# ESCUELA INTERNACIONAL DE POSGRADOS

## Trabajo Fin de Máster

## Redes Neuronales y Aprendizaje Profundo

### Fundamentos y aplicaciones

Estudiante: Rubén del Mazo Rodríguez, 2023

eip ESCUELA INTERNACIONAL POSGRADOS

## Resumen

Durante el desarrollo del máster, se han utilizado bibliotecas y entornos del lenguaje Python como son TensorFlow, Keras y Scikit-learn para las asignaturas donde se trataban las redes neuronales y el aprendizaje profundo. Estos entornos y bibliotecas ayudan a desarrollar y escalar modelos aplicables a problemas reales. Sin embargo, ¿cuáles son los principios fundamentales detrás de estos entornos? ¿Qué matemáticas y algoritmos se están aplicando por detrás? ¿Qué código pueden estar englobando esas pocas líneas necesarias para crear redes neuronales? En este trabajo me propongo analizar estos principios fundacionales de las redes neuronales y el aprendizaje profundo. Aquellos profesionales que deseen trabajar con la inteligencia artificial y el aprendizaje profundo no solo deben saber trabajar con bibliotecas y entornos, también deben conocer el qué y por qué se usan, qué hay por detrás. De esta manera se tendrá una comprensión sólida del aprendizaje profundo que permitirá desarrollar mejor el trabajo, independientemente del entorno que esté de moda, pues los principios subyacentes no varían. Se explorarán estos principios hasta crear una red neuronal de L capas personalizada paso a paso.

Por lo tanto, el objetivo de este trabajo es estudiar los conceptos fundamentales de las redes neuronales y el aprendizaje profundo, poniendo dichos conceptos en práctica. Se estudian sus capacidades y, a su vez, los retos y consecuencias del aprendizaje profundo, proporcionando conocimientos y habilidades necesarias para comenzar a aplicar el aprendizaje automático a la investigación o a la industria. El público objetivo de este trabajo no es un experto en redes neuronales, sino estudiantes y graduados de las áreas *STEM* interesados en comprender los fundamentos de las redes neuronales y el aprendizaje profundo y como implementarlos en la práctica con un lenguaje de programación.



## *Abstract*

*During the development of the master's degree, Python language libraries and frameworks such as TensorFlow, Keras and Scikit-learn have been used for the subjects dealing with neural networks and deep learning. These frameworks help to develop and scale models applicable to real problems. However, what are the fundamental principles behind these frameworks? What mathematics and algorithms are being applied behind the scenes? What code might be encompassing those few lines needed to create neural networks? In this paper I will analyze these foundational principles of neural networks and deep learning. Those professionals who wish to work with artificial intelligence and deep learning must not only know how to work with frameworks, they must also know what and why they are used, what is behind them. This will give them a solid understanding of deep learning that will allow them to do a better job, regardless of which framework is in vogue, as the underlying principles remain the same. These principles will be explored until a customized L-layer neural network is created step by step.*

*Therefore, the aim of this work is to study the fundamental concepts of neural networks and deep learning, putting these concepts into practice. It explores their capabilities and, in turn, the challenges and consequences of deep learning, providing knowledge and skills needed to start applying machine learning to research or industry. The target audience of this work is not an expert in neural networks, but rather students and graduates from STEM fields interested in understanding the fundamentals of neural networks and deep learning and how to implement them in practice with a programming language.*

# Índice

# Introducción

Como se ha mencionado en el resumen, una de las áreas que más llamó mi atención durante el desarrollo del máster, sino la que más, fue la parte de redes neuronales y el aprendizaje profundo. Decidí investigar la historia y la base de las redes neuronales y el aprendizaje profundo, más allá de los entornos o *frameworks* utilizados profesionalmente. Puede que sea mi formación como físico lo que me ha llevado a hacer esta investigación personal del funcionamiento de las redes neuronales, en vez de embarcarme directamente en aprender lo puramente práctico de este ámbito. La idea detrás de este trabajo es que, según siga avanzando en este campo, pueda comprender matices y problemáticas que no podría sin saber qué esconden las líneas de código que se programan cada día. De esta manera, y tras una introducción histórica y teórica, en este trabajo se partirá del aprendizaje automático clásico para terminar construyendo un modelo personalizable de red neuronal del número de capas que se desee. El objetivo principal es comprender cómo y por qué se construyen los distintos bloques de las redes neuronales, algunos de sus algoritmos y por qué son tan útiles. El trabajo se complementa con varios anexos que amplían la información del texto principal. Los principales puntos que se tratarán en este trabajo son los siguientes:

- Conceptos fundamentales de las redes neuronales y el aprendizaje profundo.
- Tendencias tecnológicas significativas que impulsan el auge del aprendizaje profundo.
- Arquitectura de las redes neuronales.
- Propagación hacia atrás de errores o retropropagación.
- Identificar parámetros clave en la arquitectura de una red neuronal.
- Construir, entrenar y aplicar redes neuronales.
- Explicar e implementar red neuronal eficiente y vectorizada.
- Python en el entorno de redes neuronales y aprendizaje profundo.

Aunque en este trabajo se darán definiciones formales y se utilizaran ecuaciones matemáticas, se procurará ejemplificar todo de manera que cualquiera con un mínimo de conocimientos pueda entender lo fundamental. El lenguaje de programación que se utilizará es Python, lo cual merece una explicación más allá de decir simplemente que “es el lenguaje en el que se basa todo el máster”.

## ¿Por qué Python?

Python es un lenguaje de alto nivel de código abierto, el cual tiene una filosofía de diseño que hace hincapié en permitir a los programadores expresar conceptos de forma legible y en menos líneas de código. Esta filosofía hace que el lenguaje sea adecuado para un conjunto diverso de casos de uso: scripts sencillos para web, grandes aplicaciones web (como YouTube), lenguaje de scripts para otras plataformas (como Blender y Maya de Autodesk) y aplicaciones científicas en diversas áreas, como astronomía, meteorología, física y ciencia de datos. Técnicamente es posible implementar cálculos escalares y matriciales utilizando listas en Python. Sin embargo, esto puede resultar poco manejable, y el rendimiento es pobre en comparación con lenguajes adecuados para el cálculo numérico, como MATLAB o Fortran, o incluso algunos lenguajes de propósito general, como C o C++.

Para sortear esta deficiencia, han surgido varias bibliotecas que mantienen la facilidad de uso de Python al tiempo que prestan la capacidad de realizar cálculos numéricos de manera eficiente. La biblioteca principal para los cálculos matemáticos es, sin lugar a duda, NumPy, una de las bibliotecas pioneras en llevar el cálculo numérico eficiente a Python. A su vez, Python tiene a su disposición distintos entornos de trabajo (frameworks) y bibliotecas de aprendizaje profundo, como Teras o TensorFlow. En definitiva, Python es un lenguaje relativamente sencillo de aprender, multifuncional, sostenido por una gran comunidad y con un

gran futuro por delante. Ideal para cualquiera interesado en la programación y en el aprendizaje automático. Finalicemos esta introducción con un breve recorrido de los bloques en los que se divide este trabajo.

## Bloques de temas

El trabajo ha sido dividido en bloques de temáticas, partiendo de una introducción e incrementando la complejidad poco a poco. Todos los bloques, a excepción del primero, cuentan con uno o dos archivos de código, los cuales se referenciarán. Se puede acceder a ellos a través del siguiente enlace de GitHub:

***https://github.com/fisico-alienado/TFM***

### *Bloque 1 – Introducción a las redes neuronales y el aprendizaje profundo*

Explicar qué son las redes neuronales y sus motivaciones, analizar las principales tendencias que impulsan el auge del aprendizaje profundo y dar ejemplos de dónde y cómo se aplica hoy en día. Contenidos:

- Exponer qué es una red neuronal y sus motivaciones biológicas.
- Analizar las principales tendencias que impulsan el auge del aprendizaje profundo.
- Explicar cómo se aplica el aprendizaje profundo al aprendizaje supervisado.
- Enumerar las principales categorías de modelos (CNNs, RNNs, etc.), y cuándo deben aplicarse.
- Evaluar los casos de uso apropiados para el aprendizaje profundo.

### *Bloque 2 – Conceptos básicos de las redes neuronales*

Cómo plantear un problema de aprendizaje automático con una mentalidad de red neuronal y cómo utilizar la vectorización para acelerar los modelos. Contenidos:

- Describir la construcción de la arquitectura general de un algoritmo de aprendizaje, incluyendo la inicialización de parámetros, la función de coste y el cálculo del gradiente, y la implementación de la optimización (descenso del gradiente).
- Cómo implementar versiones computacionalmente eficientes y altamente vectorizadas de modelos.
- Implementar la vectorización a través de múltiples ejemplos de entrenamiento.
- Uso de funciones y operaciones matriciales de NumPy.
- Explicar el concepto de difusión (*broadcasting*).
- Introducción a los gráficos computacionales y a la retropropagación.
- Construcción de un modelo de regresión logística estructurado como una red neuronal superficial.

El archivo de código asociado a este bloque, que se referenciará como **archivo 1**, es:

*Regresion_Logistica_como_Red_Neuronal.ipynb*

En este archivo se construye un clasificador binario utilizando la regresión logística. Se usa en la implementación una mentalidad de red neuronal, la más sencilla posible, es decir, monocapa y mononeuronal.

### *Bloque 3 – Redes neuronales poco profundas*

Explicar y construir una red neuronal poco profunda con una capa oculta, utilizando la propagación hacia delante (*forward propagation*) y la retropropagación (*backpropagation*). Contenidos:

- Describir las unidades y las capas ocultas.
- Cómo utilizar unidades con una función de activación no lineal.

- Cómo implementar la propagación hacia delante y hacia atrás.
- Cómo y por qué aplicar inicialización aleatoria a su red neuronal.
- Analizar las dimensiones de matrices y vectores para comprobar las implementaciones de redes neuronales.
- Calcular la pérdida por entropía cruzada.
- Implementación de una red neuronal de clasificación de dos clases con una sola capa oculta.

El archivo de código asociado a este bloque, que se referenciará como **archivo 2**, es:

*Red_neuronal_con_una_capa_oculta.ipynb*

En este archivo, que se debe ver a continuación del de regresión logística, implementa una red neuronal con una capa oculta. Si la regresión logística era una "red neuronal" monocapa mononeuronal, esta red neuronal va un (primer) paso más allá, siendo de dos capas, una de ellas la capa oculta. El modelo de red neuronal cambia mucho, como no podía ser de otra manera, con respecto a la pseudo red neuronal del anterior archivo.

### *Bloque 4 – Redes neuronales profundas*

Analizar los cálculos clave que subyacen en el aprendizaje profundo y, a continuación, utilizarlos para construir y entrenar redes neuronales profundas para tareas de visión artificial. Contenidos:

- Describir la estructura de bloques sucesivos de una red neuronal profunda.
- Cómo utilizar una caché para pasar información de la propagación hacia delante a la retropropagación.
- Cómo construir una red neuronal profunda de L capas.
- Explicar el papel de los hiperparámetros en el aprendizaje profundo.
- Explorar distintos tipos de inicialización y su importancia.
- Ejemplo práctico: construir una red neuronal de L capas.

Los archivos de código asociados a este bloque se referenciarán como **archivo 3**:

*Red_Neuronal_Profunda_paso_a_paso.ipynb*

y **archivo 4**:

*Aplicacion_red_neuronal_profunda.ipynb*

En el archivo 3 se quita la última restricción, el número de capas, y se implementan las funciones necesarias para construir una red neuronal profunda con tantas capas y unidades ocultas como se desee. Se volverá a crear la red neuronal con una única capa oculta, pero cambiando su estructura y funciones de activación, adaptándolas al problema de clasificación para el que será utilizada. Y las funciones de esta nueva estructura serán aprovechadas para crear el caso general de $L$ capas ocultas. De esta manera, podremos comparar en el archivo 4 la diferencia de rendimiento entre una red con una capa oculta y una red con $L-1$ capas ocultas.

En el archivo 4 se entrenarán las dos redes neuronales que fueron creadas en el archivo 3 con el conjunto de imágenes que fueron utilizadas en el archivo de la regresión logística.

### *Anexos*

Forman un bloque de varios apartados que complementan y expanden conceptos o aportan información que, de incluirse en los bloques principales, cortarían innecesariamente la lectura. Contenidos:

- Notación para redes neuronales y el aprendizaje profundo.
- Bibliotecas de cálculo: NumPy vs. SciPy.
- Bibliotecas y entornos para el aprendizaje profundo.
- *Broadcasting* con Python.
- Función de coste: Pérdida de entropía cruzada binaria

Los archivos de código asociados a este bloque se referenciarán como **anexo 1**:

*Implementacion_funciones_con_NumPy.ipynb*

y **anexo 2**:

*Vectorizacion_de_NN_con_Python.ipynb*

En el anexo 1 se muestran varias funciones y técnicas útiles con NumPy, tales como las funciones de activación más famosas, reorganización de matrices para el análisis de imágenes y normalización. En el anexo 2 se muestra la forma de implementar efectivamente las redes neuronales, mostrando cómo funciona el cálculo vectorial y matricial y la vectorización de redes neuronales.

# 1

# Introducción a las redes neuronales y el aprendizaje profundo

El aprendizaje profundo, en inglés *deep learning*, es una rama del aprendizaje automático que se basa en la creación de modelos computacionales compuestos por varias capas de procesamiento para detectar estructuras complejas en los datos que reciben. Estos modelos pueden crear varios niveles de abstracción que representan los datos y mejorar su rendimiento al poder acceder a un mayor número de datos. El aprendizaje profundo es, a su vez, parte del campo de la inteligencia artificial, IA (*Artificial Intelligence*, *AI*, en inglés). En muchas tareas, como la visión por ordenador, el reconocimiento de voz y la traducción automática, el rendimiento de los sistemas de aprendizaje profundo supera enormemente el de los sistemas de aprendizaje automático convencional. En resumen, se podría decir que el término "aprendizaje profundo" se refiere al entrenamiento de redes neuronales, en general, muy grandes.

El aprendizaje profundo ya ha transformado los negocios tradicionales de Internet, como la búsqueda web y la publicidad. Pero el aprendizaje profundo también está permitiendo la creación de productos y negocios totalmente nuevos y formas novedosas de ayudar a las personas. Todo lo que va desde la mejora de la atención sanitaria donde, por ejemplo, el aprendizaje profundo se está volviendo realmente bueno en la lectura de imágenes de rayos X o ayudando a detectar cánceres, hasta ofrecer una educación personalizada, la agricultura de precisión, e incluso los automóviles de conducción automática. Para resaltar su importancia, podemos comparar su impacto con el que generó el desarrollo de la electricidad. Hace unos 100 años, la electrificación de nuestra sociedad transformó todas las grandes industrias, desde el transporte hasta la fabricación, pasando por la sanidad, las comunicaciones y muchas más. Actualmente, se intuye un camino sorprendentemente claro para que la IA provoque una transformación igual de grande. Y, por supuesto, la parte de la IA que está creciendo más rápidamente e impulsando muchos de estos desarrollos es el aprendizaje profundo. Además, hoy en día el aprendizaje profundo es una de las habilidades más buscadas en el mundo de la tecnología. ¿Cuál es el origen de esta increíble tecnología?

## Funcionamiento del cerebro y su relación con las redes neuronales

Cuando se inventaron las redes neuronales hace muchas décadas, la motivación original era escribir un software que pudiera imitar cómo aprende y piensa el cerebro humano. Hoy en día las redes neuronales, a veces también llamadas redes neuronales artificiales, se han convertido en algo muy diferente de cómo cualquiera de nosotros podría pensar que funciona y aprende realmente el cerebro. Algunas de las motivaciones biológicas aún permanecen en la forma en que actualmente pensamos sobre las redes neuronales artificiales o informáticas.

El cerebro humano demuestra un nivel de inteligencia superior o más capaz que cualquier otro órgano u organismo biológico hasta la fecha. Las redes neuronales, que nacieron con la motivación de intentar construir un software que imitara al cerebro, comenzaron su andadura en la década de 1950, y luego cayeron en desuso por un tiempo. En los años 80 y principios de los 90 volvieron a ganar popularidad, mostrando una enorme eficacia en algunas aplicaciones como el reconocimiento de dígitos manuscritos, que ya entonces se utilizaban para leer códigos postales para correos y para leer cifras en dólares en cheques manuscritos. Pero volvieron a caer en desuso a finales de la década de 1990 y fue a partir de 2005 cuando su uso resurgió y fueron, en parte, rebautizas con el nombre de aprendizaje profundo. El aprendizaje profundo y las redes neuronales tienes significados parecidos; sin embargo, este *rebranding* para las nuevas mejoras en el campo daba una sensación de estar haciendo algo nuevo, que además sonaba muy bien a oídos del público. Así que esa resultó ser la marca que despegó hace unas dos décadas. Desde entonces, las redes neuronales han revolucionado un área de aplicación tras otra.

La primera área de aplicación en la que las redes neuronales modernas, o el aprendizaje profundo, tuvieron un gran impacto fue probablemente el reconocimiento de voz, donde se empezaron a ver sistemas de reconocimiento del lenguaje mucho mejores gracias al aprendizaje profundo moderno y a autores como Geoffrey Hinton, que fue fundamental para ello. La siguiente área donde se produjeron grandes avances fue en la visión por ordenador, que tuvo un momento álgido en 2012 con el proyecto *ImageNet*, una gran base de datos visual diseñada para su uso en la investigación de software de reconocimiento de objetos, la cual impulso avances en este campo. Otro campo donde se vieron avances significativos gracias al aprendizaje profundo fue en el procesamiento de textos o NLP (*Natural Language Processing*, Procesamiento del Lenguaje Natural). Actualmente las redes neuronales se utilizan en incontables campos: desde las predicciones climáticas a las imágenes médicas, pasando por la publicidad en línea o recomendaciones de productos. Aunque las redes neuronales actuales no tienen casi nada que ver con la forma en que aprende el cerebro, la motivación inicial fue intentar crea software que imitase al cerebro. ¿Cómo se supone que funciona el cerebro?

En la **figura 1** se muestra una ilustración de las neuronas cerebrales. Una neurona está formada por el cuerpo celular y tiene diferentes entradas. En una neurona biológica, los cables de entrada se llaman dendritas, que envían impulsos eléctricos a otras neuronas a través del cable de salida, que se llama axón. Esta neurona biológica puede enviar impulsos eléctricos que se convierten en la entrada de otra neurona, y a veces, forman nuevas conexiones.

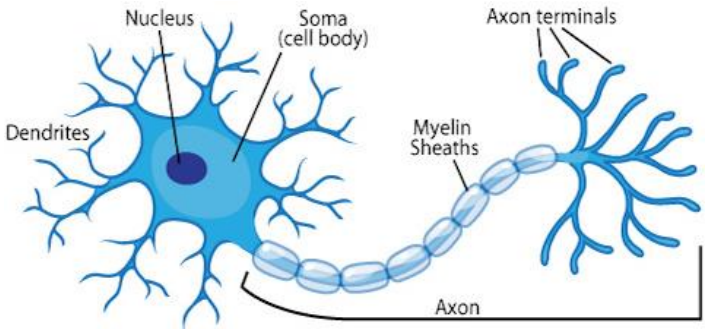


*Figura 1*. Anatomía de una neurona biológica.

Cada neurona tiene un número de entradas donde recibe impulsos eléctricos de otras neuronas, se llevan a cabo algunos procesos, y luego envía los resultados a otras neuronas por estos impulsos, que a su vez harán sus propios procesos y tal vez envíen su propia salida a otras neuronas. En la **figura 2** se representa una visión simplificada de este proceso de comunicación neuronal. Hoy en día, todo apunta a que esta es la materia de la que está hecho el pensamiento humano.

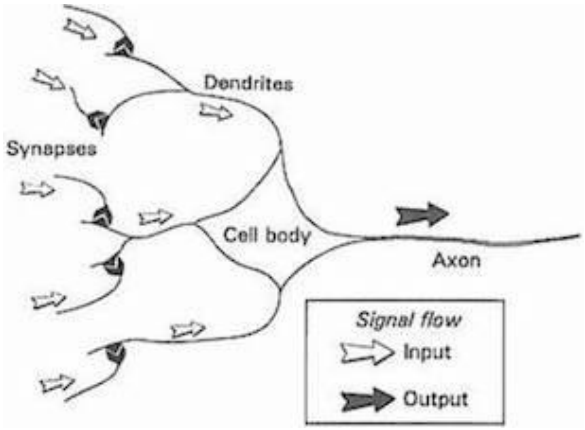


*Figura 2*. Anatomía de una neurona biológica.

Una red neuronal artificial utiliza un modelo matemático muy simplificado de lo que hace una neurona biológica. Lo que una neurona artificial hace es tomar algunas entradas, una o más, que son sólo números, realiza algunos cálculos y produce otro número, que podría ser una entrada para una segunda neurona. Cuando se construye una red neuronal artificial, o un algoritmo de aprendizaje profundo, en lugar de construir una neurona cada vez, a menudo se quieren simular muchas neuronas al mismo tiempo. Lo que estas neuronas hacen colectivamente es recibir números, realizar algún cálculo y emitir otros números. Hoy en día, la mayoría de los científicos nos advierten que deberíamos tener cuidado con esta analogía, ya que las redes neuronales fueron diseñadas para resolver problemas de aprendizaje de máquinas (en inglés, *machine learning*) y no para representar el cerebro con precisión. Sin embargo, la idea de que una neurona biológica simplificada representa la unidad central de una red neuronal es una metáfora que ha perdurado a través de las décadas.

Llegados a este punto, es importante matizar lo siguiente: aunque se ha realizado una vaga analogía entre las neuronas biológicas y las neuronas artificiales, hoy en día no se conoce cómo funciona el cerebro humano. De hecho, cada pocos años, los neurocientíficos hacen algún avance fundamental sobre el funcionamiento del cerebro. Esto es señal de que aún queda mucho que descubrir sobre el funcionamiento real del cerebro y, por tanto, los intentos de imitar ciegamente lo que hoy sabemos del cerebro humano, que francamente es muy poco, probablemente no llevarán muy lejos en la construcción de inteligencia artificial real. Desde luego, no con el nivel actual de conocimientos en neurociencia. Dicho esto, incluso con estos modelos extremadamente simplificados de una neurona, es posible construir algoritmos de aprendizaje profundo realmente potentes. Así que, a medida que se profundiza en las redes neuronales y en el aprendizaje profundo, aunque los orígenes fueron motivados por razones biológicas, no hay que tomarse la motivación biológica demasiado en serio. De hecho, la mayoría de los investigadores en aprendizaje profundo han dejado de fijarse tanto en la motivación biológica y se limitan a utilizar principios de ingeniería para averiguar cómo construir algoritmos que sean más eficaces.

## ¿Qué es una red neuronal artificial?

Si, en general, el aprendizaje profundo consiste en entrenar grandes redes neuronales, la siguiente pregunta está clara: ¿qué es una red neuronal (artificial). Una red neuronal artificial (ANN) es un modelo computacional que permite simular el comportamiento del cerebro humano y dotar a las máquinas de la capacidad de aprender de una manera similar a como lo hace nuestro cerebro. Está compuesta por capas de nodos o neuronas artificiales que reciben información del exterior o de otras neuronas, procesan esa información y generan un valor de salida que alimenta a otras neuronas de la red o son la salida hacia el exterior de la red. Cada neurona artificial está formada por un conjunto de entradas ponderadas por un peso que determina la importancia de la información recibida y un conjunto de funciones que relacionan matemáticamente los valores de las entradas, sus pesos y un sesgo para el cálculo del valor de salida de la neurona. Las redes neuronales artificiales son utilizadas en campos como el reconocimiento de voz, el reconocimiento de imágenes y el análisis predictivo.

Comencemos con un sencillo ejemplo del aprendizaje automático clásico. Tomemos una serie de puntos en una gráfica en dos dimensiones y dibujemos una línea que se ajuste a ellos lo mejor posible. A partir de unos valores de entrada, $x$ y unos valores de salida, $y$, se está generalizando una función de predicción que puede asignar a cualquier valor de entrada un valor de salida. Esto se conoce como regresión lineal y es una técnica de aprendizaje automático clásico de 200 años de antigüedad para extrapolar una función general a partir de un conjunto de pares de entrada-salida. Y es una técnica que ha sido, y sigue siendo, muy útil, puesto que hay un número incalculable de funciones para las que es difícil desarrollar ecuaciones directamente, pero es fácil recopilar ejemplos de pares de entrada y salida en el mundo real para generarlas por extrapolación. Es decir, lo que hace esencialmente es el propósito del aprendizaje automático supervisado: “aprender” una función a partir de un conjunto de ejemplos de entrenamiento, en el que cada ejemplo es un par de entrada y salida de la función. En concreto, los métodos de aprendizaje automático

deben derivar una función que pueda generalizar bien a entradas que no estén en el conjunto de entrenamiento, ya que entonces se podrá aplicarla realmente a entradas para los que no se tenga una salida y se podrán hacer predicciones.

Sin embargo, estos métodos clásicos tienen muchas limitaciones. Utilicemos un ejemplo real sencillo: tenemos un conjunto de datos de viviendas; sus tamaños y precios son conocidos. Con este conjunto de datos, se quiere ajustar una función para predecir el precio de una vivienda cualquiera en función de su tamaño. Si se está familiarizado con la regresión lineal, lo más sencillo podría ser ajustar los datos por este método de aprendizaje automático clásico, es decir, ajustar los datos a una línea recta. Sin embargo, para ser más sofisticados, se podría argumentar que, dado este ajuste, como los precios a partir de cierto tamaño serían negativos y eso no es posible, se necesita otro tipo de ajuste. Por ejemplo, se podría ajustar mediante regresión lineal hasta el punto en el que dado un tamaño el precio es cero y a partir de ese punto considerar todo tamaño inferior a precio cero. Podemos ver esto en la **figura 3**, siendo la línea roja el ajuste realizado.

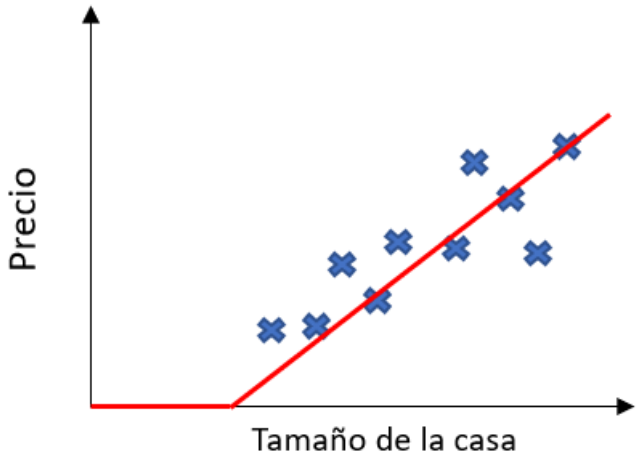


*Figura 3.* Ajuste no lineal de un problema, a priori, de regresión lineal.

Este ajuste ya no sería una regresión lineal, sino una función no lineal. En la literatura de redes neuronales, esta función que va a cero y luego se toma como una línea recta se llama función ReLU (*Rectified Lineal Unit*), que significa unidad lineal rectificada. Se emplea en redes neuronales profundas para visión artificial y reconocimiento de voz, y se utilizará más adelante en este trabajo. Se puede pensar en esta función que acabamos de ajustar a los precios de la vivienda como una red neuronal muy simple. Es la red neuronal más simple posible, que se puede representar con el siguiente esquema:

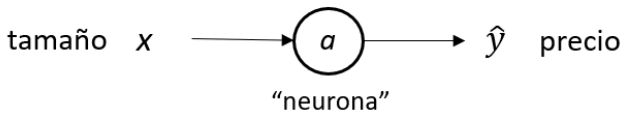


*Figura 4*. Esquema de red neuronal simple, donde "a" es la función de activación de la "neurona" e $\hat{y}$ es la salida o predicción de la red neuronal.

En el esquema de la **figura 4**, tenemos como entrada a la red neuronal el tamaño de una casa, $x$. Entra en un nodo, representado como un círculo, que es la "neurona"; la información es procesada por una función, que llamaremos función de activación. El nodo o "neurona" da como salida el precio estimado, que llamaremos $\hat{y}$. El término "activación" procede de la neurociencia y se refiere al grado en que una neurona envía una señal de salida elevada a otras neuronas situadas a continuación. Otra forma de concebir una "neurona" es como un pequeño ordenador cuya única función es recibir una o varias cifras, como el tamaño de la casa, y devolver una o varias cifras, en este caso el precio estimado de la vivienda. Una red neuronal más grande se formará creando el número deseado de nodos o "neuronas" y apilándolas o conectándolas en forma de red.

Y esta red, a su vez, tendrá el apilado que deseemos: número de capas, capas por neurona, funciones de activación, etc.

Siguiendo con el ejemplo de las viviendas, ¿qué sucede si se aumenta el número de parámetros de entrada? ¿Y si se aumenta el número de nodos y capas? Por ejemplo, sino se tiene en cuenta únicamente el tamaño de la vivienda para predecir el precio, sino también el número de habitaciones, la ubicación de la vivienda y el poder adquisitivo medio en dicha ubicación. Como se ilustra en la **figura 5**, todos los valores de entrada o características del conjunto de entrenamiento son incluidos en cada nodo o "neurona", son procesados con una función (no lineal, se explicará más adelante en el trabajo el por qué), y cada neurona devuelve un resultado que se combina con el de las otras para obtener un resultado final de salida; en este caso, el precio de la vivienda. Traducido a lenguaje humano, las "neuronas podrían estar calculando", con la combinación de los parámetros iniciales, el tamaño de la familia interesada en comprar la casa (por ejemplo, combinando tamaño y número de habitaciones), servicios accesibles a pie (de acuerdo con la zona), calidad de las escuelas (de acuerdo con el poder adquisitivo y la zona), etc. Pero el objetivo final sigue siendo el mismo: predecir el precio de la vivienda.

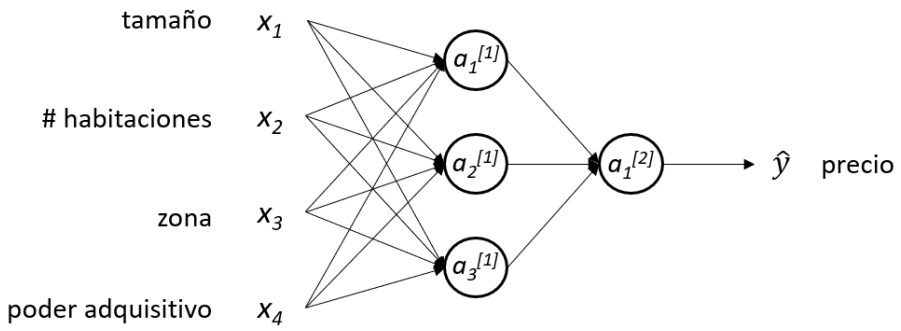


*Figura 5.* Posible representación de una red neuronal simple para el problema propuesto de las viviendas, donde $a_k^{[l]}$ es el valor de la función de activación de la "neurona" k de la capa l.

En esta representación concreta, las entradas iniciales son recibidas por tres unidades ocultas o "neuronas", que forman una capa. En la terminología de las redes neuronales, una capa es una agrupación de neuronas (una o varias) que reciben los mismos valores de entrada (o al menos similares) y que, a su vez, devuelven otros valores, resultado de las entradas y sus funciones de activación. La neurona más a la derecha conforma una capa conocida como la **capa de salida**, porque su valor de salida será la predicción de la red neuronal, $\hat{y}$. Los valores de entrada, $x_k$, conforman la **capa de entrada**, donde cada uno de ellos representa una característica del conjunto de entrenamiento. En la práctica, cada neurona en cada capa tiene acceso a todos los valores de entrada disponibles, o al menos a su mayoría, puesto que sería costosísimo decidir manualmente a qué valores tiene acceso cada neurona y por qué. La habilidad o utilidad de las redes neuronales reside en que, recibiendo todos los datos, es capaz de crear relaciones y especializar las neuronas sin que el ingeniero lo indique manualmente. Las características o valores de entrada (*inputs*) de cada ejemplo de entrenamiento pueden almacenarse o expresarse como un vector $\vec{x}$ o simplemente $x$. Este vector de características se introduce en la capa intermedia, que calcula tres valores de activación, uno por cada neurona. Estos números y estos tres valores de activación a su vez se convierten en otro vector que se envía a la capa de salida que, finalmente, emite el precio predicho.

Eso es todo lo que es una red neuronal. Un conjunto de capas con "neuronas" en las que cada una recibe un vector numérico, lo transforma con unas funciones matemáticas, y emite otro vector. Por ejemplo, la capa intermedia de la **figura 5** recibe cuatro números y emite tres. Por terminar con la nomenclatura de este apartado, las capas intermedias se denominan **capas ocultas**. Y cada elemento de estas capas es una **unidad oculta** o "neurona" artificial. Quizás no sea el mejor nombre ni el más intuitivo, pero esa terminología viene del entrenamiento de un conjunto de datos. En un conjunto de entrenamiento, se conocen los valores

de entrada, $x_k$, y los valores de salida correctos, $y$. Pero los valores intermedios en las neuronas están ocultos, y de ahí el nombre. La implementación básica de una red neuronal en el aprendizaje supervisado es bastante sencilla: solo es necesario darle un número de ejemplos de pares de entrada, $x_k$, y salida, $y$, en un conjunto de entrenamiento, y esas "neuronas" que se han agrupado en capas descubrirán cómo inferir resultados. Y lo extraordinario de las redes neuronales es que, si se les dan suficientes ejemplos de entrenamiento, las redes neuronales son extraordinariamente buenas a la hora de encontrar funciones que asignen con precisión los valores de entrada a los de salida. Por supuesto, con los datos del conjunto de entrenamiento, el objetivo es conseguir que la mayoría de las predicciones coincidan con el valor verdadero de salida.

En resumen, en una red neuronal estándar sus partes son: entradas (*inputs)*, capas (*layers*), unidades ocultas (*hidden units*) o "neuronas" artificiales y salidas (*outputs*). El número de capas y unidades ocultas es totalmente variable. Se pueden introducir tantas capas intermedias (ocultas) como se desee, con el número de unidades ocultas que se quiera entrenar en cada capa. Elegir el número correcto de capas y unidades ocultas por capa influye mucho en el rendimiento de un algoritmo de aprendizaje. En las siguientes figuras se muestran unos esquemas sencillos de redes neuronales y en el anexo y a lo largo del trabajo se mostrarán figuras de redes mayores.

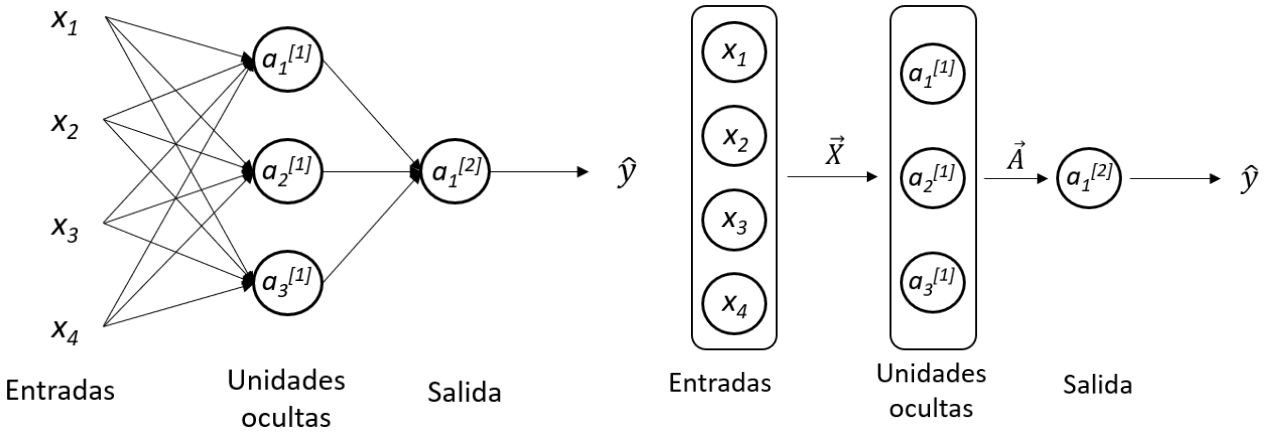


*Figura 6.* Ejemplo de representación de red neuronal estándar con representación compleja (*izq*) y con representación simplificada (*dch*).

Las redes neuronales son probablemente más útiles y potentes en el aprendizaje supervisado, es decir, en los problemas donde están mapeados unos datos de entrada a unos datos de salida, y se desea realizar predicciones a partir de nuevos datos. En el siguiente apartado veremos algunos ejemplos más de aprendizaje supervisado que demuestran la potencia y utilidad de las redes neuronales.

## Aprendizaje supervisado con redes neuronales

Cada vez se habla más de los éxitos de las redes neuronales, y los medios de masas han exagerado mucho sobre sus capacidades presentes y futuras. Y quizá parte de ese bombo esté justificado, dado lo bien que funcionan. Pero resulta que, hasta ahora, casi todo el valor económico creado por las redes neuronales ha sido a través de un tipo de aprendizaje automático: el aprendizaje supervisado. En él se utiliza un conjunto de datos de entrenamiento para enseñar a los modelos a obtener los resultados deseados. Este conjunto de datos de entrenamiento incluye entradas y salidas correctas, que permiten al modelo aprender con el tiempo. El algoritmo mide su precisión a través de la función de pérdida (*loss function*), ajustándose hasta que el error se ha minimizado lo suficiente. Así, por ejemplo, se ha mencionado su aplicación para predecir el precio de la vivienda, en la que se introducen algunas características que se intuyen relacionadas con el

valor del inmueble y se intenta obtener o estimar el precio correcto. He aquí otros ejemplos en los que las redes neuronales se han aplicado con gran eficacia:

- Posiblemente, la aplicación más lucrativa del aprendizaje profundo hoy en día sea la publicidad en línea. Quizá no es la más inspiradora, pero sin duda da muchos beneficios. Introduciendo información sobre un anuncio que el sitio web está pensando en mostrarte, y cierta información sobre el usuario, las redes neuronales se han vuelto muy buenas prediciendo si se hará clic o no en un anuncio. Mostrar a los usuarios los anuncios en los que es más probable que hagan clic ha sido una aplicación increíblemente lucrativa de las redes neuronales en múltiples empresas, porque tiene un impacto directo en los resultados de algunas de las grandes empresas de publicidad online.
- La visión por ordenador, o visión artificial, también ha avanzado mucho en los últimos años, gracias en gran medida al aprendizaje profundo. Se puede utilizar, por ejemplo, para el reconocimiento de personas, objetos, etiquetado de fotografías en categorías, etc.
- Los últimos avances en reconocimiento de voz también han sido muy interesantes. Actualmente es posible introducir como entrada un clip de audio en una red neuronal y obtener una transcripción de texto. O su uso en las IAs Siri de Apple o Alexa de Amazon.
- La traducción automática también ha avanzado enormemente gracias al aprendizaje profundo. Por ejemplo, permite que una red neuronal reciba como entrada una oración en un idioma y devuelva como salida esa oración traducida a cualquier otro idioma. Servicios gratuitos ampliamente conocidos son Google Translator y DeepL.
- Por último, mencionar los avances realizados en la conducción autónoma. Se puede introducir imágenes, por ejemplo, de lo que hay delante del coche, así como información de un radar, y a partir de esos datos entrenar a una red neuronal para que nos proporcione la posición de los demás coches en la carretera. Esto es un componente clave de los sistemas de conducción autónoma.

En general, gran parte de la creación de valor a través de las redes neuronales ha consistido en seleccionar inteligentemente lo que debería ser la entrada, y lo que debería ser la salida, para un problema concreto, y finalmente encajar este componente de aprendizaje supervisado en un sistema a menudo más grande, como un vehículo autónomo.

No hay un único tipo de arquitectura de red neuronal para todos los problemas que existen. Se han desarrollado diferentes tipos de arquitectura de redes neuronales que obtienen mejores resultados para ciertos tipos de problemas y de entrada de datos. Para los ejemplos del sector inmobiliario y la publicidad en línea, una red neuronal con arquitectura estándar podría ser más que suficiente. En el caso de la visión por ordenador, es decir, al tratar con imágenes, la arquitectura que se suele utilizar es CNN, redes neuronales convolucionales (*Convolutional Neural Networks*). El audio tiene una componente temporal (se reproduce a lo largo del tiempo), así que lo más natural es representarlo como una serie o una secuencia temporal unidimensional. Por lo tanto, el audio se clasifica como una secuencia de datos. Para los datos secuenciales se suele utilizar las RNN (*Recurrent Neural Networks*), redes neuronales recurrentes. En el lenguaje, las palabras vienen de forma secuencial, es decir, de una en una. El lenguaje también se representa de forma natural como datos secuenciales. Para estas aplicaciones se utilizan versiones más complejas de las RNN. En aplicaciones más complejas, como la conducción autónoma, se tienen tipos distintos de entradas. Como se ha comentado, se pueden tener imágenes como entradas, lo que sugiere una estructura tipo CNN, pero también se puede tener información del radar. En estos casos se suele utilizar una arquitectura híbrida más personalizada y compleja. Esto se resume en la siguiente tabla:

| **Entrada ($X$)** | **Salida ($y$)** | **Posibles aplicaciones** | **Posible arquitectura de red neuronal** |
|---|---|---|---|
| Características de la vivienda | Precio | Sector inmobiliario | Estándar |
| Anuncio, información de usuario | ¿Clic? (Sí / no) | Publicidad en línea | Estándar |
| Imagen | Categoría | Etiquetado de fotos | CNN |
| Sonido / audio | Transcripción a texto | Reconocimiento de voz | RNN |
| Idioma 1 | Idioma 2 | Traducción automática | RNN |
| Imagen, información de radar | Posición de otros coches | Conducción automática | Híbrida personalizada |

*Tabla 1*. Ejemplos de combinaciones de pares de entrada y salida, ámbito en el que se utilizan o del que se obtienen, y posible arquitectura de red neuronal óptima para entrenar con esos datos.

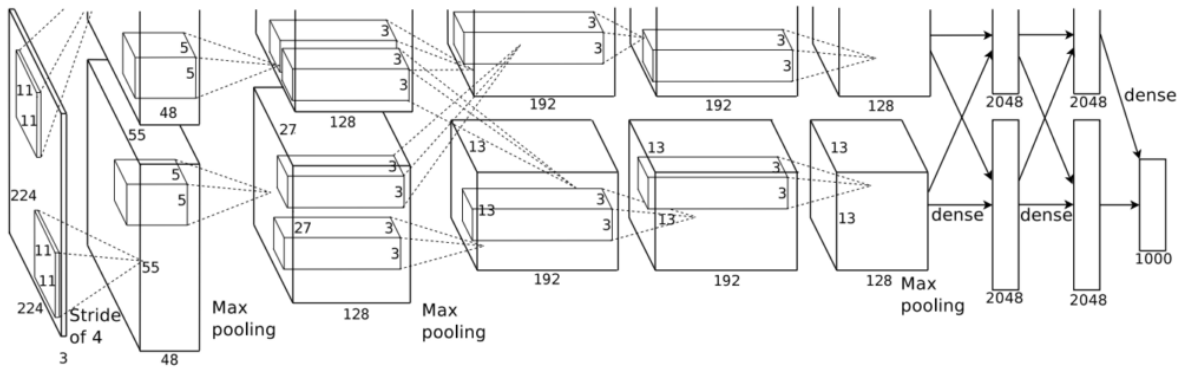


*Figura 7*. Arquitectura de la CNN de AlexNet para clasificación de imágenes de objetos.

Se puede hacer otra clasificación en función de los datos de entrada: datos **estructurados** y **no estructurados**. Los datos estructurados son esencialmente bases de datos, donde los datos están claramente definidos y clasificados con patrones que facilitan su búsqueda. Suelen clasificarse como datos cuantitativos. Los datos no estructurados son "todo lo demás", es decir, datos que no están claramente definidos u ordenados, como pueden ser pistas de audio, vídeos o publicaciones en redes sociales. Suele clasificarse como datos cualitativos. Volviendo a nuestros ejemplos, en la predicción del precio de la vivienda, los datos se pueden disponer en forma de una base de datos organizada en columnas, donde cada una representa una variable de entrada menos una, que es la salida (en este caso el precio). Para predecir si un usuario hará clic en un anuncio, es posible que se tenga información sobre el usuario, como la edad, alguna información sobre el anuncio, etc, y las etiquetas que se están tratando de predecir es si hace clic o no (se puede representar de forma binaria, con 1 o 0). Esta información se puede estructurar en una base de datos, por lo que son datos estructurados, lo que significa que cada una de las características tienen un significado muy bien definido.

Por el contrario, los datos no estructurados se refieren a datos como audio, procesado o no, o imágenes en las que se quiere reconocer los objetos presentes o el texto. En estos casos, las características pueden ser los valores de los píxeles de una imagen o las palabras individuales de un texto. Históricamente, a los ordenadores les ha resultado mucho más difícil dar sentido a los datos no estructurados que a los estructurados. De hecho, la especie humana ha evolucionado hasta ser muy buena en la comprensión de señales de audio, así como de imágenes. Y el texto, aunque más reciente en la historia evolutiva, también

somos muy buenos como especie interpretándolo. Uno de los aspectos más interesantes del auge de las redes neuronales es que, gracias a ellas y al aprendizaje profundo, los ordenadores son ahora mucho mejores interpretando datos no estructurados en comparación con hace solo unos años. Y esto crea oportunidades para muchas nuevas aplicaciones interesantes que utilizan el reconocimiento de voz, reconocimiento de imágenes, procesamiento del lenguaje natural en el texto, y mucho más.

Por ejemplo, al crear una aplicación de reconocimiento facial, el objetivo es entrenar una red neuronal que tome como entrada una imagen y obtener como resultado la identidad de la persona que aparece en ella. Pongamos que la imagen tiene 1000 por 1000 píxeles. Su representación en el ordenador es en realidad una cuadrícula de 1000 x 1000 píxeles, también llamada matriz de 1000 x 1000 de valores de intensidad de píxel. Digamos que los valores de intensidad de píxel o valores de brillo de píxel van de 0-255. Si tomáramos estos valores de intensidad de píxel y los insertásemos en un vector, acabaríamos con una lista de un millón de valores de intensidad de píxel. Un millón porque un cuadrado de 1.000 por 1.000 da un millón de números. El problema del reconocimiento facial es, ¿se puede entrenar una red neuronal que tome como entrada un vector de características con un millón de valores de luminosidad de píxeles y obtenga como resultado la identidad de la persona que aparece en la imagen? Sí, y se explicará con más detalle en el bloque 4.

Como las personas tienen una empatía natural para entender los datos no estructurados, es posible que se oiga hablar más de los éxitos de las redes neuronales en datos no estructurados en los medios de comunicación. Por ejemplo, llama mucho más la atención que el nuevo móvil de la compañía “X” sea capaz de reconocer un rostro y lo use para desbloquear el teléfono, que escuchar que la empresa “Z” ha incrementado un 5% sus beneficios gracias a un nuevo software que predice con más exactitud fluctuaciones en el mercado de viviendas. Pero resulta que gran parte del valor económico a corto plazo que están creando las redes neuronales ha sido en casos con datos estructurados, como sistemas de publicidad mucho mejores, y simplemente una capacidad mucho mejor para procesar las gigantescas bases de datos que tienen muchas empresas para hacer predicciones precisas a partir de ellas.

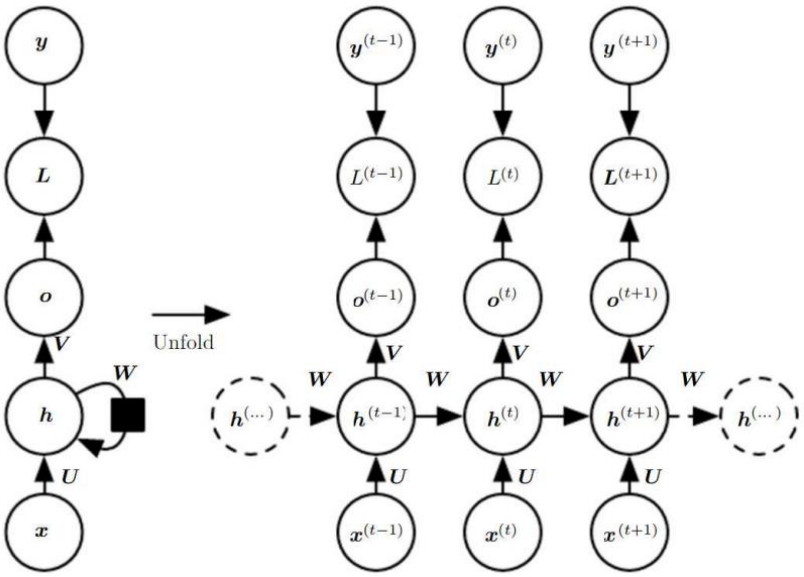


*Figura 8*. Posible arquitectura de una RNN. A diferencia de una red neuronal simple, en las RNNs la salida depende no solo de la entrada, sino también del estado del sistema.

Queda claro que las redes neuronales han transformado el aprendizaje supervisado y están generando un enorme valor económico. Sin embargo, las ideas técnicas básicas que subyacen a las redes neuronales existen desde hace décadas. Entonces, ¿por qué han despegado en los últimos años? En el siguiente apartado se explicará por qué las redes neuronales y el aprendizaje profundo se han convertido recientemente en una herramienta increíblemente potente y algunos de sus principales impulsores.

## El auge del aprendizaje profundo

Lo que ha ocurrido en nuestra sociedad en los últimos veinte años es que, para muchos problemas, hemos pasado de tener una cantidad de datos relativamente pequeña a tener una cantidad de datos gigantesca. Y todo ello gracias a la digitalización de una sociedad en la que gran parte de la actividad humana se desarrolla ahora en el ámbito digital: los ordenadores, los sitios web, las aplicaciones móviles, y las actividades en los dispositivos digitales en general crean datos y, gracias al auge de las cámaras de bajo coste integradas en nuestros teléfonos móviles, los acelerómetros, todo tipo de sensores en el Internet de las Cosas, también hemos ido recopilando cada vez más datos. Así que en los últimos años, en muchas aplicaciones hemos acumulado muchos más datos de los que los algoritmos de aprendizaje tradicionales han sido capaces de aprovechar. Esto se conoce como el auge del ***Big Data***.

En la **figura 9** se representa en el eje horizontal la cantidad de datos que tenemos disponibles, y en el eje vertical el rendimiento de predicción de los algoritmos de aprendizaje implicados. Por ejemplo, el rendimiento puede ser la precisión de un clasificador de *spam*, un predictor de clics publicitarios o la precisión de una red neuronal que averigua la posición de otros coches en la carretera para el correcto funcionamiento de un coche autónomo. Resulta que si trazamos el rendimiento de un algoritmo de aprendizaje tradicional, como las máquinas de vectores soporte, la regresión logística o la regresión lineal, en función de la cantidad de datos disponibles, podemos obtener una curva parecida a la curva roja, en la que el rendimiento mejora a medida que añadimos más datos, pero al cabo de un tiempo se estanca. Incluso cuando se alimentan esos algoritmos con más datos, es muy difícil conseguir que el rendimiento siga aumentando. Simplemente no son capaces de escalar con la cantidad de datos ahora disponibles y de aprovechar eficazmente todos estos datos.

¿Qué sucede con las redes neuronales? Lo que los investigadores de IA empezaron a observar fue que, si se entrenaba una red neuronal pequeña con estos conjuntos de datos, el rendimiento podría ser parecido a la línea azul de la **figura 9**. Si se entrenaba una red neuronal de tamaño medio, es decir, con más neuronas, su rendimiento podía parecerse a la línea morada. Y si se entrenaba una red neuronal muy grande, es decir, con muchas neuronas artificiales y muchas capas, en algunas aplicaciones el rendimiento seguía aumentando (posible representación con la línea verde). Esto significaba dos cosas: que para cierto tipo de aplicaciones en las que se dispone de muchos datos (y aquí entra en juego el término *Big Data*), si se es capaz de entrenar una red neuronal muy grande para aprovechar la enorme cantidad de datos de los que se dispone, se puede alcanzar un rendimiento muy alto en casi cualquier tarea, desde el reconocimiento del lenguaje hasta el reconocimiento de imágenes, pasando por las aplicaciones de procesamiento del lenguaje natural y muchas más, lo que no era posible con las generaciones anteriores de algoritmos de aprendizaje. Esto provocó el despegue de los algoritmos de aprendizaje profundo, junto con el desarrollo de procesadores informáticos más rápidos, incluido el auge de las GPU, las unidades de procesamiento gráfico. Las GPUs son un hardware diseñado originalmente para generar gráficos de ordenador atractivos, pero que han resultado ser muy potentes también para el aprendizaje profundo y han contribuido en gran medida a que los algoritmos de aprendizaje profundo se conviertan en lo que son hoy. Se puede representar grosso modo el rendimiento de las redes neuronales de acuerdo con la cantidad de datos y su tamaño como se ve en la **figura 9**.

Las líneas de la figura han sido dibujadas de forma idealizada para facilitar la discusión. En la literatura y estudios sobre el aprendizaje automático se representan de forma algo distinta según el estudio consultado, aunque lo fundamental se mantiene en todos ellos. Podemos obtener de la figura observaciones clave para entender el auge del aprendizaje profundo y su nivel de rendimiento:

- Si se quiere alcanzar un alto nivel de rendimiento, es necesario a menudo entrenar una red neuronal lo suficientemente grande como para aprovechar la enorme cantidad de datos. La limitación en el tamaño de las redes neuronales y su tiempo de entrenamiento ha dependido de los avances realizados en CPUs y, en los últimos años, GPUs. Es decir, del poder computacional.
- Se necesitan muchos datos para poder sobresalir por encima de los algoritmos tradicionales y redes neuronales de tamaños pequeño y mediano.

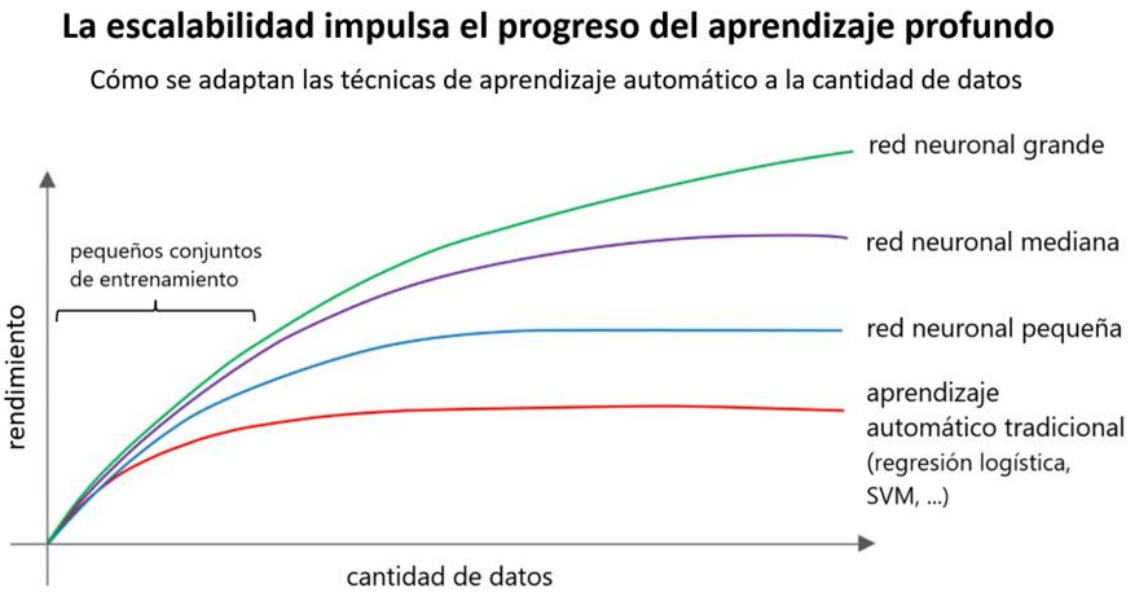


*Figura 9*. La escalabilidad impulsa el progreso del aprendizaje profundo: cómo se adaptan las técnicas de aprendizaje automático a la cantidad de datos.

Por ende, a menudo se dice que la escalabilidad ha impulsado el progreso del aprendizaje profundo, y por escala me refiero tanto al tamaño de la red neuronal, es decir, una red con muchas unidades ocultas, capas, parámetros y conexiones, como a la escala de los datos. De hecho, hoy en día una de las formas más fiables de mejorar el rendimiento de una red neuronal es entrenar una red más grande o utilizar más datos. Eso sólo funciona hasta cierto punto, porque al final no se pueden obtener más datos, o la red es tan grande que tarda demasiado en entrenarse. Pero mejorar la escalabilidad ha llevado muy lejos en el mundo del aprendizaje automático. En conjuntos de entrenamiento más pequeños, las redes neuronales no tienen por qué tener mejor rendimiento necesariamente que los algoritmos tradicionales, como se ve en la **figura 9**. Suele depender de la habilidad de los ingenieros para ajustar parámetros o elegir características correctas, por lo que podría suceder que un algoritmo SVM tuviese mejor rendimiento que una red neuronal grande en un conjunto pequeño de entrenamiento (unos pocos miles de entradas, por ejemplo). Esto es necesario matizarlo porque no todos los problemas de aprendizaje automático requieren de grandes y costosas redes neuronales. Pero para conjuntos de entrenamientos grandes de cientos de miles, o millones de datos de entrada, las redes neuronales han demostrado ser las reinas de los algoritmos de aprendizaje automático.

Se han mencionado hasta ahora dos factores clave en el auge del aprendizaje profundo: la cantidad de datos disponibles y el poder computacional para poder entrenar grandes redes neuronales. Pero falta un tercero: el uso de nuevos y mejores algoritmos. En los últimos años ha ocurrido una enorme innovación algorítmica que no se debe subestimar. Muchas de las innovaciones algorítmicas han consistido en intentar reducir el tiempo de entrenamiento de las redes neuronales, es decir, ayudar y agilizar los procesos en el hardware. Como ejemplo concreto, uno de los avances de las redes neuronales fue cambiar la función de activación sigmoidea por otro tipo de funciones, como la ReLU. Resulta que uno de los problemas del uso de funciones sigmoidales en el aprendizaje automático es que hay regiones donde la pendiente de la función, el gradiente, es casi cero, por lo que el aprendizaje se vuelve muy lento en los métodos de aprendizaje basados en descenso estocástico de gradientes y retropropagación. Este problema es conocido como el problema de desvanecimiento o explosión del gradiente. Al cambiar la función de activación de la red neuronal para utilizar esta función, ReLU, el gradiente es igual a 1 para todos los valores positivos de entrada y 0 para los valores. Aunque la pendiente de los valores negativos es 0 en esta función, resulta que es mucho menos probable que el gradiente se reduzca gradualmente hasta 0, y el simple hecho de cambiar de la función sigmoidea a la función ReLU hace que el algoritmo de aprendizaje por descenso del gradiente funcione

mucho más rápido y mejor. Hay muchos ejemplos como este en donde cambiando el algoritmo el código se ejecuta mucho más rápido y eficientemente, permitiendo entrenar redes neuronales mayores.

Otra razón por la que es importante que el entrenamiento de las redes neuronales sea rápido es debido a que el proceso de entrenamiento es muy intuitivo. A menudo se tiene una idea para la arquitectura de una red neuronal y entonces se implementa la idea en código. El código permite realizar pruebas con una fracción de los datos, cuyos resultados indican cómo funciona la red neuronal y, según los resultados, se vuelve a cambiar los detalles la red y continuaría el ciclo idea-código-pruebas hasta llegar al punto deseado. Cuando una red neuronal tarda mucho tiempo en entrenarse, conlleva una demora en dar vueltas a este ciclo y se nota una gran diferencia en la productividad. Es más sencillo construir redes neuronales eficaces cuando se implementa una idea y se ven los resultados en diez minutos, o a lo sumo un día, frente a si se tiene que entrenar la red neuronal durante un mes, lo que permite probar muchas más ideas y tener muchas más posibilidades de descubrir en la red lo que funciona bien, y lo que no, para la aplicación que se está desarrollando. Así que un cálculo más rápido ayuda en términos de acelerar la velocidad a la que se puede obtener un resultado experimental, y esto realmente ha ayudado tanto a los profesionales de las redes neuronales, así como los investigadores que trabajan en el aprendizaje profundo, a iterar mucho más rápido.

La combinación de disponibilidad de datos, hardware especializado más potente, y mejores algoritmos ha permitido el auge de las redes neuronales y el aprendizaje profundo. Y estos elementos siguen desarrollándose, por lo que en los próximos años se verán muchísimos avances en este campo. ¿Serán estos avances tan disruptivos como los de los últimos años? ¿Se cumplirán las expectativas? ¿Llegaremos a ver sistemas realmente inteligentes?

## Especulaciones sobre la inteligencia general artificial

Desde la revolución industrial y el auge de la tecnología, uno de los sueños (y de los miedos) del ser humano ha sido crear máquinas tan inteligentes y capaces, o más, que el propio ser humano. Actualmente, el sueño de construir algún día un sistema de inteligencia artificial tan inteligente como un ser humano es uno de los más inspiradores de este campo. El camino para conseguirlo no está claro y podría ser muy difícil. Nadie sabe si este camino llevará sólo unas décadas y veremos avances significativos en el transcurso de nuestras vidas, o si llevará siglos o milenios llegar hasta ahí. Echemos un vistazo a cómo es este sueño de la AGI (*Artificial General Intelligence*), la Inteligencia Artificial General (también llamada Inteligencia Artificial fuerte), y consideremos cuáles podrían ser los caminos posibles, aunque poco claros y difíciles, para llegar allí algún día. La mayoría de los expertos consideran que se ha magnificado innecesariamente a la AGI. Quizá uno de los motivos sea que la IA incluye, o se ramifica, en dos áreas muy diferentes. Una es la inteligencia artificial estrecha o débil, ANI (*Artificial Narrow Intelligence*). Se trata de un sistema de IA que hace una tarea concreta, a veces increíblemente bien, y puede ser muy valioso, como los asistentes inteligentes, el coche autónomo, la búsqueda web, o la IA aplicada a actividades como la agricultura, la industria o la medicina. En los últimos años, la inteligencia artificial ha progresado enormemente y está creando un enorme valor en el mundo actual. Dado que la ANI es un subconjunto de la IA, su rápido progreso hace lógicamente cierto que la IA también haya progresado enormemente en la última década.

Hay otra área diferente en la IA, que es la AGI ya mencionada. Existe la esperanza de construir sistemas de IA que puedan hacer cualquier cosa que pueda hacer un humano normal. A pesar de todos los avances de la ANI y, por tanto, de los enormes progresos de la IA, los expertos no creen que estemos avanzando hacia la AGI. Se cree que todos los progresos en la inteligencia artificial han hecho que la gente concluya, correctamente, que hay enormes avances en la IA, pero eso ha provocado que el público general también concluya que muchos avances en IA significan necesariamente que hay muchos avances hacia la AGI. La siguiente figura resume los párrafos anteriores.

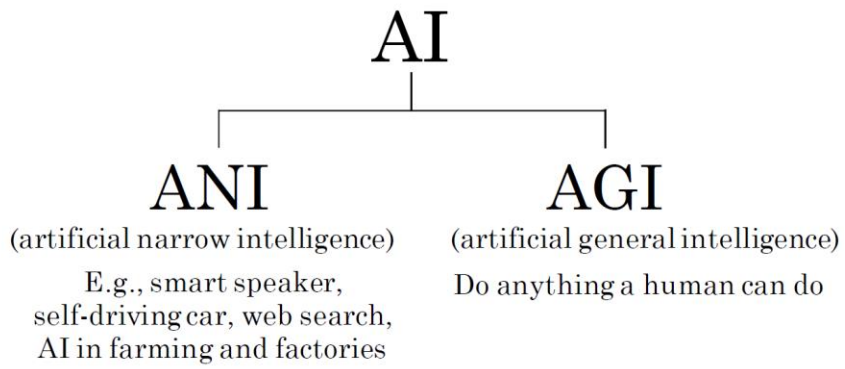


*Figura 10*. Áreas principales en las que se subdivide la inteligencia artificial.

Con el auge del aprendizaje profundo moderno se ha empezado a simular neuronas en los últimos años. Con ordenadores cada vez más rápidos e incluso GPUs, los investigadores son capaces de simular más neuronas cada año. Uno de los enfoques y esperanzas en este campo es que, si se pudieran simular muchas neuronas, se podría simular el cerebro humano, o algo parecido a un cerebro humano, y se dispondría de sistemas realmente inteligentes. Lamentablemente, no está resultando tan sencillo. Se pueden argumentar dos razones principales para ello:

- Las redes neuronales artificiales y sus capas no se parecen a las funciones que realiza cualquier neurona biológica. Estas redes son mucho más simples en partes y funciones.
- Actualmente quedan muchas preguntas fundamentales sin resolver sobre el funcionamiento del cerebro humano. Por ejemplo, exactamente cómo una neurona traza un mapa de entradas y salidas. ¿Cómo se puede replicar aquello de lo que se desconoce su funcionamiento? Intentar simularlo en un ordenador con los actuales algoritmos está muy lejos de ser un modelo exacto de lo que hace realmente el cerebro humano. Dada nuestra limitada comprensión del funcionamiento del cerebro humano, tanto actual como probablemente en un futuro próximo, intentar simularlo como vía hacia la inteligencia artificial será un camino increíblemente difícil.

Dicho esto, ¿hay alguna esperanza de que en nuestro tiempo veamos avances importantes en la AGI? Sí, hay evidencias que demuestran que la AGI puede ser algo posible en el futuro. Se han realizado algunos experimentos fascinantes con animales que demuestran o sugieren claramente que la misma zona de tejido cerebral biológico puede realizar una gama sorprendentemente amplia de tareas. Esto ha llevado a la hipótesis del "algoritmo de aprendizaje único", según la cual quizá gran parte de la inteligencia podría deberse a uno o a un pequeño número de algoritmos de aprendizaje. Si se pudiera averiguar cuáles son esos algoritmos, quizá algún día se podrían implementar en un ordenador. Veamos algunos detalles de esos experimentos.

El primero que quisiera mencionar es un resultado debido al estudio de *Roe et al*., *Visual projections routed to the auditory pathway in ferrets: receptive fields of visual neurons in primary auditory cortex*, de 1992. La parte del cerebro que se muestra en la **figura 11** es la corteza auditiva. El cerebro está configurado para enviar señales desde los oídos en forma de impulsos eléctricos, dependiendo del sonido que el oído detecte en la corteza auditiva. Resulta que si se reconfigura el cerebro de un animal, cortando el enlace entre el oído y la corteza auditiva, y en su lugar se alimentan imágenes a la corteza auditiva, entonces la corteza auditiva aprende a ver. Así, esta parte del cerebro que aprende a oír, cuando se le alimenta con datos diferentes, puede aprender a ver.

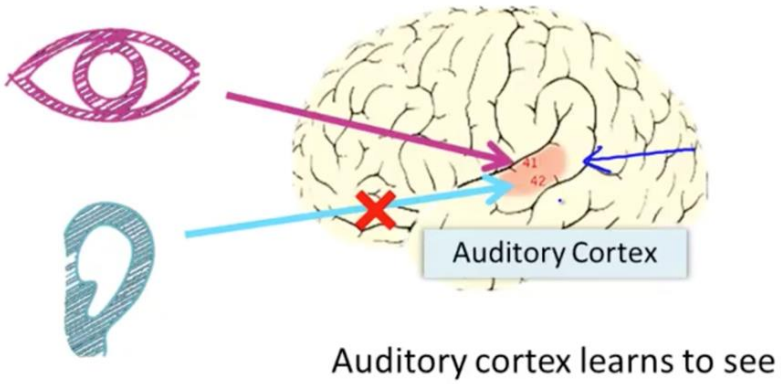


*Figura 11*. De acuerdo con el estudio de *Roe et al*. la corteza auditiva podría llegar a cumplir otras funciones sensoriales.

El segundo es debido a *Metin y Frost* en un estudio también de hace décadas llamado *Visual responses of neurons in somatosensory cortex of hamsters with experimentally induced retinal projections to somatosensory thalamus*. La parte del cerebro que se encarga del procesamiento del tacto es el córtex somatosensorial. Si se reconfigurase el cerebro de forma similar al ejemplo anterior para cortar la conexión de los sensores táctiles a esa parte del cerebro, y en su lugar alimentarlo con imágenes, entonces el córtex somatosensorial aprendería a ver. Ha habido una serie de experimentos de este tipo, mostrando que muchas partes diferentes del cerebro, en función de los datos que reciben, pueden aprender a ver, sentir, u oír, como si tal vez hubiera un algoritmo que, en función de los datos que recibe, aprende a procesar las entradas en consecuencia.

Se han construido sistemas que, mediante una cámara que puede ser montada en la frente de una persona, son capaces de mapear un patrón de voltajes en una cuadrícula en la lengua de dicha persona. Mediante el mapeo de una imagen en escala de grises a un patrón de voltajes en la lengua, esto puede ayudar a las personas invidentes a ver con la lengua. Uno de estos sistemas es la tecnología *BrainPort*. También se han llevado a cabo fascinantes experimentos con la ecolocalización humana o el sonar humano. Animales como los delfines y los murciélagos utilizan el sonar para ver, y los investigadores han descubierto que si se entrena a los humanos para que emitan chasquidos y escuchen cómo rebotan en el entorno, a veces pueden aprender cierto grado de ecolocalización humana. Esto se puede ver en el trabajo *Foundations of Orientation and Mobility*, de Blasch, Wiener y Welsh de 1997. En él, los autores describen cómo las personas pueden usar la ecolocalización para detectar objetos en su entorno y cómo esta habilidad puede ser entrenada. El estudio se realizó con un grupo de personas invidentes que habían desarrollado la capacidad de chasquear la lengua y escuchar el eco para determinar la ubicación de los objetos. Los autores midieron el espectro y el nivel de presión sonora de los chasquidos emitidos por los participantes a diferentes distancias del objeto. Los resultados mostraron que los participantes podían detectar objetos a distancias de hasta 4 metros. Otro ejemplo es un sistema llamado cinturón háptico. Una explicación sencilla de su funcionamiento sería la siguiente: si se monta un anillo de zumbadores alrededor de la cintura y se programa usando una brújula magnética de modo que, por ejemplo, los zumbadores en la dirección más al Norte estén siempre vibrando lentamente, de alguna manera se adquiere un sentido de la orientación, que algunos animales tienen, pero los humanos no. Entonces al caminar se percibiría dónde está el norte, pero sin sentir que esa parte de la cintura está vibrando, como una certeza. Esto ha sido analizado en varios estudios, siendo el primero el conducido por *Nagel et al*. con el cinturón *FeelSpace*. Un último ejemplo serían las cirugías de implantación. Por ejemplo, *Constantine-Paton y Law* consiguieron implantar un tercer ojo en una rana y comprobaron cómo su cerebro se adaptaba y aprendía de esta nueva entrada de información.

Ha habido una gran variedad de experimentos como estos que muestran que el cerebro humano es increíblemente adaptable. Los neurocientíficos dicen que es increíblemente plástico, es decir, increíblemente adaptable a la amplia gama de estímulos de entrada que recibimos los seres humanos. De esta manera la

pregunta es: si la misma pieza de tejido cerebral puede aprender a ver, tocar, sentir, o incluso otras habilidades, ¿podemos replicar este algoritmo de aprendizaje e implementarlo en un ordenador? La AGI es uno de los problemas científicos y de ingeniería más fascinantes de todos los tiempos, sin lugar a duda. Sin embargo, es igual de importante evitar exageraciones. Como se ha mencionado, no se sabe si el cerebro consiste en uno o varios algoritmos de aprendizaje o si siquiera se puede reducir a eso. En cualquier caso y a corto plazo, sin tener en cuenta la búsqueda de la AGI, el aprendizaje automático y las redes neuronales son una herramienta muy poderosa en nuestro mundo.

Respecto a los miedos a esta tecnología y sus efectos, recientemente se ha llegado a proponer pausar su desarrollo. En una carta abierta, más de 1.000 líderes y expertos en tecnología han pedido poner en pausa el desarrollo de la inteligencia artificial debido a los "riesgos profundos para la sociedad y la humanidad" que presentan las herramientas de inteligencia artificial más avanzadas. Han instado a los laboratorios de inteligencia artificial a que detengan el desarrollo de los sistemas más avanzados y piden a los desarrolladores de inteligencia artificial que "detengan inmediatamente durante al menos 6 meses el entrenamiento de sistemas de IA más poderosos que GPT-4". ¿Qué es dicha herramienta que se menciona, GPT-4? A continuación, veamos brevemente parte del estado del arte de la IA: los sistemas de IA más desarrollados en la actualidad y candidatos más cercanos hasta la fecha para llegar a convertirse en AGIs.

## GPT y ChatGPT

ChatGPT es un prototipo de *chatbot* (IA conversacional) de inteligencia artificial desarrollado en 2022 por la empresa OpenAI que se especializa en el diálogo. Este *chatbot* es un gran modelo de lenguaje, ajustado con técnicas de aprendizaje tanto supervisadas como de refuerzo. Se basa en el modelo GPT-4 de la misma empresa OpenAI, una versión mejorada de su modelo anterior, GPT-3. Fue entrenado mediante el método RLHF (*Reinforcement Learning from Human Feedback*, Aprendizaje por refuerzo a partir de información humana). El modelo inicial se entrenó mediante un ajuste fino supervisado: instructores humanos de IA proporcionaron conversaciones en las que interpretaban a ambas partes (el usuario y un asistente de IA). Este nuevo conjunto de datos de diálogo fue mezclado con el conjunto de datos de InstructGPT, un modelo hermano que está capacitado para seguir una instrucción en un *prompt* (entrada) y dar una respuesta detallada, los cuales fueron transformados en un formato de diálogo.

Para crear un modelo de recompensa para el aprendizaje por refuerzo, fue necesario que recopilasen datos de comparación, que consistían en dos o más respuestas del modelo clasificadas por calidad. Para recopilar estos datos, tomaron conversaciones que los instructores de IA mantenían con el *chatbot*. El proceso consistía en seleccionar al azar un mensaje escrito por el modelo, muestrear varias respuestas alternativas y pedir a los instructores de IA que las clasificaran. Con estos modelos de recompensa, pudieron ajustar el modelo mediante el algoritmo de Optimización de Política Próxima (PPO). Realizaron varias iteraciones de este proceso.

### *ChatGPT 4*

ChatGPT 4 es la iteración más novedosa y potente liberada hasta la fecha por OpenAI, basándose en los cimientos establecidos por su antecesor, GPT-3. Fue lanzada oficialmente el 13 de marzo de 2023 y, para sorpresa y alegría de muchos, este modelo fue integrado en el navegador Bing de la empresa Microsoft. De esta manera se solventa un problema que tiene ChatGPT, que es poder proveer de respuestas actuales y con referencias a webs. ChatGPT 3 funcionaba en base a más de 175 millones de parámetros. Es difícil saber con cuántos parámetros funciona la versión 4, porque Open AI no ha dado ese dato a fecha de publicación de este trabajo, pero podemos imaginar que la cantidad se ha multiplicado exponencialmente. Para describir sus capacidades hay que compararlo con el modelo anterior, ChatGPT 3.

| Característica | ChatGPT 3 | ChatGPT 4 |
|---|---|---|
| Parámetros | 175 millones | Más que ChatGPT 3 (número exacto no divulgado) |
| Comprensión del lenguaje | Impresionante, pero menos avanzado que ChatGPT 4 | Más avanzado, mejor comprensión del contexto, los matices y las sutilezas. |
| Comprensión contextual | Dificultades con conversaciones más largas | Mejorado, mejor para mantener la relevancia durante interacciones extendidas |
| Resolución de ambigüedad | Menos capaz | Mejor, puede hacer preguntas aclaratorias y cubrir múltiples interpretaciones |
| Creatividad y resolución de problemas | Capaz, pero menos avanzado | Ideas mejoradas, más diversas e innovadoras |
| Razonamiento lógico | Capaz, pero menos avanzado | Más robusto, mejor en la resolución de problemas complejos y en sacar conclusiones precisas |
| Pensamiento analógico | Menos avanzado | Mejorado, mejor dibujando analogías y haciendo conexiones. |
| Limitaciones | Puede generar respuestas incorrectas o sin sentido, puede ser sensible a las frases de entrada, no puede acceder a datos en tiempo real o posteriores al entrenamiento | Limitaciones similares, pero con algunas mejoras en la precisión y el procesamiento de la información. |
| Ética | Requiere investigación y colaboración continuas para garantizar un uso responsable | Preocupaciones éticas similares, con OpenAI implementando mitigaciones de seguridad |
| Modalidades de entrada | Texto | Texto, Audio, Imagen, Visual |
| Análisis de imagen | No disponible | Disponible |
| Comprensión del lenguaje | Limitado a unos pocos | Puede entender más idiomas |
| Exactitud | Menos precisa | Más preciso |
| Capacidad de *tokens* | 4096 *tokens*, o aproximadamente 8000 palabras | 32 768 *tokens*, o más de 64 000 palabras |

*Tabla 2*. Características de GPT 3 vs. GPT 4.

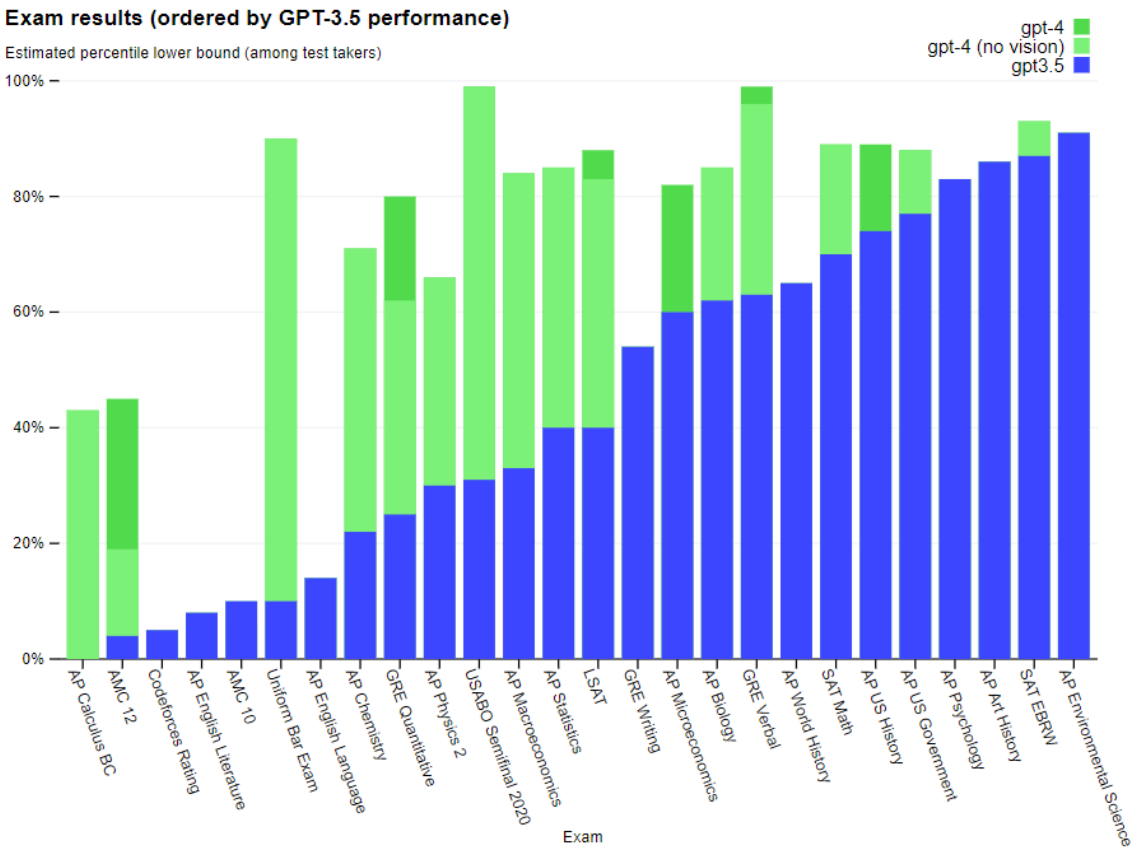


*Figura 12*. Gráfica comparativa de resultados de los modelos GPT 3 y 4 en diferentes exámenes estadounidenses.

Sorprendentemente, GPT 4 también descifró algunos exámenes de alto nivel como SAT, GRE y más. OpenAI afirma que GPT-4 podría colocarse entre el 10% de los mejores estudiantes en el "examen de la abogacía" estadounidense, que habilita para ejercer como abogado en 41 estados. También puede obtener una puntuación de 1.300 (sobre 1.600) en el SAT (un examen estandarizado que se usa extensamente para la admisión universitaria) y un cinco (sobre cinco) en los exámenes de bachillerato *Advanced Placement* de biología, cálculo, macroeconomía, psicología, estadística e historia, según las pruebas de la empresa. Esto ha sido de lo más comentado de cara al público no especializado y, la siguiente gráfica de rendimiento, de las más reproducidas tras su lanzamiento.

### *Competidores: LaMDA y Bard de Google*

La llegada de ChatGPT ha generado una gran expectación en el sector tecnológico, lo que ha llevado a grandes empresas a replantearse su estrategia. Sin embargo, para algunas compañías como Microsoft y Google, el trabajo con inteligencia artificial y el desarrollo de *chatbots* y modelos no es algo nuevo. De hecho, muchas empresas líderes en el sector de las tecnologías de la información llevan meses e incluso años trabajando con IA. Google, por ejemplo, lleva años desarrollando *chatbots* y modelos de lenguaje como BERT y LaMDA, este último considerado el más avanzado.

Google, que lidera de forma incontestable el mercado de las búsquedas en Internet, ha visto su posición amenazada con la llegada de modelos de IA capaces de ofrecer información de forma conversacional. Si se quiere buscar algo en Google, hay que mirar entre sus webs, mientras que IAs como ChatGPT están entrenadas para darte una respuesta directa. Desde el primer momento, Google ya tenía un competidor directo a ChatGPT, LaMDA. LaMDA es un modelo de lenguaje neuronal conversacional desarrollado por Google desde 2017. LaMDA significa *Language Model for Dialogue Applications* (Modelo de

Lenguaje para Aplicaciones de Diálogo) y se desarrolló a partir de la red neuronal *open source* de Google, *Transformer*, un entramado de redes neuronales artificiales profundas. Este modelo fue diseñado para dotar al software de la capacidad de interactuar mejor en una conversación natural y fluida. LaMDA se entrenó con un conjunto de datos de 1.56 TB de palabras. El modelo es capaz de mantener conversaciones fluidas y naturales con los usuarios, incluso en temas complejos.

Sin embargo, la empresa del buscador prefirió no sacarlo del laboratorio para no dañar su reputación, ya que no consideraban que todavía estuviera listo. No obstante, poco después y ante el aumento de la popularidad de ChatGPT, decidieron activar un "código rojo" para lanzar cuanto antes su propia alternativa y no quedarse así descolgados del auge que están viviendo las inteligencias artificiales. Para tratar de competir con ChatGPT, Google ha desarrollado rápidamente un sistema conversacional o *chatbot* con una versión reducida de LaMDA, Bard, que requiere menos potencia de cálculo y así podrá ser usado por más personas. Este modelo llevaba un tiempo en fase de pruebas muy cerrada, ya que solo unas pocas personas podían acceder a ella. Con esto, podrán obtener comentarios y feedback de los usuarios que lo usan, lo que combinado con las propias pruebas internas de Google ayudará a seguir desarrollando la IA y mejorando sus respuestas. Al igual que Microsoft ha integrado ChatGPT 4 en Bing, Google integrará Bard en su función de búsquedas. A fecha de redacción de este trabajo, Google Bard no ha sido lanzado de forma oficial, por lo que aún no se tiene toda la información sobre sus capacidades. La empresa intentó hacer un lanzamiento prematuro para contrarrestar el lanzamiento de GPT 4 en Bing por parte de Microsoft a principios de febrero de 2023, con resultados casi catastróficos. Una respuesta incorrecta a una consulta sobre el telescopio James Webb le costó a la empresa matriz, Alphabet, un 7% en acciones, equivalente a unos $100.000 millones de dólares.

### *Otros competidores: LlaMa de Meta y Bedrock de Amazon*

Por supuesto, las otras megacorporaciones del sector tecnológico no han querido quedarse atrás y están desarrollando a toda prisa sus propias inteligencias artificiales. Están mucho menos avanzadas que las anteriormente mencionadas, por lo que simplemente se hace un apunte aquí de su existencia.

Meta lleva unos años enfocada principalmente en el desarrollo del Metaverso, ya que su CEO Mark Zuckerberg apostó porque esta tecnología sería la puntera. De momento parece equivocado en su predicción y ello ha tenido tres grandes consecuencias: la primera, que miles de trabajadores hayan perdido su trabajo; en segundo lugar, la compañía ha perdido cientos de millones invertidos en el desarrollo de este mundo virtual y, por último, ha provocado que Meta esté a la cola en la carrera de la IA. Meta anunció su IA, *LlaMa*, en febrero de 2023, que estará pronto disponible con una licencia no comercial con el objetivo de ser utilizada en investigación u otros ámbitos académicos. Según la información disponible, no hay grandes diferencias en comparación con las IAs actuales. Las más reseñables son el uso de datos públicos, lo que posibilita que el trabajo de investigación sea compatible con el código abierto, y que será un modelo más sencillo que sus competidores, por lo que los resultados podrían ser más concretos y simples, pero con menos errores.

Por su parte, en su carta anual para accionistas, el CEO de Amazon, Andy Jassy, anunció que la inteligencia artificial generativa y los modelos grandes de lenguaje transformarán la compañía. Amazon está haciendo estas tecnologías más accesibles a través de su nueva plataforma de IA llamada Bedrock, la cual es presentada como un servicio en la nube ofrecido por *Amazon Web Services*. Bedrock permite a los desarrolladores mejorar su software con sistemas de IA generativa similares al motor detrás de ChatGPT, desarrollado por OpenAI y respaldado por Microsoft. En su fase piloto, Bedrock permite a los usuarios ejecutar funciones como generar texto para publicaciones de blog, correos electrónicos y otros documentos, así como utilizar herramientas de búsqueda y convertir textos en imágenes mediante la función *Stability AI*.

Con toda esta información se ha pretendido ilustrar, de forma general, de dónde vienen las redes neuronales y el aprendizaje profundo, qué son, su estado actual y dónde pretenden estar en el futuro. A continuación, comenzaremos a construir y explicar el funcionamiento de redes neuronales, desde una adaptación de los algoritmos de aprendizaje clásicos hasta redes neuronales de capas personalizables.

# 2

# Conceptos básicos de las redes neuronales

Este primer bloque versa sobre los fundamentos de las redes neuronales y su implementación en un lenguaje de programación, como es Python. Resulta que cuando se implementa una red neuronal, hay algunas técnicas que son realmente importantes. Por ejemplo, si se tiene un conjunto de entrenamiento de $m$ ejemplos, puede que la primera idea que se tenga para procesarlos sea con un bucle *for* que recorra los $m$ ejemplos de entrenamiento; sin embargo, al implementar una red neuronal, es preferible procesar todo el conjunto de entrenamiento sin usar dicho bucle. Veremos el porqué de esto y más técnicas aquí y en los anexos. Por otra parte, cuando se organiza el cálculo de una red neuronal, habitualmente se tiene lo que se llama un paso de propagación hacia delante (*forward propagation*, en inglés), seguido de un paso de propagación hacia atrás (*backward propagation*, en inglés). En este bloque también se hará una introducción acerca de por qué los cálculos en el aprendizaje de una red neuronal se pueden organizar en estos pasos de propagación hacia adelante y una propagación hacia atrás por separado.

La intención en este primer bloque es transmitir estas ideas de la forma más sencilla e intuitiva posible e ir escalándolas en siguientes bloques. Para ello, utilizaré un algoritmo de aprendizaje automático clásico: la regresión logística. La regresión logística es un algoritmo de clasificación y se utilizará para ejemplificar la más sencilla posible, la clasificación binaria, por lo que comenzaremos describiendo dicha clasificación. Recordemos que el archivo de referencia es el **archivo 1**.

## Clasificación binaria

En un problema de clasificación binaria, el resultado es una salida con un valor discreto. Por ejemplo, una cuenta hackeada (1) o no hackeada (0), un tumor maligno (1) o benigno (0), email spam (1) o no spam (0), etc. Uno de los objetivos prácticos en este trabajo, en el que se aplicará la teoría descrita sobre redes neuronales, será entrenar un clasificador para el cual la entrada es una imagen representada por un vector de características, $x$, que pueda predecir si la etiqueta correspondiente, $y$, es 1 o 0, significando 1 que sí es la etiqueta buscada y 0, que no. Primero de todo, veamos cómo se representa una imagen en un ordenador.

Una imagen se almacena en el ordenador en tres matrices separadas que corresponden a los canales de color rojo, verde y azul de la imagen. Las tres matrices tienen las mismas dimensiones que la imagen; por ejemplo, si la resolución de la imagen es de 64 píxeles x 64 píxeles, las tres matrices (RGB) son de 64 x 64 cada una. En la **figura 13** se muestra un ejemplo donde la imagen de entrada es un coche.

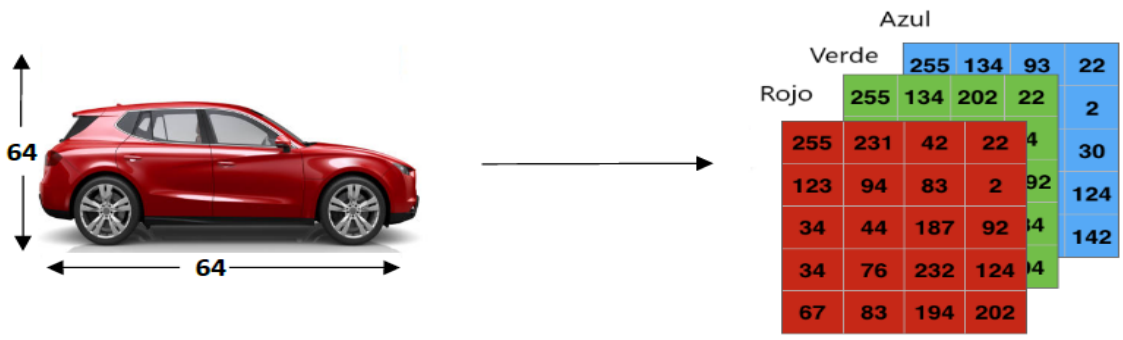


*Figura 13*. Representación de una imagen de 64 x 64 píxeles en un ordenador como matrices RGB.

El valor de cada celda en la **figura 13** representa la intensidad del píxel que se utilizará para crear un vector de características de dimensión $n$. En reconocimiento de patrones y aprendizaje automático, un vector de características representa una imagen. La tarea del clasificador consiste en determinar si el vector de características contiene o no el tipo de imagen para el cual ha sido entrenado para detectar.

Para crear un vector de características, $x$, los valores de intensidad de los píxeles serán reagrupados o "remodelados" para cada color, transformando cada matriz 3D en un vector 1D. A continuación, cada uno de esos vectores 1D se combina en un único vector, como se ve en la **figura 14** . La dimensión del vector de características de entrada en el ejemplo de esta figura sería $n = n_x = 64 \times 64 \times 3 = 12288$.

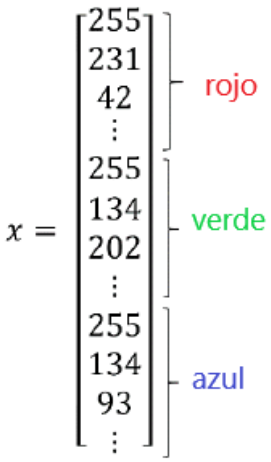


*Figura 14*. Vector de características de una imagen RGB.

Así, en la clasificación binaria de imágenes, el objetivo es que un clasificador aprenda a través de datos de entrada con la forma del vector de características, $x$, y sea capaz de predecir si la etiqueta correspondiente, $y$, es 1 o 0 (correcto/incorrecto, coche/no coche, etc). La notación que se utilizará en las explicaciones será la siguiente:

$$(x^{(i)}, y^{(i)}), \quad x^{(i)} \in \mathbb{R}^{n_x}, \; y^{(i)} \in \{0, 1\}$$

$m$ ejemplos de entrenamiento: $\{(x^{(1)}, y^{(1)}), (x^{(1)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

$m = m_{\text{train}}$ ; $m_{\text{test}}$ = # de ejemplos de test

Cada ejemplo de entrenamiento estará representado por un par, $(x^{(i)}, y^{(i)})$ donde $x$ es un vector de características n-dimensional e $y$, la etiqueta, será 0 o 1. Los conjuntos de entrenamiento se componen de $m$ ejemplos de entrenamiento, donde $(x^{(1)}, y^{(1)})$ son la entrada y etiqueta del primer ejemplo de entrenamiento, $(x^{(2)}, y^{(2)})$ el segundo ejemplo de entrenamiento, hasta $(x^{(m)}, y^{(m)})$, que es el último ejemplo de entrenamiento. Para enfatizar que la referencia es al número de casos de entrenamiento, se podrá escribir con el subíndice "*train*" y para el conjunto de datos de prueba, se utilizará el subíndice "*test*". El uso de estos términos anglosajones se debe, en primer lugar, a que es la denotación común en el ámbito de las redes neuronales; en segundo lugar, son palabras más cortas que sus equivalentes "entrenamiento" y "prueba", respectivamente.

Por último, para agrupar todos los ejemplos de entrenamiento en una notación más compacta, definamos una matriz, $X$, tomando el conjunto de entradas de entrenamiento ($m$ vectores de características $n$-dimensionales) y apilándolos en columnas. Esta matriz tendrá por tanto $m$ columnas y $n$ filas. Es importante señalar que el agrupamiento en **vectores columna** es una convención en redes neuronales que facilita la implementación, pero que también se podrían agrupar los datos en vectores fila. En Python, las dimensiones se pueden obtener con el método *.shape()*. Con respecto a las etiquetas de salida, para facilitar la implementación de una red neuronal, también es conveniente apilar los datos en vectores columna. Como

cada imagen tiene un valor por etiqueta, 1 o 0, la matriz $Y$ tendrá una fila y $m$ columnas. Más información en el anexo 1.

$$X = \underbrace{\begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}}_{m} \Big\} n_x$$

$$X \in \mathbb{R}^{n_x \times m} \rightarrow Python: X.shape \ = \ (n_x, m)$$

$$Y = \left[y^{(1)}, y^{(2)}, \dots, y^{(m)}\right], Y \in \mathbb{R}^{1 \times m} \rightarrow Python: Y.shape \ = \ (1, m)$$

Habiendo explicado el problema de clasificación binaria y la notación que se utilizará, veamos a continuación el caso del algoritmo de aprendizaje clásico regresión logística, el cual utilizaremos finalmente para ejemplificar el tipo de red neuronal más sencillo posible.

## Regresión logística

La regresión logística es un algoritmo de aprendizaje utilizado en problemas de aprendizaje supervisado cuando los valores de salida, $y$, son todos ceros o unos. Por lo tanto, es un algoritmo utilizado para clasificación binaria. El objetivo de la regresión logística es minimizar el error entre sus predicciones y los datos de entrenamiento. Siguiendo el ejemplo de la **figura 14** del apartado anterior, dada una imagen representada por un vector de características $x$, el algoritmo evaluará la probabilidad de que haya un coche en esa imagen:

$$\text{Dados : } x^{(i)}, \ \hat{y}^{(i)} = P(y^{(i)} = 1 \mid x^{(i)}), \text{ donde } 0 \leq \hat{y} \leq 1;$$

El superíndice *(i)* indica el i-ésimo ejemplo de entrenamiento. En este modelo de aprendizaje, se entrenan unos parámetros $W$ y $b$. Siendo $x^{(\mathrm{i})}$ un vector de dimensión $n_x$, el parámetro vector de pesos que lo multiplicará, $W$, también tendrá dimensión $n_x$ y el umbral o término independiente, $b$, será un número real. Recordemos que el producto escalar de dos vectores es un número real, al igual que el producto del primero traspuesto por el segundo. Ver anexo 2 para repasar conceptos de álgebra. Entonces, dada la entrada y sus parámetros, ¿cómo generamos la salida o predicción $\hat{y}^{(i)}$?

Una posible idea sería decir que la salida es la siguiente combinación lineal: $W^T x^{(i)} + b$. De hecho, esto es lo que se utilizaría en el caso de la regresión lineal. Sin embargo, esta expresión no es un algoritmo nada bueno para la clasificación binaria, puesto que el objetivo es conseguir un resultado entre 0 y 1, lo cual es difícil de cumplir porque esta expresión puede ser mucho mayor que 1 o incluso ser negativa. Y esos resultados no tienen sentido al hablar de probabilidades que se quieren restringir entre 0 y 1. Una solución es aplicar la función sigmoide al resultado de esta expresión para conseguir el rango de valores deseado. Resumen de los parámetros utilizados en la regresión logística:

- El vector de características de entrada: $x^{(i)} \in \mathbb{R}^{n_x}$, donde $n_x$ es el número de características.
- La etiqueta de entrenamiento: $y^{(i)} \in 0{,}1$.
- Los pesos: $W \in \mathbb{R}^{n_x}$, donde $n_x$ es el número de características de cada vector de entrada $x^{(i)}$ y $w_k$ es cada peso que conforma el vector.
- El umbral: $b \in \mathbb{R}$.
- La salida o predicción: $\hat{y}^{(i)} = \ \sigma\left(W^T x^{(i)} + b\right), \ \hat{y}^{(i)} \in [0{,}1]$
- La función sigmoide: $\mathrm{s} = \sigma\left(W^T x^{(i)} + b\right) = \sigma\left(z^{(i)}\right) = 1/1 + e^{-z^{(i)}}$.
- Su derivada: $\mathrm{s}' = \sigma\left(z^{(i)}\right)' = \ \sigma(z^{(i)})(1 - \ \sigma(z^{(i)}))$

Escribimos $W^T x^{(i)}$ porque, como se explica en el anexo 2 sobre producto escalar, el equivalente del producto escalar de los vectores $W$ y $x$, es el producto elemento a elemento del primero traspuesto por el segundo.

$(W^T x + b)$ es una función lineal $(\mathrm{ax} + b)$, pero como se busca una restricción de probabilidad [0,1], se utiliza la función sigmoidea. La función está acotada entre [0,1] como se muestra en el siguiente gráfico.

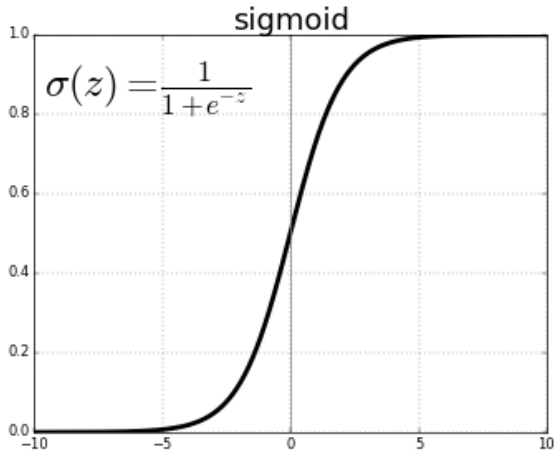


*Figura 15*. Representación gráfica de función sigmoide.

Podemos interpretar que cada peso, $w_k$, representa la influencia relativa de la entrada por la cual se multiplica, $x^{(i)}$. A menudo al término $b$ se le llama sesgo (en inglés *bias*), ya que controla qué tan predispuesta está la neurona a devolver un 1 o un 0 independiente de los pesos. Un sesgo alto hace que la neurona requiera una entrada más alta para generar una salida de 1; un sesgo bajo lo facilita. Algunas observaciones sobre el gráfico:

- Si $z$ es un número positivo grande, entonces $\sigma(z) = 1$
- Si $z$ es un número negativo, pequeño o grande, entonces $\sigma(z) = 0$
- Si $z = 0$, entonces $\sigma(z) = 0.5$

Al implementar la regresión logística, el objetivo es conseguir unos parámetros, $W$ y $b$, que hagan que $\hat{y}$ sea una buena estimación de la probabilidad de que $y$ sea igual a 1. En este trabajo, al programar las redes neuronales, se mantendrán los parámetros $W$ y $b$ por separado, aunque es posible utilizar otros enfoques donde, por ejemplo, los parámetros formen parte de un único vector.

## Regresión logística: Función de coste

Para entrenar los parámetros $W$ y $b$, es necesario definir una función de coste. Siendo $x^{(\mathrm{i})}$ el i-ésimo ejemplo de entrenamiento, dados $\{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\}$, se busca entrenar unos parámetros de tal forma que $\hat{y}^{(i)} \approx y^{(i)}$. Veamos qué función de pérdida o de error podemos utilizar para medir el rendimiento del algoritmo.

### *Función de pérdida o error*

La función de pérdida mide la discrepancia entre la predicción, $\hat{y}^{(i)}$, y el resultado deseado, $y^{(i)}$. En otras palabras, la función de pérdida calcula el error de cada ejemplo de entrenamiento con respecto a la predicción. Una opción sería utilizar la función de pérdida L2, es decir, la función de pérdida por error al cuadrado:

$$L(\hat{y}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})^2$$

Sin embargo, no es aconsejable utilizar esta función de pérdida al tratar con la regresión logística, dado que al buscar los valores de los parámetros el problema de optimización se vuelve no convexo, con múltiples óptimos locales. El algoritmo descenso del gradiente, que se verá en el siguiente apartado, no sería capaz de

encontrar un óptimo global. Por ello, se utiliza una función de pérdida diferente a L2 que juegue un papel similar pero dé como resultado un problema de optimización convexo, mucho más fácil de optimizar:

$$L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)\right) \qquad (1)$$

siendo $log$ el logaritmo natural. Esta pérdida es un caso especial de la entropía cruzada para cuando solo se tienen dos clases. Esta fórmula asume que $y$ es una probabilidad, así que están estrictamente en el rango de entre 0 y 1. Al igual que cuando se utiliza la función L2, el objetivo es que el valor de la función sea lo más pequeño posible. Para entender el por qué, veamos dos casos:

- Si $y^{(i)} = 1$: $L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\log\left(\hat{y}^{(i)}\right)$ donde $\log\left(\hat{y}^{(i)}\right)$ y $\hat{y}^{(i)}$ deben ser valores grandes para minimizar el error. Como el valor máximo de la función sigmoide es 1, su valor será aproximadamente 1.
- Si $y^{(i)} = 0$: $L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\log\left(1 - \hat{y}^{(i)}\right)$ donde $\log\left(1 - \hat{y}^{(i)}\right)$ e $\hat{y}^{(i)}$ deben ser valores pequeños para minimizar el error. Como el valor mínimo de la función sigmoide es 0, su valor será aproximadamente 0.

Esta no es la única función de pérdida posible que cumple estos comportamientos, pero se ha pretendido mostrar de forma poco exhaustiva la justificación de por qué es válida para la regresión logística. En uno de los anexos se da una justificación más formal. Hemos definido la función de pérdida con respecto a un solo ejemplo de entrenamiento. Definamos a continuación la función de coste, que mide el rendimiento en el conjunto de entrenamiento.

### *Función de coste*

Se trata de una función elegida previamente en base al tipo de problema concreto, para poder evaluar de la forma más adecuada la diferencia entre las predicciones de nuestra red y las salidas reales. La función de coste elegida es la media de la función de pérdida de todo el conjunto de entrenamiento. Su objetivo es el ya mencionado: encontrar los parámetros que minimizan la función de coste global.

$$J(\mathrm{W}, b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left[\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)\right]\right. \qquad (2)$$

Recapitulando, se ha explicado el algoritmo de regresión logística, la función de pérdida para cada ejemplo de entrenamiento, y la función de coste global según los parámetros del algoritmo. A continuación, se discutirá el algoritmo de aprendizaje que se utilizará no solo en la regresión logística, sino en todo el trabajo.

## Descenso del gradiente

El descenso del gradiente, o gradiente descendiente, es un algoritmo de optimización iterativo de primer orden que permite encontrar mínimos locales en una función diferenciable. La idea es tomar pasos de manera repetida en dirección contraria al gradiente. Esto se hace ya que esta dirección es la del descenso más empinado. Si se toman pasos con la misma dirección del gradiente, se encontrará el máximo local de la función; a esto se le conoce como el gradiente ascendente. El gradiente se calcula como la derivada multivariable de la función de coste con respecto a todos los parámetros de la red. Gráficamente sería la pendiente de la tangente a la función de coste en el punto donde nos encontramos (evaluando los pesos actuales) y matemáticamente es un vector que da la dirección y el sentido en que dicha función aumenta más rápido, por lo que hay que moverse en sentido contrario para minimizarla.

En palabras llanas, es un algoritmo que encuentra aquellos valores $W$ y $b$ que minimizan el valor de la función $J(W, b)$. Este algoritmo es válido para minimizar cualquier función, no solo la función coste para la regresión logística, y es usado ampliamente en el aprendizaje automático y profundo para entrenar algunas de las redes neuronales más avanzadas. No es el único, hay otros muy utilizados, como puede ser el algoritmo

*Adam*. Pero para explicar los fundamentos de las redes neuronales y el aprendizaje profundo, es una buena elección.

Pensemos en pocas dimensiones para poder visualizarlo. Si tuviéramos una red con sólo dos parámetros, podríamos representar la función de coste en tres dimensiones (plano XY para los valores de los parámetros, plano Z para el valor del coste). El gradiente en este caso será un vector de dos componentes (una en el eje X y otra en el eje Y), y nos marcará un sentido en el plano XY hacia donde deberíamos mover los parámetros para aumentar más rápido el coste. Su módulo será mayor cuanto mayor sea la pendiente de la tangente en el punto donde se calcula y, por consiguiente, será menor cuanto más cerca estemos de un mínimo de la función de coste (en un mínimo la pendiente es 0). La idea por tanto es movernos en sentido contrario al del gradiente, tratando de alcanzar el mínimo global. El mínimo global es el punto que tiene el menor valor posible para la función de coste $J(W, b)$ de entre todos los puntos posibles. Dependiendo de dónde se inicialicen los parámetros, se termina en un mínimo local u otro.

En la **figura 16** se muestra una ilustración del descenso del gradiente. En este diagrama, los ejes horizontales representan el plano de parámetros $W$ y $b$ (en la práctica $W$ puede tener más dimensiones, pero para facilitar la explicación, se ilustra $W$ como un valor unidimensional). La función de coste $J(W, b)$ es una superficie por encima de estos ejes horizontales. La altura de la superficie representa el valor de $J(W, b)$ en cada punto y lo que se pretende es encontrar los valores de $W$ y $b$ que corresponden al mínimo de la función de coste.

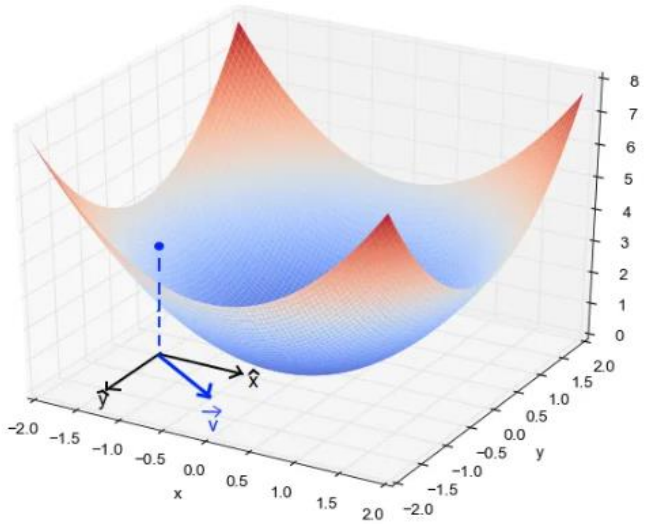


*Figura 16.* Función de coste con dos parámetros: un único valor para $W$ y $b$, representados por *x* e *y*. Se muestra también la dirección del gradiente, $\vec{v}$.

La realidad es que las funciones de coste son más complejas y no tienen únicamente dos parámetros o dimensiones, sino cientos, miles o incluso millones de ellos. Además, la superficie de la función presentará mínimos locales que durante el entrenamiento pueden confundirse con el mínimo global.

Si observamos la función de coste definida para la regresión logística, ecuación (2), es una función convexa. Esto significa que tiene forma de "cuenco", como la **figura 16**, por lo que solo tiene un mínimo, el cual es global, en contraposición con las funciones no convexas, las cuales tienen varios óptimos locales, como en la **figura 17**. Así que el hecho de que la ecuación (2) sea convexa es una de las grandes razones por las que utilizamos esta función particular para la regresión logística en vez de otras funciones de pérdida como L2. Ese tipo de funciones tienen múltiples posibles mínimos locales que dependen del punto de partida de los parámetros para su resultado final de optimización. Hay otro parámetro muy importante a tener en cuenta en este algoritmo: la tasa de aprendizaje.

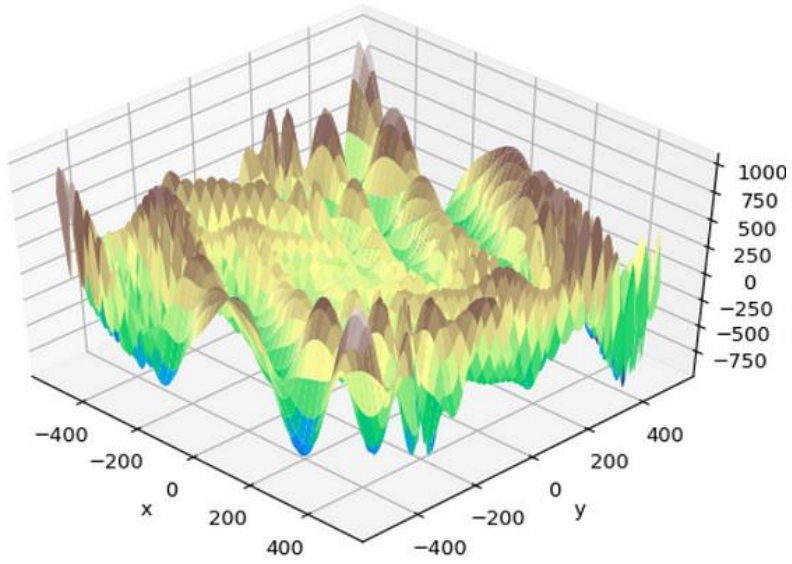


*Figura 17.* Representación de una función de coste con múltiples parámetros

## Tasa de aprendizaje

También conocida como tamaño de paso o alfa ($\alpha$), controla cómo de grande es el paso de la derivada hacia el mínimo. Suele ser un valor pequeño entre 0 y 1, y se evalúa y actualiza en función del comportamiento de la función de coste. La elección de la tasa de aprendizaje es muy importante, puesto que tendrá un gran impacto en la eficiencia de la implementación del descenso del gradiente. De hecho, si se elige mal, el algoritmo puede no llegar a funcionar. Si su valor es muy grande, el gradiente puede divergir o "rebotar" y nunca alcanzar el mínimo. Si su valor es muy pequeño, el algoritmo puede ser lento. Si bien una tasa pequeña tiene la ventaja de una mayor precisión, el número de iteraciones compromete la eficiencia general, ya que esto requiere más tiempo y cálculos para alcanzar el mínimo. En la **figura 18** se muestran ejemplos de estos comportamientos.

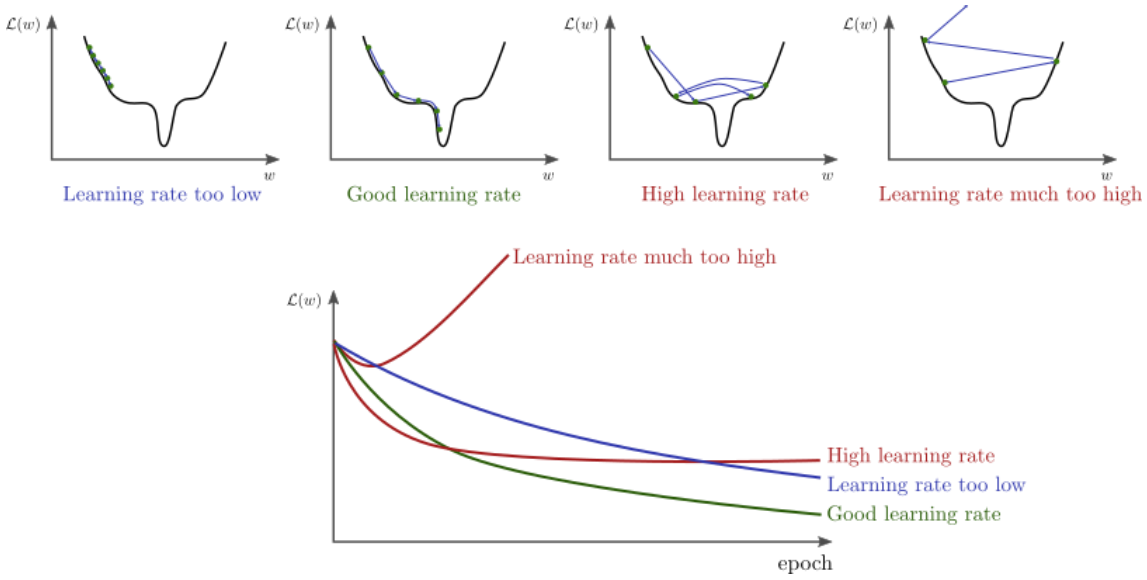


*Figura 18.* Posibles resultados del descenso del gradiente según la tasa de aprendizaje.

## Procedimiento

Para encontrar los valores $W$ y $b$, lo primero es inicializar los parámetros. En el caso de la regresión logística, casi cualquier método de inicialización funciona. En general, se pueden inicializar los valores a 0, aunque la inicialización aleatoria también es válida. Debido a que la función de coste elegida es convexa, realmente no importa dónde se inicialicen los valores, se termina llegando (aproximadamente) al mismo punto. Visualmente, mirando la **figura 16**, el gradiente lo que hace es, desde el punto inicial, "bajar hacia el mínimo en la dirección con más inclinación". Al fin y al cabo, el gradiente es una derivada que busca la máxima pendiente, la dirección del cambio máximo en la función en cada momento. Las fórmulas que aplican el descenso del gradiente son las siguientes:

$$W = W - \alpha \frac{\partial J(W,b)}{\partial W}$$

$$b = b - \alpha \frac{\partial J(W,b)}{\partial b}$$

*( 3 )*

$W$ indica cada uno de los diferentes pesos, agrupados en un vector o matriz. La derivada de la función de coste se puede pensar como la dirección que se toma hacia el mínimo buscado y, en conjunto con la tasa de aprendizaje, determina cómo de grande es cada paso hacia el mínimo. El algoritmo es sencillo de aplicar:

- Paso 1. Escogemos unos valores iniciales para $W$ y $b$.
- Paso 2. Calculamos los nuevos valores de $W$ y $b$ de acuerdo con las ecuaciones anteriores.
- Paso 3. Asignamos a $W$ y $b$ los nuevos valores calculados. Es decir, los actualizamos simultáneamente.
- Con los nuevos $W$ y $b$ repetimos los pasos 2 y 3 hasta la convergencia (hasta llegar a un mínimo o sus cercanías)

Dependiendo de si la función de coste es convexa o no, este mínimo puede ser absoluto o local. En nuestro caso, el mínimo es global. Por último, es muy importante detallar el paso 3 de actualización simultánea, para evitar confusiones. A continuación se muestra una implementación incorrecta en contraposición de la correcta:

*Implementación incorrecta*

$$temp\_W = W - \alpha \frac{\partial J(W,b)}{\partial W} \Big\} \rightarrow W = temp\_W \rightarrow temp\_b = b - \alpha \frac{\partial J(W,b)}{\partial b} \Big\} \rightarrow b = temp\_b \rightarrow Repetición\ del\ ciclo$$

*Implementación correcta*

$$\left.\begin{array}{l} temp\_W = W - \alpha \dfrac{\partial J(W,b)}{\partial W} \\ temp\_b = b - \alpha \dfrac{\partial J(W,b)}{\partial b} \end{array}\right\} \rightarrow \left.\begin{array}{l} W = temp\_W \\ b = temp\_b \end{array}\right\} \rightarrow \left.\begin{array}{l} temp\_W = W - \alpha \dfrac{\partial J(W,b)}{\partial W} \\ temp\_b = b - \alpha \dfrac{\partial J(W,b)}{\partial b} \end{array}\right\} \rightarrow Repetición\ del\ ciclo$$

Para simplificar la notación, las fórmulas podrán escribirse como sigue:

$$W = W - \alpha\, dW$$

$$b = b - \alpha\, db$$

*( 4 )*

Antes de proceder a entrenar la regresión logística con el descenso del gradiente, veamos los gráficos computacionales, herramienta que nos ayudará en todos los cálculos que involucran las redes neuronales: cálculo de la función de coste, propagación hacia adelante (*forward propagation*) y *dW* y *db* (la retropropagación o *backpropagation*).

## Gráficos computacionales

Los cálculos de una red neuronal se organizan en términos de un paso de propagación hacia delante, en el que se calcula la salida de la red neuronal, seguido de un paso de propagación hacia atrás, utilizado para calcular gradientes o derivadas. Los gráficos computacionales explican por qué están organizadas de esa manera, pues son un tipo de gráfico que se puede utilizar para representar expresiones matemáticas. Esto es similar al lenguaje descriptivo en el caso de los modelos de aprendizaje profundo, proporcionando una descripción funcional del cálculo requerido. En general, el gráfico computacional es un gráfico dirigido que se utiliza para expresar y evaluar expresiones matemáticas. Estos se pueden utilizar para dos tipos diferentes de cálculos: cálculo directo (propagación hacia delante) o cálculo regresivo (retropropagación). Algunos términos clave en gráficos computacionales son:

- Variables: están representadas por un nodo en un gráfico. Pueden ser un escalar, un vector, una matriz, un tensor o incluso otro tipo de variable.
- Los argumentos de función y la interdependencia de los datos están representados por una arista o flecha. Son similares a los punteros de nodo.
- Operación: Es una función (simple) de una o más variables. Hay un conjunto de operaciones que están permitidas. Las funciones que son más complejas se pueden representar combinando múltiples operaciones

Consideremos la siguiente función:

$$Y(a, b, c) = Y = (a + b) * (b - c)$$

Para una mejor comprensión, introduzcamos dos variables, *d* y *e*, tales que cada operación tenga una variable de salida. Reformulando:

$$d = a + b$$
$$e = b - c$$
$$Y = d * e$$

Tenemos tres operaciones, suma, resta y multiplicación. Para crear el gráfico computacional, creamos nodos, cada uno de los cuales tiene diferentes operaciones junto con variables de entrada. La dirección de la flecha indica la dirección en la que la entrada se aplica a otros nodos. Podemos encontrar el valor de salida final inicializando las variables de entrada y, en consecuencia, calculando los nodos del gráfico. Esto sería la propagación hacia delante. ¿Cómo es la retropropagación y el cálculo de las derivadas?

Si uno quiere entender las derivadas en un gráfico computacional, la clave es entender cómo un cambio en una variable genera un cambio en la variable que depende de ella. Si *a* afecta directamente a *c*, y hacemos un pequeño cambio en el valor de *a*, ¿cómo cambia *c*? Esto es la derivada parcial de *c* con respecto a *a*. Se debe seguir la **regla de la cadena** para evaluar las derivadas parciales de la variable de salida final con respecto a las variables de entrada: *a*, *b* y *c*. Por lo tanto, las derivadas parciales se pueden dar como:

$$\frac{\partial Y}{\partial a} = \frac{\partial Y}{\partial d} \times \frac{\partial d}{\partial a} = e \times 1 = e$$
$$\frac{\partial Y}{\partial b} = \frac{\partial Y}{\partial d} \times \frac{\partial d}{\partial b} = e \times 1 = e$$
$$\frac{\partial Y}{\partial c} = \frac{\partial Y}{\partial e} \times \frac{\partial e}{\partial c} = d \times -1 = -d$$

Esto nos da una idea de cómo los gráficos computacionales facilitan la obtención de derivadas mediante la retropropagación. La clave de este método es que, al calcular las derivadas, la forma más eficiente de hacerlo

es de derecha a izquierda (o de arriba abajo, como en la **figura 19**). El cálculo hacia adelante (*forward propagation*) o de izquierda a derecha sirve calcular la función de coste, $J(W, b)$, que se quiere optimizar. Y el cálculo hacia atrás (*backpropagation*) o de derecha a izquierda sirve para calcular las derivadas. Es importante conocer la regla de la cadena para entender estas derivaciones, aunque no imprescindible gracias a las herramientas de cálculo disponibles.

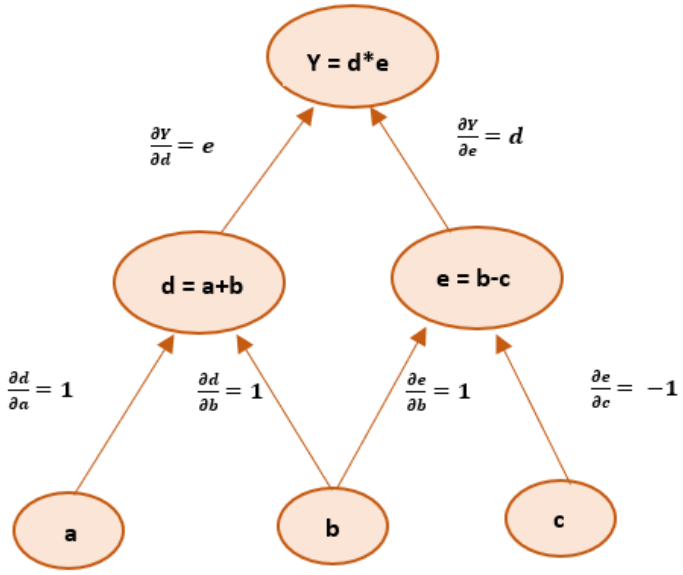


*Figura 19.* Gráfica computacional de la función de ejemplo *Y(a,b,c)*.

En el próximo apartado se aplicarán los conceptos de los gráficos computacionales en el contexto de la regresión logística, donde se mostrará exactamente lo que hay que hacer para calcular las derivadas del modelo de regresión y aplicar el algoritmo descenso del gradiente que permite aprender los parámetros del modelo.

## Descenso del gradiente aplicado a la regresión logística

En esta sección final se calcularán las derivadas necesarias para implementar el descenso del gradiente en la regresión logística, centrándonos en aquellas ecuaciones necesarias para implementarlo. Como herramienta se utilizará un gráfico computacional, pese a que para una función tan sencilla como es la regresión logística, que se puede interpretar como una red neuronal monocapa mononeurona, es innecesario. Sin embargo, esto servirá para posteriormente escalarlo y comprender mejor las redes neuronales completas. Para facilitar la explicación, imaginemos que la matriz de entrada, $X$, está formada por un único ejemplo de entrenamiento con dos características, $x_1$ y $x_2$:

$$\mathrm{X} = \mathrm{x} \in \mathbb{R}^2; \;\; \mathrm{x} = (x_1, x_2); \; W = (w_1, w_2); \;\; b$$

En ese caso, la predicción es equivalente a la salida de la regresión logística, que denominaremos "a" :

$$\hat{y} = \mathrm{a} = \sigma(z)$$

Es decir, por símil, “a” equivale a la activación de la única neurona del sistema. La función de coste quedaría reducida a la función de pérdida de un solo caso:

$$J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}, y) \xrightarrow{m=1} L(\hat{y}, \mathrm{y}) = L(\mathrm{a}, \mathrm{y})$$

El objetivo es reducir al máximo la pérdida modificando los parámetros. El orden de los pasos de propagación y cálculos necesarios se muestran en el siguiente gráfico computacional:

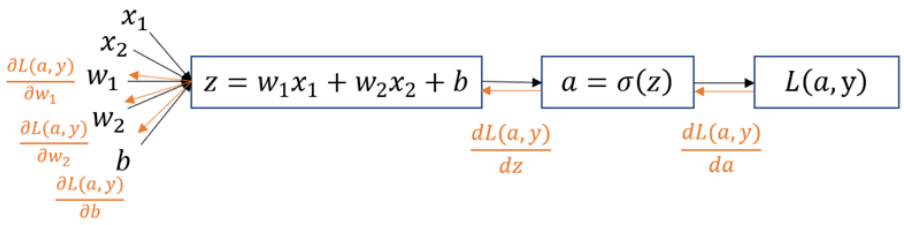


*Figura 20.* Gráfico computacional de la regresión logística, entendido como red neuronal, para un ejemplo de entrenamiento. Las flechas negras, que van de izquierda a derecha, indican la propagación hacia delante, mientras que las flechas naranjas indican la retropropagación.

Conocida la expresión $L(\mathrm{a},\mathrm{y})$, ecuación *(1)*, las derivadas serían, aplicando la regla de la cadena:

$$\frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$dz = \frac{\partial L(a,y)}{\partial z} = \frac{dL(a,y)}{da} \times \frac{\partial a}{\partial z} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) \times (a(1-a)) = a - y$$

$$dw_1 = \frac{\partial L(a,y)}{\partial w_1} = \frac{\partial L(a,y)}{\partial z} \times \frac{\partial z}{\partial w_1} = (a-y)x_1$$

$$dw_2 = \frac{\partial L(a,y)}{\partial w_2} = \frac{\partial L(a,y)}{\partial z} \times \frac{\partial z}{\partial w_2} = (a-y)x_2$$

$$db = \frac{\partial L(a,y)}{\partial b} = \frac{\partial L(a,y)}{\partial z} \times \frac{\partial z}{\partial b} = (a-y)$$

Sustituyendo en *(3)*, las fórmulas a aplicar hasta la convergencia del algoritmo serían:

$$w_1 = w_1 - \alpha \frac{\partial L(a,y)}{\partial w_1} = w_1 - \alpha(a-y)x_1$$

$$w_2 = w_2 - \alpha \frac{\partial L(a,y)}{\partial w_2} = w_2 - \alpha(a-y)x_2$$

$$b = b - \alpha \frac{\partial L(a,y)}{\partial b} = b - (a-y)$$

## Descenso del gradiente aplicado a la regresión logística para *m* casos de entrenamiento

Siguiendo con el ejemplo anterior, pero generalizándolo a un conjunto de entrenamiento, la matriz de entrada, $X$, estaría formada por $m$ ejemplos de entrenamiento con dos características cada uno, $x_1^{(i)}$ y $x_2^{(i)}$. La representación de esta red podría ser:

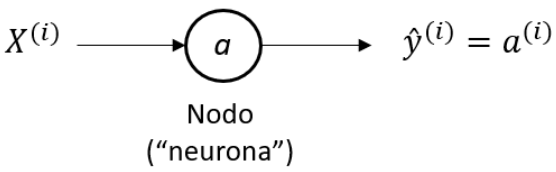


*Figura 21.* (pseudo) Red neuronal monocapa con una unidad. Posible representación conceptual de la regresión logística.

En ese caso, en la función de coste *(2)*, se puede sustituir $\hat{y}^{(i)}$ por $a^{(i)}$. La notación para este ejemplo sería:

$$X \in \mathbb{R}^{2 \times m};\ x^{(i)} = \left(x_1^{(i)}, x_2^{(i)}\right);\ W = (w_1, w_2)$$

$$\hat{y}^{(i)} = a^{(i)} = \sigma\left(z^{(i)}\right) = \sigma\left(W^T x^{(i)} + b\right) = \sigma\left(w_1 x_1^{(i)} + w_2 x_2^{(i)} + b\right)$$

Reformulando por comodidad:

$$\frac{\partial J(w,b)}{\partial w_1} = dw_1\ ;\ \frac{\partial J(w,b)}{\partial w_2} = dw_2\ ;\ \frac{\partial J(w,b)}{\partial b} = db$$

El objetivo sigue siendo minimizar este coste total. Aplicando el descenso del gradiente:

$$w_1 = w_1 - \alpha dw_1$$
$$w_2 = w_2 - \alpha dw_2$$
$$b = b - \alpha db$$

Las derivadas $dw_1$y $dw_2$ son distintas en el caso general dado que $J(W,b)$ es el sumatorio de la función de pérdida para cada ejemplo de entrenamiento. Explícitamente, en nuestro caso de dos características, el valor de las derivadas equivaldría en cada paso del descenso del gradiente a:

$$dw_1 = \frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(a^{(i)}, y^{(i)})}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_1^{(i)}$$

$$dw_2 = \frac{\partial J(w,b)}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(a^{(i)}, y^{(i)})}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_2^{(i)}$$

$$db = \frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(a^{(i)}, y^{(i)})}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

La forma más intuitiva de implementar el descenso del gradiente en Python sería con un bucle *for* donde, recorriendo todos los $m$ casos, calculásemos a la vez el paso de propagación hacia delante y el paso de retropropagación, sumando los resultados a unas variables definidas fuera del bucle y, una vez terminado este, dividir entre los $m$ ejemplos. De esta forma tendríamos $J, W$ y $b$ para cada iteración del algoritmo, el cual repetiríamos hasta la convergencia en el mínimo global. Sin embargo, esto es computacionalmente más costoso y una mejor aproximación es la **vectorización**.

La vectorización es, en pocas palabras, el conjunto de técnicas que permiten deshacerse de los bucles *for* explícitos en el código. Antes de la era del *Big Data* y el aprendizaje profundo, los conjuntos de entrenamiento no eran lo suficientemente grandes como para que la vectorización fuese imprescindible. Era interesante tener el código vectorizado porque tendía a agilizar la ejecución, pero no siempre. Actualmente se ha convertido en una necesidad clave. Consultar el **anexo 2** sobre la vectorización.

En el **archivo 1** se implementa todo lo que se ha discutido en este bloque, construyendo un modelo con el cual se harán predicciones sobre el conjunto de datos y se analizarán los resultados. Tras su consulta, continue con la lectura del trabajo.

En este bloque se ha hecho una introducción a los principios básicos de las redes neuronales, utilizando para ello un método de aprendizaje automático clásico, la regresión logística, que se puede interpretar como una pseudo red neuronal o red neuronal monocapa y mononeurona. La mayoría de lo ejemplificado con ella seguirá siendo válido y en los siguientes bloques se ampliarán y matizarán estos conceptos en redes neuronales reales. A continuación, se tratará con las primeras redes neuronales reales: las redes neuronales superficiales o poco profundas.

# 3

# Redes neuronales poco profundas

Tras explicar los conceptos generales de una red neuronal y los principales algoritmos y operaciones que se deben realizar, utilizando para ello un caso de aprendizaje automático clásico, este nuevo bloque tratará las redes neuronales poco profundas. Se comenzará explicando los componentes de una red neuronal de una capa oculta, también llamada de dos capas, según se prefiera. Estos componentes ya se mencionaron en el bloque introductorio en las **figuras 5** y **6** y se verán aquí con más en detalle, al igual que los cálculos y fórmulas necesarias en una red neuronal superficial. Se analizarán a su vez las funciones de activación, la retropropagación y la inicialización aleatoria. Tras estas explicaciones se dispondrá de las herramientas necesarias para comprender el **archivo 2** y sus resultados, donde también se menciona la teoría, pero de forma más resumida.

## Representación de redes neuronales con una capa oculta

Si las redes neuronales se componen de unidades (ocultas) ordenadas en capas, ya se ha visto con la regresión logística el caso más simple de red neuronal, monocapa con una única unidad, donde cada ejemplo de entrenamiento $x^{(i)}$ o $X^{(i)} \in \mathbb{R}^{n_x}$ está compuesto por el vector $\left\{x_1^{(i)}, \cdots, x_{n_x}^{(i)}\right\}$ de características. El siguiente paso en complejidad es pasar de una a dos capas, apareciendo la primera capa oculta, y pudiendo hablar de una red neuronal en sí, cuya representación genérica es la siguiente:

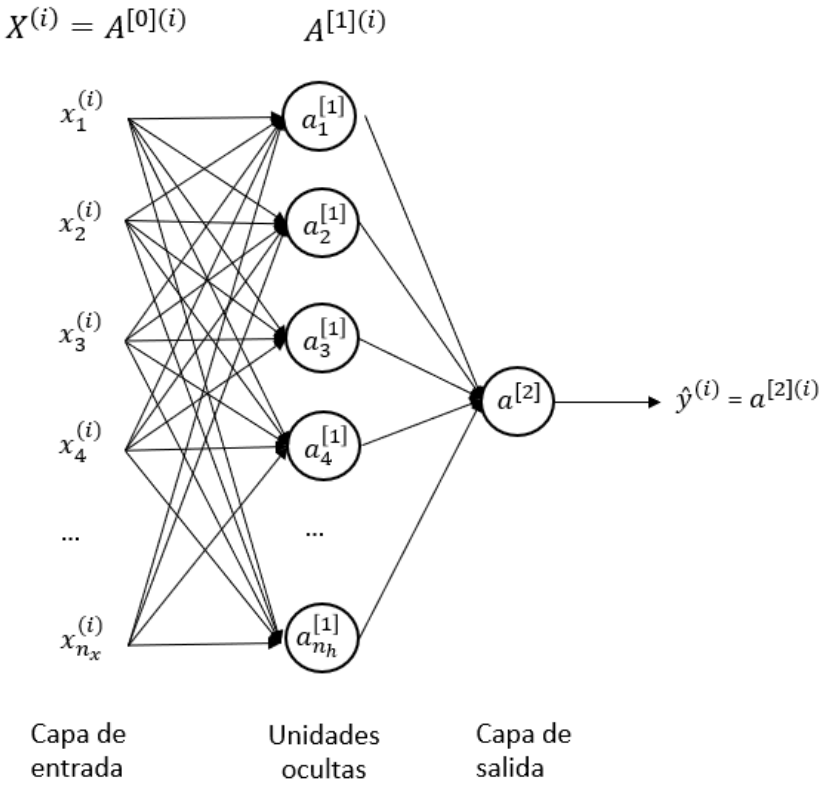


*Figura 22.* Representación genérica de una red neuronal con una capa oculta y una capa de salida con una sola neurona. Se han omitido los superíndices *(i)* en las funciones de activación por espacio en la figura. Los nodos representan entradas, activaciones o salidas, mientras que las aristas o líneas representan pesos y sesgos, ($w_{kj}^{[l]}, b_k^{[l]}$).

Una notación alternativa para denotar los valores de las características de entrada de cada ejemplo de entrenamiento, $x^{(i)}$, será $a^{[0](i)}$ o $A^{[0](i)}$ por lo que $n_x = n_h^{[0]}$. Recordemos que la letra "a" o "A" significa en redes neuronales activaciones (de la unidad oculta o capa) y se refiere a los valores que las diferentes capas de la red neuronal están pasando a las capas posteriores. Así que la capa de entrada transmite el vector $x^{(i)}$ o $a^{[0](i)}$ a la siguiente capa y esta, que es la capa oculta, generará a su vez un conjunto de activaciones, que denotaremos como $a^{[1](i)}$ o $A^{[1](i)}$. En concreto, el primer nodo generará el valor $a_1^{[1](i)}$, el segundo nodo generará el valor $a_2^{[1](i)}$ y así sucesivamente hasta $a_{n_h}^{[1](i)}$. Por lo tanto, siendo $n_h^{[1]}$ el número de neuronas en la capa $l = 1$, $a^{[1](i)}$ *o* $A^{[1](i)} \in \mathbb{R}^{n_h^{[1]}}$. Finalmente, la capa de salida genera un valor numérico $a^{[2](i)}$, el cual es equivalente a la predicción $\hat{y}^{(i)}$. Así que esto es análogo a cómo representamos la salida en la regresión logística, pero como en esta última sólo había una capa de salida, no se utilizaban los corchetes superíndice. En las redes neuronales se utiliza el corchete superíndice para indicar explícitamente de la capa que proviene el resultado. Cada capa oculta tendrá asociados los parámetros $W^{[l]}$ y $b^{[l]}$, los cuales tendrán las siguientes dimensiones:

$$W^{[l]} \in \mathbb{R}^{n_h^{[l]} \times n_h^{[l-1]}}; \quad b^{[l]} \in \mathbb{R}^{n_h^{[l]}}$$

Las capas intermedias se llaman capas ocultas. Y cada elemento de estas capas es una unidad oculta o una "neurona" artificial. Esta terminología viene del entrenamiento de un conjunto de datos. En un conjunto de entrenamiento, se conocen los valores de entrada, $x^{(i)}$, y los valores de salida correctos, $y^{(i)}$. Pero los valores intermedios en las neuronas están ocultos, y de ahí el nombre.

He mencionado que a la red de la **figura 22** se le puede denominar red neuronal con una capa oculta o red neuronal de dos capas. ¿Y qué sucede con la capa de entrada? La razón es que cuando se cuentan las capas en las redes neuronales, no se cuenta la capa de entrada. Por lo tanto, la capa oculta es la capa uno y la capa de salida es la capa dos. En la notación de este trabajo se llama a la capa de entrada capa cero, así que técnicamente habría tres capas en esta red neuronal. Pero en el uso convencional y en artículos de investigación, esta red neuronal en particular se denomina red neuronal de dos capas al no contar la capa de entrada como una capa oficial. En el siguiente apartado se mostrará qué está calculando esta red neuronal, desde que entran los ejemplos de entrenamiento, hasta la salida y predicción de la red. Es decir, se explicará cómo se produce la propagación hacia delante (*forward propagation*).

## Cálculo de la salida de una red neuronal con una capa oculta

¿Qué calcula una unidad oculta o "neurona" artificial? Calcula una función lineal $Wx + b$, seguido de una función de activación. Esto es muy similar a lo que se vio en la regresión logística. La diferencia es que en la regresión logística esta función de activación era la función sigmoide, representada por la letra griega sigma, σ. En una red neuronal, la función de activación puede ser una **función no lineal** cualquiera, que representaremos con $g(\mathrm{z})$, y su resultado será la activación (salida) de dicha neurona. Por lo tanto, cada unidad oculta efectuará el siguiente cálculo:

$$z = Wx + b$$
$$a = g(\mathrm{z})$$

denotando "a" la activación (salida) de la unidad oculta. Siendo rigurosos en la notación:

$$z_k^{[l](i)} = W_j^{[l]T} x^{(i)} + b_k^{[l]}$$
$$a_k^{[l](i)} = g^{[l]}\left(z_k^{[l](i)}\right), 1 \leq l \leq L$$

con $W_j^{[l]} \in \mathbb{R}^{n_h^{[l-1]}}$, y $b_k^{[l]}, z_k^{[l]}, a_k^{[l](i)} \in \mathbb{R}$. $g^{[l]}$ es la función de activación no lineal para la capa $l$. El superíndice *T* indica la traspuesta del vector o matriz. Para más dudas sobre los índices, acudir al anexo de notación. A

continuación, se muestra el cálculo completo para un único ejemplo de entrenamiento con cuatro características de una red neuronal formada por una capa oculta con tres unidades ocultas. De esta manera evitamos poner a las variables el superíndice $(i)$, se sobreentenderá el superíndice $(1)$. Se supondrá que las funciones de activación de la capa de salida devuelven un único valor por ejemplo de entrenamiento para facilitar el cálculo dimensional. Su representación sería la siguiente:

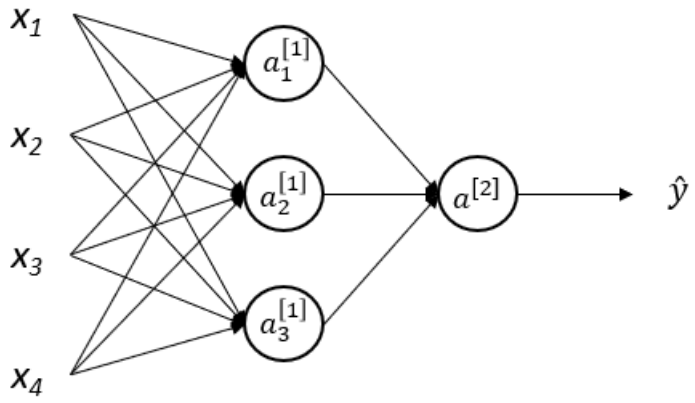


*Figura 23.* Ejemplo de red neuronal de dos capas.

Siendo el vector de entrada un vector columna $x^{(1)} = x = \{x_1, x_2, x_3, x_4\}$, los cálculos en la capa oculta serían:

$$z_1^{[1]} = W_1^{[1]T} x + b_1^{[1]}, \qquad a_1^{[1]} = g^{[1]}\left(z_1^{[1]}\right)$$
$$z_2^{[1]} = W_2^{[1]T} x + b_2^{[1]}, \qquad a_2^{[1]} = g^{[1]}\left(z_2^{[1]}\right)$$
$$z_3^{[1]} = W_3^{[1]T} x + b_3^{[1]}, \qquad a_3^{[1]} = g^{[1]}\left(z_3^{[1]}\right)$$

En la práctica, estás ecuaciones se podrían implementarían con un bucle *for*, lo cual ya se ha discutido que es ineficiente. La versión vectorizada de estos cálculos es:

$$z^{[1]} = W^{[1]}\mathrm{x} + b^{[1]} = \begin{bmatrix} \cdots W_1^{[1]T} \cdots \\ \cdots W_2^{[1]T} \cdots \\ \cdots W_3^{[1]T} \cdots \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix} = \begin{bmatrix} W_1^{[1]T} x + b_1^{[1]} \\ W_2^{[1]T} x + b_2^{[1]} \\ W_3^{[1]T} x + b_3^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix}$$

Dimensionalmente:

$$(3 \times 1) = (3 \times 4)(4 \times 1) + (3 \times 1)$$

La activación de la capa oculta es:

$$a^{[1]} = g^{[1]}\left(z^{[1]}\right) = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix}$$

siendo $a^{[1]}$ un vector $(3 \times 1)$. Los cálculos en la capa de salida serían:

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

Dimensionalmente:

$$(1 \times 1) = (1 \times 3)(3 \times 1) + (1 \times 1)$$

La activación de la capa de salida, que equivale a la predicción para el ejemplo de entrenamiento es:

$$\hat{y} = a^{[2]} = g^{[2]}\left(z^{[2]}\right)$$

En total, cuatro ecuaciones vectorizadas permiten el cálculo de la propagación hacia delante, cuyo resultado es la predicción de un único ejemplo de entrenamiento. A continuación, veamos cómo se hace este cálculo vectorizado para $m$ casos de entrenamiento, usando la red neuronal ejemplo de la **figura 23**. Si cada ejemplo de entrenamiento lo representamos como un vector columna, se cogen todos ellos y se apilan, tendríamos una matriz de entrenamiento $X \in \mathbb{R}^{n_x \times m}$. Se han utilizado minúsculas para la función lineal y para la activación de las unidades ocultas para diferenciar con el caso general con múltiples ejemplos de entrenamiento que se muestra a continuación, donde se utilizaran las mayúsculas. La implementación de las ecuaciones para $m$ ejemplos consiste en calcular las cuatro ecuaciones vistas para un ejemplo de entrenamiento $m$ veces. Si lo representamos como un bucle, se haría de la siguiente manera:

*for i = 1, … , m:*

$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$

$a^{[1](i)} = g^{[1]}\left(z^{[1](i)}\right)$

$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$

$\hat{y}^{(i)} = a^{[2](i)} = g^{[2]}\left(z^{[2](i)}\right)$

Esto no es eficiente, sobre todo cuanto mayor sea la matriz de entrada, por lo que se busca siempre la vectorización de los cálculos. Si cada resultado anterior para cada unidad oculta se almacena en matrices, tendríamos unas ecuaciones vectorizadas inmediatamente, las cuales dan los mismos resultados que ese bucle debido a las propiedades de las matrices. Consultar anexos en caso de dudas sobre matrices. El cálculo vectorizado sería el siguiente:

$$Z^{[1]} = W^{[1]}\mathrm{X} + b^{[1]} = \begin{bmatrix} \cdots W_1^{[1]T} \cdots \\ \cdots W_2^{[1]T} \cdots \\ \cdots W_3^{[1]T} \cdots \end{bmatrix} \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix} + \begin{bmatrix} \vdots & \vdots & & \vdots \\ b^{[1]} & b^{[1]} & \cdots & b^{[1]} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

$$= \begin{bmatrix} \vdots & \vdots & & \vdots \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

Dimensionalmente:

$$(3 \times m) = (3 \times 4)(4 \times m) + (3 \times m)$$

donde el vector columna de sesgos se expande por *broadcasting* de $(3 \times 1)$ a $(3 \times m)$. Consultar anexos. La activación de la capa oculta es:

$$A^{[1]} = g^{[1]}\left(Z^{[1]}\right) = \begin{bmatrix} \vdots & \vdots & & \vdots \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

siendo $A^{[1]}$ un vector $(3 \times m)$. Los cálculos en la capa de salida son:

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

Dimensionalmente:

$$(1 \times m) = (1 \times 3)(3 \times m) + (1 \times m)$$

donde, de nuevo, el vector columna de sesgos se expande por *broadcasting* de $(1 \times 1)$ a $(1 \times m)$. La activación de la capa de salida, que equivale a la predicción para todo el conjunto de entrenamiento es:

$$\hat{y} = A^{[2]} = g^{[2]}\left(Z^{[2]}\right)$$

Las dimensiones de la matriz de predicción serán:

$$\hat{y} \text{ o } \hat{Y} \in \mathbb{R}^{n_y \times m}$$

$n_y$ es el número de etiquetas o clases que se están prediciendo por cada ejemplo de entrenamiento. Respecto a los ejemplos prácticos que se ven en este trabajo, se trata de problemas de clasificación binaria donde se predice a qué categoría (solo una) pertenece cada ejemplo de entrenamiento. Por lo tanto en este tipo de problemas, $n_y = 1$.

A continuación, se discutirán distintas funciones de activación no lineales que se pueden utilizar, $g(\mathrm{z})$, además de la sigmoide, ya utilizada en el **archivo 1.**

## Funciones de activación

Al construir una red neuronal, una de las decisiones que hay que tomar es qué función de activación se usará en cada capa oculta y en la capa de salida. Tanto en las redes neuronales artificiales como biológicas, una neurona no sólo transmite la entrada que recibe. Existe un paso adicional, una función de activación, que es análoga a la tasa de potencial de acción disparado en el cerebro. La función de activación utiliza nuestra ya definida función de activación lineal, $z = Wx + b$ , y la transforma una vez más como salida. Se han propuesto muchas funciones de activación, de las cuales hasta ahora solo se ha mencionado y utilizado la función sigmoide. En el **anexo 1** se muestran en Python distintos tipos de funciones de activación.

Las funciones de activación no lineales se usan para propagar la salida de los nodos de una capa hacia la siguiente capa en una red neuronal. Estas funciones permiten incorporar el modelado de datos de entrada no lineales a la red. Otros ejemplos de funciones de activación no lineales son la tangente hiperbólica (*tanh*) y la función *ReLU* (unidad lineal rectificada) y sus derivaciones. Veamos estas funciones que podremos utilizar como activación en cada capa, $a = g(\mathrm{z})$.

### *Tangente hiperbólica*

La tangente hiperbólica de un número real, $z$, se designa mediante $tanh(z)$ y se define como el cociente entre el seno y el coseno hiperbólicos del número real $z$. Si se sustituye de acuerdo con las definiciones de seno y coseno hiperbólicos, se obtiene una fórmula más directa para la tangente hiperbólica.

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Su rango de valores es $tanh(z) \in (-1,1)$. Gráficamente:

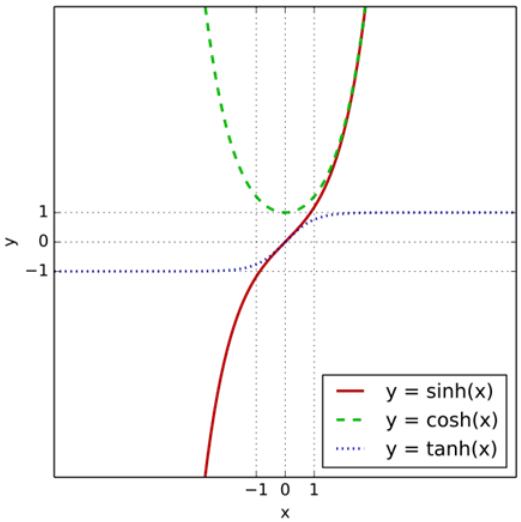


*Figura 24.* Representación de las funciones *sinh(x)*, *cosh(x)* y *tanh(x)*.

Su derivada es:

$$tanh'(z) = 1 - tanh^2(z)$$

Vemos que es una versión desplazada de la función sigmoidea. Casi siempre funciona mejor que la función sigmoidea porque con valores entre +1 y -1, la media de las activaciones que salen de la capa oculta está más cerca de ser cero. Esto implica que, cuando se entrena un algoritmo de aprendizaje, puede centrar los datos y hacer que tengan media cercana a cero utilizando la función *tanh* en lugar de la función sigmoide. Y esto permite que el aprendizaje para la siguiente capa sea un poco más fácil. La función sigmoide sigue siendo utilizada en la capa de salida, sobre todo en problemas de clasificación binaria o aquellos problemas donde las etiquetas están entre cero y uno.

Una de las desventajas tanto de la función sigmoidea como de la función *tanh* es que si $z$ es muy grande o pequeña, entonces el gradiente de la derivada de la pendiente de esta función se vuelve muy pequeño y esto puede ralentizar el descenso del gradiente. Es el ya mencionado problema de desvanecimiento o explosión del gradiente**.** Este problema nos lleva a otra función muy popular que lo soluciona.

### *Unidades lineales rectificadas*

ReLU (*Rectified Lineal Units)*, que significa unidad lineal rectificada. Se define simplemente como $R(z) = \max(0, z)$, donde $z$ es la entrada a la "neurona" (unidad oculta). El valor de esta función va de infinito a cero de forma lineal y luego sigue como línea recta de valor constante cero para todo valor de entrada negativo. En otras palabras, las *ReLUs* permiten el paso de todos los valores positivos sin cambiarlos, pero asigna todos los valores negativos a 0.

$$R(z) = \max(0, z) = \frac{z + |z|}{2} = \begin{cases} z \; si \; z > 0 \\ 0 \; en \; caso \; contrario \end{cases}$$

Su derivada:

$$R'(z) = \begin{cases} 1 \; si \; z > 0 \\ 0 \; si \; z < 0 \\ indeterminado \; si \; z = 0 \end{cases}$$

Su rango de valores es $R(z) \in [0, \infty)$. Gráficamente:

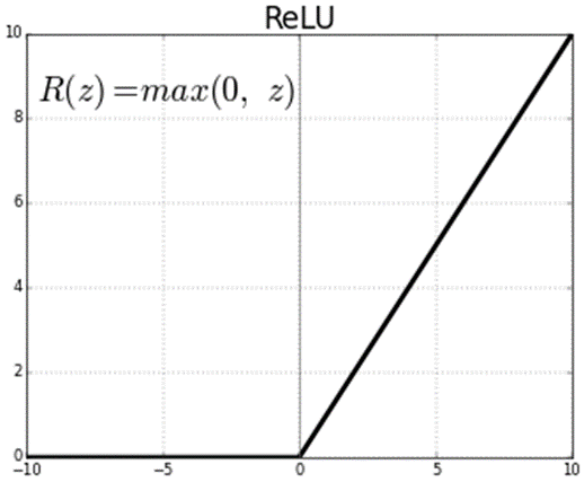


*Figura 25.* Representación de la función ReLU

Vemos que existe la (peligrosa) posibilidad de que el valor de la derivada sea indeterminado si $z$ es exactamente cero, lo cual al implementarlo computacionalmente daría problemas. Sin embargo, la posibilidad real de que $z$ sea cero en el código es improbable y hay técnicas de optimización que convierten

esta posibilidad en un subgradiente que permite que incluso en ese caso el descenso del gradiente (o el algoritmo de optimización utilizado) siga funcionando. Otra solución en la implementación es forzar que la derivada equivalga a cero o a uno si $z$ es cero.

$$R'(z) = \begin{cases} 1 \; si \; z \geq 0 \\ 0 \; si \; z < 0 \end{cases} \quad || \quad R'(z) = \begin{cases} 1 \; si \; z > 0 \\ 0 \; si \; z \leq 0 \end{cases}$$

Las unidades lineales rectificadas encuentran aplicaciones en visión por ordenador y reconocimiento del habla utilizando redes neuronales profundas y neurociencia computacional. Hay variantes de la función ReLU, de las cuales veremos una a continuación.

*Leaky ReLU*

La función *leaky* ReLU o LReLU permite un pequeño gradiente positivo cuando la entrada, $z$, no está activa, es decir, es menor o igual que cero.

$$LR(z) = \max(0.01 * z, z) = \begin{cases} z \; si \; z > 0 \\ 0.01 * z \; en \; caso \; contrario \end{cases}$$

Su derivada:

$$LR'(z) = \begin{cases} 1 \; si \; z > 0 \\ 0.01 \; si \; z < 0 \\ indeterminado \; si \; z = 0 \end{cases}$$

o para evitar problemas de implementación:

$$LR'(z) = \begin{cases} 1 \; si \; z \geq 0 \\ 0.01 \; si \; z < 0 \end{cases}$$

Su rango de valores es $LR(z) \in (-\infty, \infty)$. Gráficamente:

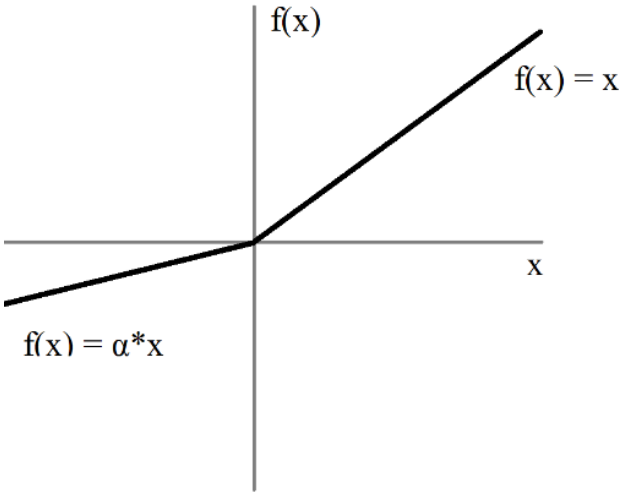


*Figura 26.* Representación de la función *Leaky* ReLU

Una de las principales ideas de por qué tienden a dar mejores resultados es que la activación rectificada da lugar a representaciones dispersas, lo que significa que no muchas neuronas tienen que emitir valores distintos de cero para una entrada determinada. Se ha demostrado que la dispersión es beneficiosa para el aprendizaje profundo, tanto porque representa la información de una manera más robusta como porque conduce a una eficiencia computacional significativa (si un gran número de las neuronas emiten cero, se puede ignorar a la mayoría de ellas y realizar los cálculos mucho más rápido).

Hay otras variantes, como ReLU paramétrica, que es el caso general del que deriva LReLU; GELU (Unidad Lineal de Error Gaussiano, *Gaussian-error linear unit*), etc. Aunque existen funciones de activación aún más recientes, la mayoría de las redes neuronales utilizan ReLU o una de sus variantes.

## ¿Por qué ReLU desplazó a la función sigmoide como preferente para la activación?

Las funciones sigmoides fueron la base de la mayoría de las redes neuronales por muchas décadas, aunque en los últimos años han perdido popularidad. El motivo principal es que las redes neuronales de muchas capas se vuelven muy difíciles de entrenar dado el problema de desvanecimiento del gradiente. En su lugar, la mayoría de las redes neuronales actuales usan la función ReLU o alguna de sus variantes, que solucionan el problema del desvanecimiento del gradiente, es decir, unidades ocultas o regiones donde el gradiente es casi cero y el aprendizaje se vuelve muy lento. Pensemos por ejemplo en el algoritmo descenso del gradiente, que se basa en actualizar los parámetros de acuerdo con sus derivadas. Si estas son pequeñas o cercanas a cero, los parámetros cambian muy lentamente y no convergerán o lo harán muy lentamente a sus valores óptimos, consumiendo recursos y tiempo.

Como se mencionó en el bloque 1, al cambiar la función de activación de la red neuronal para utilizar la función ReLU, el gradiente es igual a 1 para todos los valores positivos de entrada y 0 para los valores negativos. Y aunque la pendiente de los valores negativos es 0 en esta función, resulta que es mucho menos probable que el gradiente se reduzca gradualmente hasta 0 y el simple hecho de cambiar de la función sigmoidea a la función ReLU hace que el algoritmo descenso del gradiente funcione mucho más rápido. Hay muchos ejemplos como este en donde cambiamos el algoritmo para que el código se ejecute mucho más rápido y esto permite entrenar redes neuronales más grandes.

La función $tanh$ también adolece del problema de desaparición del gradiente, pero a diferencia de la función sigmoide, al estar centrada en cero, tiende a funcionar mejor. Podemos ver en la siguiente imagen un porcentaje de uso aproximado, es decir, la popularidad de las diferentes funciones de activación:

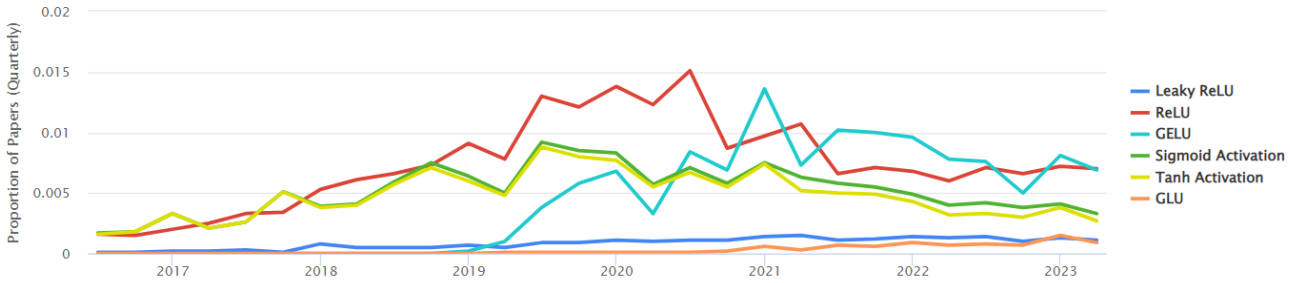


*Figura 27.* Popularidad de las diferentes funciones de activación. ReLU y alguna de sus variantes son las más utilizadas, pero *tanh* y sigmoide siguen teniendo cierto uso.

## Consejos prácticos sobre la elección de la función de activación

Si las etiquetas de salida son cero, uno, o valores entre medias, o si se está en un problema de clasificación binaria, la función de activación sigmoide es una elección natural para la capa de salida. También se utiliza la función sigmoide en la capa de salida en un problema de clasificación multiclase, donde la salida son múltiples etiquetas. Sin embargo, si la clasificación es multiclase con salida una única etiqueta, se utiliza en la salida la función de activación *softmax*. Se hace un análisis de ella en el **anexo 1**.

Para el resto de capas ocultas, se puede utilizar *tanh* o ReLU. Esta última se ha convertido cada vez más en la elección por defecto de la función de activación, aunque la primera sigue siendo utilizada. La principal desventaja de ReLU, que la derivada sea igual a cero cuando $z$ es negativo, en la práctica esto no suele dar problemas. Hay otras versiones de ReLU que resuelven algunas de las desventajas, como LReLU. Sin embargo, y como hemos visto en la gráfica de uso de las funciones de activación, la mayoría de estas variantes apenas son utilizadas, con excepción de GeLU. Cada una tiene un nicho concreto de problemas para

los que pueden ser útiles. En cualquier caso, dado que es muy difícil saber de antemano qué función de activación es mejor, una regla general muy sencilla sería probar todas y quedarse con la que funcione mejor según los criterios que tenga el proyecto: métrica(s) a optimizar, condiciones suficientes en otras métricas, etc. En caso de falta de tiempo o recursos, interviene la experiencia para interpretar qué funciones de activación puedan dar lugar a mejores resultados.

| Problem Type | Output Type | Final Activation Function | Loss Function |
|---|---|---|---|
| Regression | Numerical value | Linear | Mean Squared Error (MSE) |
| Classification | Binary outcome | Sigmoid | Binary Cross Entropy |
| Classification | Single label, multiple classes | Softmax | Cross Entropy |
| Classification | Multiple labels, multiple classes | Sigmoid | Binary Cross Entropy |

*Figura 28.* Funciones de activación y de coste para la capa de salida en los problemas más comunes.

## ¿Por qué son necesarias funciones de activación no lineales?

Las funciones de activación no lineales son necesarias en las redes neuronales porque su objetivo es producir un límite de decisión no lineal a través de combinaciones no lineales de los pesos y las entradas. Realizan la transformación no lineal de la entrada, permitiendo a la red neuronal aprender y realizar tareas más complejas al introducir no linealidad en la salida de una neurona. Si solo se usaran funciones de activación lineales, la red neuronal sería esencialmente un modelo de regresión lineal y no podría aprender relaciones no lineales entre la entrada y la salida. Veamos con un ejemplo por qué las funciones lineales no permiten aprender a la red neuronal. Supongamos que tenemos un único ejemplo de entrenamiento, como se vio en el cálculo de la salida de una red neuronal.

$$x^{(1)} = x = \{x_1, x_2, x_3, x_4\}$$

Las cuatro ecuaciones que calculan la propagación hacia delante, independientemente del número de unidades ocultas, son:

$$z^{[1]} = W^{[1]}\mathrm{x} + b^{[1]}$$
$$a^{[1]} = g^{[1]}\left(z^{[1]}\right)$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$\hat{y} = a^{[2]} = g^{[2]}\left(z^{[2]}\right)$$

donde $g^{[l]}$ es la función de activación no lineal de la capa l-ésima. ¿Qué sucedería si la propagación hacia delante consistiese solo en propagaciones lineales, en vez de no lineales? Las ecuaciones de activación quedarían de esta manera:

$$a^{[1]} = z^{[1]}$$
$$\hat{y} = a^{[2]} = z^{[2]}$$

Desarrollando la fórmula de la predicción:

$$\hat{y} = a^{[2]} = z^{[2]} = W^{[2]}\left(W^{[1]}\mathrm{x} + b^{[1]}\right) + b^{[2]} = \left(W^{[2]}W^{[1]}\right)x + \left(W^{[2]}b^{[1]} + b^{[2]}\right) = W'x + b'$$

El modelo estaría calculando las predicciones como una función lineal de sus características de entrada, $x$. Es decir, estaríamos calculando una regresión lineal, sin importar cuántas capas y neuronas tenga la red neuronal. En ese caso, ¿para qué molestarse en tener múltiples capas y unidades ocultas?

Otro ejemplo, el cual no se va a demostrar matemáticamente, pero que se puede encontrar en la literatura, es el siguiente: si en la capa oculta la función de activación fuese lineal, y en la capa de salida fuese la función sigmoide, este modelo sería prácticamente idéntico a una regresión logística, es decir, al modelo desarrollado en el **archivo 1**. La conclusión es que una capa oculta con función de activación lineal es prácticamente inútil, porque la composición de dos funciones lineales es en sí otra función lineal, y que el verdadero cálculo útil lo realizan funciones no lineales. Hay algún caso excepcional donde podría ser útil una función de activación lineal y es en un problema de regresión, donde las etiquetas de los resultados sean números reales. Y aun así, las capas ocultas no deberían utilizar la función de activación lineal y solo usarse en la capa de salida.

En los siguientes apartados, se explicará el concepto de retropropagación en redes neuronales y la importancia de la inicialización, la cual en la regresión logística se despachó inicializando los parámetros cero.

## Retropropagación

Introduzcamos el importante concepto de retropropagación en redes neuronales oficialmente, puesto que se han realizado unos cálculos derivativos para optimizar un algoritmo, pero quizá hace falta alguna explicación de por qué es necesaria y tan importante. De hecho la retropropagación (*backpropagation*) fue la razón principal por la que las redes neuronales volvieron a la popularidad en los 90s tras el estancamiento desde los años 70s. Posteriormente las grandes mejoras tuvieron que ver con la potencia de CPUs y GPUs.

¿Por qué es necesaria la retropropagación? En un inicio, las redes neuronales solo calculaban la propagación hacia delante. Básicamente, dados unos pesos iniciales, generalmente pequeños, se hacían los cálculos de la propagación hacia delante y, si el resultado, la predicción, no coincidía con la etiqueta del ejemplo de entrenamiento, se ajustaban los pesos y se volvía repetir este ciclo una y otra vez. Esto aplicaba tanto con las funciones de activación lineales que se utilizaron al principio del desarrollo de las redes neuronales como las no lineales que comenzaron a usarse más adelante. Las “neuronas” o unidades ocultas eran percibidas en general como “Perceptrones”.

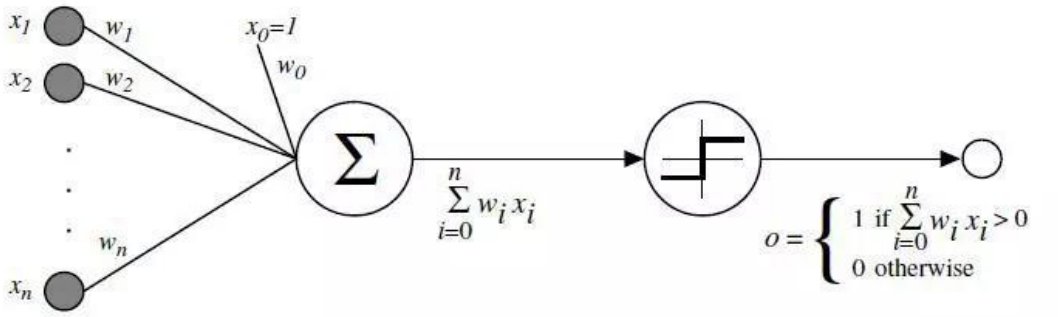


*Figura 29.* Diagrama que muestra el funcionamiento de un Perceptrón de Rosenblatt.

La razón por la que este método no funcionaba para redes de múltiples capas es bastante intuitiva: si cada ejemplo de entrenamiento sólo especifica la salida o etiqueta correcta para la capa de salida final, ¿cómo se podía saber cómo ajustar los pesos de las capas anteriores? No se podía, por supuesto. La respuesta, a pesar de que llevó algún tiempo deducirla, demostró estar basada en una antigua herramienta de cálculo: la **regla de la cadena**. La clave estaba en que si las neuronas de la red neuronal no eran exactamente perceptrones, sino que calculaban la salida con una función de activación no lineal pero también diferenciable, no sólo se podía utilizar la derivada para ajustar el peso y minimizar el error, sino que la regla de la cadena también podía utilizarse para calcular la derivada de todas las neuronas de una capa anterior y, por tanto, también se conocería la forma de ajustar sus pesos. En palabras más intuitivas, se podía utilizar el cálculo derivativo para “asignar parte de la culpa” de cualquier error del conjunto de entrenamiento en la capa de salida a cada neurona de la capa oculta anterior. Y esta “culpa” de los errores se podía extender hacia atrás a cada capa

oculta, empezando siempre por la de salida. De esta forma se retropropaga el error y de ahí el nombre del término.

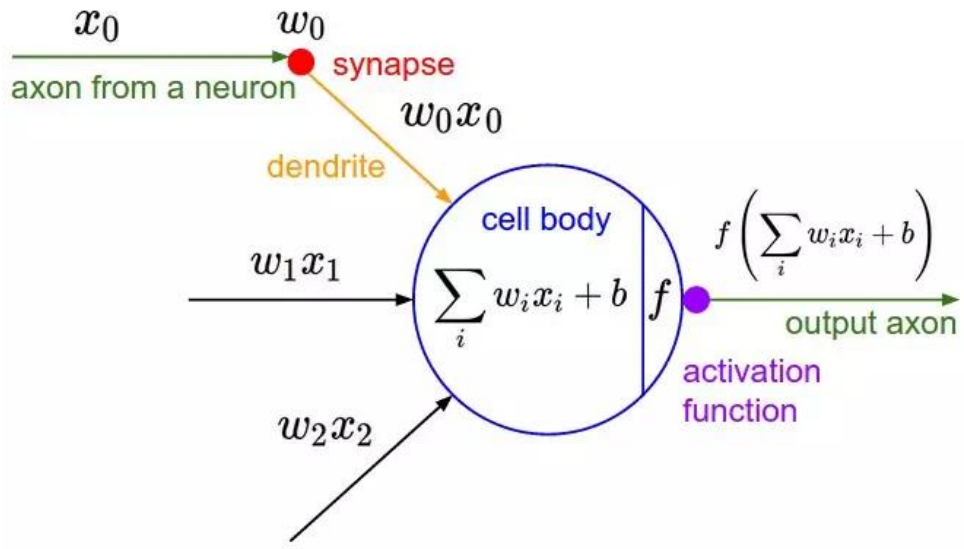


*Figura 30.* Otro diagrama que muestra la inspiración biológica. La función de activación (*activation function*) es lo que hoy se llama función no lineal que se aplica a la suma ponderada de entrada para producir la salida de la neurona artificial; en el caso del perceptrón de Rosenblatt, la función no es más que una operación de umbralización.

Con este procedimiento se podía averiguar cuánto cambiaba el error si se variaba cualquier peso de la red neuronal, incluidos los de las capas ocultas, y utilizar una técnica de optimización (durante mucho tiempo, inicialmente el descenso del gradiente) para encontrar los pesos óptimos que minimizasen el error. Esto resolvía el problema de cómo entrenar redes neuronales con múltiples capas. A partir de ese momento, el principal problema era cómo hacer esa montaña de cálculos, cuya respuesta estuvo, afortunadamente, en la potencia de cálculo de los ordenadores.

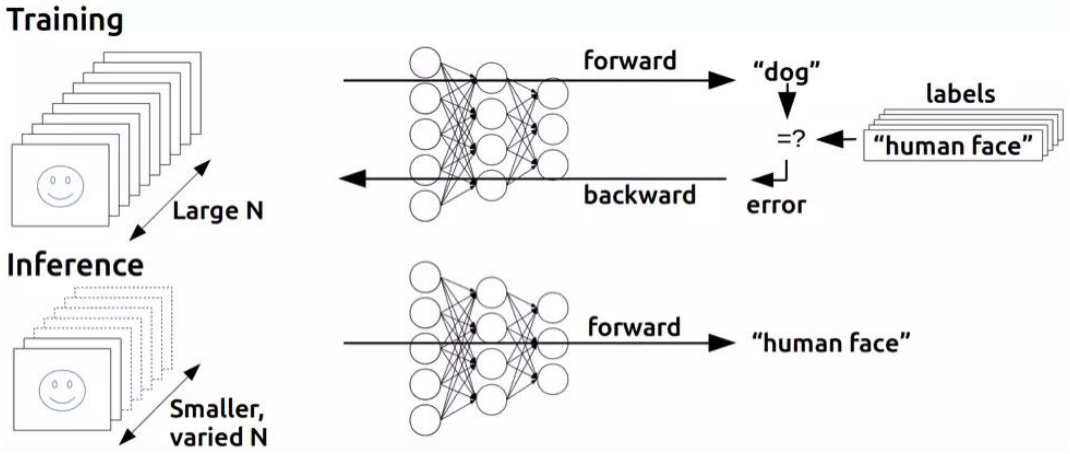


*Figura 31.* Esquema de la idea básica de la retropropagación (*backward propagation*).

Continuemos con la inicialización de los parámetros de la red, cuya importancia es capital.

## Inicialización aleatoria

Cuando se entrena una red neuronal, es importante inicializar los pesos, $W^{[l]}$, aleatoriamente. Para la regresión logística, es válido inicializar los pesos a cero, pero para una red neuronal inicializar los pesos a cero

y a continuación aplicar el descenso del gradiente, no funcionaría. Veamos con un sencillo ejemplo por qué. A continuación tenemos una red de dos capas muy pequeña, con solo dos características de entrada por ejemplo de entrenamiento y una capa oculta de dos unidades.

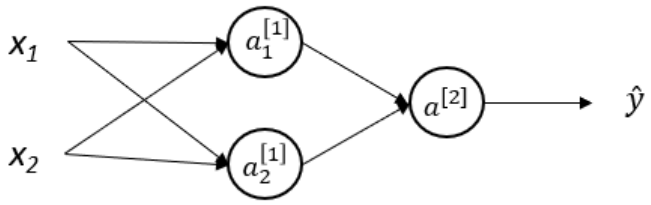


*Figura 32.* Ejemplo de red neuronal de dos capas con dos características de entrada.

Por lo tanto, $n_x = n_h^{[0]} = 2$ y $n_h^{[1]} = 2$. Partamos de aquello que se quiere justificar que no funciona, es decir, inicialicemos la matriz de pesos asociada a la capa oculta, $W^{[1]}$, a cero:

$$W^{[1]} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

e inicialicemos el vector de sesgos también a cero:

$$b^{[1]} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

El vector de sesgos si se puede inicializar como ceros sin causar ningún problema de aprendizaje a la red. El problema de esta inicialización de pesos es que, sin importar cual sea el ejemplo de entrenamiento de entrada, los valores de activación serán iguales en la propagación hacia delante:

$$z^{[1]} = W^{[1]}\mathrm{x} + b^{[1]} = \begin{bmatrix} \cdots W_1^{[1]T} \cdots \\ \cdots W_2^{[1]T} \cdots \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \end{bmatrix} = \begin{bmatrix} W_1^{[1]T} x + b_1^{[1]} \\ W_2^{[1]T} x + b_2^{[1]} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \end{bmatrix}$$

$$a^{[1]} = g^{[1]}\left(z^{[1]}\right) = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \end{bmatrix}$$

Al ser los valores de $z^{[1]}$ iguales (y además nulos) implica que, sin importar qué función de activación sea:

$$a_1^{[1]} = a_2^{[1]}$$

Se puede demostrar para la retropropagación que se cumple a su vez que:

$$dz_1^{[1]} = dz_2^{[1]}$$

¿Qué significa esto? Que ambas unidades ocultas son completamente idénticas, es decir, simétricas en el cálculo de las funciones. Por inducción, después de cada iteración de entrenamiento, se puede demostrar que las unidades ocultas seguirán calculando exactamente lo mismo. Por lo tanto, como:

$$dW^{[1]} = dz^{[1]} x^T$$

tendremos que $dW_1^{[1]T} = dW_2^{[1]T}$. Por lo que cada vez que se actualicen los pesos con el gradiente, la primera fila y la segunda serán exactamente las mismas.

Es posible construir una prueba por inducción donde, si se inicializan todos los pesos a cero, como ambas unidades ocultas empiezan calculando la misma función y ambas tienen la misma influencia en la unidad de salida, después de una iteración, esa misma afirmación sigue siendo cierta, las dos unidades ocultas siguen siendo simétricas. Y por lo tanto, por inducción, después de dos iteraciones, tres iteraciones y

así sucesivamente, no importa cuánto tiempo se entrene la red neuronal, ambas unidades ocultas siguen calculando exactamente la misma función. Por ello, no tendría sentido tener más de una unidad oculta, puesto que ambas están calculando lo mismo. Este razonamiento funciona sin importar el número de unidades ocultas en la red neuronal. Está claro que sería un resultado no deseado, puesto que la idea es que cada unidad oculta calcule diferentes funciones y obtenga diferentes resultados. La solución es inicializar los pesos de forma aleatoria. ¿Cómo hacemos esto? En Python con NumPy una forma de proceder sería la siguiente:

$$W^{[1]} = np.random.randn(n_h^{[1]}, n_x) * t$$
$$b^{[1]} = np.zeros(shape = (n_h^{[1]}, 1))$$
$$W^{[1]} = np.random.randn(n_h^{[1]}, n_x) * t$$
$$b^{[1]} = np.zeros(shape = (n_h^{[1]}, 1))$$

donde *t* es un número pequeño, generalmente decimal. *np.random.randn()* genera una variable gaussiana aleatoria, que es habitual multiplicar por un número pequeño, por ejemplo, 0.01. Los vectores de sesgos no tienen el problema de ruptura de simetría, por lo que es correcto inicializarlo a ceros, mientras las matrices de pesos se inicialicen aleatoriamente.

¿Por qué se prefieren valores de inicialización pequeños? Para las funciones de activación como sigmoide y *tanh*, si los pesos son muy grandes, estarán en los extremos izquierdo y derecho de la gráfica, donde la pendiente de la derivada es muy poco pronunciada o incluso plana, haciendo el aprendizaje más lento. Si por ejemplo estamos ante un problema de clasificación donde la función de activación de la capa de salida es la función sigmoide, no es adecuado tener pesos iniciales grandes, que saturarían pronto la función y el aprendizaje. Ver las gráficas del apartado de funciones de activación. En caso de no tener estas funciones de activación (por ejemplo, que todas las funciones de activación sean ReLU), el tamaño de los pesos iniciales no sería un gran problema. Para una red neuronal pequeña, un valor como 0.01 o similar funciona bien, pero para redes neuronales profundas, esa constante deberá cambiarse por otros valores. Se hablará de ello en el próximo y último bloque. En cualquier caso, estos valores seguirían siendo pequeños y la explicación general dada en este apartado sigue siendo válida. Su implementación se puede ver en el **archivo 2**.

En el siguiente apartado, se explicará la retropropagación y el descenso del gradiente para una red neuronal de dos capas, que es lo que permite que la red aprenda los parámetros adecuados.

## Descenso del gradiente en redes neuronales

El algoritmo descenso del gradiente, con el cual se aplica la retropropagación (*backpropagation*), ya se explicó en el apartado de regresión logística. A continuación, veamos cómo se particulariza el cálculo a una red con una capa oculta sin límite en el número de unidades ocultas. Los parámetros que componen la red y hay que optimizar son $W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}$ donde las dimensiones de cada parámetro son:

$$W^{[l]} \in \mathbb{R}^{n_h^{[l]} \times n_h^{[l-1]}}; \quad b^{[l]} \in \mathbb{R}^{n_h^{[l]}}$$

siendo $n_h^{[l]}$ el número de unidades ocultas ("neuronas") de la capa l-ésima. El subíndice "h" es la inicial del término *hidden*, traducido oculto/a/s. Por simplificar la explicación y por la naturaleza de los ejemplos de código que se muestran en este trabajo, supondremos que estamos realizando clasificación binaria, con la capa de salida compuesta por una única neurona, por lo que $n_h^{[2]} = 1$. Para un conjunto de entrenamiento, la matriz de entrada, $X$, estaría formada por $m$ ejemplos de entrenamiento con $n_x$ características de entrada. La función de coste sería:

$$J(W, b) = J\left(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}\right) = J\left(A^{[2]}, Y\right) = \frac{1}{m} L\left(\hat{Y}, Y\right) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)$$

con $\hat{y}^{(i)} = a^{[2](i)}$y $A^{[2]} = \hat{Y}$. Hay muchas maneras de expresar la función, aunque el objetivo sigue siendo minimizar este coste global. En nuestro caso se particulariza a la función de pérdida de entropía cruzada que ya se explicó. Por comodidad, de nuevo nombraremos las derivadas parciales como:

$$dW^{[1]} = \frac{\partial J(W,b)}{\partial W^{[1]}};\quad dW^{[2]} = \frac{\partial J(W,b)}{\partial W^{[2]}};\quad db^{[1]} = \frac{\partial J(W,b)}{\partial b^{[1]}};\quad db^{[2]} = \frac{\partial J(W,b)}{\partial b^{[2]}}$$

Y el cálculo del descenso del gradiente para entrenar y optimizar los parámetros consistirá en repetir las siguientes ecuaciones, actualizando los parámetros tras cada iteración como ya se mencionó en el bloque de la regresión logística. Esto se hace hasta, como mínimo, cierta convergencia, siendo lo ideal el mínimo absoluto.

$$\begin{aligned} W^{[1]} &= W^{[1]} - \alpha dW^{[1]} \\ W^{[2]} &= W^{[2]} - \alpha dW^{[2]} \\ b^{[1]} &= b^{[1]} - \alpha db^{[1]} \\ b^{[2]} &= b^{[2]} - \alpha db^{[2]} \end{aligned}$$

Por supuesto se utilizará la versión vectorizada de los parámetros en la derivación, no sumatorios. El gráfico computacional de la red neuronal es:

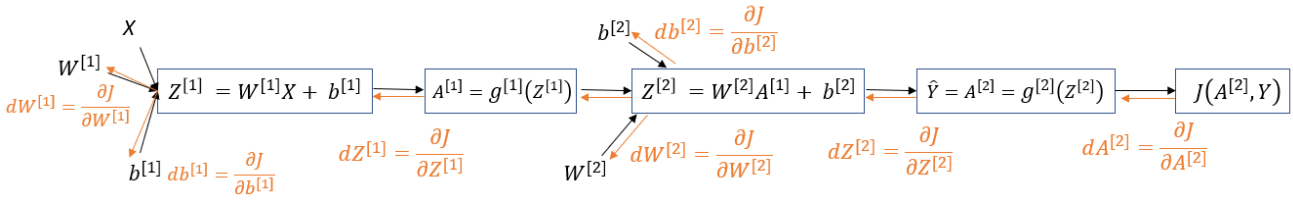


*Figura 33.* Gráfico computacional para una red neuronal con dos capas para un conjunto de entrenamiento completo.

En color negro están los pasos de la propagación hacia delante, que se realiza de izquierda a derecha, y en naranja están los pasos de la retropropagación, que va de derecha a izquierda. Recordemos que para la retropropagación se utiliza la regla de la cadena.

Las derivadas dependerán de la función de coste y de activación elegida en cada caso. En el ejemplo visto de clasificación binaria, se puede seguir utilizando como función de pérdida la entropía cruzada. Respecto a las funciones de activación, sus derivadas se dieron en el apartado de funciones de activación. Al no haber una elección fija de funciones como en el bloque de la regresión logística, no se pueden dar los resultados finales, pero sí desarrollar un poco más las derivadas teniendo en cuenta lo siguiente: si estamos suponiendo una única neurona en la capa de salida en un problema de clasificación, podemos utilizar la función sigmoide. Con estas condiciones, los pasos de retropropagación para cada ejemplo de entrenamiento, $x^{(i)}$, serían:

$$\begin{aligned} dz^{[2]} &= a^{[2]} - y \\ dW^{[2]} &= dz^{[2]}a^{[1]^T} = (a^{[2]} - y)a^{[1]^T} \\ db^{[2]} &= dz^{[2]} = (a^{[2]} - y) \\ dz^{[1]} &= W^{[2]^T}dz^{[2]} * g^{[1]\prime}(z^{[1]}) \\ dW^{[1]} &= dz^{[1]}x^T \\ db^{[1]} &= dz^{[1]} \end{aligned}$$

donde el superíndice *T* indica la traspuesta del vector o matriz, el símbolo "*" indica producto elemento a elemento y el apóstrofe la derivada con respecto al argumento. El valor de $dz^{[2]}$ es el mismo que en el hallado en el descenso del gradiente para la regresión logística porque estamos utilizando la misma función de

pérdida y la función sigmoide en la capa de salida. Luego, al ser $z^{[l]}$ una función lineal, su derivada es inmediata para los valores de $dW^{[2]}$ y $db^{[2]}$. Cuando pasamos a la capa oculta, desconocemos la función de activación, pero al ser su argumento una función lineal, podemos expresar $dz^{[1]}$ de esa manera. Y finalmente los valores de $dW^{[1]}$ y $db^{[1]}$ tendrán la misma forma que para la capa de salida, puesto que es la misma función lineal. Pero no conocemos el valor de $dz^{[1]}$. Se puede profundizar en los cálculos y notación estricta matemática en las referencias. Y los pasos generales sumados todos los ejemplos de entrenamiento, pero obviamente vectorizados**,** serían:

$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$$
$$db^{[2]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[2](i)} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$
$$dZ^{[1]} = W^{[2]^T} dZ^{[2]} * g^{[1]\prime}\left(Z^{[1]}\right)$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[1](i)} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

donde para el sumatorio ya se ha indicado cómo se haría con NumPy, puesto que se utiliza en el **archivo 2**. El argumento *axis = 1* indica que la suma se realiza a lo largo de las filas, mientras que *keepdims = True* indica que las dimensiones reducidas se mantendrán como tamaño uno, es decir, que no tendremos vectores unidimensionales, sino bidimensionales. Por ejemplo, si no lo utilizásemos, las dimensiones de $db^{[2]}$ serían $(n_h^{[2]},)$ en vez de $(n_h^{[2]}, 1)$, lo cual tiene mucha importancia a la hora de hacer cálculos en NumPy. Ver **anexo 2**.

Aunque en este trabajo los ejemplos y cálculos van a estar centrados, por simplicidad, en la clasificación binaria, existen más tipos de clasificación. Aprovechando la **figura 28**, veamos sus diferencias:

- Clasificación binaria: se refiere a las tareas de clasificación que tienen dos etiquetas de clase. Ejemplos incluyen la detección de correo electrónico no deseado (spam o no), la predicción de abandono (si un cliente se irá o no) y la detección de fraude (fraudulento o no).
- Clasificación multiclase: se refiere a las tareas de clasificación que tienen más de dos etiquetas de clase. Ejemplos incluyen la clasificación de imágenes (identificación de objetos en una imagen), el análisis de sentimientos (positivo, negativo o neutral) y el reconocimiento del habla (reconocimiento de palabras habladas).
- Clasificación multietiqueta: se refiere a las tareas de clasificación que tienen múltiples etiquetas de clase para cada instancia. Ejemplos incluyen la anotación de imágenes (identificación de múltiples objetos en una imagen), la clasificación de género musical (clasificación de una canción en múltiples géneros) y la categorización de texto (asignación de múltiples temas a un documento).

Algunos cálculos no difieren en absoluto, como por ejemplo el resultado de $dZ^{[2]}$. La demostración se puede encontrar en las referencias.

Con esto se finaliza este tercer bloque. Se ha mostrado cómo configurar una red neuronal con una capa oculta (o dos capas), a inicializar sus parámetros correctamente, hacer predicciones utilizando la propagación hacia delante y cómo calcular la retropropagación para aprender los valores de los parámetros que minimizan el coste global utilizando el algoritmo descenso del gradiente. Y se ha puesto en práctica en el **archivo 2**. En el siguiente y último bloque se hablará de todo esto, pero generalizado a una red neuronal de L-capas. Todos estos bloques han sido explicaciones preliminares y reducciones del problema general que ayudarán a entenderlo.

# 4

# Redes neuronales profundas

Recapitulando, hasta ahora se ha discutido y ejemplificado la propagación hacia delante y retropropagación para la regresión logística (entendida como pseudo red neuronal) y para una red neuronal de dos capas (una única capa oculta). A su vez, se ha explicado la importancia de la vectorización, por qué es importante inicializar los parámetros aleatoriamente, y se han propuesto y explicado varias funciones de activación. En este último bloque se explicará cómo pasar de una red neuronal poco profunda (*shallow Neural Network*) a una red neuronal profunda (*deep Neural Network*). Este bloque viene acompañado de dos archivos, el **archivo 3** y el **archivo 4**. En el primero se arman todas las funciones que componen una red neuronal de número de capas personalizables, utilizando como base las funciones que componen una red neuronal de una capa oculta. En el archivo 4 se aplican estos dos modelos a un conjunto de datos y se analizan y comparan los resultados.

En primer lugar, ¿qué es una red neuronal profunda? Simplemente, aquella que tiene más de una, o dos, capas ocultas. Este término fue acuñado en la primera década de los 2000s, cuando se demostró que redes neuronales con muchas capas podían ser entrenadas si los pesos de la red eran inicializados de formas adecuadas. En la publicación "*Greedy Layer-Wise Training of Deep Networks*" de Yoshua Bengio et al., se presentó un sólido argumento de que los métodos de aprendizaje automático profundo, es decir, métodos con muchos pasos de procesamiento, o equivalentemente con representaciones jerárquicas de las características de los datos, son más eficientes para los problemas complejos que los métodos superficiales, de los que son ejemplos las redes neuronales de dos capas o las máquinas de vectores de soporte. No solo eso, sino que hay problemas o funciones que las redes neuronales profundas pueden aprender que las redes neuronales poco profundas son incapaces.

Es muy difícil, sobre todo cuanto más complejo es un problema, elegir desde el primer momento el número de capas adecuadas para la red. Una forma de proceder al abordar un problema de aprendizaje automático en el que se quiera implementar redes neuronales profundas podría ser el siguiente: aplicar la regresión logística como pseudo red neuronal, luego redes poco profundas de una y dos capas ocultas y, continuar aumentando el número de capas hasta el punto óptimo. Es decir, considerar el número de capas ocultas como otro hiperparámetro que ajustar y validar con el conjunto de datos de desarrollo. Por ello, esta es otra de las razones por las que este trabajo se ha estructurado de la manera que lo ha hecho: no solo para explicar paso a paso conceptos y aplicaciones de las redes neuronales de menor a mayor complejidad, sino porque en proyectos reales se hacen pruebas de concepto con modelos menos complejos, de los cuales se saca información para que los equipos luego puedan decidir mejor qué modelo, cuántas capas y cuántas unidades ocultas desarrollar.

## Representación de redes neuronales con L capas

La notación y explicaciones dadas para la red con una capa oculta siguen siendo válidas y únicamente hay que matizar y ampliar la notación. En el bloque anterior se habló de notación para cada ejemplo de entrenamiento, por lo que para no repetir y ampliar explicaciones, nos referiremos a todo el conjunto de entrenamiento $X \in \mathbb{R}^{n_x \times m}$. Cada capa tendrá $n_h^{[l]}$ unidades ocultas y una salida o activación, $A^{[l]} \in \mathbb{R}^{n_h^{[l]} \times m}$. La capa de salida genera una matriz de predicciones $A^{[L]} = \hat{y}$ o $\hat{Y} \in \mathbb{R}^{n_y \times m}$. Los pesos y sesgos siguen teniendo las mismas dimensiones previamente mencionadas:

$$W^{[l]} \in \mathbb{R}^{n_h^{[l]} \times n_h^{[l-1]}}; \quad b^{[l]} \in \mathbb{R}^{n_h^{[l]}}$$

Para más información o dudas sobre notación, consultar anexos. Veamos una red neuronal de L capas y sus partes, con el estilo que se ha mostrado en los bloques anteriores.

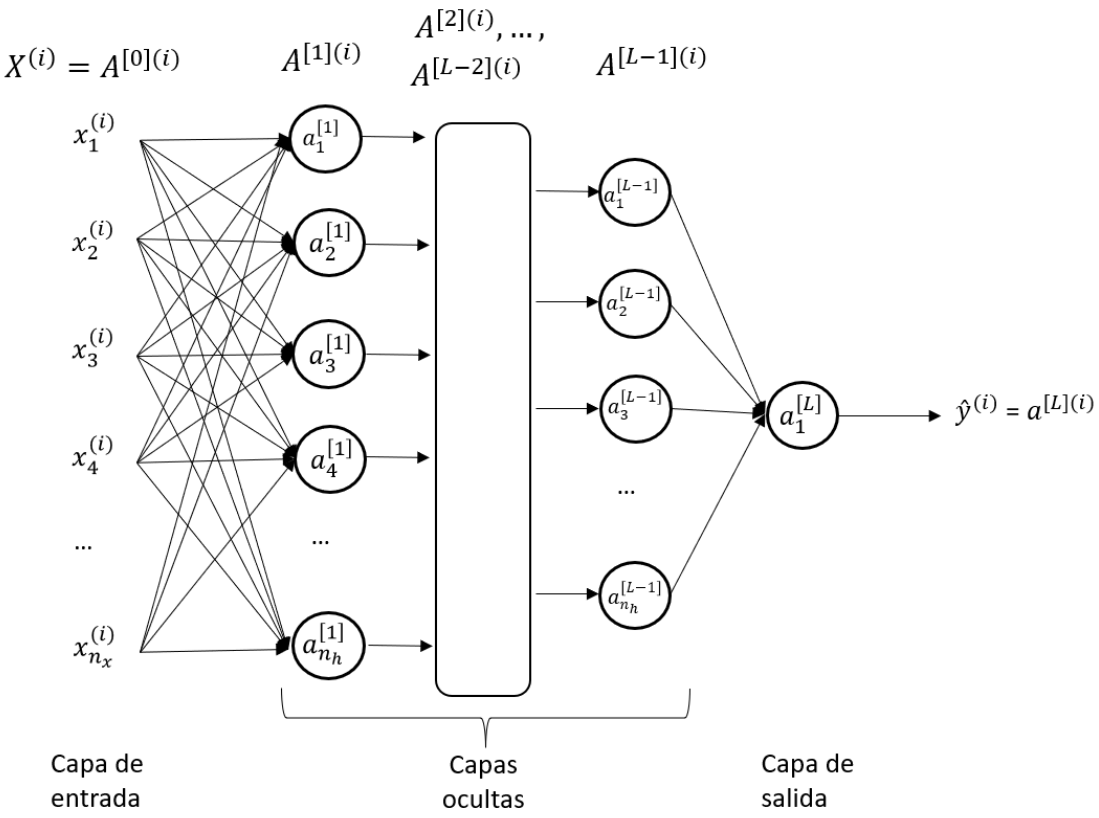


*Figura 34.* Representación genérica de una red neuronal con L-1 capas ocultas (o L capas, según se prefiera). Se han omitido los superíndices *(i)* en las funciones de activación por espacio en la figura. Los nodos representan entradas, activaciones o salidas, mientras que las aristas o líneas representan pesos y sesgos, $(w_{kj}^{[l]}, b_k^{[l]})$.

Para ejemplificar, la siguiente tabla del **archivo 3** muestra en detalle las dimensiones y activaciones por capas para un conjunto de entrenamiento de dimensiones $n_x = 12288$ y $\mathrm{m} = 209$:

| | Dimensiones de W | Dimensiones de b | Activacion lineal | Dimensiones de la activación lineal |
|---|---|---|---|---|
| **Capa 1** | $(n^{[1]}, 12288)$ | $(n^{[1]}, 1)$ | $Z^{[1]} = W^{[1]}X + b^{[1]}$ | $(n^{[1]}, 209)$ |
| **Capa 2** | $(n^{[2]}, n^{[1]})$ | $(n^{[2]}, 1)$ | $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$ | $(n^{[2]}, 209)$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| **Capa L-1** | $(n^{[L-1]}, n^{[L-2]})$ | $(n^{[L-1]}, 1)$ | $Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$ | $(n^{[L-1]}, 209)$ |
| **Capa L** | $(n^{[L]}, n^{[L-1]})$ | $(n^{[L]}, 1)$ | $Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$ | $(n^{[L]}, 209)$ |

*Tabla 3*. Dimensiones de los parámetros de una red de L capas tras la inicialización para $n_x = 12288$ y $\mathrm{m} = 209$. Extraída del archivo 3 de este trabajo.

Se menciona durante todo el trabajo que cuando se calcula en Python la función de activación lineal, $Wx + b$, se ejecuta el *broadcasting* para vectores y matrices. Por ejemplo, si:

$$W = \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix} \quad X = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix} \quad b = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

Entonces la función lineal de activación, tras el *broadcasting*, es:

$$WX + b = \begin{bmatrix} (w_{00}x_{00} + w_{01}x_{10} + w_{02}x_{20}) + b_0 & (w_{00}x_{01} + w_{01}x_{11} + w_{02}x_{21}) + b_0 & \cdots \\ (w_{10}x_{00} + w_{11}x_{10} + w_{12}x_{20}) + b_1 & (w_{10}x_{01} + w_{11}x_{11} + w_{12}x_{21}) + b_1 & \cdots \\ (w_{20}x_{00} + w_{21}x_{10} + w_{22}x_{20}) + b_2 & (w_{20}x_{01} + w_{21}x_{11} + w_{22}x_{21}) + b_2 & \cdots \end{bmatrix}$$

Las redes neuronales profundas funcionan realmente bien para muchos problemas, y no es sólo que necesiten ser grandes redes neuronales, sino que específicamente necesitan ser profundas y tener muchas capas ocultas. ¿A qué se debe esto? ¿Qué está calculando una red profunda?

## ¿Por qué necesitamos redes profundas?

Retomemos el ejemplo mencionado en la introducción sobre los datos no estructurados con el problema del reconocimiento facial. La imagen de entrada vectorizada, $X$, se introduce en una primera capa de neuronas. Esta es la primera capa oculta, que extrae algunas características. La salida de esta primera capa oculta se envía a una segunda capa oculta, cuya salida a su vez se envía a una tercera capa, y así sucesivamente hasta que, finalmente, se llega a la capa de salida, que estima la probabilidad de que se trate de una persona en particular. Algo muy interesante sería tratar de visualizar lo que estas capas ocultas están intentando calcular en una red neuronal que ha sido entrenada con muchas imágenes de rostros.

En la primera capa oculta, se podría encontrar una neurona que está buscando la línea vertical del inferior de la imagen o un borde vertical. Una segunda neurona podría estar buscando un borde orientado. Una tercera neurona buscando una línea con esa orientación, y así sucesivamente. Al tratar con imágenes, en las primeras capas de una red neuronal es posible que las neuronas busquen líneas o bordes muy cortos en la imagen. En las siguientes capas ocultas, estas neuronas pueden aprender a agrupar muchas líneas y segmentos de bordes cortos para buscar partes de rostros. Cada una de las pequeñas cajas cuadradas de la **figura 35** es una visualización de lo que esa neurona podría estar tratando de detectar. En la última capa oculta de este ejemplo, la red neuronal puede estar agregando diferentes partes de las caras para intentar detectar la presencia o ausencia de formas faciales más grandes y gruesas. Por último, al detectar en qué medida el rostro se corresponde con distintas formas faciales, se crea un variado conjunto de características que ayudan a la capa de salida a determinar la identidad de la persona retratada.

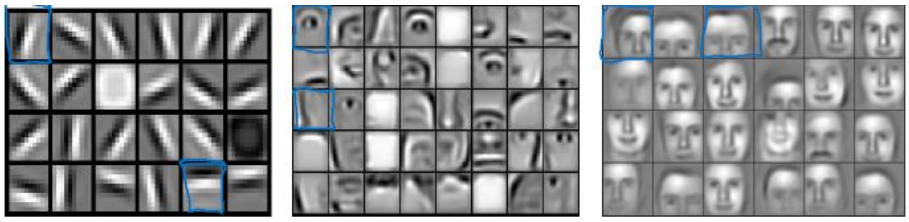

*Figura 35.* Visualización del aprendizaje de las neuronas en distintas capas de la red neuronal para un problema de reconocimiento facial.

El hecho de que de las redes neuronales puedan aprender por sí mismas estos detectores de características en las distintas capas ocultas es fundamental, y justifica su uso y su éxito. En este ejemplo, nadie la instruyó para que buscara pequeños bordes cortos en las primeras capas; ojos, narices y partes de la cara en las siguientes capas, y formas de cara más completas en las últimas capas. La red neuronal es capaz de averiguar estas cosas por sí misma a partir de los datos. En la visualización de la **figura 35**, las neuronas de las primeras capas se muestran mirando zonas pequeñas para buscar bordes. En las intermedias se miran porciones más

grandes de la imagen, y las capas finales se miran ventanas aún más grandes. Se podría utilizar esta red neuronal y entrenarla para que, en vez de reconocer rostros, reconociera coches, elefantes, sillas, etc, si se entrenara con el conjunto de imágenes correspondiente. Simplemente añadiendo diferentes datos, la red neuronal aprende automáticamente a detectar características muy diferentes para intentar predecir si la imagen es de un coche, o de una persona, o de aquello para lo que se la haya entrenado. Así es como podría funcionar una red neuronal para aplicaciones de visión artificial por ordenador.

Este procedimiento no solo aplica a imágenes, sino a gran variedad de datos, como el reconocimiento de voz. Aunque no podemos visualizar el sonido, si podemos dividirlo en partes sencillas que, agrupadas, forman conjuntos complejos. Por ejemplo, las primeras capas podrían especializarse en detectar características de bajo nivel de la forma de la onda de audio (el tono, ruido blanco, etc). Componiendo esas ondas de bajo nivel, las siguientes capas pueden aprender a detectar las unidades básicas del sonido, los fonemas. Esos fonemas forman palabras y las palabras forman frases y oraciones. Esta podría ser la forma en que aprendiese la red neuronal a reconocer la voz.

El patrón común a estos ejemplos es un aprendizaje de las características más sencillas en las primeras capas, mientras que según se avanza en la red neuronal, las capas profundas calculan características realmente complejas. Entonces, ¿por qué son necesarias tantas capas ocultas? La respuesta a esta pregunta es que queremos que la red neuronal aprenda funciones complejas. Las primeras capas de una red profunda aprenden funciones sencillas y, a medida que profundizamos, la red aprende diversas funciones sofisticadas que suelen ser incomprensibles para los humanos.

Otra pregunta a la que habría que responder es, ¿por qué no utilizar sólo unas pocas capas compuestas por un gran número de unidades ocultas? La respuesta está en la teoría de circuitos. La teoría de circuitos afirma que necesitamos un número exponencial de unidades ocultas en una red poco profunda para conseguir una precisión similar a la de una red profunda “pequeña” a la hora de calcular ciertas funciones complejas. Por tanto, para evitar ese factor exponencial y permitir que la red aprenda funciones complejas, es preferible una red profunda formada por muchas capas ocultas.

En cualquier caso, aunque la solución final a un problema sea el uso de una red neuronal profunda, al abordarlo se suele aplicar la estrategia de ir de lo más sencillo a lo más complejo, es decir, aplicar aprendizaje automático clásico, como la regresión logística, seguir con redes neuronales poco profundas y, finalmente, y si el problema lo necesita, crear redes neuronales profundas. A continuación, se abordará cómo implementar los componentes clave de toda red neuronal, es decir, la propagación hacia delante y la retropropagación. Esta vez, aplicado a una red neuronal de *L* capas.

## Los bloques constitutivos de las redes neuronales profundas

A estas alturas, queda claro que cada una de las capas de la red neuronal tiene sus parámetros $W^{[l]}, b^{[l]}$. En la propagación hacia delante, cada capa recibe como entrada un vector o matriz de activación $A^{[l-1]}$, donde si $l = 1$, $A^{[0]} = X$, y devuelve como salida un vector o matriz de activación $A^{[l]}$. En ese proceso de propagación hacia delante, vamos a almacenar todas las variables mencionadas, y también $Z^{[l]}$, en una variable “cache”. En la retropropagación, estas variables almacenadas de cada capa serán utilizadas para calcular los gradientes de los parámetros, $dW^{[l]}$ y $db^{[l]}$, necesarios para actualizar el valor de los parámetros con el algoritmo descenso del gradiente que se utiliza en este trabajo. Y no solo para realizar cálculos, también sirven para validar dimensiones, puesto que las derivadas de los parámetros tienen las dimensiones de las variables sin derivar. La entrada en cada capa en la retropropagación es $dA^{[l]}$ y esa “cache” mencionada, y las salidas son $dA^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$. Visualicemos las entradas que utilizamos y las salidas que producimos durante las propagaciones hacia delante y hacia atrás:

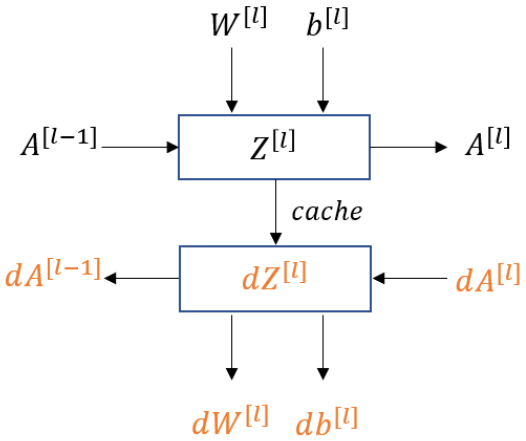


*Figura 36.* Visión general de entradas, salidas y cálculos que se realizan en cada capa de una red neuronal.

Aplicando este razonamiento a cada capa de la red neuronal, el proceso completo de cálculo es como sigue: se introduce a la primera capa la matriz de ejemplos de entrenamiento, $X$, la cual calcula las funciones de activación lineal y no lineal (*sigmoide, tanh, ReLU, etc*), $Z^{[1]}$ y $A^{[1]}$, utilizando los parámetros $W^{[1]}$ y $b^{[1]}$. Todos estos valores son almacenados en una variable *cache*. $A^{[1]}$ es el parámetro de entrada de la siguiente capa y el proceso de cálculo y almacenamiento se repite de esta manera en cada capa hasta que, en la última, se calcula la predicción, $A^{[L]}$. Esto compone la propagación hacia delante. Para la retroprogación, se hará una secuencia inversa al número de capas, empezando por la última hasta llegar a la primera. La entrada de cada capa en este paso será $dA^{[l]}$ y la salida será $dA^{[l-1]}$. A su vez en este proceso se obtienen los valores de $dZ^{[l]}$, $dW^{[l]}$ y $db^{[l]}$. La siguiente figura muestra en detalle el proceso completo:

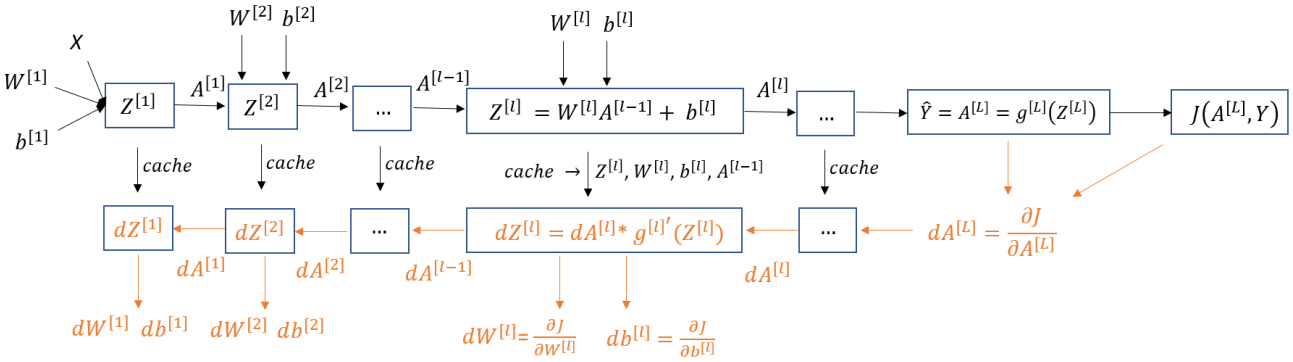


*Figura 37.* Gráfico computacional de una red neuronal de L capas. Muestra cada iteración de entrenamiento de los parámetros de la red profunda.

Al final de cada iteración se aplica el algoritmo del descenso del gradiente con la tasa de aprendizaje elegida. Vistos los bloques que constituyen una red neuronal profunda y su secuencia, pasemos a describir cómo es la propagación hacia delante en este tipo de red.

## Propagación hacia delante en una red neuronal profunda

Recordemos que la propagación hacia delante es la forma que tiene la red neuronal de predecir un resultado dado un conjunto de entrenamiento y partiendo de unos pesos y sesgos que hay que inicializar. Y esos pesos y sesgos se irán optimizando mediante la retropropagación y el algoritmo de actualización elegido. El concepto general y el cálculo detallado de lo que sucede en cada capa fue ejemplificado en el bloque 3. Lo

que se hará a continuación es generalizarlo para un número *L* de capas. Tomando todo el conjunto de entrenamiento, $X \in \mathbb{R}^{n_x \times m}$, las fórmulas que rigen la propagación hacia delante son:

- Fórmula general de activación lineal para cada capa:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \text{ con } X = A^{[0]}$$

( 5 )

con representación matricial:

$$Z^{[l]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ z^{[l](1)} & z^{[l](2)} & \dots & z^{[l](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

- Fórmula general de activación para cada capa :

$$A^{[l]} = g^{[l]}\left(Z^{[l]}\right), \ 1 \leq l \leq L$$

( 6 )

donde $g^{[l]}$ es la función de activación no lineal de la capa L-ésima. La representación matricial es:

$$A^{[l]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ a^{[l](1)} & a^{[l](2)} & \dots & a^{[l](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

Veamos un ejemplo donde, teniendo en mente la **figura 34**, el conjunto de entrenamiento tiene dimensiones $X \in \mathbb{R}^{3 \times m}$ y la red neuronal es de tres capas ocultas (cuatro capas) con la siguiente distribución:

$$n_x = n_h^{[0]} = 3 \ ; n_h^{[1]} = 5 \ ; n_h^{[2]} = 5 \ ; n_h^{[3]} = 3 \ ; n_y = n_h^{[4]} = 1$$

La forma no vectorizada y, por tanto, poco eficiente, de calcular la propagación hacia delante sería la siguiente:

*for i = 1, … , m:*

$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$
$a^{[1](i)} = g^{[1]}\left(z^{[1](i)}\right)$
$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = g^{[2]}\left(z^{[2](i)}\right)$
$z^{[3](i)} = W^{[3]}a^{[2](i)} + b^{[3]}$
$a^{[3](i)} = g^{[3]}\left(z^{[3](i)}\right)$
$z^{[4](i)} = W^{[4]}a^{[3](i)} + b^{[4]}$
$\hat{y}^{(i)} = a^{[4](i)} = g^{[4]}\left(z^{[4](i)}\right)$

La forma vectorizada que se debe utilizar siempre a nivel de código por eficiencia es:

$Z^{[1]} = W^{[1]}X + b^{[1]} \rightarrow (5 \times m) = (5 \times 3)(3 \times m) + (5 \times m)$
$A^{[1]} = g^{[1]}\left(Z^{[1]}\right) \rightarrow (5 \times m)$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \rightarrow (5 \times m) = (5 \times 5)(5 \times m) + (5 \times m)$
$A^{[2]} = g^{[2]}\left(Z^{[2]}\right) \rightarrow (5 \times m)$
$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]} \rightarrow (3 \times m) = (3 \times 5)(5 \times m) + (3 \times m)$
$A^{[3]} = g^{[3]}\left(Z^{[3]}\right) \rightarrow (3 \times m)$

$Z^{[4]} = W^{[4]}A^{[3]} + b^{[4]} \rightarrow (1 \times m) = (1 \times 3)(3 \times m) + (1 \times m)$
$\hat{Y} = A^{[4]} = g^{[4]}(Z^{[4]}) \rightarrow (1 \times m)$

donde a continuación de las ecuaciones se ha puesto su cálculo dimensional. Y las dimensiones del vector de sesgos pasan a matriz con $m$ columnas por *broadcasting*. Resulta que, al implementar una red neuronal profunda, una de las formas de aumentar las probabilidades de estar implementando el código correctamente es pensar sistemática y cuidadosamente sobre las dimensiones de las matrices con las que se está trabajando. Es muy importante señalar que las derivadas de los parámetros tienen las mismas dimensiones. Es decir, una de las herramientas de depuración de las que disponemos es una simple comprobación con papel y lápiz. Y, conocidas las dimensiones que debe tener cada parámetro y resultado, podemos preguntar en el código el tamaño de estos en cualquier momento para saber que el resultado es el que debe. En Python, el método que se ha utilizado en las funciones de este trabajo para asegurar que las dimensiones son correctas es *assert()*. Y la forma abreviada en la que se implementaría este ejemplo en una función de Python sería:

*for l = 1, … , 4:*

$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}, con\ X = A^{[0]}$
$A^{[l]} = g^{[l]}(Z^{[l]})$

Y, como se dijo en el bloque 3, en los ejemplos prácticos que se han visto en este trabajo se ha tratado con problemas de clasificación binaria, por lo que $n_y = 1$. Por lo tanto, la forma general de calcular la propagación hacia delante es con el bucle de 1 a L, es decir, *for l = 1, … , L*. De esta manera se realiza la propagación hacia delante y el cálculo dimensional de las matrices de los parámetros. Lo discutido en este apartado debería resultar familiar, puesto que la explicación principal se realizó en el bloque de la red neuronal con una capa oculta y aquí lo que se ha hecho es "repetir" el cálculo el número necesario de veces. A continuación, se verá la retropropagación.

## Retropropagación en redes neuronales profundas

La retropropagación es la forma que tiene la red neuronal de afinar los parámetros con los que se realizan las predicciones. A la hora de implementar la retropropagación, utilizaremos la misma lógica que para el modelo de dos capas, pero aprovecharemos la "cache" almacenada. En cada iteración de la propagación hacia delante se guarda una *cache* que contiene los valores $Z^{[l]}, W^{[l]}, b^{[l]}$ y $A^{[l]}$, que se utilizan para calcular los gradientes. En cada paso, se utiliza el *cache* correspondiente a la capa $l$ para realizar la retropropagación sobre la capa $l$. La entrada de cada capa en este paso será $dA^{[l]}$ y la salida será $dA^{[l-1]}$. A su vez en este proceso se obtienen los valores de $dZ^{[l]}$, $dW^{[l]}$ y $db^{[l]}$. Veamos las fórmulas que rigen la retropropagación, independientemente de cómo sea la función de coste:

- Fórmula general de la derivada de la activación lineal en cada capa:

$$dZ^{[l]} = \frac{\partial J(W,b)}{\partial Z^{[l]}} = dA^{[l]} * g^{[l]\prime}(Z^{[l]}) \quad (7)$$

  donde "*" es multiplicación elemento a elemento.

- Fórmula general de la derivada de la función de activación en cada capa:

$$dA^{[l-1]} = \frac{\partial J(W,b)}{\partial A^{[l-1]}} = W^{[l]^T} dZ^{[l]} \quad (8)$$

Usando esta fórmula, podemos reescribir la derivada de la activación lineal como:

$$dZ^{[l]} = W^{[l+1]^T} dZ^{[l+1]} * g^{[l]\prime}(Z^{[l]})$$

- Fórmulas generales de las derivadas de los parámetros de la red neuronal en cada capa:

$$dW^{[l]} = \frac{\partial J(W,b)}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]^T}$$

$$db^{[l]} = \frac{\partial J(W,b)}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

( 9 )

Como vemos en la **figura 37** en la retropropagación, el primer cálculo es $dA^{[l]}$, seguido de $dZ^{[l]}$ y con este valor se puede calcular $dA^{[l-1]}$, $dW^{[l]}$ y $db^{[l]}$. Todas las ecuaciones se obtienen mediante la regla de la cadena. Por supuesto se utilizará la versión vectorizada de los parámetros en la derivación, no sumatorios. La retropropagación de una iteración de aprendizaje de la red neuronal es:

$dA^{[L]}$-> $dZ^{[L]}$ -> $dW^{[L]}$, $db^{[L]}$

*for l = (L-1), … , 1:*

$dZ^{[l]}$

$dW^{[l]}$

$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepdims = True)$

$dA^{[l-1]}$

donde se ha añadido la forma vectorizada de calcular $db^{[l]}$ en Python, tal y como se explicó en el bloque de redes neuronales superficiales. No se ha hecho el desglose con el ejemplo del apartado anterior (propagación hacia delante) por ser muy repetitivo y considerar que el procedimiento queda suficientemente ejemplificado y es extrapolable. Antes de unir todo en un algoritmo de aprendizaje completo con el descenso del gradiente, se hará una breve discusión sobre los parámetros e hiperparámetros de las redes neuronales profundas y sobre otros tipos de inicialización.

## Parámetros e hiperparámetros

Expliquemos de forma general los hiperparámetros que se utilizan para una red de L capas y qué hacen. Para ser efectivo desarrollando redes neuronales profundas no solo se necesita una correcta organización e inicialización de los parámetros de la red, sino también definir correctamente los hiperparámetros. Los hiperparámetros son un conjunto de parámetros que determinan cómo se entrena y la estructura de la red neuronal, y que controlan los valores que finalmente tendrán $W^{[l]}$ y $b^{[l]}$. Juegan un papel vital para decidir si la red neuronal que se entrena es aplicable al problema que se está tratando de resolver o no. Los hiperparámetros utilizados en este trabajo son:

- Tasa de aprendizaje, α.
- Número de iteraciones del algoritmo de aprendizaje.
- Número de capas de la red neuronal, *L*.
- Número de unidades ocultas en cada capa, $n_h^{[l]}$.
- Funciones de activación en cada capa (*sigmoide, tanh, ReLU, etc*).

En el aprendizaje profundo existen más tipos de hiperparámetros. Algunos que no se utilizan en este trabajo:

- Momento o impulso: es un enfoque de optimización por descenso de gradiente que añade un porcentaje del vector de actualización anterior al vector de actualización actual para acelerar el proceso de aprendizaje.
- Tamaño del lote (*batch*): se refiere a la cantidad de datos que se propagan a través de la red neuronal en cada iteración. En otras palabras, es el número de muestras que se utilizan para calcular el error y actualizar los pesos de la red. El tamaño del lote se configura antes del entrenamiento y puede ser un número fijo o un número variable.
- Regularización: técnica utilizada para evitar el sobreajuste (*overfitting*) y mejorar la generalización de las redes neuronales. Consiste en añadir un término de regularización a la función de pérdida, que penaliza los pesos grandes o las arquitecturas complejas del modelo.

Previamente, en la época del aprendizaje automático clásico, no había tantos hiperparámetros que ajustar. En la época del aprendizaje profundo, hay muchos y muchas posibles combinaciones de ellos. Su aplicación es un proceso muy empírico, es decir, es un proceso donde hay que realizar muchas pruebas hasta llegar a la combinación adecuada. Obviamente la experiencia es un grado que puede ayudar a acortar los procesos de prueba y error. Se podría resumir el proceso en un ciclo idea-código-experimento: se decide aplicar ciertos hiperparámetros con ciertos valores; se codifica la red con esas especificaciones; se ejecuta el modelo entrenado sobre conjuntos de desarrollo y de prueba; se repite el ciclo, ajustando de nuevo los hiperparámetros para procurar obtener mejores resultados que los anteriores.

Por supuesto influyen más factores, como la(s) métrica(s) fundamentales y las métricas suficientes o complementarias que tiene que cumplir nuestro modelo. Es decir, puede que para una aplicación la métrica fundamental sea la precisión y en otra el valor F1. Y disminuir el error de una métrica en la mayoría de los casos provoca empeorar los resultados de otras. Se ha visto en el código de este trabajo cómo disminuir el coste en el conjunto de entrenamiento implicaba una mejor precisión, pero provocaba sobreajuste y peor rendimiento en el conjunto de validación.

Hay múltiples campos en los que se aplica el aprendizaje profundo, tanto para datos estructurados (publicidad online, motores de búsqueda, etc) como no estructurados (visión por ordenador, reconocimiento de voz, procesamiento de lenguaje, etc) y en cada uno funcionan mejor unos hiperparámetros y sus combinaciones que otros. Además, los valores óptimos no son estáticos, puede que con el tiempo haya que cambiarlos por otros mejores debidos a cambios en, por ejemplo, los componentes del ordenador (CPUs y GPUs mejores) u otros motivos, como nuevos tipos de datos que se tengan que predecir.

## Otros tipos de inicialización

En el primer apartado de este bloque sobre las redes neuronales profundas, se menciona que "[..] *se demostró que redes neuronales con muchas capas podían ser entrenadas si los pesos de la red eran inicializados de formas adecuadas*". Esas formas de inicialización adecuada se refieren, en general, a inicializar los pesos de la red neuronal de forma distinta a la aleatoria. Es decir, que la inicialización aleatoria de parámetros mencionada en el bloque sobre las redes neuronales poco profundas es insuficiente para entrenar adecuadamente una red neuronal profunda. Como nota histórica es interesante mencionar lo siguiente:

> *[…] Mientras el aprendizaje profundo se abría paso en la industria, la comunidad investigadora no se quedaba quieta. El descubrimiento de que el uso eficiente de las GPU y de la potencia de cálculo en general era tan importante hizo que la gente examinara supuestos que se habían mantenido durante mucho tiempo y se planteara preguntas que quizá deberían haberse hecho hace mucho tiempo, a saber, ¿por qué exactamente no funciona bien la retropropagación? La idea de preguntarse por qué no funcionaban los métodos antiguos, en lugar de por qué sí lo hacían los nuevos, llevó a Xavier Glort y Yoshua Bengio a escribir en 2010 "Understanding the*

*difficulty of training deep feedforward neural networks". En él, analizaban dos hallazgos muy significativos:*

- *La función de activación no lineal elegida para las unidades ocultas de una red neuronal influye mucho en el rendimiento, y la que se suele utilizar por defecto no es una buena elección.*
- *No era tan problemático en sí elegir pesos aleatorios, sino elegir pesos aleatorios sin tener en cuenta para qué capa son los pesos. El viejo problema del desvanecimiento del gradiente ocurre, básicamente, porque la retropropagación implica una secuencia de multiplicaciones que invariablemente resultan en derivadas más pequeñas para las capas anteriores. Es decir, a menos que los pesos se elijan con escalas de diferencia según la capa a la que pertenezcan, este simple cambio produce mejoras significativas.*

En los **archivos 3** y **4** se utiliza la función *inicialización_profunda()* para inicializar los parámetros de una red profunda. Al inicializar los pesos, si en el código hubiera puesto una inicialización aleatoria:

```
parametros['W' + str(l)] = np.random.randn(dims_capas[l], dims_capas[l - 1]) * 0.01
```

como para el caso de la red neuronal poco profunda, sería una inicialización insuficiente para un modelo de *L* capas. Si ejecutásemos en el **archivo 4** la inicialización con esa línea, obtendríamos el siguiente resultado:


Veamos qué tal las predicciones:

```
In [18]: predicciones_entrenamiento_L = predecir(train_x, train_y, parametros_L)
         Precision: 65.55023923444976 %

In [19]: predicciones_test_L = predecir(test_x, test_y, parametros_L)
         Precision: 34.00000000000001 %
```

*Figura 38.* Curva de aprendizaje y precisión del conjunto de entrenamiento y de prueba en el modelo de red neuronal profunda del archivo 4. Caso en el que la inicialización de los pesos es aleatoria.

Comparemos estos resultados con los que se obtienen al utilizar una inicialización mejor, la utilizada en el archivo 4, y que a continuación se explicará:


Veamos qué tal las predicciones:

```
predicciones_entrenamiento_L = predecir(train_x, train_y, parametros_L)
```

```
Precision: 99.04306220095691 %
```

```
predicciones_test_L = predecir(test_x, test_y, parametros_L)
```

```
Precision: 82.0 %
```

*Figura 39.* Curva de aprendizaje y precisión del conjunto de entrenamiento y de prueba en el modelo de red neuronal profunda del archivo 4. Caso en el que la inicialización de los pesos no es aleatoria, sino del tipo Xavier.

Obtenemos un resultado peor con la inicialización aleatoria, tanto en la curva de aprendizaje como en la precisión del conjunto de entrenamiento y validación. Es decir, de acuerdo con el segundo punto de Glort y Bengio, el mayor problema del pobre rendimiento de la inicialización aleatoria para cada capa no es en sí esa inicialización, sino que sea la misma para cada capa, sin ajustar esa aleatoriedad a la circunstancia de cada capa: función de activación elegida para ella, número de unidades ocultas, posición de la capa en la red neuronal, etc.

Queda clara la importancia de una inicialización adecuada de los parámetros del modelo. Históricamente en el desarrollo de las redes neuronales, las primeras, que fueron de una capa oculta y luego de unas pocas capas ocultas, utilizaban una inicialización aleatoria. En la época del aprendizaje profundo, esta inicialización fue superada por otras más eficientes. Ahora bien, ¿cuál ha sido la inicialización elegida para los **archivos 3** y **4** en la red profunda? Se trata de la *inicialización Xavier* (*Glort*), cuya regla para cada peso es:

$$w_{kj}^{[l]} = N\left(0, \frac{1}{\sqrt{n_h^{[l]}}}\right)$$

donde $N$ representa una distribución normal con media cero y desviación estándar $1/\sqrt{n_h^{[l]}}$. El objetivo de la *inicialización de Xavier* es inicializar los pesos de forma que la varianza de las salidas de cada capa sea la misma que la varianza de las entradas de esa capa. Esta varianza constante ayuda a evitar que el gradiente explote o desaparezca. ¿Cómo se consigue esto en Python? Con la biblioteca NumPy podemos utilizar *np.random.randn().* Es una función de la biblioteca NumPy que devuelve una muestra (o muestras) de la distribución normal estándar. Por ejemplo, para obtener muestras aleatorias de la distribución normal con media *μ* y desviación estándar *σ*, se puede usar: *σ * np.random.randn(...) + μ*. Por eso mismo en la función de inicialización profunda, *inicialización_profunda()*, se escribe:

```
parametros['W' + str(l)] = np.random.randn(dims_capas[l], dims_capas[l - 1]) *
                           np.sqrt(dims_capas[l -1])
```

donde $\sigma = 1/\sqrt{n_h^{[l]}}$ y $\mu = 0$. Para llevar a cabo el entrenamiento de la red neuronal una vez elegida su arquitectura, lo primero que se debe hacer es inicializar sus parámetros. Si se parte de cero (sin un modelo pre-entrenado) es bastante común inicializar los pesos aleatoriamente para ayudar a la red, rompiendo su simetría. El fin es evitar que todas las neuronas de una capa acaben aprendiendo lo mismo. Se suelen generar los pesos de cada capa usando una distribución normal de media cero y varianza $1/n_h^{[l]}$ o $2/n_h^{[l]}$. Este valor de la varianza depende en parte de la función de activación que se coloque a la salida de la neurona. Los parámetros de sesgo se inicializan a cero. A partir de aquí se procede iterativamente siguiendo un algoritmo de optimización, que tratará de minimizar la diferencia entre la salida real y la estimada por la red.

En el siguiente y último apartado, se combinarán las dos propagaciones con el coste global para, finalmente, aplicar el descenso del gradiente a una red de L capas.

## Descenso del gradiente en una red neuronal profunda

Veamos cómo particularizar el algoritmo del descenso del gradiente para una red de *L* capas. A diferencia de los otros bloques, donde se juntó esta explicación con la de la retropropagación, al ser el caso general merecía un apartado independiente. Las dimensiones de los parámetros no han cambiado en ningún momento. Una vez calculada $A^{[L]}$ en la propagación hacia delante, hay que calcular el coste global para comprobar que el modelo está aprendiendo los parámetros óptimos. Como se ha hecho en todo este trabajo, se utilizará la pérdida de entropía cruzada, particularizada para una red con *L* capas, como función de pérdida:

$$J(W,b) = J\left(A^{[L]},Y\right) = \frac{1}{m}L\left(\hat{Y},Y\right) = \frac{1}{m}\sum_{i=1}^{m}L\left(\hat{y}^{(i)},y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\mathrm{a}^{[\mathrm{L}](i)}\right) + \left(1-y^{(i)}\right)\log\left(1-\mathrm{a}^{[\mathrm{L}](i)}\right)\right)$$

Dada esta función de coste, el valor de $dA^{[L]}$ es:

$$dA^{[L]} = \frac{\partial J}{\partial A^{[L]}} = -\frac{Y}{A^{[L]}} + \frac{1-Y}{1-A^{[L]}}$$

donde la ecuación se muestra vectorizada para todos los componentes. Se puede ver la demostración de la derivación en las referencias. La fórmula general para $dZ^{[l]}$ es conocida, ecuación *(7)*, pero se puede calcular con una expresión concreta si estamos en un problema de clasificación donde en la última capa la función de activación es sigmoide:

$$g^{[L]\prime}\left(Z^{[L]}\right) = \sigma'\left(Z^{[L]}\right) = \sigma\left(Z^{[L]}\right) * \left(1-\sigma\left(Z^{[L]}\right)\right) = A^{[L]} * \left(1-A^{[L]}\right)$$

porque $A^{[L]} = g^{[L]}\left(Z^{[L]}\right) = \sigma\left(Z^{[L]}\right)$. Por ello, juntando ambos resultados, llegamos a la conocida expresión:

$$dZ^{[L]} = dA^{[L]} * g^{[L]\prime}\left(Z^{[L]}\right) = A^{[L]} - Y$$

y el cálculo del descenso del gradiente para entrenar y optimizar los parámetros consistirá en repetir el siguiente bucle, actualizando los parámetros tras cada iteración. Esto se hace hasta, como mínimo, cierta convergencia, siendo lo ideal el mínimo absoluto.

*for l = 1, … , L:*

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

Se actualizan de esta manera los parámetros para cada capa, de 1 a L.

# Conclusión

Se ha querido recorrer de forma extensa, pero a la vez de la manera más sencilla posible, el mundo de las redes neuronales y el aprendizaje profundo, tanto a nivel histórico, como teórico y práctico. Es un campo amplísimo, casi infinito, del cual ha sido muy difícil elegir de qué hablar y cómo enfocarlo. Finalmente, puesto que me he introducido en las redes neuronales gracias a este máster, he decidido no empezar "la casa por el tejado", sino ampliar y profundizar en las bases de las redes neuronales y así tener un conocimiento sólido para proseguir con el estudio del aprendizaje profundo. ¿Qué sería lo siguiente?

Habría que abrir la "caja negra", adentrarse en los entresijos del aprendizaje profundo para comprender los procesos que impulsan su rendimiento y permiten generar buenos resultados de forma sistemática: afinar los hiperparámetros, los algoritmos de aprendizaje utilizados, y aplicar técnicas como la regularización y el *dropout*, estudiando la varianza y el sesgo. Esto se podría hacer con el mismo enfoque con el que se ha realizado este trabajo y utilizando la red neuronal general de *L* capas programada.

Entendidas las bases conceptuales y matemáticas, se podría pasar a utilizar para proyectos reales las bibliotecas y entornos de aprendizaje profundo que existen en la actualidad. Las principales se explican en un anexo. Estas bibliotecas facilitan que, con poco conocimiento de lo que realmente hace por detrás el código, se pueda llegar a trabajar con éxito en un proyecto de aprendizaje automático. Sin embargo, considero que a largo plazo, si se pretende trabajar en esta área, se deben conocer las bases explicadas en este trabajo. Armados con los conocimientos fundamentales y manejando alguna de estas bibliotecas de código, se podría iniciar el estudio de las famosas arquitecturas de redes neuronales, como las CNNs o RNNs.

Espero que este trabajo sea de interés para cualquiera que lo lea y pueda servir de base para posteriores estudios en redes neuronales y el aprendizaje profundo.

# Anexos

## Notación para redes neuronales y el aprendizaje profundo

En este anexo se recopila la notación, principales fórmulas y representaciones que aparecen en este trabajo. Sería útil para el lector tenerlo delante al leer el trabajo, especialmente donde se utilicen o expliquen fórmulas.

### 1. Notación para redes neuronales

Comentarios generales:

- El superíndice *(i)* denota el ejemplo de entrenamiento i-ésimo.
- El superíndice *[l]* denota la capa l-ésima de la red neuronal.
- El subíndice *k* denota la fila k-ésima en un vector o matriz. También puede indicar el número de la unidad oculta ("neurona") o característica del ejemplo de entrenamiento.
- El subíndice *j* denota la columna j-ésima. También puede indicar el número de la unidad oculta o "neurona".

Tamaños:

- $m$ : número de ejemplos que componen el conjunto de datos. Será la dimensión de las columnas en las matrices.
- $n_x$ : tamaño de los datos de entrada. Equivale al número de características de cada ejemplo de entrenamiento.
- $n_y$ : tamaño del vector de salida de la red. Equivale a las etiquetas o clases correctas del conjunto de entrenamiento.
- $n_h^{[l]}$ : número de unidades ocultas ("neuronas") de la capa l-ésima. El subíndice "h" es la inicial del término "*hidden*", traducido "oculto/a/s". En un bucle se puede escribir la siguiente equivalencia: $n_x = n_h^{[0]}$ y $n_y = n_h^{[L]}$ *o* $n_h^{[\text{nº de capas ocultas}+1]}$
- $L$ : número de capas en la red neuronal.

Objetos:

Nota aclaratoria: si un vector tiene una única dimensión, es la dimensión de las filas, indicando que es un vector columna; $[l\ -\ 1]$ indica la capa anterior; matrices y vectores suelen ir representados en mayúsculas.

- $x$ *o* $X \in \mathbb{R}^{n_x \times m}$ es la matriz de entrada.
- $x^{(i)}$ o $X^{(i)} \in \mathbb{R}^{n_x}$ es el ejemplo de entrenamiento i-ésimo.
- $x_k^{(i)} \in \mathbb{R}$ es el elemento de la fila k-ésima del vector de entrenamiento i-ésimo.
- $y$ *o* $Y \in \mathbb{R}^{n_y \times m}$ es la matriz de las etiquetas o clases correspondientes a los ejemplos de entrenamiento, $X$.
- $y^{(i)}$ o $Y^{(i)} \in \mathbb{R}^{n_y}$ es el vector columna de etiquetas del vector de entrenamiento i-ésimo.
- $y_k^{(i)} \in \mathbb{R}$ es el elemento de la fila k-ésima del vector de etiquetas i-ésimo.
- $W^{[l]} \in \mathbb{R}^{n_h^{[l]} \times n_h^{[l-1]}}$ es la matriz de pesos de la capa l-ésima, formada por los vectores de pesos traspuestos, $W_j^{[l]}$, de cada unidad oculta en la capa *l*.

- $w_j^{[l]}$ o $W_j^{[l]} \in \mathbb{R}^{n_h^{[l-1]}}$: denota el vector columna de pesos correspondiente a la unidad oculta j-ésima de la capa l-ésima.
- $w_{kj}^{[l]} \in \mathbb{R}$ es el peso correspondiente a la fila k-ésima de la columna j-ésima de la capa l-ésima. Es cada uno de los pesos individuales de una unidad oculta.
- $b^{[l]} \in \mathbb{R}^{n_h^{[l]}}$ es el vector columna de sesgos de la capa l-ésima.
- $b_k^{[l]} \in \mathbb{R}$ es el sesgo correspondiente a la fila k-ésima de la capa l-ésima.
- $a^{[l]}$ *o* $A^{[l]} \in \mathbb{R}^{n_h^{[l]} \times m}$ es la matriz de activación/salida de la capa l-ésima.
- $a^{[l](i)}$ *o* $A^{[l](i)} \in \mathbb{R}^{n_h^{[l]}}$ es el vector de activación/salida de la capa l-ésima del vector de entrenamiento i-ésimo.
- $a_k^{[l](i)} \in \mathbb{R}$ es el valor de activación correspondiente a la fila (unidad oculta) k-ésima de la capa l-ésima del vector de entrenamiento i-ésimo.
- $\hat{y}$ o $\hat{Y} \in \mathbb{R}^{n_y \times m}$ es la matriz de predicciones del conjunto de entrenamiento $X$. Es equivalente a $A^{[L]}$.
- $\hat{y}^{(i)} \in \mathbb{R}^{n_y}$ es el vector de salida predicho del vector de entrenamiento i-ésimo. Es equivalente a $a^{[L](i)}$.

Propagación hacia delante (*forward propagation*):

- Fórmula general de activación lineal del vector de entrenamiento i-ésimo:

$$Z^{[l](i)} = W^{[l]} A^{[l-1](i)} + b^{[l]}$$

- Fórmula general de activación del vector de entrenamiento i-ésimo:

$$A^{[l](i)} = g^{[l]}\left(Z^{[l](i)}\right), 1 \leq l \leq L$$

  donde $g^{[l]}$ es la función de activación no lineal (*sigmoide, tanh, ReLU*, etc) de la capa l-ésima.

- Fórmula general de activación del vector de entrenamiento i-ésimo (elemento a elemento):

$$a_k^{[l](i)} = g^{[l]}\left(\left(\sum_j^{n_h^{[l-1]}} w_{kj}^{[l]} a_k^{[l-1](i)}\right) + b_k^{[l]}\right) = g^{[l]}\left(z_k^{[l](i)}\right)$$

Función de coste:

- $J(X, W, b, y)$ o $J(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)$ denota la función de coste.
- Función de coste utilizada en este trabajo:

$$J(\hat{y}, y) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)\right)$$

Retropropagación (*backward propagation*):

- Fórmula general de la derivada de la activación lineal en cada capa:

$$dZ^{[l]} = \frac{\partial J(W, b)}{\partial Z^{[l]}} = dA^{[l]} * g^{[l]\prime}\left(Z^{[l]}\right)$$

  donde "*" es multiplicación elemento a elemento.

- Fórmula general de la derivada de la función de activación en cada capa:

$$dA^{[l-1]} = \frac{\partial J(W,b)}{\partial A^{[l-1]}} = W^{[l]^T} dZ^{[l]}$$

Usando esta fórmula, podemos reescribir la derivada de la activación lineal como:

$$dZ^{[l]} = W^{[l+1]^T} dZ^{[l+1]} * g^{[l]\prime}\left(Z^{[l]}\right)$$

- Fórmulas generales de las derivadas de los parámetros de la red neuronal en cada capa:

$$dW^{[l]} = \frac{\partial J(W,b)}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]^T}$$

$$db^{[l]} = \frac{\partial J(W,b)}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

Algoritmo descenso del gradiente:

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

## 2. Representaciones en el aprendizaje profundo

Para las representaciones:

- los nodos representan entradas, activaciones o salidas.
- las aristas o líneas representan pesos o sesgos.

He aquí dos ejemplos de posibles representaciones del aprendizaje profundo general. El número de neuronas y capas se puede extender *ad infinitum*. Ambas representaciones son equivalentes.

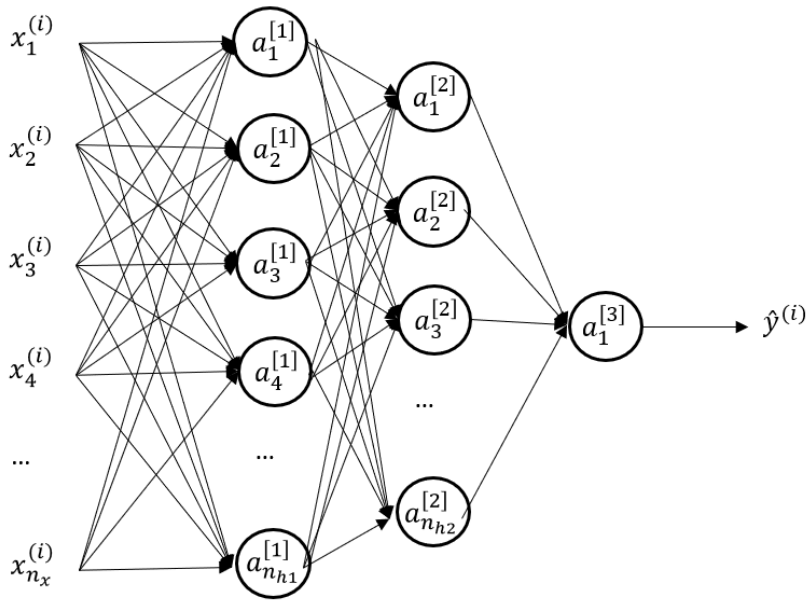


*Figura 40.* Red completa: representación comúnmente utilizada para redes neuronales. Ejemplo de una red neuronal de tres capas. Por una mejor estética, se omiten los detalles sobre los parámetros que deberían aparecer en las aristas ($w_{kj}^{[l](i)}, b_k^{[l](i)}$).

También se han omitido los superíndices *(i)* en las funciones de activación. Fuente: elaboración propia.

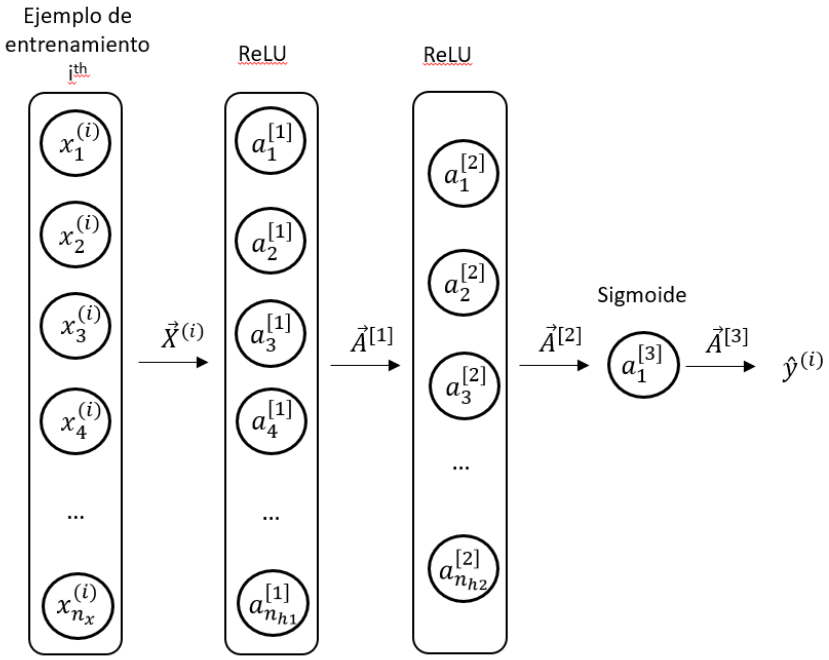


*Figura 41.* Red simplificada: representación donde se omiten las aristas. Ejemplo de una red neuronal de tres capas, equivalente a la figura 1. Se han especificado en esta representación las funciones de activación, pero únicamente como ejemplo. Fuente: elaboración propia.

## Bibliotecas de cálculo: NumPy vs. SciPy

A la hora de crear el código de las redes neuronales de este trabajo se ha optado por la biblioteca de cálculo NumPy, no utilizando en ningún momento una biblioteca que podría haberse empleado también, como es SciPy. ¿Cuáles son las diferencias que han decantado la balanza por NumPy?

*NumPy*

NumPy es la abreviatura de *Numerical Python*. Es una biblioteca escrita en Python, C y Fortran para funciones matemáticas de alto nivel. Proporciona un objeto array multidimensional de alto rendimiento, y herramientas para trabajar con estos arrays, lo que permite evitar el problema de lentitud de ejecución de los algoritmos. Cualquier algoritmo puede expresarse como una función sobre matrices, lo que permite ejecutarlos rápidamente. Razón clave por la que se ha utilizado en este trabajo, por la vectorización para ejecuciones más rápido del código.

*SciPy*

SciPy es la abreviatura de *Scientific Python*. Es una biblioteca escrita en Python, C, Fortran, C++ y Cython para el análisis numérico. Contiene una gran variedad de subpaquetes y dispone de una colección de funciones científicas, como agrupación, tratamiento de imágenes, integración, diferenciación, optimización de gradientes, etc.

1. Tipo de operaciones: NumPy se utiliza sobre todo cuando se trabaja con conceptos de ciencia de datos y estadística. Scipy se utiliza para operaciones complejas como funciones algebraicas, diversos algoritmos numéricos, etc.
2. *Arrays o matrices*: Las *arrays* NumPy son matrices multidimensionales de objetos que son del mismo tipo, es decir, homogéneos. SciPy no tiene estos conceptos de array ya que es más funcional y no tiene restricciones de homogeneidad.
3. Lenguaje y velocidad: NumPy está principalmente escrito en *C* y, gracias a su estructura de *arrays*, tiene un tiempo de ejecución más rápido que SciPy.

Queda claro que se utiliza en este trabajo NumPy por la velocidad de computación de sus operaciones, aunque también por su simplicidad, puesto que los algoritmos se han creado manualmente y, como se comenta en el anexo sobre las bibliotecas de aprendizaje profundo, en caso de querer implementar algoritmos y funcionalidades complejas, se pasaría a utilizar una de ellas, no SciPy.

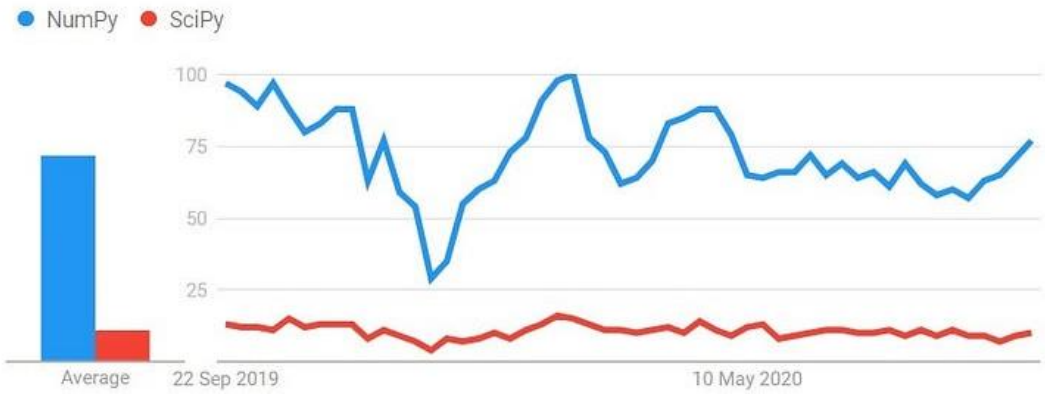


*Figura 42.* Interés a lo largo del tiempo en NumPy y SciPy. Vemos que la comunidad prefiere NumPy (combinado con otras librerías), mientras que SciPy ha quedado relegado a un pequeño, pero constante, nicho de usos.

## Bibliotecas y entornos para el aprendizaje profundo

En proyectos reales no se suele crear el código que hace funcionar a las redes neuronales desde cero, pues no sería práctico. Es importante entender las bases de lo que se está utilizando y los algoritmos del aprendizaje profundo, pero crear desde cero grandes modelos, redes convolucionales o recurrentes, etc, es muy costoso para cualquier empresa y equipo de trabajo. Afortunadamente, existen muchas bibliotecas y entornos (*frameworks*) de aprendizaje profundo que ayudan a implementar estos modelos. Algunos de ellos son: TensorFlow, PaddlePaddle, PyTorch, Caffe y Keras. Todos ellos son utilizados por desarrolladores de aprendizaje profundo en proyectos reales, acelerando significativamente el desarrollo de aplicaciones. No solo reducen en gran medida el tiempo dedicado a la creación de código, sino que incluyen optimizaciones que aceleran la ejecución del código, al igual que con la vectorización la biblioteca NumPy permite que el código de este trabajo se ejecute más rápido. Analicemos los tres principales *frameworks* por uso: Keras, TensorFlow y PyTorch.

### *Keras*

Es una eficaz interfaz de programación de aplicaciones (API) de redes neuronales de alto nivel escrita en Python. Este *framework* de redes neuronales de código abierto está diseñada para proporcionar una rápida experimentación con redes neuronales profundas y puede ejecutarse sobre CNTK, TensorFlow y Theano.

Keras se centra en ser modular, fácil de usar y extensible. No se encarga de los cálculos de bajo nivel, sino que los transfiere a otra biblioteca llamada *Backend*. Fue adoptada e integrada en TensorFlow a mediados de 2017. Los usuarios pueden acceder a ella a través del módulo *tf.keras*. Sin embargo, Keras todavía puede funcionar por separado y de forma independiente.

### *Tensorflow*

Es un *framework* de aprendizaje profundo integral de código abierto desarrollado por Google y publicada en 2015. Es conocido por su soporte de documentación y formación, sus opciones de producción y despliegue escalables, sus múltiples niveles de abstracción y su compatibilidad con diferentes plataformas, como Android.

TensorFlow es un entorno de matemática simbólica utilizado para redes neuronales y es el más adecuado para la programación de flujo de datos en una amplia gama de tareas. Ofrece múltiples niveles de abstracción para construir y entrenar modelos. Además, como ya se ha mencionado, TensorFlow ha adoptado Keras, lo que hace que la comparación entre ambos parezca problemática. No obstante, se contrastarán los dos *frameworks* para que la comparación sea completa, sobre todo porque los usuarios de Keras no tienen por qué utilizar TensorFlow.

### *PyTorch*

PyTorch es un entorno de aprendizaje profundo relativamente nuevo basado en Torch. Desarrollado por el grupo de investigación de IA de Facebook y de código abierto en GitHub en 2017, se utiliza para aplicaciones de procesamiento del lenguaje natural. PyTorch tiene una reputación de simplicidad, facilidad de uso, flexibilidad, uso eficiente de la memoria y gráficos computacionales dinámicos. También se siente que su lenguaje es nativo, lo que hace que la codificación sea más manejable y aumenta la velocidad de procesamiento.

### *PyTorch vs TensorFlow*

Tanto TensorFlow como PyTorch ofrecen abstracciones útiles que facilitan el desarrollo de modelos reduciendo el código repetitivo. Se diferencian en que PyTorch está enfocado en Python y orientado a objetos, mientras que TensorFlow ofrece una gran variedad de opciones, como es compatibilidad con C++ y CUDA. PyTorch se utiliza actualmente en muchos proyectos de aprendizaje profundo, y su popularidad está aumentando entre los investigadores de IA, aunque de las tres bibliotecas principales, es la menos popular.

Las tendencias muestran que esto podría cambiar pronto. Cuando los investigadores necesitan flexibilidad, capacidad de depuración y una duración de entrenamiento corta, eligen PyTorch. Funciona en Linux, macOS y Windows.

Gracias a su biblioteca bien documentada y a la abundancia de modelos entrenados y tutoriales, TensorFlow es la herramienta favorita de muchos profesionales de la industria e investigadores. TensorFlow ofrece una mejor visualización, lo que permite a los desarrolladores depurar mejor y hacer un seguimiento del proceso de entrenamiento. PyTorch, sin embargo, sólo proporciona una visualización limitada. TensorFlow también supera a PyTorch en el despliegue de modelos entrenados en producción, gracias a *TensorFlow Serving*. PyTorch no ofrece dicho servicio, por lo que los desarrolladores tienen que utilizar Django o Flask como servidor back-end.

En el área del paralelismo de datos, PyTorch obtiene un rendimiento óptimo al confiar en el soporte nativo para la ejecución asíncrona a través de Python. Sin embargo, con TensorFlow, se debe codificar y optimizar manualmente cada operación ejecutada en un dispositivo específico para permitir el entrenamiento distribuido. En resumen, se puede replicar todo lo realizado en PyTorch en TensorFlow, realizando un trabajo extra.

Se puede decir que, si se es novato en el aprendizaje profundo o se quiere participar en investigaciones o I+D+i, aprender PyTorch primero es buena opción debido a su popularidad en la comunidad investigadora. Sin embargo, si se está familiarizado con el aprendizaje automático y el aprendizaje profundo, y el objetivo es conseguir un trabajo en la industria lo antes posible, quizá el enfoque debería ser primero TensorFlow.

### *PyTorch vs Keras*

Ambas opciones son buenas si se es novato en el aprendizaje profundo. Los matemáticos y los investigadores experimentados preferirán probablemente PyTorch. Keras es más adecuado para los desarrolladores que quieren una biblioteca *plug-and-play* que les permita construir, entrenar y evaluar sus modelos rápidamente. Keras también ofrece más opciones de despliegue y una exportación de modelos más sencilla. Sin embargo, PyTorch es más rápido que Keras y tiene mejores capacidades de depuración.

Ambas plataformas gozan de suficiente popularidad como para ofrecer abundantes recursos de aprendizaje. Keras tiene un excelente acceso a código reutilizable y tutoriales, mientras que PyTorch cuenta con un extraordinario apoyo de la comunidad y un desarrollo activo. Keras es el mejor cuando se trabaja con conjuntos de datos pequeños, prototipado rápido y soporte back-end múltiple. Es el framework más popular gracias a su simplicidad comparativa. Funciona en Linux, MacOS y Windows.

### *TensorFlow vs Keras*

TensorFlow es una plataforma integral de código abierto, una biblioteca para múltiples tareas de aprendizaje automático, mientras que Keras es una biblioteca de redes neuronales de alto nivel que se ejecuta sobre TensorFlow. Ambas proporcionan una API de alto nivel que se utiliza para construir y entrenar modelos fácilmente, pero Keras es más fácil de usar porque está integrado en Python. Los investigadores recurren a TensorFlow cuando trabajan con grandes conjuntos de datos y detección de objetos y necesitan una funcionalidad excelente y un alto rendimiento. TensorFlow funciona en Linux, MacOS, Windows y Android.

Comparar TensorFlow y Keras puede no ser la mejor manera de enfocar la cuestión, ya que Keras funciona como una envoltura del framework de TensorFlow. Así, se puede definir un modelo con la interfaz de Keras, que es más fácil de usar, y luego utilizar TensorFlow cuando se necesite usar una característica que Keras no tiene, o se esté buscando una funcionalidad específica de TensorFlow. De este modo, se puede colocar el código de TensorFlow directamente en el pipeline de entrenamiento o modelo de Keras. En resumen, son complementarios y lo ideal sería saber manejar ambas bibliotecas.

## ¿Cuál es mejor?

Cada proyecto tiene unas necesidades diferentes, por lo que la decisión se reduce a qué características son más importantes. En la siguiente tabla se resumen las principales características de cada uno:

| | Keras | PyTorch | TensorFlow |
|---|---|---|---|
| Nivel de API | Alto | Bajo | Alto |
| Arquitectura | Simple, concisa, legible | Compleja, menos legible | No es fácil de usar |
| Tratamiento de datos | Conjuntos pequeños | Conjuntos grandes, alto rendimiento | Conjuntos grandes, alto rendimiento |
| Depuración del código | Redes sencillas, la depuración no suele ser necesaria | Buenas capacidades de depuración | Difícil realizar depuraciones del código |
| ¿Tiene modelos pre-entrenados? | Sí | Sí | Sí |
| Popularidad | La más popular | Tercer puesto | Segundo puesto |
| Velocidad de ejecución | Lenta, bajo rendimiento | Rápida, alto rendimiento | Rápida, alto rendimiento |
| Escrita en | Python | Lua | C++, CUDA, Python |

*Tabla 4*. Principales características de las bibliotecas de aprendizaje profundo Keras, PyTorch y Tensorflow.

Siguiendo la máxima "el conocimiento no ocupa lugar", un investigador o desarrollador de aprendizaje profundo tratará de aprender a utilizar tantos *frameworks* como sea posible. En otras palabras, el debate Keras vs. PyTorch vs. TensorFlow debería animar a conocer los tres, en qué coinciden y en qué se diferencian. Pero como hay que empezar por uno de ellos, esta descripción puede ayudar a elegir con cual iniciarse.

*¿Qué criterio podemos seguir si formamos parte de un equipo de trabajo en una empresa a la hora de decidir qué biblioteca de aprendizaje profundo utilizar?*

Es importante no adherirse a una única biblioteca o *framework*, pues este puede quedarse desfasado o ser superado en rendimiento por otros. He aquí una serie de criterios generales a la hora de elegir cuál utilizar:

- Facilidad de uso y programación, tanto en el desarrollo de la red neuronal, sus iteraciones y su despliegue en producción para el uso real.
- Velocidad de ejecución, especialmente en el entrenamiento de grandes conjuntos de datos.
- Biblioteca de código abierto (*open source*).

Sobre el último punto, desafortunadamente en la industria del software algunas empresas tienen un historial de código abierto sobre el que han mantenido un control único y, pasados unos años, tras ganar popularidad y usuarios, comienzan a restringir el uso u obligan a pagar.

### *Comparación de rendimiento entre Python puro, NumPy y TensorFlow*

¿Cómo se comparan estos esquemas? ¿Cuánto más rápido se ejecuta la aplicación cuando se implementa con NumPy en lugar de Python puro? ¿Y TensorFlow? Podemos ver en la **figura 43** el resultado al realizar una comparación del rendimiento de una implementación en Python puro, NumPy y TensorFlow de un algoritmo iterativo simple para estimar los coeficientes de un problema de regresión lineal, los resultados fueron:

| Implementation | Elapsed Time |
|---|---|
| Pure Python with list comprehensions | 18.65s |
| NumPy | 0.32s |
| TensorFlow on CPU | 1.20s |

*Figura 43.* Comparación de tiempos de ejecución de un algoritmo sencillo con diferentes implementaciones para comparar tiempos de ejecución.

Mientras que las soluciones NumPy y TensorFlow son competitivas (en CPU), la implementación pura de Python ocupa un distante tercer lugar. Aunque Python es un robusto lenguaje de programación de propósito general, sus bibliotecas orientadas al cálculo numérico le ganan la partida cuando se trata de grandes operaciones por lotes sobre matrices. Pese a que el ejemplo de NumPy resultó ser más rápido que TensorFlow en este caso, es importante señalar que TensorFlow realmente brilla para casos más complejos. Con el sencillo problema analizado, el uso de TensorFlow podría decirse que equivale a "usar un mazo para romper una nuez", como dice el refrán. Con TensorFlow, es posible construir y entrenar redes neuronales complejas en cientos o miles de servidores multi-GPU; con NumPy, no.

## Broadcasting con Python

*Broadcasting* ("propagación" en español) es otra técnica muy utilizada para hacer que el código en Python se ejecute más rápidamente. Es una estrategia o método de planificación de la biblioteca *NumPy* de Python, para operar y hacer más eficientes los cálculos con matrices multidimensionales (*arrays*). El *broadcasting* permite aplicar la vectorización a *arrays* de diferente tamaño. Sujeto a ciertas restricciones, la matriz más pequeña se "propaga" a través de la matriz más grande para que tengan formas compatibles. El *broadcasting* proporciona un medio para vectorizar operaciones de matriz para que el bucle ocurra en C en lugar de en Python, el cual es más rápido. En otras palabras, el *broadcasting* es una poderosa herramienta para escribir un código corto y normalmente intuitivo que realiza sus cálculos de forma muy eficiente.

Ilustremos esta técnica con un ejemplo. En la siguiente tabla se muestra un ejemplo de ciertos valores nutricionales en alimentos por cada 100g:

| | Manzanas | Carne de vaca | Huevos | Patatas |
|---|---|---|---|---|
| Carbohidratos | 52.0 | 0.3 | 3.8 | 62.0 |
| Proteínas | 1.5 | 101.0 | 48.0 | 7.0 |
| Grasas | 1.6 | 127.0 | 97.0 | 0.8 |

*Tabla 5*. Valores nutricionales de ciertos alimentos por cada 100g.

Esta tabla es una matriz de tres filas por cuatro columnas, la cual denominaremos *A*. Se quiere calcular el porcentaje de calorías (carbohidratos, proteínas y grasas) de cada alimento por cada 100g. Para ello, habrá que sumar los elementos de cada columna y dividirlos entre 100. ¿Se puede hacer sin utilizar explícitamente un bucle *for*? Sí, con dos simples líneas de código: una para sumar todos los elementos por columna y una segunda para dividir por columna. Veamos el código:

```
In [1]: import numpy as np

        A = np.array([[52.0,0.3,3.8,62.0],
                      [1.5,101.0,48.0,7.0],
                      [1.6,127.0,97.0,0.8]])
        print(A)
        print(A.shape)

[[ 52.    0.3   3.8  62. ]
 [  1.5 101.   48.    7. ]
 [  1.6 127.   97.    0.8]]
(3, 4)

In [2]: calorias = A.sum(axis=0)
        # axis=0 suma todos los elementos por columnas; axis
        print(calorias)
        print(calorias.shape)

[ 55.1 228.3 148.8  69.8]
(4,)

In [3]: porcentaje = 100*A/calorias.reshape(1,4)
        print(porcentaje)
        print(porcentaje.shape)

[[94.3738657   0.13140604  2.55376344 88.8252149 ]
 [ 2.72232305 44.24003504 32.25806452 10.0286533 ]
 [ 2.90381125 55.62855891 65.18817204  1.14613181]]
(3, 4)
```

*Figura 44.* Operaciones con matrices de NumPy sin hacer uso del bucle *for*.

El método *.reshape()* es algo redundante aquí pero, ante la duda, no cuesta nada utilizarlo ya que su tiempo de ejecución es *O(1)*. Lo que vemos a continuación es que hemos dividido una matriz 3x4 entre otra 1x4. ¿Cómo es esto posible? Lo que ha sucedido es que *NumPy* ha "estirado" la dimensión diferente de la matriz más pequeña, es decir, que internamente para realizar el cálculo, ha convertido la matriz 1x4 en una matriz 3x4, replicando la fila original dos veces. ¿Qué condiciones deben cumplir las matrices para que funcione el *broadcasting*?

### *Regla general del broadcasting*

Al operar con dos matrices, *NumPy* compara sus dimensiones. Comienza comparando la dimensión final (la que está más a la derecha) y continua la comparación hacia la izquierda. Dos dimensiones son compatibles si:

a) Son iguales.
b) Una de ellas es 1.

Si no se cumple alguna de estas dos condiciones, se lanza una excepción *ValueError*. Las matrices de entrada no necesitan tener el mismo número de dimensiones. La matriz resultante tendrá el mismo número de dimensiones que la matriz de entrada con la mayor cantidad de dimensiones, donde el tamaño de cada dimensión es el tamaño más grande de la dimensión correspondiente entre las matrices de entrada. Se puede resumir estas reglas de la siguiente manera:

$$matriz\ (n,m) \rightarrow \begin{matrix} + \\ - \\ / \\ * \end{matrix} \rightarrow \begin{matrix} (1,m) \rightarrow (n,m) \\ (n,1) \rightarrow (n,m) \end{matrix}$$

Veamos algún ejemplo más con las siguientes figuras:

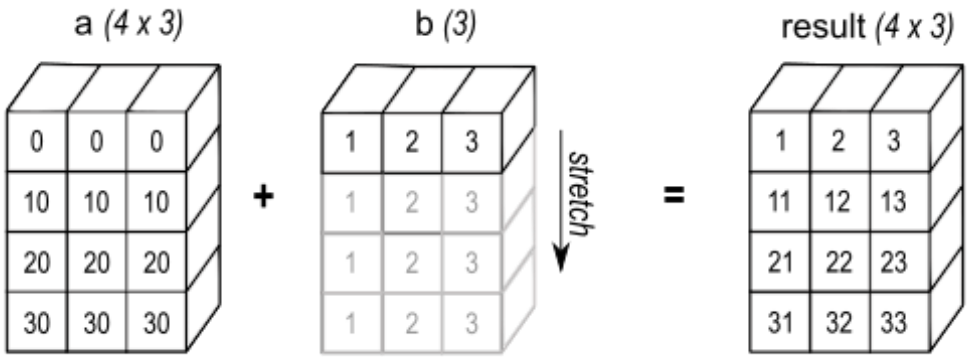


*Figura 45.* La suma de una matriz unidimensional con una matriz bidimensional sucede si el número de elementos de matriz de 1D coincide con el número de columnas de matriz de 2D.

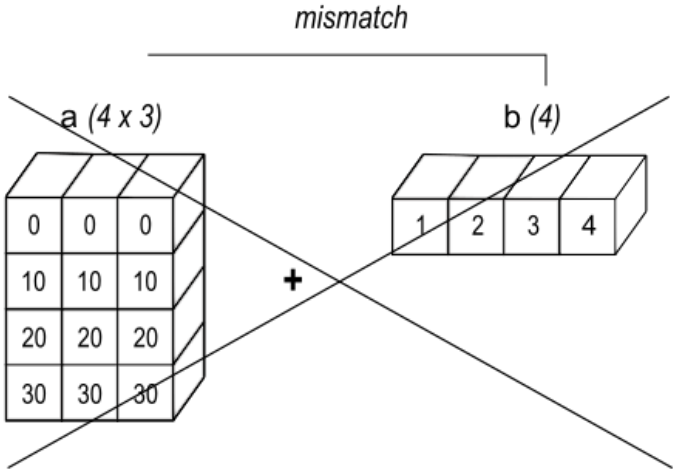


*Figura 46.* Cuando las dimensiones finales de las matrices son distintas, la propagación falla porque es imposible alinear los valores de las filas de la primera matriz con los elementos de la segunda para realizar la suma elemento a elemento.

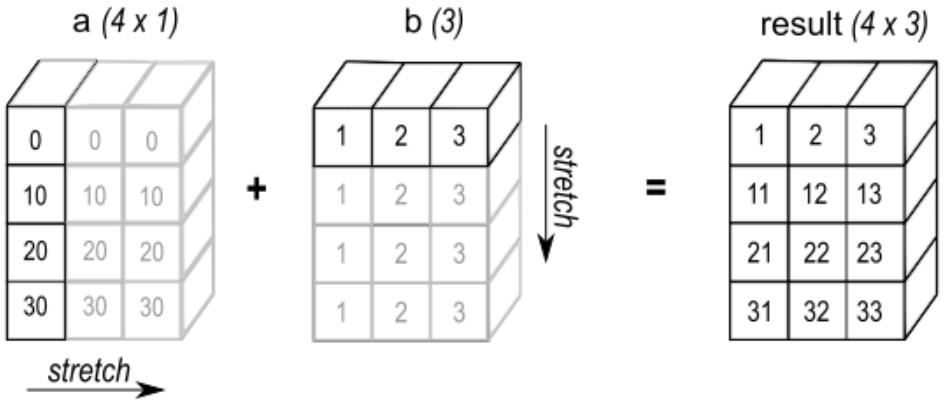


*Figura 47.* En algunos casos, la propagación amplía ambas matrices para formar una matriz de salida mayor que las iniciales. Esto solo sucede si tienen diferentes dimensiones unidimensionales.

Se pueden ver más ejemplos en el código de los anexos y a lo largo de los distintos archivos *.ipynb* de redes neuronales.

## Función de coste: Pérdida de entropía cruzada binaria

En el apartado donde se introducía la regresión logística, se escribió una función de pérdida, $L\left(\hat{y}^{(i)}, y^{(i)}\right)$, mencionando que no es la única posible que cumple los requisitos necesarios. En este anexo se pretende dar una justificación a porqué se utiliza esta y no otra. Definimos la regresión logística como:

$$\hat{y}^{(i)} = \sigma\left(W^T x^{(i)} + b\right) = \sigma\left(z^{(i)}\right) = \frac{1}{1 + e^{-z^{(i)}}};\ \ 0 < \hat{y}^{(i)} < 1$$

donde queremos interpretar la predicción $\hat{y}^{(i)}$ como $\hat{y}^{(i)} = P(y^{(i)} = 1 \mid x^{(i)})$, siendo $y^{(i)}$ la etiqueta de entrenamiento. Es decir, queremos que la salida $\hat{y}$ nos diga cuál es la probabilidad de que $y^{(i)} = 1$ dado un valor de $x^{(i)}$. Otra forma de escribir esta igualdad es definiendo $P(y^{(i)} \mid x^{(i)})$, para los dos casos posibles, 0 y 1 (recordemos que estamos en un caso de clasificación binaria):

- Si $y^{(i)}$= 1 → $P\left(y^{(i)} \mid x^{(i)}\right) = \hat{y}^{(i)}$
- Si $y^{(i)}$= 0 → $P\left(y^{(i)} \mid x^{(i)}\right) = 1 - \hat{y}^{(i)}$

El primer punto significa que si el valor de $y^{(i)}$ es igual a 1, entonces la probabilidad de $y^{(i)}$ dado $x^{(i)}$ es igual a $\hat{y}^{(i)}$. El segundo indica la inversa del primero, ya que $\hat{y}^{(i)} \in [0,1]$. ¿Qué ecuación $P(y^{(i)} \mid x^{(i)})$ engloba estos dos casos? Se pueden englobar como la función de probabilidad de una distribución de Bernoulli:

$$P(\mathrm{y} \mid x) = \ \hat{y}^{y}(1 - \hat{y})^{(1-y)} \quad y = 0, 1$$

He eliminado el superíndice *(i)* por limpieza en la explicación. Vemos que si sustituimos $y$ por 1 o 0, volvemos a las expresiones iniciales. En estadística hay un concepto, la *entropía de Shannon*, que mide la incertidumbre de una fuente de información. La entropía puede ser considerada como una medida de la incertidumbre y de la información necesaria para, en cualquier proceso, poder acotar, reducir o eliminar la incertidumbre. Es también la cantidad de «ruido» o «desorden» que contiene o libera un sistema.

Su definición matemática es bastante intuitiva: cada evento posible (obtener 1 o 0 en nuestro caso) tiene un grado de indeterminación inicial, *k*, donde todos los eventos son supuestamente equiprobables. La probabilidad de que se dé una de estas combinaciones es la inversa de esa incertidumbre, *p = 1/k*. Si tomamos su logaritmo neperiano, es fácil ver que *ln(k) = -ln(p).* Como cada estado tiene una probabilidad *p*, la entropía vendrá dada por la suma ponderada de los productos de la probabilidad por la incertidumbre:

$$H(\mathrm{y}) = -\sum_{y \in [0,1]} p(y) \log p(y)$$

El signo (-) indica que minimizar la función de pérdida $H(\mathrm{y})$ equivale a maximizar el logaritmo de la probabilidad. ¿Cuál es la entropía en un ensayo de Bernoulli? Sustituimos $p(y)$ por $P(\,y \mid x\,)$ para *y = 1* e *y = 0,* y tenemos la función de pérdida definida en la regresión logística:

$$H(\,\mathrm{y} \mid x\,) = L(\hat{y}, \mathrm{y}) = -\left(\mathrm{y}\log\left(\hat{y}\right) + (1 - \mathrm{y})\log\left(1 - \hat{y}\right)\right)$$

Es importante señalar que la elección de la base del logaritmo depende del contexto en el que se esté trabajando. Por ejemplo, en teoría de la información se utiliza el logaritmo en base dos para medir la entropía en bits. En estadística, el neperiano es buena elección. ¿Por qué se utiliza el logaritmo, sea o no neperiano? El logaritmo es una función estrictamente monótonamente creciente. Esto significa que si $x_1 < x_2$, entonces $ln(x_1) < ln(x_2)$. Por lo tanto, optimizar $P(\,y \mid x\,)$ es similar a optimizar $\log P(\,y \mid x\,)$ y así se pueden aprovechar las propiedades de los logaritmos.

Si queremos saber cuál es la entropía total de todos los casos de entrenamiento, no hay más que sumar sus entropías individuales, y el objetivo será reducir esta entropía total lo máximo posible:

$$H(\,\mathrm{y} \mid x\,)_{total} = \sum_{i=1}^{m} L(\hat{y}, \mathrm{y})$$

Finalmente, por conveniencia, para asegurar que la escala de los resultados sea mejor, dividimos entre el número de ejemplos, *m*, y obtenemos la función de coste descrita en la regresión logística.

$$J(W,b) = \frac{1}{m} H(\mathrm{y} \mid x)_{total} = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, \mathrm{y})$$

De acuerdo con el **método de máxima verosimilitud** de la estadística, escogeremos como valores estimados de los parámetros (*W* y *b* en nuestro caso) aquellos que maximizan la probabilidad de que ocurra lo observado (que y sea 0 o 1). Es decir:

$$W, b \rightarrow \max\left(P(\mathrm{y} \mid x)\right)$$

Como ya se ha explicado anteriormente que $P(\mathrm{y} \mid x)$ es similar a optimizar $\log P(\mathrm{y} \mid x)$, y resulta que para nuestra función de probabilidad su logaritmo es:

$$\log P(\mathrm{y} \mid x) = \mathrm{y} \log(\hat{y}) + (1 - \mathrm{y}) \log(1 - \hat{y})$$

Por lo tanto, maximizar $P(\mathrm{y} \mid x)$ es maximizar $\log P(\mathrm{y} \mid x)$. Y, reformulando la ecuación de entropía:

$$H(\mathrm{y} \mid x) = L(\hat{y}, \mathrm{y}) = -\log P(\mathrm{y} \mid x)$$

Entonces, minimizar la función de coste $J(w,b)$ equivaldrá a maximizar $\log P(\mathrm{y} \mid x)$, cumpliendo el método de máxima verosimilitud. Esto justifica la función de pérdida y coste elegidas para la regresión logística a la hora de estimar los parámetros *W* y *b* (bajo el supuesto de que nuestros ejemplos de entrenamiento se distribuyeron de manera idéntica e independiente).

# Créditos de las figuras

Todas las figuras que aparecen en el trabajo no mencionadas a continuación son de elaboración propia.

1. *Nerve Cell*. © <https://askabiologist.asu.edu/neuron-anatomy>
2. *Simplified neuron body within a network*. © *An Introduction to Neural Networks*. Gurney (1997).
7. *AlexNet Convolutional Neural Network Architecture for Object Classification © ImageNet Classification with Deep Convolutional Neural Networks*. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton (2012).
8. Red neuronal recurrente. © Redes neuronales recurrente con puertas. Gutiérrez Naranjo, Miguel A. (2018).
10. *Demystifying AI*. © *AI for Everyone.* Yan-Tak Ng, Andrew (Coursera).
11. *Auditory cortex learns to see*. © *Machine Learning*, Y. Ng, Andrew (Coursera).
12. *Exam results*. © *GPT-4*. OpenAI (2023). <https://openai.com/research/gpt-4>
15. *Sigmoid activation function*. © *Neural Networks, Machine Learning for Artists*. <https://ml4a.github.io/ml4a/neural_networks/>
16. *A visual interlude. © Understanding gradient descent*. Bendersky, Eli (2016). <https://eli.thegreenplace.net/2016/understanding-gradient-descent/>
17. *Optimization.* © <https://docs.scipy.org/doc/scipy/tutorial/optimize.html>
18. *Loss function*. © *What learning rate should I use?*. Interpretación de Benjamin D. Hammel en <http://www.bdhammel.com/learning-rates/> de *Convolutional Neural Networks for Visual Recognition* <https://cs231n.github.io/neural-networks-3/>.
19. *Computational Graphs in Deep Learning*. © <https://www.geeksforgeeks.org/computational-graphs-in-deep-learning/>.
24. *The hyperbolic sine (red), hyperbolic cosine (green) and hyperbolic tangent (blue) graphed on the same axes.* <https://en.wikipedia.org/wiki/Hyperbolic_functions>.
25. *ReLU activation function. © Neural Networks, Machine Learning for Artists*. <https://ml4a.github.io/ml4a/neural_networks/>
26. *Leaky ReLU. © Intro to Optimization in Deep Learning: Vanishing Gradients and Choosing the Right Activation Function*. Ayoosh Kathuria <https://blog.paperspace.com/vanishing-gradients-activation-function/>.
27. *Usage over time*. © *Leaky ReLU*. <https://paperswithcode.com/method/leaky-relu>.
28. *Summary Table*. © *Deep Learning: Which Loss and Activation Functions should I use?* Stacey Ronaghan. <https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8>.
29. *A diagram showing how the Perceptron works.* <https://www.skynettoday.com/overviews/neural-net-history>.
30. *A cartoon drawing of a biological neuron (left) and its mathematical model (right). © Convolutional Neural Networks for Visual Recognition* <https://cs231n.github.io/neural-networks-1/>.

31. *Deep learning training compared to inference. © Inference: The Next Step in GPU-Accelerated Deep Learning*, Michael Andersch. <https://developer.nvidia.com/blog/inference-next-step-gpu-accelerated-deep-learning/>.

35. *Layers learn from specific object categories. © Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations*. Honglak Lee, Roger Grosse, Rajesh Ranganath, Andrew Y. Ng. <http://robotics.stanford.edu/~ang/papers/icml09-ConvolutionalDeepBeliefNetworks.pdf>.

42. *Interest over time*. <https://medium.com/analytics-vidhya/numpy-vs-scipy-3c4dee3403db>.

43. *Elapsed time to run the algorithm. © Pure Python vs NumPy vs TensorFlow Performance Comparison*. Candido, Renato. <https://realpython.com/numpy-tensorflow-performance/>.

45. *Broadcasting.* © NumPy organization <https://numpy.org/doc/stable/user/basics.broadcasting.html>.

46. *Ibid.*

47. *Ibid.*

Tablas:

2. *Differences between ChatGPT and ChatGPT 4.* <https://www.greataiprompts.com/guide/difference-between-chatgpt3-and-chatgpt4/>.

4. *Keras vs PyTorch vs TensorFlow*. <https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article>.

# Referencias

Debido al alto número de fuentes consultadas y ser la mayoría extractos de cursos, investigaciones y blogs, se ha decidido seguir la siguiente organización. En primer lugar, se citará una bibliografía genérica de fuentes recurrentemente usadas en el trabajo. En segundo lugar, se hará un listado de otras fuentes consultadas o utilizadas puntualmente. Las fuentes señaladas sin autor con "—", son aquellas donde no se ha podido encontrar el nombre y apellidos del autor, pero cuyo trabajo está bien documentado.


Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2, 1--127. doi: 10.1561/2200000006. Se puede leer en: <http://www.iro.umontreal.ca/~lisa/pointeurs/TR1312.pdf>.

Bengio, Yoshua, *et al*. *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, 2014. <https://arxiv.org/abs/1406.2572v1>.

Canziani, Alfredo & LeCun, Yann. *Deep Learning*, NYU Center for Data Science. <https://atcold.github.io/pytorch-Deep-Learning/>.

Chen, Kai; Dean, Jeff *et al*. *Building high-level features using large scale unsupervised learning,* 2012. <https://arxiv.org/abs/1112.6209v5>.

Gajowniczek, K., Chmielewski, L.J., Orłowski, A., Ząbkowski, T. (2017). *Generalized Entropy Cost Function in Neural Networks*. In: Lintas, A., Rovetta, S., Verschure, P., Villa, A. (eds) Artificial Neural Networks and Machine Learning – ICANN 2017. ICANN 2017. Lecture Notes in Computer Science(), vol 10614. Springer, Cham. <https://doi.org/10.1007/978-3-319-68612-7_15>. Se puede leer en: <https://www.researchgate.net/publication/320590931_Generalized_Entropy_Cost_Function_in_Neural_Networks>.

Goh. *Why Momentum Really Works*, Distill, 2017. <http://doi.org/10.23915/distill.00006>.

Katanforoosh, Kunin et al. *Parameter optimization in neural networks*, deeplearning.ai, 2019. <https://www.deeplearning.ai/ai-notes/optimization/index.html>.

Katanforoosh, Kunin. *Initializing neural networks*, deeplearning.ai, 2019. <https://www.deeplearning.ai/ai-notes/initialization/index.html>.

Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey. *Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems* (p./pp. 1097--1105), 2012. Se puede ver en: <https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>.

Kurenkov, Andrey. *A Brief History of Neural Nets and Deep Learning*, 2020, <https://www.skynettoday.com/overviews/neural-net-history>.

Lalin, Jonas. *Feedforward Neural Networks in Depth, Part 1: Forward and Backward Propagations*, 2021. <https://jonaslalin.com/2021/12/10/feedforward-neural-networks-part-1/>.

Lalin, Jonas. *Feedforward Neural Networks in Depth, Part 2: Activation Functions*, 2021. <https://jonaslalin.com/2021/12/10/feedforward-neural-networks-part-2/>.

Lalin, Jonas. *Feedforward Neural Networks in Depth, Part 3: Cost Functions*, 2021. <https://jonaslalin.com/2021/12/10/feedforward-neural-networks-part-3/>.

Lalin, Jonas. *How Backpropagation Is Able To Reduce the Time Spent on Computing Gradients*, 2021. <https://jonaslalin.com/2021/10/12/forward-vs-reverse-accumulation-mode/>.

LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* 521, 436–444 (2015). <https://doi.org/10.1038/nature14539>. Se puede leer en: <http://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf>.

LeCun, Yan et al. *Efficient BackProp*, *Neural Networks: Tricks of the Trade, chapter 2, Springer Berlin / Heidelberg,* 1998. <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>.

Li, Fei-Fei; Li, Yunzhu; Gao, Ruohan. *Deep Learning for Computer Vision* Stanford University course, 2023. <http://cs231n.stanford.edu/>.

Minnar, Alex. *Deep Learning Basics: Neural Networks, Backpropagation and Stochastic Gradient Descent*. 2015. <http://alexminnaar.com/2015/02/14/deep-learning-basics.html>.

Montenegro Díaz, Álvaro M.; Montenegro Reyes, Daniel M. Fundamentos de IA y AP, 2022. <https://aprendizajeprofundo.github.io/Libro-Fundamentos/>.

Nielsen, Michael. *Neural Networks and Deep Learning*, 2019, <http://neuralnetworksanddeeplearning.com/>.

Olah, Christopher. *Calculus on Computational Graphs: Backpropagation*, 2015. <https://colah.github.io/posts/2015-08-Backprop/>.

Olah, Christopher. *Neural Networks, Types, and Functional Programming*, 2015. <https://colah.github.io/posts/2015-09-NN-Types-FP/>.

OpenAI, *GPT-4*, 2023. <https://openai.com/research/gpt-4>

Ruder, Sebastian. *An overview of gradient descent optimization algorithms*, 2017. <https://arxiv.org/abs/1609.04747v2>.

Schmidhuber, Jürgen. Deep Learning in Neural Networks: An Overview. *Neural Networks,* Volume 61, January 2015, Pages 85-117 (DOI: 10.1016/j.neunet.2014.09.003). <https://arxiv.org/pdf/1404.7828v4.pdf>.

Tang, An *et al*. *Canadian Association of Radiologists White Paper on Artificial Intelligence in Radiology*, Canadian Association of Radiologists Journal 69(2), 2018. doi: 10.1016/j.carj.2018.02.002. Se puede ver en: <https://www.researchgate.net/publication/324457640_Canadian_Association_of_Radiologists_White_Paper_on_Artificial_Intelligence_in_Radiology>.

Y. Bengio, L. Pascal, D. Popovici, and H. Larochelle, “Greedy layer-wise training of deep networks,” *Advances in Neural Information Processing Systems*, vol. 19, p. 153, 2007. Se puede ver en: <https://proceedings.neurips.cc/paper_files/paper/2006/file/5da713a690c067105aeb2fae32403405-Paper.pdf>.

Y. NG, Andrew. *Unsupervised Feature Learning and Deep Learning Tutorial*, 2013. <http://deeplearning.stanford.edu/tutorial/>.

—, *How neural networks are trained.* <https://ml4a.github.io/ml4a/how_neural_networks_are_trained/>.

—, *Looking inside neural nets.* <https://ml4a.github.io/ml4a/looking_inside_neural_nets/>.

—, *Neural networks.* <https://ml4a.github.io/ml4a/neural_networks/>.

*INTRODUCCIÓN A LAS REDES NEURONALES Y EL APRENDIZAJE PROFUNDO*


Marcus, Gary. *Is "Deep Learning" a Revolution in Artificial Intelligence?*, 2012. <https://www.newyorker.com/news/news-desk/is-deep-learning-a-revolution-in-artificial-intelligence>.

—, ¿Qué son las redes neuronales? *IBM*. <https://www.ibm.com/mx-es/topics/neural-networks>.

—, *ImageNet*. <https://en.wikipedia.org/wiki/ImageNet> y <https://www.image-net.org/>.


*CONCEPTOS BÁSICOS DE LAS REDES NEURONALES*


Bubeck, Sebastian *et al*. *Sparks of Artificial General Intelligence: Early experiments with GPT-4*, Cornell University, 2023. <https://arxiv.org/abs/2303.12712>.

Goertzel, Ben. *Artificial General Intelligence: Concept, State of the Art, and Future Prospects*, Journal of Artificial General Intelligence, 2014. doi:10.2478/jagi-2014-0001. <https://www.researchgate.net/publication/271390398_Artificial_General_Intelligence_Concept_State_of_the_Art_and_Future_Prospects>.

Kaercher, Silke *et al*. *Sensory Augmentation for the Blind*, Frontiers in Human Neuroscience 6:37, 2012. doi:10.3389/fnhum.2012.00037. <https://www.researchgate.net/publication/221689414_Sensory_Augmentation_for_the_Blind>.

Nielsen, Michael. *Chapter* 3*, The cross-entropy cost function*.

Nuñez, Michael. *Titans of AI Andrew Ng and Yann LeCun oppose call for pause on powerful AI systems*, VentureBeat, 2023. <https://venturebeat.com/ai/titans-of-ai-industry-andrew-ng-and-yann-lecun-oppose-call-for-pause-on-powerful-ai-systems/>.

OpenAI. *Introducing ChatGPT*, OpenAI authors, 2022. <https://openai.com/blog/chatgpt>.

Schulman, John *et al*. *Proximal Policy Optimization*, OpenAI, 2017. <https://openai.com/research/openai-baselines-ppo>.

Uszkoreit, Jakob. *Transformer: A Novel Neural Network Architecture for Language Understanding*, Google Research, 2017. <https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html>.

Y. Ng, Andrew. *Unsupervised Feature Learning and Deep Learning*. <https://redirect.cs.umbc.edu/courses/graduate/678/fall14/visionaudio.pdf>.

—, *What is ChatGPT (Everything You Need To Know)*. <https://www.greataiprompts.com/guide/how-does-chatgpt-works/#What_are_Prompts_in_ChatGPT>.

—, *AWS announces Amazon Bedrock and multiple generative AI services and capabilities*, Amazon, 2023. <https://www.aboutamazon.eu/news/aws/aws-announces-amazon-bedrock-and-multiple-generative-ai-services-and-capabilities>.

—, *Computational Graphs in Deep Learning*. <https://www.geeksforgeeks.org/computational-graphs-in-deep-learning/>.

—, *Introducing LLaMA: A foundational, 65-billion-parameter large language model*, Meta, 2023. <https://ai.facebook.com/blog/large-language-model-llama-meta-ai/>.

—, *LaMDA: our breakthrough conversation technology*, Google, 2021. <https://blog.google/technology/ai/lamda/>.

*REDES NEURONALES POCO PROFUNDAS*


Agrawal, Rochak. *Shallow Neural Networks*, 2019. <https://towardsdatascience.com/shallow-neural-networks-23594aa97a5>.

Durán, Jaime. *Everything You Need to Know about Gradient Descent Applied to Neural Networks*, Medium. <https://medium.com/yottabytes/everything-you-need-to-know-about-gradient-descent-applied-to-neural-networks-d70f85e0cc14>.

Hannun, Awni; Maas, Andrew; Y. Ng, Andrew. *Rectifier Nonlinearities Improve Neural Network Acoustic Models*, Computer Science Department, Stanford University, 2013. <https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf>.

Himanshu, S. *Activation Functions: Sigmoid, tanh, ReLU, Leaky ReLU, PReLU, ELU, Threshold ReLU and Softmax basics for Neural Networks and Deep Learning*, Medium, 2019. <https://himanshuxd.medium.com/activation-functions-sigmoid-relu-leaky-relu-and-softmax-basics-for-neural-networks-and-deep-8d9c70eed91e>.

Montenegro y Montenegro. "Funciones de activación", de Fundamentos de IA y AP <https://aprendizajeprofundo.github.io/Libro-Fundamentos/Redes_Neuronales/Cuadernos/Activation_Functions.html>

Pandey, Rahul. *How to deal with vanishing and exploding gradients*, 2022. <https://medium.com/geekculture/how-to-deal-with-vanishing-and-exploding-gradients-in-neural-networks-24eb00c80e84>

—, *Hyperbolic functions*, Wikipedia. <https://en.wikipedia.org/wiki/Hyperbolic_functions#Hyperbolic_tangent>.

—, *Rectifier (neural networks),* Wikipedia. <https://en.wikipedia.org/wiki/Rectifier_(neural_networks)>.


*REDES NEURONALES PROFUNDAS*


Agrawal, Rochak. *Deep Neural Networks*, 2019. <https://medium.com/analytics-vidhya/deep-neural-networks-d14051d7c4f3>.

Dellinger, James. *Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming*, 2019. <https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79>.

Glort, Xavier & Bengio, Yoshua. *Understanding the difficulty of training deep feedforward neural networks*, 2010. <https://www.semanticscholar.org/paper/Understanding-the-difficulty-of-training-deep-Glorot-Bengio/b71ac1e9fb49420d13e084ac67254a0bbd40f83f>.

Katanforoosh, Kunin.

Lee, Honglak; Grosse, Roger; Y. Ng., Andrew. *Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations*, 2009. In A. P. Danyluk, L. Bottou & M. L. Littman (eds.), ICML (p./pp. 77), ACM. ISBN: 978-1-60558-516-1. Se puede ver en: <http://robotics.stanford.edu/~ang/papers/icml09-ConvolutionalDeepBeliefNetworks.pdf>.

Y. Ng, Andrew. *Xavier Initialization and Regularization*, 2022 <https://cs230.stanford.edu/section/4/>.

—, *Vanishing gradient problem*, Wikipedia. <https://en.wikipedia.org/wiki/Vanishing_gradient_problem>.

Bhadra, Anushka. *Numpy vs SciPy*, 2020 <https://medium.com/analytics-vidhya/numpy-vs-scipy-3c4dee3403db>.

Bhagwat, Swapnil. *Tensorflow vs Keras vs Pytorch: Which Framework is the Best?* <https://atlassystems.com/blog/tensorflow-vs-keras-vs-pytorch-which-framework-is-the-best/>.

Candido, Renato. *Pure Python vs NumPy vs TensorFlow Performance Comparison*. <https://realpython.com/numpy-tensorflow-performance/>.

Rondón, Izary. *Pytorch vs Keras vs Tensorflow*, Escuela Internacional de Posgrados, 2022. <https://eiposgrados.com/blog-python/pytorch-keras-tensorflow/>.

Sharma, Sumit. *What is the difference between NumPy and SciPy in Python?* <https://www.educative.io/answers/what-is-the-difference-between-numpy-and-scipy-in-python>.

Terra, John. *Keras vs Tensorflow vs Pytorch: Key Differences Among Deep Learning* <https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article>.

—, *Broadcasting*, NumPy organization <https://numpy.org/doc/stable/user/basics.broadcasting.html>.

—, Distribución Bernoulli, Wikipedia. <https://es.wikipedia.org/wiki/Distribuci%C3%B3n_Bernoulli>.

—, *Entropy (information theory)*, Wikipedia. <https://en.wikipedia.org/wiki/Entropy_(information_theory)>.

—, *SciPy*, Wikipedia. <https://es.wikipedia.org/wiki/SciPy>.