Escuela
Politécnica
Superior

Exploración de arquitecturas neuronales basadas en Kolmogorov-Arnold Networks

Máster Universitario en Ciencia de Datos

Trabajo Fin de Máster

Autor:
Vicent Baeza Esteve
Tutor/es:
Jorge Calvo Zaragoza

Universitat d'Alacant
Universidad de Alicante

Junio 2025

Exploración de arquitecturas neuronales basadas en Kolmogorov-Arnold Networks

Estudio sobre las arquitecturas neuronales basadas en Kolmogorov-Arnold Networks y sus aplicaciones en inteligencia artificial

Autor
Vicent Baeza Esteve

Tutor/es
Jorge Calvo Zaragoza
Departamento de Lenguajes y Sistemas Informáticos

Máster Universitario en Ciencia de Datos

Escuela Politécnica Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Junio 2025

Resumen

Este proyecto tiene como objetivo explorar las Redes Kolmogorov-Arnold (KAN), un tipo de red neuronal basado en el teorema de representación de Kolmogorov-Arnold, que sostiene que cualquier función continua de varias variables puede representarse como una suma de funciones continuas de una variable. Este enfoque teórico ofrece un marco alternativo a las arquitecturas convencionales de redes neuronales profundas. A través de la implementación y evaluación de redes KAN en distintas aplicaciones de ciencia de datos, se busca comparar su rendimiento con modelos tradicionales como las redes neuronales basadas en perceptrones, midiendo su precisión, eficiencia computacional y capacidad de generalización. Se realizarán experimentos utilizando conjuntos de datos de tareas típicas como regresión, clasificación o predicción en series temporales.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

1 Introducción

La inteligencia artificial es uno de los campos de la informática que más se ha desarrollado recientemente, especialmente al tener en cuenta el crecimiento explosivo que ha tenido en los últimos años. Hace tan solo unas pocas décadas, la inteligencia artificial era un campo muy limitado, utilizado únicamente en ciertas aplicaciones de forma muy puntual. No obstante, gracias al tremendo desarrollo del hardware informático que ha ocurrido desde entonces y al consecuente progreso de las técnicas de machine learning, hoy en día la inteligencia artificial se ha convertido en una parte fundamental de la informática. Gracias al desarrollo de modelos avanzados de machine learning, se han conseguido grandes avances en la clasificación de imágenes [1], el procesamiento de lenguaje natural [2] y la detección de objetos [3], junto con una innumerable cantidad de otras tareas.

Actualmente, prácticamente todos los modelos de machine learning complejos están basados en redes neuronales artificiales. Esto es principalmente por la capacidad de las redes neuronales artificiales de poder aproximar casi cualquier función matemática dados los suficientes parámetros, cosa que hace que se puedan aplicar en una gran variedad de problemas y situaciones. Es gracias a esta increíble flexibilidad y adaptabilidad que las redes neuronales artificiales se han convertido en un pilar del machine learning y de la inteligencia artificial actuales. El diseño de las redes neuronales artificiales está basado en la combinación de transformaciones lineales con parámetros variables junto con transformaciones no-lineales fijas. Repitiendo esta combinación varias veces, las redes neuronales son capaces de aproximar de forma eficiente una gran variedad de funciones matemáticas complejas, necesitando variar únicamente los parámetros de sus transformaciones lineales [4].

No obstante, recientemente ha surgido una alternativa a esta combinación de transformaciones lineales y no-lineales: las redes Kolmogórov-Arnold, o *Kolmogorov-Arnold Networks* (KAN) en inglés. Esta arquitectura novedosa, basada en el teorema de representación de Kolmogórov-Arnold [5], establece una alternativa a la estructura tradicional de las redes neuronales, y hace posible tener transformaciones no-lineales con parámetros variables. Las redes KAN tienen el potencial de mejorar ciertos aspectos de las redes neuronales, como su interpretabilidad, fiabilidad o capacidad de realizar tareas de aprendizaje continuo, además de mejorar la precisión de los modelos en ciertas tareas [6].

Para la realización del trabajo se ha analizado la estructura y el funcionamiento de las redes KAN, prestando especial atención al artículo científico que las propuso en 2024 [6] y a otras publicaciones relacionadas. Se ha explicado esta novedosa arquitectura desde cero, centrándose especialmente en como se diferencian las redes Kolmogórov-Arnold de las redes neuronales ya establecidas. También se han explorado las diferencias que existen a la hora de entrenar y ejecutar este tipo de arquitecturas, junto con las ventajas, desventajas y limitaciones respecto a las redes neuronales tradicionales. A partir de esto, se ha elaborado una implementación desde cero en Python de este tipo de redes, con el fin de mostrar una forma de implementar este tipo de redes y de apoyar la explicación de las mismas.

Además, se han realizado múltiples experimentos en los que se compara el rendimiento de las redes KAN convolucionales con el de las redes CNN tradicionales, utilizando una estructura convolucional KAN derivada de las capas convolucionales de las redes CNN (ver apartado 3.2.1). Se han realizado experimentos para comparar la eficiencia de las redes respecto al número de parámetros (apartado 5.3), la eficiencia respecto al número de datos de entrenamiento (apartado 5.4), la calibración de los modelos entrenados (apartado 5.5) y la calidad de los resultados obtenidos al realizar aprendizaje continuo por fases (apartado 5.6). La metodología concreta de todos los experimentos se describe en sus respectivos apartados.

1.1 Objetivos

Este trabajo tiene los siguientes objetivos concretos:

- **Hacer una revisión de la base teórica de las redes KAN**: Explicar claramente las redes KAN, además de todos los conocimientos necesarios de machine learning y inteligencia artificial necesarios para comprender el funcionamiento de las redes KAN, prestando especial atención a las diferencias principales entre las redes neuronales tradicionales y las redes KAN

- **Hacer una implementación propia de un modelo KAN**: Realizar una implementación en Python de una arquitectura KAN básica, intentando implementar todos los componentes esenciales desde cero, con el fin de mostrar como se podría llegar a realizar una de estas implementaciones.

- **Evaluar y comparar el rendimiento ante alternativas**: Realizar experimentos para intentar medir las propiedades de las redes KAN, comparando los resultados obtenidos con las redes neuronales tradicionales. Se ha decidido realizar estos experimentos comparando las redes KAN convolucionales con las redes CNN (redes convolucionales tradicionales), dada la relativa escasez de experimentos que utilizan las redes KAN convolucionales y comparan los resultados con redes CNN.

- **Hacer un análisis de ventajas y limitaciones de las redes KAN**: Realizar una análisis de las ventajas y desventajas actuales de las arquitecturas KAN, prestando especial atención a las posibles aplicaciones de las redes KAN y a sus ventajas respecto a las arquitecturas contemporáneas de machine learning

- **Proponer líneas futuras de investigación**: Proponer futuras líneas de investigación relacionadas con las redes KAN, haciendo énfasis en las líneas de investigación más prometedoras o que tengan mayor utilidad para el machine learning y la inteligencia artificial

1.2 Estructura del trabajo

A continuación se describe la estructura del trabajo, junto con un breve resumen del contenido de cada capítulo.

- **Capítulo 1 - Introducción**: introducción al trabajo, explicando el propósito del trabajo, los objetivos concretos y la estructura del mismo
- **Capítulo 2 - Inteligencia artificial**: introducción de los conceptos fundamentales de machine learning y de inteligencia artificial, donde se explican las ideas necesarias para entender el resto del trabajo. Se presta especial atención a las técnicas y arquitecturas mencionadas en la explicación las redes KAN o que han sido utilizadas para realizar los experimentos.
- **Capítulo 3 - Redes Kolmogórov-Arnold**: explicación detallada de las redes KAN, describiendo todo el fundamento matemático que las acompaña, haciendo énfasis en las diferencias entre las redes KAN y las redes neuronales tradicionales
- **Capítulo 4 - Implementación en Python**: implementación en Python desde cero de las redes KAN, junto con una explicación detallada del código de las funciones principales utilizadas en la implementación
- **Capítulo 5 - Experimentos**: se detallan y describen los experimentos realizados para establecer las propiedades de las redes KAN convolucionales y comparar el rendimiento con el de las redes convolucionales ya establecidas.
- **Capítulo 6 - Conclusiones**: las conclusiones de todo el trabajo, incluyendo un resumen de los resultados obtenidos en los experimentos y las futuras líneas de investigación propuestas

2 Inteligencia artificial

Para poder entender de forma correcta el funcionamiento de las redes KAN, primero es necesario establecer ciertos conceptos fundamentales del campo de inteligencia artificial. En este apartado del trabajo se ha realizado una breve introducción de muchos de estos conceptos necesarios para entender las redes neuronales actuales, entrando específicamente en el detalle de algunas de las técnicas y arquitecturas en las que están basadas las redes KAN, y que serán necesarias para entender los consecuentes apartados de este trabajo.

El campo de la inteligencia artificial se dedica a la resolución de una gran cantidad de problemas, como la clasificación de imágenes [1], el procesamiento de lenguaje natural [2] o la detección de objetos [3], entre muchos otros. Mediante el uso de ciertos mecanismos y procesos que se verán más adelante, es posible ajustar los parámetros de un modelo de machine learning de tal forma que los resultados del modelo se aproximen los resultados necesarios para resolver la tarea buscada, de forma que el modelo sea capaz de reconocer patrones tan complejos y/o numerosos que su implementación mediante programación tradicional se vuelve inviable [4]. Este proceso de ajustar los parámetros de un modelo con el fin de que el modelo resultante resuelva una tarea se conoce como el entrenamiento de un modelo.

En la inteligencia artificial existen muchos tipos de tareas, cada una con su conjunto de técnicas y arquitecturas asociados. No obstante, es común clasificar las tareas de inteligencia artificial en 3 tipos generales:

- Aprendizaje supervisado: consiste en proveer al modelo de muchos ejemplos con las salidas esperadas para cada ejemplo. A partir de estos ejemplos, el modelo es capaz de ir disminuyendo el error de salida y pasa, poco a poco, a ir prediciendo cada vez mejor los resultados a partir del conjunto de datos del que se extrajeron los ejemplos. A la que aumenta la complejidad de los patrones que se quieren aprender o la precisión necesaria, se requieren más y más datos, cosa que aumenta también el tiempo de entrenamiento del modelo. Además, si no se proveen de suficientes datos o los datos no son de suficientemente buena calidad, se pueden obtener resultados subóptimos a la hora de entrenar el modelo [7].

- Aprendizaje no supervisado: para entrenar un modelo utilizando aprendizaje no supervisado, necesitamos proveer de ejemplos pero no necesariamente sus correspondientes salidas. En este tipo de aprendizaje el modelo extrae patrones de los datos mediante el análisis de patrones o de semejanzas entre los datos. Aunque construir un conjunto de datos para este tipo de aprendizaje es más sencillo, extraer información útil no es tan simple como para el aprendizaje supervisado, ya que muchas veces los resultados van a ser más difíciles de interpretar y aprovechar [8].

- Aprendizaje por refuerzo: a diferencia que con los otros dos tipos de aprendizaje, el modelo no recibe ningún conjunto de datos. Aprende mediante la interacción con el

entorno, siendo recompensado o penalizado dependiendo de las acciones que tome y sus consecuencias. Estos algoritmos son específicos a un problema concreto, y necesitan una función de recompensa bien diseñada para que el modelo aprenda a interactuar con el entorno de la forma que se espera. Mediante prueba y error el modelo poco a poco aprende las acciones que obtienen la mayor recompensa para el entorno, y de esta forma va optimizando sus interacciones para obtener la recompensa máxima [9].

A día de hoy, la gran mayoría de arquitecturas de inteligencia artificial están basadas en redes neuronales [10]. Este tipo de arquitecturas están formadas por neuronas artificiales, que son nodos en un grafo computacional que calculan un valor a partir de sus conexiones a otras neuronas artificiales de la red. Estas redes de neuronas estaban inspiradas originalmente en las redes neuronales biológicas encontradas en el cerebro, e imitan de forma muy simplificada como las neuronas en el tejido cerebral están conectadas entre ellas [11].

En los siguientes apartados se procede a explicar el funcionamiento de las redes neuronales y su entrenamiento mediante el uso del aprendizaje supervisado. Primero se abordarán ciertos casos simples, como el funcionamiento de una única neurona artificial, y se desarrollará paso a paso la teoría necesaria para entender como funciona una red en su totalidad. Además, se abordarán con especial detalle los conceptos necesarios para entender las redes KAN, que se tratarán en los apartados siguientes.

Durante todo este trabajo ha sido necesario representar una gran variedad de modelos computacionales. Para representar estos modelos se ha utilizado un grafo que muestra el flujo de datos del modelo, en concordancia con el formato representado en la figura 2.1.

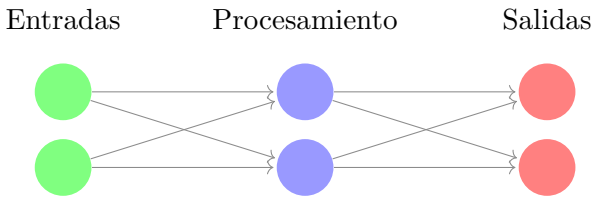


Figura 2.1: Representación de un modelo computacional como un grafo, con entradas (verde), nodos de procesamiento (azul) y nodos de salda (rojo). Las aristas del grafo indican el flujo de datos del modelo. Fuente: elaboración propia

2.1 Función objetivo

Antes de poder empezar a explicar el funcionamiento de las redes neuronales, es necesario concretar exactamente el objetivo a la hora de utilizar este tipo de estructuras. De forma muy simplificada, al entrenar un modelo mediante aprendizaje supervisado lo que se intenta es aproximar una función objetivo $f^* : \mathbb{R}^n \to \mathbb{R}^m$, que define el comportamiento que es deseable que el modelo muestre. Por ejemplo, una función objetivo podría ser: dados los píxeles RGB de una imagen decir si es una imagen de un gato o de un perro. Si las imágenes de entrada son de $25 \cdot 25$ píxeles, podríamos codificar la función como $f^* : \mathbb{R}^{25 \cdot 25 \cdot 3} \to \{0, 1\}$, siendo una salida de 0 una imagen de un gato y una salida de 1 una imagen de un perro. Otras funciones pueden ser mucho más complejas de modelar, hasta el punto de que en algunos casos puede ser necesario utilizar una aproximación, ya que la función objetivo puede ser inviable

o imposible de definir exactamente [12].

Aunque muchas funciones objetivo son fáciles de describir informalmente, producir un algoritmo que compute estas funciones mediante programación tradicional es prácticamente imposible. No obstante, lo que si que es viable es producir ejemplos concretos de la función. De forma simplificada, es necesario recopilar muchas parejas de valores de entrada x_i y valores de salida y_i, de forma que los valores de salida correspondan a la función objetivo ($y_i = f^*(x_i)$), o que al menos aproximen la salida correcta ($y_i \approx f^*(x_i)$) si no se pueden conseguir resultados exactos. Siguiendo el ejemplo ya establecido, sería necesario conseguir muchas imágenes RGB de $25 \cdot 25$ píxeles de gatos (en cuyo caso su $y_i = 0$) y de perros (en cuyo caso su $y_i = 1$).

El objetivo del aprendizaje supervisado es que, mediante el uso de todos estos ejemplos, se obtenga un modelo que se pueda ir modificando poco a poco para que produzca resultados que se aproximen más y más a los ejemplos del conjunto de datos. Si los datos seleccionados son representativos y se evitan ciertos problemas que pueden ocurrir durante el entrenamiento [7], se espera que, a partir del conjunto finito de ejemplos el modelo resultante produzca una salida f que aproxime la función objetivo f^* para todas las posibles entradas x_i ($f(x) \approx f^*(x)\ \forall x \in \mathbb{R}^m$).

La calidad de los resultados obtenidos tras el entrenamiento puede variar tremendamente, ya que depende de muchos factores que tienen que ser analizados en detalle. Estos factores incluyen la cantidad y calidad de los datos utilizados, la complejidad de la función objetivo y el diseño del modelo de machine learning entre muchos otros. Para obtener resultados buenos se tienen que tener en cuenta muchos de estos factores y realizar pruebas frecuentes que evalúen los resultados obtenidos. La evaluación de los modelos después del entrenamiento se estudia en detalle en el apartado 2.5.

2.2 Neuronas artificiales

En el contexto de las redes neuronales artificiales, una neurona es un único nodo de la red, que tiene varias entradas de datos y produce una única salida a partir de estas entradas. La neurona calcula el valor de salida realizando una suma ponderada de las entradas, multiplicando cada entrada por su peso correspondiente. Además, también se añade al resultado un valor que no se multiplica por ninguna salida, al que se le llama *bias* o sesgo.

El resultado de esta suma, que se conoce como la activación de la neurona, se pasa por la una función no-lineal, con la finalidad de que las neuronas no sean transformaciones lineales de las entradas, ya que entonces la red neuronal no sería capaz de representar funciones no-lineales. Estas funciones no-lineales son las funciones de activación, y son una parte fundamental de las redes neuronales artificiales [13].

Sea σ la función de activación, x las entradas, w los pesos y b el bias, podemos representar matemáticamente el resultado de la función con la fórmula siguiente:

$$y = \sigma\left(b + \sum_{i=1}^{n} w_i x_i\right) \tag{2.1}$$

Para simplificar la ecuación anterior, normalmente se añade una "entrada" x_0 cuyo valor es siempre 1, y se utiliza w_0 como el *bias*. Aunque el resultado es el mismo, la ecuación se simplifica bastante, por lo que es preferible utilizar la forma simplificada:

$$y = \sigma\left(\sum_{i=0}^{n} w_i x_i\right) \tag{2.2}$$

Visualmente, podemos representar la ecuación anterior de la siguiente forma:

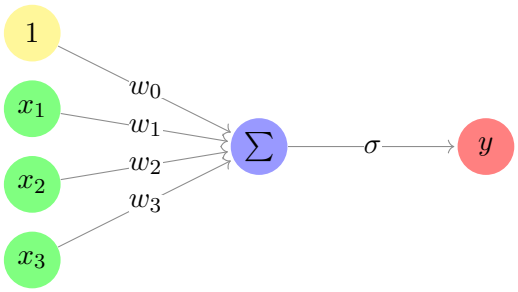


Figura 2.2: Visualización de una neurona artificial con 3 entradas, en la que se representan las entradas (x_i), los pesos (w_i), la suma ponderada y la función de activación (σ). Fuente: elaboración propia

Esta es la fórmula más básica para las neuronas artificiales. Existen muchas otras variantes de esta fórmula que se utilizan en ciertos contextos específicos.

2.2.1 Funciones de activación

Existen muchas funciones de activación utilizadas en modelos de redes neuronales, teniendo cada una ciertas ventajas y desventajas que hay que tener en cuenta. La gran mayoría son funciones no lineales, que se utilizan para hacer que la salida de la neurona deje de tener una relación lineal con las entradas. Esto es particularmente importante al considerar como una red neuronal encadena varias neuronas entre sí, ya que sin el uso de funciones no lineales una red neuronal solo es capaz de producir salidas lineales a sus entradas, y por lo tanto no sería útil a la hora de aproximar una gran cantidad de funciones objetivo [13]. A continuación se pueden ver algunos ejemplos de funciones de activación comúnmente utilizadas:

- Lineal: $\sigma(x) = x$
- Sigmoide: $\sigma(x) = 1/(1 + e^{-x})$
- Tangente hiperbólica: $\sigma(x) = \tanh(x)$
- ReLU: $\sigma(x) = \max(0, x)$
- SiLU: $\sigma(x) = x/(1 + e^{-x})$
- Softplus: $\sigma(x) = \ln(1 + e^{x})$

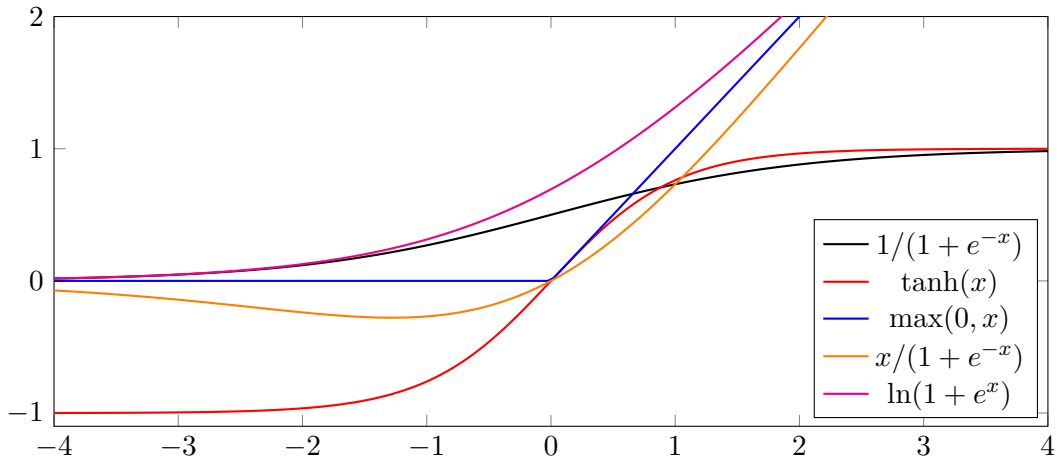


Figura 2.3: Representación gráfica de las funciones de activación sigmoide (negro), tangente hiperbólica (rojo), ReLU (azul), SiLU (naranja) y softplus (rosa). Fuente: elaboración propia

2.3 Redes neuronales artificiales

Una red neuronal es un conjunto de neuronas artificiales conectadas entre sí. Las primeras neuronas de la red, que no dependen de la salida de ninguna otra neurona, reciben directamente su valor. Es mediante estas neuronas que se le pasan los datos de entrada a la red. Estas neuronas se conocen como las neuronas de entrada de la red. De forma análoga, las últimas neuronas de la red son las neuronas de salida. Las neuronas de salida no tienen ninguna neurona conectada a su salida, y su valor calculado es uno de los valores de salida de la red.

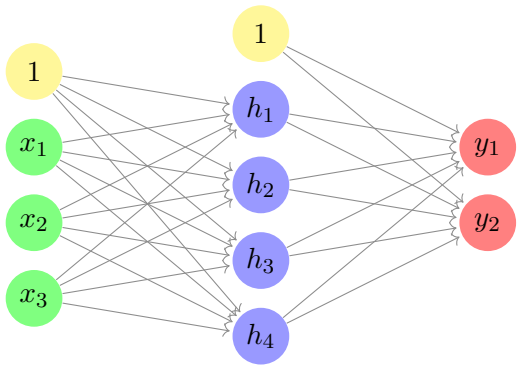


Figura 2.4: Red neuronal con 3 neuronas de entrada (verde), 4 neuronas ocultas (azul) y 2 neuronas de salida (rojo). Fuente: elaboración propia

Por razones tanto de eficiencia como de funcionalidad, en una red neuronal las neuronas se suelen agrupar en capas. Generalmente, todas las neuronas de una capa se comportan de la misma forma, y se conectan con otras capas de forma similar. Además, todas las neuronas de la misma capa suelen tener la misma función de activación y los mismos parámetros de configuración. De esta forma, es posible diseñar grandes redes neuronales con miles o millones

de neuronas sin tener que especificar la función de activación, las conexiones o los parámetros de configuración de cada una de las neuronas de la red. Las capas que contienen las neuronas de entradas son las capas de entrada, las que contienen las neuronas de salida son las capas de salida y el resto de capas son las capas ocultas.

2.3.1 Capas densas

La capas de neuronas más básicas son las capas densas, también conocidas como *Fully Connected Layers*, o capas completamente conectadas. En este tipo de capas cada neurona de la capa está conectada a todas entradas recibidas por la capa. Un ejemplo visual de una de estas capas se puede ver en la figura 2.5.

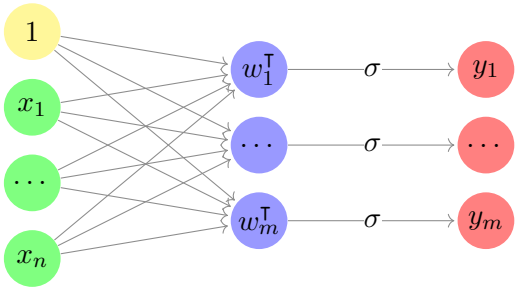


Figura 2.5: Representación visual de una capa densa con n entradas y m salidas, siendo w_i los pesos de cada neurona y σ la función de activación de la capa. Fuente: elaboración propia

Matemáticamente, una capa densa se puede modelar a partir de la fórmula 2.2 ya introducida anteriormente. Sean x las salidas de la capa anterior y w_{ij} el peso w_i de la neurona j de la capa, se calcula la salida y de cada neurona de la siguiente forma:

$$y_j = \sigma\left(\sum_i w_{ij} x_i\right) \tag{2.3}$$

Al tratar w y x como vectores, es posible simplificar la ecuación:

$$y_j = \sigma\left(\begin{bmatrix} w_{0j} & \dots & w_{nj} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}\right) = \sigma\left(w_{\bullet j}^\intercal x\right) \tag{2.4}$$

A partir de esto, también se es posible definir el comportamiento de toda la capa densa utilizando una sola ecuación:

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{00} & \dots & w_{n0} \\ \vdots & \ddots & \vdots \\ w_{0m} & \dots & w_{nm} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}\right) = \sigma\left(w^\intercal x\right) \tag{2.5}$$

Para que los cálculos de las ecuaciones 2.4 y 2.5 sean equivalentes a los de la ecuación 2.3

es necesario transponer la matriz de pesos w. Además, en concordancia con la simplificación realizada en la ecuación 2.2, $x_0 = 1$ para todas las ecuaciones anteriores, ya que w_{0j} es el sesgo de cada una de las neuronas de la capa.

Una vez obtenida la fórmula para calcular la salida de una capa densa, podemos calcular la salida de toda una red. Esto se puede hacer calculando la salida de la primera capa de la red a partir de las entradas de la red, y luego utilizando las salidas de la primera red como las entradas de la siguiente capa. A partir de esto, podemos ir encadenando todas las capas de la red, hasta llegar a la salida de la última capa, que representa la salida de la red entera.

Sean $w_1, \ldots, w_l$ los pesos de cada capa de la red, $\sigma_1, \ldots, \sigma_l$ las funciones de activación de cada capa de la red, y $\Phi_i(x) = \sigma_i(w_i^\intercal x)$, se obtiene la siguiente fórmula para calcular todos los pesos de la red, siendo Φ_i la función que representa cada una de las capas de la red:

$$\begin{aligned} y &= \Phi_l \circ \Phi_{l-1} \circ \cdots \circ \Phi_2 \circ \Phi_1 \,(x) \\ &= \sigma_l(w_l^\intercal \ \ldots \ \sigma_2(w_2^\intercal \ \sigma_1(w_1^\intercal)) \ldots) \end{aligned} \tag{2.6}$$

2.3.2 Capas convolucionales

Cuando estamos tratando con objetos que contienen una gran cantidad de información estructurada, como imágenes o ficheros de audio, se puede volver muy costoso utilizar capas densas, ya que la cantidad de pesos requeridos crece muy rápidamente con el tamaño de las entradas. Por ejemplo, para imágenes de 100×100 píxeles, tendríamos 10 000 neuronas para la capa de entradas de la red. Para conectar una capa densa con esta cantidad de neuronas, necesitaríamos una matriz de pesos de 10000×10000 elementos. Para imágenes más grandes, este número se vuelve aún mayor.

Para intentar reducir el coste computacional y a la vez explotar la estructura de los datos recibidos se crearon las capas convolucionales [14]. Este tipo de capas, en vez de conectar todas las neuronas entre sí, hacen que solo puedan afectar a una salida las entradas "cercanas" a esta salida. Esto es análogo a como se calculan las convoluciones matemáticas discretas [15].

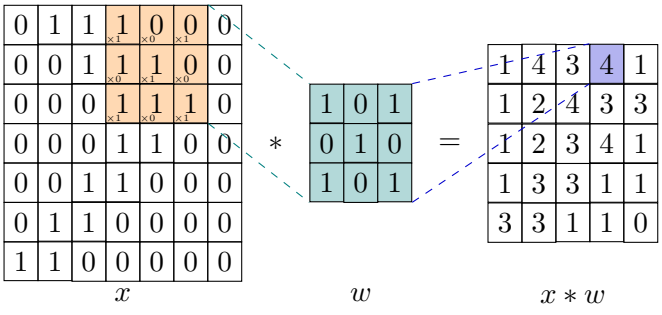


Figura 2.6: Representación gráfica de una convolución discreta 2D, cuyos operandos y resultado son matrices bidimensionales. Fuente: [16]

Las entradas de las capas convolucionales tienen cierta forma, la cual depende de la estructura de los datos de entrada:

- Datos 1D (ficheros de audio, series temporales, etc.): hay n posiciones de entrada,

teniendo cada una s datos, por lo que recibimos una entrada de $\mathbb{R}^{n \times s}$.

- Datos 2D (imágenes): hay $n \times m$ posiciones de entrada, teniendo cada una s datos, por lo que tenemos una entrada de $\mathbb{R}^{n \times m \times s}$.
- Datos 3D o superiores (videos, nubes de puntos 3D, etc.): hay $n_1 \times \cdots \times n_m$ posiciones de entrada, teniendo cada una s datos, por lo que tenemos una entrada de $\mathbb{R}^{n_1 \times \cdots \times n_m \times s}$.

Las capas convolucionales tienen *kernels*, que son estructuras matriciales que contienen todos los pesos entrenables de la capa. Cada uno de estos *kernels* tiene el mismo número de dimensiones que los datos de entrada, aunque su tamaño en cada una de estas dimensiones depende del tamaño de kernel elegido. Si tenemos una entrada de datos de forma $n \times m \times s$ y un tamaño de kernel de $p \times q$, entonces cada kernel tendrá una forma de $p \times q \times s$. El tamaño de kernel normalmente es un número bastante pequeño comparado con las dimensiones de entrada, siendo el valor más común 3×3. Podemos ver un ejemplo de un kernel $3 \times 3 \times 1$ en la figura 2.6, que es equivalente a realizar una convolución discreta 2D con un operando de tamaño 3×3.

Las capas convolucionales también hacen uso de funciones de activación, con el fin de que el resultado producido por la capa no sea una combinación lineal de sus salidas. Si no se utilizasen funciones de activación, al igual que para las capas densas, esto impediría que la red sea capaz de representar funciones no-lineales y por lo la red no sería capaz de aprender correctamente la gran mayoría de funciones objetivo [13].

Una vez establecida la forma de los kernels se pueden definir matemáticamente las salidas producidas por una capa convolucional. Sea $x \in \mathbb{R}^{n \times m \times s}$ la entrada, $w^k \in \mathbb{R}^{p \times q \times s}$ los kernels de la capa, $a * b$ la convolución discreta de a y b, y σ la función de activación, podemos definir las salidas de una capa convolucional de la siguiente forma:

$$y_{i,j,k} = \sigma\left((x * w^k)_{i,j}\right) = \sigma\left(\sum_{a,b,c} x_{i+a,j+b,c}\, w^k_{a,b,c}\right) \tag{2.7}$$

Las redes convolucionales también hacen uso de capas de pooling, que las hacen más resistentes al ruido a la vez de que reducen el tamaño de la red sin afectar demasiado a la generalización de la misma. Estas capas, normalmente introducidas después de las capas convolucionales, generan valores dividiendo los datos en regiones y calculando un único valor para cada región. Lo más frecuente es, para cada región 2×2 quedarse con el valor máximo (conocido como *Max Pooling*) o el valor medio (*Average Pooling*) [17]. En la figura 2.7 se puede ver un ejemplo de Max Pooling.

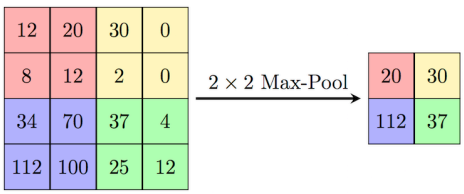


Figura 2.7: Visualización de una operación de Max Pooling bidimensional, con regiones de tamaño 2×2. Fuente: [18]

2.4 Entrenamiento

Hemos visto como calcular los valores de salida de una red neuronal. No obstante, para que la red sea capaz de resolver problemas computacionales vamos a tener que entrenarla. El entrenamiento de una red es el proceso de actualizar los parámetros entrenables, de forma que el resultado que produce la red se ajuste al resultado deseado. Estos parámetros normalmente son las matrices de pesos de las capas (w_i), aunque dependiendo de la arquitectura de la red y de las capas usadas pueden existir mas parámetros modificables durante el entrenamiento.

Para poder entrenar una red, primero es necesario cuantificar el error que ha cometido. El error E es simplemente lo lejos que está cada salida producida por la red de la salida esperada para ciertos datos de entrada. Para definir matemáticamente el error, tenemos que seleccionar la función de pérdida. Hay muchas funciones de pérdida, siendo una de las más simples la diferencia cuadrada entre la salida obtenida y la salida esperada:

$$E_i = (y_i - \hat{y}_i)^2 \tag{2.8}$$

La función de pérdida define exactamente como estamos midiendo el error entre la salida producida (y) y la salida esperada ($\hat{y}$). Existen muchas otras funciones de pérdida [19], cada una con sus usos, ventajas y desventajas.

2.4.1 Retropropagación

Para actualizar los pesos de una red neuronal, es necesario calcular la derivada respecto al error para cada parámetro entrenable de la red. Una vez obtenida esta derivada, podemos ajustar cada uno de los parámetros utilizando la derivada obtenida de forma que el error de la red se disminuya tras la actualización. Para calcular estas derivadas, normalmente se utiliza retropropagación (*backpropagation* en inglés), que es un algoritmo que permite propagar el error desde las capas de salida hasta las capas de entrada para poder calcular las derivadas de forma eficiente [20].

La siguiente sección explica el algoritmo de backpropagation. Para poder describirlo de forma precisa, va a ser necesario primero definir la notación que se ha utilizado. Se utilizarán las siguientes variables durante la explicación del algoritmo:

- x_i^l: valores de entrada de la capa l.
- $w_{i,j}^l$: pesos de la capa l.
- z^l: notación para $\left(w^l\right)^\intercal x^l$. Esto implica que $z_i^l = (w_{\bullet,i}^l)^\intercal x = w_{i,\bullet}^l x$.
- σ_l: función de activación de la capa l.
- a^l: valores de salida de la capa l. $a^l = \sigma_l(z^l)$
- y: salida de la red. $y = a^L$, siendo L la cantidad de capas de la red y, por lo tanto, el índice de la última capa.
- $\hat{y}$: salida esperada de la red.
- E: error de la red

- E' derivada del error de la red respecto a y, $E' = \partial E / \partial y$

La derivada que buscamos obtener para actualizar los pesos de la red es $\partial E / \partial w^l_{i,j}$ para todos los pesos de todas las capas de la red. Para poder calcular esta derivada de forma eficiente para todas las capas definiremos un valor intermedio, llamado delta (δ). Los deltas de una capa son la derivada del error respecto a z^l ($\delta^l_i = \partial E / \partial z^l_i$). A partir de estos valores, es muy simple calcular la derivada de los pesos $\partial E / \partial w^l_{i,j}$:

$$\frac{\partial E}{\partial w^l_{i,j}} = \frac{\partial E}{\partial z^l_i} \frac{\partial z^l_i}{\partial w^l_{i,j}} = \delta^l_i x^l_j \tag{2.9}$$

Entonces, para calcular $\partial E / \partial w^l_{i,j}$ únicamente necesitamos calcular δ^l_i para todas las capas de la red. Podemos calcular el delta de la última capa de la red (δ^L_i) a partir de E de forma directa:

$$\delta^L_i = \frac{\partial E_i}{\partial z^L_i} = \frac{\partial E_i}{\partial a^L_i} \frac{\partial a^L_i}{\partial z^L_i} = \frac{\partial E_i}{\partial y_i} \frac{\partial \sigma_L(z^L_i)}{\partial z^L_i} = E'_i \cdot \sigma'_L(z^L_i) \tag{2.10}$$

Los deltas del resto de capas de la red se pueden calcular a partir de los deltas de la capa siguiente de la red. Para obtener la fórmula, primero vamos a aplicar la definición de δ^l_i para ponerlo en términos de δ^{l+1}:

$$\delta^l_i = \frac{\partial E}{\partial z^l_i} = \sum_j \left(\frac{\partial E}{\partial z^{l+1}_j} \frac{\partial z^{l+1}_j}{\partial z^l_i} \right) = \sum_j \left(\delta^{l+1}_j \frac{\partial z^{l+1}_j}{\partial z^l_i} \right) \tag{2.11}$$

Simplificamos $\partial z^{l+1}_j / \partial z^l_i$:

$$\frac{\partial z^{l+1}_j}{\partial z^l_i} = \frac{\partial w^{l+1}_{j,\bullet} x^{l+1}}{\partial z^l_i} = \frac{\partial w^{l+1}_{j,\bullet} \sigma_l(z^l)}{\partial z^l_i} = w^{l+1}_{j,i} \sigma'_l(z^l_i) \tag{2.12}$$

Combinando las ecuaciones 2.11 y 2.12, obtenemos la siguiente fórmula para calcular δ^l a partir de δ^{l+1}:

$$\delta^l_i = \sum_j \left(\delta^{l+1}_j \, w^{l+1}_{j,i} \, \sigma'_l(z^l_i) \right) = \delta^{l+1} \, w^{l+1}_{\bullet,i} \cdot \sigma'_l(z^l_i) = (w^{l+1}_{i,\bullet})^\intercal \, \delta^{l+1} \cdot \sigma'_l(z^l_i) \tag{2.13}$$

A partir de las ecuaciones 2.10 y 2.13, podemos calcular $\delta^1, \ldots, \delta^L$ calculando δ^L con la ecuación 2.10 y aplicando la ecuación 2.13 para obtener el resto de valores. A partir de esto, es posible calcular las derivadas de todos los pesos utilizando la ecuación 2.9.

2.4.2 Descenso por gradiente

Una vez tenemos las derivadas de todos los pesos de la red, podemos ajustarlos según las derivadas mediante el uso de un optimizador. Un optimizador es el algoritmo que controla como cambiamos los pesos de la red para minimizar el error de forma eficiente. Uno de los optimizadores más sencillos es descenso por gradiente, que consiste en restar a los pesos una fracción de su derivada. La fracción que restamos la controlamos con la *learning rate*, que es un parámetro ajustable. Una learning rate demasiado baja significa que la red neuronal

tardará más de lo necesario en entrenarse, mientras que una learning rate demasiado alta puede causar problemas de estabilidad y hacer que el modelo no se entrene bien. Sea μ la learning rate y $\partial E/\partial w_{ij}^l$ la derivada de $w_{i,j}^l$ respecto al error de la red, tenemos la siguiente fórmula para actualizar los pesos de la red:

$$w_{i,j}^l \leftarrow w_{i,j}^l - \mu \frac{\partial E}{\partial w_{i,j}^l} \tag{2.14}$$

Aunque este optimizador tan simple es capaz de minimizar muchas funciones, existen muchos otros optimizadores que, aunque sean más complejos, consiguen minimizar el error de forma mucho más rápida [21].

2.5 Evaluación de modelos

Una vez se ha realizado el entrenamiento de un modelo, es de vital importancia tener la capacidad de medir lo acertados que son los resultados producidos por el modelo a la hora de realizar la tarea para la que ha sido entrenado. En este apartado se explican varias técnicas y conceptos relevantes a la hora de evaluar el rendimiento de modelos de machine learning, que serán utilizados cuando se analicen e interpreten los resultados obtenidos en los experimentos realizados (ver capítulo 5). Este apartado se centra únicamente en explicar técnicas y conceptos relevantes para medir el rendimiento de modelos entrenados en tareas de clasificación, ya que los experimentos se han realizado utilizando modelos entrenados para este tipo de tareas.

2.5.1 División de datos

A la hora de entrenar y evaluar modelos de inteligencia artificial, en la gran mayoría de casos es imposible proporcionar al modelo todos los posibles casos de entrada y salida que tiene que aprender. En vez de eso, se intenta que el modelo generalice estos casos a partir de un conjunto limitado de datos de entrenamiento. Por lo tanto, a la hora de evaluar un modelo generalmente lo que queremos medir es su capacidad de generalizar los patrones aprendidos en el entrenamiento al encontrarse con datos nuevos, no lo bien que predice los datos ya aprendidos.

Es por esto que, a la hora de medir el rendimiento de los modelos se suelen dividir los datos en dos conjuntos: uno de entrenamiento y otro de evaluación. Los datos del conjunto de entrenamiento únicamente se utilizan para entrenar el modelo, mientras que los datos de evaluación solo se utilizan para medir el rendimiento del modelo. De esta forma, el rendimiento obtenido a la hora de evaluar el modelo reflejará mucho mejor la capacidad de generalización del modelo [22].

Para asegurar que la división de datos es efectiva, hay que asegurarse que no hay muestras repetidas en el conjunto de datos de entrenamiento y evaluación. En caso de que existan algunas muestras que estén en ambos conjuntos, los resultados obtenidos no serán del todo representativos de la capacidad de generalización del modelo, ya que para esos datos el modelo ya ha tenido una oportunidad de aprender el valor correcto durante el entrenamiento [23].

2.5.2 Logits

En tareas de clasificación los modelos se entrenan para, dado un conjunto de entradas, predecir a cuál de las posibles clases de salida pertenece. No obstante, en la gran mayoría de arquitecturas, el modelo no produce una única salida que nos dice cuál clase ha predecido, si no que produce un valor para cada clase posible. Estos valores se conocen como los *logits* del modelo.

Los logits producidos por un modelo pueden ser cualquier número real. Es por esto que, generalmente, los logits se normalizan antes de ser utilizados. Aunque existen varios métodos, uno de los más comunes es la normalización *softmax* [24], que emplea la función exponencial para normalizar los logits de forma que todos los logits normalizados estén entre 0 y 1, y la suma de todos los logits normalizados sea 1. La salida función softmax $\sigma : \mathbb{R}^n \mapsto [0,1]^n$ se puede definir de la siguiente forma:

$$\sigma(l)_i = \frac{e^{l_i}}{\sum_j e^{l_j}} \tag{2.15}$$

Una vez tenemos los logits normalizados, podemos saber cuál es la clase predecida por el modelo cogiendo la clase cuyo logit sea mayor:

$$y = \underset{i}{\operatorname{argmax}}\ \sigma(l_i) = \underset{i}{\operatorname{argmax}}\, l_i \tag{2.16}$$

También es relevante la *confianza* que tiene el modelo, que es el valor del logit normalizado de la clase predecida:

$$c = \max_i \sigma(l_i) \tag{2.17}$$

Los resultados con menor confianza producidos por un modelo deberían tratarse con cuidado, ya que generalmente estos resultados serán menos fiables que los resultados con mayor confianza. No obstante, aunque una menor confianza generalmente indica un resultado menos fiable y viceversa, no se debería interpretar la confianza directamente como la probabilidad de que la salida predecida por un modelo sea correcta, ya que estos dos valores pueden diferir significativamente [25]. La diferencia entre la confianza y la probabilidad de que una predicción sea correcta se explora más adelante en el apartado 2.5.5.

2.5.3 Métricas de rendimiento

Una de las principales características que queremos medir a la hora de evaluar un modelo es el rendimiento, es decir, lo bien que realiza la tarea para la que ha sido entrenado. Para medir el rendimiento de modelos de clasificación existen muchas métricas, cada una con sus ventajas y desventajas [26].

2.5.3.1 Tasa de aciertos

A la hora de entrenar modelos de clasificación, una de las métricas mas simples e intuitivas es la tasa de aciertos, que es la cantidad de aciertos obtenidos por el modelo dividido entre la cantidad de datos totales en el conjunto de evaluación. Para obtener esta métrica, simplemente

tenemos que comparar la cantidad de veces que el modelo ha acertado la clase entre la cantidad de muestras totales de evaluación. Sean y las clases predecidas por el modelo e $\hat{y}$ las clases esperadas del conjunto de evaluación, tenemos la siguiente definición:

$$\text{tasa de aciertos} = \frac{\text{aciertos}}{\text{total}} = \frac{|i : y_i = \hat{y}_i|}{|\hat{y}_i|} \tag{2.18}$$

La tasa de aciertos, aunque es una métrica muy intuitiva y simple, en mucho casos puede no ser ideal para medir el rendimiento de los modelos, especialmente cuando las clases del conjunto de evaluación tengan un número de muestras muy desequilibrado [27].

2.5.3.2 Precisión, exhaustividad y f-score

Para calcular muchas de las métricas de rendimiento de modelos de clasificación, es necesario calcular el número de muestras verdaderas positivas (VP), verdaderas negativas (VN), falsas positivas (FP) y falsas negativas (FN) de cada clase [26]. Podemos ver en la tabla 2.1 como a partir de la predicción y del modelo y el valor real de la muestra $\hat{y}$ podemos definir a cuál de las cuatro clasificaciones (VP, VN, FP, FN) pertenece cada muestra.

Valor verdadero	Predicción	
	Positivo ($y = k$)	Negativo ($y \neq k$)
Positivo ($\hat{y} = k$)	Verdaderos positivos (VP)	Falsos negativos (FN)
Negativo ($\hat{y} \neq k$)	Falsos positivos (FP)	Verdaderos negativos (VN)

Tabla 2.1: Relación entre la predicción y y el valor verdadero $\hat{y}$ respecto a una clase k. Muestra como se definen los verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos para la clase k a partir de la predicción y el valor verdadero. Fuente: elaboración propia

Matemáticamente, podemos calcular la cantidad de verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos de una clase k de la siguiente forma:

$$\begin{aligned} \text{VP}_k &= |i : y_i = k \wedge \hat{y}_i = k| \\ \text{VN}_k &= |i : y_i \neq k \wedge \hat{y}_i \neq k| \\ \text{FP}_k &= |i : y_i = k \wedge \hat{y}_i \neq k| \\ \text{FN}_k &= |i : y_i \neq k \wedge \hat{y}_i = k| \end{aligned} \tag{2.19}$$

A partir de estos valores, podemos calcular la precisión y la exhaustividad de cada clase, dos métricas que se utilizan frecuentemente para medir el rendimiento de los modelos:

- La precisión mide la fracción de valores predecidos como positivos que son realmente positivos. Se define como la tasa de verdaderos positivos frente al total de valores predecidos como positivos, $\text{VP}/(\text{VP} + \text{FP})$.
- La exhaustividad la fracción de valores realmente positivos que son predecidos como

positivos. Se define como la tasa de verdaderos positivos frente al total de valores realmente positivos, $\mathrm{VP}/(\mathrm{VP}+\mathrm{FN})$.

Estas dos métricas, aunque en ciertos casos se suelen utilizar de forma directa, normalmente se combinan para formar el Valor-F, o *f-score*, que se define como la media armónica de la precisión y la exhaustividad. Al igual que para la precisión y la exhaustividad, esta métrica se calcula para cada clase. Podemos definir la *f-score* para una clase k de la siguiente forma:

$$\mathrm{F}_{1,k} = 2\,\frac{\mathrm{precision}_k \cdot \mathrm{exhaustividad}_k}{\mathrm{precision}_k + \mathrm{exhaustividad}_k} = \frac{2\mathrm{VP}_k}{2\mathrm{VP}_k + \mathrm{FP}_k + \mathrm{FN}_k} \tag{2.20}$$

También es posible ponderar la media armónica utilizando un parámetro $\beta > 0$, como se puede ver ecuación 2.21. Este parámetro indica la cantidad de veces que es más importante la exhaustividad que la precisión. Para $\beta > 1$ se le otorga mayor ponderación a la exhaustividad, mientras que para $\beta < 1$ se le da mayor ponderación a la precisión.

$$\mathrm{F}_{\beta,k} = (1+\beta^2)\,\frac{\mathrm{precision}_k \cdot \mathrm{exhaustividad}_k}{\beta^2 \cdot \mathrm{precision}_k + \mathrm{exhaustividad}_k} = \frac{(1+\beta^2)\mathrm{VP}_k}{(1+\beta^2)\mathrm{VP}_k + \beta^2\mathrm{FP}_k + \mathrm{FN}_k} \tag{2.21}$$

La f-score es una métrica mucho más útil que la tasa de aciertos cuando el conjunto de datos presenta un número de muestras desequilibrado entre clases. Es por esto que, aunque es una métrica más compleja y más difícil de interpretar, normalmente es preferible medir el rendimiento de los modelos en términos de f-score [27].

2.5.4 Métricas de eficiencia

Aunque en un mundo ideal las métricas de rendimiento serían las únicas métricas necesarias para evaluar que modelos son mejores, a la hora de medir el rendimiento de modelos en el mundo real también nos importa minimizar los costes asociados con entrenar y utilizar estos modelos. Para tener los costes en cuenta, existen las métricas de eficiencia, que intentan cuantificar alguno de los costes del modelo y tenerlo en cuenta junto con el rendimiento.

Dado el coste al que puede llegar el entrenamiento y uso de modelos de inteligencia artificial, las métricas de eficiencia pueden ser incluso más importantes que las métricas de rendimiento, especialmente a la hora de comparar arquitecturas con costes muy diferentes.

2.5.4.1 Eficiencia respecto al número de parámetros

A la hora de entrenar modelos de inteligencia artificial, generalmente los modelos con mayor cantidad de parámetros obtienen mejores resultados que los modelos con menor cantidad, especialmente a la hora de realizar tareas complejas [28]. No obstante, el coste y el tiempo de entrenamiento de un modelo aumenta rápidamente con la cantidad de parámetros del mismo, hasta el punto que entrenar un modelo con una gran cantidad de parámetros puede llegar ser prohibitivamente caro y/o lento. Es por esto que es de vital importancia que los modelos utilicen de forma eficiente los parámetros, de forma que maximicen el rendimiento dada una cantidad fija de parámetros.

Para realizar un análisis de eficiencia respecto al número de parámetros hay que tener en cuenta el rendimiento obtenido por los modelos para cierta cantidad de parámetros. Para

esto, generalmente va a ser necesario definir una seria de modelos con un rango amplio de parámetros, y medir el rendimiento obtenido con cada uno de los modelos. Aunque esto sea relativamente fácil de realizar, interpretar los resultados obtenidos puede ser más complejo, ya que es posible que ciertas arquitecturas de modelos no sean siempre mejores que otras, y que estas solo obtengan un mejor rendimiento para cierto rango de parámetros. Además, dependiendo de las circunstancias exactas y del uso planificado del modelo, es posible que se prefiera un modelo peor pero menos costoso o viceversa.

En el apartado 5.3 se ha realizado un experimento que intenta medir la eficiencia respecto al número de parámetros de las redes KAN frente a la de las redes tradicionales.

2.5.4.2 Eficiencia respecto al número de datos de entrenamiento

Otro aspecto a tener en cuenta a la hora de entrenar modelos es la cantidad de datos de entrenamiento requeridos para obtener un buen rendimiento. Como regla general, a mayor complejidad de la tarea que queremos resolver, mayor es la cantidad de datos necesarios para entrenar un modelo con buen rendimiento [29]. No obstante, elaborar conjunto de datos grandes puede ser increíblemente costoso. Además, una vez obtenido el conjunto, se requiere de mayor tiempo de entrenamiento para entrenar con un conjunto de datos más grande, cosa que aumenta el coste de entrenamiento. Es por esto que, a la hora de entrenar modelos de inteligencia artificial, es preferible utilizar arquitecturas y modelos que necesiten de una menor cantidad de datos de entrenamiento para obtener buenos resultados [30].

Para realizar un análisis de eficiencia respecto al número de datos de entrenamiento hay que tener en cuenta el rendimiento obtenido para cierta cantidad de datos de entrenamiento. Esto generalmente se hace entrenando el mismo modelo con varias fracciones de un conjunto de datos predefinido, y midiendo el rendimiento obtenido al entrenar con cada una de las fracciones del conjunto de datos de entrenamiento. Al igual que para la eficiencia respecto al número de parámetros, dependiendo del caso de uso del modelo es posible que sea preferible utilizar un modelo con menor rendimiento que requiera una menor cantidad de datos o viceversa.

En el apartado 5.4 se ha realizado un experimento que intenta medir la eficiencia respecto al número de datos de entrenamiento de las redes KAN frente a la de las redes tradicionales.

2.5.4.3 Eficiencia respecto al tiempo de entrenamiento

Otro de los costes de entrenar modelos de inteligencia artificial es el tiempo que tarda el modelo en ser entrenado. Al igual que con los otros tipos de costes, es preferible que el tiempo necesario para entrenar el modelo sea lo menor posible, con el fin de minimizar el coste necesario para entrenar el modelo [31].

Aunque el tiempo de entrenamiento sea una medida más directa del coste necesario para entrenar modelos, en muchos casos es preferible medir la eficiencia en términos de otros costes, ya que el tiempo de entrenamiento puede variar dependiendo del *hardware* utilizado para el entrenamiento. Es por esto que es muy complicado comparar resultados de distintos experimentos si estos han utilizado sistemas diferentes [32]. No solo eso, si no que en muchos casos ocurre que ciertos modelos van mejor en algunos sistemas que en otros, cosa que hace que los resultados puedan no ser tan informativos como se esperaba, incluso cuando se está entrenando en exactamente el mismo sistema. Además de todo esto, también hay que tener

en cuenta el entorno de entrenamiento, ya que si el equipo en el que se está entrenando el modelo está siendo utilizado para otros programas o tareas es posible que los tiempos varíen por eso.

2.5.5 Calibración

Como ya se ha visto en el apartado 2.5.2, además de la clase predecida podemos obtener la confianza que tiene el modelo en que la respuesta correcta es esa clase. Utilizando ese número, es posible (en teoría) distinguir los casos en los que el modelo está seguro que la salida es de la clase predecida, de los casos en los que el modelo no tenga tanta certeza. No obstante, aunque la confianza es un valor entre 0 y 1, no podemos necesariamente interpretarlo como la probabilidad de que la clase sea correcta. Que el modelo tenga confianza del 95% no significa que la respuesta que ha dado sea la correcta en un 95% de los casos [33].

Para medir el grado de discrepancia entre la confianza del modelo y la probabilidad de que la muestra predecida sea correcta, tenemos que estudiar la calibración del modelo. Modelos bien calibrados tendrán valores parecidos para la confianza y la probabilidad de que la muestra sea correcta, mientras que modelos mal calibrados no [34]. Existen varias formas de medir la calibración de los modelos. En los siguientes apartados, explicamos dos de las más comunes. En el apartado 5.5 se ha realizado un análisis de la calibración de varios modelos, utilizando estas dos métricas que veremos a continuación.

2.5.5.1 Rendimiento por intervalo de confianza

Una manera visual de medir la calibración de un modelo es utilizando *Calibration Plots*. Estos gráficos se basan en agrupar las muestras de evaluación en función de la confianza predecida por el modelo y medir la calibración de cada uno de los grupos [35]. Existen varias estrategias para dividir las muestras en base a la confianza, aunque la más común es hacer una división uniforme respecto a la confianza de las muestras, en la que cada grupo de muestras representa una fracción uniforme del intervalo $[0, 1]$. Por ejemplo, para $N = 5$ grupos, tendríamos en un grupo todas las muestras con confianza $c \in [0, 0.2)$, en otro todas las muestras con confianza $c \in [0.2, 0.4)$, etc. La fórmula general para encontrar el grupo G de una muestra con confianza c en una división uniforme en N grupos es

$$G(x) = \lfloor c(x) \cdot N \rfloor, \tag{2.22}$$

de forma que las muestras están divididas en los grupos $G_0, \ldots, G_{N-1}$. Una vez agrupadas todas las muestras, se calcula para cada grupo la confianza media y la tasa de aciertos media. Si el modelo está bien calibrado estos dos valores serán cercanos, y viceversa [34]. También se suele calcular la cantidad de muestras en cada grupo, ya que un grupo con menor cantidad de muestras es menos relevante que un grupo con mayor cantidad. Podemos definir la confianza media $\text{conf}(G_i)$ y la tasa de aciertos media $\text{acc}(G_i)$ de un grupo de muestras G_i de la siguiente forma:

$$\begin{aligned} \text{conf}(G_i) &= \frac{1}{|G_i|} \sum_{x \in G_i} c(x) \\ \text{acc}(G_i) &= \frac{|x \in G_i : y_x = \hat{y}_x|}{|G_i|} \end{aligned} \tag{2.23}$$

Los valores de confianza y tasa de aciertos media para cada grupo se suelen representar en un gráfico de líneas, que muestra la confianza media de cada grupo respecto a la tasa de aciertos media. Estos gráficos se conocen como curvas de probabilidad de calibración o *Probability Calibration Curves*, y se utilizan para medir de forma visual la calibración de un modelo de inteligencia artificial.

2.5.5.2 Error de calibración esperado

Aunque la técnica descrita anteriormente se puede utilizar para discernir visualmente la calibración de un modelo, no es fácil de utilizar para comparar la calibración de varios modelos. Como tenemos 3 cantidades para cada grupo de muestras (confianza media, tasa de aciertos media y cantidad de muestras), realizar un análisis utilizando estos números puede ser complejo. No obstante, existe otra métrica que combina toda esta información con el fin de proporcionar un único valor con el que podemos medir la calibración del modelo. Esta métrica, conocida como el error de calibración esperado (ECE) o *Expected Calibration Error*, mide la diferencia esperada entre la confianza y la tasa de aciertos de cada grupo de muestras G_i, ponderándolo respecto al número de muestras de cada grupo [34]. Se define con la siguiente fórmula, siendo n la cantidad total de muestras, G_i el grupo i-ésimo de muestras, conf() la confianza media y acc() la tasa de aciertos media:

$$\text{ECE} = \frac{1}{n} \sum_i |G_i| \cdot |\text{acc}(G_i) - \text{conf}(G_i)| \tag{2.24}$$

Como el ECE mide el error de calibración, un menor ECE significa una mejor calibración de un modelo, y viceversa. Aunque el ECE es una métrica mucho más simple para comparar la calibración de los modelos, no nos dice nada de como se varía la confianza y la tasa de errores en toda la distribución de muestras. Es por esto que, aunque en los análisis de calibración el ECE sea la métrica principal, las curvas de calibración se siguen utilizando para poder ver más detalladamente como se comporta un modelo [33].

2.5.6 Aprendizaje continuo y el olvido catastrófico

El aprendizaje continuo es la capacidad de aprender dada nueva información sin olvidar la información ya aprendida. Esta es una habilidad fundamental para cualquier sistema que opere en un entorno con información dinámica. No obstante, las redes neuronales artificiales y la gran mayoría de modelos de inteligencia artificial actuales no tienen esta capacidad [36]. Es más, al entrenar estos modelos con información nueva, estos suelen casi inmediatamente reemplazar los patrones aprendidos al entrenar con la información anterior por la información nueva, "olvidando" toda la información aprendida previamente. Este fenómeno se conoce como el olvido catastrófico [37].

Como consecuencia de este comportamiento, los modelos actuales de inteligencia artificial se tienen que entrenar con toda la información necesaria de golpe, sin poder realizar un entrenamiento incremental. Este problema hace que modificar modelos grandes de inteligencia artificial es muy costoso, incluso si es solo para introducir una pequeña cantidad de información nueva, ya que es necesario entrenar el modelo también con toda la información ya aprendida, en vez de únicamente con la nueva información.

Para medir la capacidad que tiene un modelo para realizar aprendizaje continuo, especialmente en tareas de clasificación, se suele utilizar un entrenamiento por fases. En cada una de las fases se entrena el modelo con un subconjunto de las clases del conjunto de entrenamiento. Después de cada fase, se mide el rendimiento del modelo no solo en las clases entrenadas en esa fase, si no también en las clases entrenadas en fases anteriores. De esta forma, podemos medir la retención de información anterior del modelo, y a partir de ahí medir como de bueno es el modelo a la hora de mantener la información anterior integrando la información nueva [38]. Podemos ver un análisis que intenta medir la capacidad de varios modelos para realizar tareas de aprendizaje continuo en la sección 5.6, donde se realiza un entrenamiento por fases similar al que se ha descrito.

3 Redes Kolmogórov-Arnold

Las redes Kolmogórov-Arnold, o KAN (*Kolmogórov-Arnold Networks*), son una nueva alternativa a las redes neuronales tradicionales. Este tipo de red, como se estudia en los apartados siguientes, están basadas en una estructura y unos principios radicalmente diferentes que los encontrados en las redes tradicionales actuales.

En las redes tradicionales, que a partir de ahora llamaremos redes MLP (*Multi-Layer Perceptron*), cada una de las capas lineales seguidas por funciones de activación no lineales. Las capas lineales de la red contienen pesos entrenables, mientras que las funciones de activación no lineales son fijas (no entrenables). A partir de esta combinación de elementos (transformaciones lineales entrenables y funciones no-lineales fijas), una red tradicional es capaz de aproximar casi cualquier función matemática [10].

Las redes KAN, no obstante, difieren significativamente de este planteamiento. En vez de combinar capas lineales entrenables y funciones de activación no entrenables, las redes Kolmogorov-Arnold permiten el entrenamiento directo de transformaciones no lineales entrenables. Gracias a esta combinación de las propiedades de las capas lineales y de las funciones de activación en uno, las redes KAN no necesitan utilizar funciones de activación ni muchos de los otros mecanismos típicos de las redes neuronales tradicionales, ya que son capaces de representar cualquier función matemática únicamente utilizando capas KAN.

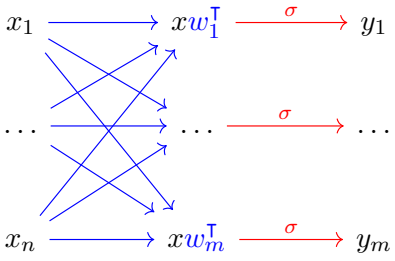
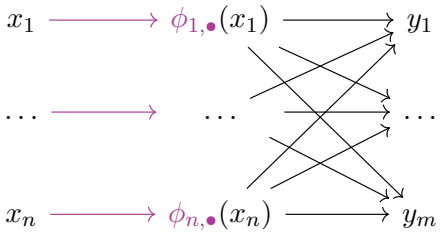
Red MLP	Red KAN
$y = (\sigma_n \circ W_n \circ \cdots \circ \sigma_2 \circ W_2 \circ \sigma_1 \circ W_1)(x)$	$y = (\Phi_n \circ \cdots \circ \Phi_2 \circ \Phi_1)(x)$
	

Tabla 3.1: Comparación de la estructura de las redes MLP con las redes KAN, mostrando las funciones no-lineales en azul, los parámetros entrenables lineales en rojo, y los parámetros entrenables no-lineales en morado. Fuente: elaboración propia

A diferencia de las redes tradicionales, que están basadas en el teorema de aproximación universal [4], la teoría matemática detrás de las redes KAN está basada en el teorema de representación de Kolmogórov-Arnold [5]. Es a partir de la generalización de este teorema que se obtienen las fórmulas que definen la estructura interna de las redes KAN.

Como se estudia más adelante, es posible utilizar muchas estructuras matemáticas para

implementar el componente principal de la estructura interna de las redes KAN. Cada una de estas estructuras tiene sus ventajas y desventajas, por lo que puede resultar complejo elegir la estructura óptima a la hora de diseñar un modelo que utilice redes Kolmogórov-Arnold.

3.1 Teorema de representación

Las redes KAN están basadas en el teorema de representación de Kolmogórov-Arnold [5]. Este teorema establece que cualquier función continua de múltiples variables se puede construir a partir de sumas de funciones de una variable, siempre que todas sus entradas estén acotadas en un rango finito. Esto significa que podemos representar cualquier $f : [a_k, b_k]^n \mapsto \mathbb{R}$ a partir de sumas y funciones de una variable. Específicamente, podemos re-escribir f de la siguiente forma, siendo $g_i : \mathbb{R} \mapsto \mathbb{R}$ y $h_{i,j} : [a_i, b_i] \mapsto \mathbb{R}$:

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{2n} g_i \left(\sum_{j=1}^{n} h_{i,j}(x_j) \right) \tag{3.1}$$

Como se puede ver en la ecuación 3.1, podemos representar cualquier función de n variables a partir de $2(n+1)$ funciones g_i y $(2n+1)\times n$ funciones $h_{i,j}$. Aunque pueda parecer sorprendente, este teorema se puede aplicar a cualquier función continua cuyas entradas estén acotadas, por lo que siempre tiene que existir para este tipo de funciones una forma de descomponerlas en funciones de una única variable. Analizando en detalle la fórmula 3.1, la función está dividida en dos capas, la primera formada por los resultados de las funciones $h_{i,j}$ y la segunda formada por las funciones g_i:

$$f(x_1, \ldots, x_n) = \underbrace{\sum_{i=0}^{2n} g_i}_{2^{\text{a}}\text{ capa}} \underbrace{\left(\sum_{j=1}^{n} h_{i,j}(x_j) \right)}_{1^{\text{a}}\text{ capa}} \tag{3.2}$$

La arquitectura KAN, como veremos en los siguientes apartados, es una generalización de esta representación de funciones de varias variables como sumas de funciones de una variable.

3.2 Capas KAN

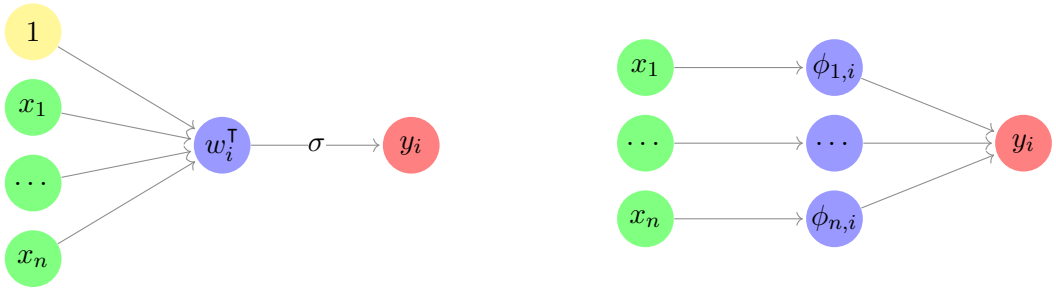
Las redes KAN, al igual que las redes neuronales tradicionales, también se organizan en capas, con el fin de poder diseñar, entrenar y utilizar redes con una gran cantidad de parámetros sin tener que especificar todo manualmente. De forma similar a las redes MLP, además, las redes KAN también normalmente comparten la configuración para todas las neuronas de una capa de la red, de forma que se compartan de forma similar y se conectan de la misma forma con otras capas de la red.

No obstante, a pesar de todas estas similitudes la estructura interna de una capa KAN es muy diferente a las de las redes tradicionales. Mientras que las redes tradicionales están formadas por matrices de pesos lineales, las capas KAN se organizan como sumas de funciones de una variable, de forma similar a las capas vistas en la fórmula del teorema de representación de Kolmogórov-Arnold (ver ecuación 3.2).

Al estar formadas por capas, las capas utilizadas por las redes KAN son compatibles con las capas de las redes tradicionales. De esta forma, se pueden construir modelos mixtos que utilicen capas KAN y capas tradicionales a la vez.

3.2.1 Capas densas

Las capas densas de las redes KAN, como ya se ha mencionado anteriormente, están directamente basadas en las capas de funciones del teorema de representación de Kolmogórov-Arnold. De esta forma, cada salida de una capa KAN densa está formada por una suma de funciones de una variable de todas las entradas de la capa.



(a) Cálculo de una salida de una capa densa MLP **(b)** Cálculo de una salida de una capa densa KAN

Figura 3.1: Comparación del cálculo de una salida de una capa densa MLP (a) con el de una capa densa KAN (b). Hágase notar que en la capa densa MLP la no linearidad (σ) se procesa después de la suma, mientras que en la capa KAN las no linealidades ($\phi_{1,i}, \ldots, \phi_{n,i}$) se procesan antes de la suma. Fuente: elaboración propia

Para una capa KAN de n entradas y m salidas, necesitaremos $n \times m$ funciones de una variable para construir una capa densa KAN. Sean $\phi_{i,j} : \mathbb{R} \mapsto \mathbb{R}$ las funciones de una variable de la capa, tenemos la siguiente fórmula para calcular las salidas de la capa, de forma que la función $\phi_{i,j}$ recibe la entrada x_i y afecta a la salida y_j:

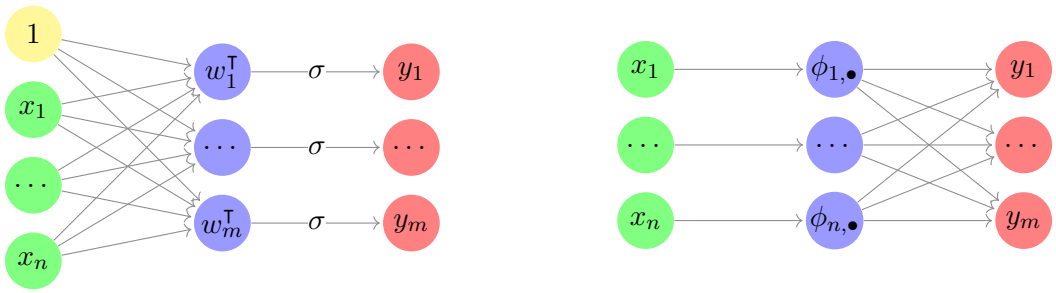
$$y_j = \sum_i \phi_{i,j}(x_i) \tag{3.3}$$

De forma similar a las capas tradicionales, también podemos representar la capa mediante el uso de matrices, utilizando una matriz de funciones $\Phi : \mathbb{R}^n \mapsto \mathbb{R}^m$ que contiene todas las funciones $\phi_{i,j}$ de la capa:

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \underbrace{\begin{bmatrix} \phi_{1,1}(\cdot) & \cdots & \phi_{n,1}(\cdot) \\ \vdots & \ddots & \vdots \\ \phi_{1,m}(\cdot) & \cdots & \phi_{n,m}(\cdot) \end{bmatrix}}_{\Phi} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \tag{3.4}$$

A partir de la fórmula 3.4, podemos ver que la estructura de las capas densas KAN es similar a la estructura de las capas densas tradicionales (ver ecuación 2.5), solo que sustituyendo cada uno de los pesos encontrados en la capa densa tradicional por una función ϕ. Además, cabe destacar que las capas densas KAN no tienen un término de *bias*, ya que el sesgo está

contenido en las funciones $\phi_{i,j}$ de la capa, como se puede observar en la figuras 3.1 y 3.2.



(a) Capa densa MLP **(b)** Capa densa KAN

Figura 3.2: Comparación de la estructura de una capa densa de una red MLP (a) con una capa densa de una red KAN (b). En las capas densas MLP primero se suma y luego se aplica la función no lineal (σ), mientras que en las KAN primero se aplican funciones no lineales a todas las entradas ($\phi_{1,\bullet}, \ldots, \phi_{n,\bullet}$) y después se suman los resultados obtenidos. Fuente: elaboración propia

Una capa densa KAN de n entradas y m salidas, si cada una de las funciones de la capa tienen k parámetros entrenables (ver apartado 3.3), tiene $O(n\,m\,k)$ parámetros entrenables. Esto, aunque en teoría es bastante mayor que los $O(n\,m)$ parámetros entrenables requeridos para entrenar una capa densa MLP, en la práctica no es tan problemático, ya que gracias al mayor poder representativo de las redes KAN se pueden reducir las neuronas de una capa significativamente manteniendo un nivel de rendimiento similar al de una capa MLP con mayor cantidad de parámetros [6].

3.2.2 Capas convolucionales

Como ya se ha analizado y estudiado en el apartado 2.3.2, las capas convolucionales permiten a los modelos de machine learning aprender de forma mucho más efectiva y eficiente los patrones internos de datos estructurados, como imágenes o ficheros de audio. Basándonos en la división de funciones en capas de funciones de una variable presentada por el teorema de representación de Kolmogórov-Arnold (ver ecuación 3.2), es posible generalizar la estructura de las convoluciones tradicionales para que estén basadas en sumas de funciones de una variable. De esta forma, obtenemos las convoluciones KAN [39].

Las convoluciones KAN, matemáticamente, son similares a las convoluciones discretas tradicionales, solo que se ha sustituido el kernel lineal de las convoluciones tradicionales por un kernel formado por funciones no lineales de una variable. De esta forma, cada una de las salidas de la convolución se obtiene a partir de una suma de las salidas de las funciones del kernel, manteniendo la estructura de las capas convolucionales MLP a la vez que implementa la suma de funciones del teorema de representación [40]. En la figura 3.3 se puede ver una representación visual de una operación de convolución KAN de dos dimensiones.

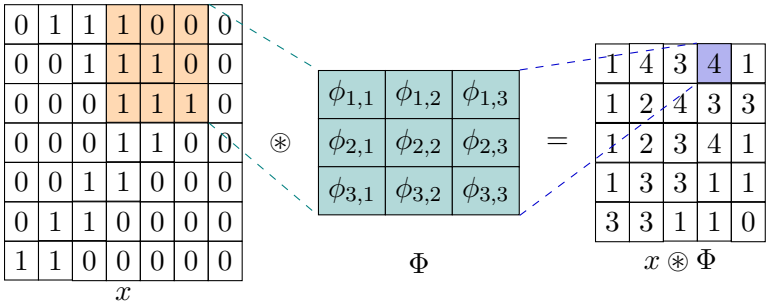


Figura 3.3: Representación de una convolución KAN 2D. Fuente: elaboración propia, basada en [16]

Al igual que para las capas convolucionales tradicionales, las dimensiones del kernel de una capa convolucional KAN dependen de la cantidad de dimensiones de los datos de entrada de la capa (ver apartado 2.3.2). Para datos 2D, siguiendo el ejemplo de las redes convolucionales tradicionales establecido en la ecuación 2.7, siendo $x \in \mathbb{R}^{n \times m \times s}$ la entradas de la capa y $\phi^k : \mathbb{R} \mapsto \mathbb{R}^{p \times q \times s}$ los kernels de la capa, podemos definir el comportamiento de la capa a partir de las convoluciones KAN (denotadas como $x \circledast y$) de la siguiente forma:

$$y_{i,j,k} = (x \circledast \phi^k)_{i,j} = \sum_{a,b,c} \phi^k_{a,b,c} \left(x_{i+a,j+b,c}\right) \tag{3.5}$$

De forma similar a las capas densas KAN, las capas convolucionales KAN también tienen una cantidad de parámetros superior que las capas convolucionales MLP. Para una capa convolucional con kernels de tamaño $p \times q \times s$, teniendo cada una de las funciones de los kernels k parámetros entrenables, tenemos $O(p\,q\,s\,k)$ parámetros entrenables, comparados con los $O(p\,q\,s)$ de las capas convolucionales tradicionales.

3.3 Funciones KAN

Las capas KAN, ya sean densas o convolucionales, están basadas en las suma de funciones de una variable, tal y como se especifica en el teorema de representación de Kolmogórov-Arnold (ver apartado 3.1). Para poder construir las capas KAN en acorde con el teorema de representación, es necesario idear un método de construcción de funciones cuyas funciones resultantes sean capaces de aproximar cualquier función de una variable. En otras palabras, necesitamos una serie de fórmulas matemáticas que nos permitan aproximar cualquier función $f : \mathbb{R} \mapsto \mathbb{R}$.

Este método, no obstante, deberá además satisfacer las siguientes propiedades para que su uso en las redes KAN sea ideal:

- Las funciones deberían tener una cantidad variable de parámetros, de forma que a mayor cantidad de parámetros mayor sea el detalle de la función producida. Esto permite variar la cantidad de parámetros dependiendo de la complejidad de las funciones de una variable a aprender, añadiendo flexibilidad y permitiendo la implementación de ciertas técnicas avanzadas a la hora de entrenar modelos KAN (ver apartado 3.3.1.3)

- Las funciones producidas deberían ser continuas y derivables en todo su dominio, con el fin de poder hacer uso de métodos de entrenamiento basados en el cálculo de gradientes y backpropagation (ver apartado 3.4.1)
- Las funciones deberían poder aproximar cualquier función de una variable dados los suficientes parámetros. Si el método de aproximación de funciones de una variable no es capaz de aproximar cualquier función, es posible que la red no sea capaz de aproximar cualquier resultado, y que eso cree limitaciones o dificultades a la hora de entrenar la red [41].
- La evaluación de estas funciones debería de ser rápida y eficiente, con el fin de minimizar el coste y tiempo necesario para entrenar y utilizar la red KAN producida.
- La cantidad de parámetros entrenables necesarios para poder representar estas funciones debería ser lo más baja posible, con el fin de minimizar el tamaño de los modelos obtenidos y, además, de prevenir ciertos problemas a la hora de entrenar el modelo [7].

Aunque hay muchos métodos y técnicas matemáticas que satisfacen estas propiedades, en las redes KAN normalmente se utilizan splines para implementar las funciones entrenables de una variable [6, 42].

3.3.1 Splines

Una spline, matemáticamente, es una función continua y derivable construida por varios trozos polinómicos. El orden k de la spline determina el orden de los polinomios que la constituyen. Estos trozos se controlan por los nodos t_i de la spline, que marcan a partir de que valor acaba un trozo y empieza el siguiente. Los nodos de una spline tienen que cumplir que $t_0 \leq t_1 \leq \cdots \leq t_{m-1}$. Una spline solo puede tener valores distintos de 0 en el intervalo $[t_0, t_{m-1}]$.

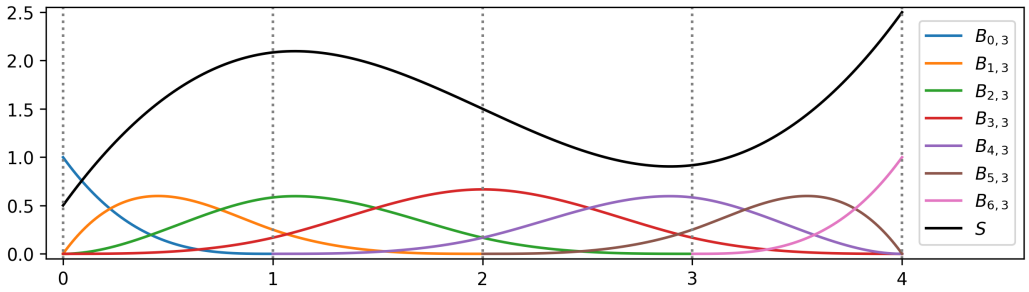


Figura 3.4: Ejemplo de una spline uniforme en el que se han representado las funciones base de orden 3 de la spline ($B_{0,3}, \ldots, B_{6,3}$; en varios colores), junto a la spline resultante al combinar todas estas funciones base (S, en negro), utilizando los valores $\alpha_i = (0.5, 1.5, 2.5, 1.5, 0.5, 1.5, 2.5, 1.5, 0.5, 1.5)$ para combinar las funciones base. La spline tiene nodos en $0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4$ y es de orden 3. Fuente: elaboración propia

Cada una de las funciones base depende únicamente de los nodos t_i de la spline, y solo produce valores en el intervalo $[0, 1]$. Una vez establecidos los nodos de la spline (y por lo

tanto también las funciones base), podemos aproximar cualquier función como una suma ponderada de funciones base [41]. Por lo tanto, si $B_{i,k} : \mathbb{R} \mapsto [0, 1]$ son las funciones base de orden k para unos nodos t_i, y α_i es valor por el que multiplicamos $B_{i,k}$ en la suma ponderada, podemos calcular la salida de la spline $S : \mathbb{R} \mapsto \mathbb{R}$ con la siguiente fórmula:

$$S(x) = \sum_i \alpha_i B_{i,k}(x) \tag{3.6}$$

Cabe destacar que, para aproximar cualquier función de una variable utilizando splines, únicamente es necesario variar los parámetros α_i. Los parámetros $B_{i,k}$ o t_i no son necesarios para poder aproximar cualquier función utilizando splines [41]. Un ejemplo de esto se puede ver en la figura 3.5, en la que se ha utilizado una spline para aproximar la función $\operatorname{sinc} x = \frac{\sin x}{x}$ obteniendo los parámetros α_i correspondientes a partir de unos t_i dados.

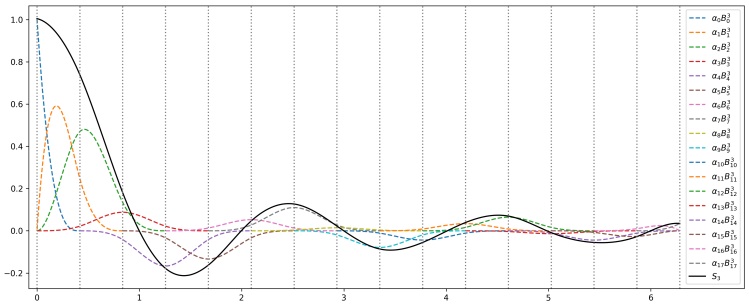


Figura 3.5: Spline uniforme de orden 3 aproximando $\operatorname{sinc} x$ en el intervalo $[0, 2\pi]$, representando la spline final junto con todas las funciones base $(B_{0,3}, \ldots, B_{17,3})$ multiplicadas por su valor correspondientes $(\alpha_0, \ldots, \alpha_{17})$. Los valores de los nodos, distribuidos uniformemente en el intervalo $[0, 2\pi]$, se han representado como líneas verticales grises. Fuente: elaboración propia

Aunque las splines requieran muchos parámetros y cálculos para definirse completamente, ya que hay que definir t_i y calcular todas las $B_{i,j}$ correspondientes a partir de estos valores, realizar aproximación de funciones una vez obtenidas las funciones base $B_{i,j}$ es relativamente simple, ya que únicamente es necesario variar los parámetros α_i, dejando todos los otros términos fijos. En las implementaciones KAN eficientes, esta propiedad se explota para calcular todas las $B_{i,j}$ primero, y luego reutilizando los valores obtenidos con todos los parámetros α_i diferentes de todas las splines de la red [43].

3.3.1.1 Funciones base

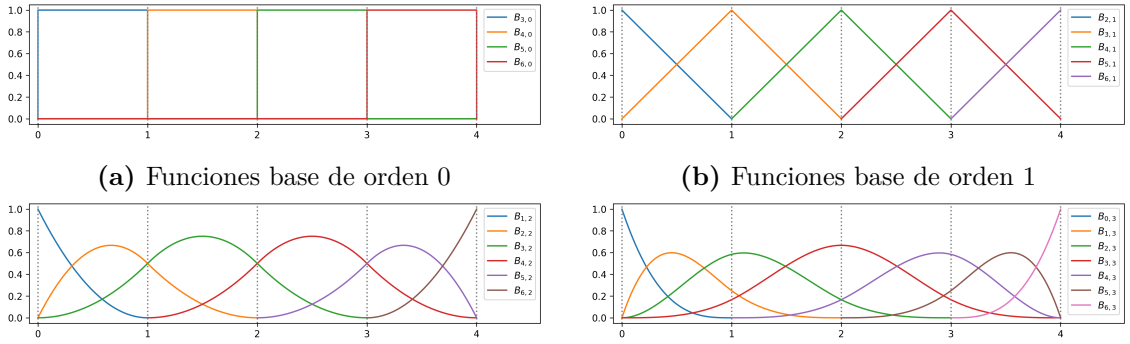
A partir de los nodos de una spline, podemos definir las funciones base de la misma. Una spline de orden k tiene funciones base de orden $0, 1, \ldots, k$. Dependiendo del orden, las funciones base de una spline se comportan de forma diferente:

- Funciones base de orden 0: son funciones que devuelven 1 si x están entre los nodos t_i y t_{i+1}, o 0 si no.
- Funciones base de orden 1: interpolan linealmente entre dos funciones base de orden 0,

en el rango $[t_i, t_{i+2}]$

- Funciones base de orden 2: interpolan cuadráticamente entre dos funciones base de orden 1, en el rango $[t_i, t_{i+3}]$
- Funciones base de orden k: realizan una interpolación de orden k entre dos funciones base de orden $k-1$, en el rango $[t_i, t_{i+k+1}]$

Se puede ver una representación visual de las funciones base de orden 0, 1, 2 y 3 de una spline en la figura 3.6.



(a) Funciones base de orden 0 **(b)** Funciones base de orden 1

(c) Funciones base de orden 2 **(d)** Funciones base de orden 3

Figura 3.6: Funciones base de orden 0 (a), orden 1 (b), orden 2 (c) y orden 3 (d) para una spline uniforme de orden 3, con nodos $0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4$. Fuente: elaboración propia

Matemáticamente, las funciones base de una spline se definen utilizando las fórmulas de Cox-de Boor [44]. Esta fórmula define en el caso base las funciones base de orden 0, y a partir de ahí define recursivamente las funciones base de orden superior. Sean t_i los nodos de la spline, y teniendo $t_0 \leq t_1 \leq \cdots \leq t_{m-1}$ podríamos definir las funciones base de la siguiente forma:

$$\begin{aligned} B_{i,0}(x) &= \begin{cases} 1 & \text{si} \quad t_i \leq x < t_{i+1} \\ 0 & \text{en otro caso} \end{cases} \\ B_{i,k}(x) &= \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x) \end{aligned} \tag{3.7}$$

Aunque esta es la forma en la que se suele presentar la fórmula de Cox-de Boor, esta es una simplificación, ya que no tiene en cuenta que es posible que dos nodos adyacentes coincidan ($t_i = t_{i+1}$ para algún valor i). En este caso, si se estudia en detalle el caso recursivo de la fórmula de Cox-de Boor, se estaría realizando una división por 0 a la hora de calcular alguna de las funciones base de orden 1. Además, si llegasen a coincidir varios nodos seguidos, también se estaría efectuando esta división por cero a la hora de calcular ciertas funciones base de órdenes superiores.

Para que la fórmula de Cox-de Boor funcione correctamente incluso en estos casos, se añade la condición de que, cuando se realice una división por cero en la fórmula Cox-de Boor, hay

que considerar que el resultado de la división es 0. En otras palabras, hay que considerar $a/0 = 0$ para que la fórmula 3.7 funcione correctamente en todos los casos.

Si se incorpora este comportamiento ($a/0 = 0$) a la ecuación de Cox-de Boor en sí, se obtienen las siguientes ecuaciones:

$$\begin{aligned} B_{i,0}(x) &= \begin{cases} 1 & \text{si} \quad t_i \leq x < t_{i+1} \\ 0 & \text{en otro caso} \end{cases} \\ b_{i,k,1}(x) &= \begin{cases} \frac{x-t_i}{t_{i+k}-t_i} B_{i,k-1}(x) & \text{si} \quad t_{i+k} \neq t_i \\ 0 & \text{en otro caso} \end{cases} \\ b_{i,k,2}(x) &= \begin{cases} \frac{t_{i+k+1}-x}{t_{i+k+1}-t_{i+1}} B_{i+1,k-1}(x) & \text{si} \quad t_{i+k+1} \neq t_{i+1} \\ 0 & \text{en otro caso} \end{cases} \\ B_{i,k}(x) &= b_{i,k,1}(x) + b_{i,k,2}(x) \end{aligned} \tag{3.8}$$

Aunque todo esto pueda parecer un detalle menor, resulta que en el contexto de las redes KAN es una distinción bastante importante, ya que como veremos en el apartado 3.3.1.3 las splines más utilizadas en las redes KAN suelen tener varios nodos que coinciden.

3.3.1.2 Derivadas

Como ya se ha visto anteriormente, para entrenar modelos de inteligencia artificial es casi siempre necesario calcular las derivadas respecto al error de la salida producido por el modelo. Las derivadas de las splines pueden ser calculados a partir de la fórmula Cox-de Boor [45]. Las derivadas obtenidas a partir de derivar la fórmula 3.7 se pueden ver a continuación:

$$\begin{aligned} S'(x) &= \sum_i \alpha_i B'_{i,k}(x) \\ B'_{i,k}(x) &= \frac{k}{t_{i+k} - t_i} B_{i,k-1}(x) - \frac{k}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x) \end{aligned} \tag{3.9}$$

Al igual que para la fórmula de Cox-de Boor, para que la fórmula de las derivadas funcione cuando dos nodos coinciden hay que considerar que $a/0 = 0$. Teniendo esto en cuenta, la fórmulas para calcular las derivadas de las funciones base de una spline son las siguientes:

$$\begin{aligned} B'_{i,k}(x) &= b'_{i,k,1}(x) - b'_{i,k,1}(x) \\ b'_{i,k,1}(x) &= \begin{cases} \frac{k}{t_{i+k}-t_i} B_{i,k-1}(x) & \text{si} \quad t_{i+k} \neq t_i \\ 0 & \text{en otro caso} \end{cases} \\ b'_{i,k,2}(x) &= \begin{cases} \frac{k}{t_{i+k+1}-t_{i+1}} B_{i+1,k-1}(x) & \text{si} \quad t_{i+k+1} \neq t_{i+1} \\ 0 & \text{en otro caso} \end{cases} \end{aligned} \tag{3.10}$$

3.3.1.3 Splines uniformes y grid intervals

En el contexto de las redes KAN, se suelen utilizar splines uniformes para representar las funciones de una variable, ya que tienen muchas propiedades deseables, como que mantienen la continuidad para todas las derivadas en todo su dominio [46] o que son relativamente eficientes de calcular.

Para hacer que una spline sea uniforme, simplemente es necesario repetir el primer y último nodo k veces, siendo k el orden de la spline. De esta forma, tendríamos los siguientes nodos:

$$\mathbf{t} = (\underbrace{t_0, t_1, \ldots, t_{k-1}}_{\text{nodos iniciales}}, \underbrace{t_k, \ldots, t_{m-k-1}}_{\text{nodos centrales}}, \underbrace{t_{m-k}, \ldots, t_{m-1}}_{\text{nodos finales}}), \tag{3.11}$$

En la fórmula 3.11, todos los nodos iniciales iguales al primer nodo central ($t_0 = t_1 = \cdots = t_k$) y todos los nodos finales son iguales al último nodo central ($t_{m-k-1} = t_{t-k} = \cdots = t_{m-1}$). Por lo tanto, si utilizamos splines uniformes, los únicos valores necesarios para definir los nodos de la spline son los nodos centrales ($t_k, \ldots, t_{m-k-1}$).

En el contexto de las KAN, a los nodos centrales en las splines uniformes se le suele llamar la *grid* de la spline. Introduciremos a partir de aquí la notación g_i para representar la grid, teniendo $\mathbf{g} = (g_0, \ldots, g_{m-2k-1}) = (t_{k+1}, \ldots, t_{2k})$. El vector de nodos $\mathbf{t}$, expresado utilizando esta notación, sería el siguiente:

$$\mathbf{t} = (\underbrace{g_0, g_0, \ldots, g_0}_{\text{nodos iniciales}}, \underbrace{g_0, g_1, \ldots, g_{m-2k-1}}_{\text{nodos centrales (grid)}}, \underbrace{g_{m-2k-1}, \ldots, g_{m-2k-1}}_{\text{nodos finales}}) \tag{3.12}$$

Por razones de eficiencia, los nodos de la grid se suelen distribuir uniformemente en todo el rango de la grid [43]. Por lo tanto, tenemos dos parámetros para controlar completamente los nodos de las splines:

- Tamaño de la grid (G): la cantidad de nodos de la grid. Puede variar enormemente dependiendo de la complejidad de las funciones que se están intentando aproximar y del tamaño de la red KAN.
- Rango de la grid: intervalo de valores en el que se distribuyen los nodos de la grid. Suele ser $[0, 1]$ o $[-1, 1]$, excepto en las capas de salida.

3.3.1.4 Alternativas

Aunque en las redes Kolmogorov-Arnold normalmente se utilizan splines, tal y como propuso la publicación original [6], se han ido desarrollando varias alternativas adicionales. Muchas de estas alternativas, aunque más complejas matemáticamente, tienen algunas características interesantes que hacen que funcionen mejor en ciertos casos. Las alternativas más importantes a las splines en las redes KAN son las siguientes:

- Polinomios de Chebyshev: basados en la fórmula de recurrencia de Chebyshev [47], esta familia de polinomios también es capaz de aproximar funciones de una variable. Hay varios estudios e implementaciones utilizándolos como alternativa más eficiente a las splines en ciertos casos [48, 49].

- Series de Fourier: utilizan series de Fourier [50] para aproximar las funciones de una variable. Aunque las series de Fourier tengan menos estabilidad numérica en sus extremos [51], tienden a ser más estables en su centro que otros métodos, permitiendo que en ciertos casos funcionen mejor [52, 53].

- Polinomios de Jacobi: basados en la función hipergeométrica y la función gamma [54, 55], estos polinomios son bastante interesantes a la hora de aproximar funciones con gran variabilidad, aunque son bastante más costosos de calcular que otras alternativas [56, 57].

- Wavelets: basados en la transformada ondícula utilizada en el procesamiento de señales [58], los wavelets son capaces de aproximar fácilmente y con relativamente pocos parámetros funciones con una gran variabilidad y que pueden contener con componentes cíclicos [59, 60]. No obstante, pierden precisión a la hora de representar funciones que no varían rápidamente [61].

- Activaciones ReLU: utilizan polinomios de activaciones ReLU y otras combinaciones de activaciones para aproximar funciones de una variable KAN. Aunque no sean tan precisas como otros métodos, son mucho más rápidas que las KAN tradicionales, tienen pocos parámetros y mantienen muchas de las propiedades interesantes de las redes KAN [62, 63].

3.3.2 Función residual

Para implementar las redes KAN es necesario poder aproximar eficientemente funciones de una variable. Esto se puede hacer mediante varios métodos de aproximación de funciones: splines, polinomios de Chebyshev, etc. Sea $f : \mathbb{R} \mapsto \mathbb{R}$ la función obtenida por el aproximador elegido, entonces tendríamos la siguiente definición para las funciones KAN utilizadas por la red:

$$\phi(x) = f(x) \tag{3.13}$$

Esta definición, aunque sencilla, hace que las redes KAN sean muy costosas de entrenar en la práctica [6, 43]. Es por esto que, aunque f ya es capaz de aproximar cualquier función de una variable, se añade otra función adicional a las funciones KAN para mejorar su velocidad de entrenamiento. Esta función es la función residual, que es fija (no entrenable). Además, también se suele añadir un peso para la función residual y otro para la función del aproximador, ambos entrenables. Sea $f : \mathbb{R} \mapsto \mathbb{R}$ la función del aproximador, $w_f \in \mathbb{R}$ su peso, $b : \mathbb{R} \mapsto \mathbb{R}$ la función residual y $w_b \in \mathbb{R}$ su peso, podemos definir una función KAN ϕ de la siguiente forma:

$$\phi(x) = w_b\, b(x) + w_f\, f(x) \tag{3.14}$$

Al utilizar splines como el aproximador de la red KAN tendríamos la siguiente definición para las funciones KAN de la red, siendo w_b, w_f y α_i los parámetros entrenables:

$$\phi(x) = w_b\, b(x) + w_f \sum_i \alpha_i B_{i,k}(x) \tag{3.15}$$

Para las funciones residuales casi siempre se eligen funciones no lineales, ya que juegan un papel similar al de las funciones de activación tradicionales en las primeras fases del entrenamiento de la red [6]. Especialmente, se suele elegir la función SiLU (Sigmoid Linear Unit), con la que se tiene $b(x) = x/(1 + e^{-x})$ en las ecuaciones 3.14 y 3.15.

3.4 Entrenamiento

El entrenamiento de las redes KAN es muy similar al entrenamiento de las redes neuronales tradicionales. Al igual que estas últimas, las redes KAN generalmente se entrenan calculando el error observado en la salida producida por la red, comparando las salidas producidas con las salidas esperadas para ciertos datos de entrada. A partir de este error y mediante el uso de retropropagación (*backpropagation*), es posible obtener los gradientes respecto al error de todos los parámetros de la red. Una vez calculados los gradientes, se utiliza un optimizador para variar los parámetros de la red de forma eficiente, al igual que en las redes neuronales tradicionales (ver el apartado 2.4).

3.4.1 Retropropagación

Al igual que con las redes tradicionales, las gradientes de las redes KAN se calculan mediante retropropagación. No obstante, como la estructura de las capas es diferente, los gradientes de la red también se van a tener que calcular de forma diferente. En una red KAN típica que utilice splines, tal y como se puede ver en la ecuación 3.15, tenemos los pesos entrenables w_b y w_f, junto con los parámetros entrenables de la spline α_k. Para poder entrenar una red KAN, tenemos que calcular las derivadas de estos pesos frente al error E de la red.

El primer paso para poder calcular todo esto es darse cuenta de que cualquiera de estos parámetros únicamente afecta su salida correspondiente de la capa. Por lo tanto, podemos descomponer las derivadas de la siguiente forma, siendo $w^l_{b,i,j}$, $w^l_{f,i,j}$ y $\alpha^l_{k,i,j}$ los parámetros w_b, w_f y α_k de la capa l de la red para la entrada i y la salida j, y x^l_i la entrada i de la capa l:

$$\begin{aligned}
\frac{\partial E}{\partial w^l_{b,i,j}} &= \frac{\partial E}{\partial x^{l+1}_j}\frac{\partial x^{l+1}_j}{\partial w^l_{b,i,j}} \\
\frac{\partial E}{\partial w^l_{f,i,j}} &= \frac{\partial E}{\partial x^{l+1}_j}\frac{\partial x^{l+1}_j}{\partial w^l_{f,i,j}} \\
\frac{\partial E}{\partial \alpha^l_{k,i,j}} &= \frac{\partial E}{\partial x^{l+1}_j}\frac{\partial x^{l+1}_j}{\partial \alpha^l_{k,i,j}}
\end{aligned} \tag{3.16}$$

Al igual que se ha hecho para la sección de backpropagation de las redes neuronales tradicionales, se ha utilizado la notación δ^l para denotar los deltas de la capa l. Los deltas son los valores intermedios que se propagan hacia atrás entre capas (ver apartado 2.4.1 para más detalles), siendo definidos para las redes neuronales tradicionales como la derivada del error respecto a la multiplicación de las entradas y los pesos ($\partial E/\partial w^\intercal x$). No obstante, para las redes KAN necesitamos utilizar una definición alternativa (ya que la matriz de pesos w no

existe), por lo que se suelen definir a partir como la derivada del error respecto a las entradas de la capa ($\partial E/\partial x$). Por lo tanto, tenemos que $\delta_i^l = \partial E/\partial x_i^l$.

Utilizando esta definición, podemos simplificar las fórmulas de la ecuación 3.16 sustituyendo los términos $\partial E/\partial x$ por los deltas correspondientes:

$$\begin{aligned}\frac{\partial E}{\partial w_{b,i,j}^l} &= \delta_j^{l+1}\frac{\partial x_j^{l+1}}{\partial w_{b,i,j}^l}\\ \frac{\partial E}{\partial w_{f,i,j}^l} &= \delta_j^{l+1}\frac{\partial x_j^{l+1}}{\partial w_{f,i,j}^l}\\ \frac{\partial E}{\partial \alpha_{k,i,j}^l} &= \delta_j^{l+1}\frac{\partial x_j^{l+1}}{\partial \alpha_{k,i,j}^l}\end{aligned} \tag{3.17}$$

Aunque la definición es un poco diferente, sigue siendo posible calcular directamente los deltas de la última capa utilizando la función de pérdida. Sea L la cantidad de capas de la red (y por lo tanto el índice de la última capa), y sea E' la derivada del error respecto a las salidas de la última capa, tenemos la siguiente ecuación para calcular δ^L:

$$\delta_i^L = \frac{\partial E}{\partial x_i^L} = E_i' \tag{3.18}$$

Para calcular los deltas de las otras capas de la red, es necesario conocer los deltas de la capa anterior, por lo que tenemos que calcular δ^l a partir de δ^{l+1}. Para esto, primero vamos a obtener la relación entre δ^l y δ^{l+1} a partir de la definición:

$$\delta_i^l = \frac{\partial E}{\partial x_i^l} = \sum_j \left(\frac{\partial E}{\partial x_j^{l+1}}\frac{\partial x^{l+1}}{\partial x_i^l}\right) = \sum_j \left(\delta_j^{l+1}\frac{\partial x_j^{l+1}}{\partial x_i^l}\right) \tag{3.19}$$

Como se puede ver en la ecuación 3.19, para calcular δ^l a partir de δ^{l+1} es necesario calcular el término $\partial x_j^{l+1}/\partial x_i^l$. Este término dependerá del tipo de capa y de las fórmulas empleadas. Para desarrollar el resto de este apartado, a partir de aquí se asume que estamos tratando con una capa densa KAN basada en splines sin ninguna modificación particular, por lo que se obtiene la siguiente fórmula:

$$x_j^{l+1} = \sum_i \phi_{i,j}^l(x_i^l) \tag{3.20}$$

A partir de esto, es posible calcular $\partial x_j^{l+1}/\partial x_i^l$:

$$\frac{\partial x_j^{l+1}}{\partial x_i^l} = \frac{\partial \sum_i \phi_{i,j}^l(x_i^l)}{\partial x_i^l} = \sum_i \frac{\partial \phi_{i,j}^l(x_i^l)}{\partial x_i^l} = \sum_i \phi_{i,j}'^l(x_i^l) \tag{3.21}$$

Para obtener la fórmula final para el cálculo de δ_i, sustituimos el resultado de la ecuación 3.21 en la ecuación 3.19:

$$\delta_i = \sum_j \left(\delta_j^{l+1} \frac{\partial x_j^{l+1}}{\partial x_i^l} \right) = \sum_j \left(\delta_j^{l+1} \sum_i \phi_{i,j}'^l(x_i^l) \right) \tag{3.22}$$

Podemos obtener la derivada de $\phi(x)$ a partir de su definición:

$$\phi_{i,j}'^l(x) = \left(w_{b,i,j}^l b(x_i^l) + w_{f,i,j}^l f_{i,j}(x_i^l) \right)' = w_{b,i,j}^l b'(x_i^l) + w_{f,i,j}^l f_{i,j}'^l(x_i^l) \tag{3.23}$$

A partir de las ecuaciones 3.18 y 3.22, podemos calcular todos los deltas de la red $\delta^1, \ldots, \delta^L$ secuencialmente. Una vez se han calculado los deltas de todas las capas, solo es necesario calcular los términos $\partial x_j^{l+1}/\partial w_{b,i,j}^l$, $\partial x_i^{l+1}/\partial w_{f,i,j}^l$ y $\partial x_i^{l+1}/\partial \alpha_{k,i,j}^l$ para obtener todas las derivadas de todos los parámetros de la red. Podemos calcular la derivada de w_b con un poco de desarrollo:

$$\frac{\partial x_j^{l+1}}{\partial w_{b,i,j}^l} = \frac{\partial \sum_j \phi_{i,j}^l(x_i^l)}{\partial w_{b,i,j}^l} = \frac{\partial \phi_{i,j}^l(x_i^l)}{\partial w_{b,i,j}^l} = \frac{\partial w_{b,i,j}^l b^l(x_i^l) + w_{f,i,j}^l f(x_i^l)}{\partial w_{b,i,j}^l} = b^l(x_i^l) \tag{3.24}$$

De forma muy similar también se puede calcular la derivada de w_f:

$$\frac{\partial x_j^{l+1}}{\partial w_{f,i,j}^l} = \frac{\partial \sum_j \phi_{i,j}^l(x_i^l)}{\partial w_{f,i,j}^l} = \frac{\partial \phi_{i,j}^l(x_i^l)}{\partial w_{f,i,j}^l} = \frac{\partial w_{b,i,j}^l b^l(x_i^l) + w_{f,i,j}^l f_{i,j}^l(x_i^l)}{\partial w_{f,i,j}^l} = f_{i,j}^l(x_i^l) \tag{3.25}$$

Como estamos utilizando una spline como aproximador, podemos calcular las derivadas de los parámetros α_k a partir de la ecuación 3.10. Si estuviésemos utilizando otro aproximador, habría que calcular las derivadas de sus parámetros de la forma que corresponda.

$$\frac{\partial x_j^{l+1}}{\partial \alpha_{k,i,j}^l} = \frac{\partial w_{f,i,j}^l f_{i,j}^l(x_i^l)}{\partial \alpha_{k,i,j}^l} = \frac{\partial w_{f,i,j}^l \sum_k \alpha_{k,i,j}^l B_k^l(x_i^l)}{\partial \alpha_{k,i,j}^l} = \frac{\partial w_{f,i,j}^l \alpha_{k,i,j}^l B_k^l(x_i^l)}{\partial \alpha_{k,i,j}^l} = w_{f,i,j}^l B_k^l(x_i^l) \tag{3.26}$$

Si sustituimos los resultados de las ecuaciones 3.24, 3.25 y 3.26 en la ecuación 3.16 obtenemos las fórmulas para calcular las derivadas respecto al error de todos los parámetros de la red:

$$\begin{aligned} \frac{\partial E}{\partial w_{b,i,j}^l} &= \delta_j^{l+1} b^l(x_i^l) \\ \frac{\partial E}{\partial w_{f,i,j}^l} &= \delta_j^{l+1} f_{i,j}^l(x_i^l) \\ \frac{\partial E}{\partial \alpha_{k,i,j}^l} &= \delta_j^{l+1} w_{f,i,j}^l B_k^l(x_i^l) \end{aligned} \tag{3.27}$$

Utilizando estas fórmulas, podemos aplicar backpropagation de la misma forma que para las redes neuronales tradicionales, y por lo tanto obtener todas las derivadas respecto al error de todos los parámetros de cualquier red KAN.

3.4.2 Grid extension

Una propiedad interesante de muchos de los aproximadores de funciones de una variable utilizados en las redes KAN (splines, polinomios de Chebyshev, etc.) es que se pueden sustituir fácilmente por aproximadores con una mayor cantidad de parámetros sin variar prácticamente la función producida por el aproximador. Por ejemplo, en el caso de estar utilizando splines, hay que sustituir la spline por otra que tenga una grid con mayor cantidad de nodos, y calcular para esta última los parámetros necesarios para que la función producida sea lo más parecida posible a la producida por la spline original.

Gracias a esta técnica, es posible empezar entrenando la red con pocos parámetros en cada aproximador e ir aumentando gradualmente la cantidad de parámetros a la que avanza el entrenamiento. A todo este proceso se conoce como *grid extension*, o extensión de la grid. Se puede ver una representación visual del proceso de grid extension aplicado a splines en la figura 3.7.

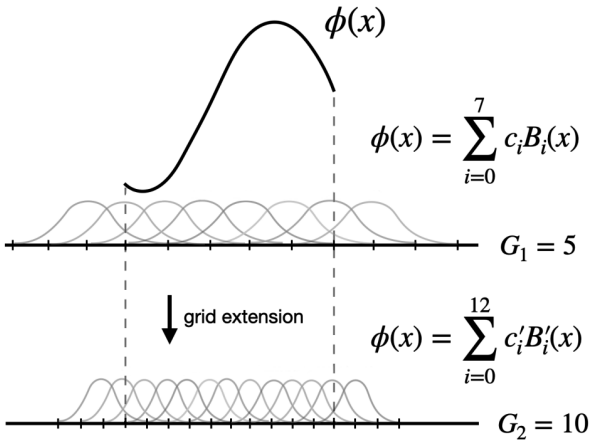


Figura 3.7: Representación visual del proceso de grid extension, mostrando la transformación de una spline con G (tamaño de la grid) $= 5$ a una con $G = 10$. La spline original tiene 7 funciones base, mientras que la spline expandida tiene 12 funciones base. Fuente: [6]

Al utilizar grid extension es posible entrenar la red con pocos parámetros en las primeras fases del entrenamiento, cosa que hace que aumente mucho la velocidad de entrenamiento pero reduce el detalle que es capaz de representar el modelo. A partir de ahí, una vez el modelo ya ha optimizado la estructura general de la función objetivo, se va aumentando poco a poco la cantidad de parámetros, que tiene el efecto de aumentar la cantidad de detalle que es capaz de representar el modelo. Progresivamente, la precisión obtenida por el modelo sigue aumentando, hasta alcanzar la cantidad óptima de detalle para la función que está siendo optimizada. Seguir añadiendo parámetros después de alcanzar la cantidad óptima hace que el modelo tenga parámetros innecesarios y que en ciertos casos se puedan obtener peores resultados que un modelo con la cantidad de parámetros óptima [7].

Al medir regularmente el rendimiento del modelo, es posible detectar la cantidad de parámetros óptima para maximizar el rendimiento del modelo sin continuar aumentando innecesariamente la cantidad de parámetros del mismo. Esto hace que el modelo final tras aplicar grid extension durante el entrenamiento tenga un mayor rendimiento y una menor cantidad de

parámetros, además de reducir también la cantidad de recursos dedicados al entrenamiento en las primeras fases.

El procedimiento de grid extension se puede implementar de muchas formas. La más sencilla y la utilizada en el artículo original que propuso las redes KAN es aumentar la grid en ciertas épocas, de forma que la red tenga suficiente tiempo entre extensiones para acercarse al mínimo de pérdida local para esa cantidad de parámetros. Un ejemplo de esto se puede ver en la figura 3.8.

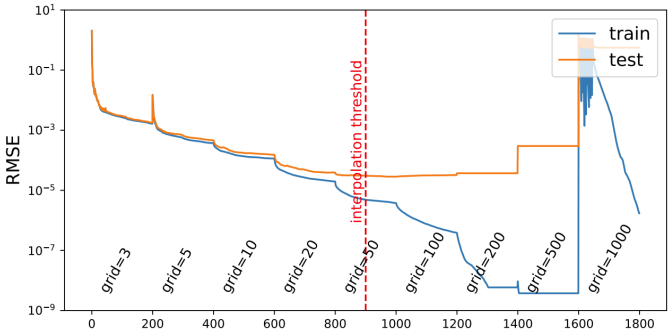


Figura 3.8: Error de entrenamiento (train) y de generalización (test) de una red KAN entrenada con grid extension en intervalos fijos, mostrando en rojo el punto en el que el modelo genera el menor error de test. Se puede ver como, aunque al aumentar el tamaño de la grid el error de entrenamiento siempre disminuye, para el error de generalización si que existe un punto óptimo en el que deja de disminuir y empieza a aumentar. Fuente: [6]

Normalmente, la grid extension se suele implementar con extensiones en puntos arbitrarios del entrenamiento, quedando normalmente a decisión del programador el punto exacto en la que se realiza cada extensión y el tamaño de la grid que pasará a tener la red tras cada extensión. No obstante, otros esquemas más sofisticados que el mostrado anteriormente implementan la grid extension de forma dinámica, aumentándola automáticamente cuando ven que el rendimiento del modelo se ha quedado estancado con la cantidad de parámetros actual.

3.4.2.1 Cálculo de parámetros de los aproximadores

Aunque no existe una fórmula exacta para obtener los nuevos parámetros de cada aproximador tras realizar una grid extension, podemos utilizar métodos de optimización numérica que hagan que las funciones producidas antes y después de la grid extension sean lo más parecidas posible. Por ejemplo, podríamos minimizar el error cuadrático producido por la función para los valores del conjunto de entrenamiento. Sea α los parámetros del aproximador original, β los parámetros tras expandir la grid, f_α y f_β las salidas producidas por el aproximador para los parámetros α y β, y X el conjunto de entrenamiento, se obtiene la siguiente fórmula para describir los parámetros tras realizar la grid extension:

$$\beta = \underset{\beta_i \in \mathbb{R}}{\operatorname{argmin}} \sum_{x \in X} \left(f_\beta(x) - f_\alpha(x)\right)^2 \tag{3.28}$$

Las fórmulas de f_α y f_β dependen de los aproximadores que se estén utilizando en la red. En el caso de que se estén utilizando splines, a partir de la definición obtenemos $f_\alpha(x) =$

$\sum_i \alpha_i B_{i,k}(x)$, por lo que tendríamos la siguiente expresión para obtener los nuevos parámetros tras la expansión de la grid:

$$\beta = \underset{\beta_i \in \mathbb{R}}{\operatorname{argmin}} \sum_{x \in X} \left(\sum_i \beta_i B'_{i,k}(x) - \sum_i \alpha_i B_{i,k}(x) \right)^2 \tag{3.29}$$

Hay que destacar que en la ecuación 3.29 las funciones base de f_α y f_β no son iguales ($B_{i,k}(x) \neq B'_{i,k}(x)$), ya que expandir la grid modifica los nodos de la grid, y por lo tanto también modifica las funciones base.

3.5 Propiedades

Las redes KAN, gracias a su arquitectura basada en la suma de funciones de una variable, tienen propiedades muy interesantes al ser comparadas con las redes neuronales tradicionales. En este aparatado se enumeran y explican las propiedades más importantes y su utilidad en el campo de la inteligencia artificial.

3.5.1 Interpretabilidad

Las redes KAN, al estar basadas en la suma de funciones, son mucho más interpretables que las redes neuronales tradicionales. Mientras que las redes neuronales tradicionales necesitan una gran cantidad de pesos y capas para aproximar de forma precisa relaciones complejas, las redes KAN no necesitan muchas capas, ya que son capaces de aproximar tanto de forma local (en cada uno de los aproximadores) como global (en la red como conjunto). De esta forma, se pueden representar relaciones complejas de forma relativamente sencilla, mientras que con redes neuronales tradicionales esto es una tarea mucho más compleja, y el resultado normalmente es mucho más difícil de interpretar. Se puede ver un ejemplo de esto en la figura 3.9.

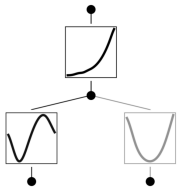


Figura 3.9: KAN entrenada para aproximar la función $e^{sin(\pi x_1)+x_2^2}$. Como se puede ver, la red ha aprendido la estructura de la función, habiendo obtenido una función con forma de $\sin(\pi x)$ para x_1, otra con forma de x^2 para x_2, y una con forma de e^x para la suma de ambas funciones anteriores. La opacidad de las funciones indica la escala de cada una. Fuente: [6]

Esto es especialmente cierto cuando las relaciones entre las entradas y las salidas de la red son funciones continuas, ya que la red KAN que utilicemos para aprender estas funciones probablemente aprenderá directamente la estructura matemática de las relaciones entre las entradas y las salidas, en vez de aproximar estas relaciones como combinaciones de pasos lineales y de funciones no-lineales, como hacen las redes neuronales tradicionales.

3.5.1.1 Regresión simbólica

Es posible aprovechar la capacidad de las redes Kolmogórov-Arnold de aprender la estructura interna de las relaciones entre variables para obtener la fórmula de la relación entre las entradas y salidas producidas por la red. A este proceso se le conoce como regresión simbólica.

La regresión simbólica consiste en, básicamente, ver si las relaciones entre las entradas y las salidas de una capa coinciden con funciones matemáticas pre-definidas. De esta forma, si vemos que todos los componentes de una red KAN están representando funciones matemáticas, podemos obtener una fórmula que represente la relación aprendida por la red. Siguiendo el ejemplo de la figura 3.9, podemos ver en la figura 3.10 un ejemplo del proceso a seguir para obtener la formula simbólica de una red KAN.

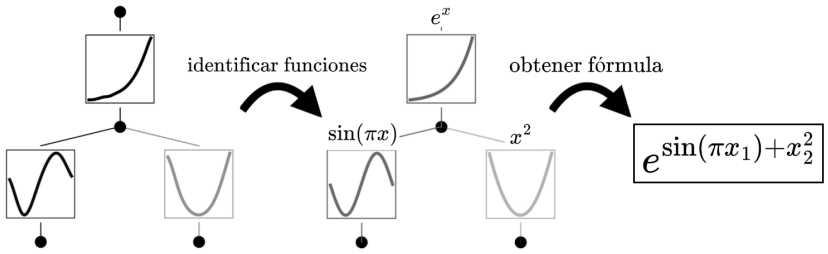


Figura 3.10: Regresión simbólica para una red KAN entrenada para aproximar la función $e^{sin(\pi x_1)+x_2^2}$. La opacidad de cada función indica la escala de la función. Fuente: [6]

Hay que destacar que, para poder realizar regresión simbólica de forma efectiva, no tenemos solo que detectar funciones simples de una variable, si no también funciones complejas con varios parámetros ajustables. Por ejemplo, si queremos detectar todas las posibles funciones sinusoides, tenemos que detectar $f(x) = c_1 \sin(c_2 x + c_3)$, por lo que tenemos 3 parámetros (c_1, c_2 y c_3) que tenemos que ajustar para poder detectar este tipo de funciones. La cantidad de parámetros crece rápidamente con la complejidad de las funciones que queremos detectar, por lo que la regresión simbólica se suele limitar a funciones matemáticas relativamente simples [64].

Aunque en teoría utilizando este método es posible obtener la función de una red KAN, es posible que no sea posible obtener una función matemática para alguna de las relaciones internas de la red KAN. En estos casos, podemos dejar indicado en la fórmula resultante la expresión exacta utilizada para calcular esa representación, aunque dependiendo de la cantidad de parámetros de la grid es posible que la fórmula obtenida sea enorme.

3.5.2 Aprendizaje continuo

Las redes neuronales tradicionales tienen una tendencia de olvidar los datos de aprendizaje anteriores al aprender nueva información, de forma que solo se fijan en la información más reciente. Este fenómeno se conoce como *Catastrophic Forgetting*, o olvido catastrófico. Es por esto que, actualmente, hay que entrenar la red con todos los datos en el proceso de aprendizaje, ya que si no la red rápidamente olvidará todos los patrones de los datos anteriores y los reemplazará por los patrones de los datos nuevos.

Las redes KAN, gracias a su arquitectura y su descomposición en funciones de una variable,

son más capaces de aprender nuevos datos sin tener que olvidar los patrones de los datos ya aprendidos, especialmente en casos en los que los datos de entrenamiento son datos numéricos continuos [6]. En estos casos, las redes KAN son capaces de mantener los conocimientos previos e incorporar a estos los patrones de los nuevos datos que está aprendiendo, sin necesitar la inclusión de todos los datos previamente aprendidos en el conjunto de entrenamiento.

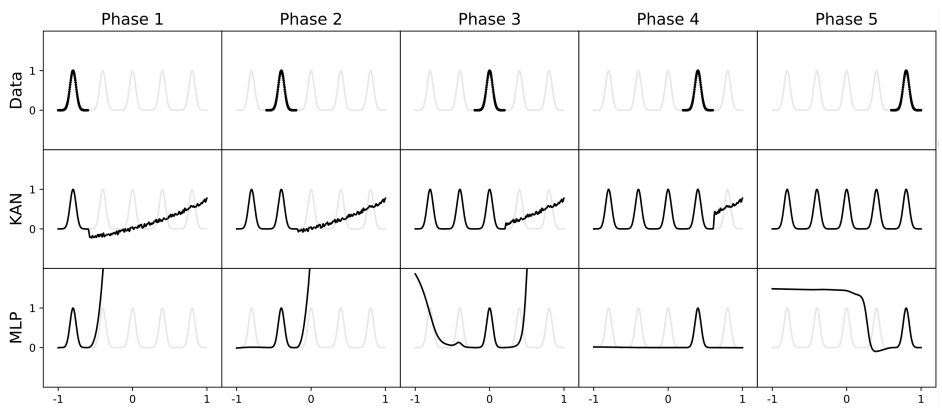


Figura 3.11: Entrenamiento por fases de una red KAN y MLP. En cada fase las redes se han entrenado con una parte de los datos. Como se puede ver, la red MLP olvida los datos de las fases anteriores, mientras que la red KAN es capaz de mantenerlos y así aprender correctamente el patrón de todos los datos. Fuente: [6]

Una de las mayores utilidades que puede tener esta propiedad de las redes KAN es la creación de sistemas que sean capaces de aprender con datos en tiempo real. Actualmente, estos datos tienen que ser concatenados con el resto de datos del conjunto de datos, lo que hace que incorporarlos al modelo sea costoso y lento. Utilizando técnicas que se aprovechen del aprendizaje continuo de las redes KAN es posible que puedan construir sistemas que aprendan constantemente fuentes de datos en tiempo real, y por lo tanto que se puedan crear modelos que siempre dispongan de los datos más actualizados.

3.5.3 Generalización de los datos

A la hora de entrenar modelos de inteligencia artificial, utilizamos grandes conjuntos de datos para intentar preparar al modelo con todos los posibles casos que podría ver. No obstante, a la hora de utilizar estos modelos en el mundo real, en la gran mayoría de los casos es imposible darle al modelo de machine learning todos los posibles casos. Incluso con modelos entrenados con conjuntos de datos enormes, siempre hay casos que el modelo no ha visto durante el entrenamiento. Es por esto que la capacidad de generalización del modelo es tan importante, ya que para funcionar correctamente en el mundo real el modelo tiene que ser capaz de generalizar los conocimientos aprendidos durante el entrenamiento.

El caso más extremo de esto se conoce como detección fuera de la distribución, o *Out-Of-Distribution Detection* en inglés. En estos casos, se intenta que el modelo sea capaz de generalizar datos muy diferentes con los que ha sido entrenado, ya sean datos preprocesados de forma diferente, datos de otro dataset, o datos sintéticos creados específicamente para incurrir este tipo de generalización [65].

Las redes KAN tienden a generalizar los datos aprendidos mucho mejor, de forma que los modelos entrenados resultantes son capaces de lidiar con este tipo de datos de forma mucho más efectiva [6], incluso teniendo una mejor capacidad de poder predecir correctamente muestras de fuera de distribución. Las redes KAN pueden suponer un gran avance en ese aspecto del machine learning, permitiendo la creación de modelos que se comporten de forma más robusta ante datos nuevos, incluso siendo capaces en algunos casos de extraer información correcta de datos radicalmente diferentes de los datos de entrenamiento originales.

4 Implementación en Python

Dado que las redes KAN son una arquitectura muy novedosa, con el fin de mostrar exactamente su funcionamiento y dar a conocer como se podría llegar a realizar su implementación, se ha realizado una implementación simple de las redes KAN en Python. Para simplificar el código, se ha implementado una arquitectura secuencial en la que todas las capas de la red son capas densas KAN, teniendo cada capa splines con grids fijas que no se pueden modificar durante el entrenamiento. La función residual también es fija, siendo la función SiLU [66].

La implementación se ha realizado utilizando tres clases: `KANNeuron`, que implementa una única función KAN junto con su spline; `KANLayer`, que implementa una capa densa KAN con todas las funciones KAN correspondientes; y `KAN`, que implementa una red KAN completa. Además de estas tres clases, se han implementado algunas funciones auxiliares que no son parte de ninguna de las tres clases.

Durante la implementación, se ha intentado optimizar bastante el código, utilizando cuando ha sido posible operaciones vectoriales de numpy [67], que son mucho más rápidas que las operaciones equivalentes en Python estándar. No obstante, se ha decidido no realizar muchas optimizaciones para priorizar la claridad y limpieza del código, ya que el objetivo principal de esta implementación es mostrar el funcionamiento de las redes KAN de forma práctica.

4.1 Funciones auxiliares

A lo largo de la implementación se han utilizado unas pocas funciones auxiliares que no son parte de ninguna de las 3 clases implementadas. Estas son la función sigmoide (`sigmoid`), la función SiLU (`silu`) y la derivada de la función SiLU (`silu_d`). También se ha implementado otra función más (nombrada `zdiv`), que calcula a/b, pero devuelve 0 cuando $b = 0$. Esta última función es necesaria para implementar la fórmula de Cox-de Boor (ver ecuación 3.8).

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def silu(x):
    return np.multiply(x, sigmoid(x))

def silu_d(x): # derivada de SiLU
    s = sigmoid(x)
    return np.multiply(s, (1 + x*(1-s)))

def zdiv(a, b): # a/b, pero b=0 devuelve 0
    with np.errstate(divide='ignore', invalid='ignore'): # ignorar warnings
        c = np.true_divide(a,b)
        c[c == np.inf] = 0
```

```
15        return np.nan_to_num(c)
```

Código 4.1: Funciones auxiliares utilizadas en la implementación propia de redes KAN, que definen las función base utilizada (SiLU), su derivada y la función zdiv, que implementa la división de dos números pero devuelve 0 cuando el divisor es 0, y se utiliza para implementar las fórmulas de Cox-de Boor

4.2 Clase KANNeuron

La clase más básica en la implementación es la clase `KANNeuron`, que representa una única función KAN de la red. La clase implementa la spline de la función KAN, calculando todos los parámetros y funciones base. Las variables de la clase son las siguientes:

- `k`: el grado de la spline
- `b`: el valor de la función residual
- `b_d`: el valor de la derivada de la función residual
- `grid`: la grid de la spline
- `min`: el valor mínimo de la grid
- `max`: el valor máximo de la grid
- `bases`: el valor de las funciones base de la spline
- `bases_d`: el valor de las derivadas de las funciones base de la spline
- `s`: el valor de la spline
- `s_d`: el valor de la derivada de la spline
- `wb`: el peso que de la función residual
- `ws`: el peso de la spline

Hay que mencionar que, aunque no lo parezca a primera vista, muchas de estas variables son vectores (`b`, `b_d`, `s`, `s_d`) o matrices (`bases`, `bases_d`), ya que se calculan para varios valores en paralelo. Cabe también decir que, por razones que veremos en el apartado 4.2.1, es necesario que `grid` sea un vector fila (matriz $1 \times n$) para que la implementación funcione correctamente.

4.2.1 Método spline()

El método más complejo de la clase `KANNeuron` es el método `spline()`, que recalcula todas las variables de la spline al recibir un nuevo conjunto de datos de entrada. Para implementarlo de forma eficiente, se han empleado arrays de la librería `numpy`, que son capaces de realizar operaciones en paralelo de forma mucho más eficiente que las estructuras predeterminadas de Python [67]. El código de este método está basado en el código de splines de Prateek

Gupta [42], aunque se ha modificado bastante para adaptarlo a los requerimientos de la implementación actual.

El método recibe un parámetro, `x`, que es el conjunto de valores para el que hay que calcular valores. La función requiere que `x` sea un vector columna (matriz $m \times 1$) para funcionar correctamente. El método, además, está basado en la fórmula de Cox-de Boor (ver ecuación 3.8). Como se puede ver en la fórmula, el primer paso es calcular el caso base, $B_{i,0}$. Para calcularlo en paralelo para todos los puntos, extraeremos de `grid` las variables t_i y t_{i+1}, utilizando la notación de *slicing* de Python [68]. A partir de t_i y t_{i+1} es posible calcular $B_{i,0}$ con el siguiente código:

```
self.bases = (x >= self.grid[:,:-1]) * (x < self.grid[:, 1:])
```

Código 4.2: Código utilizado en la implementación propia de las redes KAN para calcular el caso base de la fórmula recursiva Cox-de Boor, utilizado para el cálculo de las splines de la red KAN

Como `grid` es un vector fila y `x` es un vector columna, todas las operaciones que hagamos entre ellos nos darán todos los posibles resultados. Este código, por lo tanto, almacena en la variable `bases` una matriz de tamaño $m \times (n-1)$, de tal forma que el elemento i, j de la matriz es igual a $B_{i,0}(x_j) = t_i \leq x_j < t_{i+1}$. Podemos implementar de forma similar el caso recursivo de Cox-de Boor, calculando t_i, t_{i+1}, t_{i+k} y t_{i+k+1} a partir de `grid` y calculando $B_{i,k-1}$ y $B_{i+1,k-1}$ a partir de los valores almacenados anteriormente en la variable `bases`.

Una vez calculados todas estas variables auxiliares, podemos fácilmente calcular el nuevo valor de `bases` para el caso recursivo siguiendo la fórmula 3.8. Si ejecutamos el caso recursivo k veces, entonces el valor de `bases` será igual al resultado de calcular cada una de las funciones base de la spline para cada punto en x. Para implementar las divisiones de la fórmula Cox-de Boor, se ha utilizado la función auxiliar `zdiv()`.

```
self.bases = (x >= self.grid[:,:-1]) * (x < self.grid[:, 1:]) # caso base
for i in range(1, self.k+1) # caso recursivo
    ti = self.grid[:, :-k-1] # [0:n-k-1]
    ti1 = self.grid[:, 1:-k] # [1:n-k]
    tik = self.grid[:, k:-1] # [k:n-1]
    tik1 = self.grid[:, k+1:] # [k+1:n]
    b0 = self.bases[:, :-1]
    b1 = self.bases[:, 1:]
    p0 = zdiv(x-ti, tik-ti) * b0
    p1 = zdiv(tik1-x, tik1-ti1) * b1
    self.bases = p0 + p1
```

Código 4.3: Código utilizado para el cálculo completo de las funciones base de las splines. El código calcula de forma iterativa la fórmula de Cox-de Boor, utilizando un bucle para calcular cada uno de los órdenes de la spline. Calcula también ciertas variables auxiliares utilizadas para calcular las derivadas de las splines posteriormente

Lo único que nos queda es calcular las variables `bases_d`, `s` y `s_d`, que se puede hacer aplicando las fórmulas 3.6, 3.8 y 3.10. El código completo se puede ver a continuación:

```python
def spline(self, x):
    self.bases = (x >= self.grid[:,:-1]) * (x < self.grid[:, 1:]) # caso base
    for k in range(1, self.k+1): # caso recursivo
        ti = self.grid[:, :-k-1] # [0:m-k-1]
        ti1 = self.grid[:, 1:-k] # [1:m-k]
        tik = self.grid[:, k:-1] # [k:m-1]
        tik1 = self.grid[:, k+1:] # [k+1:m]
        b0 = self.bases[:, :-1]
        b1 = self.bases[:, 1:]
        p0 = zdiv(x-ti,tik-ti) * b0
        p1 = zdiv(tik1-x,tik1-ti1) * b1
        self.bases = p0 + p1

    # derivadas
    b0_d = zdiv(k,tik-ti) * b0
    b1_d = zdiv(k,tik1-ti1) * b1
    self.bases_d = b0_d - b1_d

    # valor spline
    self.s = np.sum(self.coefs * self.bases, axis=1, keepdims=True)
    self.s_d = np.sum(self.coefs * self.bases_d, axis=1, keepdims=True)
```

Código 4.4: Código completo del método spline() de la clase KANNeuron, responsable de calcular todos los valores de splines de la clase. Incorpora el código visto anteriormente utilizado para implementar la fórmula Cox-de Boor, y a partir de eso calcula las funciones base (self.bases), las derivadas de las funciones base (self.bases_d), el valor de la spline (self.s) y el valor de la derivada de la spline (self.s_d)

4.2.2 Método train()

El método train() actualiza los pesos de la neuronas (`wb`, `ws` y `coefs`) a partir de un vector de deltas y de una tasa de aprendizaje (`delta` y `lr`).

Como se puede ver en la ecuación 3.27, para calcular el error de una neurona a partir del vector de deltas correspondiente, tenemos la expresión $\partial E/\partial w^l_{b,i,j} = \delta^{l+1}_j b^l(x^l_i)$. Como el valor de $b^l(x^l_i)$ lo almacenamos en la variable `self.b` al realizar el feedforward de la red (en el método `__call__`), podemos calcular el error de `wb` fácilmente a partir de la fórmula. Para ajustar el valor de `wb`, calculamos el error y lo multiplicamos por `lr`, la tasa de entrenamiento. Como cada valor de delta afecta de forma independiente a `wb`, aplicaremos la fórmula respecto a la media del valor medio del vector `delta`. Sea `avgDelta` la variable que contiene el valor medio de `delta`, nos queda el código `self.wb -= lr * avgDelta * self.b`.

De forma similar, podemos actualizar `ws` utilizando la fórmula $\partial E/\partial w^l_{S,i,j} = \delta^{l+1}_j S(x^l_i)$. Como $S(x^l_i)$ está ya almacenado en la variable `self.s`, obtenemos el código `self.ws -= lr * avgDelta * self.s`.

Por último, para actualizar los coeficientes de la spline (almacenados en la variable `coefs`), utilizaremos $\partial E/\partial \alpha^l_{k,i,j} = \delta^{l+1}_j w^l_{S,i,j} B_k(x^l_i)$, también obtenido de la ecuación 3.27. No obstante, a diferencia que para `wb` y `ws`, cada delta afecta de forma diferente a los valores de `coefs`, ya que para cada coeficiente tenemos que multiplicarlo con el resultado de evaluar su función

base de la spline ($B_k(x_i^l)$ para el coeficiente $\alpha_{k,i,j}^l$). Es por esto que, para actualizar `coefs`, es necesario utilizar el producto matricial (`@` en Python). Como el resultado es una matriz, pero necesitamos un vector, utilizaremos `np.squeeze` para eliminar la dimensión vacía de la matriz. Podemos ver todo el código resultante en el código 4.5.

```
def train(self, delta, lr):
    avgDelta = np.mean(delta)
    self.coefs -= np.squeeze(lr * self.ws * (delta.T @ self.bases))
    self.wb -= np.sum(lr * avgDelta * self.b)
    self.ws -= np.sum(lr * avgDelta * self.s)
```

Código 4.5: Código del método train() de la clase KANNeuron, responsable de actualizar todos los coeficientes de la spline (self.coefs) y los pesos (self.wb y self.ws) a partir del vector de deltas y de la learning rate recibida

Lo único que queda por explicar es que el código utilizado para actualizar `wb` y `ws` devuelve un vector de numpy de un único elemento, aunque `wb` y `ws` son valores numéricos. Es por esto que se ha añadido la función `np.sum` para convertir los vectores a valores numéricos.

4.2.3 Otros métodos

El resto de métodos de la clase `KANNeuron` son bastante simples. Es por eso que, en vez de explicar todo el código de cada uno de estos métodos, se va a dar un pequeño resumen del propósito de cada método en este apartado.

- Método `__init__`: Constructor de la clase, que inicializa todas las variables y comprueba que la forma de `grid` y `coefs` es compatible.
- Método `__call__`: Overload del operador "`()`" para la clase. Al ejecutarse, calcula todas las variables internas a partir del parámetro `x`, utilizando el método `spline()` para realizar la mayoría de los cálculos relacionados con la spline de la neurona. Devuelve el resultado de evaluar la neurona a partir de `x`.
- Método `phi_d`: calcula la derivada del valor de salida respecto a `x`, utilizando los valores internos ya calculados por `__call__` y `spline`. Este método es necesario en las siguientes clases para implementar otros aspectos de la red.

4.2.4 Código completo

El código completo en Python de la clase `KANNeuron` se puede ver a continuación:

```
class KANNeuron:
    def __init__(self, grid, coefs, k, wb, ws):
        self.k = k
        self.wb = wb
        self.ws = ws
        self.coefs = coefs
        self.grid = grid
        self.min = np.min(grid)
```

```
        self.max = np.max(grid)
        assert(grid.shape[1]-k-1 == self.coefs.shape[0])

    def __call__(self, x):
        self.x = np.clip(x, self.min, self.max) # clamp to fit grid
        self.spline(self.x)
        self.b = silu(self.x)
        self.b_d = silu_d(self.x)
        return self.wb * self.b + self.ws * self.s

    def spline(self, x):
        self.bases = (x >= self.grid[:,:-1]) * (x < self.grid[:, 1:]) # k = 0
        for k in range(1, self.k+1):
            ti = self.grid[:, :-k-1] # [0:m-k-1]
            ti1 = self.grid[:, 1:-k] # [1:m-k]
            tik = self.grid[:, k:-1] # [k:m-1]
            tik1 = self.grid[:, k+1:] # [k+1:m]
            b0 = self.bases[:, :-1]
            b1 = self.bases[:, 1:]
            p0 = zdiv(x-ti,tik-ti) * b0
            p1 = zdiv(tik1-x,tik1-ti1) * b1
            self.bases = p0 + p1

        p0_d = zdiv(k,tik-ti) * b0
        p1_d = zdiv(k,tik1-ti1) * b1
        self.bases_d = p0_d - p1_d
        self.s = np.sum(self.coefs * self.bases, axis=1, keepdims=True)
        self.s_d = np.sum(self.coefs * self.bases_d, axis=1, keepdims=True)

    def phi_d(self):
        return self.wb * self.b_d + self.ws * self.s_d

    def train(self, delta, lr):
        avgDelta = np.mean(delta)
        self.coefs -= np.squeeze(lr * self.ws * (delta.T @ self.bases))
        self.wb -= np.sum(lr * avgDelta * self.b)
        self.ws -= np.sum(lr * avgDelta * self.s)
```

Código 4.6: Código completo de la clase KANNeuron utilizada en la implementación propia de las redes KAN

4.3 Clase KANLayer

La siguiente clase de la implementación es la clase `KANLayer`, que representa una capa densa KAN de una red. Las variables de la clase son las siguientes:

- `nIn`: número de entradas de la capa
- `nOut`: número de salidas de la capa

- `xavier`: valor calculado a partir de `nIn` y `nOut` utilizado para realizar *Xavier initialization* [69], que es un método para inicializar los pesos de la red de forma aleatoria
- `grid`: la grid de todas las neuronas de la capa. Utilizado en la inicialización
- `k`: el grado de las splines de las neuronas de la capa. Utilizado en la inicialización
- `n`: número de coeficientes para cada neurona de la capa. Utilizado en la inicialización
- `neurons`: matriz $n^{in} \times n^{out}$ de objetos `KANNeuron`, que almacena todas las neuronas de la capa
- `activations`: las activaciones de todas las neuronas de la red. Se actualiza automáticamente cada vez que se llama al método `__call__`

4.3.1 Método ___call___()

El método `__call__` es responsable de evaluar todas las neuronas de la capa para una conjunto de valores de entrada `x`. Primero, el método comprueba que la matriz `x` contiene la cantidad de entradas necesaria para poder ejecutar la capa (`nIn`). A partir de esto, calculamos el valor de `activations` para el valor del parámetro `x`, llamando al método `__call__` de cada una de las `KANNeuron` de `neurons` pasándole la entrada correspondiente de `x`.

Una vez calculado `activations`, sumamos todos los valores que corresponden a la misma salida, y comprobamos que el resultado tiene las dimensiones correctas respecto a `x` y que tiene la cantidad necesaria de valores de salida (`nOut`).

```
def __call__(self, x):
    assert(x.shape[-1] == self.nIn)
    self.activations = np.array([[self.neurons[i, j](x[:,[i]])
        for j in range(self.nOut)] for i in range(self.nIn)])
    result = np.squeeze(np.sum(self.activations, axis=0).T)
    assert(result.shape[1] == self.nOut)
    assert(result.shape[0] == x.shape[0])
    return result
```

Código 4.7: Código del método ___call___() de la clase KANLayer, que calcula el resultado de una capa KAN. Calcula los resultados parciales de todos los objetos KANNeuron internos a la capa KAN, y devuelve el resultado correspondiente. También comprueba que las dimensiones de los datos de entrada y de salida producidos son correctas, utilizando el método assert() de Python.

4.3.2 Método train()

El método `train` de la clase `KANLayer` es responsable de calcular a partir de los deltas recibidos de la capa siguiente calcular los deltas de entrada de la capa anterior en la red. También llama al método train de todas las neuronas de `neurons` con los deltas correspondientes, actualizando así los parámetros de todas las neuronas de la capa.

Primero es necesario asegurar que el parámetro `delta` tiene dimensiones compatibles, comprobando que coincide con la cantidad de salidas de la capa. A partir de esto, se almacena en

`product` el resultado de multiplicar el delta correspondiente con la derivada de cada neurona, tal y como se especifica en la ecuación 3.27. Entonces, sumamos los valores de `product` y almacenamos los nuevos deltas en la variable `result`. Por último, comprobamos que los nuevos deltas tienen las dimensiones esperadas, y llamamos al método train de todas las neuronas en `neurons` con el delta correspondiente.

```
def train(self, deltas, lr):
    assert(deltas.shape[1] == self.nOut)
    product = np.array([[np.expand_dims(deltas[:,j], 1) * self.neurons[i, j].↩
        ↪ phi_d() for j in range(self.nOut)] for i in range(self.nIn)]).T
    result = np.squeeze(np.sum(product.T, axis=1).T)
    assert(result.shape[1] == self.nIn)
    assert(result.shape[0] == deltas.shape[0])

    for j in range(self.nOut):
        for i in range(self.nIn):
            self.neurons[i, j].train(np.expand_dims(deltas[:,j], 1), lr)

    return result
```

Código 4.8: Código del método train() de la clase KANLayer, que actualiza todos los pesos de la capa KAN dados los deltas y la learning rate llamando al método train de todos los objetos KANNeuron internos. Además, calcula y devuelve los deltas de la capa anterior de la red para poder realizar backpropagation utilizando el resultado devuelto por la función, comprobando que las dimensiones de los deltas de entrada y salida son correctas.

4.3.3 Otros métodos

El resto de métodos de la clase `KANLayer` son los siguientes:

- Método `initNeuron`: método que inicializa una neurona mediante *Xavier inicialization*, utilizando el valor `self.xavier` calculado en `__init__`. Devuelve el objeto `KANNeuron` que se ha creado e inicializado.
- Método `__init__`: Constructor de la clase, que inicializa todas las variables y llama múltiples veces a `initNeuron` y guarda el resultado en la matriz `neurons`.

4.3.4 Código completo

El código completo de la clase `KANLayer` es el siguiente:

```
class KANLayer:
    def initNeuron(self):
        coefs = norm.rvs(scale=0.01, size=self.n)
        aw = norm.rvs(scale=self.xavier, size=1) # Xavier initialization
        return KANNeuron(self.grid, coefs, self.k, aw, 1)

    def __init__(self, nIn, nOut, grid, k=3):
```

```
        self.nIn = nIn
        self.nOut = nOut
        self.xavier = math.sqrt(2 / (nIn + nOut))
        self.grid = grid
        self.k = k
        self.n = grid.shape[1] - k - 1
        self.neurons = np.array([[self.initNeuron() for j in range(nOut)] for i
            ↪ in range(nIn)])

    def __call__(self, x):
        assert(x.shape[-1] == self.nIn)
        self.activations = np.array([[self.neurons[i,j](x[:,[i]])
            for j in range(self.nOut)] for i in range(self.nIn)])
        result = np.squeeze(np.sum(self.activations, axis=0).T)
        assert(result.shape[1] == self.nOut)
        assert(result.shape[0] == x.shape[0])
        return result

    def train(self, deltas, lr):
        assert(deltas.shape[1] == self.nOut)
        product = np.array([[np.expand_dims(deltas[:, j], 1)
            * self.neurons[i, j].phi_d()
            for j in range(self.nOut)] for i in range(self.nIn)]).T
        result = np.squeeze(np.sum(product.T, axis=1).T)
        assert(result.shape[1] == self.nIn)
        assert(result.shape[0] == deltas.shape[0])

        for j in range(self.nOut):
            for i in range(self.nIn):
                self.neurons[i, j].train(np.expand_dims(deltas[:,j], 1), lr)

        return result
```

Código 4.9: Código completo de la clase KANLayer utilizada en la implementación propia de las redes KAN.

4.4 Clase KAN

La clase `KAN` es la clase más simple de la implementación, ya que la mayoría de su funcionalidad es llamar a las funciones de las clase `KANLayer`. La clase `KAN` es una red KAN completa, incluyendo un método para evaluar la red y otro para entrenarla ajustando sus pesos. Tiene los siguientes métodos:

- Método `__init__`: inicializa la red, utilizando una lista que codifica la cantidad de entradas y salidas de cada capa. El elemento i es la cantidad de entradas de la capa i, mientras que el elemento $i+1$ es la cantidad de salidas. A partir de esta lista, se crean todos los objetos `KANLayer` y se almacenan en la variable `layers`.
- Método `__call__`: calcula el valor de la red para unos datos de entrada `x`, pasando los datos por todas las capas. Devuelve el resultado obtenido.

- Método `train`: utilizando el último valor de `self.x`, calcula los deltas de la última capa, y va llamando al método `train` de todas las capas en orden inverso para realizar backpropagation.

El código completo de la clase `KAN` se puede ver a continuación:

```python
class KAN:
    def __init__(self, l, grid, k=3):
        self.layers = []
        for i in range(len(l)-1):
            self.layers.append(KANLayer(l[i], l[i+1], grid, k))

    def __call__(self, x):
        self.x = x
        for layer in self.layers:
            self.x = layer(self.x)
        return self.x

    def train(self, y, lr):
        deltas = 2*(y - self.x)
        for layer in reversed(self.layers):
            deltas = layer.train(deltas, lr)
```

Código 4.10: Código completo de la clase KAN utilizada en la implementación propia de redes KAN.

5 Experimentos

En este capítulo realizamos varios experimentos con el fin de verificar algunas de las propiedades de las redes KAN, y de compararlas con las redes tradicionales. Dada la falta de experimentos respecto al tema, se ha decidido realizar experimentos para medir el rendimiento, eficiencia y otros aspectos de las redes convolucionales KAN. Además, como las redes convolucionales son una clase de red fundamental para muchas aplicaciones, especialmente para el procesamiento de datos estructurados (imágenes, audio, etc.), comparar las redes convolucionales tradicionales y las redes convolucionales KAN nos podrá dar una pista de si las redes KAN se podrían utilizar para mejorar las redes convolucionales actuales.

Para todos los experimentos realizados en este apartado se ha utilizado el framework `pytorch` [70] para definir, entrenar y evaluar los modelos. Todos los resultados obtenidos en los experimentos se han obtenido a partir de modelos entrenados con el optimizador `AdamW` [71], configurado utilizando λ (weight decay) $= 0.0001$ y lr (learning rate) $= 0.01$. Además, se ha empleado el *scheduler* `ExponentialLR` [72] para reducir la *learning rate* del optimizador dependiendo de la época de entrenamiento, con γ (gamma) $= 0.9$. Todos los modelos se han entrenado durante 20 épocas utilizando esta configuración, siendo entrenados únicamente con las muestras del conjunto de datos de entrenamiento correspondiente al experimento. Para obtener los resultados tras el entrenamiento se ha medido la tasa de aciertos y la *f-score* de los modelos obtenidos con el conjunto de evaluación del experimento.

5.1 Conjuntos de datos utilizados

Para evaluar el rendimiento de redes convolucionales, existen muchos conjunto de datos estandarizados, cada uno con sus ventajas e inconvenientes [73]. Para este trabajo se ha decidido utilizar los datasets MNIST y CIFAR-10. Se han elegido estos conjuntos de datos no solo por su amplio uso a la hora de medir el rendimiento de modelos y arquitecturas de clasificación de imágenes, sino también por su reducido coste computacional necesario para entrenar modelos con estos datasets y medir su rendimiento.

5.1.1 MNIST

MNIST es un conjunto de datos estándar utilizado para comprobar la eficacia de arquitecturas en tareas de clasificación de imágenes [74]. El dataset está formado por imágenes de 28×28 en blanco y negro, representando cada una un dígito del 0 al 9. En la figura 5.1 podemos ver algunos ejemplos de imágenes del conjunto de datos.

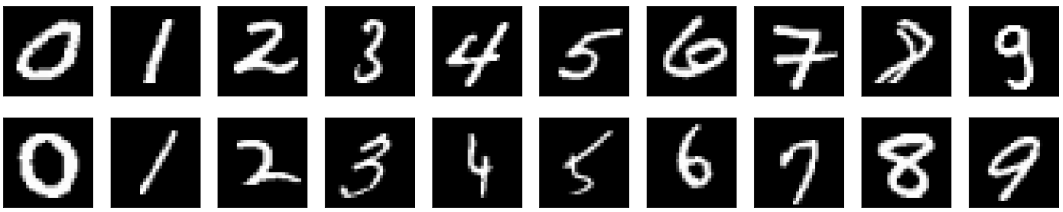


Figura 5.1: Muestras de imágenes de cada una de las 10 clases del dataset MNIST. Fuente: elaboración propia

MNIST contiene 60000 imágenes de 28×28 píxeles con su correspondiente dígito $(0 - 9)$, teniendo exactamente 6000 muestras para cada una de sus 10 clases. Utilizaremos 50000 de estas muestras para entrenar los modelos, y las 10000 restantes para evaluar los resultados de los modelos producidos por el proceso de entrenamiento.

Las imágenes de MNIST se han normalizado respecto a la media y la varianza de los valores de los píxeles de las muestras del conjunto de entrenamiento, de forma de que la distribución de los valores de los píxeles del conjunto de entrenamiento tenga media 0 y varianza 1. Se ha normalizado utilizando 0.1307 para la media y 0.3081 para la varianza. A parte de esta normalización, no se ha aplicado ningún otro pre-procesado a las imágenes de MNIST.

5.1.2 CIFAR-10

CIFAR-10 es otro conjunto de datos frecuentemente utilizado en tareas de clasificación de imágenes, para medir la eficiencia de modelos de una forma estándar [75]. Al igual que MNIST, está formados por una gran cantidad de muestras de imágenes agrupadas en 10 clases. Estas clases son `airplane` (aviones), `automobile` (coches), `bird` (pájaros), `cat` (gatos), `deer` (ciervos), `dog` (perros), `frog` (ranas), `horse` (caballos), `ship` (barcos) y `truck` (camiones). En la figura 5.2 podemos ver algunos ejemplos de imágenes del conjunto de datos. Las imágenes de CIFAR-10 son imágenes a color que contienen 32×32 píxeles. Como cada uno de los canales de las imágenes solo puede ir de 0 a 255, las imágenes están codificadas en RGB-8.

Figura 5.2: Muestras de imágenes de cada una de las 10 clases del dataset CIFAR-10, incluyendo (de izquierda a derecha) imágenes de aviones, coches, pájaros, gatos, ciervos, perros, ranas, caballos, barcos y camiones, que son las 10 clases de imágenes del conjunto de datos. Fuente: elaboración propia

El conjunto de datos CIFAR-10 contiene 60000 imágenes con $32 \times 32 \times 3$ valores cada una, además de su clase correspondiente. Al igual que para MNIST, se ha dividido el dataset en dos, dejando 50000 muestras para el conjunto de entrenamiento y 10000 para el de evaluación.

Las imágenes de CIFAR-10 también se han normalizado, para que la distribución de los

valores de los píxeles tenga media 0 y varianza 1. Como cada píxel tiene 3 valores (ya que las imágenes son RGB), tenemos que normalizar cada uno de los 3 canales de las imágenes por separado. Se han utilizado los valores 0.4914, 0.4822 y 0.4465 para normalizar respecto a la media; y 0.2023, 0.1994 y 0.2010 para normalizar respecto a la varianza. No se ha aplicado ningún otro procesamiento a las imágenes del dataset.

5.2 Arquitecturas utilizadas

Dado que MNIST y CIFAR-10 son conjuntos de datos formado por imágenes, se han utilizado arquitecturas con capas convolucionales bidimensionales, con el fin de que los modelos aprendan de forma efectiva la estructura de los datos del dataset (ver apartado 2.3.2 para más información). Utilizaremos capas convolucionales bidimensionales tanto para las redes convolucionales tradicionales como para redes KAN.

5.2.1 Redes CNN

Para las redes tradicionales, hemos utilizado una red neuronal convolucional, o CNN. La arquitectura utilizada contiene dos capas convolucionales, que se han implementado utilizado la clase `Conv2D` [76] de Pytorch. La salida de cada una de estas capas convolucionales se pasa por una función de activación ReLU. Después de pasar el resultado de cada capa por la función de activación, la salida de las capas convolucionales se pasa también por un MaxPool2D de tamaño 2×2, con el fin de mejorar la generalización del modelo y a la vez de reducir la cantidad de parámetros de las siguientes capas. Después de realizar todo esto para las dos capas convolucionales, se aplanan los datos y se pasan por una capa densa. A esta capa densa la sigue, al igual que a las capas convolucionales, una función de activación ReLU. Las capas densas utilizan la clase `Linear` [77] de Pytorch.

Con el fin de combatir y reducir el *overfitting* del modelo durante el proceso de entrenamiento, se ha añadido una capa de Dropout después de la primera capa densa. Esta capa anula un porcentaje de las entradas recibidas de forma aleatoria durante el entrenamiento, haciendo que todos los valores anulados sean 0. De esta forma, la red no depende tanto de la memorización de una única configuración de valores, ya que en algunos casos la red no podrá depender de que estos valores no sean anulados. Esto, si se utiliza correctamente, puede aumentar significativamente la capacidad de generalización del modelo. Se ha utilizado una capa con 50% Dropout, que significa que durante el entrenamiento, cada entrada tiene cada vez un 50% de probabilidad de ser anulada.

Después del Dropout, se ha utilizado una segunda capa densa, aunque sin función de activación. De esta forma, no se limitan las posibles salidas del modelo a un rango determinado. En la figura 5.3 se puede ver una representación gráfica de la estructura completa de los modelos CNN utlizados en los experimentos.

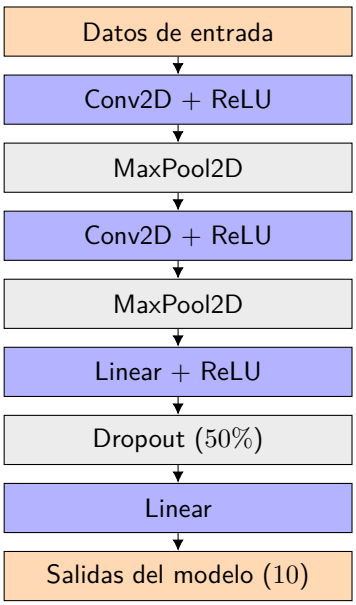


Figura 5.3: Estructura de las redes CNN utilizadas en los experimentos, con las partes entrenables del modelo en azul y la entrada/salida de datos en naranja. El modelo es una red convolucional bastante estándar, con dos capas convolucionales, dos capas densas, funciones de activación ReLU, Max Pooling y una capa de Dropout. Fuente: elaboración propia

Los modelos CNN, como se puede deducir a partir de la arquitectura utilizada, tienen 3 parámetros configurables: la cantidad de filtros de salida de la primera capa convolucional, la cantidad de filtros de salida de la segunda capa convolucional, y la cantidad de neuronas de salida de la primera capa densa. El resto de los parámetros de las capas no son configurables, ya que tienen que tener ciertos valores para que la arquitectura concuerde con los datos de entrada recibidos y para que la red produzca los datos de salida esperados.

5.2.2 Redes Conv-KAN

Para las redes KAN, se ha empleado una estructura muy similar a la de las redes CNN, utilizando capas convolucionales KAN para sustituir las capas convolucionales y capas KAN densas para sustituir las capas densas tradicionales. Se ha utilizado la capa `FastKANLayer` [43] para las capas KAN densas, y `ConvKAN` [40] para las capas convolucionales KAN. Cabe notar que la capa FastKANLayer, con el fin de calcular los resultados más rápidamente, utiliza una aproximación de las funciones base de splines, basada en el uso de funciones base radiales gaussianas [78]. Aunque este proceso no produce exactamente los mismos resultados, obtiene resultados prácticamente idénticos con un speedup de 3.33 sobre una implementación que no utilice esta aproximación [79]. Para las redes convolucionales, no se ha encontrado una implementación que utilice funciones base radiales para aproximar las funciones KAN, así que se ha utilizado una capa con las funciones KAN sin esta optimización.

Aunque se ha intentado diseñar una estructura lo más similar posible a la estructura utilizada para las redes CNN, siguen habiendo algunas diferencias notables entre las dos arquitecturas. La principal es que, como se están utilizando capas densas y convolucionales KAN, no es necesario (ni útil) utilizar funciones de activación tras las capas KAN. Es por

esto que no se ha utilizado ninguna función de activación, ya que las no-linealidades de las capas KAN deberían ser suficiente para que la red pueda ser capaz de aproximar cualquier función.

Otra diferencia mayor es que se ha quitado la capa Dropout. Al principio se probó a utilizarla, pero rápidamente fue descubierto que las capas KAN densas no pueden combinarse correctamente con capas Dropout. Esto es posible que se deba a la estructura interna de las capas KAN, y que haga que este tipo de redes no sean muy permisivas a la pérdida de parte de la información [80].

La última diferencia es que, aunque se ha sustituido la primera capa densa por una capa KAN, la segunda se ha dejado como una capa densa tradicional. Esto es por que, si nos fijamos en la estructura utilizada para las capas CNN, la segunda capa densa no tiene función de activación. Como nuestro objetivo es crear dos arquitecturas similares, cambiar la segunda capa densa por una capa KAN es equivalente a introducir una función de activación a las segunda capa densa, que no se suele hacer en el contexto de las redes convolucionales. Es por esto que, para que la arquitectura sea lo más similar posible, se ha dejado la segunda capa densa como una capa densa tradicional.

En la figura 5.4 se puede ver una representación gráfica de la estructura completa de los modelos Conv-KAN utilizados en los experimentos.

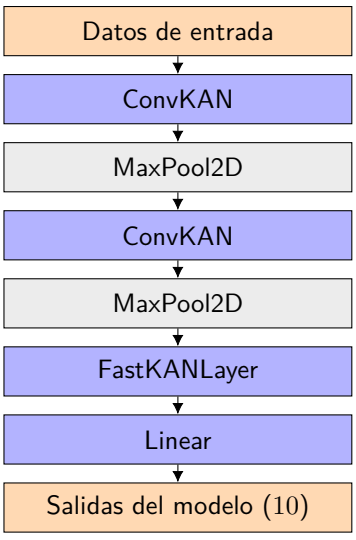


Figura 5.4: Estructura de las redes Conv-KAN utilizadas en los experimentos, con las partes entrenables en azul y la entrada/salida en naranja. Es una adaptación de la estructura de la figura 5.3 para el uso de capas KAN, necesitando la eliminación de la capa dropout y de las funciones de activación. Fuente: Elaboración propia

Al igual que con modelos CNN, los modelos Conv-KAN producidos por la arquitectura descrita anteriormente tendrán 3 parámetros configurables: la cantidad de filtros de salida de la primera capa convolucional, la cantidad de filtros de salida de la segunda capa convolucional, y la cantidad de neuronas de salida de la primera capa densa.

5.3 Eficiencia respecto al número de parámetros

En esta sección mediremos los resultados obtenidos en redes KAN y redes convolucionales tradicionales respecto al número de parámetros. Para esto, entrenaremos varios modelos ConvKAN y CNN, y estudiaremos cuál arquitectura produce mejores resultados para cada número de parámetros del modelo.

Se han elegido 5 modelos Conv-KAN y 5 modelos CNN de varios tamaños, intentando que cada uno de los modelos KAN tenga una cantidad comparable de parámetros que el modelo CNN correspondiente. Los modelos entrenados para realizar este experimento se pueden ver en la tabla 5.1.

Arquitectura	Modelo	C_1	C_2	D	Parámetros totales	
					MNIST	CIFAR-10
Conv-KAN	CKAN8	8	8	8	21378	29330
	CKAN12	12	12	12	47182	63862
	CKAN16	16	16	16	83066	111642
	CKAN20	20	20	20	129030	172670
	CKAN24	24	24	24	185074	246946
CNN	CNN10	10	20	40	22370	31350
	CNN15	15	30	60	49900	69970
	CNN20	20	40	80	88330	123890
	CNN25	25	50	100	137660	193110
	CNN30	30	60	120	197890	277630

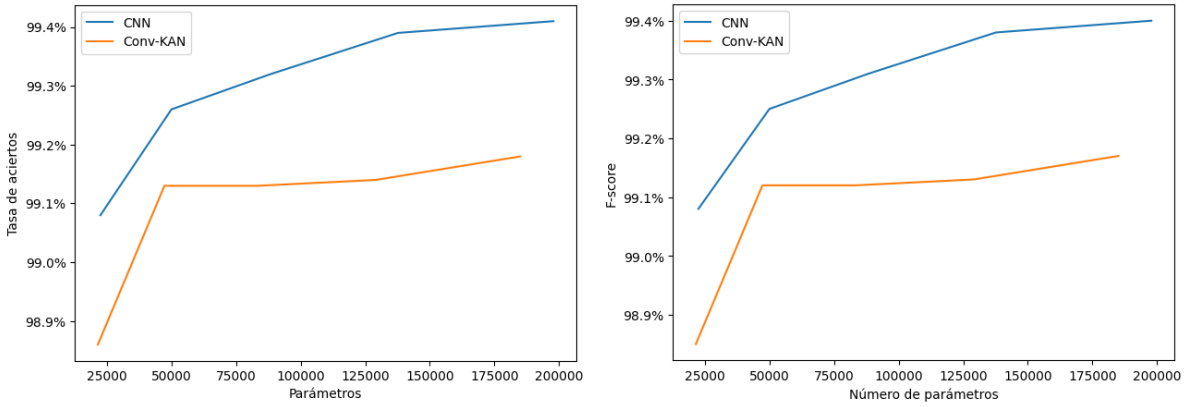
Tabla 5.1: Arquitecturas Conv-KAN y CNN entrenadas para realizar el experimento de eficiencia de parámetros, siendo C_1 la cantidad de filtros de la primera capa convolucional, C_2 la cantidad de filtros de la segunda capa convolucional y D la cantidad de neuronas de la primera capa densa. Las capas convolucionales y densas de los modelos Conv-KAN utilizan los valores predeterminados para el tamaño de grid, utilizando todas una grid uniforme. Los modelos tienen cantidades diferentes de parámetros para MNIST que para CIFAR-10, ya que las imágenes de MNIST son de distinto tamaño que las de CIFAR-10, y las imágenes de CIFAR-10 son en color RGB mientras que las de MNIST son en blanco y negro. Fuente: elaboración propia

5.3.1 MNIST

Los resultados obtenidos tras entrenar los modelos descritos anteriormente para el dataset MNIST se pueden ver en la tabla 5.2. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.5.

Modelo	Parámetros totales	Tasa de aciertos	F-score
CKAN8	21378	98.9%	98.8%
CKAN12	47182	99.1%	99.1%
CKAN16	83066	99.1%	99.1%
CKAN20	129030	99.1%	99.1%
CKAN24	185074	99.2%	99.2%
CNN10	22370	99.1%	99.1%
CNN15	49900	99.3%	99.2%
CNN20	88330	99.3%	99.3%
CNN25	137660	99.4%	99.4%
CNN30	197890	**99.4%**	**99.4%**

Tabla 5.2: Resultados obtenidos para el dataset MNIST en el experimento de eficiencia respecto al número de parámetros tras entrenar los modelos descritos en la tabla 5.1. Los mejores resultados en términos de tasa de aciertos y de F-score están en negrita. Fuente: elaboración propia



(a) Resultados en términos de tasa de aciertos

(b) Resultados en términos de f-score

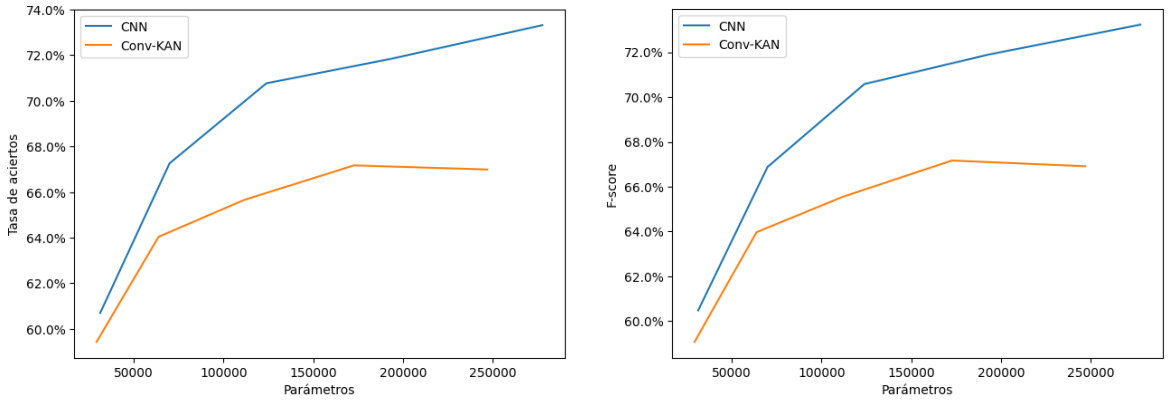
Figura 5.5: Resultados del experimento de eficiencia respecto al número de parámetros para el dataset MNIST, en términos de tasa de aciertos (a) y f-score (b) frente al número de parámetros de los modelos entrenados de las arquitecturas CNN y Conv-KAN. Fuente: elaboración propia

5.3.2 CIFAR-10

Los resultados obtenidos tras entrenar los modelos descritos en la tabla 5.1 para el dataset CIFAR-10 se pueden ver en la tabla 5.3. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.6.

Modelo	Parámetros totales	Tasa de aciertos	F-score
CKAN8	29330	59.4%	59.1%
CKAN12	63862	64.0%	64.0%
CKAN16	111642	65.7%	65.5%
CKAN20	172670	67.2%	67.2%
CKAN24	246946	67.0%	66.9%
CNN10	31350	60.7%	60.5%
CNN15	69970	67.3%	66.9%
CNN20	123890	70.8%	70.6%
CNN25	193110	71.8%	71.9%
CNN30	277630	**73.3%**	**73.2%**

Tabla 5.3: Resultados obtenidos para el dataset CIFAR-10 en el experimento de eficiencia respecto al número de parámetros tras entrenar los modelos descritos en la tabla 5.1. Los mejores resultados en términos de tasa de aciertos y de F-score están en negrita. Fuente: elaboración propia



(a) Resultados en términos de tasa de aciertos

(b) Resultados en términos de f-score

Figura 5.6: Resultados del experimento de eficiencia respecto al número de parámetros para el dataset CIFAR-10, en términos de tasa de aciertos (a) y f-score (b) frente al número de parámetros de los modelos entrenados de las arquitecturas CNN y Conv-KAN. Fuente: elaboración propia

5.3.3 Resultados

Como se ha podido ver en los resultados anteriores, las redes KAN son menos eficientes en términos de parámetros que las redes convolucionales tradicionales. Esta deficiencia ocurre en todos los conjuntos de datos probados, y es bastante significativa, obteniendo las redes convolucionales KAN un rendimiento consistentemente menor que las redes CNN para un

número similar de parámetros, tanto en términos de tasa de aciertos como en términos de f-score.

Esto se debe, principalmente, a la gran cantidad de parámetros que necesitan las redes KAN para definir las capas densas, ya que las redes KAN necesitan mayor cantidad de parámetros para cada neurona. En una red tradicional, una capa densa de N entradas y M salidas necesitará $O(n\,m)$ parámetros. No obstante, en un capa KAN, una capa densa necesitará una cantidad de parámetros proporcional a la cantidad de nodos de la grid g, por lo que esta capa necesitará $O(n\,m\,g)$ parámetros. Aunque el valor g no suele ser sea demasiado alto (generalmente está entre 4 y 32), esto sigue significando que las redes KAN suelen utilizar mucho más parámetros para la misma arquitectura que la red tradicional equivalente. Es por esto que, para compararlas en términos de parámetros, se ha tenido que reducir la arquitectura de las redes KAN (especialmente en la capa lineal), además de utilizar un tamaño de grid bastante reducido (4), cosa que ha empeorado bastante el rendimiento de los modelos utilizados en este experimento.

5.4 Eficiencia respecto al número de datos de entrenamiento

Como ya se ha explicado en el apartado 2.5.4.2, a la hora de entrenar modelos de machine learning es preferible utilizar arquitecturas y modelos que necesiten una menor cantidad de datos de entrenamiento para obtener resultados buenos. Para intentar medir esta propiedad, entrenaremos los modelos utilizando únicamente una fracción de los datos de entrenamiento del los conjuntos de datos y estudiaremos los resultados obtenidos. Como cada modelo ha de ser entrenado varias veces, solo realizaremos el experimento con 2 modelos CNN y 2 modelos Conv-KAN. Estos modelos están descritos en la tabla 5.4.

Arquitectura	Modelo	C_1	C_2	D
Conv-KAN	CKAN8	8	16	32
	CKAN16	16	32	64
CNN	CNN8	8	16	32
	CNN16	16	32	64

Tabla 5.4: Arquitecturas Conv-KAN y CNN entrenadas para realizar el experimento de eficiencia de datos de entrenamiento, siendo C_1 la cantidad de filtros de la primera capa convolucional, C_2 la cantidad de filtros de la segunda capa convolucional, y D la cantidad de neuronas de la primera capa densa. Fuente: elaboración propia

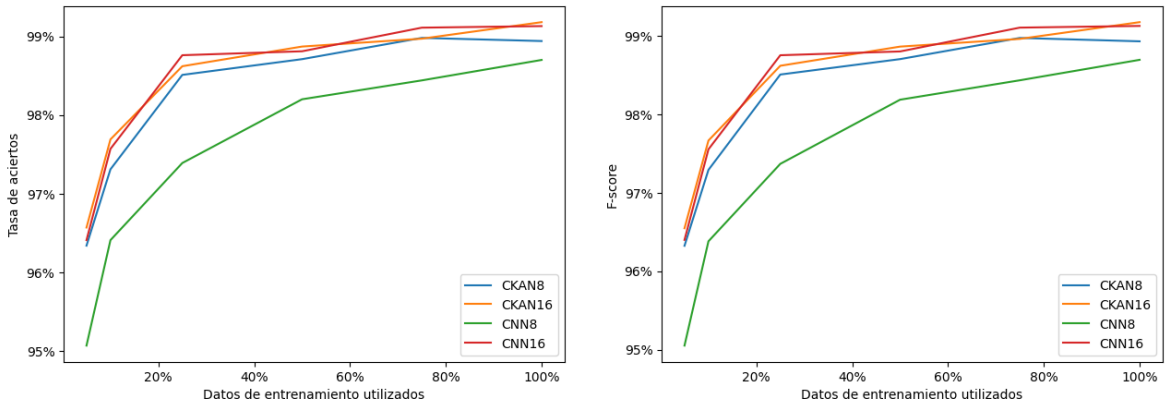
Para cada modelo de la tabla anterior, se va a entrenar el modelo utilizando 5, 10, 25, 50, 75 o 100% de los datos de entrenamiento totales. Para cada uno de estos entrenamientos, se medirá el rendimiento obtenido tras el entrenamiento en términos de tasa de aciertos y de f-score, con el fin de medir cuál arquitectura es más eficiente en términos de datos de entrenamiento.

5.4.1 MNIST

Los resultados obtenidos tras entrenar los modelos descritos en la tabla 5.4 para el dataset MNIST se pueden ver en la tabla 5.5. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.7.

Modelo	Métrica	Datos de entrenamiento utilizados (%)					
		5%	10%	25%	50%	75%	100%
CKAN8	Tasa de aciertos	96.3%	97.3%	98.5%	98.7%	99.0%	98.9%
	F-score	96.3%	97.2%	98.5%	98.7%	99.0%	98.9%
CKAN16	Tasa de aciertos	**96.6%**	**97.7%**	98.6%	**98.9%**	99.0%	**99.2%**
	F-score	**96.5%**	**97.7%**	98.6%	**98.9%**	99.0%	**99.2%**
CNN8	Tasa de aciertos	95.1%	96.4%	97.4%	98.2%	98.4%	98.7%
	F-score	95.0%	96.4%	97.4%	98.2%	98.4%	98.7%
CNN16	Tasa de aciertos	96.4%	97.6%	**98.8%**	98.8%	**99.1%**	99.1%
	F-score	96.4%	97.5%	**98.7%**	98.8%	**99.1%**	99.1%

Tabla 5.5: Resultados obtenidos para el dataset MNIST en el experimento de eficiencia respecto al número de datos de entrenamiento, tras entrenar los modelos descritos en la tabla 5.4 con un 5, 10, 25, 50, 75 y 100% de las muestras del conjunto de datos de entrenamiento. Los mejores resultados en términos de tasa de aciertos y f-score para cada porcentaje de datos de entrenamiento se han resaltado en negrita. Fuente: elaboración propia



(a) Resultados en términos de tasa de aciertos

(b) Resultados en términos de f-score

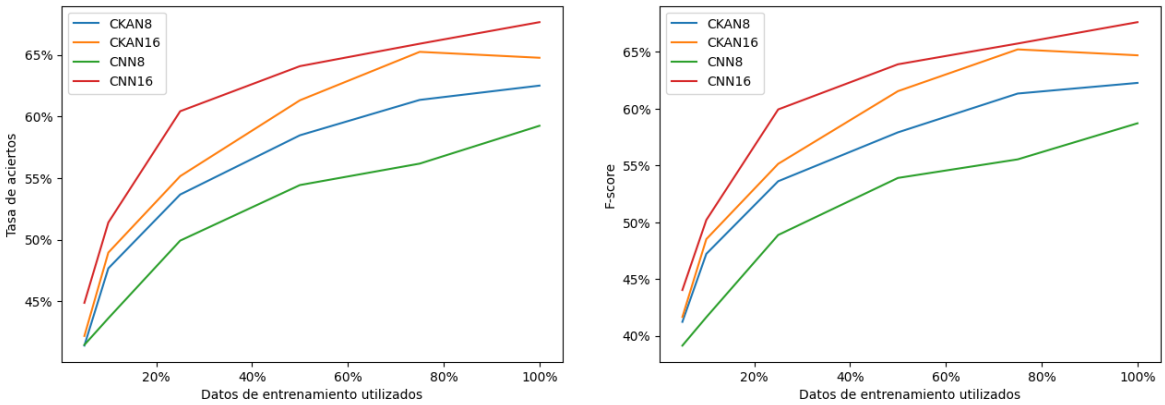
Figura 5.7: Resultados obtenidos para el dataset MNIST en el experimento de eficiencia respecto a la cantidad de datos de entrenamiento, mostrando el rendimiento de los modelos obtenidos en términos de tasa de aciertos (a) y f-score (b) frente al porcentaje de datos utilizado durante el entrenamiento. Fuente: elaboración propia

5.4.2 CIFAR-10

Los resultados obtenidos tras entrenar los modelos descritos en la tabla 5.4 para el dataset CIFAR-10 se pueden ver en la tabla 5.6. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.8.

Modelo	Métrica	Datos de entrenamiento utilizados (%)					
		5%	10%	25%	50%	75%	100%
CKAN8	Tasa de aciertos	41.4%	47.7%	53.7%	58.5%	61.3%	62.5%
	F-score	41.2%	47.2%	53.6%	57.9%	61.3%	62.3%
CKAN16	Tasa de aciertos	42.2%	49.0%	55.2%	61.3%	65.2%	64.8%
	F-score	41.7%	48.5%	55.1%	61.5%	65.2%	64.7%
CNN8	Tasa de aciertos	41.5%	43.6%	49.9%	54.4%	56.2%	59.2%
	F-score	39.1%	41.6%	48.9%	53.9%	55.5%	58.7%
CNN16	Tasa de aciertos	**44.9%**	**51.4%**	**60.4%**	**64.1%**	**65.9%**	**67.7%**
	F-score	**44.0%**	**50.2%**	**59.9%**	**63.9%**	**65.7%**	**67.6%**

Tabla 5.6: Resultados obtenidos para el dataset CIFAR-10 en el experimento de eficiencia respecto al número de datos de entrenamiento, tras entrenar los modelos descritos en la tabla 5.4 con un 5, 10, 25, 50, 75 y 100% de las muestras del conjunto de datos de entrenamiento. Los mejores resultados en términos de tasa de aciertos y f-score para cada porcentaje de datos de entrenamiento se han resaltado en negrita. Fuente: elaboración propia



(a) Resultados en términos de tasa de aciertos **(b)** Resultados en términos de f-score

Figura 5.8: Resultados obtenidos para el dataset CIFAR-10 en el experimento de eficiencia respecto a la cantidad de datos de entrenamiento, mostrando el rendimiento de los modelos obtenidos en términos de tasa de aciertos (a) y f-score (b) frente al porcentaje de datos utilizado durante el entrenamiento. Fuente: elaboración propia

5.4.3 Resultados

Como se puede ver en la tabla 5.5, en algunos casos para el dataset MNIST las redes convolucionales KAN obtienen un mejor rendimiento que las redes convolucionales tradicionales dada la misma fracción del conjunto de datos de entrenamiento del dataset MNIST. No obstante, aunque en algunos casos se haya obtenido un rendimiento mayor, los resultados entre los modelos CKAN16 y CNN16 son tan similares que no se puede concluir con certeza que las redes convolucionales tengan un mayor rendimiento para el dataset MNIST en términos de eficiencia de datos de entrenamiento para ninguna de las fracciones de datos comprobadas. La única cosa que se observa de forma clara es que el modelo CNN8 obtiene resultados significativamente peores que el resto de modelos, incluyendo el modelo CKAN8, cosa que puede indicar que las redes convolucionales KAN tengan una mayor eficiencia en términos de datos de entrenamiento para modelos de menor tamaño.

En los resultados para CIFAR-10, podemos ver que aunque los modelos CKAN16 y CKAN8 obtienen resultados bastante mejores que el modelo CNN8, el modelo CNN16 obtiene un mayor rendimiento que ambos modelos Conv-KAN para todas las fracciones de datos de entrenamiento comprobadas. Aunque es posible que las redes KAN obtengan resultados con mayor rendimiento para modelos reducidos, este resultado indica claramente que la ventaja que existe no necesariamente se generaliza a modelos de mayor tamaño, hasta el punto de que las redes convolucionales tradicionales parecen tener mejor rendimiento que las redes convolucionales KAN para modelos de mayor tamaño.

5.5 Calibración

La calibración de un modelo indica la correlación entre la confianza producida por un modelo y la probabilidad real de que una muestra sea correcta (ver apartado 2.5.5) para más detalles. Dado que obtener modelos con buena calibración puede ser muy importante dependiendo de la aplicación del caso de uso del modelo, vamos a medir la calibración de modelos de arquitecturas Conv-KAN y compararla con la calibración obtenida por los modelos convolucionales tradicionales. Se medirá la calibración de los modelos descritos en la tabla 5.7.

Arquitectura	Modelo	C_1	C_2	D
Conv-KAN	CKAN8	8	16	32
	CKAN16	16	32	64
CNN	CNN8	8	16	32
	CNN16	16	32	64

Tabla 5.7: Arquitecturas Conv-KAN y CNN entrenadas para realizar el experimento de calibración, siendo C_1 la cantidad de filtros de la primera capa convolucional, C_2 la cantidad de filtros de la segunda capa convolucional, y D la cantidad de neuronas de la primera capa densa. Fuente: elaboración propia

Para cada uno de los 4 modelos de la tabla 5.7, mediremos la calibración obtenida tras el entrenamiento, utilizando tanto el Error de Calibración Esperado (ECE) como la tasa de

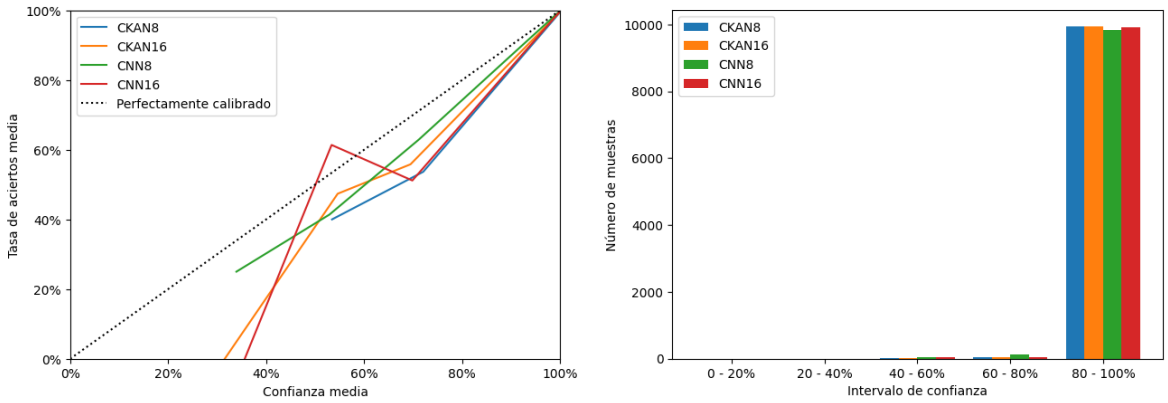
aciertos respecto al intervalo de confianza (ver apartados 2.5.5.1 y 2.5.5.2). Para medir y calcular ambas métricas se han dividido las muestras de los conjuntos de datos de evaluación en 5 grupos uniformes en función de la confianza predecida por el modelo para cada muestra, de forma que se han agrupado las muestras en los grupos de $0-20\%$, $20-40\%$, $40-60\%$, $60-80\%$ y $80-100\%$ en función de su confianza.

5.5.1 MNIST

Los resultados obtenidos tras entrenar los modelos descritos en la tabla 5.1 para el dataset MNIST se pueden ver en la tabla 5.2. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.5.

Modelo	ECE	Tasa de aciertos respecto al intervalo de confianza				
		$0-20\%$	$20-40\%$	$40-60\%$	$60-80\%$	$80-100\%$
CKAN8	0.70%	—	—	40.0%	53.7%	99.3%
CKAN16	0.51%	—	0.0%	47.4%	55.8%	99.5%
CNN8	**0.39%**	—	25.0%	41.4%	62.9%	99.5%
CNN16	0.44%	—	0.0%	61.4%	51.2%	99.5%

Tabla 5.8: Resultados obtenidos en el experimento de calibración para el dataset MNIST, mostrando el ECE obtenido de cada modelo junto con la tasa de aciertos obtenida respecto a la probabilidad predecida media. Para el cálculo del ECE y de la tasa de aciertos respecto a la confianza se han agrupado las muestras en 5 grupos, con intervalos de confianzas $0-20\%$, $20-40\%$, $40-60\%$, $60-80\%$ y $80-100\%$. Las celdas vacías de la tabla indican que no hay ninguna muestra en su intervalo de confianza para el modelo correspondiente. Fuente: elaboración propia



(a) Curvas de calibración **(b)** Cantidad de muestras

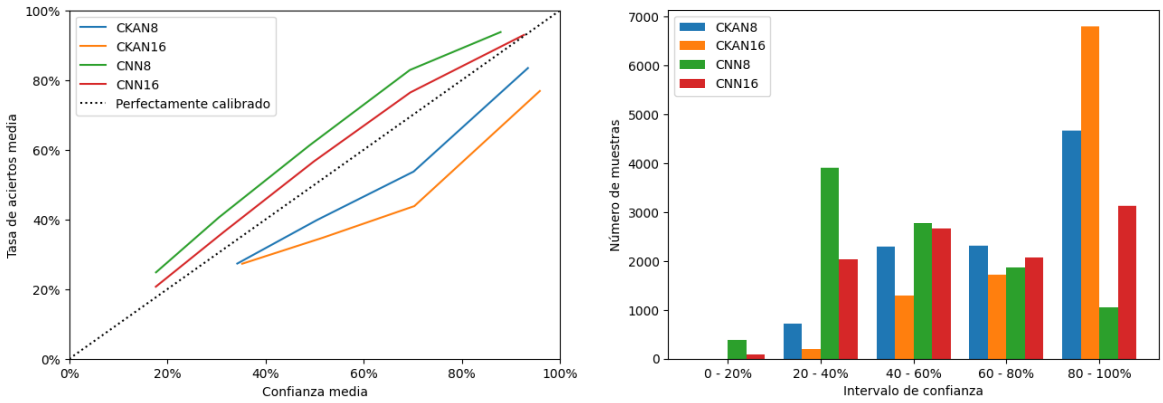
Figura 5.9: Gráficas que visualizan los resultados del experimento de calibración para el dataset MNIST, mostrando las curvas de calibración obtenidas para cada modelo junto con la cantidad de muestras que pertenecen a cada intervalo, tras agrupar las muestras en 5 grupos, con intervalos de confianzas $0-20\%$, $20-40\%$, $40-60\%$, $60-80\%$ y $80-100\%$. En la gráfica de curvas de calibración también se muestra la línea de calibración óptima, en la que la tasa de aciertos media de un modelo coincide con la confianza producida. Fuente: elaboración propia

5.5.2 CIFAR-10

Los resultados obtenidos tras entrenar los modelos descritos en la tabla 5.1 para el dataset CIFAR-10 se pueden ver en la tabla 5.3. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.6.

Modelo	ECE	Tasa de aciertos respecto al intervalo de confianza				
		$0-20\%$	$20-40\%$	$40-60\%$	$60-80\%$	$80-100\%$
CKAN8	11.42%	—	27.3%	39.8%	53.7%	83.4%
CKAN16	19.81%	—	27.3%	34.8%	43.8%	76.9%
CNN8	10.78%	24.8%	40.6%	61.3%	82.9%	93.8%
CNN16	**4.38%**	20.7%	36.3%	56.7%	76.4%	92.9%

Tabla 5.9: Resultados obtenidos en el experimento de calibración para el dataset CIFAR-10, mostrando el ECE obtenido de cada modelo junto con la tasa de aciertos obtenida respecto a la probabilidad predecida media. Para el cálculo del ECE y de la tasa de aciertos respecto a la confianza se han agrupado las muestras en 5 grupos, con intervalos de confianzas $0-20\%$, $20-40\%$, $40-60\%$, $60-80\%$ y $80-100\%$. Las celdas vacías de la tabla indican que no hay ninguna muestra en su intervalo de confianza para el modelo correspondiente. Fuente: elaboración propia



(a) Curvas de calibración **(b)** Cantidad de muestras

Figura 5.10: Gráficas que visualizan los resultados del experimento de calibración para el dataset CIFAR-10, mostrando las curvas de calibración obtenidas para cada modelo junto con la cantidad de muestras que pertenecen a cada intervalo, tras agrupar las muestras en 5 grupos, con intervalos de confianzas $0-20\%$, $20-40\%$, $40-60\%$, $60-80\%$ y $80-100\%$. En la gráfica de curvas de calibración también se muestra la línea de calibración óptima, en la que la tasa de aciertos media de un modelo coincide con la confianza producida. Fuente: elaboración propia

5.5.3 Resultados

Los resultados obtenidos con el dataset MNIST, aunque todos los modelos obtienen calibración muy buena, están muy distorsionados dado el rendimiento de los modelos, obteniendo todos un tasa de aciertos media superior al 98%. Esto, como se puede ver en la figura 5.9, causa una tremenda distorsión en las muestras, haciendo que la gran mayoría estén en un único intervalo de confianza.

Es por esto que, como los resultados en MNIST no van a ser demasiado indicativos, vamos a centrarnos especialmente en los resultados para el dataset CIFAR-10. En estos resultados se puede ver que obtenemos curvas de calibración mucho más estables, junto con una distribución de muestras mucho más equilibrada en cada intervalo de confianza. A partir de estos resultados, podemos ver que los modelos Conv-KAN tienden a producir una mayor confianza que la tasa de aciertos, mientras que los modelos CNN tienden a producir una menor confianza que tasa de aciertos para el dataset CIFAR-10. No obstante, podemos ver que las curvas de calibración de los modelos CNN son mucho más cercanas a la línea ideal de calibración que las curvas de los modelos Conv-KAN. Además, se puede observar en la tabla 5.9 que los modelos CNN también tienen un error de calibración esperado (ECE) mucho menor que los modelos Conv-KAN.

5.6 Aprendizaje continuo

Actualmente, los modelos de inteligencia artificial no tienden mucha capacidad para retener información ya aprendida si no se refuerza continuamente esta información en el entrenamiento (ver apartado 2.5.6 para más información). No obstante, las redes KAN formadas

exclusivamente por capas densas KAN son capaces de en ciertos casos retener bastante bien la información aprendida previamente [6]. Para comprobar si esta propiedad se extiende a las redes convolucionales KAN, diseñaremos un experimento con el fin de intentar comparar las capacidades de realizar aprendizaje continuo y de retener información de las redes convolucionales KAN frente a las redes CNN. Los modelos utilizados de ambas arquitecturas se pueden ver en la tabla 5.10.

Arquitectura	Modelo	C_1	C_2	D
Conv-KAN	CKAN8	8	16	32
	CKAN16	16	32	64
CNN	CNN8	8	16	32
	CNN16	16	32	64

Tabla 5.10: Modelos Conv-KAN y CNN utilizados para el experimento de aprendizaje continuo, siendo C_1 la cantidad de filtros de la primera capa convolucional, C_2 la cantidad de filtros de la segunda capa convolucional, y D la cantidad de neuronas de la primera capa densa. Fuente: elaboración propia

El experimento de aprendizaje continuo consiste en entrenar los modelos en fases, de manera que en cada fase solo reciben un subconjunto de las clases totales a aprender. Como MNIST y CIFAR-10 tienen 10 clases, dividiremos el entrenamiento en 5 fases, de forma que el modelo será entrenado con 2 de las 10 clases en cada fase del entrenamiento. Para evaluar lo bien que retiene información tras cada fase el modelo se ha evaluado con las muestras de evaluación de todas las clases ya vistas, incluyendo las clases de fases anteriores. Se puede ver las clases exactas utilizadas para el entrenamiento y la evaluación para cada fase en la tabla 5.11.

Fase	Clases de entrenamiento	Clases de evaluación
Fase 1	0 y 1	0 y 1
Fase 2	2 y 3	$0-3$
Fase 3	4 y 5	$0-5$
Fase 4	6 y 7	$0-7$
Fase 5	8 y 9	$0-9$ (todas)

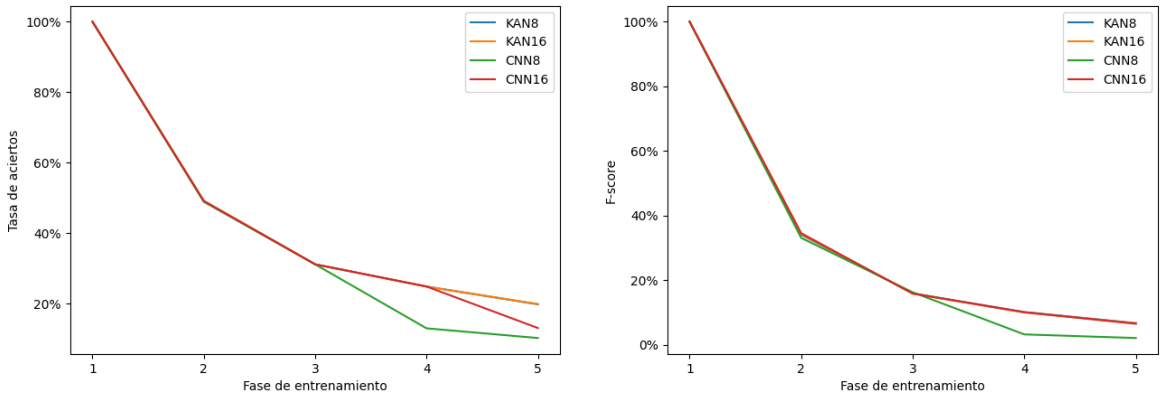
Tabla 5.11: Clases utilizadas para el entrenamiento y evaluación en cada fase para el experimento de aprendizaje continuo. En cada fase se ha entrenado el modelo con dos clases no vistas anteriormente, y se ha evaluado el modelo con todas las clases de esa fase y de las fases anteriores, de forma que al llegar a la última fase el modelo se evalúa con todas las clases. Fuente: elaboración propia

5.6.1 MNIST

Los resultados obtenidos tras entrenar los modelos descritos en la tabla 5.10 para el dataset MNIST se pueden ver en la tabla 5.12. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.11.

Modelo	Métrica	Fase de entrenamiento				
		Fase 1	Fase 2	Fase 3	Fase 4	Fase 5
CKAN8	Tasa de aciertos	**99.9%**	49.0%	**31.1%**	**24.8%**	19.7%
	F-score	**99.9%**	34.0%	15.9%	**10.1%**	**6.7%**
CKAN16	Tasa de aciertos	**99.9%**	**49.1%**	**31.1%**	24.7%	**19.8%**
	F-score	**99.9%**	34.3%	15.8%	10.0%	6.6%
CNN8	Tasa de aciertos	99.8%	48.8%	31.0%	12.9%	10.2%
	F-score	99.8%	33.0%	**16.2%**	3.2%	2.1%
CNN16	Tasa de aciertos	**99.9%**	49.0%	**31.1%**	**24.8%**	13.0%
	F-score	**99.9%**	**34.5%**	15.8%	10.0%	6.5%

Tabla 5.12: Resultados obtenidos en el experimento de aprendizaje continuo para el conjunto de datos MNIST tras cada una de las 5 fases de entrenamiento descritas en la tabla 5.11. Los mejores resultados obtenidos en términos de tasa de aciertos y de f-score para cada fase de entrenamiento están en negrita. Fuente: elaboración propia



(a) Resultados en términos de tasa de aciertos **(b)** Resultados en términos de f-score

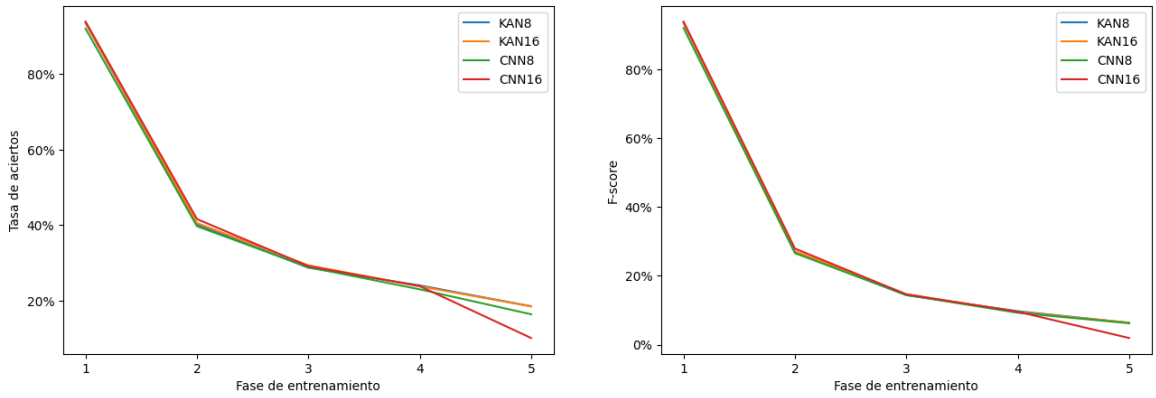
Figura 5.11: Visualización de los resultados obtenidos en el experimento de aprendizaje continuo para el conjunto de datos MNIST para cada uno de los modelos entrenados. Los resultados se han visualizado en términos de tasa de aciertos (a) y de f-score (b) para cada una de las 5 fases de entrenamiento descritas en la tabla 5.11. Fuente: elaboración propia

5.6.2 CIFAR-10

Los resultados obtenidos tras entrenar los modelos descritos en la tabla 5.10 para el dataset CIFAR-10 se pueden ver en la tabla 5.13. Se han representado los resultados obtenidos respecto a la tasa de aciertos y respecto a la f-score en la figura 5.12.

Modelo	Métrica	Fase de entrenamiento				
		Fase 1	Fase 2	Fase 3	Fase 4	Fase 5
CKAN8	Tasa de aciertos	93.6%	40.2%	28.7%	24.0%	**18.5%**
	F-score	93.5%	26.9%	14.4%	9.7%	**6.4%**
CKAN16	Tasa de aciertos	93.6%	40.5%	**29.3%**	23.8%	18.4%
	F-score	93.5%	27.2%	**14.7%**	9.5%	6.4%
CNN8	Tasa de aciertos	92.1%	39.7%	28.8%	23.0%	16.3%
	F-score	92.0%	26.6%	14.5%	9.3%	6.2%
CNN16	Tasa de aciertos	**93.9%**	**41.6%**	29.1%	**23.8%**	10.1%
	F-score	**93.9%**	**27.9%**	14.6%	**9.6%**	1.9%

Tabla 5.13: Resultados obtenidos en el experimento de aprendizaje continuo para el conjunto de datos CIFAR-10 tras cada una de las 5 fases de entrenamiento descritas en la tabla 5.11. Los mejores resultados obtenidos en términos de tasa de aciertos y de f-score para cada fase de entrenamiento están en negrita. Fuente: elaboración propia



(a) Resultados en términos de tasa de aciertos

(b) Resultados en términos de f-score

Figura 5.12: Visualización de los resultados obtenidos en el experimento de aprendizaje continuo para el conjunto de datos CIFAR-10 para cada uno de los modelos entrenados. Los resultados se han visualizado en términos de tasa de aciertos (a) y de f-score (b) para cada una de las 5 fases de entrenamiento descritas en la tabla 5.11. Fuente: elaboración propia

5.6.3 Resultados

Como se puede ver, en ambos conjuntos de datos se observa que las redes convolucionales KAN suelen producir mejores resultados en las fases posteriores del entrenamiento. Aunque esto pueda significar que las redes Conv-KAN tengan mejor capacidad de retención de información que las redes CNN, no se observa una retención de información significativa, como se ha observado en redes KAN no convolucionales. Por lo tanto, los experimentos realizados parecen indicar que las redes KAN pierden su capacidad de retener información aprendida anteriormente al utilizar capas KAN convolucionales.

6 Conclusiones

Como se ha podido ver durante el desarrollo de este trabajo, las redes Kolmogórov-Arnold son un tipo de red neuronal fundamentalmente diferente a las redes neuronales tradicionales. Se ha visto como se forman las capas densas y convolucionales KAN a partir de la composición de funciones base, explicando toda la teoría matemática necesaria para entender su funcionamiento correctamente. Además, también se han estudiado las diferentes estructuras matemáticas existentes para formar la base de las redes KAN, aunque durante el desarrollo del trabajo nos hemos centrado principalmente en las splines, dado que son las estructuras utilizadas más frecuentemente en las redes KAN, además también de ser las utilizadas en el trabajo original que propuso las redes KAN.

Con los experimentos, se ha intentado verificar si muchas de las propiedades que tienen las redes KAN densas se mantienen a la hora de usar redes KAN convolucionales. En estos experimentos, se ha cuantificado la eficiencia respecto al número de parámetros del modelo, la eficiencia respecto al número de datos de entrenamiento, la calibración y la capacidad de retención de información en tareas de aprendizaje continuo del modelo. Tras medir estas propiedades para varios modelos convolucionales KAN y compararlas con modelos convolucionales tradicionales, no se han observado muchas de las propiedades observadas en las redes KAN densas.

Aunque este tipo de redes aún son experimentales, son un tipo de arquitectura muy novedosa e interesante, que tiene potencial para revolucionar el campo si ciertos aspectos de la arquitectura KAN se generalizan a modelos más grandes. Son de especial importancia la capacidad de retención de información en tareas de aprendizaje continuo y la interpretabilidad de los modelos, que son dos propiedades que carecen los modelos contemporáneos de machine learning. Dado lo recientes que son este tipo de redes y que aún la gran mayoría de experimentos se han realizado con modelos de pequeño tamaño, habrá que ver si estas propiedades se pueden utilizar con modelos de mayor tamaño.

6.1 Futuras líneas de investigación

Durante la realización de este trabajo se han analizado y estudiado muchas de las aplicaciones y propiedades de las redes Kolmogórov-Arnold. No obstante, principalmente por la falta de recursos de computación, hemos tenido que limitar los experimentos a una única faceta de las redes KAN: las redes KAN convolucionales.

A continuación se enumeran algunas de las aplicaciones potenciales de las redes KAN que no se han analizado o explicado, junto con un pequeño resumen de su posible utilidad:

- Analizar redes KAN basadas en otras estructuras matemáticas: aunque existan muchas estructuras matemáticas que se puedan aplicar a las redes KAN, no existe mucha literatura analizando las ventajas y desventajas específicas de cada una, y que mida el rendimiento de cada una para varias aplicaciones.

- Probar otras arquitecturas Conv-KAN: en los experimentos de este trabajo se ha utilizado una arquitectura Conv-KAN basada en las redes CNN tradicionales, estructurando las capas de forma similar a las redes convolucionales basadas en capas MLP. Es posible que otras estructuras, como el uso exclusivo de capas convolucionales o la incorporación de otras técnicas vistas en otros tipos de redes puedan mejorar el rendimiento y/o la eficiencia obtenidos.

- Redes KAN recurrentes: Al igual que las redes KAN se pueden extender a las redes convolucionales, también se pueden extender a las redes recurrentes o RNNs. Como actualmente no existen implementaciones de redes KAN recurrentes, se necesitaría adaptar las arquitecturas LSTM [81] o BRNN [82] a las redes KAN para comprobar su funcionamiento. Esto, además, permitiría la construcción de redes KAN convolucinales-recurrentes, o redes CRNN que utilicen capas KAN para su funcionamiento. No obstante, como no existen implementaciones o estudios anteriores, sería necesario elaborar una implementación para comprobar las propiedades de las redes KAN recurrentes.

- Transformers KAN: Al igual que para las redes recurrentes, también es posible extender la arquitectura KAN a los transformers. A diferencia que para las redes recurrentes, si que existen estudios e implementaciones de transformers KAN [83, 84, 85]. Igualmente, como la literatura para este tipo de arquitecturas es muy limitada, se podría realizar un estudio comparativo similar al realizado en este trabajo para las redes convolucionales KAN, analizando el potencial de las arquitecturas que combinan las redes KAN con los transformers.

- Estudiar como adaptar las capas Dropout para su funcionamiento en las redes KAN, para que funcionen de forma efectiva en las arquitecturas que utilicen capas KAN densas o convolucionales [80].

- Estudiar el uso de las redes KAN para otras tareas de inteligencia artificial, como tareas de regresión numérica, detección de objetos, segmentación, etc.; explorando como se comportan las redes KAN a la hora de ser entrenadas para realizar este tipo de tareas y de si las propiedades comúnmente asociadas con las redes KAN se mantienen en este tipo de tareas.

- Estudiar formas alternativas de entrenar redes KAN para aprovechar al máximo sus propiedades, como el uso de grid-extension estático o dinámico [6], o el uso de algoritmos de entrenamiento diseñados específicamente para las redes KAN, como su construcción utilizando el método de Newton-Kaczmarz [86].

6.2 Cumplimiento de objetivos

En cuanto a los objetivos del trabajo, podemos concluir lo siguiente:

- **Hacer una revisión de la base teórica de las redes KAN**: se ha realizado una explicación de toda la base teórica y matemática de las redes KAN, explicando los conocimientos necesarios para poder entender su funcionamiento, incluyendo el conocimiento de machine learning e inteligencia artificial necesario para entender las redes

KAN y como se diferencian de las redes neuronales tradicionales. Además de esto, se han estudiado ciertas técnicas de entrenamiento específicas a las redes KAN y las muchas alternativas que existen para implementar la estructura interna de las capas de las redes KAN.

- **Hacer una implementación propia de un modelo KAN**: se ha realizado una implementación de un modelo KAN en el apartado 4, con el fin de ver como se podría llegar a realizar una implementación de las arquitecturas KAN. Se ha decidido realizar una implementación sencilla y sin muchas opciones a propósito, para poder mostrar las ideas y las estructuras requeridas para llegar a realizar una implementación claramente, al no necesitar complicar demasiado el código al realizar una implementación más general. Para que la implementación sea razonablemente eficiente, aunque no se han realizado optimizaciones que compliquen demasiado el código, se han utilizado estructuras de `numpy` [67] siempre que ha sido posible con el fin de calcular eficientemente los parámetros de la red.

- **Evaluar y comparar el rendimiento ante alternativas**: en el apartado 5 se han realizado múltiples experimentos con el fin de confirmar y comprobar el rendimiento y la eficiencia de las redes KAN convolucionales, comparando los resultados de la redes KAN convolucionales con los resultados obtenidos por las redes convolucionales tradicionales. Para poder entender todo este proceso, en el apartado 2.5 se han explicado muchas de las técnicas de evaluación de modelos utilizadas en los experimentos, prestando especial atención a las métricas utilizadas en tareas de clasificación de imágenes, ya que este tipo de tareas son las que se han utilizado para realizar los experimentos.

- **Hacer un análisis de ventajas y limitaciones de las redes KAN**: en muchas de las secciones del capítulo 3, se han mencionado muchas de las ventajas inherentes a las redes KAN, citando estudios previos que han investigado en detalle estas propiedades de las redes KAN, como su mayor eficiencia a la hora de predecir ciertos datos, su mayor robustez en tareas de aprendizaje continuo o su interpretabilidad. Además, también se han enumerado las múltiples limitaciones, como la cantidad de parámetros que requieren, el tiempo requerido para su uso y entrenamiento, o que algunas de las técnicas estándar de machine learning, como el dropout, no se pueden aplicar a las redes KAN

- **Proponer líneas futuras de investigación**: en la sección 6.1 se han enumerado las muchas posibles líneas de investigación posibles. Como las redes KAN son una arquitectura muy reciente, existen muchas líneas de investigación interesantes, como la implementación de las redes KAN en otros tipos de arquitecturas (como los transformers o las redes recurrentes) o el estudio de posibles técnicas de machine learning específicas a las redes KAN

6.3 Conclusiones personales

Durante la realización de este trabajo, ha sido necesario aprender sobre un tema novedoso y poco investigado. Esto ha requerido una indagación diligente de las pocas publicaciones sobre

el tema, además de gran cantidad de prueba y error a la hora de realizar la implementación propia y los experimentos. No obstante, aunque ha sido un trabajo bastante extenso y largo, considero que he aprendido muchísimo durante la realización del mismo.

Lo primero, gracias a este trabajo he reforzado profundamente mis conocimientos de inteligencia artificial. Al tener que explicar muchos de estos conceptos fundamentales de forma breve pero práctica, he tenido que indagar a cerca de una gran cantidad de estos conocimientos, aprendiendo mucho al repasar y revisar todas las ideas fundamentales de la inteligencia artificial. Especialmente, al intentar explicar el algoritmo de retropropagación de las redes neuronales, he tenido que aprender bastantes detalles para poder realizar una explicación detalla e intuitiva del algoritmo.

Más importante aún, para poder realizar este trabajo he necesitado aprender sobre las redes KAN. Uno de los principales problemas ha sido la falta de información a cerca del tema, al existir una cantidad muy limitada de publicaciones y de información disponible, dado lo reciente que es la arquitectura KAN. Esto es en parte lo que ha hecho que el trabajo sea tan satisfactorio e interesante, ya que investigar y aprender a cerca de un tema tan novedoso de la inteligencia artificial no es algo que haya hecho anteriormente.

Las principales dificultades del trabajo han ocurrido a la hora de realizar la implementación propia en Python. Dediqué una gran cantidad de tiempo a perfeccionarla y a corregir numerosos problemas que habían surgido, ya que durante todo su desarrollo la implementación ha presentado múltiples problemas constantes, la mayoría causados por la falta de información al respecto. Esto me ha obligado a experimentar mucho durante la realización de la implementación, probando múltiples alternativas y opciones hasta resolver los problemas existentes. Ha sido mediante la corrección de estos problemas y mediante el perfeccionamiento de la implementación que he aprendido muchos de los detalles específicos de las redes KAN. Estos detalles no solo me han ayudado a reforzar mi intuición y conocimiento a cerca de las redes KAN, si no que me han permitido realizar una mejor explicación de las mismas durante el desarrollo del trabajo.

Durante los experimentos, también han surgido una multitud de problemas, especialmente causados por lo experimentales y recientes que son muchas de las implementaciones de las redes KAN, especialmente teniendo en cuenta que se han utilizado redes KAN convolucionales. A parte de reforzar mis conocimientos de `pytorch`, he aprendido mucho a cerca de como implementar modelos personalizados KAN. Esto ha sido en gran parte por que he necesitado en múltiples ocasiones leer y analizar el código fuente de las implementaciones KAN utilizadas en los experimentos, ya que muchas de estas implementaciones tienen una documentación muy poco detallada, o incluso en algunos casos no disponen de ninguna documentación.

Incluso teniendo en cuenta todos los problemas y dificultades, considero que el trabajo ha sido una experiencia muy enriquecedora para mis conocimientos de inteligencia artificial y machine learning, ya que me ha permitido reforzar muchas de las ideas y conceptos fundamentales de estos campos. Además, la realización del trabajo me ha permitido aprender mucho a cerca de un tema muy interesante y novedoso, y me ha ofrecido una nueva perspectiva a cerca del desarrollo e implementación de modelos de inteligencia artificial.

Bibliografía

[1] L. K. Boran and A. H. Joya, "From Pixels to Predictions: A Comprehensive Survey of Image Classification," *International Journal for Research in Applied Science & Engineering Technology*, vol. 12, no. 11, 2024.

[2] L. Qin, Q. Chen, X. Feng, Y. Wu, Y. Zhang, Y. Li, M. Li, W. Che, and P. S. Yu, "Large Language Models Meet NLP: A Survey," 2024.

[3] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee, "A Survey of Modern Deep Learning based Object Detection Models," 2021.

[4] M. T. Augustine, "A Survey on Universal Approximation Theorems," 2024.

[5] K. A. N., "On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition," *Translations American Mathematical Society*, vol. 2, no. 28, pp. 55–59, 1963.

[6] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou, and M. Tegmark, "KAN: Kolmogorov-Arnold Networks," 2024.

[7] Geeks for Geeks, "Underfitting and Overfitting." https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/. Accedido: 19/4/2025.

[8] Geeks for Geeks, "Unsupervised Learning." https://www.geeksforgeeks.org/unsupervised-learning/. Accedido: 19/4/2025.

[9] Geeks for Geeks, "Reinforcement Learning." https://www.geeksforgeeks.org/what-is-reinforcement-learning/. Accedido: 19/4/2025.

[10] D. Kriesel, "A Brief Introduction to Neural Networks," 2007.

[11] C. A. L. Bailer-Jones, R. Gupta, and H. P. Singh, "An introduction to Artificial Neural Networks," 2001.

[12] A. M. Geoffrion, "Objective function approximations in mathematical programming," *Mathematical Programming*, vol. 13, pp. 23–37, Dec 1977.

[13] Geeks for Geeks, "Activation Functions." https://www.geeksforgeeks.org/activation-functions-neural-networks/. Accedido: 19/4/2025.

[14] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," 2015.

[15] Better Explained, "Intuitive guide to convolution." https://betterexplained.com/articles/intuitive-convolution/. Accedido: 20/4/2025.

[16] Janosh Riebesell, "TikZ.net: Convolution Operator." https://tikz.net/conv2d/. Accedido: 21/2/2025.

[17] Dhanush Kumar, "Max Pooling." https://medium.com/@danushidk507/max-pooling-ef545993b6e4, 2023. Accedido: 20/4/2025.

[18] Computer Science Wiki, "MaxpoolSample2.png." https://computersciencewiki.org/index.php/File:MaxpoolSample2.png. Accedido: 20/4/2025.

[19] Geeks for Geeks, "Common Loss Functions." https://www.geeksforgeeks.org/ml-common-loss-functions/, 2025. Accedido: 20/4/2025.

[20] Brent Scarff, "Understanding Backpropagation." https://towardsdatascience.com/understanding-backpropagation-abcc509ca9d0/, 2021. Accedido: 20/4/2025.

[21] Pytorch, "torch.optim." https://docs.pytorch.org/docs/stable/optim.html. Accedido: 21/5/2025.

[22] H. Bichri, A. Chergui, and H. Mustapha, "Investigating the Impact of Train / Test Split Ratio on the Performance of Pre-Trained Models with Custom Datasets," *International Journal of Advanced Computer Science and Applications*, vol. 15, 01 2024.

[23] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning: With applications in R*, p. 176. Springer US Springer, 2013.

[24] H. Phillips, "A Simple Introduction to Softmax." https://medium.com/@hunter-j-phillips/a-simple-introduction-to-softmax-287712d69bac. Accedido: 11/5/2025.

[25] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On Calibration of Modern Neural Networks," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 1321–1330, PMLR, 06–11 Aug 2017.

[26] Geeks for Geeks, "Metrics for Machine Learning models." https://www.geeksforgeeks.org/metrics-for-machine-learning-model/. Accedido: 11/5/2025.

[27] R. Sandhu, "Class Imbalance Vs Accuracy." https://medium.com/@rajsandhu1989/class-imbalance-vs-accuracy-9739f5deece0. Accedido: 11/5/2025.

[28] S. Amarendra, "Optimizing Parameter Efficiency in Machine Learning Models: A Focus on Reducing Memory Overhead with L-BFGS-Optimized Algorithms," *Journal of Electrical Systems*, vol. 20, pp. 298–314, 04 2024.

[29] Y. Yang, H. Kang, and B. Mirzasoleiman, "Towards Sustainable Learning: Coresets for Data-efficient Deep Learning," *International Conference on Machine Learning (ICML)*, 2023.

[30] T. Y. Liu and B. Mirzasoleiman, "Data-Efficient Augmentation for Training Neural Networks," *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[31] F. Faghri, "Training Efficiency and Robustness in Deep Learning," 2021.

[32] J. Thiyagalingam, M. Shankar, G. Fox, and T. Hey, "Scientific Machine Learning benchmarks," *Nature Reviews Physics*, vol. 4, pp. 413–420, Jun 2022.

[33] C. Wang, "Calibration in Deep Learning: A Survey of the State-of-the-Art," 2024.

[34] M. Pavlovic, "Understanding Model Calibration: A gentle introduction and visual exploration of calibration and the expected calibration error (ECE)," 2025.

[35] NapsterInBlue, "Using calibration curves to pick your classifier." https://napsterinblue.github.io/notes/machine_learning/model_selection/calibration_curves/. Accedido: 11/5/2025.

[36] Gido M. van de Ven and Nicholas Soures and Dhireesha Kudithipudi, "Continual Learning and Catastrophic Forgetting," 2024.

[37] M. McCloskey and N. J. Cohen, "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem," in *Catastrophic Interference in Connectionist Networks* (G. H. Bower, ed.), vol. 24 of *Psychology of Learning and Motivation*, pp. 109–165, Academic Press, 1989.

[38] D.-W. Zhou, Q.-W. Wang, Z.-H. Qi, H.-J. Ye, D.-C. Zhan, and Z. Liu, "Class-Incremental Learning: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, p. 9851–9873, 2024.

[39] A. D. Bodner, A. S. Tepsich, J. N. Spolski, and S. Pourteau, "Convolutional kolmogorov-arnold networks," 2025.

[40] V. Starostin, "ConvKAN." https://github.com/StarostinV/convkan. Accedido: 12/5/2025.

[41] G. Farin, "8 - B-Spline Curves," in *Curves and Surfaces for CAGD (Fifth Edition)* (G. Farin, ed.), The Morgan Kaufmann Series in Computer Graphics, pp. 119–146, Morgan Kaufmann, fifth edition ed., 2002.

[42] P. Gupta, "KAN Tutorial: Splines." https://github.com/pg2455/KAN-Tutorial/blob/main/1_splines.ipynb, 2024. Accedido: 3/2/2025.

[43] Z. Li, "FastKAN." https://github.com/ZiyaoLi/fast-kan. Accedido: 12/5/2025.

[44] C. K. Shene, "De Boor's Algorithm." https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/de-Boor.html. Accedido: 20/4/2025.

[45] C.-K. Shene, "Derivatives of a B-spline Curve." https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/B-spline/bspline-derv.html. Accedido: 7/2/2025.

[46] Neil Dodgson, "B-splines." https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node4.html. Accedido: 20/4/2025.

[47] T. J. Rivlin, *Chebyshev polynomials*. Courier Dover Publications, 2020.

[48] S. SS, K. AR, G. R, and A. KP, "Chebyshev Polynomial-Based Kolmogorov-Arnold Networks: An Efficient Architecture for Nonlinear Function Approximation," 2024.

[49] Guo Dawei, "ChebyKAN." https://github.com/SynodicMonth/ChebyKAN. Accedido: 20/4/2025.

[50] Wolfram MathWorld, "Fourier Series." https://mathworld.wolfram.com/FourierSeries.html. Accedido: 20/4/2025.

[51] M. Bôcher, "Introduction to the Theory of Fourier's Series," *Annals of Mathematics*, vol. 7, no. 3, pp. 81–152, 1906.

[52] J. Zhang, Y. Fan, K. Cai, and K. Wang, "Kolmogorov-Arnold Fourier Networks," 2025.

[53] Gist Noesis, "FourierKAN." https://github.com/GistNoesis/FourierKAN/N. Accedido: 20/4/2025.

[54] Wolfram MathWorld, "Jacobi Polynomial." https://mathworld.wolfram.com/JacobiPolynomial.html. Accedido: 20/4/2025.

[55] Wolfram Mathworld, "Gamma Function." https://mathworld.wolfram.com/GammaFunction.html. Accedido: 20/4/2025.

[56] Alireza Afzal Aghaei, "fKAN: Fractional Kolmogorov-Arnold Networks with trainable Jacobi basis functions," 2024.

[57] SpaceLearner, "JacobiKAN." https://github.com/SpaceLearner/JacobiKAN. Accedido: 20/4/2025.

[58] O. Rioul and M. Vetterli, "Wavelets and signal processing," *IEEE Signal Processing Magazine*, vol. 8, no. 4, pp. 14–38, 1991.

[59] Z. Bozorgasl and H. Chen, "Wav-KAN: Wavelet Kolmogorov-Arnold Networks," 2024.

[60] Zavareh Bozorgasl and Hao Chen, "Wav-KAN." https://github.com/zavareh1/Wav-KAN. Accedido: 20/4/2025.

[61] D. H. Griffel, "Wavelets and operators," *The Mathematical Gazette*, vol. 79, no. 484, p. 227–228, 1995.

[62] Q. Qiu, T. Zhu, H. Gong, L. Chen, and H. Ning, "ReLU-KAN: New Kolmogorov-Arnold Networks that Only Need Matrix Addition, Dot Multiplication, and ReLU," 2024.

[63] Q. Qiu and T. Zhu, "ReLU-KAN." https://github.com/quiqi/relu_kan. Accedido: 20/4/2025.

[64] Y. Wang, N. Wagner, and J. M. Rondinelli, "Symbolic regression in materials science," *MRS Communications*, vol. 9, no. 3, p. 793–805, 2019.

[65] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized out-of-distribution detection: A survey," 2024.

[66] Pytorch, "SiLU." https://pytorch.org/docs/stable/generated/torch.nn.SiLU.html. Accedido: 7/2/2025.

[67] NumPy, "NumPy user guide." https://numpy.org/doc/stable/user/index.html. Accedido: 27/5/2025.

[68] W. Schools, "NumPy Array Slicing." https://www.w3schools.com/python/numpy/numpy_array_slicing.asp. Accedido: 7/2/2025.

[69] S. K. Kumar, "On weight initialization in deep neural networks," 2017.

[70] Pytorch Foundation, "Pytorch." https://pytorch.org/. Accedido: 11/5/2025.

[71] Pytorch, "AdamW." https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html. Accedido: 3/5/2025.

[72] Pytorch, "ExponentialLR." https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ExponentialLR.html. Accedido: 3/5/2025.

[73] Papers with Code, "Datasets: Image Classification." https://paperswithcode.com/datasets?task=image-classification. Accedido: 18/5/2025.

[74] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[75] A. Krizhevsky, "The CIFAR-10 and CIFAR-100 datasets." https://www.cs.toronto.edu/~kriz/cifar.html. Accedido: 12/5/2025.

[76] Pytorch, "Conv2d." https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html. Accedido: 12/5/2025.

[77] Pytorch, "Linear." https://docs.pytorch.org/docs/stable/generated/torch.nn.Linear.html. Accedido: 12/5/2025.

[78] M. Amirian and F. Schwenker, "Radial Basis Function Networks for Convolutional Neural Networks to Learn Similarity Distance Metric and Improve Interpretability," *IEEE Access*, vol. 8, p. 123087–123097, 2020.

[79] Z. Li, "FastKAN README." https://github.com/ZiyaoLi/fast-kan/blob/master/README.md. Accedido: 13/5/2025.

[80] M. G. Altarabichi, "DropKAN: Regularizing KANs by masking post-activations," 2024.

[81] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, p. 2222–2232, 2017.

[82] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, p. 2673–2681, 1997.

[83] X. Yang and X. Wang, "Kolmogorov-arnold transformer," 2024.

[84] A. Dash, "kanformers." https://github.com/akaashdash/kansformers. Accedido: 17/5/2025.

[85] Z. Chen, Gundavarapu, and W. DI, "Vision-KAN." https://github.com/chenziwenhaoshuai/Vision-KAN. Accedido: 17/5/2025.

[86] M. Poluektov and A. Polar, "Construction of the Kolmogorov-Arnold representation using the Newton-Kaczmarz method," 2025.