrita mediante modelos basados en la Mecánica de la Fractura Elástica Lineal (MFEL), siempre que se cumplan ciertos requisitos, como la condición de *plasticidad a pequeña escala*, el comportamiento *elástico lineal* del material y la *suficiente separación entre la zona plástica y los bordes del componente*. Además, es necesario que el tamaño de la grieta sea grande frente a la microestructura del material, de forma que se pueda considerar un medio homogéneo y continuo. A partir de ese momento, el crecimiento de la grieta sigue un patrón relativamente predecible, influido por la *amplitud de las cargas aplicadas*, la *frecuencia de carga*, el tipo de material y las condiciones ambientales.

Dentro de este contexto, uno de los modelos más empleados para describir la propagación de grietas bajo cargas cíclicas es la ley de Paris, que relaciona la velocidad de crecimiento de la grieta por ciclo de carga (da/dN) con el rango del *factor de intensidad de tensiones* (ΔK). Esta formulación, pese a su simplicidad, ha demostrado ser extremadamente útil para predecir la evolución del daño en una gran variedad de materiales y geometrías. Sin embargo, también presenta limitaciones, especialmente cuando se trata de capturar fenómenos como el *cierre de grieta*, el *efecto de la relación de carga* o el *comportamiento de grietas pequeñas*, entre otros.

2.2. El fallo por fatiga

La fatiga es un proceso de degradación progresiva que ocurre en los materiales cuando se someten a *cargas cíclicas*, incluso si estas se mantienen por debajo del *límite elástico*. A diferencia del fallo por sobrecarga estática, que se manifiesta de forma

inmediata al superar una resistencia material, la fatiga actúa de forma acumulativa, generando daño localizado que se desarrolla con el tiempo y que puede culminar en una fractura completa del componente sin señales previas evidentes.

El mecanismo básico de la fatiga implica la aplicación repetida de una carga variable en el tiempo, lo que produce una oscilación de tensiones en determinados puntos del material. Esta oscilación genera *microdeformaciones* que se concentran en zonas concretas, como *concentradores de tensiones* o irregularidades superficiales. Con cada ciclo, estas microzonas sufren una degradación que, eventualmente, conduce a la formación de una grieta. Una vez iniciada, dicha grieta tiende a crecer perpendicularmente a la dirección principal de la carga, siguiendo la trayectoria de *máxima tensión de tracción*.

La *morfología* de una fractura por fatiga presenta características distintivas que permiten identificar visualmente el proceso. En una superficie de rotura típica se distinguen dos zonas claramente diferenciadas: una primera región relativamente lisa y pulida, donde la grieta ha crecido de forma progresiva, y una segunda región, más rugosa y de aspecto frágil, correspondiente al fallo final del componente. En la zona de crecimiento estable suelen observarse marcas concéntricas conocidas como *marcas de playa* o *beach marks*, que reflejan la evolución escalonada del frente de grieta bajo la acción de los ciclos. A escalas mayores de aumento, pueden identificarse *estriaciones*, que representan el avance de la grieta en cada ciclo de carga, aunque su visibilidad depende del material y del tipo de fatiga.

Existen distintos regímenes de fatiga según la magnitud y frecuencia de las cargas. Se habla de fatiga de alto número de ciclos (*High Cycle Fatigue*, HCF) cuando el fallo se produce tras más de 10^4 o 10^5 ciclos, habitualmente bajo tensiones moderadas y en el régimen elástico. En cambio, la fatiga de bajo número de ciclos (*Low Cycle Fatigue*, LCF) ocurre cuando las tensiones aplicadas provocan *deformaciones plásticas* apreciables, generalmente por encima del límite elástico, y el número de ciclos hasta el fallo es menor. En estos casos, la *acumulación de daño plástico por ciclo* juega un papel clave en el proceso de degradación.

Además de estos regímenes clásicos, existen otras variantes del fenómeno que dependen de las condiciones ambientales o de carga. La *corrosión-fatiga* combina la acción cíclica con un entorno químicamente agresivo, lo que acelera tanto la iniciación como el crecimiento de las grietas. La fatiga térmica se produce cuando el componente está sometido a gradientes de temperatura que inducen *tensiones térmicas cíclicas*, incluso sin necesidad de una carga mecánica externa. Por otro lado, la *fretting-fatigue* aparece en zonas de contacto donde existen pequeños deslizamientos entre superficies, lo que genera *desgaste*, *oxidación* y concentraciones locales de tensiones que favorecen la formación de grietas.

Todos estos modos de fallo comparten un rasgo común: la sensibilidad del material a la *repetición de cargas*. Por ello, el análisis del comportamiento en fatiga se ha convertido en una parte esencial del diseño estructural, especialmente en componentes

que deben resistir millones de ciclos a lo largo de su vida útil.

2.3. Mecánica de la fractura elástica lineal

El análisis del crecimiento de grietas en materiales metálicos sometidos a cargas cíclicas requiere herramientas que permitan cuantificar el estado de tensiones en las proximidades del frente de grieta. En este contexto, la Mecánica de la Fractura Elástica Lineal (MFEL) ha demostrado ser una de las metodologías más eficaces, siempre que se respeten sus hipótesis de aplicación. Esta teoría se basa en la suposición de que el material se comporta de forma *elástica* y que la zona afectada por la plasticidad en el vértice de la grieta es suficientemente pequeña como para no alterar el campo de tensiones previsto por la elasticidad lineal.

La MFEL proporciona un marco riguroso para describir cómo las grietas afectan al campo de tensiones en un cuerpo, y permite predecir cuándo una grieta existente crecerá de manera estable o inestable en función de parámetros geométricos, materiales y de carga. Sus fundamentos descansan en tres pilares: el criterio energético de Griffith, los mecanismos de nucleación de grietas, y la caracterización mediante el *factor de intensidad de tensiones*.

2.3.1. Teoría de Griffith

La primera formulación teórica rigurosa del crecimiento de grietas en sólidos frágiles se debe a A. A. Griffith, quien en 1921 propuso un enfoque energético para explicar por qué los materiales fallan a tensiones mucho menores que la resistencia teórica de sus enlaces atómicos. Según su modelo, una grieta se propaga cuando la *energía elástica liberada* durante su crecimiento es igual o superior a la *energía superficial necesaria* para crear nuevas superficies.

Griffith expreso matematicamente el balance energético entre la energía elástica almacenada y la energía superficial consumida. Para una grieta de longitud $2a$ en una placa sometida a tracción, obtuvo que la tensión crítica necesaria para la propagación inestable de la grieta viene dada por:

$$\sigma_c = \sqrt{\frac{2E\gamma_s}{\pi a}} \tag{2.1}$$

donde E es el módulo de Young, γ_s es la energía superficial por unidad de área, y a es la semilongitud de la grieta. Esta expresión, aunque inicialmente desarrollada para materiales frágiles como el vidrio, sirvió de punto de partida para extender la teoría de la fractura a materiales metálicos, incorporando posteriormente conceptos como la plasticidad localizada en el vértice de la grieta.

A partir de esta base, se introdujo un parámetro más general y práctico: el factor de intensidad de tensiones K. Este parámetro cuantifica la *intensidad del campo de tensiones elásticas* en el vértice de una grieta, y depende de tres elementos principales:

la *tensión nominal aplicada*, la *geometría del componente* y la *longitud de la grieta*. Para una configuración simple, como una placa infinita con una grieta central bajo tracción uniforme, K se define como:

$$K = \sigma\sqrt{\pi a} \tag{2.2}$$

El valor de K es único para cada configuración geométrica y de carga, y permite comparar diferentes situaciones de fractura. Su utilidad radica en que resume toda la influencia geométrica y de carga en un único parámetro intensificador de tensiones. Al mismo tiempo, permite definir una propiedad del material llamada tenacidad a la fractura K_c, que representa el umbral a partir del cual la grieta se propaga de forma inestable.

Sustituyendo la expresión de σ_c en la ecuación anterior, se obtiene:

$$K_c = \sigma_c\sqrt{\pi a} = \sqrt{2E\gamma_s} \tag{2.3}$$

Así, se obtiene el siguiente criterio de fallo: si el valor de K aplicado alcanza o supera el valor crítico del material K_c, la grieta se propaga de forma inestable:

$$\boxed{K \geq K_c \quad \Rightarrow \quad \text{crecimiento inestable de la grieta}} \tag{2.4}$$

Esta condición, conocida como *criterio de Irwin-Griffith*, es válida siempre que se cumplan las hipótesis de la MFEL, en particular la *plasticidad a pequeña escala*. El valor de K_c, determinado experimentalmente, se suele representar como K_{IC} en condiciones de deformación plana, lo que garantiza su independencia respecto al espesor y otras variables geométricas.

Las unidades tanto del *Factor de Intensidad de Tensiones*, como de la *Tenacidad a la Fractura* del material se expresan en $\mathrm{MPa}\sqrt{\mathrm{m}}$, lo que refleja la combinación de una magnitud tensional con la raíz de una longitud, coherente con su interpretación como parámetro de intensificación de tensiones.

2.3.2. Formación de grietas

Aunque el modelo de Griffith asume la existencia previa de una grieta, en la realidad los materiales suelen fallar por la formación y posterior propagación de grietas que no están inicialmente presentes a escala macroscópica. Este proceso de *formación*, o *nucleación*, está estrechamente relacionado con el comportamiento plástico a nivel microestructural.

En metales dúctiles, la deformación plástica se produce por el *movimiento de dislocaciones* a través de planos de deslizamiento. Cuando estas dislocaciones se acumulan en determinadas zonas, por ejemplo en la vecindad de inclusiones, bordes de grano o partículas de segunda fase, se generan *concentraciones de tensión localizadas*. Esta acumulación puede dar lugar a la formación de *microvacíos*, que al crecer y coalescer

terminan formando una grieta incipiente. Así, la formación de grietas en metales se explica por mecanismos de nucleación asociados al deslizamiento y a la interacción entre dislocaciones y microestructuras.

En condiciones de carga cíclica, este proceso puede verse amplificado por el *endurecimiento o ablandamiento cíclico* del material, la aparición de *bandas de deslizamiento persistentes* y la generación de *intrusiones y extrusiones* superficiales, todos ellos fenómenos que facilitan la iniciación de una grieta por fatiga.

2.4. Crecimiento de grietas por fatiga

Una vez iniciada una grieta en un componente sometido a cargas cíclicas, su crecimiento progresivo puede conducir al fallo del material. Por ello, comprender cómo se propaga una grieta bajo este tipo de solicitaciones resulta esencial para predecir la vida útil restante de un componente.

La magnitud que se utiliza para cuantificar el avance de la grieta en función de los ciclos de carga es la velocidad de crecimiento de grieta por ciclo da/dn, donde a es la longitud de la grieta y N el número de ciclos aplicados. Esta velocidad no es constante: bajo condiciones de carga cíclica de amplitud constante, los ensayos muestran que el crecimiento se acelera progresivamente conforme la grieta se alarga. El valor da/dn se obtiene de forma experimental midiendo los cambios de longitud de la grieta a lo largo de los ciclos, y se representa gráficamente frente al *rango del factor de intensidad de tensiones*, ΔK, en escala logarítmica.

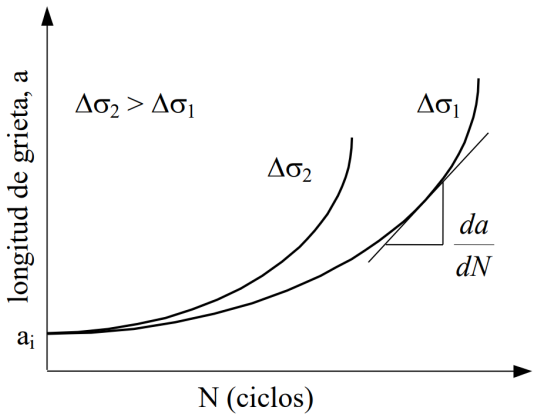

Figura 2.1: Velocidad de crecimiento de grieta

2.4.1. Ley de Paris

Una vez iniciada una grieta en un componente sometido a cargas cíclicas, su crecimiento no ocurre de forma aleatoria, sino que sigue un patrón que puede analizarse y modelarse. Uno de los modelos más utilizados para describir este fenómeno es la ley de Paris, propuesta por Paris y Gómez-Anderson, que establece una relación empírica

entre la velocidad de crecimiento de grieta por ciclo de carga y el rango del factor de intensidad de tensiones [2].

La magnitud *da/dN* representa la velocidad con la que la grieta crece por cada ciclo de carga aplicado. Esta velocidad se obtiene experimentalmente registrando el avance de la grieta en función del número de ciclos. El crecimiento se representa gráficamente frente al **rango del factor de intensidad de tensiones** ΔK, una variable fundamental que mide la variación de intensidad de tensiones en el vértice de la grieta a lo largo de un ciclo.

El rango ΔK se define como:

$$\Delta K = K_{\text{máx}} - K_{\text{mín}} \tag{2.5}$$

donde $K_{\text{máx}}$ y $K_{\text{mín}}$ son los valores máximo y mínimo del factor de intensidad de tensiones en un ciclo. Estos valores dependen directamente de las tensiones aplicadas, de la longitud de la grieta y de la geometría del componente. En el caso más sencillo, como una probeta con grieta central y carga uniaxial, se expresan como:

$$K_{\text{máx}} = Y\,\sigma_{\text{máx}}\sqrt{\pi a} \quad ; \quad K_{\text{mín}} = Y\,\sigma_{\text{mín}}\sqrt{\pi a} \tag{2.6}$$

donde Y es un factor adimensional que depende de la relación entre la longitud de la grieta a y el ancho de la probeta W, y $\sigma_{\text{máx}}$, $\sigma_{\text{mín}}$ son las tensiones extremas del ciclo. Esto permite reescribir el rango como:

$$\Delta K = Y\,(\sigma_{\text{máx}} - \sigma_{\text{mín}})\sqrt{\pi a} = Y\,\Delta\sigma\,\sqrt{\pi a} \tag{2.7}$$

Una vez definido ΔK, Paris y Gómez-Anderson observaron que la velocidad de crecimiento de la grieta en régimen estable podía describirse mediante una ley de potencias:

$$\frac{da}{dn} = C(\Delta K)^m \tag{2.8}$$

donde C y m son constantes que dependen del material, del entorno y de las condiciones de carga. Esta ley es válida en una región específica del crecimiento de grietas conocida como **región II**, y su aplicación es fundamental en ingeniería para estimar la vida restante de componentes con grietas detectadas.

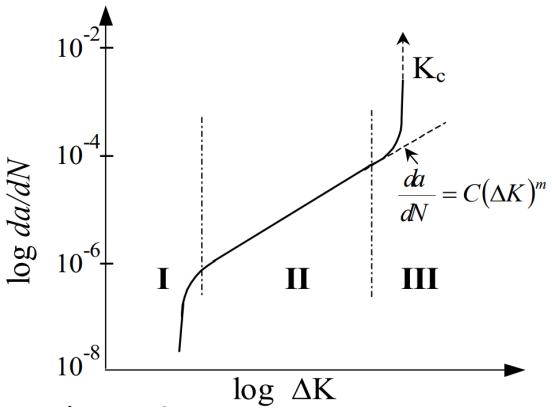

Figura 2.2: Ley de Paris y regiones del crecimiento de grieta

La representación gráfica de da/dN frente a ΔK en escala logarítmica da lugar a una curva típica dividida en tres regiones como se observa en la figura 2.2:

- **Región I (umbral):** a valores bajos de ΔK, la grieta no se propaga de forma apreciable. Existe un valor umbral ΔK_{th} por debajo del cual el crecimiento es prácticamente nulo.
- **Región II (crecimiento estable):** es la zona donde la ley de Paris se aplica con precisión. El crecimiento de la grieta sigue una relación estable y repetible, con pendiente definida en escala log-log.
- **Región III (propagación rápida):** cuando ΔK se aproxima al valor crítico K_c, la grieta se propaga aceleradamente hasta alcanzar la fractura. En esta zona, el comportamiento deja de ser predecible mediante la ley de Paris.

El uso de la ley de Paris permite estimar, mediante integración, la vida restante de un componente con una grieta existente, siempre que el régimen de carga sea constante y se conozcan los valores de a_0 y a_f (longitud inicial y final de la grieta). Sin embargo, este modelo no considera variables adicionales como la relación de carga R, el cierre de grieta, o el espesor del componente.

2.4.2. Influencia de la relación de carga R

La relación de carga se define como:

$$R = \frac{K_{\text{mín}}}{K_{\text{máx}}} \tag{2.9}$$

y tiene una gran influencia en la velocidad de crecimiento de grietas. Para un mismo valor de ΔK, distintas relaciones de carga pueden dar lugar a velocidades de propagación diferentes como puede observarse en la figura 2.3. En general, un valor alto de R tiende a acelerar el crecimiento, mientras que un valor bajo lo ralentiza.

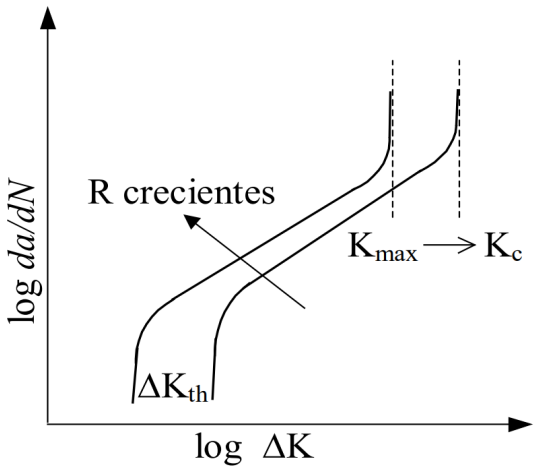

Figura 2.3: Influencia de la relación de carga R

Este comportamiento se relaciona con el cierre de grieta. Durante una parte del ciclo, la grieta puede no estar completamente abierta, lo que reduce el rango efectivo de tensiones en la punta:

$$\Delta K_{\text{ef}} = K_{\text{máx}} - K_{\text{op}} \tag{2.10}$$

Para tener en cuenta este efecto, se han propuesto modelos como:

Ley de Walker:

$$\frac{da}{dN} = C(\Delta K)^m (1-R)^\gamma \tag{2.11}$$

Modelo de Forman:

$$\frac{da}{dN} = \frac{C(\Delta K)^m}{(1-R)K_c - \Delta K} \tag{2.12}$$

2.4.3. Influencia del espesor

El espesor del componente afecta directamente al estado tensional en la vecindad del frente de grieta y a su velocidad de crecimiento. En cuerpos delgados, se tiende a un *estado de tensión plana*, con mayor deformación plástica local y mayor tenacidad aparente. La curva $\frac{da}{dN}$ vs. ΔK se desplaza hacia la derecha.

En componentes gruesos, se alcanza el *estado de deformación plana*, con fuerte triaxialidad que confina la zona plástica. El crecimiento es más estable y se mide el valor intrínseco K_{IC}. En este régimen, la curva se desplaza hacia la izquierda, reflejando un comportamiento más conservador.

3. Aprendizaje automático

3.1. Inteligencia Artificial

La *Inteligencia Artificial* es una disciplina de la informática que busca desarrollar sistemas capaces de simular procesos propios de la inteligencia humana mediante el uso de algoritmos y ordenadores. Su objetivo es lograr que las máquinas resuelvan problemas, tomen decisiones o realicen tareas para los que no han sido entrenados y que normalmente requieren razonamiento, aprendizaje o percepción.

Aunque no existe una definición única y universalmente aceptada, la IA puede entenderse como la capacidad de las máquinas para adquirir conocimientos a partir de datos, adaptarse a nuevas situaciones y aplicar lo aprendido para ejecutar acciones de forma autónoma. Esta disciplina abarca múltiples áreas, como el aprendizaje automático, el procesamiento del lenguaje natural, la visión por ordenador, la robótica, la clasificación de información o el diagnóstico médico.

En esencia, la IA representa una nueva forma de abordar problemas complejos, basada en la integración del conocimiento, el análisis de grandes volúmenes de datos y la toma de decisiones automatizada. Su desarrollo ha transformado múltiples sectores y continúa ampliando sus aplicaciones en campos como la industria, la salud, las finanzas, la educación y el transporte.

3.2. Machine Learning

El *Machine Learning*, o aprendizaje automático, es una rama de la inteligencia artificial cuyo objetivo es dotar a las máquinas de la capacidad de aprender, tomar decisiones y ejecutar tareas sin intervención humana directa. A diferencia de los sistemas programados con reglas fijas, los algoritmos de *Machine Learning* aprenden a partir de la experiencia y los datos, adaptando su comportamiento sin necesidad de ser explícitamente programados para cada tarea.

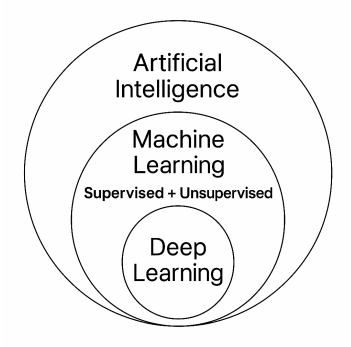

Figura 3.1: Jerarquía de la Inteligencia Artificial

Estos algoritmos funcionan analizando grandes volúmenes de datos, a partir de los cuales identifican patrones, correlaciones o estructuras que les permiten predecir comportamientos futuros o clasificar información. Una vez entrenado, el modelo puede tomar decisiones o hacer predicciones basándose en nuevos datos de entrada, lo que lo hace útil en aplicaciones como reconocimiento de voz, análisis de imágenes, detección de fraudes o sistemas de recomendación.

A diferencia de una ley física, que se deriva de principios fundamentales y describe el comportamiento del mundo natural con base en relaciones causales comprobadas, un modelo de *Machine Learning* no tiene necesariamente una conexión directa con la realidad física. Su objetivo no es explicar fenómenos, sino ajustarse a los datos de entrada para predecir resultados conocidos. El modelo aprende a partir de ejemplos en los que ya se conoce la salida correcta, y su utilidad depende de su capacidad para generalizar ese conocimiento a nuevos casos. Mientras que una ley física pretende ser universal y válida en cualquier contexto, un modelo de aprendizaje automático solo refleja los patrones contenidos en los datos con los que fue entrenado.

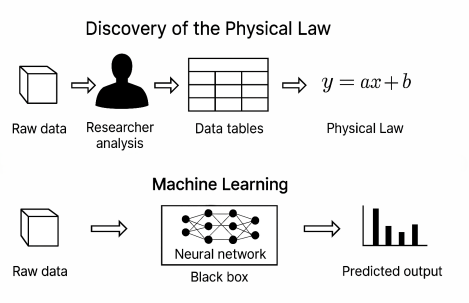

Figura 3.2: Enfoque convencional vs. *machine learning*: de la formulación explícita a la predicción automática.

El rendimiento de un modelo de *Machine Learning* mejora cuanto mayor es la cantidad y calidad de los datos con los que se entrena. Con más datos, el modelo puede aprender de forma más robusta, generalizar mejor y obtener resultados más precisos. Además, estos modelos se pueden reentrenar o ajustar conforme se dispone de nuevos datos, lo que permite que evolucionen y se adapten a contextos cambiantes.

Existen tres tipos principales de aprendizaje automático, clasificados según la forma en que el algoritmo recibe los datos:

- **Aprendizaje supervisado:** el modelo se entrena con un conjunto de datos que incluye tanto las entradas como las salidas esperadas. El objetivo es que el modelo aprenda a predecir la salida correcta para nuevas entradas. Es el enfoque más utilizado en tareas de regresión y clasificación. Los algoritmos empleados en este trabajo corresponden a esta categoría.
- **Aprendizaje no supervisado:** en este caso, el modelo no dispone de salidas conocidas, y su tarea consiste en encontrar estructuras ocultas en los datos, como agrupaciones o asociaciones. Se aplica comúnmente en reducción de dimensionalidad, segmentación de clientes o análisis exploratorio.

- **Aprendizaje por refuerzo:** el modelo aprende a tomar decisiones a través de ensayo y error, recibiendo recompensas o penalizaciones según su comportamiento. Este tipo de aprendizaje es habitual en robótica, juegos y sistemas de control autónomo.

3.3. Redes Neuronales Artificiales

Las redes neuronales artificiales (RNA) son modelos computacionales inspirados en el funcionamiento del cerebro humano. Se utilizan para resolver tareas complejas como reconocimiento de patrones, clasificación, regresión, predicción o procesamiento de lenguaje natural. Gracias a su capacidad para aprender a partir de los datos, se han convertido en herramientas fundamentales dentro del campo del aprendizaje automático.

Entre sus principales ventajas destacan la capacidad para modelar relaciones no lineales complejas, su adaptabilidad a diferentes tipos de datos (tablas, imágenes, texto, audio) y su mejora progresiva al ser entrenadas con grandes volúmenes de información. También pueden aplicarse a una amplia variedad de problemas sin necesidad de cambiar su estructura base.

Sin embargo, también presentan desventajas. Requieren grandes cantidades de datos y recursos computacionales para alcanzar un buen rendimiento, pueden ser difíciles de interpretar (son modelos opacos) y su funcionamiento depende de una correcta configuración de muchos parámetros, lo que exige experiencia y tiempo en su ajuste.

3.3.1. Neuronas

Las redes neuronales están formadas por unidades llamadas *neuronas artificiales*. Cada neurona recibe una o más entradas, las multiplica por unos pesos asociados, suma los resultados y les aplica una transformación mediante una función de activación. El resultado se propaga a la siguiente capa. El conjunto de conexiones y pesos determina cómo la red procesa la información y realiza sus predicciones. Durante el entrenamiento, los pesos se ajustan para minimizar el error entre la salida generada por la red y la salida esperada.

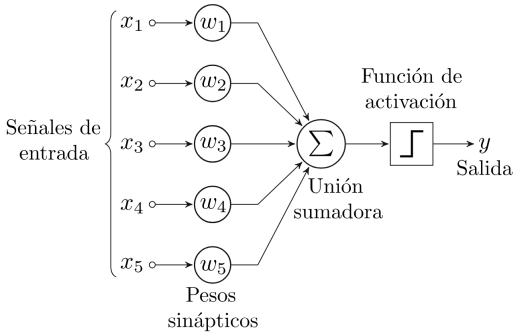

Figura 3.3: Esquema de una neurona

3.3.2. Función de activación

La función de activación es una operación matemática que se aplica a la salida de cada neurona. Su función principal es introducir no linealidad en el modelo, lo que permite que la red aprenda relaciones complejas entre variables. Sin esta función, la red actuaría como un modelo lineal, sin importar su profundidad.

Entre las funciones más utilizadas se encuentran:

- **ReLU (Rectified Linear Unit):** devuelve el valor de entrada si es positivo y cero si es negativo. Es simple y eficiente, y suele usarse en redes profundas.

$$f(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ x & \text{si } x \geq 0 \end{cases} \tag{3.1}$$

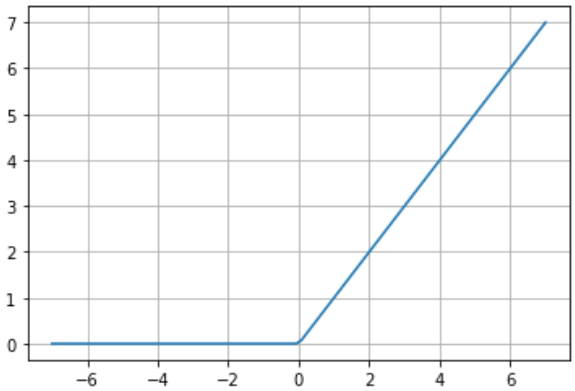

Figura 3.4: Representación de la función ReLU.

- **Sigmoide:** transforma la entrada en un valor entre 0 y 1. Se emplea cuando se requiere interpretar la salida como una probabilidad.

$$f(x) = \frac{1}{1 - e^{-x}} \tag{3.2}$$

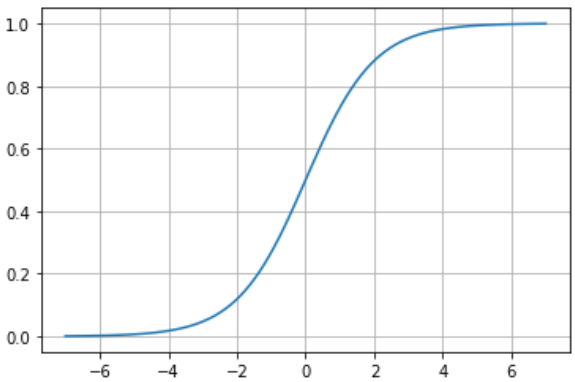

Figura 3.5: Representación de la función Sigmoide.

- **Tangente hiperbólica:** similar a la sigmoide, pero sus salidas están en el rango [–1, 1], lo que puede mejorar el flujo de información en capas intermedias.

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.3}$$

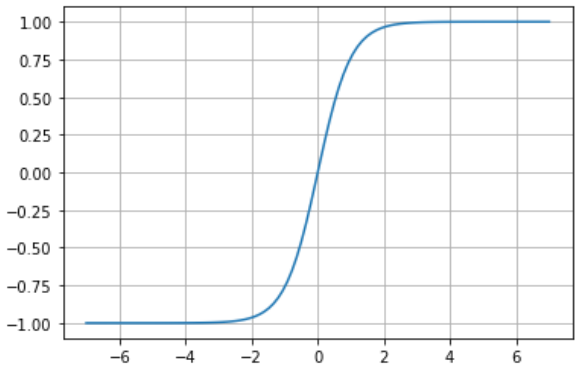

Figura 3.6: Representación de la función Tangente Hiperbólica.

Tipos de redes según el número de capas

Las redes neuronales se clasifican por su profundidad:

- **Redes superficiales:** tienen solo una capa oculta entre la entrada y la salida. Son más simples y rápidas de entrenar, pero limitadas en su capacidad de representación.

- **Redes profundas (Deep Neural Networks, DNN):** contienen múltiples capas ocultas. Estas redes son capaces de aprender representaciones jerárquicas y extraer patrones complejos a partir de los datos. A cambio, requieren más datos y potencia de cómputo, y son más propensas al sobreajuste si no se regulan adecuadamente.

3.3.3. Función de pérdida

La función de pérdida mide el error entre la salida del modelo y la salida real esperada. Es un elemento central en el proceso de entrenamiento, ya que guía la actualización de los pesos a través del algoritmo de optimización (como *gradient descent*). En tareas de regresión se suele usar el error cuadrático medio (MSE), mientras que en clasificación es común emplear la entropía cruzada (*cross-entropy*). Cuanto menor es el valor de la función de pérdida, más se está ajustando la red a los datos.

3.4. Tipos de redes neuronales artificiales

3.4.1. Perceptrón Multicapa

El Perceptrón

El *Perceptrón* es la unidad básica de una red neuronal artificial. Fue propuesto por Frank Rosenblatt en 1958 [3] y representa el modelo más simple de una neurona artificial: toma un conjunto de entradas numéricas, las pondera con pesos, las suma y aplica una función de activación (originalmente una función escalón). En problemas de clasificación, ese mecanismo produce una salida binaria que decide entre dos clases linealmente separables.

Sin embargo, su utilidad va más allá de la clasificación. Si la función de activación se sustituye por la identidad, el perceptrón actúa como un regresor lineal, aprendiendo directamente una combinación lineal de las entradas. Y si se emplea una activación suave y acotada, como la tangente hiperbólica, el mismo esquema se transforma en un regresor no lineal capaz de aproximar curvas continuas. De este modo, además de sentar las bases para redes más profundas, el perceptrón ofrece una solución unificada para tareas de clasificación y de regresión, tanto lineales como no lineales.

Funcionamiento del Perceptrón

El perceptrón recibe un vector de características $\mathbf{x} = (x_1, \ldots, x_d)^\top$, aplica un vector de pesos $\mathbf{w} = (w_1, \ldots, w_d)^\top$ y un sesgo b, y genera el *potencial interno*:

$$z \;=\; \mathbf{w}^\top \mathbf{x} + b. \tag{3.4}$$

En problemas de regresión no lineal convertimos ese potencial en una salida continua, por ejemplo mediante la tangente hiperbólica (*funcion de activación*) que comprime los valores al rango $(-1, 1)$:

$$\hat{y} \;=\; f(z) \;=\; \tanh(z) \tag{3.5}$$

Para determinar el valor de los pesos, se buscan los valores que minimicen la *función de pérdida*. El error total sobre N ejemplos será:

$$E \;=\; \frac{1}{2}\sum_{i=1}^{N}\big(y_i - \hat{y}_i\big)^2 \;=\; \frac{1}{2}\sum_{i=1}^{N}\Big(y_i - \tanh(\mathbf{w}^\top \mathbf{x}_i + b)\Big)^2 \tag{3.6}$$

Para minimizar E buscamos los parametros que anulan sus derivadas:

$$\frac{\partial E}{\partial \mathbf{w}} = \mathbf{0} \qquad\qquad \frac{\partial E}{\partial b} = 0 \tag{3.7}$$

Con $\tanh'(z) = 1 - \tanh^2(z) = 1 - \hat{y}_i^2$ obtenemos las derivadas explicitas:

$$\frac{\partial E}{\partial \mathbf{w}} = -\sum_{i=1}^{N}\big(y_i - \hat{y}_i\big)\,\big(1 - \hat{y}_i^2\big)\,\mathbf{x}_i \qquad \frac{\partial E}{\partial b} = -\sum_{i=1}^{N}\big(y_i - \hat{y}_i\big)\,\big(1 - \hat{y}_i^2\big) \tag{3.8}$$

Igualarlas a cero conduce al sistema de ecuaciones normales (solución analítica); si preferimos un procedimiento iterativo, basta con desplazar los parámetros en la dirección opuesta al gradiente (método del descenso por el gradiente):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial E}{\partial \mathbf{w}} \qquad b \leftarrow b - \eta \frac{\partial E}{\partial b} \tag{3.9}$$

donde η es la *tasa de aprendizaje*. Repetir este ajuste hace que la superficie $\hat{y} = \tanh(\mathbf{w}^\top \mathbf{x} + b)$ se amolde a la nube de datos $(\mathbf{x}_i, y_i)$, proporcionando una aproximación no lineal a los valores continuos buscados.

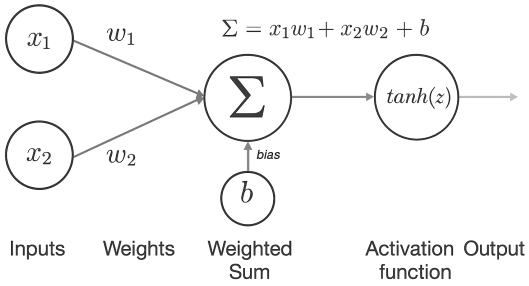

Figura 3.7: Perceptrón

Perceptrón Multicapa

El Perceptrón Multicapa (o *MLP*, por sus siglas en inglés) extiende la idea del perceptrón simple al encadenar varias capas de neuronas. En lugar de conectar directamente las entradas con la salida, las señales pasan primero por una o más capas intermedias—denominadas capas ocultas—cada una compuesta por varias neuronas semejantes al perceptrón básico.

Cada neurona de una capa oculta recibe la salida de la capa anterior, la combina y aplica su propia función de activación, generando un conjunto de salidas paralelas que sirven de entrada a la siguiente capa. Este encadenamiento permite a la red aprender transformaciones progresivamente más complejas de los datos: las primeras capas extraen características sencillas (p. ej. bordes o texturas en imágenes), mientras que las capas sucesivas combinan esas características en patrones de orden superior.

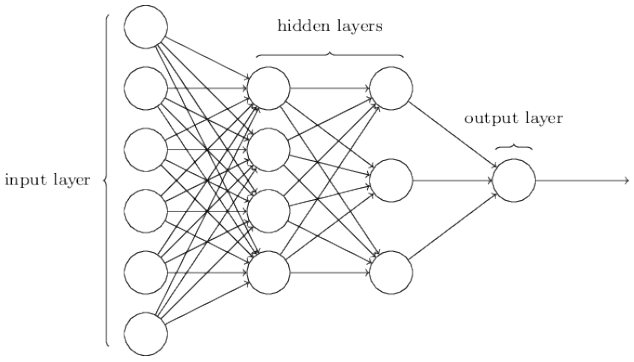

Figura 3.8: Esquema del Perceptrón Multicapa

El proceso de entrenamiento sigue un principio equivalente al descenso de gradiente, pero distribuido a lo largo de todas las capas mediante el algoritmo de *retropropagación* (backpropagation). Gracias a este mecanismo, el error cometido en la salida final se reparte hacia atrás por la red, permitiendo ajustar los pesos de cada capa según su contribución al desenfoque total.

De este modo, el Perceptrón Multicapa supera las limitaciones del perceptrón simple y se convierte en un modelo capaz de aproximar prácticamente cualquier función continua, lo que explica su éxito en tareas de visión por computador, procesamiento de lenguaje natural y un sinfín de aplicaciones de regresión y clasificación complejas.

3.4.2. Redes de Base Radial

Una red de base radial es un tipo de red neuronal diseñada para aproximar funciones multivariadas: construye una función $\hat{f}(\mathbf{x})$ que reproduce los pares entrada-salida observados durante el entrenamiento. La idea se basa en la interpolación con funciones de base radial —como la gaussiana— obteniendo sus coeficientes de forma automática [4, 5].

La red consta de tres capas sucesivas:

- **Entrada:** Recibe el vector $\mathbf{x} \in \mathbb{R}^n$ sin modificarlo.
- **Oculta:** Se emplean m funciones de base radial $\varphi_j\big(\|\mathbf{x}-\mathbf{c}_j\|\big)$, normalmente *gaussianas*, donde cada centro $\mathbf{c}_j$ define una región local de influencia.

$$\varphi(r) \;=\; \exp\!\left(-\frac{r^2}{2\sigma^2}\right) \tag{3.10}$$

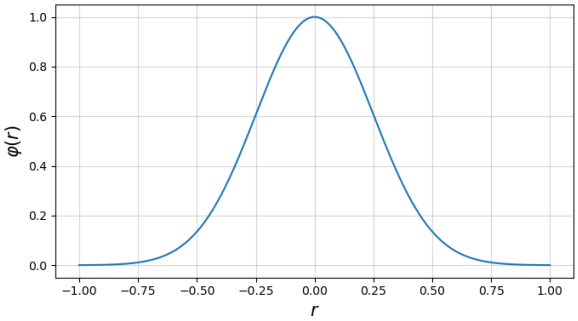

Figura 3.9: Base radial gaussiana con $\sigma = 1$

- **Salida** (Figura 3.10)*:* La no linealidad queda confinada a la capa oculta y los pesos $\{w_j\}$ se ajustan resolviendo un problema de mínimos cuadrados lineal que combina las respuestas de la capa oculta:

$$\hat{f}(\mathbf{x}) \;=\; \sum_{j=1}^{m} w_j\,\varphi_j\big(\|\mathbf{x}-\mathbf{c}_j\|\big) + b \tag{3.11}$$

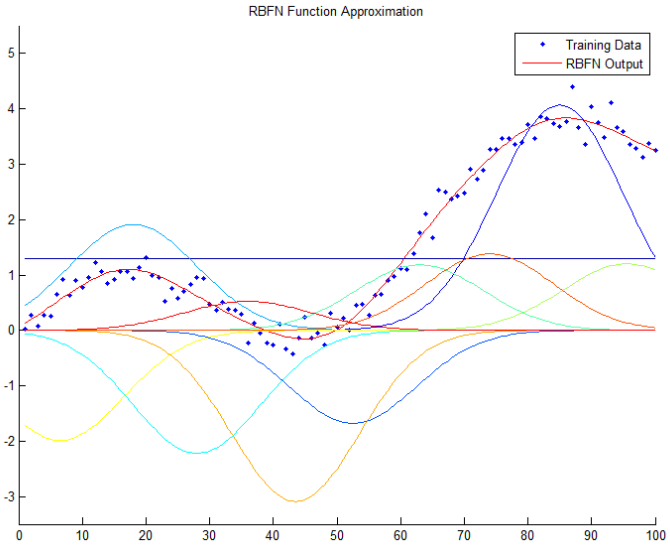

Figura 3.10: Suma de funciones gausianas (una función por neurona).

Cada neurona oculta depende únicamente de la distancia euclídea al centro $\mathbf{c}_j$. Esta simetría esférica hace que la activación sea intensa cerca del centro y decrezca al alejarse, generando regiones de influencia locales y favoreciendo el aprendizaje rápido sin un acoplamiento global excesivo.

1. **Selección de centros**: Elegir un número adecuado de centros $\{\mathbf{c}_j\}_{j=1}^{m}$ y distribuirlos en el espacio de entrada (por ejemplo, mediante k-means).

2. **Determinación de anchuras**: Asignar a cada centro su anchura σ_j:

$$\sigma_j \;=\; \frac{\text{distancia media al vecino más cercano}}{\sqrt{2m}} \tag{3.12}$$

3. **Cálculo de pesos**: Para cada ejemplo de entrenamiento $i = 1, \ldots, N$ conocemos la salida deseada $y^{(i)}$ y la respuesta de la red $\hat{y}^{(i)} = \sum_{j=1}^{m} w_j \, \Phi_{ij}$ Definimos

$$\Phi_{ij} \;=\; \varphi_j\big(\|\mathbf{x}^{(i)} - \mathbf{c}_j\|\big) \tag{3.13}$$

y buscamos los pesos $\mathbf{w}$ que minimicen la suma de errores cuadrados:

$$E(\mathbf{w}) \;=\; \sum_{i=1}^{N}\Big(y^{(i)} - \sum_{j=1}^{m} w_j \, \Phi_{ij}\Big)^2 \tag{3.14}$$

Anular el gradiente

$$\frac{\partial E}{\partial w_j} = 0 \tag{3.15}$$

conduce a las ecuaciones normales $(\Phi^\top \Phi)\mathbf{w} = \Phi^\top \mathbf{y}$, que se resuelven mediante regresión lineal.

Las RBF entrenan con gran rapidez porque el ajuste final es lineal, poseen capacidad de aproximación universal y ofrecen una interpretación geométrica intuitiva. Sin embargo, su rendimiento depende de la selección del número de centros, de sus anchuras y de su posición; en espacios de alta dimensión pierden eficiencia y, con un gran número de bases, la matriz de pesos se vuelve costosa de invertir [6].

3.5. Diseño de Redes Neuronales

3.5.1. Preparación de los datos

Una etapa crítica en la construcción de modelos de regresión es la limpieza del conjunto de datos. Es necesario detectar y tratar los *outliers*, es decir, aquellos valores atípicos que se alejan significativamente del patrón general de los datos, ya que pueden influir de forma desproporcionada en los resultados y desviar el aprendizaje del modelo. También deben manejarse los datos faltantes, que pueden imputarse con medidas como la media o la mediana, o eliminarse si son escasos y no afectan la representatividad del conjunto.

Además, no todos los rasgos de entrada contribuyen al rendimiento del modelo por igual. Algunos tienen una relación directa con la variable objetivo, mientras que otros pueden introducir ruido o redundancia. Incluir demasiados atributos puede hacer que el modelo sea innecesariamente complejo, aumente el riesgo de sobreajuste y reduzca la capacidad de generalización. Por eso, es fundamental seleccionar solo aquellos rasgos con relevancia comprobada.

3.5.2. Normalización y estandarización

Las redes neuronales son sensibles a la escala de las variables de entrada, por lo que es fundamental aplicar una transformación previa a los datos. La normalización ajusta los valores para que se encuentren en un rango definido, como [0, 1] o [–1, 1], mientras que la estandarización transforma los datos para que tengan media cero y desviación estándar uno. Ambas técnicas mejoran la estabilidad numérica, aceleran el proceso de entrenamiento y ayudan a que el modelo converja de forma más eficiente.

3.5.3. Ajuste y sobreajuste

En el aprendizaje supervisado, el diseño de modelos robustos implica encontrar un equilibrio entre la capacidad del modelo para ajustarse a los datos de entrenamiento y su habilidad de generalización con nuevos datos de entrada. Este equilibrio se conoce como punto óptimo o *sweet spot*, y es esencial para lograr una predicción fiable sobre casos.

En este tipo de aprendizaje, el modelo se entrena a partir de un conjunto de datos etiquetados con el objetivo de predecir sus salidas. Para evaluar correctamente su desempeño, los datos deben dividirse, al menos, en dos subconjuntos: *entrenamiento* y *prueba*. En muchos casos, se incluye también un conjunto de *validación*. Durante el entrenamiento, el modelo aprende la relación entre las variables de entrada y la salida esperada. Luego, en la fase de prueba, se evalúa su capacidad para predecir resultados sobre ejemplos no vistos, esto se denomina aprendizaje inductivo. La capacidad de un modelo para inducir correctamente determina su grado de generalización.

No obstante, esta capacidad puede verse comprometida por dos fenómenos: el subajuste y el sobreajuste.

- El subajuste o *underfitting* se presenta cuando el modelo es demasiado simple para capturar los patrones subyacentes en los datos. Esto suele deberse a un conjunto de entrenamiento insuficiente o a una arquitectura limitada, lo que termina produciendo una excesiva generalizacion de nuestro modelo.
- El sobreajuste o *overfitting* ocurre cuando el modelo aprende no solo los patrones relevantes, sino también el ruido y las fluctuaciones del conjunto de entrenamiento. Nuestro modelo se encuentra ceñido sobre los datos de entrenamiento y su capacidad de generalización es nula.

Para alcanzar el *sweet spot* y lograr un buen equilibrio entre generalización y ajuste, es fundamental disponer de un conjunto de datos amplio, representativo y equilibrado, que permita al modelo captar patrones generales en lugar de detalles aislados o ruido. Además, dividir correctamente los datos en conjuntos de entrenamiento y prueba facilita una evaluación objetiva del rendimiento y la optimización del modelo.

3.5.4. División del conjunto de datos

Para que un modelo de aprendizaje automático funcione correctamente, es necesario evaluar su rendimiento durante su desarrollo. Por esta razón, el conjunto de datos disponible se divide en dos subconjuntos con propósitos distintos. Esta separación permite entrenar el modelo, ajustar sus parámetros y, finalmente, comprobar su capacidad para generalizar. Sin esta división, sería imposible detectar si el modelo está aprendiendo realmente los patrones del problema o simplemente memorizando los datos.

- **Entrenamiento:** Este conjunto contiene los datos que se utilizan directamente para enseñar al modelo. Durante el proceso de entrenamiento, el modelo ajusta sus parámetros en función de los patrones presentes en estos datos. Una buena selección del conjunto de entrenamiento es fundamental para que el modelo aprenda correctamente.
- **Prueba:** Este conjunto se utiliza para evaluar el rendimiento del modelo mientras se entrena. Sirve para ajustar los hiperparámetros y evitar el sobreajuste, ya que permite verificar si el modelo está empezando a memorizar los datos en lugar de aprender patrones generales.

3.5.5. Validación cruzada *k-fold*

El método *k-fold cross validation* es una técnica utilizada para evaluar la capacidad de generalización de un modelo y reducir el riesgo de obtener resultados sesgados por una partición aleatoria de los datos. Consiste en dividir el conjunto original en k partes iguales o *folds* (Figura 3.11). Luego, el modelo se entrena k veces, utilizando en cada iteración $k-1$ folds para entrenamiento y el fold restante para prueba.

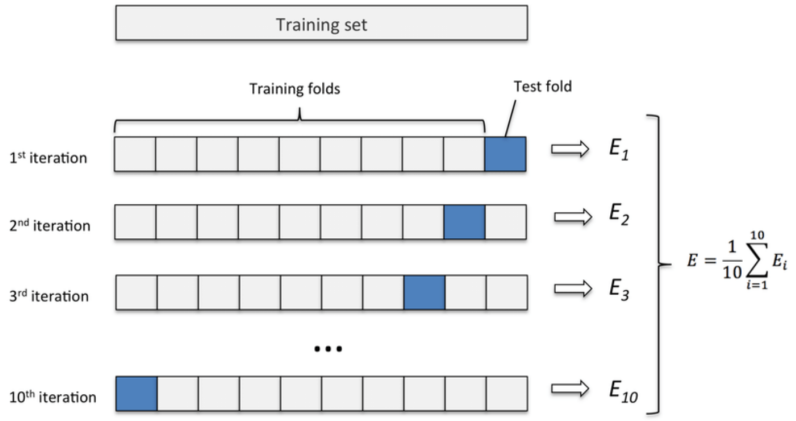

Figura 3.11: Diagrama de validación cruzada mediante k-fold

Este procedimiento garantiza que todos los datos sean usados tanto para entrenar como para validar el modelo en diferentes combinaciones. Al finalizar, se calcula la media de las métricas obtenidas en cada iteración, lo que proporciona una estimación más robusta y confiable del rendimiento real del modelo. Además, permite detectar posibles problemas de sobreajuste o subajuste que podrían pasar desapercibidos en una única partición.

3.5.6. Entrenamiento del modelo

El diseño y entrenamiento de una red neuronal implica ajustar diversos componentes que determinan su rendimiento y capacidad de generalización. Estos se pueden agrupar en tres bloques principales: arquitectura, optimización y regularización.

Arquitectura del modelo

La arquitectura define la estructura interna de la red. Esto incluye:

- **Número de capas ocultas y neuronas por capa:** La estructura de la red neuronal determina su capacidad para modelar relaciones complejas. Aumentar el número de capas y neuronas permite representar relaciones no lineales más sofisticadas, pero también incrementa el riesgo de sobreajuste si no se cuenta con suficientes datos. Por el contrario, una red muy simple puede no aprender lo suficiente (subajuste).

- **Funciones de activación:** Las **funciones de activación** introducen no linealidad en las redes neuronales, permitiendo que el modelo aprenda relaciones complejas entre las variables. Actúan sobre la salida de cada neurona, determinando su activación en función de la suma ponderada de las entradas.

  Sin estas funciones, la red solo podría representar relaciones lineales, sin importar su profundidad. Las más comunes son las funciones *sigmoide*, *tangente hiperbolica* o *ReLu*.

- **Pesos e inicialización:** Los *pesos* son los parámetros que una red neuronal ajusta durante el entrenamiento para aprender la relación entre las entradas y las salidas. Determinan la influencia que cada entrada tiene en la activación de una neurona.

  Al comenzar el entrenamiento, estos pesos deben inicializarse con valores adecuados. Una mala inicialización puede causar que los gradientes se vuelvan demasiado pequeños (desvanecimiento) o demasiado grandes (explosión), dificultando el aprendizaje. Existen técnicas que ajustan los valores iniciales en función del número de entradas y salidas de cada neurona, favoreciendo una propagación estable del gradiente durante el entrenamiento.

Optimización del entrenamiento

La fase de entrenamiento ajusta los pesos de la red para minimizar el error entre la salida predicha y la real. Para ello se consideran:

- **Función de pérdida:** La función de pérdida (Figura 3.12) mide cuán lejos está la salida del modelo respecto al valor real. En regresión, el error cuadrático medio (MSE) penaliza más los errores grandes, siendo útil cuando se quiere minimizar grandes desviaciones. El error absoluto medio (MAE) trata todos los errores por igual y es menos sensible a valores atípicos. La elección entre ambas depende de la naturaleza del problema y la tolerancia a errores extremos.

- **Algoritmos de optimización:** Son métodos que actualizan los pesos de la red neuronal con el objetivo de minimizar la función de pérdida. Utilizan el gradiente de dicha función para orientar los ajustes. El algoritmo más básico es el *descenso de gradiente estocastico* (Stochastic Gradient Descent), que actualiza los pesos con base en pequeños lotes de datos, pero puede ser lento o inestable si no se configura adecuadamente. Optimizadores como *Adam* y *L-BFGS* mejoran este proceso adaptando la tasa de aprendizaje a cada parámetro, lo que acelera la convergencia y mejora la estabilidad en redes más complejas.

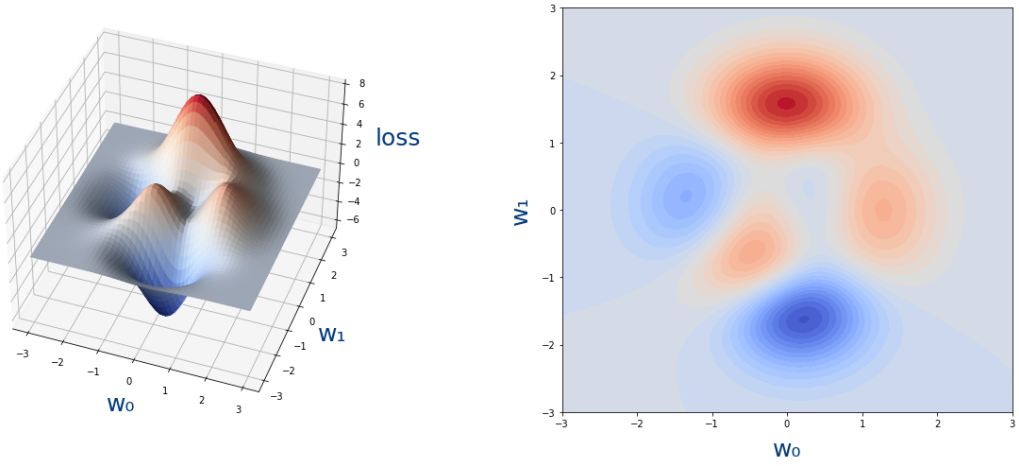

Figura 3.12: Representación de la función de pérdida para una red con dos pesos.

- **Hiperparámetros de entrenamiento:** Los hiperparámetros son configuraciones definidas antes del entrenamiento que no se aprenden directamente a partir de los datos, pero influyen de forma significativa en el comportamiento y rendimiento del modelo. A diferencia de los pesos de la red, que se ajustan automáticamente durante el aprendizaje, los hiperparámetros deben seleccionarse mediante prueba y error, validación cruzada u otras técnicas de optimización. Algunos de los más importantes son:

  - *Batch size:* define cuántos ejemplos se procesan antes de actualizar los pesos. Tamaños pequeños pueden hacer el entrenamiento más ruidoso pero favorecen la generalización; tamaños grandes son más estables pero pueden llevar a modelos menos flexibles.
  - *Épocas:* indica cuántas veces se recorre completamente el conjunto de entrenamiento. Si se usan pocas épocas, el modelo puede quedarse corto en su aprendizaje; demasiadas pueden hacer que memorice los datos y pierda capacidad de generalización (Figura 3.13).
  - *Tasa de aprendizaje:* controla el tamaño de los pasos que da el optimizador al actualizar los pesos. Una tasa demasiado alta puede provocar inestabilidad y saltarse óptimos; una demasiado baja puede hacer que el modelo tarde mucho en converger o quede atrapado en soluciones poco eficaces.

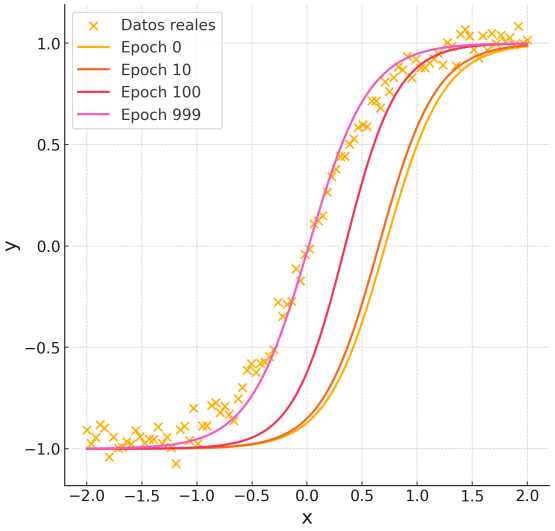

Figura 3.13: Evolución de la curva aprendida por *época*.

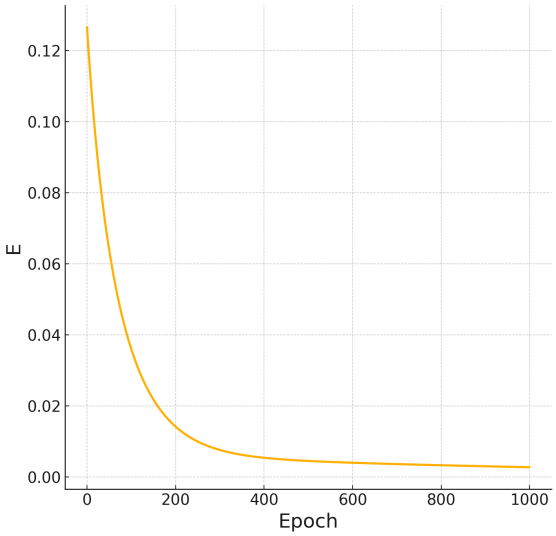

Figura 3.14: Disminución del *error* por *época*.

Regularización

Durante el entrenamiento de modelos, es común aplicar técnicas de regularización para mejorar su capacidad de generalización. Estas técnicas buscan evitar que el modelo se ajuste demasiado a los datos de entrenamiento, ayudándolo a centrarse en los patrones relevantes y a ignorar información innecesaria o ruido.

Dos formas habituales de regularización son L1 y L2, ambas incorporadas como términos adicionales en la función de pérdida del modelo. Su efecto se basa en restringir la magnitud de los pesos, es decir, los valores internos que determinan la influencia de cada variable de entrada.

- La *regularización L1* tiende a reducir algunos pesos exactamente a cero. Esto implica que ciertas variables de entrada dejan de tener impacto en el modelo, lo cual equivale a una forma automática de selección de características. De esta manera, L1 favorece modelos más simples y centrados solo en las variables más relevantes.
- La *regularización L2*, en cambio, no anula los pesos, pero sí los reduce de forma gradual. Esto ayuda a mantener el modelo equilibrado y evita que dependa excesivamente de valores individuales. L2 suaviza las actualizaciones durante el entrenamiento, contribuyendo a un aprendizaje más estable y progresivo.

Ambas técnicas tienen como objetivo reducir el riesgo de sobreajuste y mejorar el rendimiento del modelo al enfrentarse a datos no vistos.

En conjunto, el aprendizaje automático representa una nueva forma de abordar problemas complejos: en lugar de partir de reglas explícitas o modelos formulados manualmente, se entrena a las máquinas para que encuentren por sí mismas patrones útiles en los datos. Esta capacidad de aprender, adaptarse y mejorar con la experiencia ha transformado múltiples disciplinas y continuará siendo una pieza clave en el desarrollo de tecnologías inteligentes.

4. Metodología

4.1. Software utilizado

Para la implementación de los modelos, el procesamiento de los datos y la elaboración del presente documento, se han empleado diversas herramientas de software y librerías especializadas. A continuación, se describen brevemente las principales aplicaciones y recursos utilizados durante el desarrollo del trabajo.

Software:

- ***Python***: Lenguaje de alto nivel empleado para el análisis de datos y la programación de algoritmos [7].
- ***Spyder***: Entorno de desarrollo integrado (*IDE*) utilizado para escribir, depurar y ejecutar código *Python* de forma eficiente [8].
- ***Anaconda***: Distribución de *Python* que facilita la gestión de entornos y paquetes, especialmente útil en ciencia de datos y machine learning [9].
- ***Excel***: Herramienta de hoja de cálculo utilizada para la manipulación preliminar de datos y análisis exploratorio sencillo [10].

Librerías:

- ***NumPy***: Librería fundamental para el cálculo numérico en *Python*, utilizada para manejar vectores, matrices y operaciones algebraicas [11].
- ***Pandas***: Herramienta esencial para la manipulación y análisis de datos estructurados, especialmente en formato tabular [12].
- ***pickle***: Librería estándar de *Python* utilizada para la serialización y almacenamiento de objetos, permitiendo guardar y recuperar modelos entrenados de manera eficiente [13].
- ***Scikit-learn***: Biblioteca de *machine learning* que proporciona herramientas para la construcción, entrenamiento y validación de modelos predictivos [14].
- ***Matplotlib***: Librería de visualización utilizada para representar gráficamente datos y resultados de forma clara y personalizable [15].

4.2. Algoritmos empleados

En el presente trabajo se ha optado por utilizar dos enfoques complementarios de redes neuronales para modelar el comportamiento del titanio: una *Red de Base Radial (RBFN)* y un *Perceptrón Multicapa (MLP)*. La elección de estos algoritmos responde tanto a las características específicas del problema como a las propiedades de cada modelo.

Red de Base Radial (RBFN): este tipo de red es especialmente adecuado para problemas en los que el comportamiento de la variable objetivo varía de forma diferente en distintas regiones del espacio de entrada. Su estructura basada en funciones base localizadas permite modelar eficazmente relaciones complejas con una interpretación relativamente sencilla. Además, su entrenamiento es rápido y estable, dado que la capa de salida se ajusta mediante regresión lineal. Por ello, la RBFN resulta idónea para obtener un modelo interpretable y con buena capacidad de adaptación a datos no lineales [16].

Perceptrón Multicapa (MLP): por su parte, el MLP ofrece una arquitectura más flexible y general, capaz de aproximar cualquier función continua bajo ciertas condiciones. Su capacidad para aprender representaciones jerárquicas mediante múltiples capas ocultas lo convierte en una herramienta poderosa para capturar dependencias complejas y no lineales entre variables. Además, el uso de funciones de activación no lineales permite modelar comportamientos que no podrían capturarse con una simple combinación lineal de las entradas.

El uso combinado de RBFN y MLP permite, por tanto, contrastar los resultados obtenidos con arquitecturas de diferente naturaleza: mientras que la RBFN aporta un modelo con representación local y entrenamiento eficiente, el MLP proporciona mayor flexibilidad y capacidad de modelado global. Esta estrategia permite explorar distintas perspectivas del problema y obtener una visión más completa de las capacidades predictivas de los modelos.

4.3. Titanio Ti-6Al-4V

4.3.1. Preparación de los datos

El Ti-6Al-4V es una aleación de titanio muy utilizada por su alta resistencia, baja densidad y excelente resistencia a la corrosión. Destaca en sectores como el aeroespacial, biomédico y automotriz, especialmente cuando se requiere una gran relación resistencia-peso y estabilidad térmica.

El conjunto de datos utilizado en este apartado corresponde a medidas experimentales del comportamiento del titanio Ti-6Al-4V. Se trabaja con un conjunto de datos extraído de *NASGRO* [17]. El conjunto de datos está compuesto por cuatro curvas correspondientes a distintos valores de la relación de carga ($R = 0{,}1$, $R = 0{,}33$, $R = 0{,}5$, $R = 0{,}75$), compuestas por un total de 320 puntos.

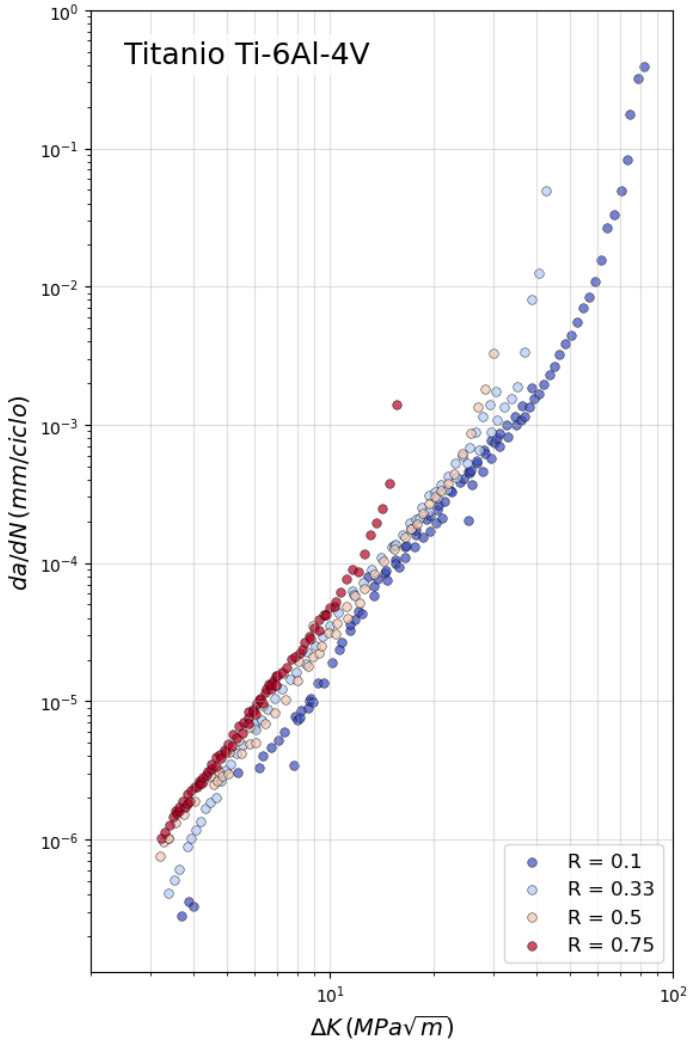

Figura 4.1: Conjunto de datos del Titanio Ti-6Al-4V

Por lo tanto, el modelo se ha entrenado utilizando dos rasgos de entrada: ΔK y R, y una variable de salida: la velocidad de propagación de grieta, da/dN.

El archivo de datos empleado para el entrenamiento del modelo es un fichero en formato *Excel*, compuesto por las siguientes columnas:

da/dN (mm/ciclo)	ΔK (MPa$\sqrt{\text{m}}$)	R

Cuadro 4.1: Estructura del fichero Excel utilizado para el entrenamiento del modelo.

Para facilitar la modelización y mejorar la estabilidad numérica de los modelos, se realizó un preprocesado común a ambos algoritmos implementados. En primer lugar, se aplicó una transformación logarítmica decimal a ΔK y da/dN. Esta transformación permite reducir la asimetría de las distribuciones originales. Posteriormente, se llevó a cabo una normalización de las variables mediante *escalado estandar* (`StandardScaler`).

Finalmente, el conjunto de datos completo se dividió en dos subconjuntos:

- **Entrenamiento** (80 % de los datos), utilizado para el ajuste de los modelos.
- **Prueba** (20 % de los datos), reservado para la evaluación final del rendimiento de cada modelo.

La partición se realizó de forma aleatoria, con una semilla fija para asegurar la reproducibilidad de los experimentos. Este mismo conjunto de entrenamiento y prueba se empleó tanto para el modelo RBFN como para el modelo MLP, de modo que los resultados obtenidos sean directamente comparables.

4.3.2. Modelado con Red de Base Radial (RBFN)

En primer lugar, se determinó el número optimo de centros (*neuronas*). Para ello, se evaluó el rendimiento del modelo (error cuadrático medio en entrenamiento y prueba) para diferentes valores del número de neuronas, en un rango comprendido entre 17 (raíz del número de muestras) y 51 (triple del valor inicial). Para calcular la posición de los centros se utilizó el algoritmo *k-means* que distribuye los centros de manera representativa sobre el espacio de entrada de modo que cada función cubra una región significativa de éste, adaptándose a la distribución real de los datos. La elección final de 18 centros se basó en el equilibrio entre la capacidad de generalización y la complejidad del modelo, observando la evolución de los errores en ambos conjuntos.

Como se aprecia en la Figura 4.2, los errores entre entrenamiento y prueba presentan una alta discrepancia para un número bajo de centros, lo que indica un modelo subajustado. A medida que el número de centros aumenta, ambos errores tienden a reducirse, acercándose entre sí, lo que sugiere una mejora en la capacidad de generalización. Sin embargo, a partir de cierto punto (por ejemplo, más de 45 centros), se observa un

comportamiento inestable del error de prueba, con picos pronunciados, lo que puede estar asociado a sobreajuste. En este contexto, el valor de 18 centros se identifica como una opción que ofrece un compromiso adecuado: mantiene errores relativamente bajos y estables tanto en entrenamiento como en prueba, evitando discrepancias extremas.

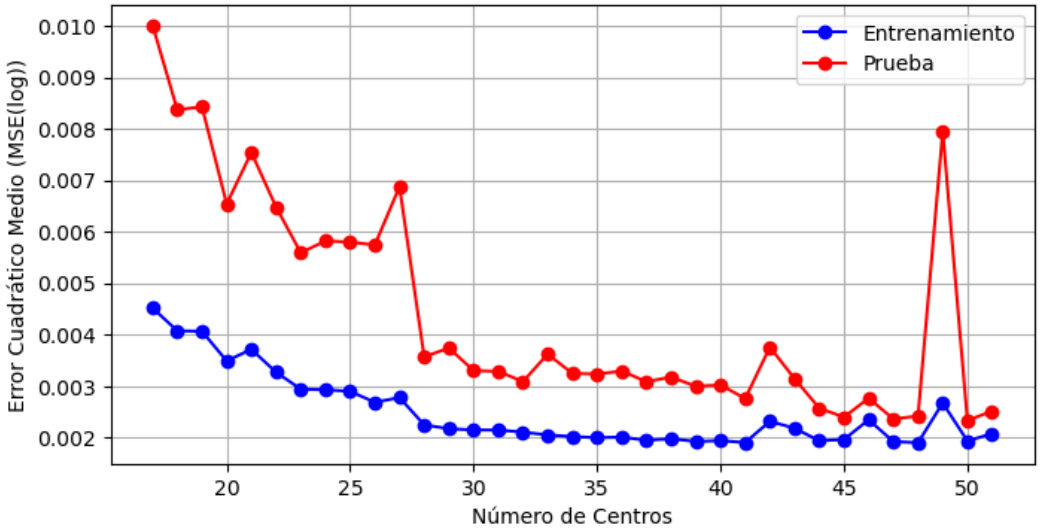

Figura 4.2: Comparación del error entre Entrenamiento y Prueba para diferentes números de centros.

El parámetro de dispersión de las funciones gaussianas σ_j se calculó como la media de las distancias euclídeas entre los centros obtenidos, un criterio habitual en este tipo de redes para asegurar una cobertura adecuada del espacio de entrada.

Una vez definidos los centros $\mathbf{c}_j$ y las anchuras σ_j, se calculan los pesos w_j resolviendo un problema de mínimos cuadrados. El objetivo es ajustar una combinación lineal de funciones de base radial φ_j que reproduzca lo mejor posible las salidas deseadas a partir de los datos de entrenamiento.

$$\hat{y}(\mathbf{x}) = \sum_{j=1}^{m} w_j \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}_j\|^2}{2\sigma_j^2}\right) + b \tag{4.1}$$

Los pesos w_j se determinan minimizando la suma de los errores al cuadrado entre las predicciones del modelo y los valores reales:

$$\min_{\mathbf{w},b} \frac{1}{2} \sum_{i=1}^{N} (y_i - \hat{y}(\mathbf{x}_i))^2 \tag{4.2}$$

Una vez entrenado, el modelo resultante se guarda en un archivo de tipo `.pkl`, lo que permite su posterior reutilización sin necesidad de reentrenarlo. En un segundo código independiente, destinado a la generación de gráficos, se introduce un intervalo de valores para ΔK y R, con el fin de representar visualmente las predicciones del modelo junto a los valores reales del conjunto de datos.

4.3.3. Modelado con Perceptrón Multicapa (MLP)

Para el modelado del *Perceptron Multicapa* se ha utilizando la clase `MLPRegressor` de la librería `scikit-learn`.

Durante el diseño de la arquitectura de la red se exploraron diferentes configuraciones, variando tanto el número de capas ocultas como el número de neuronas en cada capa (Figura 4.3). Para cada configuración, se registraron los errores en los conjuntos de entrenamiento y prueba, siguiendo un proceso iterativo similar al realizado con la RBFN. La arquitectura final seleccionada está compuesta por dos capas ocultas con 10 y 4 neuronas respectivamente.

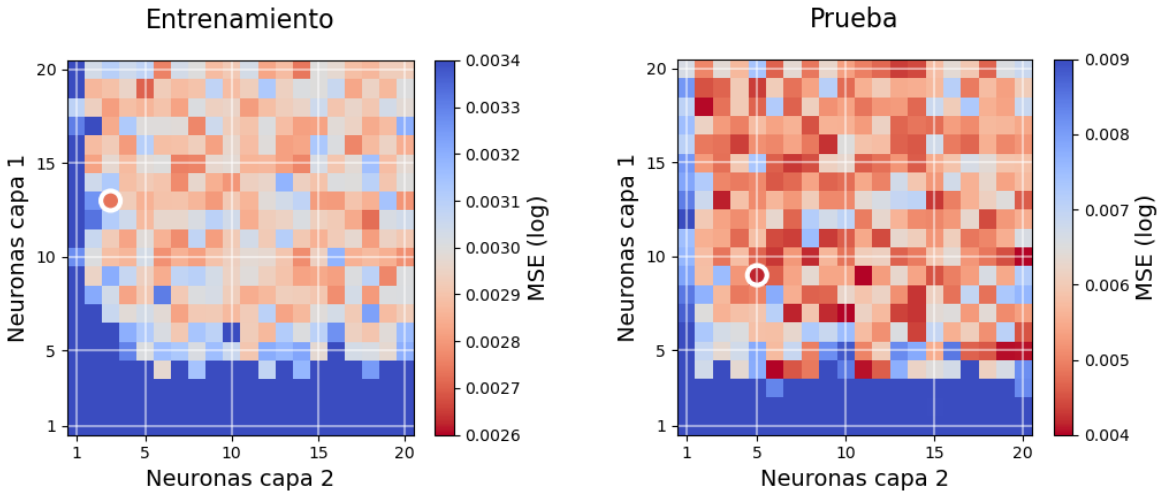

Figura 4.3: MSE en entrenamiento y prueba.

Se empleó la función de activación *tangente hiperbólica* (`tanh`), una elección clásica para problemas de regresión con datos normalizados y en dominio logarítmico, que proporciona una buena capacidad de modelado de relaciones no lineales suaves como es nuestro caso.

El optimizador seleccionado fue *L-BFGS*, un algoritmo quasi-Newton eficiente para conjuntos de datos de tamaño moderado, que tiende a converger en menos épocas que métodos basados en gradiente estocástico [18]. Se aplicó una regularización de tipo L2 con una *tasa de aprendiazaje* $\alpha = 0{,}001$, valor recomendado para reducir el riesgo de sobreajuste en redes neuronales entrenadas sobre datos normalizados [19].

El entrenamiento incorporó un mecanismo de parada temprana (*early stopping*) con un 15 % del conjunto de entrenamiento reservado para validación interna. La cantidad máxima de épocas se fijó en 1000, garantizando así una convergencia adecuada sin restricciones innecesarias.

El modelo resultante se guarda en un archivo de tipo `.pkl`, lo que permite su posterior reutilización sin necesidad de reentrenarlo.

4.4. Aluminio 2024-T351 (Conjunto 1)

4.4.1. Preparación de los datos

El aluminio 2024-T351 es una aleación con cobre que ofrece gran resistencia mecánica y buena maquinabilidad. Es común en estructuras aeronáuticas, gracias a su ligereza y rigidez, aunque tiene menor resistencia a la corrosión que otras aleaciones de aluminio.

El conjunto de datos utilizado en este apartado corresponde a medidas experimentales del comportamiento del aluminio 2024-T351. Se trabaja con un conjunto de datos experimentales obtenidos previamente por el Área de Materiales. El conjunto de datos está compuesto por ocho curvas correspondientes a distintos valores de la relación de carga ($R = 0{,}1$, $R = 0{,}3$, $R = 0{,}5$, $R = 0{,}7$) para dos espesores de probeta distintos (4 mm y 12 mm) y compuestas por un total de 370 puntos.

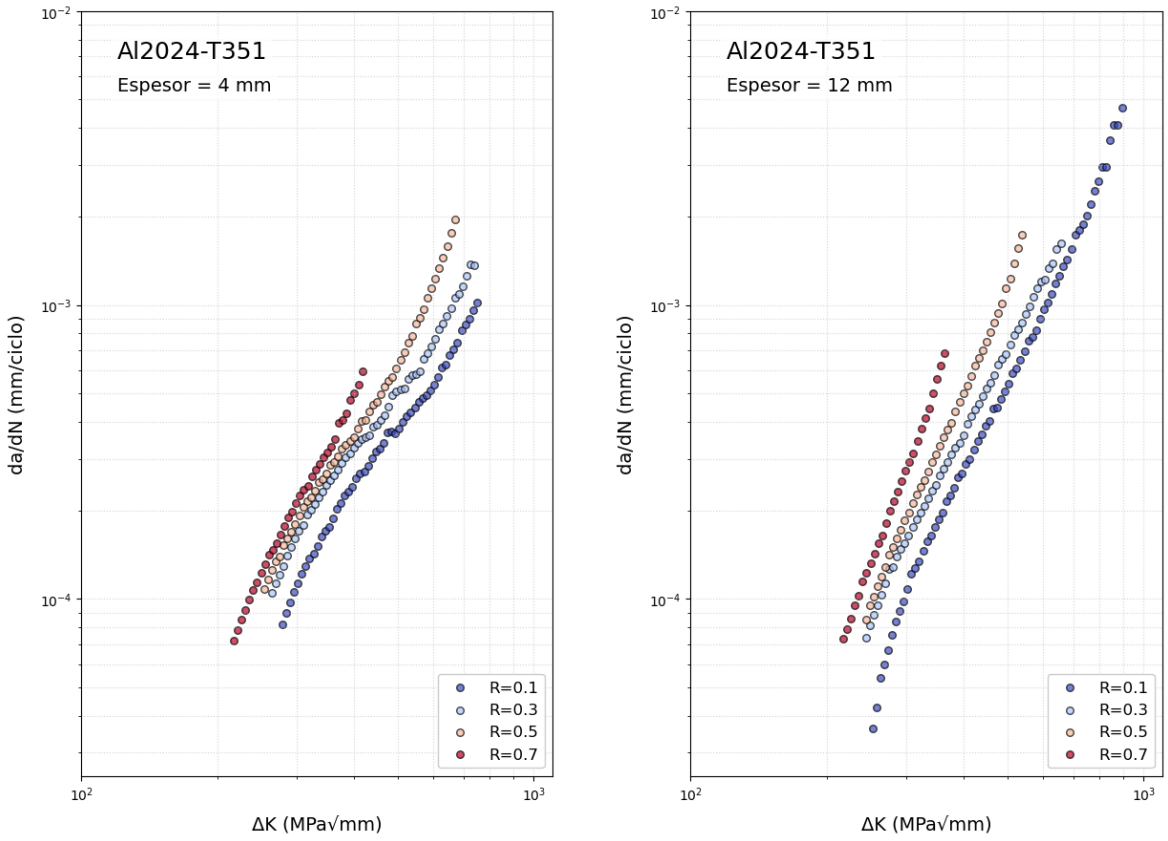

Figura 4.4: : Conjunto de datos perteneciente al aluminio 2024-T351: 4 mm y 12 mm.

El modelo se ha entrenado usando tres rasgos de entrada; ΔK, R y el *espesor*, y da/dN como variable de salida.

El archivo de datos empleado para el entrenamiento del modelo es un fichero en formato Excel, compuesto por las siguientes columnas:

da/dN (mm/ciclo)	ΔK (MPa$\sqrt{\text{mm}}$)	R	$espesor(mm)$

Cuadro 4.2: Estructura del fichero *Excel* utilizado para el entrenamiento del modelo.

4.4.2. Modelado con Red de Base Radial (RBFN)

Como en la Red de Base Radial anterior, se determino la posición de los centros mediante el algoritmo `K-Means`. Se seleccionó el número de centros a partir de un proceso exploratorio (Figura 4.5), evaluando el rendimiento del modelo en un intervalo de neuronas de 19 (raiz del numero de muestras) a 58 (triple del valor inicial). La elección final de 26 centros se basó en el equilibrio entre la capacidad de generalización y la complejidad del modelo, observando la evolución de los errores en ambos conjuntos.

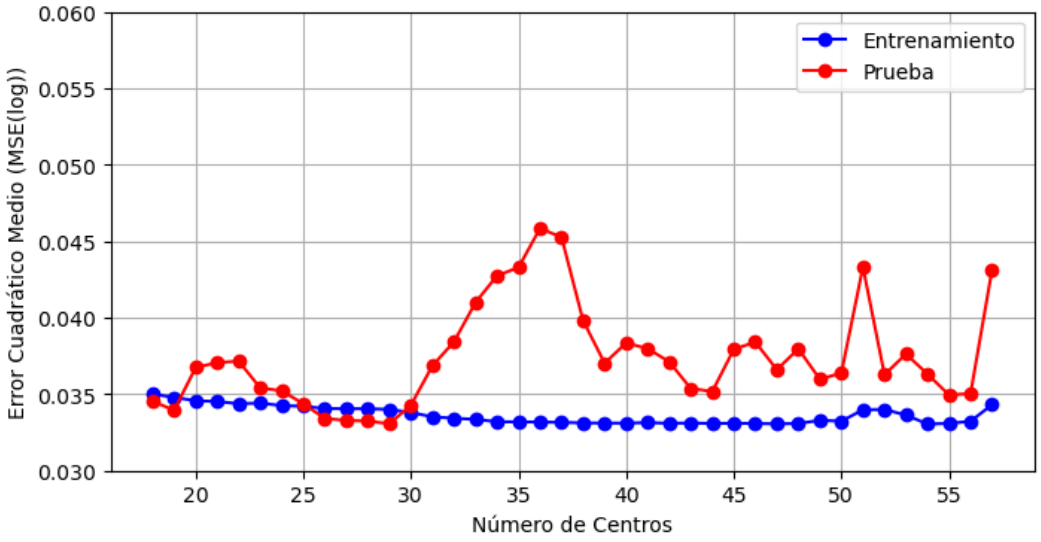

Figura 4.5: Comparación del error entre Entrenamiento y Prueba para diferentes números de centros.

El parámetro de dispersión σ_j se calculó como la media de las distancias euclídeas entre los centros obtenidos y se hallaron los pesos w_j minimazando la funcion de pérdida.

Una vez entrenado, el modelo resultante se guarda en un archivo de tipo `.pkl`, lo que permite su posterior reutilización sin necesidad de reentrenarlo.

4.4.3. Modelado con Perceptrón Multicapa (MLP)

Se determinó la configuración de neuronas en las dos capas ocultas mediante un análisis exploratorio (Figura 4.6). En ambos mapas de calor se aprecia con nitidez cómo varía el MSE (en escala logarítmica) al combinar distintos tamaños de las capas ocultas. En entrenamiento, el mínimo MSE se localiza en la configuración (13,4) —es

decir, 13 neuronas en la primera capa y 4 en la segunda—; de forma análoga, en prueba el par (11, 14) ofrece el error más bajo.

Aunque la combinación (13, 4) minimiza el error en entrenamiento y (11, 14) lo hace en prueba, ninguna de ellas ofrece a la vez buen ajuste y alta generalización. Al comparar arquitecturas vecinas, (10, 5) alcanza un MSE de prueba casi idéntico al mínimo absoluto y un error de entrenamiento ligeramente mayor, lo que refleja un modelo más estable. Además, al reducir una neurona por capa, se simplifica el diseño y se acelera el entrenamiento sin perder rendimiento. Por todo ello, (10, 5) es la configuración elegida como mejor compromiso entre precisión y robustez.

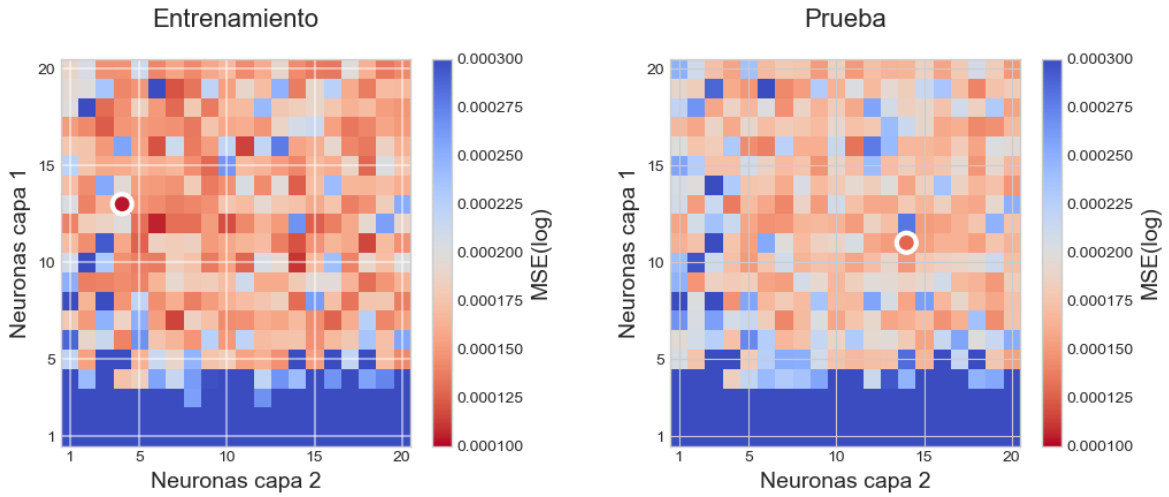

Figura 4.6: MSE en entrenamiento y prueba.

La arquitectura final del modelo y los hiperparámetros seleccionados se detallan a continuación:

- Dos capas ocultas con 10 y 5 neuronas respectivamente, función de activación tangente hiperbólica.
- Optimizador L-BFGS, adecuado para problemas de tamaño moderado con convergencia rápida.
- Regularización L2 con $\alpha = 0{,}001$, para controlar la varianza sin penalizar la capacidad de ajuste.
- Parada temprana activada.
- Número máximo de épocas: 1000, con semilla 42 para asegurar la reproducibilidad.

El entrenamiento definitivo fue almacenado como un archivo `.pkl`, incluyendo tanto los pesos de la red como los escaladores, permitiendo así su reutilización posterior en la generación de predicciones y visualización de resultados.

4.5. Aluminio 2024-T351 (Conjunto 2)

4.5.1. Preparación de los datos

Por último se ha trabajado con dos conjuntos de datos:

- Conjunto anterior: 8 curvas para espesores de 4 mm y 12 mm y relaciones de carga $R = 0{,}1$, $R = 0{,}3$, $R = 0{,}5$ y $R = 0{,}7$ (370 puntos).
- Conjunto ampliado: 36 curvas para espesores comprendidos entre 2 mm y 25.4 mm y relaciones de carga entre $R = 0{,}01$ y $R = 0{,}8$ (1808 puntos).

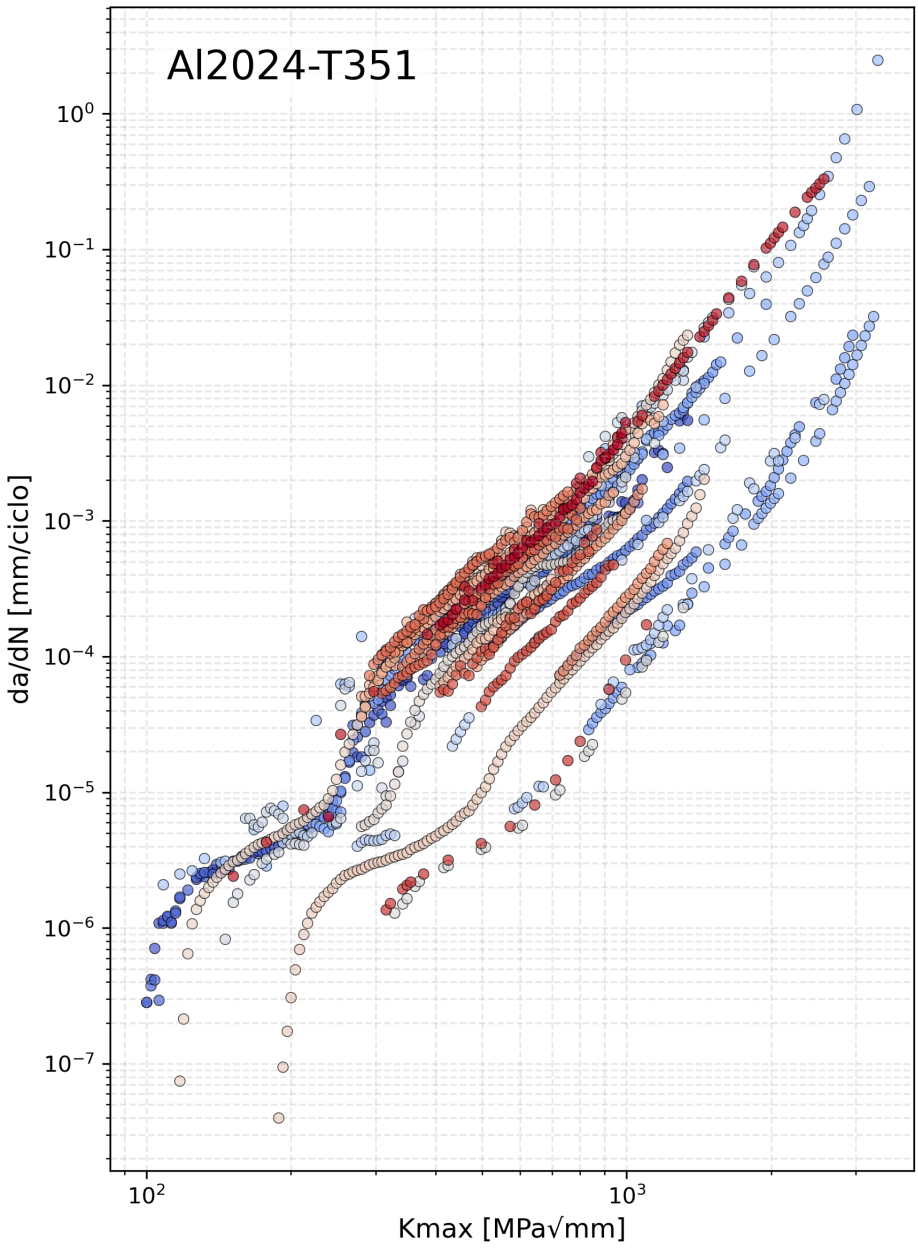

Figura 4.7: Conjunto de datos de aluminio 2024-T351 ampliado.

El modelo se ha entrenado usando tres rasgos de entrada; ΔK, R y el *espesor*, y da/dN como variable de salida.

El archivo de datos empleado para el entrenamiento del modelo es un fichero en formato Excel, compuesto por las siguientes columnas:

da/dN (mm/ciclo)	ΔK (MPa$\sqrt{\text{mm}}$)	R	$espesor(mm)$

Cuadro 4.3: Estructura del fichero *Excel* utilizado para el entrenamiento del modelo.

Dado que muchas de las curvas del nuevo conjunto presentaban datos muy dispares y con posible presencia de valores atípicos, se aplicó el algoritmo RANSAC (Random Sample Consensus) como método de detección y eliminación de *outliers*. Esta técnica consiste en ajustar un modelo a subconjuntos aleatorios de datos, identificando los puntos que mejor se ajustan al modelo estimado y descartando aquellos que se desvían significativamente. En este caso, cada curva fue ajustada individualmente mediante un polinomio de grado cinco, aplicando RANSAC de forma previa al entrenamiento del modelo, con el objetivo de conservar únicamente los datos representativos del comportamiento real del material (Figura 4.8).

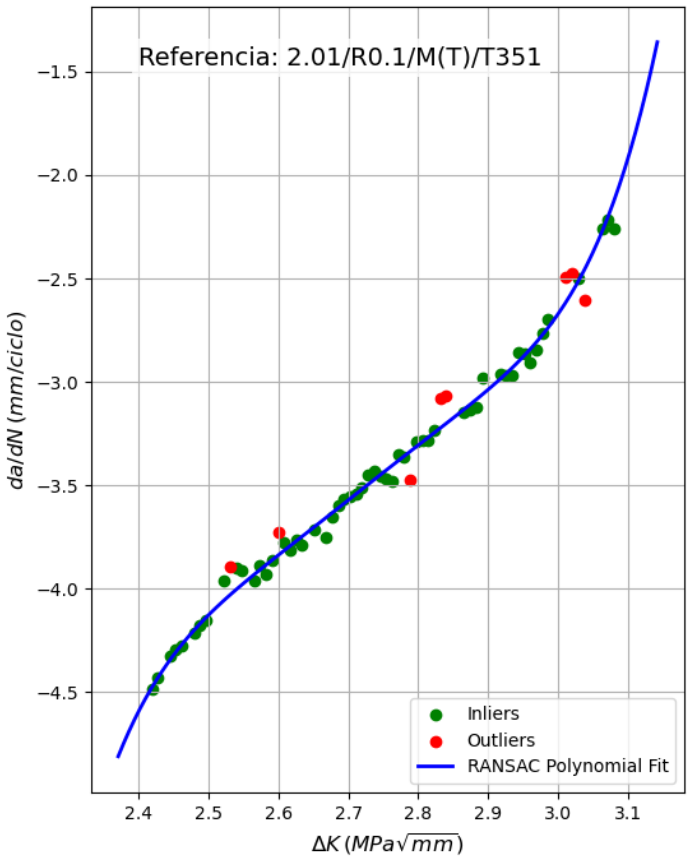

Figura 4.8: Detección de *outliers* sobre una curva mediante RANSAC.

De nuevo, se aplicó la transformación logarítmica decimal sobre ΔK y da/dN y el

escalado estándar sobre todas las variables. Por último, se hizo una división aleatoria de los datos en los conjuntos de entrenamiento y test del 80 % y del 20 % respectivamente, con una semilla fija para asegurar la reproducibilidad de los experimentos. También se reservaron un par de curvas completas para comprobar posteriormente la capacidad de predicción del modelo.

4.5.2. Análisis exploratorio de los datos

Con el objetivo de comprender mejor la estructura del conjunto de datos antes del entrenamiento del modelo, se ha realizado un análisis exploratorio que incluye tanto visualizaciones de unas variables frente a otras, como el cálculo de la matriz de correlación de Pearson. Para ello, se ha aplicado una transformación logarítmica en base 10 a las variables da/dN y ΔK, con el fin de linealizar su relación.

En la matriz de correlación (Figura 4.10) se observa una fuerte correlación positiva entre $\log_{10}$(da/dN) y $\log_{10}(\Delta K)$ (r $\approx$ 0.95), lo que valida la importancia de esta variable como rasgo principal en el modelado. En contraste, la variable R presenta una correlación negativa moderada con ΔK (r $\approx$ -0.40) y débil con da/dN, lo que sugiere un posible efecto indirecto o no lineal. El espesor, por su parte, muestra correlaciones débiles con el resto de las variables, indicando que podría no tener un papel determinante en el comportamiento general del conjunto de datos.

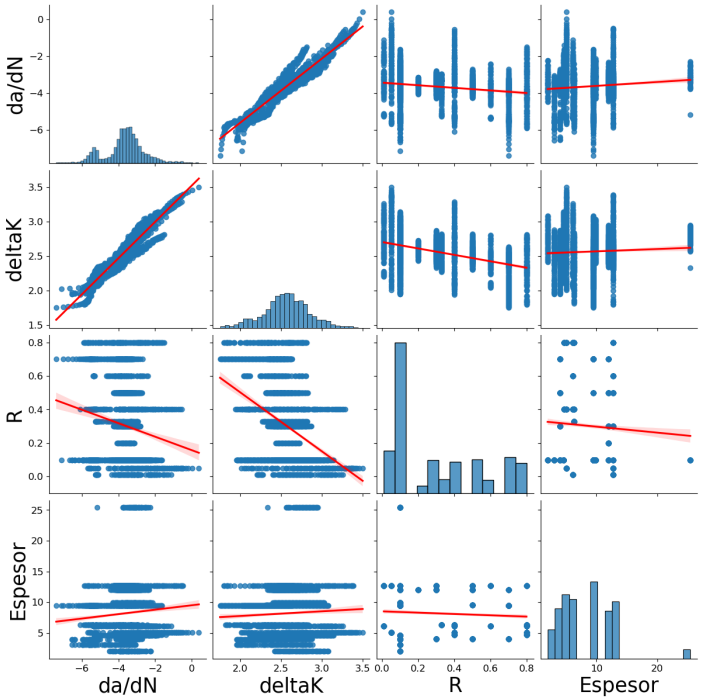

Figura 4.9: Correlaciones lineales entre los rasgos.

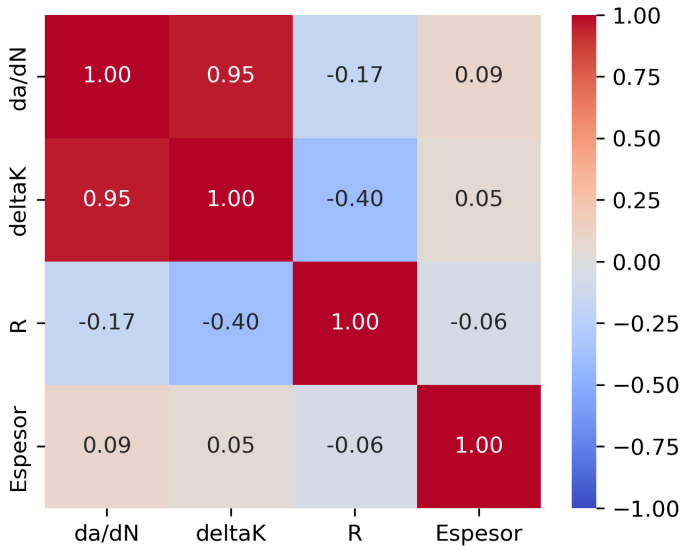

Figura 4.10: Matriz de correlación de Pearson.

En cuanto al gráfico de espesor frente a R (Figura 4.11), se observa que la gran mayoría de los datos están concentrados en espesores entre 5 mm y 12 mm. Destaca la presencia de una única curva correspondiente a un espesor de 25 mm, lo que indica una representatividad limitada de esa condición experimental concreta. Esta curva ha sido eliminada por encontrarse muy alejada del resto de datos.

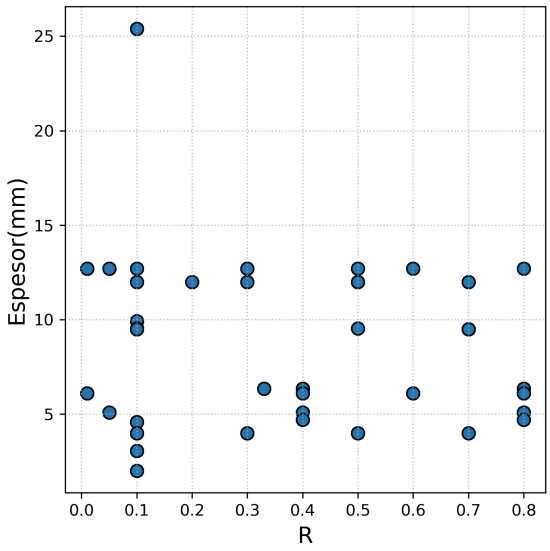

Figura 4.11: Distribución de espesores frente a R.

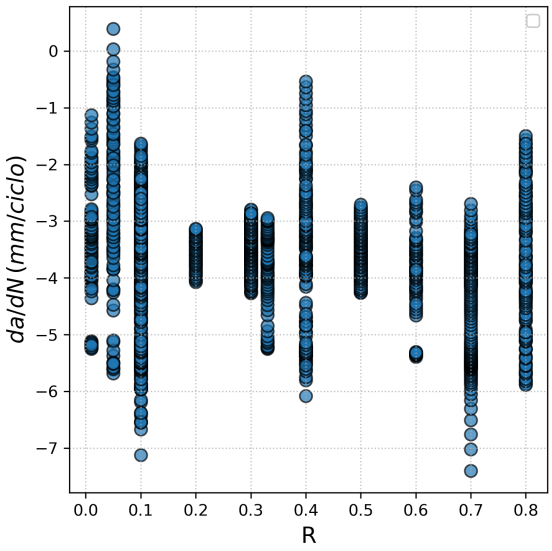

Figura 4.12: Dispersión de da/dN en función del valor de R.

4.5.3. Esquema de validación K-Fold

Para contrastar la capacidad de generalización antes de fijar definitivamente la arquitectura, el conjunto completo de datos se sometió a una validación cruzada estratificada de cinco *folds*, manteniendo en cada caso la proporción de los distintos valores del parámetro R. En los experimentos preliminares el muestreo disponible era tan reducido que aplicar K-fold habría generado *folds* casi vacíos y métricas poco fiables; con el incremento actual de datos cada *fold* contiene ya suficientes ejemplos, de modo que la técnica resulta tanto viable como informativa.

Métrica	**Valor promedio**
Error cuadrático medio (MSE)	$5{,}96 \times 10^{-3}$
Coeficiente de determinación (R^2)	0,9924
Tiempo de entrenamiento por fold	0,73 s

Cuadro 4.4: Resultados medios obtenidos en validación cruzada (5-Fold).

Durante cada iteración se registraron el error cuadrático medio (MSE) y el coeficiente de determinación R^2 sobre los datos de prueba. Los resultados muestran un rendimiento alto, con errores bajos y valores de R^2 próximos a 1 en todos los pliegues. Además, el tiempo de entrenamiento por iteración fue reducido, lo que hace que el proceso sea eficiente computacionalmente. En conjunto, la validación confirma que el modelo no depende de una partición concreta y generaliza correctamente en todo el dominio experimental.

4.5.4. Modelado con Perceptrón Multicapa (MLP)

Se determino el número de neuronas mediante un proceso exploratorio (Figura 4.13). Al aumentar de 1 a 7 neuronas el modelo mejora de forma muy notable (rápida subida de R^2 y gran reducción de MSE); sin embargo, a partir de 8 a 10 neuronas se estabiliza ($R^2 \approx 0{,}99$ y $MSE \approx 0{,}005$) y añadir más unidades apenas mejora el resultado pero sí aumenta el coste computacional. Por ello se eligieron 10 neuronas: el punto óptimo donde se obtiene casi todo el beneficio de complejidad sin comprometer la eficiencia.

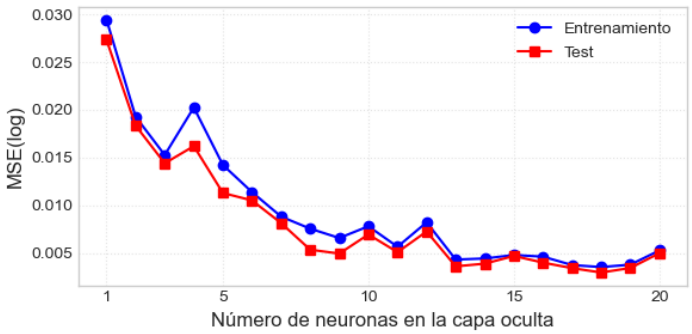

Figura 4.13: : Comparación del error entre Entrenamiento y Prueba para diferentes números de centros.

Se unieron los dos conjuntos de datos de la siguiente forma:

- Dado que las curvas pertenecientes a los espesores 4 mm y 12 mm son las de mejor calidad (*Conjunto 1*), las que presentan menos ruido y forman un conjunto equilibrado, se ponderaron para que el modelo se ciñiese y aprendiese más sobre ellas.
- Curvas depuradas (*inliers* validados) sin ponderación adicional (*Conjunto 2*).

Los hiperparámetros y criterios de optimización utilizados en el entrenamiento del modelo fueron los siguientes:

- Optimizador L-BFGS, elegido por su rapidez de convergencia en lotes pequeños.
- Regularización L2 con $\alpha = 0{,}001$, suficiente para atenuar la varianza sin penalizar la capacidad de ajuste.
- Número máximo de iteraciones (épocas): 10.000, con semilla 42 para garantizar la reproducibilidad.

El entrenamiento definitivo fue almacenado como un archivo `.pkl`, incluyendo tanto los pesos de la red como el escalador, para su posterior reutilización en la fase de análisis y visualización de resultados.

4.6. Proceso de evaluación

La evaluación de ambos modelos se llevó a cabo sobre el conjunto de prueba reservado, y se basó tanto en métricas cuantitativas como en análisis visual complementario.

Las métricas cuantitativas consideradas fueron:

- **Error cuadrático medio (MSE)**, calculado en la escala original de los datos.
- **Coeficiente de determinación ($\mathbf{R^2}$)**, evaluado en la escala original.

Asimismo, se registraron el tiempo de entrenamiento y el uso de memoria asociados a cada modelo, como indicadores adicionales de eficiencia computacional.

Además de las métricas numéricas, se contempló un análisis visual de los resultados, mediante la representación gráfica de las predicciones frente a los valores reales y de los errores residuales. Dichas visualizaciones permiten evaluar cualitativamente la capacidad de generalización de los modelos y detectar posibles patrones o sesgos en las predicciones.

5. Resultados y discusión

En esta sección se presentan y analizan los resultados obtenidos mediante la aplicación de los modelos de **Red de Base Radial (RBFN)** y **Perceptrón Multicapa (MLP)** sobre las bases de datos del titanio Ti-6Al-4V y el aluminio 2024-T351.

Tras describir las métricas cuantitativas obtenidas por ambos modelos, se procede a una evaluación visual complementaria basada en la comparación entre predicciones y valores reales.

Además del análisis puramente cuantitativo, se prestará especial atención a la interpretación física de los resultados, evaluando si el comportamiento aprendido por los modelos es coherente con los principios conocidos de la propagación de grietas por fatiga en materiales metálicos. De este modo, se pretende validar no solo la capacidad predictiva de los modelos, sino también su consistencia con el conocimiento teórico del fenómeno estudiado.

5.1. Titanio Ti-6Al-4V

Las métricas de rendimiento obtenidas por ambos modelos se resumen en la siguiente tabla:

Modelo	**MSE (log)**	**R^2**	**Tiempo (s)**	**Memoria (kB)**
RBFN	1.9602e-04	0.8951	0.06	133
MLP	1.2288e-04	0.9237	0.10	28

Cuadro 5.1: Resumen de métricas de rendimiento obtenidas por los modelos RBFN y MLP para el titanio.

Los resultados muestran que ambos modelos logran un ajuste adecuado de los datos, con coeficientes de determinación R^2 superiores ceracnos o superiores a 0.9.

El modelo MLP presenta un rendimiento cuantitativo ligeramente superior al modelo RBFN, tanto en términos de MSE como de R^2.

En cuanto a la eficiencia computacional, el modelo RBFN resulta considerablemente más rápido de entrenar y requiere menos memoria, lo que puede ser una ventaja en entornos con restricciones de recursos.

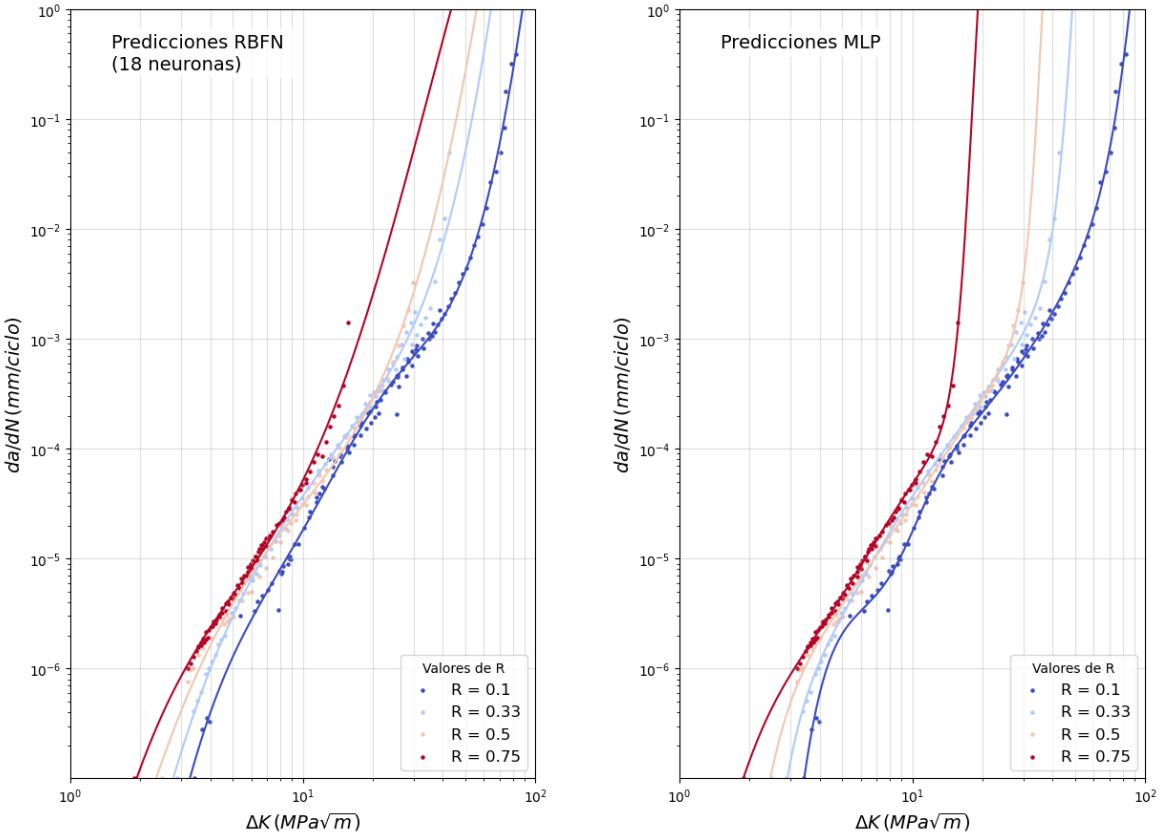

Figura 5.1: Curvas da/dN frente a ΔK para RBFN y MLP.

Al analizar las predicciones del MLP y de la RBFN (Fig. 5.1), se observan varios puntos clave:

1. **Zona I (umbral):** Ambas redes reproducen correctamente la abrupta caída de $\mathrm{d}a/\mathrm{d}N$ al aproximarse al rango umbral de crecimiento, lo que indica que los modelos capturan adecuadamente la transición hacia velocidades prácticamente nulas cuando ΔK se acerca a ΔK_{th}.

2. **Zona II (zona de Paris).** En el rango intermedio, donde Paris predice una relación lineal en escala log–log ($\mathrm{d}a/\mathrm{d}N \propto \Delta K^m$), las predicciones de ambas arquitecturas siguen muy de cerca esa pendiente lineal. El MLP tiende a ajustarse ligeramente mejor, captando las no linealidades presentes en valores de R bajos.

3. **Zona III (propagación inestable).** Cuando ΔK crece hacia valores altos, el MLP capta mucho mejor el fuerte repunte de $\mathrm{d}a/\mathrm{d}N$ que conduce al colapso final (zona III). Este "carácter vertical" en la parte derecha del gráfico confirma que los modelos han aprendido implícitamente el valor crítico K_c, más allá del cual la grieta se propaga de forma inestable.

4. **Efecto de R.** Para valores de R bajos solo el MLP ha sico capaz de capturar las no linealidades que se presentan en las curvas.

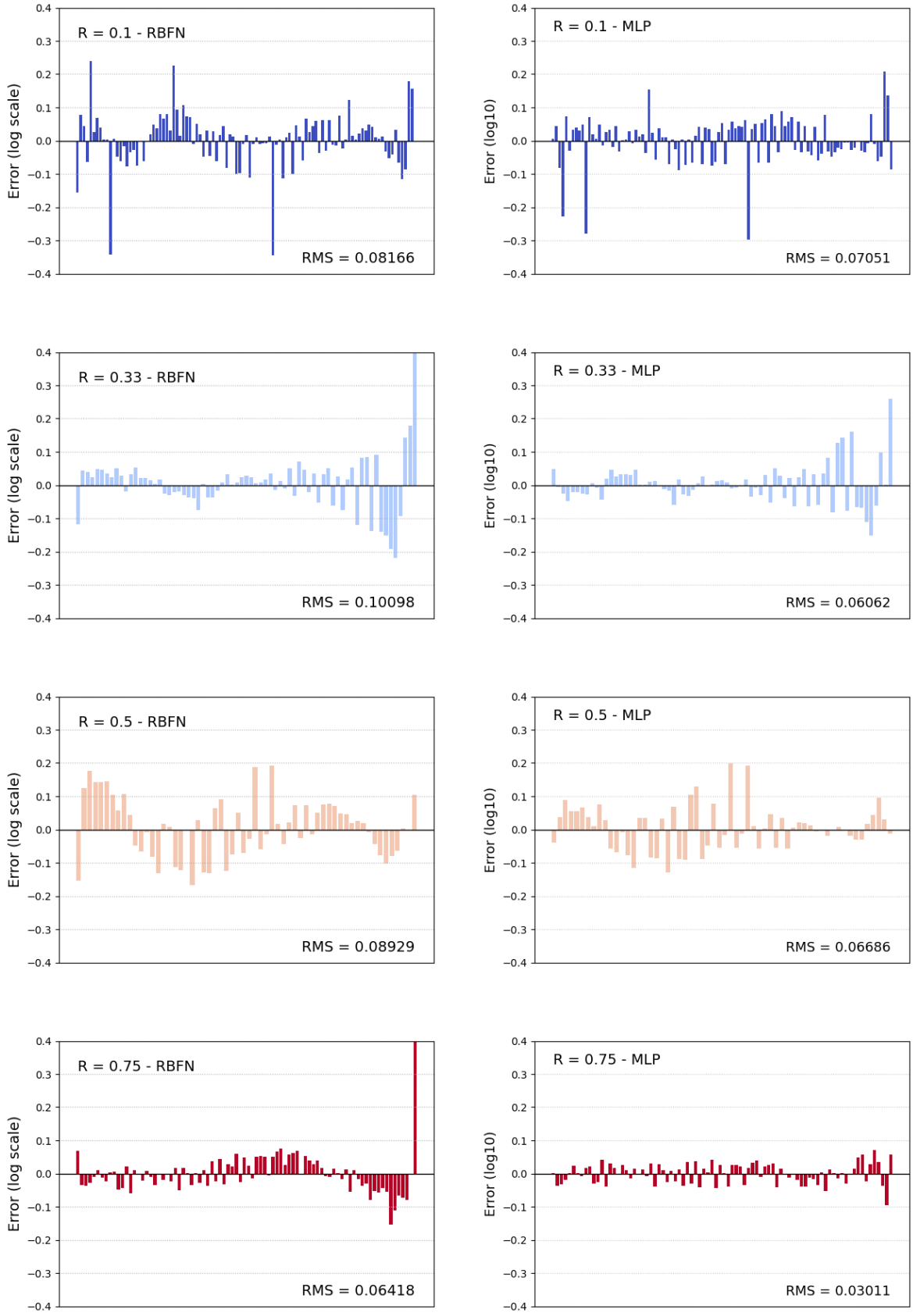

Figura 5.2: Comparación del MSE ente RBNF y MLP para cada valor de R.

En todas las condiciones de carga (R), el MLP presenta errores más bajos y distribuciones más centradas respecto a la RBFN, destacando especialmente en R = 0.75, donde su RMS se reduce a la mitad. Esto sugiere una mejor capacidad del MLP para generalizar y ajustar los datos en todas las curvas analizadas (Figura 5.2).

Influencia del factor de carga R

En las predicciones de la RBFN para valores de R con los que no ha sido entrenada(Fig. 5.3), las curvas correspondientes a distintos valores de R se entrelazan entre sí, lo que evidencia que el modelo no discrimina correctamente los comportamientos cuando R varía de manera continua. En contraste, las curvas obtenidas con el MLP mantienen siempre su orden relativo (no se cruzan), mostrando así una evolución coherente y consistente del crecimiento de grieta a lo largo de todo el rango de relaciones de carga.

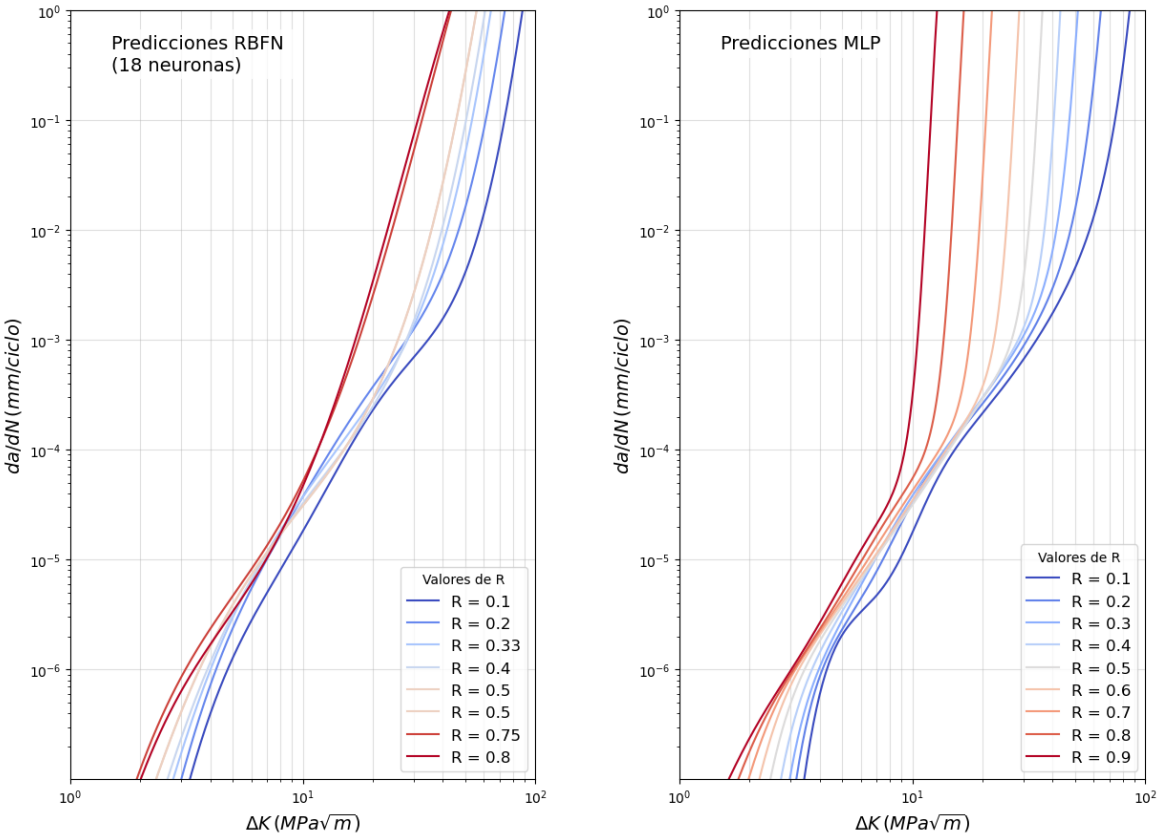

Figura 5.3: Curvas para valores de $R \in [0.1, 0.9]$.

En conjunto, el MLP y la RBFN no sólo han ajustado bien los datos experimentales, sino que también respetan los comportamientos límite de la mecánica de fractura por fatiga: las regiones de umbral y de colapso rápido (zonas I y III) aparecen naturalmente en sus predicciones. El modelo MLP se muestra superior en capacidad de generalizar y predecir sobre curvas con las que no ha sido entrenado asi como capturando las no linealidades que aparacen en valores bajos de R.

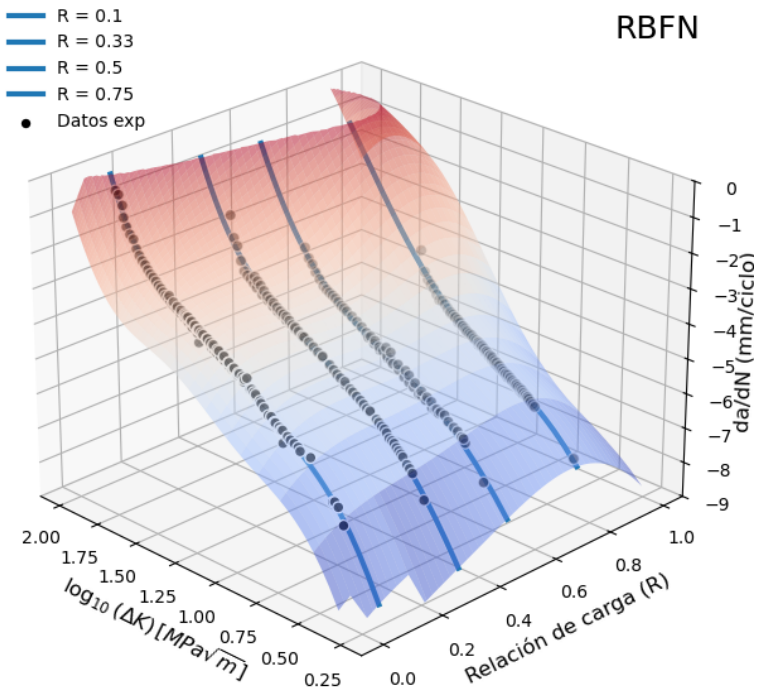

Figura 5.4: Superficies da/dN frente a ΔK, con $R \in [0.1, 0.9]$.

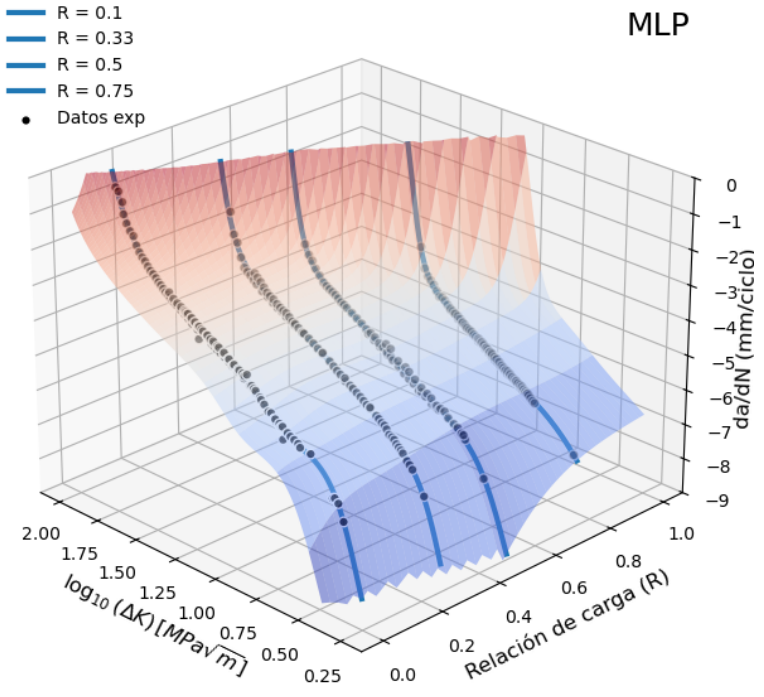

Figura 5.5: Superficies da/dN frente a ΔK, con $R \in [0.1, 0.9]$.

5.2. Aluminio 2024-T351 (Conjunto 1)

Las métricas de rendimiento obtenidas por ambos modelos se resumen en la siguiente tabla:

Modelo	**MSE (log)**	**R^2**	**Tiempo (s)**	**Memoria (kB)**
RBFN	3.5166e-09	0.9923	0.16	248
MLP	2.0966e-09	0.9954	0.07	12

Cuadro 5.2: Resumen de métricas de rendimiento obtenidas por los modelos RBFN y MLP para el aluminio 2024-T351 con dos espesores.

Los resultados muestran que ambos modelos logran un ajuste adecuado de los datos, con coeficientes de determinación R^2 superiores a 0.99 y errores cuadráticos medios reducidos.

El modelo MLP presenta un rendimiento cuantitativo ligeramente superior al modelo RBFN, tanto en términos de MSE como de R^2, así como en la eficiencia computacional. Requiere menos memoria y tiempo que la RBFN

Al analizar las predicciones del MLP (Fig. 5.7) y de la RBFN (Fig. 5.6), se observan varios puntos clave:

- **Zona I (umbral):** en la zona baja de ΔK (cercana a ΔK_{th}), tanto la RBFN como el MLP reproducen el brusco descenso de da/dN hacia valores prácticamente nulos, tal como establece la teoría de umbral.

- **Zona II (zona de Paris):** ambas arquitecturas siguen la pendiente lineal esperada en escala log–log ($da/dN \propto (\Delta K)^m$), si bien la RBFN tiende a aplanar las pequeñas inflexiones propias de R medios, mientras que el MLP mantiene con mayor fidelidad la ligera no linealidad de la región.

- **Zona III (propagación inestable):** el MLP muestra claramente el repunte casi vertical de da/dN al acercarse a K_c, cumpliendo el comportamiento asintótico en su ultimo tramo; en cambio, la RBFN (especialmente para 4 mm) desvía las curvas hacia la derecha en lugar de elevarlas, rompiendo ese carácter asintótico y subestimando el abrupto salto hacia la fractura.

Aunque ambas redes respetan adecuadamente las regiones I y II, solo el MLP conserva la respuesta asintótica vertical de la región III mientras que la RBFN falla al modelar correctamente la subida final.

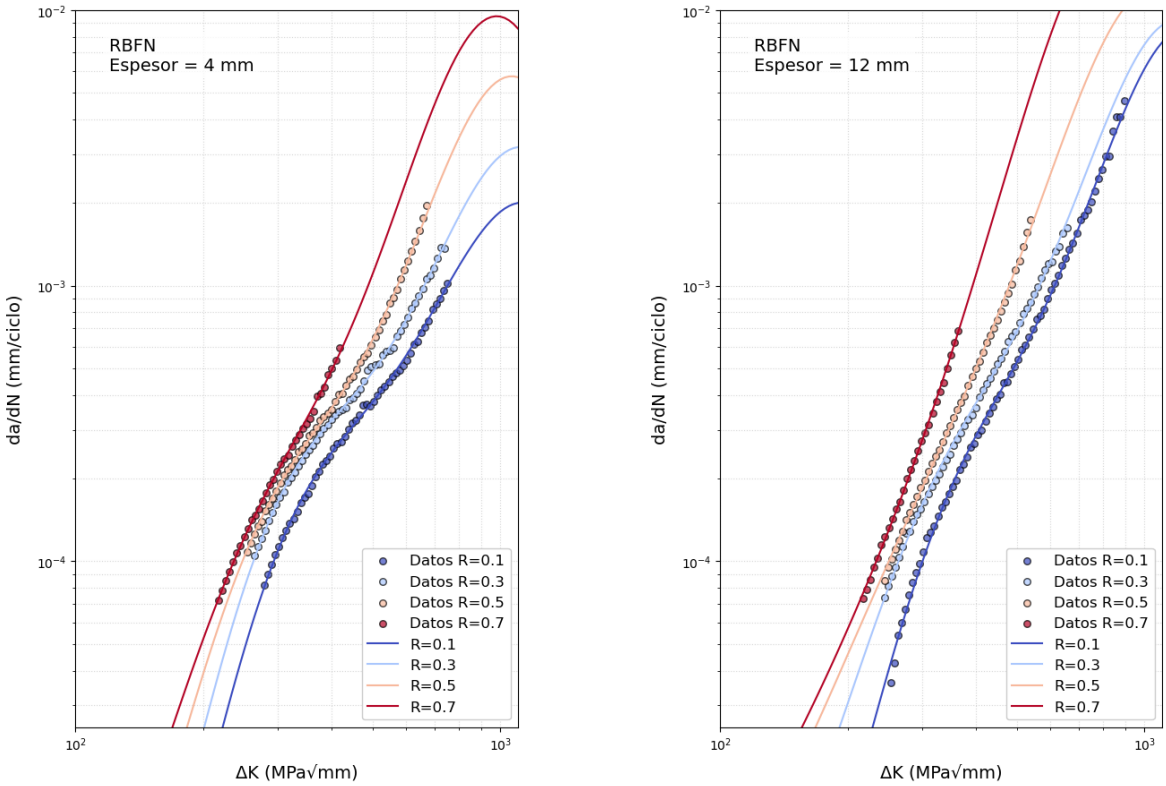

Figura 5.6: RBFN: Curvas da/dN frente a ΔK para 4 mm y 12 mm.

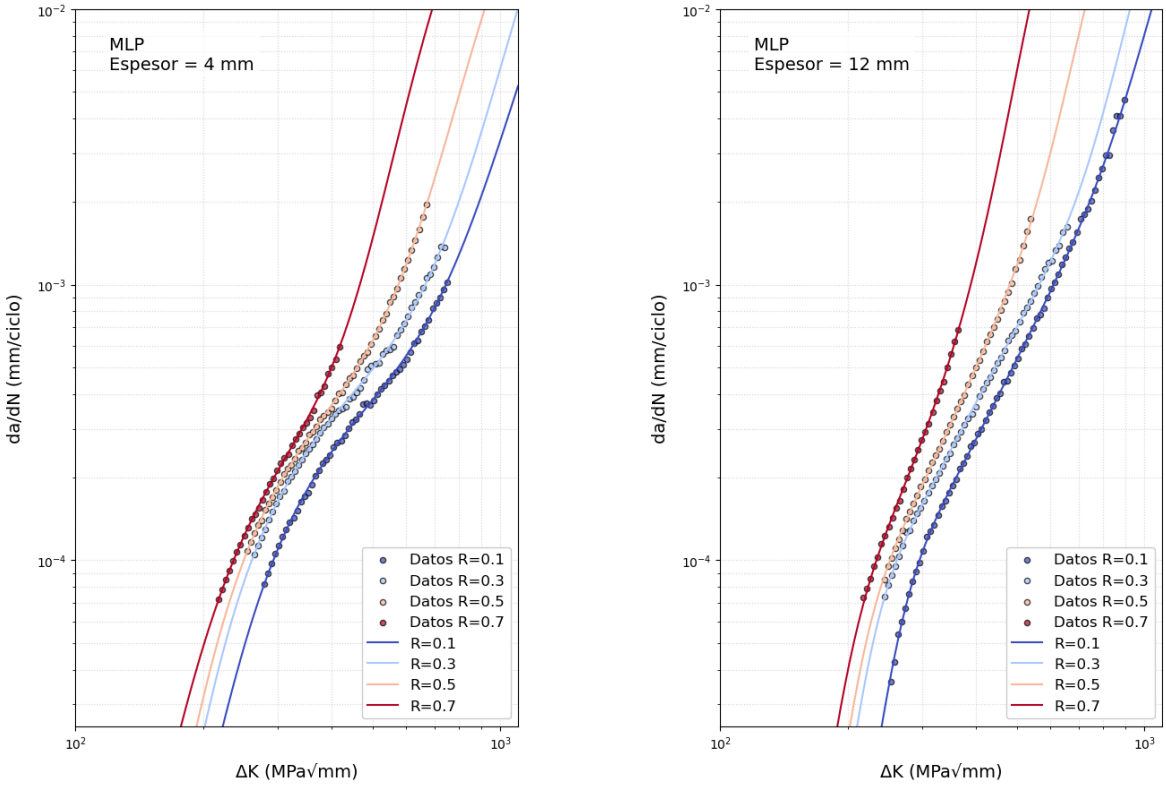

Figura 5.7: MLP: Curvas da/dN frente a ΔK para 4 mm y 12 mm.

Influencia del factor de carga R

Observando las predicciones de la RBFN y MLP de curvas da/dN frente a ΔK para distintas relaciones de carga dentro de un mismo espesor se puede destacar:

- **RBFN** (Fig. 5.8)**:** Las curvas siguen el orden esperado a medida que aumenta el valor del factor de carga R, mostrando una separación progresiva y coherente. Para valores altos de R, las predicciones tienden a solaparse, lo que indica una menor sensibilidad del modelo en esa región. En la Zona III (región de propagación rápida), algunas curvas se cruzan ligeramente o muestran descensos anómalos en la parte superior del gráfico. Este comportamiento se debe a la naturaleza local de las funciones de base radial, las cuales tienden a decrecer fuera de su región de influencia provocado este tipo de comportamiento en regiones alejadas del los datos.

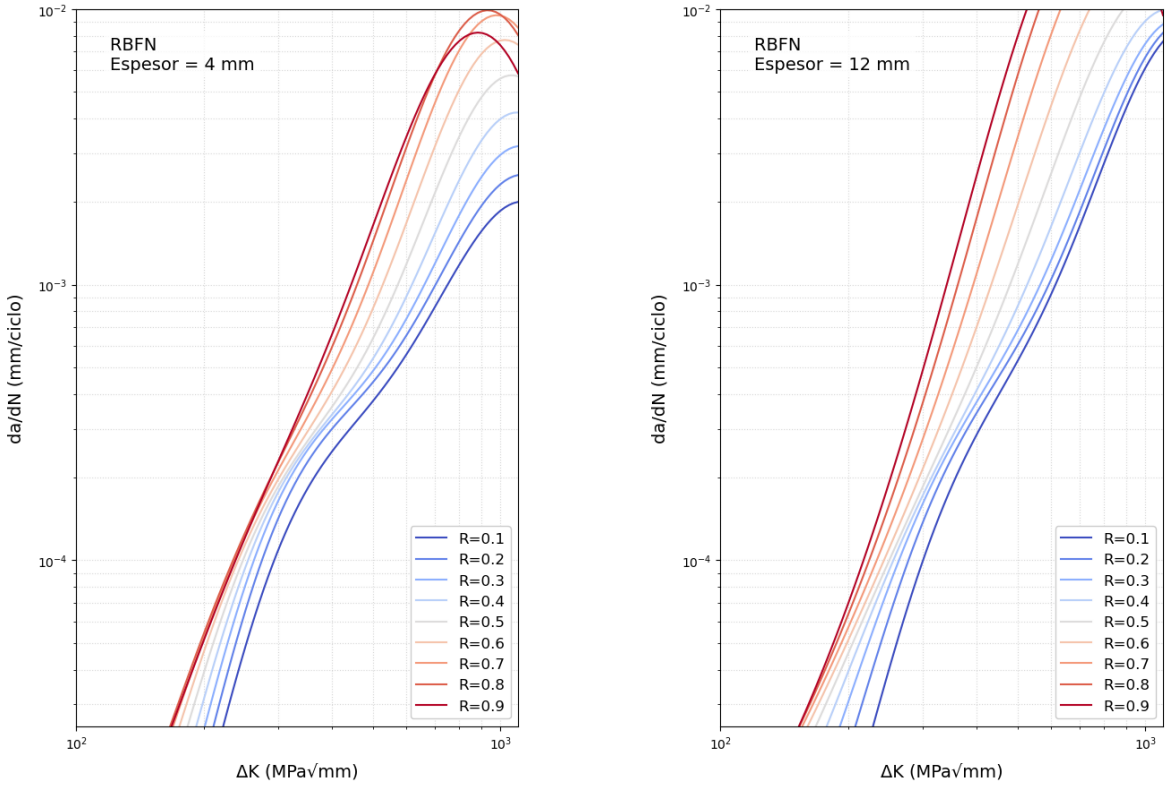

Figura 5.8: RBFN: Curvas da/dN frente a ΔK para $R \in [0.1, 0.9]$ en espesores de 4 mm y 12 mm.

- **MLP** (Fig. 5.9)**:** Las curvas para distintos valores de R permanecen siempre ordenadas y ascienden "hacia arriba" a medida que R crece, sin cruzarse entre sí.

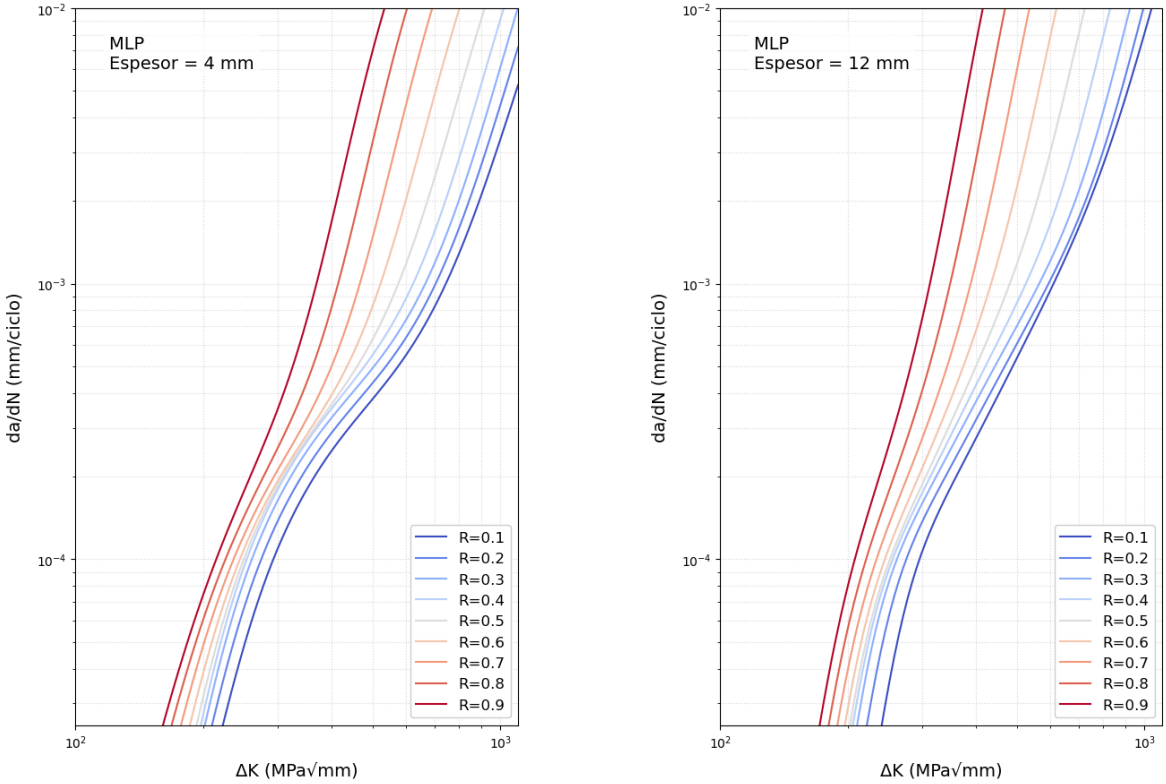

Figura 5.9: MLP: Curvas da/dN frente a ΔK para $R \in [0.1, 0.9]$ en espesores de 4 mm y 12 mm.

Solo el MLP mantiene el orden relativo de las curvas de da/dN vs ΔK, mientras que la RBFN (especialmente para 4 mm) no consigue modelar correctamente el comportamiento asintóticode la Zona III.

Aunque la RBFN ha mostrado un comportamiento inferior al del MLP en la predicción de curvas para valores intermedios de R, los resultados obtenidos son claramente superiores a los obtenidos previamente al aplicar la misma arquitectura sobre el conjunto de datos del titanio. Esta mejora puede atribuirse, en gran medida, al aumento del número de muestras disponibles, lo que ha permitido una representación más completa del dominio y un entrenamiento más efectivo de la red radial.

En las siguientes figuras se representan las superficies de predicción generadas por la RBFN y el MLP para distintas relaciones de carga $R \in [0, 1]$, con espesor constante de 4 mm y 12 mm respectivamente. Se muestran las curvas da/dN frente a ΔK, obtenidas mediante interpolación sobre la malla continua de entrada.

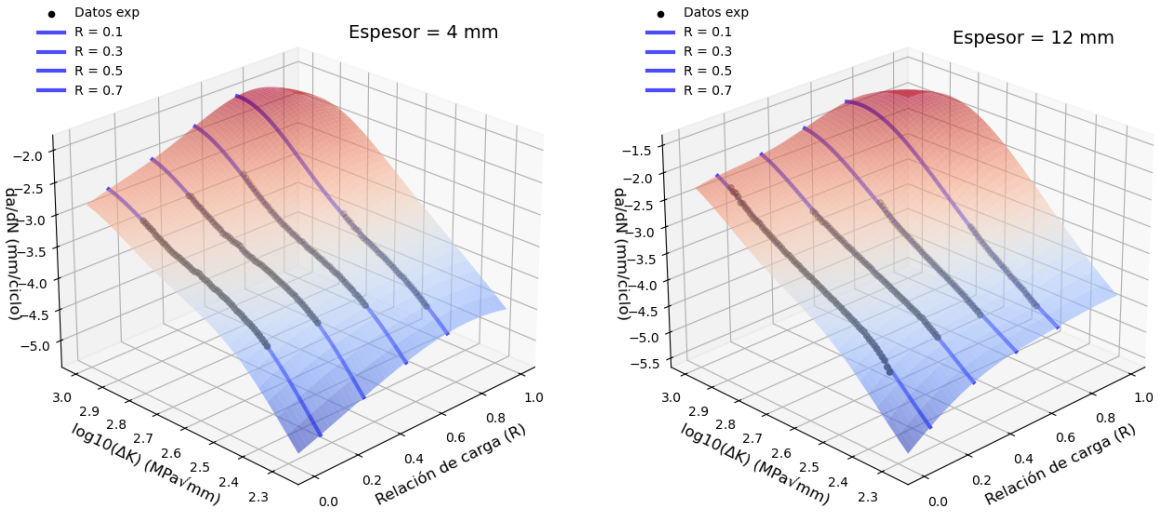

Figura 5.10: Superficies de predicción de la RBFN para curvas da/dN frente a ΔK, con $R \in [0, 1]$.

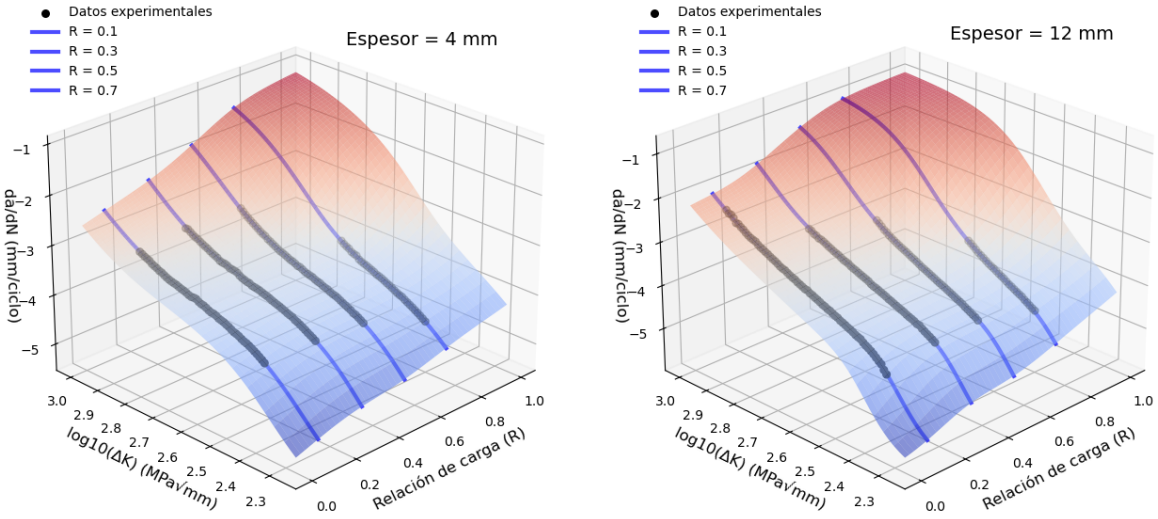

Figura 5.11: Superficies de predicción del MLP para curvas da/dN frente a ΔK, con $R \in [0, 1]$.

Estas superficies permiten visualizar cómo varía la respuesta del modelo en función de R, evidenciando tanto la progresión esperada del crecimiento de grieta como la suavidad del ajuste. Es especialmente destacable que el MLP mantiene una curvatura suave y continua sin solapamientos, incluso entre valores de R no observados durante el entrenamiento, lo que refuerza su capacidad de generalización. En cambio, aunque la RBFN presenta un ajuste razonable en las zonas cubiertas por los datos, su superficie muestra cierta rigidez y pérdida de pendiente para valores altos de ΔK, especialmente en el caso de menor espesor.

Influencia del espesor

Observando las gráficas para las predicciones de curvas da/dN frente a ΔK con distintos espesores para la misma relación de cargas, sacamos las siguientes conclusiones:

- **RBFN** (Fig. 5.12)**:** las curvas para los distintos espesores quedan entrelazadas, sin respetar el orden físico (*a mayor espesor, mayor* da/dN). Al suavizar en exceso, la red radial pierde la capacidad de generalizar cómo cambia la propagación de grieta al variar el espesor.

- **MLP** (Fig. 5.13)**:** las predicciones del perceptrón multicapa mantienen la jerarquía correcta: las curvas correspondientes a mayores espesores quedan sistemáticamente por encima de las más finas, replicando fielmente el desplazamiento observado en las curvas experimentales a lo largo de todo el rango de espesores.

Solo el MLP ofrece un modelado robusto de la dependencia con el espesor, mientras que la RBFN no logra capturar el efecto monotónico que imponen las variaciones geométricas del probeta.

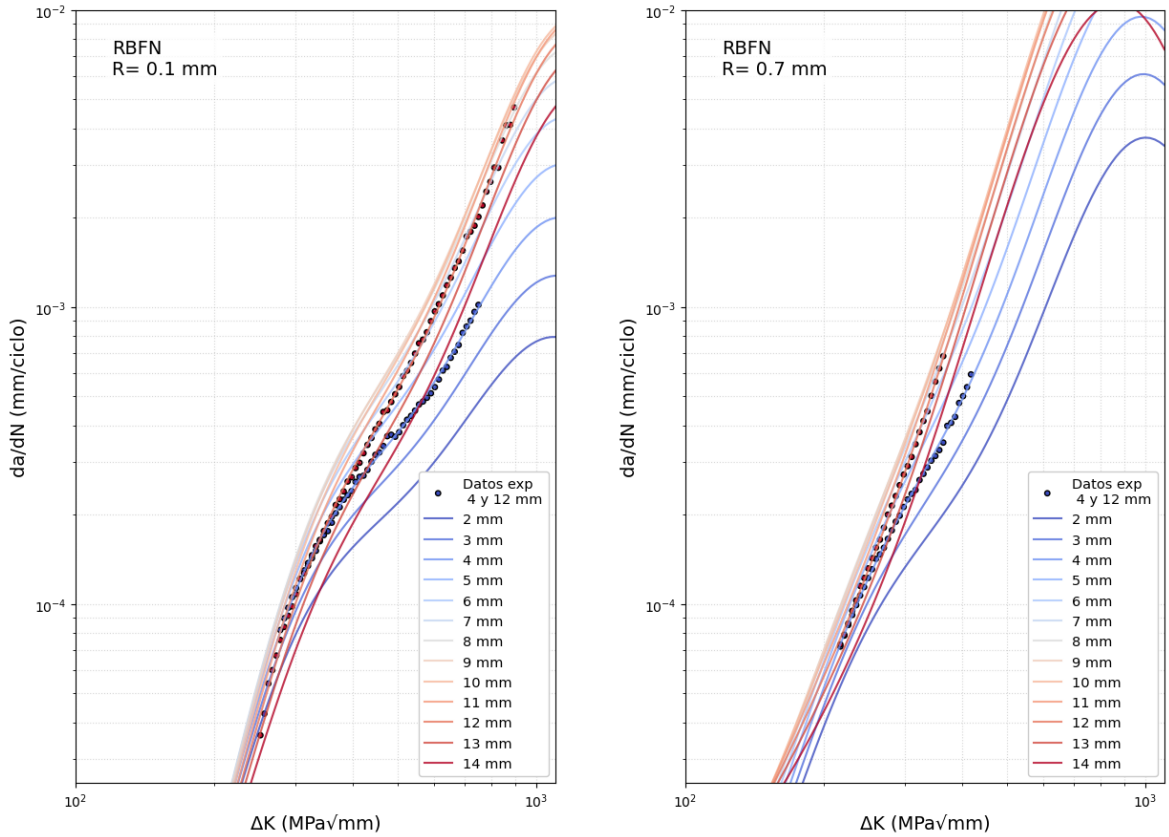

Figura 5.12: RBFN: Curvas da/dN frente a ΔK para espesores entre 2 mm y 14 mm para $R = 0{,}1$ y $R = 0{,}7$

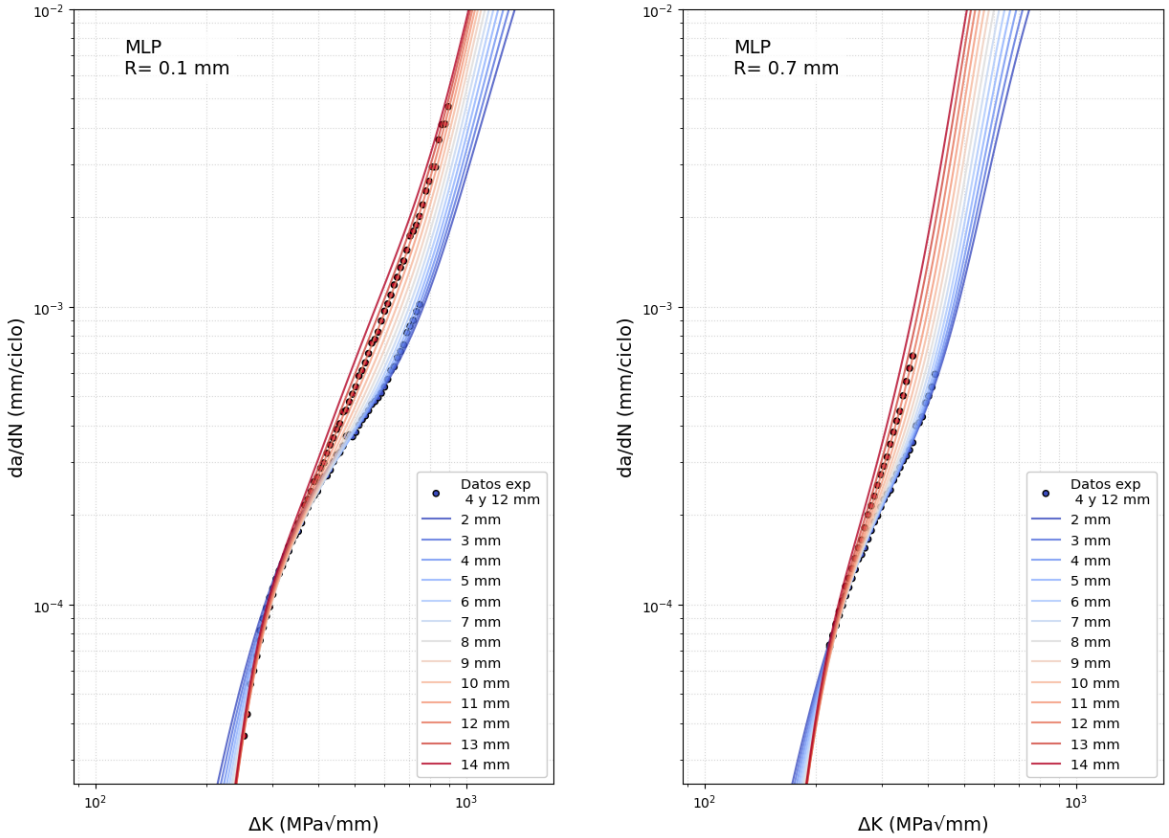

Figura 5.13: MLP: Curvas da/dN frente a ΔK para espesores entre 2 mm y 14 mm para $R = 0{,}1$ y $R = 0{,}7$

Tanto los modelos RBFN como MLP entrenados con esta base de datos han mostrado un rendimiento claramente superior al obtenido en el caso del titanio, a pesar de haberse incorporado un rasgo adicional de entrada como es el espesor. Esta mejora general se atribuye al aumento del volumen de datos y a una mayor representatividad del dominio. No obstante, dado que el MLP ha demostrado un comportamiento significativamente más robusto y preciso frente a la RBFN, se decidió que la siguiente base de datos —de mayor tamaño y complejidad— fuese modelada exclusivamente con el MLP.

5.3. Aluminio 2024-T351 (Conjunto 2)

Tal como se justificó anteriormente, dado el elevado volumen y la alta variabilidad del conjunto de datos, se optó por entrenar únicamente un modelo MLP, descartando el uso de RBFN al haber demostrado un rendimiento insuficiente en escenarios previos. Las métricas de rendimiento obtenidas por el modelo se muestran en la siguiente tabla:

Modelo	**MSE (log)**	**R^2**	**Tiempo (s)**	**Memoria (kB)**
MLP	5.3786e-03	0.9930	0.94	124

Cuadro 5.3: Métricas de rendimiento obtenidas por el modelo MLP para el aluminio 2024-T351.

Las métricas obtenidas reflejan un rendimiento altamente satisfactorio del modelo MLP. El coeficiente de determinación $R^2 = 0{,}9930$ evidencia una excelente capacidad explicativa, lo que confirma que el modelo captura adecuadamente la variabilidad del fenómeno. Además, el tiempo de entrenamiento y el uso de memoria se mantienen dentro de márgenes razonables, lo que refuerza la viabilidad práctica del modelo incluso en entornos con recursos computacionales limitados.

A continuación se presentan las gráficas de predicción frente a datos reales para evaluar visualmente la capacidad del modelo. Dado que el conjunto de datos incluye un gran número de curvas individuales, se ha optado por mostrar únicamente una selección representativa de los casos más relevantes. Esta decisión responde al objetivo de no saturar el documento con figuras repetitivas, manteniendo así la claridad expositiva sin comprometer la solidez del análisis. El resto de curvas, no incluidas aquí por brevedad, presentan un comportamiento similar en cuanto a ajuste y tendencia general.

- **Curva de 3.05 mm (R = 0.1)** (Fig. 5.14)**:** Esta curva destaca por su morfología bien definida y la clara transición entre las tres regiones características. El modelo reproduce con precisión tanto el umbral como la fase lineal y la aceleración final, lo que la convierte en una buena muestra del ajuste general alcanzado. Puede apreciarse la no linealidad que aparece al final de la Zona I en las curvas con valores de R bajos

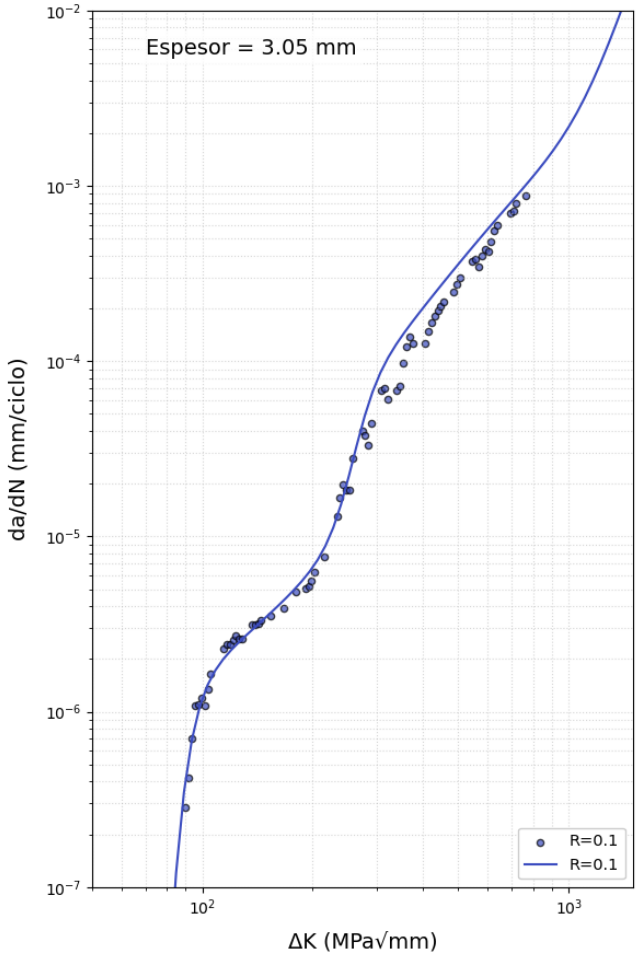

Figura 5.14: Predicción curvas

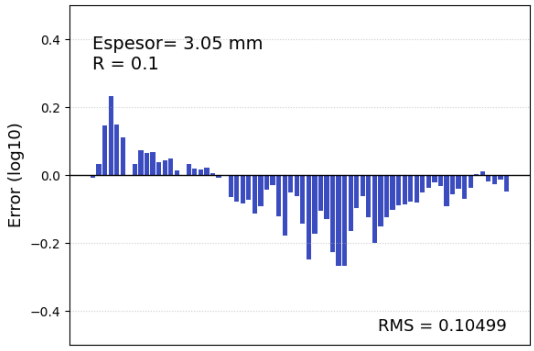

Figura 5.15: MSE entre el valor real y predicho.

- **Curva de 6.1 mm (R entre 0.01 y 0.8)** (Fig. 5.16)**:** Se observa como el modelo responde ante una amplia variación de R manteniendo constante el espesor. Las predicciones siguen correctamente la progresión esperada y no se cruzan entre sí, lo que refleja una buena comprensión de la relación entre R y la velocidad de propagación.

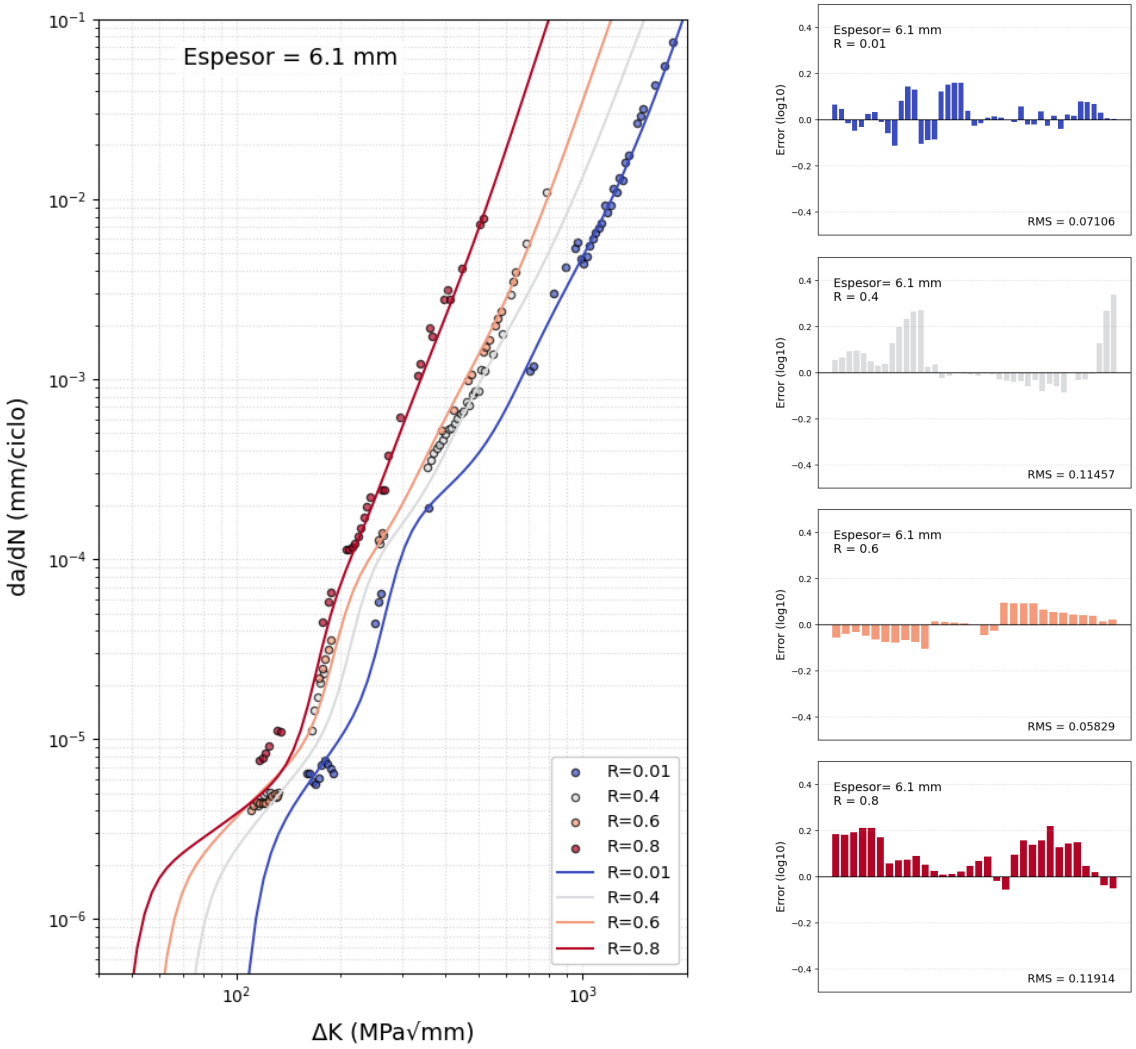

Figura 5.16: Curva predicha (izquierda) y errores por relación de carga (derecha).

- **Curva de 9.5 mm (R = 0.1 y 0.7)** (Fig. 5.17)**:** Estás curvas recorren un rango bastante amplio de ΔK. En este caso se aprecia claramente cómo el modelo diferencia entre dos valores de R muy separados. Las curvas mantienen su orden y no se solapan, lo cual refuerza la idea de que el modelo ha aprendido a respetar la evolución física del material con el cambio en la relación de carga.

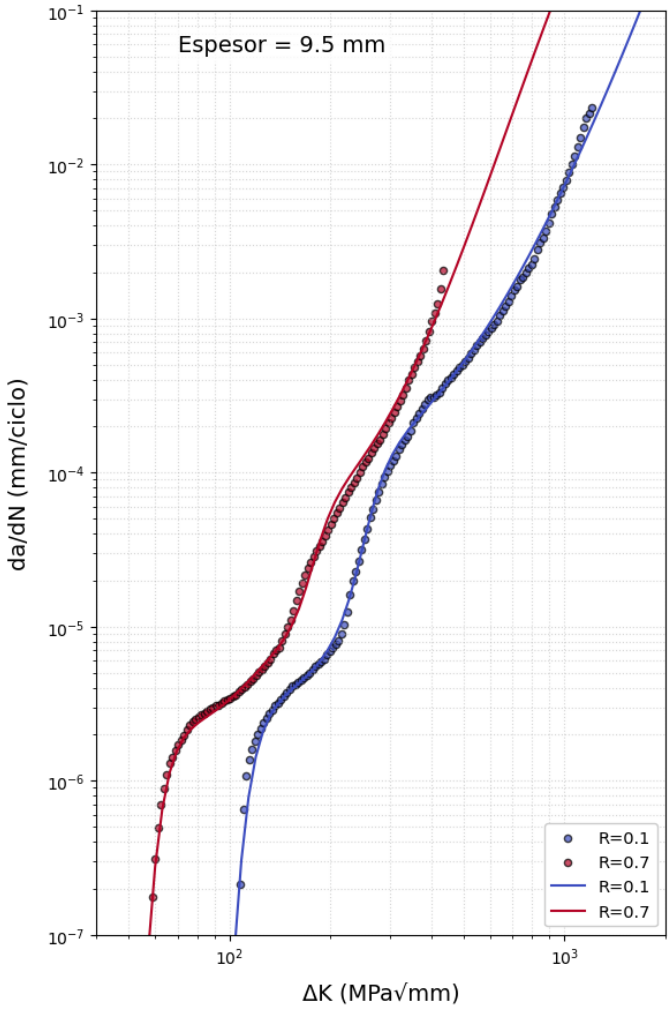

Figura 5.17: Curvas de predicción frente a datos experimentales.

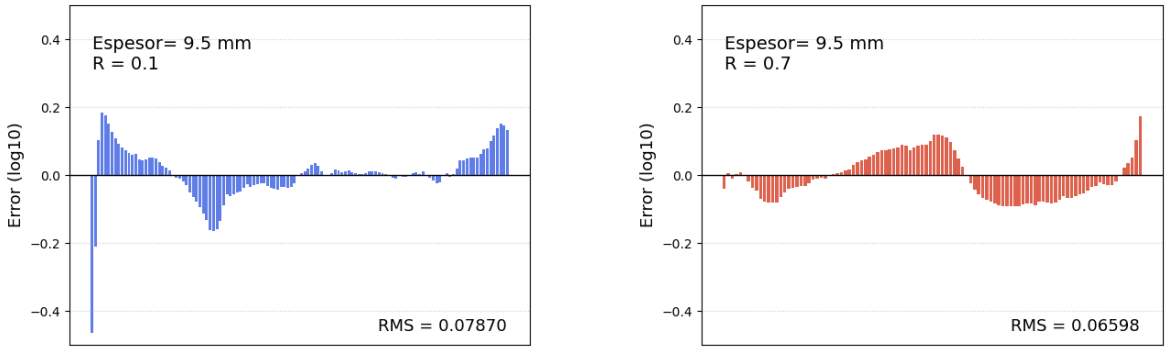

Figura 5.18: MSE entre el valor real y predicho.

- **Curva de 12.7 mm (R = 0.01 y 0.05)** (Fig. 5.19)**:** A pesar de que los valores de R son muy cercanos y pequeños, el modelo logra seguir la tendencia, lo que demuestra su capacidad de resolución en zonas más complejas.

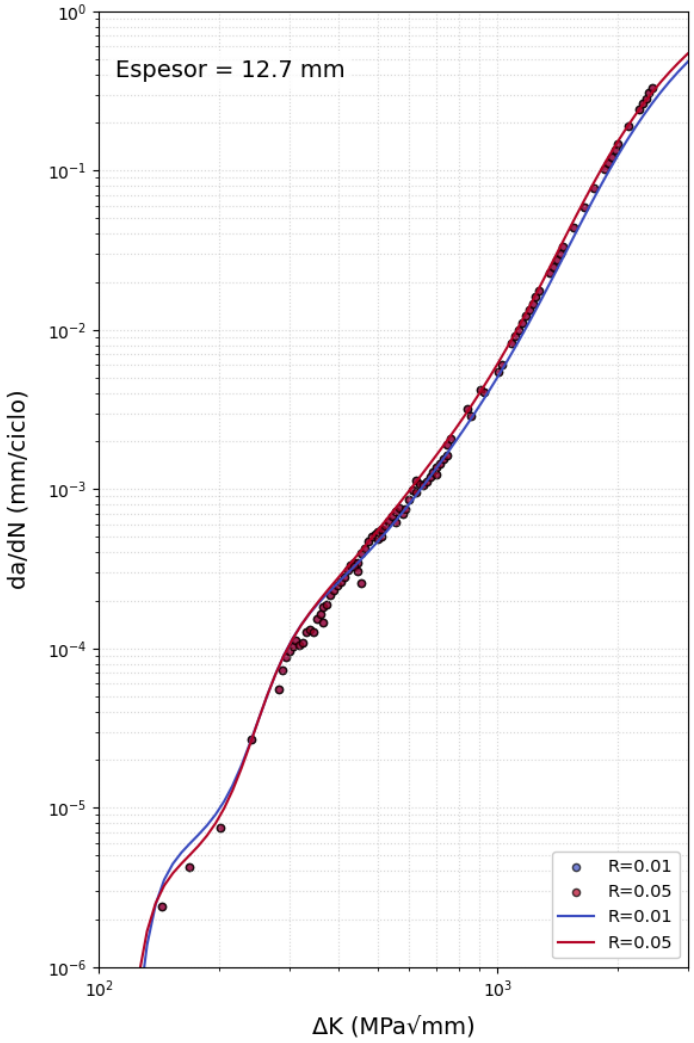

Figura 5.19: Curvas de predicción frente a datos experimentales.

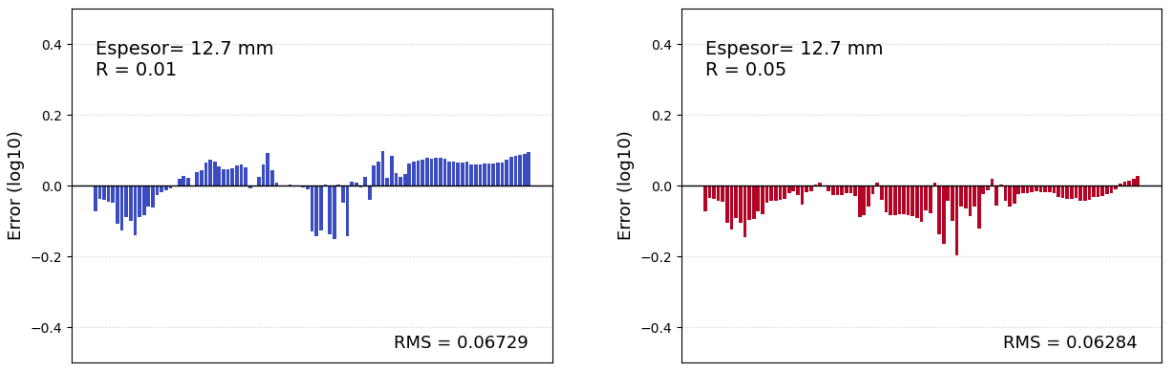

Figura 5.20: MSE entre el valor real y predicho.

Validación en curvas no entrenados

Cuando se aplica el modelo a espesores no incluidos en el entrenamiento (3,05 mm y 9,91 mm) (Figura 5.21), las predicciones siguen fielmente la tendencia experimental en todo el rango de ΔK, lo que indica que ha captado con solidez el comportamiento del crecimiento de grieta. Esto demuestra una buena capacidad de generalización, que debería confirmarse evaluando su rendimiento en otros espesores.

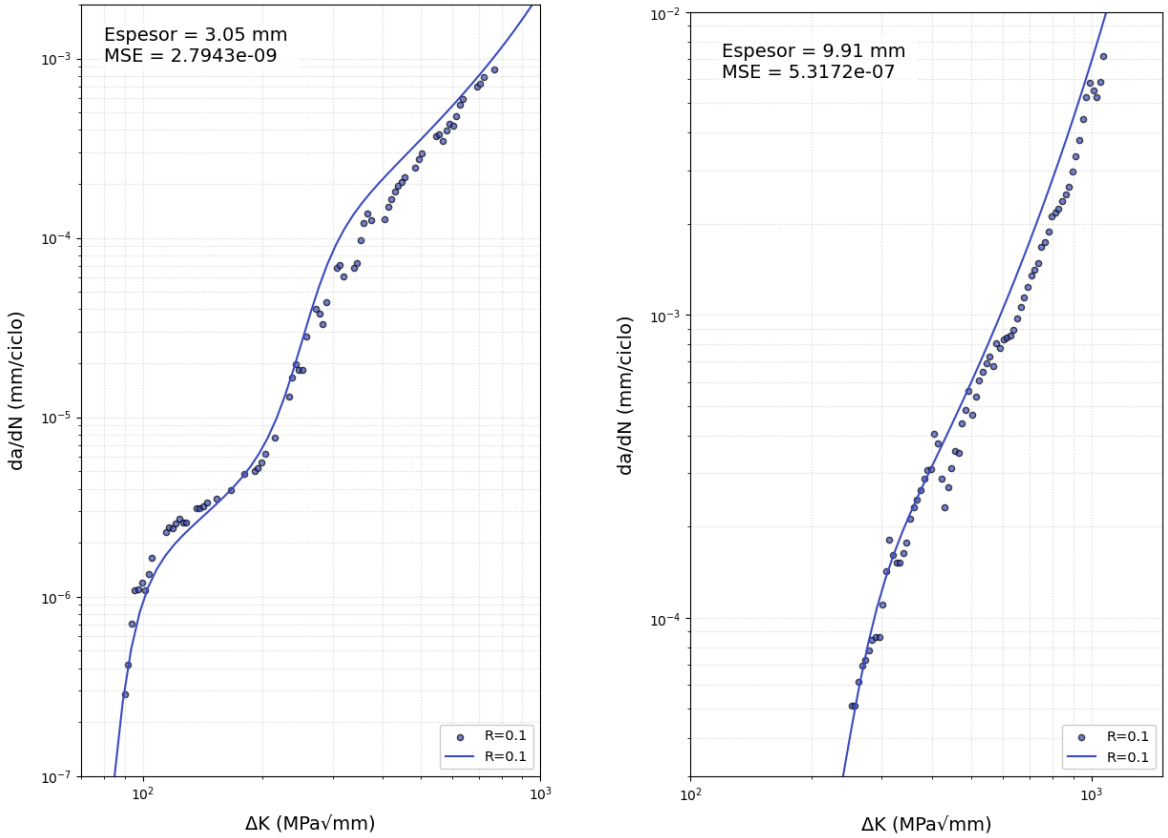

Figura 5.21: Predicciones para curvas de prueba

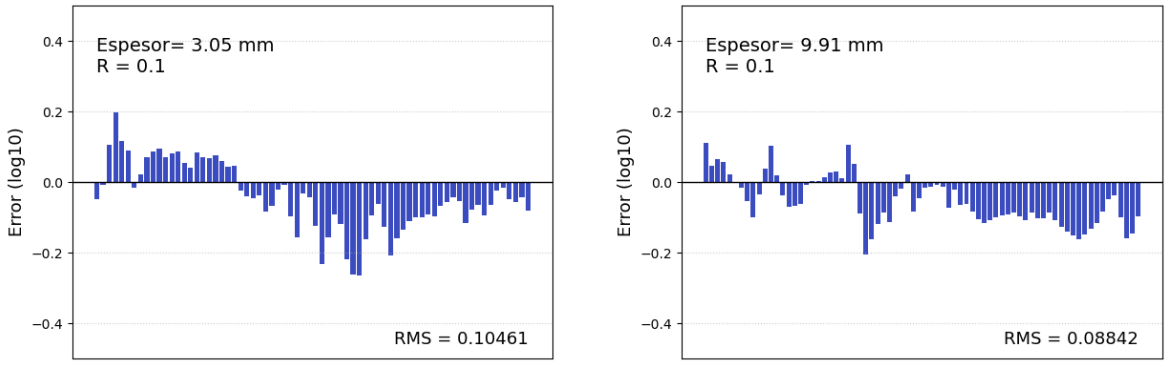

Figura 5.22: MSE entre el valor real y predicho.

Variación de la relación de carga

El modelo capta de forma clara el efecto de la relación de carga R sobre la curva da/dN-ΔK en dos espesores muy distintos. Tanto para chapa de 9,5 mm como de 2 mm, al aumentar R las predicciones se desplazan hacia la derecha y quedan ordenadas de menor a mayor R sin cruces. En la lámina de 2 mm el desplazamiento entre curvas es algo más pronunciado. Estos resultados confirman que el modelo reproduce la influencia de R de forma estable y coherente en ambos casos.

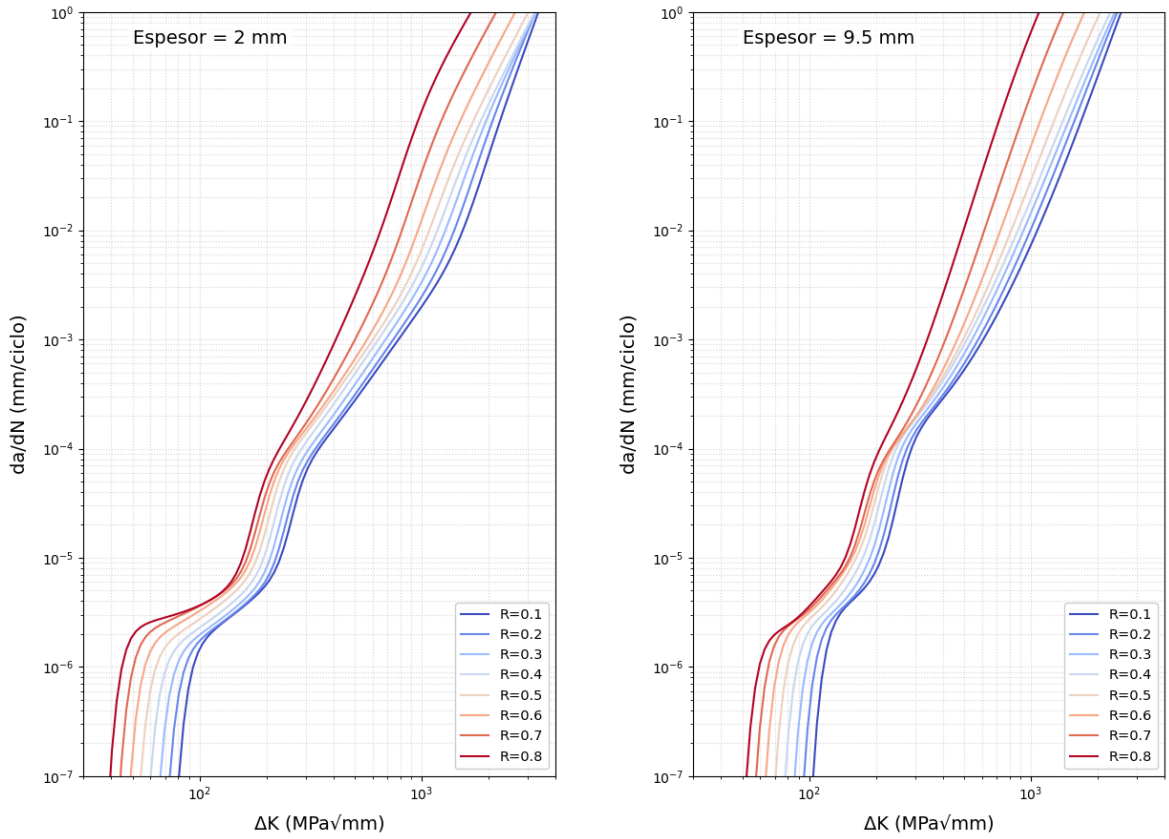

Figura 5.23: Curvas da/dN frente a ΔK para $R \in [0.1,0.7]$ en espesores de 2 mm y 9.5 mm.

Influencia del espesor

En general, el modelo reproduce el comportamiento esperado al incrementar el espesor a relación de carga constante: las curvas de crecimiento de grieta muestran una tendencia creciente con el espesor, especialmente en la parte alta del rango de ΔK. No obstante, en la zona inferior de las curvas se observa una inversión en algunos casos, donde curvas de menor espesor superan a otras más gruesas, lo que rompe la progresión esperada. En los valores bajos y medios de ΔK apenas hay diferencias entre curvas, pero a partir de un cierto umbral, la velocidad de propagación aumenta con el espesor de forma más clara.

El MLP está intentando acomodar simultáneamente efectos contradictorios de espesores extremos. Para corregirlo, será necesario enriquecer el conjunto de datos con especímenes de espesores intermedios, o explorar arquitecturas que capten mejor la dependencia continua de da/dN con el espesor.

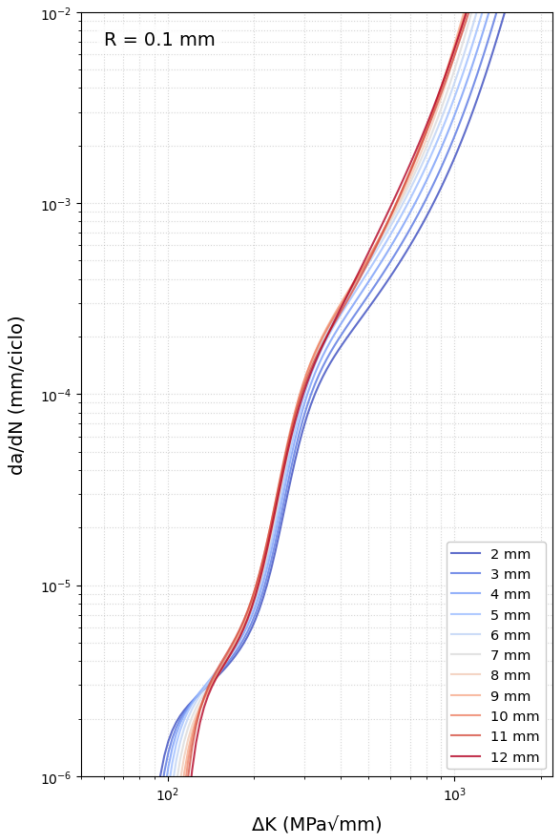

Figura 5.24: Variación del espesor con R constante.

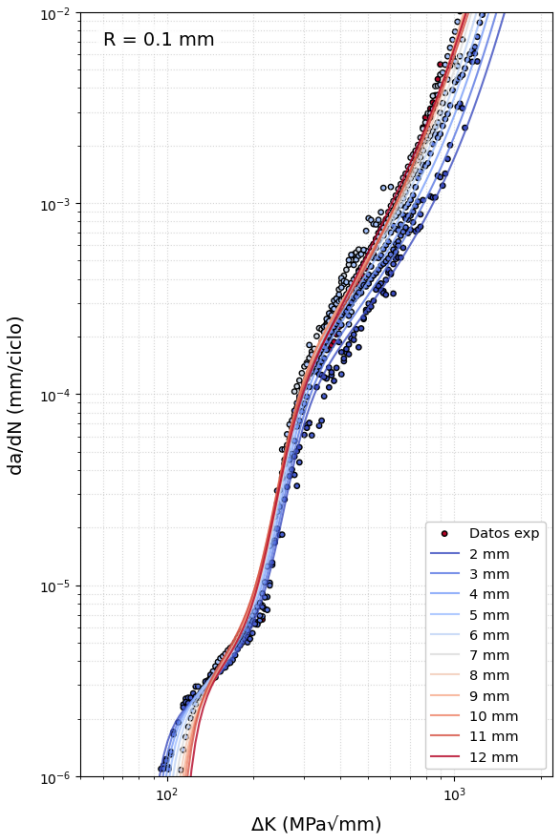

Figura 5.25: Variación del espesor con R constante con datos de entrenamiento.

6. Conclusiones y lineas futuras

6.1. Conclusiones

- A partir de los resultados obtenidos en este trabajo se concluye que tanto la Red de Base Radial (RBFN) como el Perceptrón Multicapa (MLP) son capaces de modelar correctamente la propagación de grietas por fatiga en función del factor de intensidad de tensiones (ΔK), la relación de carga (R) y el espesor del material. Sin embargo, existen diferencias importantes entre ambos modelos que condicionan su aplicabilidad práctica.

- La RBFN presenta ciertas limitaciones en cuanto a la capacidad de generalización y estabilidad en predicciones fuera del conjunto de entrenamiento. Concretamente, para valores intermedios de R y espesores no entrenados, muestra comportamientos inconsistentes.

- Por otro lado, el MLP ha demostrado ser considerablemente más robusto y generalizable. Las predicciones realizadas mediante esta red mantienen consistentemente el orden físico y la tendencia esperada, reproduciendo correctamente las tres regiones características y mostrando estabilidad incluso en situaciones no vistas durante el entrenamiento.

- En términos cuantitativos, el MLP alcanzó valores inferiores de error cuadrático medio (MSE) y mayores coeficientes de determinación (R^2) en todas las condiciones evaluadas, validando así su mayor capacidad predictiva frente a la RBFN.

- El modelo MLP se ha consolidado como una herramienta eficaz para predecir la velocidad de propagación de grietas en condiciones experimentales variadas. Su capacidad para adaptarse a escenarios con múltiples variables y rangos amplios lo convierte en una opción especialmente útil cuando se busca precisión en entornos complejos. Futuras investigaciones podrían enfocarse en ampliar el conjunto de datos en zonas críticas y en explorar arquitecturas alternativas que refuercen aún más su capacidad de generalización.

6.2. Lineas futuras

A partir de los resultados obtenidos, se identifican varias líneas de trabajo que podrían explorarse en futuras investigaciones. Una de ellas consiste en aumentar la densidad de datos experimentales, especialmente en rangos críticos de espesor y relación de carga R, con el fin de mejorar la continuidad y regularidad de las predicciones en zonas con alta variabilidad.

Otra línea prometedora sería la incorporación de arquitecturas más avanzadas basadas en técnicas de *deep learning*, como redes neuronales con múltiples entradas especializadas o modelos que integren atención sobre variables físicas. En esta misma dirección, se podría explorar el uso de *embeddings*, codificando variables discretas como el tipo de material, geometría de la probeta o incluso condiciones de ensayo, permitiendo así al modelo aprender representaciones internas más abstractas y transferibles.

También resulta interesante estudiar estrategias de *transfer learning*, entrenando un modelo general con múltiples materiales y ajustándolo luego a condiciones específicas mediante reentrenamiento parcial. Finalmente, la validación cruzada en materiales nuevos, con distinta morfología microestructural o mecanismos de fatiga más complejos, permitiría evaluar la capacidad real del modelo para generalizar fuera del dominio actual.

Bibliografía

[1] Dowling NE. Mechanical Behavior of Materials: Engineering Methods for Deformation, Fracture, and Fatigue. 4th ed. Pearson; 2012.

[2] Paris P, Erdogan F. A critical analysis of crack propagation laws. Journal of Basic Engineering. 1963;85(4):528-34.

[3] Rosenblatt F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review. 1958;65(6):386-408.

[4] Broomhead DS, Lowe D. Multivariable Functional Interpolation and Adaptive Networks. Royal Signals and Radar Establishment; 1988. RSRE-MEMO-4148.

[5] Bishop CM. Pattern Recognition and Machine Learning. New York: Springer; 2006.

[6] Park J, Sandberg IW. Universal Approximation Using Radial-Basis-Function Networks. Neural Computation. 1991;3(2):246-57.

[7] Python Software Foundation. Python language reference (version 3.x); n.d. Accessed: 2025-06-11. Available from: `https://www.python.org/`.

[8] Spyder IDE. Scientific Python Development Environment; n.d. Accessed: 2025-06-11. Available from: `https://www.spyder-ide.org/`.

[9] Anaconda Inc . Anaconda Distribution; n.d. Accessed: 2025-06-11. Available from: `https://www.anaconda.com/`.

[10] Microsoft Corporation. Microsoft Excel; n.d. Accessed: 2025-06-11. Available from: `https://www.microsoft.com/en-us/microsoft-365/excel`.

[11] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature. 2020;585(7825):357-62. Available from: `https://doi.org/10.1038/s41586-020-2649-2`.

[12] McKinney W. Data structures for statistical computing in Python. In: van der Walt S, Millman J, editors. Proceedings of the 9th Python in Science Conference; 2010. p. 56-61. Available from: `https://doi.org/10.25080/Majora-92bf1922-00a`.

[13] Foundation PS. pickle — Python object serialization; 2024. Python Standard Library Documentation. Available from: `https://docs.python.org/3/library/pickle.html`.

[14] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research. 2011;12:2825-30. Available from: `https://www.jmlr.org/papers/v12/pedregosa11a.html`.

[15] Hunter JD. Matplotlib: A 2D graphics environment. Computing in Science & Engineering. 2007;9(3):90-5. Available from: https://doi.org/10.1109/MCSE.2007.55.

[16] Wang H, Zhang W, Sun F, Zhang W. A Comparison Study of Machine Learning Based Algorithms for Fatigue Crack Growth Calculation. Materials. 2017;10(5):543. Available from: https://www.mdpi.com/1996-1944/10/5/543.

[17] NASGRO Fatigue Crack Growth Computer Program, version 6.0. Reference manual; 2010. NASA Johnson Space Center.

[18] Byrd RH, Lu P, Nocedal J, Zhu C. A Limited Memory Algorithm for Bound Constrained Optimization. SIAM Journal on Scientific Computing. 1995;16(5):1190-208. Available from: https://doi.org/10.1137/0916069.

[19] Goodfellow I, Bengio Y, Courville A. Deep Learning. MIT Press; 2016. Available from: https://www.deeplearningbook.org/.

Anexos

Índice general

A. Códigos Titanio Ti-6Al-4V

A.1 Comparación MSE Prueba Entrenamiento

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
from scipy.spatial.distance import cdist
from sklearn.metrics import mean_squared_error

# Cargar datos
df = pd.read_excel(r"BBDD/datos.xlsx")

# Aplicar log10 a las columnas 0 y 1
X = np.column_stack((np.log10(df.iloc[:, 1]).values, df.iloc[:, 2].values))  # Rasgos (columnas 1 y 2)
y = np.log10(df.iloc[:, 0]).values.reshape(-1, 1)  # Variable predicha (columna 0)

# Normalizaci n de datos
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y)

# Dividir en conjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_size=0.2, random_state=42)

# Definir rango de centros desde n_heuristico hasta 2 * n_heuristico
n_heuristico = int(np.sqrt(len(X_train)))
n_centers_range = range(2, 50)

# Listas para almacenar los errores
train_mses = []
test_mses = []

# Funci n radial Gaussiana
def gaussian_rbf(x, center, sigma):
    return np.exp(-cdist(x, center, 'sqeuclidean') / (2 * sigma**2))

# Iterar sobre el rango de centros
```

```
for n_centers in n_centers_range:
    # Aplicar K-Means para encontrar los centros
    kmeans = KMeans(n_clusters=n_centers, random_state=42,
        n_init=10)
    kmeans.fit(X_train)
    centers = kmeans.cluster_centers_

    # Calcular activaciones
    sigma = np.mean(cdist(centers, centers, 'euclidean'))  #
        Dispersi n promedio
    Phi_train = gaussian_rbf(X_train, centers, sigma)
    Phi_test = gaussian_rbf(X_test, centers, sigma)

    # Ajustar regresi n lineal sobre las activaciones
    model = LinearRegression()
    model.fit(Phi_train, y_train)

    # Predicciones
    y_train_pred = model.predict(Phi_train)
    y_test_pred = model.predict(Phi_test)

    # Calcular el error cuadr tico medio (MSE) para
        entrenamiento y prueba
    train_mse = mean_squared_error(y_train, y_train_pred)
    test_mse = mean_squared_error(y_test, y_test_pred)

    # Almacenar los errores
    train_mses.append(train_mse)
    test_mses.append(test_mse)

    print(f'N mero de centros: {n_centers}, MSE en
        entrenamiento: {train_mse}, MSE en prueba: {test_mse}')

# Graficar los errores
plt.figure(figsize=(10, 6))
plt.plot(n_centers_range, train_mses, marker='o', linestyle='-',
    color='b', label='Entrenamiento')
plt.plot(n_centers_range, test_mses, marker='o', linestyle='-',
   color='r', label='Prueba')
plt.xlabel('N mero de Centros')
plt.ylabel('Error Cuadr tico Medio (MSE)')
plt.title('Comparaci n de MSE entre Entrenamiento y Prueba para
   Diferentes N meros de Centros')
plt.legend()
plt.grid(True)
plt.show()
```

A.2 RBFN

```
import numpy as np
import pandas as pd
import pickle
import time
import psutil
import os
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
from scipy.spatial.distance import cdist

def get_memory_kb():
    """Obtiene el uso de memoria en kilobytes (kB)"""
    process = psutil.Process(os.getpid())
    return process.memory_info().rss / 1024  # Convertir a kB

# Cargar datos
df = pd.read_excel(r"BBDD/datos.xlsx")

# Aplicar log10 a las columnas 0 y 1
X = np.column_stack((np.log10(df.iloc[:, 1]).values, df.iloc[:,
   2].values))
y = np.log10(df.iloc[:, 0]).values.reshape(-1, 1)

# Medici n inicial de memoria y tiempo
mem_inicio = get_memory_kb()
tiempo_inicio = time.time()

# Normalizaci n de datos
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y)

# Dividir en conjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
   y_scaled, test_size=0.2, random_state=42)

# Configuraci n RBFN
n_centers = 18  # N mero de neuronas RBF

# Entrenamiento K-Means para centros RBF
kmeans = KMeans(n_clusters=n_centers, random_state=42, n_init
   =10)
kmeans.fit(X_train)
```

```
centers = kmeans.cluster_centers_

# Funci n radial Gaussiana
def gaussian_rbf(x, center, sigma):
    return np.exp(-cdist(x, center, 'sqeuclidean') / (2 * sigma
       **2))

# Calcular sigma (dispersi n)
sigma = np.mean(cdist(centers, centers, 'euclidean'))

# Calcular activaciones RBF
Phi_train = gaussian_rbf(X_train, centers, sigma)
Phi_test = gaussian_rbf(X_test, centers, sigma)

# Entrenamiento capa de salida
model = LinearRegression()
model.fit(Phi_train, y_train)

# Predicciones
y_pred = model.predict(Phi_test)
y_pred_original = scaler_y.inverse_transform(y_pred)
y_test_original = scaler_y.inverse_transform(y_test)

# C lculo de m tricas
mse = mean_squared_error(10**y_test_original, 10**
   y_pred_original)
mse_log = mean_squared_error(y_test_original, y_pred_original)
r2 = r2_score(10**y_test_original, 10**y_pred_original)

# Medici n final de recursos
tiempo_entrenamiento = time.time() - tiempo_inicio
mem_consumida = get_memory_kb() - mem_inicio

# Resultados
print("\n" + "="*50)
print("EVALUACI N DEL MODELO RBFN")
print("="*50)
print(f"- Neuronas RBFN: {n_centers}")
print(f"- MSE: {mse:.4e}")
print(f"- R  : {r2:.4f}")
print(f"- Tiempo entrenamiento: {tiempo_entrenamiento:.2f}
   segundos")
print(f"- Memoria utilizada: {mem_consumida:.2f} kB")
print("="*50 + "\n")

# Guardar modelo
modelo_exportar = {
    'scaler_X': scaler_X,
    'scaler_y': scaler_y,
```

```
    'centers': centers,
    'sigma': sigma,
    'model': model,
    'metricas': {
        'mse_lineal': mse,
        'r2_score': r2,
        'tiempo_entrenamiento': tiempo_entrenamiento,
        'memoria_kb': mem_consumida,
        'n_centers': n_centers
    }
}

with open('modelo_rbfn_exportado.pkl', 'wb') as f:
    pickle.dump(modelo_exportar, f)
```

A.3 MLP

```
import numpy as np
import pandas as pd
import pickle
import time
import psutil
import os
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score


def get_memory_kb():
    """Obtiene el uso de memoria en kilobytes (kB)"""
    process = psutil.Process(os.getpid())
    return process.memory_info().rss / 1024  # Convertir a kB

# Cargar datos
df = pd.read_excel(r"BBDD/datos.xlsx")

# Aplicar log10 a las columnas 0 y 1
X = np.column_stack((np.log10(df.iloc[:, 1]).values, df.iloc[:,
   2].values))
y = np.log10(df.iloc[:, 0]).values.reshape(-1, 1)

# Normalizaci n de datos
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y)

# Dividir en conjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
   y_scaled, test_size=0.2, random_state=42)

# Configuraci n del modelo
mlp = MLPRegressor(
    hidden_layer_sizes=(10, 4,),
    activation='tanh',
    solver='lbfgs',
    alpha=0.001,
    batch_size=32,
    learning_rate='adaptive',
    max_iter=1000,
    early_stopping=True,
    validation_fraction=0.15,
    random_state=42
```

```
)

# Medici n de memoria y tiempo inicial
mem_inicio = get_memory_kb()
tiempo_inicio = time.time()

# Entrenamiento del modelo
mlp.fit(X_train, y_train.ravel())

# Medici n de memoria y tiempo final
tiempo_entrenamiento = time.time() - tiempo_inicio
mem_final = get_memory_kb()
mem_consumida = mem_final - mem_inicio

# Predicciones
y_pred = mlp.predict(X_test).reshape(-1, 1)

# Desnormalizar
y_pred_original = scaler_y.inverse_transform(y_pred)
y_test_original = scaler_y.inverse_transform(y_test)

# Calcular m tricas en escala original (sin log)
mse = mean_squared_error(10**y_test_original, 10**
    y_pred_original)
r2 = r2_score(10**y_test_original, 10**y_pred_original)

# Calcular MSE en escala logar tmica (opcional)
mse_log = mean_squared_error(y_test_original, y_pred_original)

# Predicciones para todos los datos
y_pred_full = mlp.predict(X_scaled).reshape(-1, 1)
y_pred_full_original = scaler_y.inverse_transform(y_pred_full)

# Resultados
print("\n" + "="*50)
print("M TRICAS DE EVALUACI N DEL MODELO")
print("="*50)
print(f"- MSE: {mse:.4e}")
print(f"- R  : {r2:.4f}")
print(f"- Tiempo de entrenamiento: {tiempo_entrenamiento:.2f}
    segundos")
print(f"- Memoria utilizada: {mem_consumida:.2f} kB")
print("="*50 + "\n")

# Guardar modelo
modelo_exportar = {
    'scaler_X': scaler_X,
    'scaler_y': scaler_y,
    'mlp': mlp,
```

```
    'metricas': {
        'mse': mse,
        'r2': r2,
        'tiempo_entrenamiento': tiempo_entrenamiento,
        'memoria_utilizada': mem_consumida
    }
}

with open('modelo_mlp_exportado.pkl', 'wb') as f:
    pickle.dump(modelo_exportar, f)
```

A.4 Representación 2D

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
import pandas as pd
from matplotlib.cm import coolwarm
from matplotlib.colors import Normalize
from scipy.spatial.distance import cdist

def gaussian_rbf(x, center, sigma):
    """Funci n de base radial gaussiana"""
    return np.exp(-cdist(x, center, 'sqeuclidean') / (2 * sigma
        **2))

# Cargar modelo RBFN guardado
with open('modelo_rbfn_exportado.pkl', 'rb') as f:
    modelo = pickle.load(f)

df = pd.read_excel(r"BBDD/datos.xlsx")

# Extraer componentes del modelo RBFN
scaler_X = modelo['scaler_X']
scaler_y = modelo['scaler_y']
centers = modelo['centers']
sigma = modelo['sigma']
rbf_model = modelo['model']


R_values = [0.1, 0.33, 0.5, 0.75]
#R_values = [0.1, 0.2, 0.3, 0.4, 0.5, 0.5, 0.7, 0.8]

norm = Normalize(vmin=min(R_values), vmax=max(R_values))

# Crear figura
plt.figure(figsize=(6, 9))

# Generar valores de  K  para predicci n (log-distribuidos)
dK_range = np.logspace(0, 2, 300)  # De 10^0 a 10^2 (1 a 100)

# Graficar datos y predicciones para cada valor de R
for R_val in R_values:
    # Obtener color del colormap
    color = coolwarm(norm(R_val))

    # Filtrar datos experimentales
    mask = df.iloc[:, 2] == R_val
    subdf = df[mask]

```

```
    # Graficar datos experimentales
    plt.scatter(subdf.iloc[:, 1], subdf.iloc[:, 0],
                color=color, label=f'R = {R_val}', s=5)

    # Preparar datos para predicci n RBFN
    X_pred = np.column_stack((np.log10(dK_range), np.full_like(
        dK_range, R_val)))
    X_pred_scaled = scaler_X.transform(X_pred)

    # Calcular activaciones RBF
    Phi_pred = gaussian_rbf(X_pred_scaled, centers, sigma)

    # Predecir y transformar
    y_pred = scaler_y.inverse_transform(rbf_model.predict(
        Phi_pred).reshape(-1, 1))
    da_dN_pred = 10**y_pred.flatten()

    # Graficar curva predicha por RBFN
    plt.plot(dK_range, da_dN_pred, color=color, linewidth=1.5)

# Configuraci n del gr fico
plt.title(f'Predicciones RBFN ({len(centers)} neuronas)',
    fontsize=14)
plt.xlim(1, 100)
plt.ylim(1.01e-7, 1)
plt.xlabel(r'$\Delta K \, (MPa\sqrt{m})$', fontsize=12)
plt.ylabel(r'$da/dN \, (mm/ciclo)$', fontsize=12)
plt.xscale('log')
plt.yscale('log')
plt.grid(axis='x', which='both', alpha=0.4)
plt.legend(title='Valores de R', loc='lower right', fontsize=10)
plt.tight_layout()
plt.show()
```

A.5 Representación 3D

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
import pandas as pd
from scipy.spatial.distance import cdist

def gaussian_rbf(x, center, sigma):
    """Funci n de base radial gaussiana"""
    return np.exp(-cdist(x, center, 'sqeuclidean') / (2 * sigma
       **2))

# Cargar modelo RBFN
with open('modelo_rbfn_exportado.pkl', 'rb') as f:
    modelo = pickle.load(f)
df = pd.read_excel(r"BBDD/datos.xlsx")

# Extraer componentes RBFN
scaler_X = modelo['scaler_X']
scaler_y = modelo['scaler_y']
centers = modelo['centers']
sigma = modelo['sigma']
rbf_model = modelo['model']

# Configuraci n de valores
dK_values = np.linspace(100, 1.8, 300)  # Reducido puntos para
   mejor rendimiento
R_values = np.linspace(0, 1, 300)
R_specific = [0.1, 0.33, 0.5, 0.75]
colors = ['#1f77b4', '#1f77b4', '#1f77b4', '#1f77b4']  # Azul
   uniforme
z_min, z_max = -9, 0  # L mites para log10(da/dN)

# Crear figura 3D
fig = plt.figure(figsize=(10.5, 7.5))
ax = fig.add_subplot(111, projection='3d')

# 1. Generar malla y aplicar m scara global
dK_mesh, R_mesh = np.meshgrid(dK_values, R_values)
X_pred = np.column_stack((np.log10(dK_mesh.ravel()), R_mesh.
   ravel()))
X_pred_scaled = scaler_X.transform(X_pred)

# Calcular activaciones RBF y predicciones
Phi_pred = gaussian_rbf(X_pred_scaled, centers, sigma)
y_pred = scaler_y.inverse_transform(rbf_model.predict(Phi_pred).
   reshape(-1, 1))
log_da_dN = np.log10(10**y_pred.reshape(dK_mesh.shape))
```

```
43
44 # M scara global
45 valid_mask = (log_da_dN >= z_min) & (log_da_dN <= z_max)
46 log_dK_masked = np.ma.masked_where(~valid_mask, np.log10(dK_mesh
       ))
47 R_mesh_masked = np.ma.masked_where(~valid_mask, R_mesh)
48 log_da_dN_masked = np.ma.masked_where(~valid_mask, log_da_dN)
49
50 # Superficie 3D con activaciones RBF
51 surf = ax.plot_surface(
52     log_dK_masked, R_mesh_masked, log_da_dN_masked,
53     cmap='coolwarm', alpha=0.5, edgecolor='none', antialiased=
          True)
54
55 # 2. Curvas espec ficas con RBFN (l neas continuas)
56 for R_val, color in zip(R_specific, colors):
57     # Preparar datos para curva espec fica
58     X_curve = np.column_stack((np.log10(dK_values), np.full_like
          (dK_values, R_val)))
59     X_curve_scaled = scaler_X.transform(X_curve)
60
61     # Calcular activaciones RBF
62     Phi_curve = gaussian_rbf(X_curve_scaled, centers, sigma)
63     y_curve = scaler_y.inverse_transform(rbf_model.predict(
          Phi_curve).reshape(-1, 1))
64     log_da_dN_curve = np.log10(10**y_curve).flatten()
65
66     # Aplicar m scara
67     curve_mask = (log_da_dN_curve >= z_min) & (log_da_dN_curve
          <= z_max)
68
69     # Graficar curva (l nea continua)
70     ax.plot(
71         np.ma.masked_where(~curve_mask, np.log10(dK_values)),
72         np.ma.masked_where(~curve_mask, np.full_like(dK_values,
              R_val)),
73         np.ma.masked_where(~curve_mask, log_da_dN_curve),
74         color=color, linewidth=3, linestyle='-',  # L nea
              continua
75         label=f'R␣=␣{R_val}'
76     )
77
78 # 3. Puntos experimentales
79 exp_mask = (np.log10(df.iloc[:, 0]) >= z_min) & (np.log10(df.
       iloc[:, 0]) <= z_max)
80 ax.scatter(
81     np.ma.masked_where(~exp_mask, np.log10(df.iloc[:, 1])),
82     np.ma.masked_where(~exp_mask, df.iloc[:, 2]),
83     np.ma.masked_where(~exp_mask, np.log10(df.iloc[:, 0])),
```

```
    color='black', s=40, edgecolor='white',
    label='Datos experimentales'
)

# Configuraci n de ejes
ax.invert_xaxis()
ax.set_zlim(z_min, z_max)
ax.set_xlabel(r'$\log_{10}(\Delta K) \, [MPa\sqrt{m}]$',
   fontsize=12, labelpad=15)
ax.set_ylabel('R ratio', fontsize=12, labelpad=15)
ax.set_zlabel(r'$\log_{10}(da/dN) \, [mm/ciclo]$', fontsize=12,
   labelpad=15)
ax.set_title(f'Superficie RBFN ({len(centers)} neuronas)',
   fontsize=14)
ax.legend(fontsize=10, loc='upper left')
ax.view_init(elev=25, azim=-45)  # Mejor   ngulo   de
   visualizaci n

plt.tight_layout()
plt.show()
```

B. Códigos Aluminio 2024-T351

B.1 Correlación entre rasgos

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


df = pd.read_excel(r"BBDD/T351_filtrado.xlsx")

plt.figure(figsize=(5, 5),dpi=300)

plt.scatter(df.iloc[:, 2],
            df.iloc[:, 3],
            s=60, alpha=0.7, edgecolor='k'
            )

plt.xlabel('R', fontsize=14)
plt.ylabel('Espesor(mm)', fontsize=14)
#plt.title(f'Comportamiento para R={R_valor} con diferentes
   espesores', fontsize=16)

plt.grid(True, which="both", linestyle=':', alpha=0.8)

plt.tight_layout()
plt.show()

plt.figure(figsize=(5, 5), dpi=300)

plt.scatter(df.iloc[:, 2],
            np.log10(df.iloc[:, 0]),
            s=60, alpha=0.7, edgecolor='k'
            )

plt.xlabel('R', fontsize=14)
plt.ylabel(r'$da/dN␣\,␣(mm/ciclo)$', fontsize=14)
#plt.title(f'Comportamiento para R={R_valor} con diferentes
   espesores', fontsize=16)

plt.grid(True, which="both", linestyle=':', alpha=0.8)
plt.legend(fontsize=10, framealpha=1)

plt.tight_layout()
plt.show()


```

```
43
44 # 2. Aplicar log10 a columnas 0 y 1 (y manejar ceros/negativos
      si es necesario)
45 df.iloc[:, 0] = np.log10(df.iloc[:, 0].replace(0, np.nan))  #
      Evitar log(0)     NaN
46 df.iloc[:, 1] = np.log10(df.iloc[:, 1].replace(0, np.nan))  # Si
       hay negativos, usar log1p o transformaci n alternativa
47
48 # 3. Seleccionar columnas 0, 1, 2, 3 (ajusta los  ndices  si es
      necesario)
49 subset = df.iloc[:, :4]  # o df[[0, 1, 2, 3]]
50
51 # 4. Matriz de correlaci n (Pearson)
52 corr_matrix = subset.corr()
53
54 # Heatmap
55 plt.figure(figsize=(5,4), dpi = 300)
56 sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", vmin=-1,
      vmax=1, fmt=".2f")
57 plt.show()
58
59 # Crear el pairplot y guardarlo en una variable
60 g = sns.pairplot(
61     subset,
62     kind="reg",
63     height=3,
64     aspect=1,
65     plot_kws={"line_kws": {"color": "red"}}
66 )
67
68 # T tulo general
69
70
71 # Aumentar tama o de etiquetas de los ejes
72 for ax in g.axes.flatten():
73     if ax is not None:
74         ax.set_xlabel(ax.get_xlabel(), fontsize=25)
75         ax.set_ylabel(ax.get_ylabel(), fontsize=25)
76         ax.tick_params(labelsize=12)
77
78
79 plt.show()
```

B.2 RANSAC

```
from sklearn.linear_model import RANSACRegressor,
    LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import re

# Carga de datos
df_total = pd.read_excel("BBDD/datos_2.xlsx")
archivo_validados = "BBDD/inliers_validados.xlsx"

# Cargar validados si existe
if os.path.exists(archivo_validados):
    df_validados = pd.read_excel(archivo_validados)
else:
    df_validados = pd.DataFrame(columns=df_total.columns)

# Obtener referencias  nicas  en pendientes
referencias_pendientes = df_total.iloc[:,12].unique()

# Bucle por referencias pendientes
for referencia in referencias_pendientes:
    df_ref = df_total[df_total.iloc[:,12] == referencia].copy()
    if len(df_ref) < 10:
        continue

    X = np.log10(df_ref.iloc[:,1].values).reshape(-1,1)
    y = np.log10(df_ref.iloc[:,0].values)

    # parametros
    degree = 6
    residual_threshold = 0.1
    min_samples = 16

    poly_model = make_pipeline(PolynomialFeatures(degree=degree)
        , LinearRegression())
    ransac = RANSACRegressor(
        estimator=poly_model,
        residual_threshold=residual_threshold,
        min_samples=min_samples,
        random_state=0
    )

    ransac.fit(X, y)
```

```
    inliers_mask = ransac.inlier_mask_
    outliers_mask = ~inliers_mask

    # Mostrar la gr fica
    plt.figure(figsize=(6, 8))
    plt.scatter(X[inliers_mask], y[inliers_mask], color='green',
        label='Inliers')
    plt.scatter(X[outliers_mask], y[outliers_mask], color='red',
        label='Outliers')

    ransac.fit(X[inliers_mask], y[inliers_mask])
    X_fit = np.linspace(min(X)*0.98, max(X)*1.02, 100).reshape
       (-1,1)
    y_fit = ransac.predict(X_fit)

    plt.plot(X_fit, y_fit, color='blue', linewidth=2, label='
       RANSAC␣Polynomial␣Fit')
    plt.title(f"VALIDACION\nReferencia:␣{referencia}")
    plt.legend()
    plt.grid(True)
    plt.show()

    # Preguntar si aceptas esta curva
    print("Opciones:␣s␣=␣v lida␣(solo␣inliers),␣n␣=␣mantener␣en
       ␣pendientes,␣t␣=␣v lida␣completa␣(curva␣completa)")
    decision = input("  Guardar ␣como?␣(s/n/t):␣").strip().lower
       ()

    # L gica de guardado
    if decision == "s":
        # Guardar solo inliers en validados
        df_inliers_ref = df_ref[inliers_mask]
        df_validados = pd.concat([df_validados, df_inliers_ref],
            ignore_index=True)

        # Eliminar curva completa de pendientes
        df_total = df_total[df_total.iloc[:,12] != referencia]

        # Guardar la gr fica
        referencia_safe = re.sub(r'[^A-Za-z0-9_\-]', '_', str(
           referencia))
        plt.savefig(f"BBDD/grafica_ref_{referencia_safe}.png",
           dpi=300)

    elif decision == "n":
        # No se hace nada     la curva queda en pendientes
        pass

    elif decision == "t":
```

```
86          # Guardar curva completa en validados
87          df_validados = pd.concat([df_validados, df_ref],
                ignore_index=True)
88
89          # Eliminar curva completa de pendientes
90          df_total = df_total[df_total.iloc[:,12] != referencia]
91
92          # Guardar la gr fica
93          referencia_safe = re.sub(r'[^A-Za-z0-9_\-]', '_', str(
                referencia))
94          plt.savefig(f"BBDD/grafica_ref_{referencia_safe}.png",
                dpi=300)
95
96      # Cerrar figura
97      plt.close()
98
99  # Guardar resultados
100 df_validados.to_excel(archivo_validados, index=False)
101 df_total.to_excel("BBDD/curvas_pendientes.xlsx", index=False)
```

B.3 MLP con Validación Cruzada

```
import numpy as np
import pandas as pd
import pickle
import time
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split, KFold,
    cross_validate
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.pipeline import Pipeline


def main():

    df_high = pd.read_excel("BBDD/datos_1.xlsx")     #Datos
        Ponderados
    df_low = pd.read_excel("BBDD/datos_2.xlsx")


    X_high = np.column_stack((
        np.log10(df_high.iloc[:, 1]).values,  # Feature 1 (log-
            transformado)
        df_high.iloc[:, 2].values,            # Feature 2
        df_high.iloc[:, 3].values             # Feature 3
    ))
    y_high = np.log10(df_high.iloc[:, 0]).values.reshape(-1, 1)
        # Target (log-transformado)

    X_low = np.column_stack((
        np.log10(df_low.iloc[:, 1]).values,   # Feature 1 (log-
            transformado)
        df_low.iloc[:, 2].values,             # Feature 2
        df_low.iloc[:, 3].values              # Feature 3
    ))
    y_low = np.log10(df_low.iloc[:, 0]).values.reshape(-1, 1)
          # Target (log-transformado)

    ponderado = 4

    X_high_tripled = np.repeat(X_high, ponderado, axis=0)
    y_high_tripled = np.repeat(y_high, ponderado, axis=0)

    # 4. Combinar los conjuntos de datos
    X = np.vstack((X_high_tripled, X_low))
    y = np.vstack((y_high_tripled, y_low)).ravel()

    # 2. Pipeline con escalado y modelo
```

```
    model = Pipeline([
        ('scaler', StandardScaler()),
        ('mlp', MLPRegressor(
            hidden_layer_sizes=(10,),
            activation='tanh',
            solver='lbfgs',
            alpha=0.001,
            max_iter=5000,
            random_state=42
        ))
    ])

    # 3. Validaci n cruzada (5-Fold)
    print("\n" + "="*50)
    print("VALIDACI N CRUZADA (5-Fold)")
    print("="*50)

    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    cv_results = cross_validate(
        model, X, y,
        cv=kf,
        scoring=['neg_mean_squared_error', 'r2'],
        return_train_score=True,
        n_jobs=-1
    )

    # Resultados de CV
    print(f"MSE promedio (validaci n): {-cv_results['
       test_neg_mean_squared_error'].mean():.4e}")
    print(f"R  promedio (validaci n): {cv_results['test_r2'].
       mean():.4f}")
    print(f"Tiempo promedio por fold: {cv_results['fit_time'].
       mean():.2f} segundos")

    # 4. Entrenamiento final y evaluaci n con test set
    print("\n" + "="*50)
    print("EVALUACI N FINAL CON TEST SET")
    print("="*50)

    X_train, X_test, y_train, y_test = train_test_split(X, y,
       test_size=0.2, random_state=42)

    start_time = time.time()
    model.fit(X_train, y_train)
    training_time = time.time() - start_time

    # Predicciones y m tricas
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
```

```
87      r2 = r2_score(y_test, y_pred)
88
89      print(f"- MSE (escala original): {mse:.4e}")
90      print(f"- R  (escala original): {r2:.4f}")
91      print(f"- Tiempo de entrenamiento: {training_time:.2f}
            segundos")
92
93      # 5. Guardar modelo
94      modelo_exportar = {
95          'model': model,
96          }
97
98      with open('modelo_mlp_optimizado.pkl', 'wb') as f:
99          pickle.dump(modelo_exportar, f)
100
101 if __name__ == "__main__":
102     main()
```

B.4 Representación 2D: Curvas para distintos factores de carga y espesor constante

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
import pandas as pd
from matplotlib.colors import Normalize
from matplotlib.cm import coolwarm

# Configuraci n
espesor_constante = 2# mm
R_vals = [0.1, 0.3, 0.5, 0.7]

R_vals = [0.1]
R_vals = [0.1, 0.2, 0.3,0.4,0.5,0.6,0.7]
#R_vals = [0.01, 0.05]
norm = Normalize(vmin=min(R_vals), vmax=max(R_vals))

# 1. Carga de datos y modelo
df = pd.read_excel(r"BBDD/T351_filtrado.xlsx")
df = df[df.iloc[:, 3] == espesor_constante]  # Filtro por
    espesor

with open('modelo_mlp_optimizado.pkl', 'rb') as f:
    modelo = pickle.load(f)

model = modelo['model']


# 3. Generaci n de predicciones (manejo correcto de logs)
deltaK_range = np.logspace(np.log10(40), np.log10(3000), 100)  #
    Escala logar tmica
results = {}

for r_val in R_vals:



    # Preparar datos de entrada (atenci n a las
       transformaciones)
    X_pred = np.column_stack((
        np.log10(deltaK_range),  # Aplicamos log10 a  K  (como
           en el entrenamiento)
        np.full_like(deltaK_range, r_val),
        np.full_like(deltaK_range, espesor_constante)
    ))

```

```
    # Normalizaci n y predicci n

    y_pred = modelo['model'].predict(X_pred)


    # Almacenar resultados en escala natural (sin log)
    results[r_val] = (deltaK_range, 10**y_pred)  # Convertimos
       de log10 a escala natural

# 4. Visualizaci n corregida
plt.figure(figsize=(6, 9))

# Graficar datos experimentales (en escala natural)
"""
for r_val in R_vals:
    color = coolwarm(norm(r_val))
    mask = (np.abs(df.iloc[:, 2] - r_val) < 0.05)  # Tolerancia
       para valores de R
    if any(mask):
        plt.scatter(df.loc[mask, df.columns[1]],  #  K  en
           escala natural
                df.loc[mask, df.columns[0]],   # da/dN en
                   escala natural
                color=color, s=20, alpha=0.7, edgecolor='k',
                label=f'Datos R={r_val}')
"""
# Graficar predicciones (en escala natural)
for r_val in R_vals:
    color = coolwarm(norm(r_val))
    deltaK_pred, da_dN_pred = results[r_val]
    plt.plot(deltaK_pred, da_dN_pred,
             color=color, linewidth=1.5, linestyle='-',
             label=f'Predicci n R={r_val}')

# Configuraci n de ejes (log-log)
plt.xscale('log')
plt.yscale('log')
plt.xlabel(' K ( M P a mm )', fontsize=14, labelpad=10)
plt.ylabel('da/dN (mm/ciclo)', fontsize=14, labelpad=10)
plt.title(f'Curvas da/dN vs K \nEspesor ={espesor_constante}
   mm',
          fontsize=16, pad=20)

# Mejores ajustes de visualizaci n
plt.grid(True, which="both", linestyle=':', alpha=0.5)
plt.legend(loc = 'lower right',fontsize=10.5, framealpha=1)

plt.xlim(40,3000)  # L mites coherentes con los datos
plt.ylim(1e-7, 1)
```

```
86
87
88 plt.tight_layout()
89 plt.show()
```

B.5 Representación 2D: Curvas para distintos espesores y factores de carga constante

```python
import numpy as np
import matplotlib.pyplot as plt
import pickle
import pandas as pd
from matplotlib.colors import Normalize
from matplotlib.cm import coolwarm

# Configuraci n
R_valor = 0.1# Valor fijo de R
espesores = np.arange(2, 13)  # Espesores de 1 a 12 mm

norm = Normalize(vmin=min(espesores), vmax=max(espesores))

# 1. Carga de datos y modelo
df = pd.read_excel(r"BBDD/T351_filtrado.xlsx")
df_r01 = df[np.abs(df.iloc[:, 2] - R_valor) < 0.01]  # Filtrar
    por R=0.1

with open('modelo_mlp_optimizado.pkl', 'rb') as f:
    modelo = pickle.load(f)

# 2. Extraer componentes del modelo
model = modelo['model']


# Funci n RBF vectorizada
def gaussian_rbf(x, centers, sigma):
    return np.exp(-np.sum((x[:, np.newaxis] - centers)**2, axis
        =2) / (2 * sigma**2))

# 3. Generaci n de predicciones
deltaK_range = np.logspace(np.log10(50), np.log10(2000), 120)  #
    Rango  K
results = {}

for espesor in espesores:
    # Preparar datos de entrada
    X_pred = np.column_stack((
        np.log10(deltaK_range),
        np.full_like(deltaK_range, R_valor),
        np.full_like(deltaK_range, espesor)
    ))

    # Normalizaci n y predicci n
    y_pred = modelo['model'].predict(X_pred)
```

```
43
44      results[espesor] = (deltaK_range, 10**y_pred)  # Guardar
            predicciones
45
46  # 4. Visualizaci n
47  plt.figure(figsize=(6, 9))
48
49  # Graficar datos experimentales (todos los espesores para R=0.1)
50  plt.scatter(df_r01.iloc[:, 1],  #  K
51              df_r01.iloc[:, 0],  # da/dN
52              c=df_r01.iloc[:, 3],  # Color por espesor
53              cmap='coolwarm', s=15, alpha=1, edgecolor='k',
54              label='Datos␣experimentales')
55
56  # Graficar predicciones para cada espesor
57  for espesor in espesores:
58      color = coolwarm(norm(espesor))
59      deltaK_pred, da_dN_pred = results[espesor]
60      plt.plot(deltaK_pred, da_dN_pred,
61               color=color, linewidth=1.5, alpha=0.8,
62               label=f'{espesor}␣mm')
63
64  # Configuraci n del gr fico
65  plt.xscale('log')
66  plt.yscale('log')
67  plt.xlabel(' K ␣( M P a mm )', fontsize=14)
68  plt.ylabel('da/dN␣(mm/ciclo)', fontsize=14)
69  plt.title(f'Comportamiento␣para␣R={R_valor}␣con␣diferentes␣
        espesores', fontsize=16, pad=20)
70
71  plt.grid(True, which="both", linestyle=':', alpha=0.5)
72  plt.legend(loc = 'lower␣right',fontsize=10, framealpha=1)
73  plt.xlim(50, 2000)  # L mites coherentes con los datos
74  plt.ylim(1e-6, 1e-2)
75
76  plt.tight_layout()
77  plt.show()
```

B.6 Representación 3D

```python
import numpy as np
import matplotlib.pyplot as plt
import pickle
import pandas as pd

# Cargar el modelo guardado
with open('modelo_mlp_optimizado.pkl', 'rb') as f:
    modelo = pickle.load(f)

df = pd.read_excel(r"BBDD/T351_filtrado.xlsx")

# Extraer componentes del modelo
model = modelo['model']


# Generar malla de valores para predicci n
deltak_range = np.linspace(50, 1000, 200)  # Primer rasgo
R_values = np.linspace(0, 1, 200)          # Segundo rasgo
espesor_constante = 9.91
df = df[df.iloc[:, 3] == espesor_constante]

# Crear malla 2D (INTERCAMBIO: primero R, luego deltak para intercambiar ejes)
R_mesh, Deltak_mesh = np.meshgrid(R_values, deltak_range)

# Preparar datos para predicci n (a adiendo el espesor constante)
X_pred = np.column_stack((
    np.log10(Deltak_mesh.flatten()),  # Aplicamos log10 al primer rasgo
    R_mesh.flatten(),
    np.full_like(Deltak_mesh.flatten(), espesor_constante)
))


# Hacer predicciones
y_pred = modelo['model'].predict(X_pred)


#y_pred = 10**y_pred  # Convertir de log10 a escala original

# Dar forma a las predicciones para la malla
Z = y_pred.reshape(Deltak_mesh.shape)

# Crear figura 3D
fig = plt.figure(figsize=(14, 10))
ax = fig.add_subplot(111, projection='3d')
```

```
45
46 # Graficar superficie principal
47 surf = ax.plot_surface(R_mesh,
48                         np.log10(Deltak_mesh),
49                         Z,
50                         cmap='coolwarm',
51                         linewidth=0,
52                         antialiased=True,
53                         alpha=0.6,
54                         )
55
56
57 ax.scatter(df.iloc[:, 2],
58            np.log10(df.iloc[:, 1]),
59            np.log10(df.iloc[:, 0]),
60            color='black', s=30,label='Datos␣experimentales')
61
62 # Graficar curvas para valores espec ficos de R
63 R_highlight = [0.1, 0.3, 0.5, 0.7]
64
65 for r_val in R_highlight:
66     # Encontrar el  ndice  m s cercano en R_values
67     idx = np.abs(R_values - r_val).argmin()
68
69     # Extraer los datos para este R espec fico
70     x_curve = np.full_like(deltak_range, r_val)
71     y_curve = np.log10(deltak_range)
72     z_curve = Z[:, idx]
73
74     # Graficar la curva
75     ax.plot(x_curve, y_curve, z_curve,
76             color='blue',
77             linewidth=3,
78             alpha=0.7,)
79
80 # Configuraci n de ejes
81 ax.set_xlabel('Relaci n␣de␣carga␣(R)', fontsize=12)
82 ax.set_ylabel('log10( K )␣( M P a m )', fontsize=12)
83 ax.set_zlabel('da/dN␣(mm/ciclo)', fontsize=12)
84 ax.set_title(f'Superficie␣de␣predicci n␣RBFN␣con␣curvas␣
    destacadas\nEspesor␣constante␣=␣{espesor_constante}␣mm',
    fontsize=14)
85
86 # Invertir eje Y (ahora log10( K ))
87 ax.invert_yaxis()
88 ax.invert_xaxis()
89
90 # A adir leyenda
91 ax.legend(fontsize=10)
```

```
92
93 # Rotar para mejor perspectiva
94 ax.view_init(elev=25, azim=45)
95
96 # Ajustar dise o y mostrar
97 plt.tight_layout()
98 plt.show()
```

B.7 Representación MSE punto a punto

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
import pandas as pd
from matplotlib import cm
from matplotlib.colors import Normalize

# Cargar modelo MLP exportado
with open('modelo_mlp_optimizado.pkl', 'rb') as f:
    modelo = pickle.load(f)

model = modelo['model']

# Cargar datos
df = pd.read_excel("BBDD/T351_filtrado.xlsx")

# Par metros
espesor_constante = 9.91
R_vals = [0.1]

# Filtrado por espesor
df = df[df.iloc[:, 3] == espesor_constante]

# Configuraci n de color
cmap = cm.coolwarm
norm = Normalize(vmin=min(R_vals), vmax=max(R_vals))

# Visualizaci n de errores por curva de R
for r_val in R_vals:
    color = cmap(norm(r_val))
    mask = np.abs(df.iloc[:, 2] - r_val) < 0.05

    if not any(mask):
        continue

    # Preparar datos de entrada
    x_exp = df.loc[mask, df.columns[1]].values
    y_exp = df.loc[mask, df.columns[0]].values

    X_input = np.column_stack((
        np.log10(x_exp),
        np.full_like(x_exp, r_val),
        np.full_like(x_exp, espesor_constante)
    ))

    # Predicci n y errores en escala log
    y_pred_log = model.predict(X_input)
```

```
48      y_exp_log = np.log10(y_exp)
49      errores = y_exp_log - y_pred_log
50      rms = np.sqrt(np.mean(errores**2))
51
52      # Gr fico de barras
53      plt.figure(figsize=(6, 4))
54      plt.bar(range(len(errores)), errores, color=color, width
            =0.8)
55      plt.axhline(0, color='black', linewidth=1)
56      plt.grid(axis='y', linestyle=':', alpha=0.6)
57      plt.ylim(-0.5, 0.5)
58      plt.xticks([])
59
60      plt.text(0.95, 0.03, f"RMS␣=␣{rms:.5f}", ha='right', va='
            bottom',
61               transform=plt.gca().transAxes, fontsize=13)
62      plt.text(0.05, 0.8, f"Espesor=␣{espesor_constante}␣mm␣\nR␣=␣
            {r_val}", fontsize=14,
63               ha='left', va='bottom', transform=plt.gca().
                    transAxes)
64
65      plt.ylabel("Error␣(log10)", fontsize=13)
66      plt.tight_layout()
67      plt.show()
```