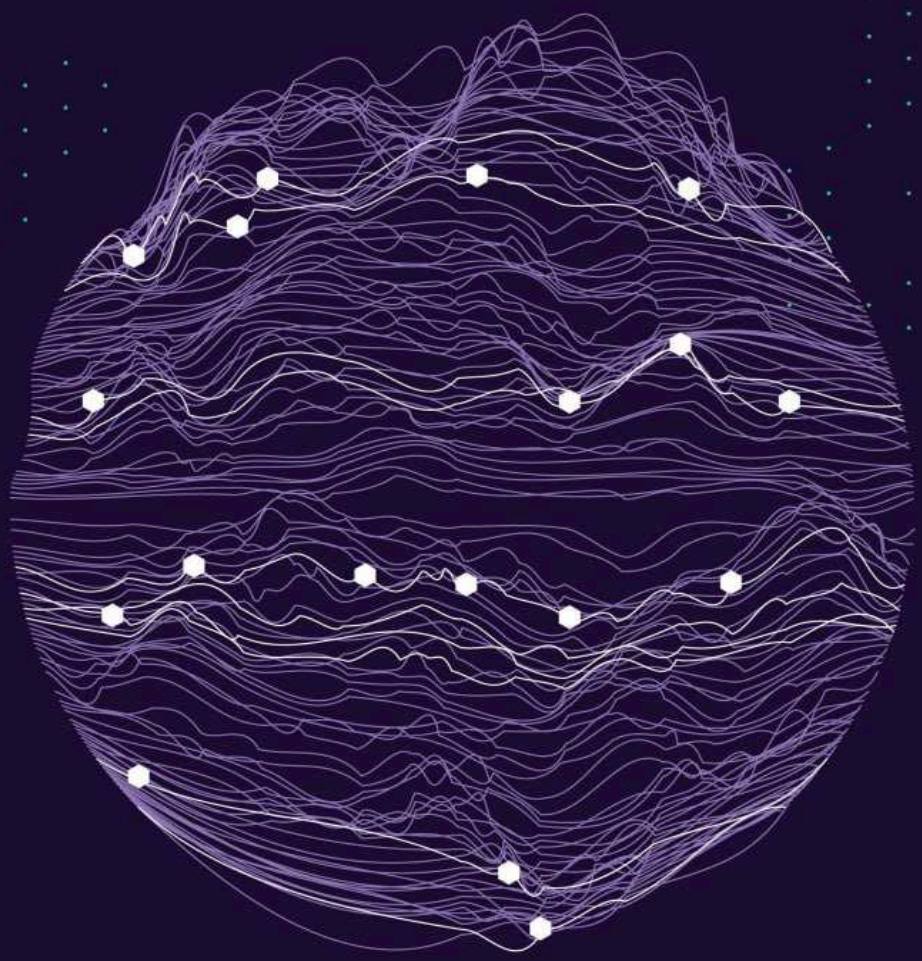
Introducción a la Inteligencia Artificial con Python

Juan Sensio



savvily

Introducción a la inteligencia artificial con Python

Introducción a la inteligencia artificial con Python

Edición 2025

Juan Sensio

Primera edición de esta colección: Mayo 2025
ISBN: 978-84-127192-8-4

Este libro se lo dedico a Suki

Cuando estés acorralado y no haya esperanza de un mañana, no desesperes.
Y recuerda, más grande será la caída.
Etcétera...
”Coronel Jack O’Neill”

Índice general

Preludio

¡Hola! Mi nombre es Juan Sensio, y en este libro aprenderás sobre Inteligencia Artificial. En concreto, aprenderás los fundamentos del Aprendizaje Profundo, o *Deep Learning* en inglés, la rama de la Inteligencia Artificial que se ocupa del estudio de las **Redes Neuronales** y que más impacto está teniendo en nuestra sociedad a día de hoy. Imagino que conoces ChatGPT, el asistente virtual de OpenAI del que tanto se está hablando en el momento de escribir estas líneas. ¿Adivinas qué tecnología hay detrás de este sistema de Inteligencia Artificial? Efectivamente, el Aprendizaje Profundo.

¿Por qué este libro? Pues la verdad es que hasta yo mismo me hago esa pregunta. Existen muchos libros y cursos disponibles *online* sobre el tema, muchos de ellos excelentes (y que te iré recomendando a su debido tiempo), por lo que no es sencillo encontrar un factor diferenciador. Es por este motivo que, tras darle muchas vueltas, he decido que voy a escribir un libro para mí. Y si de paso le sirve a alguien más, pues mejor que mejor. Así pues, en este libro he plasmado todo aquello que me hubiese gustado aprender en mi época de estudiante, así como todo lo que quiero tener a mano en mi etapa actual de desarrollador de *software* y emprendedor, todo en un mismo lugar.

Quizás vale la pena en este punto explicarte quién soy y por qué deberías confiar en que la información que encontrarás en este libro tiene mucho valor. Como ya he dicho antes, mi nombre es Juan Sensio (si bien no es exactamente el nombre que aparece en mi DNI, ese es el nombre con el que me identifico, mi AKA, mi nombre artístico). Al terminar el instituto comencé la carrera de Ingeniería Aeronáutica en la Universitat Politècnica de Catalunya (UPC), no por vocación sino porque de entre las opciones que estaban a mi alcance parecía la más *cool*. Terminé la carrera, sin pena ni gloria, tras lo cual empecé un doctorado (no me preguntes por qué, pero en aquel momento mis prioridades estaban alejadas del mundo laboral o académico). Sin embargo, fue durante esos años en los que descubrí una de mis grandes pasiones: la **programación**. En mi etapa de estudiante de doctorado aprendí a programar, usando el lenguaje de programación C++ (ya hablaremos más adelante sobre lenguajes de programación) desarrollando métodos numéricos para la simulación de fluidos compresibles turbulentos (y sí, es tan guay como suena). Ver cómo el código que escribía daba como resultado predicciones físicamente precisas y realistas del comportamiento del aire alrededor del ala de un avión es algo que a día de hoy me sigue fascinando. No obstante, a medida que pasaron los años, me di cuenta de que el mundo universitario

no era lo mío. Fue entonces cuando empecé a investigar cómo podía aplicar la programación a otros campos. De entre ellos, la **Inteligencia Artificial** (IA de ahora en adelante) captó mi atención al momento, y coincidió que por aquel entonces se estaba viviendo un resurgir tras varios ciclos de pesimismo (conocidos como los inviernos de la IA) gracias a los avances en Aprendizaje Profundo. Así pues, decidí hacer un máster en IA en la Universitat Politècnica de València (UPV), lugar en el que conocí al mejor profesor que he tenido, Roberto Paredes (un saludo), tras lo cual empecé a trabajar como desarrollador de *software*. Tras varios años de experiencia y después de pasar por varias empresas (unas más grandes, otras más pequeñas) me di cuenta de que lo de trabajar para otros no era del todo lo mío. Fue entonces cuando inicié mi etapa como emprendedor, en la cual me encuentro ahora mismo. Empecé creando un blog y canal de Youtube compartiendo tutoriales relacionados con la programación y la IA a la vez que co-fundé EarthPulse con mi socia Laura Moreno (otro saludo), *startup* centrada en la intersección entre la IA y la Observación de la Tierra.

Hechas las presentaciones, en este libro he intentado plasmar parte de lo que he aprendido durante mi camino, centrarme en los conceptos más fundamentales del Aprendizaje Profundo, de manera que alguien que se encuentre en una situación similar (ya sea en su etapa de estudiante, profesional o emprendedor) tenga a su alcance una fuente a la que recurrir para aprender, asentar o refrescar estos conocimientos. Con este objetivo, he organizado el contenido de la siguiente manera:

1. En el primer capítulo definiremos qué es la Inteligencia Artificial, y veremos cómo ha evolucionado a lo largo de los años hasta llegar al momento actual. Veremos los tipos de IA que existen, ejemplos de aplicaciones y, por último, entrenarás tu primera red neuronal para detectar dígitos manuscritos en imágenes de manera automática.
2. El segundo capítulo está enfocado en su totalidad al aprendizaje del lenguaje de programación `Python`, aunque también veremos algunos conceptos básicos de la computación. Saber programar en este lenguaje será vital para seguir avanzando en el libro.
3. En el tercer capítulo aprenderemos a usar `Pytorch`, una librería en `Python` para diseñar y entrenar redes neuronales. Por el camino también asentaremos conceptos fundamentales relacionados con el Aprendizaje Profundo: el álgebra lineal, el cálculo diferencial y la optimización.
4. En el cuarto capítulo nos adentraremos en las entrañas del Aprendizaje Profundo, aprendiendo los fundamentos de las redes neuronales en los que librerías como `Pytorch` basan su funcionamiento, por lo que al finalizar este capítulo serás capaz

de crear tus propias redes neuronales desde cero.
5. En el quinto capítulo veremos diferentes tipos de redes neuronales y sus aplicaciones, centrándonos en la arquitecturas más potentes a día de hoy como las redes neuronales convolucionales o los `Transformers`.
6. En el sexto capítulo aprenderemos a entrenar redes neuronales de manera efectiva, aprovechando todos los recursos a nuestro alcance. Para ello, te daré una *receta* para entrenar redes neuronales que te servirá como punto de partida para tus propios proyectos o investigaciones.

Por último, al final del libro te dejaré unos consejos sobre cómo continuar aprendiendo sobre el Aprendizaje Profundo. Sin más dilación, y con la esperanza de haberte convencido de que vale la pena leer este libro, ¡empecemos!

1. Inteligencia Artificial

La IA en nuestra sociedad

La mayoría de las personas tienen un concepto de la *Inteligencia Artificial* (IA de ahora en adelante) adquirido a través de la literatura y el cine de ciencia ficción. Ya sea desde obras clásicas como las de Mary Shelley (*Frankenstein*), Isaac Asimov (*Las tres leyes de la robótica*, Yo Robot), Arthur C. Clarke (*2001: Una odisea del espacio*) o Philip K. Dick (*¿Sueñan los androides con ovejas eléctricas?*) hasta otras más recientes como *Terminator*, *Matrix* o *Ex Machina*, el deseo de crear seres inteligentes de forma artificial (y las posibles consecuencias que esto puede conllevar, raramente positivas para los pobres humanos) ha sido una constante. Y, debido a que esta es la única fuente de información que la mayoría de gente ha consumido, el miedo a la IA y la preocupación de que esta nos acabe controlando, o algo peor, es capaz de apoderarse del colectivo. Estos hechos son especialmente ciertos en el momento de escribir este libro, en el que aplicaciones como ChatGPT y similares parecen estar en todas las conversaciones, incluso con familiares o amigos que nunca se habían preocupado o ni siquiera habían oído hablar del tema. El miedo a la pérdida de empleos que la IA puede ocasionar y el cambio de paradigma socio-económico que puede conllevar no es poco común. Ante esta situación, el principal objetivo de este capítulo es acercar la IA de verdad (no la que vemos en series o películas, sino la que permite a aplicaciones como ChatGPT funcionar) a la gente, para que se pueda entender mejor qué es y qué no es la IA, cuáles son sus cosas buenas y malas, qué limitaciones presenta y qué maravillosas posibilidades abre.

¿Hay motivos para preocuparse? Pues como toda tecnología disruptiva con gran potencial de impacto, si no se usa de manera positiva, sí. De la misma forma que la energía nuclear puede ser usada para crear bombas atómicas o para generar energía eléctrica, o un cohete puede usarse para llevar astronautas a la luna o para bombardear una ciudad al otro lado del mundo, la IA es una herramienta que puede usarse tanto para el bien como para el mal, y depende de nosotros (de todos nosotros) asegurarnos que se usa con fines positivos. Pero incluso en el caso en el que la IA se use para fines positivos, esto no implica que los profundos cambios sociales y económicos resultantes no tengan consecuencias negativas. Hemos sido testigos a lo largo de la historia de situaciones similares, las más

recientes con la llegada de la Revolución Industrial (siglo XVIII en adelante) y la Revolución Digital (siglo XX). En ambos casos, la implantación de nuevas tecnologías ha supuesto la desaparición de muchos puestos de trabajo, pero también la creación de otros nuevos, y en general un aumento de la calidad de vida de la población. Una reestructuración social que, si bien es cierto necesita de un período de transición, se traduce en una mejora a nivel global a largo plazo.

Ante esta situación es clave asegurarnos que la IA se usa para *aumentarnos* como humanos, lo cual puede significar sustituirnos en aquellas tareas que, pese a ser necesarias, no aportan gran valor o que son de extremada peligrosidad. Sin embargo, la realidad es que estamos lejos de un sistema de IA que sea capaz de igualar, y menos superar, la capacidad humana. Si bien es cierto que hay sistemas de IA que son mucho mejores que las personas en tareas concretas (como jugar al ajedrez o al *go*), estos sistemas son incapaces de hacer nada más. Además, para entrenar dichos modelos de IA se requieren ingentes cantidades de datos y capacidad computacional. Por otro lado, la inteligencia humana se caracteriza por un elevado nivel de abstracción y adaptabilidad a nuevas situaciones, siendo capaz de aprender con pocos ejemplos y de generalizar a situaciones no vistas previamente con una mínima inversión energética. Esto no significa que la IA nunca vaya a ser capaz de llegar a tales niveles, de hecho la creencia general entre los investigadores y profesionales del sector es que sí (y, además, pronto) dando lugar a lo que se conoce como **Inteligencia Artificial General** (AGI por sus siglas en inglés). Es por este motivo que es de vital importancia que los desarrollos de IA se hagan de forma ética y responsable, minimizando el potencial daño que puedan llegar a causar.

Si tuviese que dar mi opinión, y al ser este mi libro te la voy a dar, creo que la Inteligencia Artificial General será el fin del *homo sapiens* tal y como lo conocemos (en cierto grado, nuestra dependencia de internet y los dispositivos conectados ya lo ha hecho). Pero esto no significa que la IA nos dominará o exterminará, visión compartida por algunos grupos de la sociedad, en parte por culpa de la imagen dada en las novelas o películas mencionadas anteriormente, sino que será el catalizador del siguiente paso evolutivo del ser humano. Hace ya varios siglos que la evolución por selección natural dejó de funcionar, siendo sustituida por la evolución por selección artificial. Somos los humanos los que decidimos cómo evolucionan las plantas y animales que comemos o tenemos como mascotas, y lo mismo ocurrirá con la siguiente generación de *homo cyborg* en el que la tecnología mejorará al ser humano en todos sus aspectos y abrirá nuevas posibilidades que ahora mismo son impensables (inmortalidad, exploración espacial, descubrimiento de los secretos del universo, etc.). Al fin y al cabo, la AGI será la última gran invención del ser humano. Una vez

desarrollada, podremos pedirle que resuelva el resto de nuestros problemas. Esta es la visión que me motiva cada día a levantarme de la cama y seguir trabajando en este apasionante campo.

¿Por qué deberías aprender Inteligencia Artificial?

Ya sea para aprovechar el gran potencial de la IA o minimizar el riesgo que pueda tener un tu día a día, creo que aprender cómo funciona esta tecnología es de vital importancia para todo el mundo. Por un lado, saber qué es (y qué no es) la IA te ayudará a no dejarte engañar o sucumbir a la paranoia social. Por otro lado, este conocimiento te ayudará a ejercer un cierto control sobre estos sistemas, minimizando el riesgo de convertirte en una de sus marionetas. A nivel práctico, saber crear sistemas de IA te permitirá automatizar tareas repetitivas y aburridas en tu trabajo, o incluso crear nuevas oportunidades de negocio y emprendimiento. Además, la IA es una de las tecnologías con mayor demanda laboral, por lo que si eres estudiante o estás buscando un cambio de trabajo, aprenderla te abrirá las puertas a un gran número de oportunidades.

Breve resumen histórico

Más allá de la ciencia ficción y especulaciones futuristas, la IA formal como rama de la ciencia de la computación tiene sus orígenes cerca de 1950, gracias al trabajo de Alan Turing (al cual se debe el famoso *test de Turing*). Si bien el trabajo de Turing fue muy importante para el campo de la computación, sus ideas entorno a la IA nunca salieron del papel. La primera aplicación práctica aparece pocos años después, cuando Frank Rosenblatt desarrolla el Perceptrón (1957). Este sistema físico (implementado en *hardware*, no *software*) era capaz de reconocer patrones visuales simples, como formas geométricas. Varios receptores fotoeléctricos se conectaban a un circuito electrónico que, tras un proceso de aprendizaje, era capaz de distinguir entre diferentes patrones. Dicho proceso se basaba en la modificación manual de los pesos de las conexiones entre los receptores y el circuito electrónico, de forma que se maximizase la diferencia entre los patrones que se querían distinguir. Este trabajo pionero cimentó las bases de lo que hoy en día se conoce como *Aprendizaje Profundo* y que descubriremos a lo largo de este libro. Por mucho tiempo que pase no dejo de maravillarme ante tal dispositivo, simplemente sublime. Sin embargo, el Perceptrón tenía importantes limitaciones, que veremos más adelante y, junto con la falta

de capacidad computacional de la época, hizo que el interés en la IA decayese, dando lugar a lo que se conoce como el **primer invierno de la IA**.

La siguiente revolución ocurre en la década de los 80, cuando Geoffrey Hinton desarrolla el algoritmo de *backpropagation* que permite entrenar redes neuronales de múltiples capas. Este algoritmo, junto con el aumento de la capacidad computacional, permitió el desarrollo de sistemas de IA más complejos y potentes, como, por ejemplo, las redes neuronales Convolucionales (Yann Lecun). Fue en este periodo, durante los 90, en el que se desarrolló el sistema Deep Blue, capaz de ganar al campeón del mundo de ajedrez Garry Kasparov en 1997. Sin embargo, Deep Blue no se puede considerar un sistema de IA de Aprendizaje Profundo, ya que no está basado en redes neuronales ni entrenado a partir de datos, sino que se basa en un algoritmo de búsqueda muy potente capaz de analizar millones de posibles movimientos en cada turno. Aun así, en la época fue considerado como un hito en la IA, y movió la frontera de lo que se consideraba como IA. Esto es algo bastante común en el campo, y que refleja lo difícil que es definir qué es y qué no es la IA. Pese a ello, el interés en la IA volvió a decaer dando lugar a lo que se conoce como el **segundo invierno de la IA**. Esto fue debido en parte a que los sistemas de Aprendizaje Profundo no eran capaces de superar a otros sistemas de *Machine Learning* como las Máquinas de Soporte Vectorial (SVM por sus siglas en inglés), que eran más fáciles de entrenar y más eficientes computacionalmente. El motivo, como se vio más tarde, era la falta de datos y capacidad computacional. Simplemente, su tiempo aún no había llegado.

Este segundo invierno duró hasta 2012, cuando Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton desarrollaron AlexNet, un sistema de IA capaz de reconocer objetos en imágenes con una precisión nunca vista hasta la fecha. La clave del éxito de este sistema fue el uso de GPUs para acelerar el entrenamiento de redes neuronales profundas, combinado con el uso de grandes cantidades de datos. Desde ese momento el interés en la IA no ha dejado de crecer, dando lugar a múltiples avances como Alpha Go, un sistema creado en 2017 por DeepMind (ahora parte de Google) capaz de ganar al campeón del mundo de Go, Lee Sedol. Este hito fue muy importante, ya que el Go es un juego mucho más complejo que el ajedrez, y el número de posibles movimientos es mucho mayor de lo que cualquier ordenador clásico podría procesar en toda una vida. Alpha Go utiliza un sistema de Aprendizaje Profundo basado en redes neuronales convolucionales, entrenado a partir de millones de partidas de Go jugadas por humanos, que es capaz de distinguir entre movimientos buenos y malos para reducir el espacio de búsqueda de posibles movimientos, lo cual podría compararse a la *intuición* de un jugador humano. En su versión más actual, Alpha Zero es capaz de alcanzar un nivel sobrehumano en pocas horas de entrenamiento sin ningún tipo

de conocimiento previo simplemente jugando contra sí mismo, en varios juegos como el ajedrez, el Go y otros. Unos años más tarde, en 2020, OpenAI anunció GPT-3, un sistema de IA capaz de generar texto a partir de una frase de entrada. Este sistema, basado en una nueva arquitectura de redes neuronales conocida como *Transformer*, es capaz de generar textos de una calidad muy alta, y es capaz de responder a preguntas, traducir textos, generar código, etc. GPT-3 sienta las bases de una nueva época en la que la IA está cada vez más presente en nuestro día a día a través de productos como ChatGPT o GitHub Copilot (el cual, por cierto, me está ayudando a escribir este libro, ¡muchas gracias!).

¿Veremos un tercer invierno de la IA? Basándonos en la historia, podría ser. Posibles factores serían que las arquitecturas actuales no fuesen capaces de seguir mejorando, aunque de momento no se vislumbra un límite y a más datos y computación que les damos, mejores modelos obtenemos (hay gente que piensa que simplemente escalando los modelos actuales con más datos y más GPUs alcanzaremos la AGI). Otros motivos podrían ser que la IA no sea capaz de resolver problemas reales (desarrollo de nuevos medicamentos o cura de enfermedades, descubrimiento de nuevos materiales, etc.), o que el miedo a su uso paralice su desarrollo. Sin embargo, la coyuntura actual no parece indicar que esto vaya a ocurrir, y la IA está cada vez más presente en nuestras vidas: en nuestros *smartphones*, coches autónomos, sugiriéndonos los mejores productos que comprar en internet, etc.

Pero, ¿qué es la Inteligencia Artificial?

Inteligencia Artificial, Aprendizaje Profundo, redes neuronales, *Machine Learning*… Si no estás familiarizado con este campo es posible que te estés liando un poco con todos estos conceptos que he ido soltando a lo largo del capítulo. Es por este motivo que en este apartado voy a definir todos estos conceptos y cómo se relacionan entre sí.

Inteligencia Artificial

Si bien no existe una definición consensuada sobre qué es la IA, algunos intentos son:

- IA es la ciencia de hacer que las máquinas hagan cosas que requieren inteligencia cuando las hacen las personas (Marvin Minsky, 1968)
- Se dice que un programa aprende de la experiencia E con respecto a alguna clase de tareas T y medida de rendimiento P, si su rendimiento en tareas en T, según lo medido

por P, mejora con la experiencia E (Mitchell, 1997)
- La IA es la ciencia e ingeniería de hacer máquinas inteligentes (McCarthy, 2007)

Como norma general, desconfía de cualquier definición que incluya en ella el propio concepto que se trata de definir. Como ya he comentado anteriormente, definir la IA no es tarea fácil. Una definición actual que me gusta particularmente es la que da François Chollet (creador de Keras) en su artículo *On the measure of Intelligence*, en la que define la inteligencia como una medida de la eficiencia para aprender nuevas habilidades. Queremos que la futura AGI sea capaz de aprender nuevas habilidades y llevar a cabo nuevas tareas con la mínima cantidad de datos y computación posible. Esto es algo que los humanos hacemos muy bien, y que los sistemas de IA actuales no son capaces de hacer (de momento).

Aprendizaje Automático (*Machine Learning*)

El Aprendizaje Automático (ML por sus siglas en inglés) es una rama de la IA que se centra en el desarrollo de algoritmos que son capaces de aprender a partir de datos de ejemplo. El objetivo es que estos algoritmos, también llamados **modelos**, sean capaces de aprender por sí mismos desde cero a través de un proceso llamado **entrenamiento** y generalizar más tarde a situaciones no vistas previamente durante el entrenamiento. El ML se puede dividir en tres categorías:

- Aprendizaje supervisado: los datos de entrenamiento incluyen las entradas y las salidas esperadas. El objetivo es que el modelo sea capaz de predecir la salida a partir de una entrada no vista previamente. Ejemplos de aplicación incluyen la clasificación de imágenes, la detección de objetos, la traducción de textos, etc.
- Aprendizaje no supervisado: los datos de entrenamiento incluyen únicamente las entradas. El objetivo es que el modelo sea capaz de encontrar patrones en los datos. Ejemplos de aplicación incluyen la agrupación de imágenes, la detección de anomalías, la generación de texto, etc.
- Aprendizaje por refuerzo: el modelo interactúa con un entorno y recibe una recompensa por cada acción que lleva a cabo. El objetivo es que el modelo aprenda a llevar a cabo la acción que maximiza la recompensa dada una situación (lo cual no implica que sea la mejor acción). Es como si entrenases a tu perro a sentarse, dándole un premio cada vez que lo haga al recibir tu orden hasta implantar ese comportamiento en su subconsciente. Ejemplos de aplicación incluyen la robótica, IAs para juegos

(como Alpha Go o Alpha Zero), etc.

Dentro del ML existen muchos tipos de algoritmos, como las redes neuronales, las máquinas de soporte vectorial, los árboles de decisión, etc. En este libro nos centraremos en las redes neuronales, ya que son las que han demostrado ser más potentes y versátiles. Aun así, el uso de algoritmos «tradicionales» sigue teniendo un peso muy importante en la industria, sobretodo en aquellas aplicaciones en las que se dispone de pocos datos. Si te interesa aprender más sobre este tipo de algoritmos y sus aplicaciones, te recomiendo el módulo de *Machine Learning* que encontrarás en mi curso online.

Un ejemplo de IA que no se fundamenta en el aprendizaje a partir de datos son los sistemas expertos, o IA simbólica. Estos algoritmos se basan puramente en reglas codificadas por expertos humanos, representando conocimiento como símbolos, pudiendo llevar a cabo razonamiento y operaciones lógicas con ellos. Deep Blue, el sistema de IA que ganó al campeón del mundo de ajedrez, es un ejemplo de sistema experto. En este caso, las reglas codificadas por expertos humanos son las reglas del ajedrez. Sin embargo, somos incapaces de codificar todas las reglas necesarias para que un sistema de IA sea capaz de llevar a cabo tareas complejas como conducir un coche o jugar al Go. Es por este motivo que la IA simbólica se ha visto relegada a un tercer plano en el panorama global de la IA. Su uso, especialmente en conjunto con las redes neuronales profundas, la generación de código u otras técnicas de IA como los algoritmos evolutivos, es una línea de investigación poco activa pero que puede dar lugar a grandes avances en el futuro (¿quizás la AGI?).

Aprendizaje Profundo (*Deep Learning*)

El Aprendizaje Profundo (DL por sus siglas en inglés) es una rama del ML que se centra en el desarrollo de modelos basados en redes neuronales profundas. Su principal ventaja frente al resto de algoritmos de ML es que son capaces de aprender de forma automática las características más importantes de los datos a la hora de llevar a cabo su tarea. Por ejemplo, para clasificar imágenes de mascotas, las redes neuronales profundas son capaces de detectar patrones de bajo nivel, como colores, formas y texturas, y combinarlos para detectar patrones de alto nivel, como orejas, ojos, nariz, etc. Esto es algo que los humanos hacemos de forma natural, pero que los algoritmos de ML tradicionales no son capaces de hacer, y que requiere de un gran esfuerzo por parte de los expertos humanos en un proceso conocido como **ingeniería de características** (*feature engineering* en inglés). Esta es la clave que hace del Aprendizaje Profundo una técnica de IA tan potente, ya que lo único que hay que hacer es darle todos los datos de los que dispongamos y un objetivo

claro, y el modelo se encargará de aprender todo lo necesario para llevar a cabo la tarea. La desventaja, por otro lado, es la inmensa cantidad de datos y capacidad computacional que requieren estos modelos para funcionar.

El Aprendizaje Profundo ha encontrado un gran aliado en las GPUs, procesadores gráficos que sirven para acelerar ciertos cálculos necesarios para renderizar gráficos en tiempo real en las pantallas de un ordenador. Estos procesadores, que fueron diseñados para llevar a cabo cálculos matriciales de forma muy eficiente, son también perfectos para acelerar el entrenamiento de redes neuronales profundas. Esto ha permitido el desarrollo de modelos cada vez más complejos y potentes, dando lugar a los grandes avances que hemos visto en los últimos años, así como a la subida en bolsa de las acciones de NVIDIA, principal fabricante de GPUs. Es tal la demanda por este tipo de *hardware*, y poca la oferta, que muchas nuevas empresas están apareciendo en el mercado para intentar cubrir esta necesidad, todas ellas con su flamante chip acelerador de IA.

Aplicaciones de la IA

Si estás leyendo este libro y has llegado hasta este punto no creo que necesite convencerte de que la IA tiene multitud de aplicaciones reales. Aun así, veamos una lista de aplicaciones del Aprendizaje Profundo, quizás descubres alguna que no conocías:

- La **Visión por Computador** (CV o *Computer Vision* en inglés) es una de las aplicaciones más populares del Aprendizaje Profundo. El objetivo es que un sistema de IA sea capaz de entender el contenido de una imagen o vídeo. Ejemplos de aplicación incluyen la clasificación de imágenes, la detección de objetos, la segmentación de imágenes, la generación de imágenes, etc. Algunos ejemplos de sistemas de IA basados en CV son los sistemas de reconocimiento facial, los sistemas de detección de objetos en vehículos autónomos o imágenes por satélite, los sistemas de detección de anomalías en imágenes médicas, etc.
- El **Procesamiento del Lenguaje Natural** (NLP o *Natural Language Processing* en inglés) es otra de las aplicaciones más populares del Aprendizaje Profundo. El objetivo es que un sistema de IA sea capaz de entender el contenido de un texto. Ejemplos de aplicación incluyen la traducción de textos, la generación de texto, la detección de sentimientos, la generación de código, etc. Algunos ejemplos de sistemas de IA basados en NLP son los asistentes virtuales como ChatGPT, los sistemas de traducción automática, los sistemas de generación de código, etc.

- El **Procesamiento de Audio** consiste en el desarrollo de IAs capaces de entender el contenido de un audio. Ejemplos de aplicación incluyen la transcripción de audio, la generación de audio, la detección de sentimientos, etc. Algunos ejemplos de sistemas de IA basados en audio son los asistentes virtuales, los sistemas de traducción automática, los sistemas de generación de código, etc. Este campo está muy relacionado con el NLP y se usa, por ejemplo, en los asistentes de voz de los *smartphones*.
- **Otras aplicaciones** incluyen el análisis de series temporales (predicción meteorológica, predicción de precios de acciones, etc.), procesado de estructuras tridimensionales (desarrollo de nuevos medicamentos, descubrimiento de nuevos materiales, etc.), robótica (robots autónomos, robots que aprenden a través de la interacción con el entorno, etc.), aplicaciones científicas (resolución de ecuaciones, etc.) y muchas más.

Si bien las aplicaciones de IA han estado tradicionalmente restringidas a un solo campo de aplicación, los modelos más potentes como GPT-4 (al menos en el momento de escribir este libro) no solo son capaces de llevar a cabo muchas aplicaciones, sino que también son capaces de trabajar con diferentes modalidades de datos a la vez (por ejemplo, imágenes y texto). Esta multimodalidad aporta una gran potencia a los modelos de IA y abre la posibilidad a aplicaciones avanzadas como la robótica, los sistemas de diagnóstico médico, etc.

Tu primera red neuronal

Para cerrar este capítulo, y antes de pasar al siguiente, vamos a crear nuestra primera IA. En este caso será una red neuronal muy sencilla, la cual entrenaremos para reconocer dígitos manuscritos. Para ello, te animo a que visites este enlace y cliques en el botón de *Entorno de ejecución > Ejecutar todas* para ejecutar el código. No te preocupes si no entiendes el código ni lo que está pasando, aprenderlo es el principal objetivo de este libro. Lo más importantes es ver que, en primer lugar, estamos descargando un conjunto de datos de internet que contiene ejemplos de la tarea que queremos llevar a cabo.

Python

```python
from sklearn.datasets import fetch_openml
import numpy as np


# descarga de datos

mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

En segundo lugar, definiremos la arquitectura de red neuronal que queremos usar.

Python

```python
from torch.nn import Sequential as S
from torch.nn import Linear as L
from torch.nn import ReLU as R

model = S(L(784,128),R(),L(128,10))
```

Después, entrenaremos la red neuronal con los datos que hemos descargado.

Python

```python
from tqdm import tqdm

X_train, X_test = torch.from_numpy(X[:60000] / 255.), torch.from_numpy(X
    [60000:] / 255.)
y_train, y_test = torch.from_numpy(y[:60000]), torch.from_numpy(y
    [60000:])
bs = 32
num_batches = len(X_train) // bs

loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(10):
    for b in tqdm(range(num_batches)):
        x = X_train[b*bs:(b+1)*bs]
        y = y_train[b*bs:(b+1)*bs]
        y_hat = model(x)
        loss = loss_fn(y_hat, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} loss: {loss.item():.3f}")
```

Por último, evaluaremos la precisión de la red neuronal con datos que no ha visto durante el entrenamiento (si no, estaríamos haciendo trampa).

Python

```python
def evaluate(model, X_test, y_test):
    acc = 0
    with torch.no_grad():
        for b in range(num_batches):
            x = X_test[b*bs:(b+1)*bs]
            y = y_test[b*bs:(b+1)*bs]
            y_hat = model(x)
            acc += torch.sum(torch.argmax(y_hat, dim=1) == y).item()
    return acc

acc = evaluate(model, X_test, y_test)
print(f"Accuracy: {acc} / {len(X_test)}")
```

Durante los siguientes capítulos iremos diseccionando este ejemplo para entender exactamente qué es lo que está pasando, con todo lujo de detalles, para que seas capaz, una vez terminado, de diseñar y entrenar tus propias redes neuronales. Como si de una cebolla se tratase, iremos quitando capas de abstracción hasta llegar al núcleo, a los conceptos fundamentales que dan como resultado las aplicaciones de IA mencionadas anteriormente, y cómo podemos usarlo para crear las nuestras propias.

2. Introducción a la Computación con Python

Muy bien, pues ahora que ya sabes de qué va esto de la Inteligencia Artificial y qué tipo de aplicaciones tiene el Aprendizaje Profundo (si no lo sabes te recomiendo que leas al capítulo anterior), probablemente tengas ganas de empezar a trastear y hacer tu primera aplicación. La buena noticia es que nunca ha sido tan sencillo y accesible para cualquier tipo de perfil. Existen multitud de herramientas `no-code` o `low-code` que te permiten crear aplicaciones de Aprendizaje Profundo sin necesidad de saber programar, y cada día aparecen nuevas, ya que el análisis de datos en general y la extracción de información *accionable* de los mismos es vital en muchas actividades. Así pues, ofrecer soluciones para gente sin conocimientos de programación o perfiles no técnicos es un mercado en auge. Sin embargo, en el momento en el que quieras hacer algo un poco más complejo o personalizado, vas a necesitar saber programar.

A partir de este punto, la mayor parte del contenido de este libro asume que sabes desenvolverte con Python, el lenguaje de programación más utilizado en el Aprendizaje Profundo. Si ya conoces este lenguaje de programación, es posible que prefieras saltar directamente al siguiente capítulo. Por el contrario, si no tienes ni idea de programar o quieres asentar las bases, te recomiendo que sigas leyendo. A lo largo de este capítulo aprenderás qué significa programar, qué es un algoritmo, qué es un lenguaje de programación y los fundamentos de la programación con el lenguaje Python.

En mi curso *online* puedes encontrar este capítulo en formato vídeo en el primer módulo, e incluso puedes ir un poco más allá en el segundo para aprender más sobre Python aplicado al análisis de datos. Ambos módulos son gratuitos y contienen multitud de ejercicios para poner a prueba todo lo que vas a aprender en este capítulo.

¿Qué es la programación?

De entre las diferentes definiciones de *programación* que podemos encontrar, la que personalmente me gusta más es la siguiente:

> Programar consiste en definir una secuencia de instrucciones que un ordenador debe ejecutar para resolver un problema.

Fácil, ¿no? Así pues, un programa no es más que una lista de instrucciones para resolver un problema. Para ello, un programa aceptará un conjunto definido de entradas (o *inputs*) y generará salidas (o *outputs*) como resultado de ejecutar las instrucciones.

Aprender a programar, entonces, no significa otra cosa que aprender a definir esta secuencia de instrucciones que un ordenador debe ejecutar para resolver el problema al que nos enfrentemos. Volviendo al ejemplo que hemos visto en el capítulo anterior, esto consiste en:

1. Descargar y preparar un conjunto de datos.
2. Definir un modelo de Aprendizaje Profundo.
3. Entrenar el modelo.
4. Evaluar el modelo.
5. Usar el modelo para hacer predicciones.

El siguiente reto será traducir estos pasos a un lenguaje que un ordenador pueda entender. Para ello, tenemos a nuestra disposición diferentes herramientas que iremos descubriendo a lo largo de este capítulo.

Un efecto secundario de aprender a programar es que, al hacerlo, también desarrollarás tu capacidad de resolución de problemas (el famoso *problem solving*) aprendiendo a pensar de manera lógica y estructurada. Esto te dotará de herramientas para resolver problemas en tu vida diaria, más allá de la programación.

¿Qué es un algoritmo?

El siguiente concepto del que vamos a hablar es el de *algoritmo*, una palabra también muy común en la programación y que hace referencia a la secuencia de instrucciones que el ordenador debe seguir para resolver un problema. Como veremos más adelante, estas instrucciones deben indicar al ordenador cómo manipular paso a paso los datos con los que estamos trabajando. Un algoritmo que es capaz de resolver un problema se conoce como un algoritmo *correcto* y *preciso*. En el caso contrario, decimos que el algoritmo tiene un *bug*, o error.

> El concepto de *bug*, bicho en inglés, se popularizó porque los primeros ordenadores fallaban a menudo debido a insectos que se colaban en los circuitos y provocaban que no funcionaran correctamente.

Otro concepto muy importante es el de la **eficiencia** (tanto temporal como en términos de memoria), y es que un algoritmo puede ser correcto, pero si tarda mucho tiempo en ejecutarse o necesita muchísima memoria puede ser inútil.

Veamos varios ejemplos de algoritmos sencillos: Vamos a diseñar varios algoritmos para encontrar una palabra en el diccionario (esos libros enormes que contienen las definiciones de todas las palabras y que solo se usan para calzar una cama que cojea). El primer algoritmo será de tipo aleatorio. Abriremos el diccionario por una página aleatoria y comprobaremos si nuestra palabra se encuentra en esa página. De ser así, habremos conseguido nuestro objetivo. En caso contrario, cerraremos el diccionario y volveremos a repetir el proceso. ¿Es este algoritmo correcto? Depende, si la palabra efectivamente se encuentra en el diccionario entonces, en el límite, cuando hayamos abierto todas las posibles páginas en algún momento habremos acertado. Esto puede pasar al primer intento o al intento un millón. Si la palabra no está en el diccionario, sin embargo, jamás lo sabremos y este algoritmo quedará atascado en un bucle infinito. ¿Es eficiente? De nuevo, depende. Si acertamos a la primera sí, pero si tenemos que abrir el diccionario un millón de veces no lo es tanto. Este tipo de algoritmo está sujeto al azar y condiciones de contorno, por lo que no es muy útil (aunque pueda funcionar muy bien en algunas ocasiones). Un algoritmo que nos permitiría encontrar la palabra que buscamos de forma segura y en condiciones controladas sería el que consiste en ir página a página, empezando por la primera y procediendo de manera secuencial hasta encontrar la página adecuada (o al final si la palabra no está en el diccionario). Este algoritmo es correcto pero poco eficiente, ya que potencialmente tendremos que recorrer todas las páginas del diccionario. Podemos acelerar este algoritmo pasando las páginas de dos en dos. En este caso recortamos el tiempo a la mitad, pero hemos introducido un *bug*. ¿Eres capaz de verlo? Efectivamente, si la palabra que buscamos se encuentra justo entre las dos páginas que pasamos, nunca la encontraremos. Este *bug* tiene fácil solución, y es que podemos volver una página atrás en el caso de que nos hayamos pasado. Sin embargo, esto asume que el diccionario está ordenado alfabéticamente. Este hecho nos permite poner de manifiesto que la manera en la que tenemos organizados nuestros datos afecta a la eficiencia de nuestros algoritmos (algo que veremos más en detalle cuando hablemos de estructuras de datos). Introducir esta condición nos permite ahora desarrollar un último algoritmo mucho más eficiente, el algoritmo de búsqueda bi-

naria (el cual probablemente lleves a cabo de manera inconsciente). Este consiste en abrir el diccionario por la mitad. En el caso de que nuestra palabra se encuentre en la primera mitad, repetiremos el proceso con esta mitad. En caso contrario, lo haremos con la segunda mitad. De esta manera, en cada paso del algoritmo, en cada *iteración*, reducimos a la mitad el número de páginas que tenemos que mirar. Este algoritmo es correcto y eficiente, y es un ejemplo de algoritmo muy utilizado para encontrar información en grandes bases de datos.

Algoritmo	Correcto	Eficiente	Condiciones
Aleatorio	No	No	Ninguna
Secuencial	Sí	No	Ninguna
Secuencial de dos en dos	Sí	No	Diccionario ordenado alfabéticamente
Búsqueda binaria	Sí	Sí	Diccionario ordenado alfabéticamente

Pseudocódigo

Antes de pasar a la implementación de algoritmos usando lenguajes de programación vamos a introducir una herramienta muy potente conocida como *pseudocódigo*. Esta técnica consiste en describir las diferentes instrucciones, los pasos del algoritmo, usando lenguaje natural pero con una estructura similar a la de un lenguaje de programación. De esta manera, podemos describir un algoritmo a alto nivel para crear una plantilla, o receta, que podemos más tarde implementar en cualquier lenguaje de programación. Por ejemplo, el pseudocódigo para el algoritmo de búsqueda secuencial sería el siguiente:

1. Abrir el diccionario por la primera página.
2. Comprobar si la palabra que buscamos se encuentra en la página actual.
    - Si es así, hemos terminado.
    - Si no, pasar a la siguiente página y volver al paso 2.
3. Si hemos llegado al final del diccionario, la palabra no se encuentra en él.

¿Te ves capaz de escribir el pseudocódigo para el algoritmo de búsqueda binaria?

Por otro lado, el pseudocódigo para el entrenamiento de nuestra red neuronal es el siguiente:

1. Preparar el conjunto de datos de entrenamiento (descarga, limpieza, procesado, …).
2. Definir la arquitectura de la red neuronal.
3. Definir la función de pérdida y el optimizador.
4. Iterar sobre el conjunto de datos.
    - Generar un *batch* de datos.
    - Calcular la salida de la red neuronal.
    - Calcular el valor de la función de pérdida.
    - Actualizar los pesos de la red neuronal.
5. Repetir el paso 4 un número de iteraciones determinado (*epochs*).
6. Evaluar el modelo.

De nuevo, no te agobies si no entiendes algunos conceptos como el de función de pérdida, optimizador, *batch* o *epochs*. Durante este libro aprenderás en detalle su significado y cómo implementarlos en Python para entrenar tus redes neuronales.

Lenguajes de programación

Pues ahora que hemos visto cómo podemos diseñar algoritmos para resolver problemas, vamos a aprender cómo podemos implementarlos en un ordenador. Para ello, necesitaremos una manera de traducir nuestras instrucciones a lenguaje máquina. Como puedes imaginar hacer esto a mano sería una tarea muy tediosa y propensa a errores, e imposible en aplicaciones complejas, por lo que se han desarrollado herramientas que nos permiten traducir de un lenguaje más manejable para las personas al lenguaje binario que entienden los ordenadores. Estas herramientas se conocen como *compiladores* e *intérpretes*, que no son más que otros programas que traducen el código de un *lenguaje de programación* a código máquina. De esta forma, podemos expresar nuestras intenciones en un lenguaje que sea fácil de entender para nosotros, y que el ordenador pueda entender y ejecutar después del proceso de *traducción*.

Existen muchísimos lenguajes de programación, que podemos categorizar en lenguajes de bajo nivel, lenguaje de alto nivel y lenguajes de programación gráficos. Los lenguajes de bajo nivel se caracterizan por proporcionar un mayor control al desarrollador sobre los recursos de *hardware* y memoria de un ordenador, permitiendo el desarrollo de algoritmos muy eficientes a costa de una mayor complejidad (que puede resultar en muchos *bugs*). Algunos ejemplos de estos lenguajes de bajo nivel son C, C++ o Rust, entre muchos otros.

Por otro lado, los lenguajes de alto nivel se caracterizan por proporcionar una mayor abstracción, permitiendo al desarrollador centrarse en la lógica del algoritmo delegando los detalles de bajo nivel al propio lenguaje o interpretador. Algunos ejemplos de estos lenguajes de alto nivel son Python, Julia o JavaScript. Si bien estos lenguajes son más fáciles de aprender, suelen ser menos eficientes que los de bajo nivel (aunque en la mayoría de aplicaciones esto no suele ser un problema, y si lo es, es posible hacer algunos *apaños* como veremos más tarde). Por último, los lenguajes de programación gráficos son aquellos que permiten al desarrollador crear programas mediante la interacción con elementos gráficos, como bloques o diagramas. Algunos ejemplos de estos lenguajes de programación gráficos son Scratch, Blockly o LabVIEW. Estas interfaces *no-code* son muy útiles para iniciarse en el mundo de la programación, pero suelen ser muy limitadas y no permiten crear aplicaciones complejas.

En lo que queda de capítulo y de libro nos centraremos en aprender y aplicar Python para el Aprendizaje Profundo. Si esto es de tu interés, ¡seguimos!

> Si nunca has programado y quieres aprender, te recomiendo invertir unas horitas jugando con Scratch antes de pasar a la siguiente sección. En este vídeo te dejo un pequeño tutorial para que aprendas los conceptos básicos.

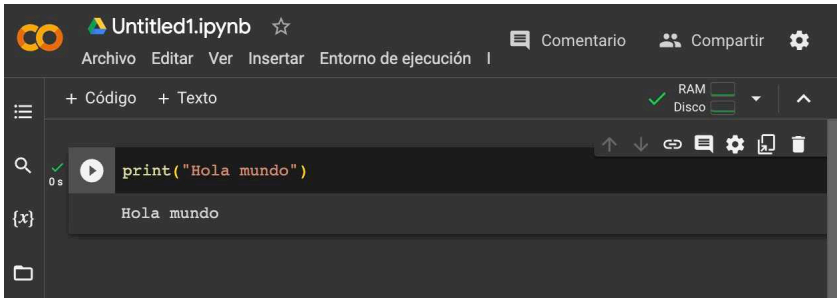
Python

Haciendo un resumen rápido de lo visto hasta ahora, programar consiste en el desarrollo de programas, o *algoritmos*, que contienen una serie de instrucciones escritas en un *lenguaje de programación* que, tras ser traducidas a código máquina, son ejecutadas por un ordenador para resolver un problema. La elección del lenguaje de programación es clave, ya que determinará el espacio de posibilidades que tendremos a nuestro alcance. En el caso del análisis de datos, *machine learning*, aprendizaje profundo y otras disciplinas englobadas en lo que llamamos Inteligencia Artificial, Python es el rey indiscutible. Esto es debido a múltiples factores. Algunos a destacar son: su facilidad de uso y aprendizaje (lo que lo hace ideal para gente que está aprendiendo a programar), es un lenguaje multiplataforma (funciona en Windows, MacOS y Linux, entre otros), es un lenguaje de código abierto y gratuito y, por último, pero lo más importante, tiene una enorme comunidad de desarrolladores y un gran ecosistema de librerías que implementan muchísima funcionalidad que podemos reapro-

vechar sin tener que desarrollarla nosotros mismos. Puedes encontrar más información sobre Python en su página web.

Primeros pasos

Como ya hemos visto en el capítulo anterior, la forma más sencilla de empezar a usar Python es usando Colab, un servicio gratuito de Google que nos permite ejecutar código Python en la nube. Para ello, simplemente tenemos que ir a la página de Colab y crear un nuevo *notebook* (un documento donde podemos escribir y ejecutar código Python). Una vez creado, podemos empezar a escribir código en las celdas de código. Por ejemplo, podemos escribir `print("Hola mundo")` y ejecutar la celda pulsando el botón de *play* que aparece a la izquierda de la celda. Si todo ha ido bien, deberías ver el texto `Hola mundo` justo debajo de la celda. ¡Enhorabuena! Acabas de ejecutar tu primer programa en Python.



Notebooks vs. *Scripts*

Los *notebooks*, o cuadernos, son entornos interactivos que permiten la ejecución de código en el navegador. Estos cuadernos están formados por celdas, que pueden ser de dos tipos: celdas de código y celdas de texto. Las celdas de código son aquellas que contienen código Python, mientras que las celdas de texto contienen texto formateado con Markdown. Además, los resultados de las celdas que contienen código son visibles en el *notebook* (incluyendo imágenes y gráficas). Esto hace de los *notebooks* un entorno ideal para el prototipado rápido de ideas, la exploración y análisis de datos e incluso la creación de informes. Con la ayuda de algunos *plugins* o extensiones incluso se puede ir más allá y crear presentaciones, páginas web, ¡o incluso libros como este mismo que estás leyendo! Por otro lado, una vez hayas desarrollado tu código y quieras ponerlo en producción,

es recomendable usar *scripts*, o archivos de texto que contienen todo el código que deba ejecutarse, como veremos más adelante.

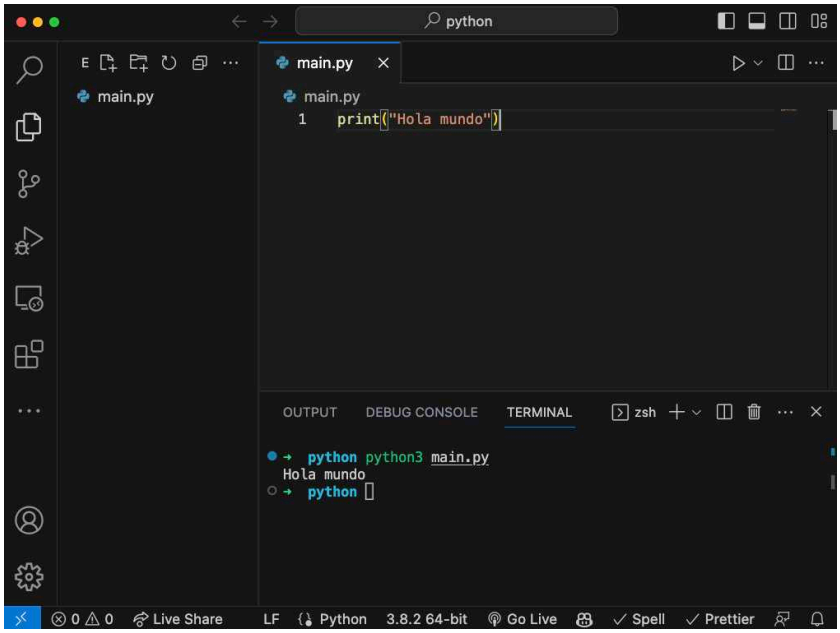
Si bien puedes trabajar con *notebooks* de manera local, luego veremos cómo, existen varios servicios que nos permiten trabajar directamente en la nube. Ya hemos visto Colab, de Google, pero también te puede interesar echar un vistazo a los *notebooks* de Kaggle. Ambos ofrecen la posibilidad, además, de ejecutar nuestro código en GPUs y TPUs, lo que nos permite acelerar el entrenamiento de nuestros modelos de aprendizaje profundo.

Todo el código que aparece en este libro podrás ejectuarlo en *notebooks* (a no ser que indique lo contrario), por lo que dejo a tu elección la plataforma que quieras usar. Si te sientes más cómodo trabajando con *notebooks*, adelante. Si prefieres trabajar con *scripts*, estupendo. En cualquier caso, te recomiendo que te familiarices con los *notebooks*, ya que son una herramienta muy útil para el desarrollo de aplicaciones de aprendizaje profundo.

Instalación

Independientemente de si decides trabajar con *notebooks* o *scripts*, te recomiendo configurar un entorno de desarrollo en tu ordenador, lo que llamamos trabajar en local u *on-premise* en contraposición a trabajar en remoto en la nube, el *cloud* (como hemos visto antes usando Colab). En el primer caso, los recursos computacionales serán aquellos de los que dispongas en tu ordenador, con las limitaciones que ello conlleva. En el segundo, tendrás más flexibilidad a la hora de elegir los recursos que necesitas a cambio de un mayor coste y complejidad.

Lo primero que necesitas para empezar a programar con Python es un editor de texto. Existen varias alternativas, pero mi recomendación es usar Visual Studio Code, un editor gratuito y de código abierto desarrollado por Microsoft con una excelente integración con Python y *notebooks*. Puedes instalarlo en cualquier sistema operativo usando las instrucciones que encontrarás en su página web. Una vez instalado, deberías ver algo como esto:



La apariencia final de tu editor dependerá de la configuración que uses, pero los tres componentes principales que usaremos son el explorador de archivos (panel de la izquierda), el panel principal (donde escribiremos el código de Python) y la terminal (panel inferior, que usaremos para ejectuar *scripts* y otros programas). Te animo a que crees un nuevo archivo, al que puedes llamar `main.py` con el mismo contenido del *notebook* que hemos visto antes (`print("Hola mundo")`). Para poder ejecutar el *script* con Python usamos el comando `python` seguido del nombre del archivo. Para nuestro ejemplo:

Bash

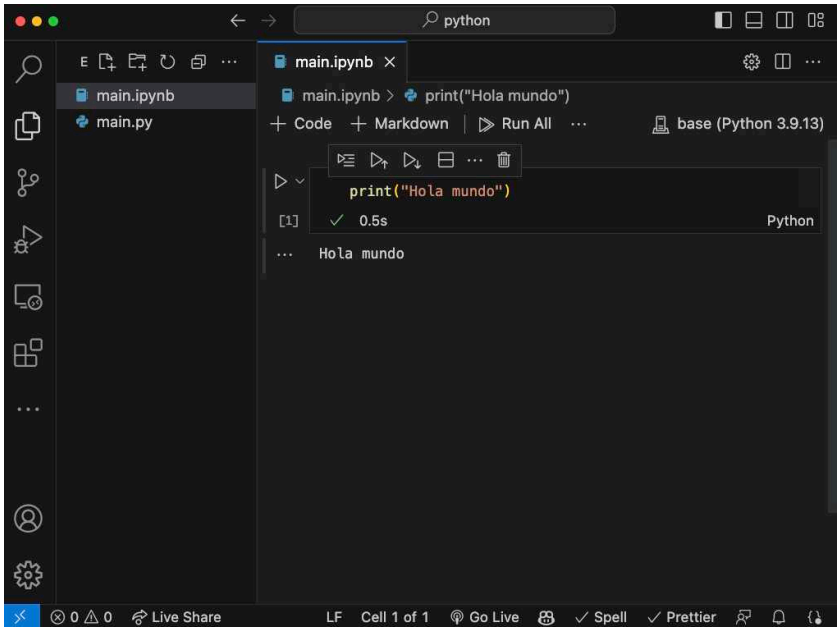
```
python main.py
```

En el ejemplo de la imagen puedes ver el comando `python3 main.py` en la terminal, ya que estoy usando la versión 3 de Python, aunque por lo general podrás usar ambas formas indistintamente.

Es probable que en tu ordenador ya tengas instalado Python. Para comprobarlo puedes ejecutar el comando anterior y ver si funciona. En caso contrario, tendrás que instalarlo. Existen varias formas de instalar Python, la más sencilla es probablemente seguir las instrucciones de la página oficial. Una alternativa consiste en usar Miniconda, una distribución

de Python que incluye además un gestor de paquetes, aunque su instalación puede ser un poco más compleja.

En cualquier caso vas a poder instalar librerías usando el gestor de paquetes `pip`. Por ejemplo, para instalar la librería `jupyter` ejecutamos el comando `pip install jupyter` en la terminal. Este paquete nos permitirá ejecutar *notebooks* en local. Para ello, ejecutamos el comando `jupyter notebook` en la terminal. Esto abrirá una nueva pestaña en nuestro navegador con el explorador de archivos de Jupyter. Desde aquí podemos crear nuevos *notebooks* o abrir los que ya tengamos en nuestro ordenador. Alternativamente, puedes abrir un *notebook* desde Visual Studio Code (mi opción favorita). Para ello, crea un archivo acabado en `.ipynb` y ábrelo en el editor.



El último paso para dejar todo listo es instalar la extensión de Python para VSCode. Para ello, abre el panel de extensiones y busca Python. ¡Instala la extensión y listo! Si necesitas más ayuda en el proceso de instalación y configuración, te recomiendo que mires este video en el que lo hacemos todo paso a paso.

También te recomiendo que te acostumbres a buscar en internet (o preguntarle a tu IA favorita) cualquier error o problema que te vayas encontrando por el camino. Te aseguro que te van a salir muchos errores, muchos, y una búsqueda rápida es la mejor forma de

resolverlos. Puedes copiar y pegar en tu navegador el error o, si es muy largo, copiar la parte del error más relevante que te pueda servir para encontrar una solución.

El interpretador de Python

Antes de seguir me gustaría dedicar unas líneas a explicar qué es lo que está pasando cuando ejecutamos el comando `python main.py` o bien cuando ejecutamos una celda de un *notebook*. En ambos casos, lo que ocurre es que un programa llamado el interpretador (que has instalado en tu ordenador al instalar Python) traduce el código al lenguaje máquina (los ceros y unos que el ordenador entiende) y lo ejecuta. Este interpretador está desarrollado usando el lenguaje de programación de bajo nivel C. Es importante diferenciar entre un interpretador y un compilador, pese a que sus funciones son similares. Un compilador traduce el código fuente a código máquina de una vez, mientras que un interpretador lo hace línea a línea. Esto implica que los lenguajes compilados, como C o C++, tengan que ser recompilados cada vez que se modifica el código fuente (lo cual puede llevar un buen rato en proyectos grandes con muchas interdependencias), mientras que los lenguajes interpretados, como Python, no lo necesitan. Esto se traduce en una gran agilidad a la hora de programar a costa de perder eficiencia en tiempo de ejecución, ya que el código máquina generado por un compilador es más eficiente que el generado por un interpretador.

El ecosistema de Python

Tal y como he mencionado anteriormente, la principal ventaja que ofrece Python frente a otros lenguajes de programación es el gran ecosistema de librerías que existen. Estas librerías son paquetes de código que otros desarrolladores han implementado y que nosotros podemos descargarnos y utilizar directamente en nuestros programas. Como norma general, si quieres hacer algo en Python, primero haz una búsqueda rápida en internet, ya que es muy probable que exista una librería que ya lo haga, ahorrándote tiempo y la posibilidad de introducir *bugs* en tu código. Algunos ejemplos de librerías enfocadas en la IA son:

- Librerías para análisis de datos: numpy, pandas, matplotlib, …
- Librerías de Machine Learning: scikit-learn, xgboost, lightgbm, …
- Librerías de Deep Learning: pytorch, tensorflow, keras, …

En este libro aprenderás a usar la librería Pytorch que te permitirá diseñar y entrenar redes neuronales sin la necesidad de implementar desde cero todo lo necesario para ello. Esta

es la que te recomiendo usar en tu día a día como estudiante, profesional o investigador del Aprendizaje Profundo. Sin embargo, es importante conocer cómo funcionan este tipo de librerías por dentro, por lo que dedicaremos un par de capítulos a ello en los que aprenderás a usar la librería de Numpy, aunque como veremos más adelante ambas son muy parecidas.

Si quieres usar estas librerías en tus proyectos, el primer paso será instalarlas. Ya hemos visto un ejemplo de cómo instalar librerías de Python, usando el gestor de paquetes `pip`. Para instalar una librería será tan fácil como escribir `pip install` en tu terminal seguido del nombre de la librería. Una vez instalada, podrás utilizarla en tus programas usando la palabra `import` seguida del nombre de la librería. Por ejemplo:

Python

```
pip install numpy
```

Instalará la librería de Numpy, y

Python

```
import numpy
```

te permitirá acceder a toda su funcionalidad desde tu código. El siguiente código usa la librería de Numpy para calcular el valor medio de una lista de números:

Python

```
import numpy

numpy.mean([1,2,3])
```

Devuelve: *2.0*

Esta información nos permite entender las primeras líneas de código de nuestro ejemplo, en las cuales importamos la librería numpy y le asignamos el alias `np`, y de la librería *scikit-learn* importamos la función `fetch_openml`, que nos permite descargar los datos que usaremos para entrenar el modelo.

Python

```
from sklearn.datasets import fetch_openml
import numpy as np
```

Más adelante ahondaremos en este aspecto, pero antes necesitamos aprender los conceptos básicos de Python.

Conceptos básicos

Uno de los primeros conceptos que debes conocer de Python es que todo son **Objetos**. Esto es un tipo de dato especial de Python y que es el principal responsable de su gran flexibilidad. Como en Python todo son **Objetos** vamos a poder, por ejemplo, enviar cualquier variable como parámetro de una función (no te preocupes si no sabes lo que es una función, ya lo veremos más adelante) sin que el interpretador se queje. Esto difiere de otros lenguajes en los que si no utilizamos los tipos adecuados en cada momento, el programa no funciona. En Python, sin embargo, solo obtendremos errores si llevamos a cabo operaciones con esos **Objetos** que no son válidos (por ejemplo, sumar un número con una letra). Esto puede dar como resultado errores difíciles de detectar, pero es que un gran poder conlleva una gran responsabilidad.

Comentarios

Los comentarios son una parte fundamental de cualquier programa, y están presentes de una manera u otra en todos los lenguajes de programación. Nos permiten añadir texto que será ignorado por el interpretador (o compilador), por lo que son ideales para explicar qué es lo que hace el código y por qué lo hace, documentarlo o incluso dejar «easter eggs» para otros desarrolladores. En Python, los comentarios se escriben usando el símbolo `#`. Todo lo que escribamos a continuación de este símbolo será ignorado por el interpretador.

Python

```
# esto es un comentario
```

Siguiendo con nuestro ejemplo, después de importar varias librerías, podemos encontrar el siguiente comentario:

Python

```
# descarga de datos
```

De nuevo, este texto será completamente ignorado por el interpretador y simplemente nos sirve para organizar y estructurar mejor nuestro código, a la vez que un tercero que lea nuestro código pueda entender mejor qué es lo que hacen las siguientes líneas.

Variables

Lo siguiente que encontramos en el ejemplo es:

Python

```
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

¡Uf, aquí están pasando muchas cosas! Vamos a ir por partes.

El siguiente concepto básico, que también encontrarás en prácticamente todos los lenguajes de programación, son las variables. Una variable no es más que una **referencia a un objeto en memoria**. El siguiente código crea una variable llamada `a` con el valor `Hola`.

Python

```
a = 'Hola'
```

Como puedes ver, para crear una variable simplemente tenemos que escribir su nombre y asignarle un valor usando el operador `=`. En Python no es necesario especificar el tipo de dato de la variable, ya que el interpretador lo deduce automáticamente. Esto es lo que se conoce como **tipado dinámico**. En otros lenguajes, como C o C++, es necesario especificar el tipo de dato de la variable al crearla. Esto es lo que se conoce como **tipado estático**.

Así pues, en nuestro ejemplo, estamos creando 3 variables: mnist, X e y. La primera, mnist, contendrá el resultado de la ejecución de la función `fetch_openml`, que descarga los datos de la base de datos MNIST como un `DataFrame` de `Pandas` (más adelante veremos qué es una función). Las otras dos variables, X e y, hacen referencia al campo `data` y `target` del `DataFrame` que acabamos de descargar, los cuales son convertidos en `arrays` de `Numpy` con el método `values` y convertidos a los tipos de datos adecuados usando el método `astype`. Como puedes ver, en simplemente dos líneas de código estamos haciendo muchas cosas en las que intervienen hasta 3 librerías diferentes. Esto es posible gracias a la gran flexibilidad de Python y su ecosistema de librerías, aunque pueda ser un poco confuso al principio.

Vamos a detenernos en este punto para entender cómo podemos representar diferentes tipos de datos en un ordenador para poder llevar a cabo diferentes tareas con ellos.

Representación de datos

Como hemos visto programar no es más que definir una secuencia de instrucciones que un ordenador debe llevar a cabo. Pero, ¿cómo podemos definir estas instrucciones? Cuando las personas nos comunicamos entre nosotros utilizamos el lenguaje natural para intercambiar información. De esta manera, si es la primera vez que quieres cocinar una paella y quieres que te salga medio bien, seguirás unas instrucciones escritas en una receta. Primero, pondrás a calentar el aceite en la paella, después añadirás el pollo, luego el conejo, después el tomate, etc. De esta manera, si sigues las instrucciones al pie de la letra, conseguirás tu objetivo. En este caso, el lenguaje natural permite traducir conceptos en las acciones necesarias para resolver el problema. Un ordenador, sin embargo, no entiende el lenguaje natural. Un ordenador solo habla el lenguaje de unos y ceros.

> Nota: Los avances en el campo del Procesamiento del Lenguaje Natural (NLP) están permitiendo una comunicación entre personas y máquinas sin precedentes, dando como resultado que perfiles no técnicos sean capaces de llevar a cabo tareas que tradicionalmente necesitaban de conocimientos de programación simplemente usando el lenguaje natural para especificar los pasos a seguir. Sin embargo, hay que distinguir entre este tipo de programación (si es que podemos llamarlo así) y lo que el ordenador es capaz de ejecutar en su *hardware*.

En palabras de Andrej Karpathy:

A la hora de buscar una manera de representar información en un ordenador, los números son la opción ideal. Podemos usar números para representar texto (por ejemplo, usando el código ASCII), imágenes (por ejemplo, usando el código RGB), sonido, etc. De esta manera, un ordenador puede almacenar y procesar cualquier tipo de información (incluyendo números, obviamente). ¡Perfecto! Y ahora, ¿cómo representamos números en un ordenador? Si bien los humanos usamos un sistema numérico en base 10 (debido a que tenemos dos manos con 10 dedos y contar más allá de 10 se les hizo complicado a nuestros antepasados) los ordenadores usan un sistema numérico en base 2. Esto es debido a que un ordenador funciona con electricidad, y los dos únicos estados que podemos representar con electricidad es o bien hay electricidad (1) o bien no la hay (0). Piensa, por ejemplo, en una bombilla que puede estar encendida o apagada.

Pero nosotros podemos contar más allá de 10, y los ordenadores son capaces de procesar ingentes cantidades de información, por lo que probablemente puedan contar más allá de 2. Cierto. Para entender cómo un ordenador es capaz de representar más información tenemos que entender primero cómo lo hacemos nosotros. Si bien en nuestro sistema en base 10 disponemos de los dígitos del 0 al 9, ¿qué ocurre cuando queremos representar, por ejemplo, el número 4269 (cuatro mil doscientos sesenta y nueve)?. En primer lugar, cogemos el dígito que está más a la izquierda, el 4, y lo multiplicamos por la base, 10, elevada a la posición en la que se encuentra el dígito, empezando a contar desde la derecha con la posición 0. Repitiendo el proceso para todos los dígitos obtenemos como resultado $4 \cdot 10^3 + 2 \cdot 10^2 + 6 \cdot 10^1 + 9 \cdot 10^0 = 4000 + 200 + 60 + 9 = 4269$. Esto es un proceso que las personas llevamos a cabo de manera inconsciente y nos permite, sin darnos cuenta, representar cualquier número en base 10.

> Te reto a que apliques la misma lógica para descubrir qué números (en base 10) se representan con los números 000, 001 y 101 en base 2.

En la jerga, se conoce como *bit* a cada uno de los dígitos que componen un número en base 2. Así pues, el número 101 se compone de 3 bits.

Por dentro, un ordenador está compuesto por miles de millones de estas *bombillas*, llamadas transistores, capaces de representar y procesar muchísimos *bits* de información. Este es el fundamento utilizado por todos los diferentes tipos de ordenadores que encontramos a nuestro alrededor, desde los grandes centros de datos llenos de servidores del tamaño de neveras repletos de procesadores (también conocido como *la nube*) hasta los pequeños ordenadores que llevamos en el bolsillo en nuestros teléfonos móviles, pasando por los

ordenadores de sobremesa, portátiles e incluso los que encontramos en coches, electrodomésticos y otros dispositivos de uso cotidiano. Y es que hoy en día podemos encontrar ordenadores en todas partes, procesando estos unos y ceros a gran velocidad.

Puedes aprender más sobre el funcionamiento de los ordenadores y sus diferentes componentes (CPUs, discos duros, memorias RAM, GPUs, …) en el siguiente video.

Representación de texto

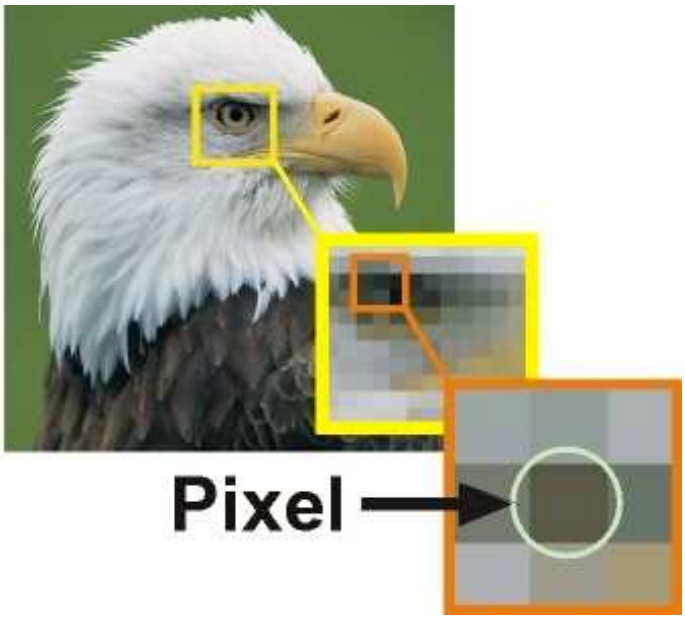
Un ordenador es capaz de representar información de tipo texto asignando, por ejemplo, un número a cada letra. De hecho, en esta idea se basan precisamente estándares como el ASCII (que usa 8 bits para representar letras y caracteres) o el Unicode (que tiene variantes de 8, 16 y 32 bits capaces de representar caracteres en cualquier idioma, incluida nuestra querida Ñ, e incluso también emojis).

Por encima de estos estándares podemos encontrar diferentes formatos de archivo que utilizan estos códigos (además de otros metadatos) para representar texto. Algunos ejemplos son los formatos PDF, DOCX, HTML, XML, JSON, CSV, etc. Simplemente, son diferentes formas de organizar los *bits*, pero que debemos conocer de antemano para poder interpretarlos correctamente (por eso no puedes abrir un pdf en Word, por ejemplo).

Representación de imágenes y vídeo

De la misma manera que podemos usar números para representar letras y símbolos, podemos hacer lo mismo para representar colores. Existen muchas formas distintas de representar colores, de la misma manera que existen diferentes estándares para representar texto. Una de ellas es el formato RGB, que permite definir cualquier color mediante la combinación de los colores rojo, azul y verde (de ahí las siglas *Red Green Blue* en inglés). Así pues, podremos representar cualquier color mediante 3 números de 8 *bits* correspondientes a la intensidad de cada componente.

Sin embargo, para representar imágenes necesitamos más de un color. Es aquí donde entra el concepto del *pixel*, y es que una imagen no es más que una cuadrícula de píxeles en el que cada uno tiene un color asignado. Cuanto mayor densidad tenga la cuadrícula, más píxeles y más definida será la imagen (decimos que tiene más resolución). En el caso contrario, a menor cantidad de píxeles, la imagen será más borrosa y *pixelada*.



Y como un vídeo no es más que imágenes en movimiento, podremos representar vídeos mediante conjuntos de imágenes añadiendo la dimensión temporal, concepto conocido como *frame rate* o tasa de fotogramas que indica cuántas imágenes se muestran por segundo (a mayor tasa de fotogramas, más fluido será el movimiento).

De nuevo, podemos encontrar múltiples formatos que utilizan estos conceptos, pero los ordenan de diferente manera, añadiendo otros metadatos, dando lugar a formatos como JPG, PNG, GIF, MP4, AVI, etc. Es posible que hayas intentado abrir un archivo mp4, por ejemplo, en un reproductor de vídeo y no haya funcionado. Esto es debido a que el reproductor no ha sido capaz de interpretar correctamente el formato del archivo, el orden en el que se encuentran los *bits* y los metadatos que contiene.

Volviendo al ejemplo

Ahora que ya sabemos cómo es posible que un ordenador represente datos, cuando nosotros creamos una variable lo que realmente está pasando por detrás es que un objeto es creado en la memoria del ordenador. Una vez el objeto se ha creado, el sistema operativo le dice a Python la dirección de memoria en la que se encuentra. Este es el valor que guarda la variable `a`, por lo que nos sirve como referencia para acceder al objeto.

Python

```
id(a) # esta es la dirección de memoria de la variable
```

Devuelve: *4417744688*

Puedes imaginar la memoria de un ordenador como una estantería en la que, en cada hueco, podemos guardar un bit de información (0 o 1). Al crear una variable, el sistema operativo buscará los huecos libres necesarios para guardar el objeto y le dirá a Python el identificador del primer hueco usado (la dirección de memoria en la que se encuentra). Si no quedan más huecos libres, el sistema operativo no podrá crear el objeto y nos devolverá un error.

Las variables en Python son **referencias dinámicas**. Esto significa que podemos cambiar el valor de una variable en cualquier momento simplemente asignándole un nuevo valor. En lenguajes de bajo nivel, como C o C++, las variables son **referencias estáticas**, por lo que una vez se crea una variable, no se puede cambiar su valor. Esto también se conoce como **mutabilidad**. En Python, la mayoría de tipos de datos son mutables, mientras que en C o C++ la mayoría son inmutables. Esto es una de las principales diferencias entre ambos lenguajes.

Python

```
a = 1  # ahora `a` es un número
```

Como puedes imaginar, en programas grandes con muchas líneas de código, tendremos muchísimas variables. Por este motivo, es importante que las nombremos de manera que nos sea fácil identificarlas y saber qué contienen. Existen varias convenciones a la hora de nombrar variables en Python que es aconsejable que sigas, ya que harán tu código más sostenible y legible por otros programadores. Algunas de estas convenciones son:

- Empezar el nombre de la variable con una letra minúscula (a excepción de las constantes, que se escriben en mayúsculas).
- Usar nombres descriptivos (si quieres guardar un *email*, usa el nombre `email` en vez de `e` o cualquier otro nombre poco descriptivo).
- Si usas varias palabras para nombrar una variable, separarlas con guiones bajos (lo que se conoce como `snake_case`).

Python

```
# snake_case (recomendado)
una_variable = 1


# CamelCase (no recomendado)
unaVariable = 1
```

Tipos básicos

En Python tenemos varios tipos de datos por defecto:

- Números enteros (`int`)
- Números decimales (`float`)
- Cadena de caracteres, también conocidas como *strings* (`str`)
- Booleanos (`bool`), que únicamente pueden tomar los valores `True` o `False`
- Cadena de caracteres en formato binario (`bytes`)
- Valores inexistentes (`None`)

Python

```
a = 1

type(a) # tipo de variable
```

Es de tipo: *int*

Para definir números decimales usamos el punto (`.`) en vez de la coma (`,`). Si usamos la coma, Python interpretará que estamos definiendo dos números enteros.

Python

```
a = 1.2

type(a)
```

Es de tipo: *float*

Python

```
a = 1,2 # esto NO es un número decimal, es una tupla (luego lo veremos)
isinstance(a, float) # ¿es `a` un número decimal?
```

Devuelve: *False*

Python

```
a = 'Hola'
type(a)
```

Es de tipo: *str*

Python

```
a = True
type(a)
```

Es de tipo: *bool*

Python

```
a = None
type(a)
```

Es de tipo: *NoneType*

Strings

Python es un lenguaje muy potente a la hora de trabajar con cadenas de caracteres. Vamos a ver algunas de las operaciones más comunes que podemos llevar a cabo con ellas. Para empezar, podemos definir *strings* usando las comillas simples o dobles.

Python

```
a = 'hola' # válido
a = "hola" # también es válido
```

Incluso podemos usar triples comillas dobles para definir cadenas de texto multilínea respetando el formateado.

Python

```
a = """

hola

        que tal

"""

print(a)
```

Devuelve:

hola

que tal

Puedes usar el operador + para concatenar cadenas de texto, generando una nueva cadena de texto con el resultado de la concatenación.

Python

```
a = "hola"
b = "¿qué tal?"

c = a + ", " +  b # concatenación de strings

c
```

c tiene el valor: *hola, ¿qué tal?*

Puedes saber el número de caracteres de una cadena de texto usando la función `len`.

Python

```
len(a), len(b), len(c)
```

Devuelve: *4, 9, 15*

Puedes acceder a un carácter concreto de una cadena de texto usando corchetes (`[]`) con la posición del carácter. Ten en cuenta que en Python los índices empiezan en 0.

Python

```
a[0], a[1], a[2], a[3]
```

Devuelve: *h, o, l, a*

Python

```
a[4]
```

Devuelve un error:

IndexError Traceback (most recent call last)

/Users/sensio/Desktop/libro/03_python.ipynb Cell 65 in 1 —-> 1 a[4]

IndexError: string index out of range

Puedes acceder a un rango de caracteres de una cadena de texto usando corchetes con el rango de posiciones separadas por dos puntos. Ten en cuenta que el primer índice es inclusivo y el segundo exclusivo. A esto se le llama *slicing*, y lo veremos en más detalle cuando veamos `listas`.

Python

```
a[:3], a[1:], a[1:3]
```

Devuelve: *hol, ola, ol*

Puedes repetir una cadena de texto usando el operador `*`.

Python

```
a*3
```

Devuelve: *holaholahola*

Puedes separar un *string* en una lista de *strings* usando el método `split`, pasándole como parámetro el carácter por el que quieres separar la cadena de texto.

Python

```
d = c.split(',') # separar por comas

d
```

d tiene el valor: *['hola', ' ¿qué tal?']*

También puedes reemplazar un carácter por otro usando el método `replace`, pasándole como parámetro el carácter que quieres reemplazar y el carácter por el que lo quieres reemplazar.

Python

```
d = c.replace('tal', 'ase')

d
```

d tiene el valor: *hola, ¿qué ase?*

Esto es solo una pequeña muestra de lo mucho que Python nos ofrece por defecto a la hora de manipular cadenas de texto. Si quieres saber más, puedes consultar la documentación oficial.

Operadores

Si bien hemos visto un primer ejemplo de cómo usar operadores con cadenas de texto (por ejemplo, el operador `+` sirve para concatenar cadenas), Python nos ofrece muchos más operadores que podemos usar con otros tipos de datos, sobre todo de tipo numérico para llevar a cabo operaciones algebraicas. Los operadores más comunes son:

- `+` para sumar
- `-` para restar
- `*` para multiplicar
- `/` para dividir
- `//` para dividir y quedarnos con la parte entera

- `**` para elevar un número a otro
- `%` para obtener el resto de una división (operador módulo)

Python

```
1 + 1
```

Devuelve: *2*

Python

```
2 - 1
```

Devuelve: *1*

Python

```
2 * 3
```

Devuelve: *6*

Python

```
1 / 2
```

Devuelve: *0.5*

Python

```
3 // 2
```

Devuelve: *1*

Python

```
2 ** 3
```

Devuelve: *8*

Python

```
3 % 2
```

Devuelve: *1*

También podemos combinar estos operadores con la asignación (=) para atribuir el resultado de una operación a una variable.

Python

```
a = 1
a += 1 # es lo mismo que hacer a = a + 1
a
```

a tiene el valor: *2*

Condicionales

Es muy común en cualquier programa que necesitemos ejecutar un bloque de código u otro en función de una condición. Para ello, disponemos de los operadores de comparación:

- `and` para comprobar que se cumplen dos condiciones
- `or` para comprobar que se cumple una de dos condiciones
- `not` para negar una condición
- $=$ para comprobar si dos valores son iguales
- $\neq$ para comprobar si dos valores son distintos
- $<$ para comprobar si un valor es menor que otro
- $<=$ para comprobar si un valor es menor o igual que otro
- $>$ para comprobar si un valor es mayor que otro
- $>=$ para comprobar si un valor es mayor o igual que otro

Python

```
True and False
```

Devuelve: *False*

Python

```
True or False
```

Devuelve: *True*

Python

```
not True or False
```

Devuelve: *False*

Python

```
1 == 2
```

Devuelve: *False*

Python

```
1 != 2
```

Devuelve: *True*

Python

```
1 < 1
```

Devuelve: *False*

Python

```
1 <= 1
```

Devuelve: *True*

Python

```
3 > 2
```

Devuelve: *True*

Python

```
3 >= 2
```

Devuelve: *True*

Combinando estos operadores con los condicionales `if`, `elif` y `else`, podemos ejecutar un bloque de código u otro en función de una condición.

Python

```python
x = 1

if x < 0:
    print('negativo')
elif x == 0:
    print('cero')
else:
    print('positivo')
```

Devuelve: *positivo*

Como puedes ver en el ejemplo usaremos el condicional `if` seguido de una condición. Si la condición se cumple, se ejecutará el bloque de código que está definido justo debajo. Si no, pasará a comprobar la siguiente condición definida con `elif`. Si ninguna de las condiciones se cumple, se ejecutará el bloque de código definido con `else`. Tanto `elif` como `else` son opcionales, y podemos tener tantos `elif` como queramos. Es importante tener en mente que en cuanto se cumpla una condición, se ejecutará el bloque de código correspondiente y el resto de condiciones no se comprobarán (por lo que es importante definir las condiciones en el orden correcto).

Python

```python
x = 1

if x < 0: # válido, pero no pasará nada
    print('negativo')
```

No devuelve nada

Python

```python
x = 1

if x < 0: # haz una cosa o la otra
    print('negativo')
else:
    print('positivo')
```

Devuelve: *positivo*

Python

```
x = 2

if x < 0:
    print('negativo')
elif x > 0 and x <= 1:
    print('uno')
elif x > 1 and x <= 2:
    print('dos')
elif x > 2 and x <= 3:
    print('tres')
else:
    print('positivo')
```

Devuelve: *dos*

Aquí es muy importante el uso de la indentación para definir los bloques de código. En Python, los bloques de código se definen con indentación, es decir, con espacios en blanco. Esto es muy importante, ya que si no respetamos la indentación, Python nos dará un error. Otros lenguajes de programación, como C o C++, usan llaves (`{}`) para definir los bloques de código.

Python

```
x = 1

if x < 0:
    print('negativo')
else:
print('positivo') # error de indentación
```

Devuelve un error:

Cell In [69], line 6

print('positivo') # error de indentación

^

IndentationError: expected an indented block

Bucles

Si bien en la sección anterior hemos visto cómo podemos ejecutar un bloque de código u otro en función de si se cumple una condición, o no, también es muy común que necesitemos ejecutar un bloque de código varias veces. Para ello, disponemos de los bucles `for` y `while`.

Python

```
for i in range(3):
    print(i)
```

Devuelve: *0 1 2*

> La función `range` nos permite generar una lista de números enteros en un rango determinado. Si le pasamos un único parámetro, generará una lista de números enteros desde el 0 hasta el número que le pasemos (sin incluirlo). Si le pasamos dos parámetros, generará una lista de números enteros desde el primer número hasta el segundo (sin incluirlo). Si le pasamos tres parámetros, generará una lista de números enteros desde el primer número hasta el segundo (sin incluirlo), incrementando el valor en el tercer parámetro.

Python

```
for i in range(2,6):
    print(i)
```

Devuelve: *2 3 4 5*

Python

```
for i in range(2,10,2):
    print(i)
```

Devuelve: *2 4 6 8*

Podemos combinar los iteradores con las palabras reservadas `continue` y `break` para controlar el flujo de ejecución de un bucle. `continue` nos permite saltarnos la ejecución del

resto del bloque de código y pasar a la siguiente iteración del bucle, mientras que `break` nos permite salir del bucle.

Python

```python
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Devuelve: *1 3 5 7 9*

Python

```python
for i in range(10):
    if i > 5:
        break
    print(i)
```

Devuelve: *0 1 2 3 4 5*

Python

```python
i = 0
while i < 3:
    print(i)
    i += 1
```

Devuelve: *0 1 2*

El uso de bucles infinitos es muy común en Python:

Python

```python
i = 0
while True:
    print(i)
    i += 1
    if i >= 3:
        break
```

Devuelve: *0 1 2*

Si usas bucles infinitos recuerda implementar una condición de salida, si no, como su nombre indica, el bucle se ejecutará infinitamente y tu programa se quedará colgado.

Volviendo al ejemplo de nuestra red neuronal, cuando hacemos lo siguiente:

Python

```
for epoch in range(10):
    for b in tqdm(range(num_batches)):
        x = X_train[b*bs:(b+1)*bs]
        y = y_train[b*bs:(b+1)*bs]
        y_hat = model(x)
        loss = loss_fn(y_hat, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} loss: {loss.item():.3f}")
```

Estamos usando un bucle doble, en el que iteramos por un número determinado de *epochs* y, por cada *epoch*, iteramos por un número determinado de *batches*. En cada iteración, ejecutamos un bloque de código que entrena nuestro modelo (en el cual entraremos en detalle en los siguientes capítulos).

Estructuras de datos secuenciales

Las estructuras de datos secuenciales nos permiten almacenar colecciones de datos, con los que luego podemos trabajar de manera conjunta. Ya hemos visto un primer ejemplo, el `range`. Sin embargo, las más comunes son las listas, `list`, las tuplas, `tuple`, los diccionarios, `dict`, y los sets, `set`. Cada una de estas estructuras de datos tiene sus puntos fuertes y débiles, y están diseñadas para diferentes tipos de tareas y condiciones.

Por un lado, la tupla es un objeto **inmutable** de longitud fija, por lo que lo usaremos cuando queramos estar seguros de que lo que hay dentro de la tupla no debe cambiar bajo ningún concepto. Podemos crear una tupla usando paréntesis.

Python

```
t = (1, 2, 3)
t
```

t tiene el valor: *1, 2, 3*

Para acceder a los valores dentro de la secuencia, utilizamos los corchetes `[]` y el índice del valor que queremos acceder. Los índices empiezan en el 0, por lo que el primer valor de la secuencia tendrá el índice 0, el segundo el índice 1, y así sucesivamente.

Python

```
t[0], t[2], t[1]
```

Devuelve: *1, 3, 2*

> Cuando trabajamos con *notebooks*, podemos mostrar el valor de una variable sin necesidad de usar la función `print`. Simplemente escribiendo el nombre de la variable, el *notebook* mostrará su valor. Si usamos varios valores separados por comas, el resultado se mostrará en forma de tupla.

Podemos conocer la cantidad de valores almacenados en la secuencia usando la función `len`.

Python

```
len(t)
```

Devuelve: *3*

Las siguientes operaciones nos permiten concatenar secuencias, repetirlas y extraer sus valores de forma independiente, respectivamente.

Python

```
t2 = (4, 5, 6)

t + t2 # concatenación de tuplas
```

t tiene el valor: *(1, 2, 3, 4, 5, 6)*

Python

```
t*2 # repetición de tuplas
```

t tiene el valor: *(1, 2, 3, 1, 2, 3)*

Python

```
a, b, c = t # desempaquetado de tuplas

c, b, a
```

Devuelve: *3, 2, 1*

Una de las ventajas que ofrece Python es que podemos tener valores de diferentes tipos dentro de una misma tupla.

Python

```
t = (1, "hola", True)
```

Y si queremos iterar por los valores de una secuencia, podemos usar un bucle `for` como ya hemos visto anteriormente.

Python

```
for item in t:
    print(item)
```

Devuelve: *1 hola True*

Una vez definida la tupla no vamos a poder modificarla, para ello deberemos usar una lista.

Python

```
t[0] = 2 # error, las tuplas son inmutables
```

Devuelve un error:

TypeError Traceback (most recent call last)

Cell In[27], line 1 —-> 1 t[0] = 2 # error, las tuplas son inmutables

TypeError: ‘tuple’ object does not support item assignment

Las listas son muy similares a las tuplas, excepto en el hecho de que sí pueden ser modificadas, son **mutables**. Podemos crear una lista usando corchetes `[]`.

Python

```
l = [1, 2, 3]
l
```

l tiene el valor: *[1, 2, 3]*

Y todo lo visto anteriormente para las tuplas, también se aplica a las listas (indexado, concatenación, repetición, iteración, etc.).

Python

```
l[0], len(l), l*2
```

Devuelve: *1, 3, [1, 2, 3, 1, 2, 3]*

Para añadir valores a una lista, podemos usar los métodos `append` o `insert`, mientras que los métodos `remove` y `pop` nos permiten eliminar valores de una lista.

Python

```
l.append(4) # añadir un elemento al final de la lista
l
```

l tiene el valor: *[1, 2, 3, 4]*

Python

```
l.insert(2, -1) # añadir un elemento en la posición 2
l
```

l tiene el valor: *[1, 2, -1, 3, 4]*

Python

```
l.pop(2) # eliminar el elemento en la posición 2
l
```

l tiene el valor: *[1, 2, 3, 4]*

Python

```
l.remove(4) # eliminar el elemento 4 (si existe)
l
```

l tiene el valor: *[1, 2, 3]*

Podemos modificar valores de una lista usando el operador de asignación = y el índice del valor que queremos modificar.

Python

```
l[1] = 0 # modificar el elemento en la posición 1
l
```

l tiene el valor: *[1, 0, 3]*

Para comprabar si un valor está dentro de una lista, podemos usar el operador `in`.

Python

```
2 in l
```

Devuelve: *False*

Python

```
2 not in l
```

Devuelve: *True*

Una característica muy potente de Python a la hora de trabajar con listas es el troceado, *slicing* en inglés. Esto nos permite extraer una parte de la lista, o incluso invertirla.

Python

```
l = [1, 2, 3, 4, 5]

l[1:3] # extraer los elementos entre las posiciones 1 y 3
```

l tiene el valor: *[2, 3]*

Python

```
l[:2] # extraer los elementos hasta la posición 2
```

l tiene el valor: *[1, 2]*

Python

```
l[2:] # extraer los elementos desde la posición 2 hasta el final
```

l tiene el valor: *[3, 4, 5]*

Python

```
l[-1] # extraer el último elemento
```

l tiene el valor: *5*

> Puedes usar índices negativos para acceder a los valores de una lista desde el final.

Python

```
l[-2:] # extraer los dos últimos elementos
```

l tiene el valor: *[4, 5]*

Python

```
l[::2] # extraer los elementos cada dos posiciones
```

l tiene el valor: *[1, 3, 5]*

Python

```
l[::-1] # extraer los elementos en orden inverso
```

l tiene el valor: *[5, 4, 3, 2, 1]*

En nuestro ejemplo de red neuronal, al entrenar el modelo le vamos dando lotes de ejemplos. Para ello, usamos el indexado para extraer los valores adecuados de las listas de datos.

Python

```
bs = 32

# ...
for b in tqdm(range(num_batches)):
    x = X_train[b*bs:(b+1)*bs]
    y = y_train[b*bs:(b+1)*bs]
```

En la primera iteración ($b = 0$), extraemos los valores desde el índice 0 hasta el índice 32. En la segunda iteración ($b = 1$), extraemos los valores desde el índice 32 hasta el índice 64. Y así sucesivamente hasta que hayamos extraído todos los valores de la lista.

Otra funcionalidad imprescindible a la hora de trabajar con listas son las funciones secuenciales. Con `enumerate` podemos obtener el índice y el valor de cada elemento de una lista.

Python

```
for i, v in enumerate(l):
    print(i, v)
```

Devuelve: *0 1 1 2 2 3 3 4 4 5*

Con `zip` podemos iterar por múltiples listas a la vez.

Python

```
l2 = [6, 7, 8, 9, 10]

for v1, v2 in zip(l, l2):
    print(v1, v2)
```

Devuelve: *1 6 2 7 3 8 4 9 5 10*

También podemos usar `reversed` para iterar por una lista en orden inverso.

Python

```
for v in reversed(l):
    print(v)
```

Devuelve: *5 4 3 2 1*

Las siguientes estructuras de datos que vamos a ver es, probablemente, la más potente y utilizada de Python, el diccionario. Un diccionario es una colección de pares clave-valor, donde cada clave es única y no puede ser modificada, mientras que los valores pueden ser de cualquier tipo. Podemos crear un diccionario usando llaves `{}`.

Python

```
d = {'a': 1, 'b': 2, 'c': 3} # diccionario: pares clave-valor
d
```

d tiene el valor: *{'a': 1, 'b': 2, 'c': 3}*

Podemos acceder a los valores de un diccionario usando la clave entre corchetes `[]`.

Python

```
d['a']
```

Devuelve: *1*

Los cuales pueden ser modificados.

Python

```
d['a'] = 4

d
```

d tiene el valor: *{'a': 4, 'b': 2, 'c': 3}*

Podemos añadir nuevos pares clave-valor a un diccionario usando el operador de asignación `=`.

Python

```
d['d'] = 5

d
```

d tiene el valor: *{'a': 4, 'b': 2, 'c': 3, 'd': 5}*

Podemos eliminar valores de un diccionario usando la palabra reservada **`del`**.

Python

```
del d['a'] # eliminar un elemento del diccionario

d
```

d tiene el valor: *{'b': 2, 'c': 3, 'd': 5}*

Para saber si una clave está dentro de un diccionario, podemos usar el operador `in`.

Python

```
"a" in d, "b" in d, "c" in d
```

Devuelve: *False, True, True*

Extraer el conjunto de claves y valores es muy sencillo.

Python

```
k = d.keys()

k
```

Devuelve: *dict_keys(['b', 'c', 'd'])*

Python

```
v = d.values()

v
```

Devuelve: *dict_values([2, 3, 5])*

Y podemos usar el método `update` para añadir nuevos valores a un diccionario.

Python

```
d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}

d1.update(d2) # añadir los elementos de d2 a d1

d1
```

d1 tiene el valor: *{'a': 1, 'b': 2, 'c': 3, 'd': 4}*

La última estructura de datos secuencial que veremos son los sets, que son conjuntos de elementos únicos. Podemos crear un `set` usando llaves `{}` con los valores separados por comas.

Python

```
s = {1, 2, 3}

s
```

s tiene el valor: *{1, 2, 3}*

Podemos calcular la unión, la intersección, o saber si un set es subconjunto usando las siguientes funciones.

Python

```
a = {1, 2, 3}
b = {3, 4, 5}

a.union(b)
```

Devuelve: *{1, 2, 3, 4, 5}*

Python

```
a.intersection(b)
```

Devuelve: *{3}*

Python

```
{1, 2}.issubset(a)
```

Devuelve: *True*

Python

```
{1, 2}.issubset(b)
```

Devuelve: *False*

Aunque probablemente el uso más común de los sets sea eliminar duplicados de una lista.

Python

```
l = [1, 2, 3, 4, 5, 5, 5, 6, 7, 8, 8]
s = set(l) # Elimina los duplicados
l = list(s) # Convierte el set en lista
l
```

l tiene el valor: *[1, 2, 3, 4, 5, 6, 7, 8]*

Manejando errores

Como has podido observar en algunos de los ejemplos anteriores, Python nos devuelve errores cuando algo falla. Por ejemplo, cuando intentamos acceder a un índice que no existe en una lista.

Python

```
l = [1,2,3]

l[4]

print("el programa sigue") # no se ejecuta
```

Devuelve un error:

IndexError Traceback (most recent call last)

/Users/sensio/Desktop/libro/03_python.ipynb Cell 207 in 3 1 l = [1,2,3] —-> 3 l[4] 5 print(“el programa sigue”)

IndexError: list index out of range

Si bien este comportamiento es útil mientras desarrollamos código, a la hora de poner un programa en producción no es deseable que si algo falla, el programa se detenga. Para ello, podemos usar bloques `try` y `except`, que nos ayudarán a hacer nuestro código más robusto y tolerante a errores, respondiendo a estos de la manera en la que queramos evitando que nuestro programa se «cuelgue».

Dentro del bloque `try` pondremos el código que queremos ejecutar. Si algo falla, se ejecutará el bloque `except` correspondiente.

Python

```
try:
    l[4]
except:
    print("Parece que te has salido de rango")

print("el programa sigue")
```

Devuelve: *Parece que te has salido de rango el programa sigue*

Podemos hacer que nuestro programa responda a diversos errores de diferente manera:

Python

```
try:
    1 + "hola" # comenta para ir al primer except
    l[4]
except IndexError:
    print("Parece que te has salido de rango")
except:
    print("Cualquier otro error")

print("el programa sigue")
```

Devuelve: *Cualquier otro error el programa sigue*

Por último, puedes extraer el mensaje de error de la siguiente manera:

Python

```
try:
    1 + "hola" # comenta para ir al primer except
    l[4]
except IndexError:
    print("Parece que te has salido de rango")
except Exception as e:
    print(e)

print("el programa sigue")
```

Devuelve: *unsupported operand type(s) for +: 'int' and 'str' el programa sigue*

E incluso lanzar tus propios errores usando la palabra reservada `raise`.

Python

```
try:
    raise Exception("Mi excepción")
except Exception as e:
    print(e)
```

Devuelve: *Mi excepción*

Funciones

Una vez vistos los conceptos básicos de Python, vamos a aprender sobre una de las características más importantes que podemos encontrar en prácticamente cualquier lenguaje de programación: las funciones. Las funciones nos permiten encapsular código para poder reutilizarlo en diferentes partes de nuestro programa. Es una excelente manera de evitar repetir código, haciéndolo más fácil de mantener y de leer. Si te encuentras programando y te das cuenta de que estás repitiendo lo mismo en diferentes partes de tu programa, es una señal clara de que deberías crear una función. Toda función tiene dos fases: la definición (o declaración) y la invocación (o ejecución). En la definición de la función, especificamos el nombre de la función, los parámetros que recibe, y el código que se ejecutará cuando sea invocada. En la invocación de la función, usamos el nombre de la función seguido de paréntesis `()`, y dentro de los paréntesis los valores de los parámetros que queremos pasarle a la función.

En el siguiente ejemplo definimos una función con la palabra reservada `def` seguido del nombre que queremos darle a nuestra función y los parámetros que espera entre los paréntesis, en este caso ninguno. La palabra reservada `pass` nos permite definir una función vacía, que no hace nada.

Python

```
def f():
    pass
```

Ahora, invocamos la función:

Python

```
f()
```

Dentro de una función podemos implementar código en Python como hemos visto hasta ahora. Sin embargo, hay que tener algunos detalles en cuenta. El primero de ellos es que cualquier variable que esté definida en el cuerpo de una función, solo son accesibles dentro de la función. Sin embargo, las variables definidas fuera de la función (variables globales) sí son accesibles dentro de la función.

Python

```
def f():
    s = "hola"
    print(s)

f()
```

Devuelve: *hola*

Python

```
s # s no existe fuera de la función
```

Devuelve un error:

NameError Traceback (most recent call last)

Cell In[3], line 1 —-> 1 s # s no existe fuera de la función

NameError: name ‘s’ is not defined

Python

```
s = "hola"

def f():
    print(s) # s existe fuera de la función, se puede usar dentro también

f()
```

Devuelve: *hola*

Python

```
s
```

Devuelve: *hola*

Podemos enviar valores a una función usando parámetros. Los parámetros son variables que se definen en la definición de la función, y que se inicializan con los valores que se le pasan a la función en la invocación.

Python

```
def f(s):
    print(s)

f("hola")
```

Devuelve: *hola*

Python

```
f("qué tal")
```

Devuelve: *qué tal*

Es posible pasar múltiples parámetros a una función, separándolos por comas.

Python

```
def f(a, b, c):
    print(a, b, c)

f("hola", "qué", "tal")
```

Devuelve: *hola qué tal*

Por defecto, el orden de los parámetros es el mismo que el orden en el que los pasamos a la función. Sin embargo, podemos especificar el nombre de los parámetros en la invocación de la función, para que el orden no importe.

Python

```
f(c="tal", a="hola", b="qué")
```

Devuelve: *hola qué tal*

Para tener parámetros opcionales, podemos asignarles un valor por defecto en la definición de la función.

Python

```
def f(a, b, c="tal"):
    print(a, b, c)

f("hola", "qué")
```

Devuelve: *hola qué tal*

Python

```
f("hola", "qué", "ase")
```

Devuelve: *hola qué ase*

Por último, también podemos devolver valores desde una función usando la palabra reservada `return`. Esto nos permite guardar el resultado de una función en una variable que podemos seguir usando en nuestro programa.

Python

```
def f(a, b):
    return a + b

x = f(1, 2)

x
```

Devuelve: *3*

Podemos retornar múltiples valores de una función separándolos por comas, lo cual nos devolverá una tupla que podemos desempaquetar.

Python

```
def f(a, b):
    return a + b, a - b

x, y = f(1, 2)

x, y
```

Devuelve: *(3, -1)*

Existen varios tipos de funciones en Python, que pueden ser muy útiles en diferentes circunstancias. Un ejemplo son las funciones anónimas, que permiten definir funciones muy

simples en una sola línea. Para ello, usamos la palabra reservada `lambda` seguida de los parámetros y el código que queremos ejecutar.

Python

```
f = lambda a, b: a + b

f(1, 2)
```

Devuelve: *3*

Estas funciones son ideales cuando necesitamos definir una función muy simple que podemos pasar, por ejemplo, como argumento a otra función.

Como has podido comprobar a estas alturas, Python es un lenguaje muy flexible que nos permite hacer prácticamente cualquier cosa con nuestro código. En el caso de las funciones, podemos crear funciones que acepten cualquier cantidad de argumentos.

Python

```
def f(*args, **kwargs):
    print(args)
    print(kwargs)

f("hola", 1, True, a=2, b="hello")
```

Devuelve: *('hola', 1, True) {'a': 2, 'b': 'hello'}*

En este caso, todos los argumentos que pasemos a la función sin nombre, se agrupan en la tupla llamada `args`, mientras que los argumentos con nombre se agrupan en el diccionario llamado `kwargs`. Esto nos permite crear funciones muy flexibles que pueden aceptar cualquier cantidad de argumentos, lo cual es muy útil en funciones que pueden cambiar su implementación a menudo sin necesidad de modificar el código que las invoca, o funciones que llaman a otras funciones que esperan unos parámetros en concreto pero que no son usados por la primera. Sin embargo, esto requiere que tengamos mucho cuidado con la implementación de la función, ya que puede ser muy fácil cometer errores. Por lo general, es mejor evitar este tipo de funciones, y usar parámetros opcionales con valores por defecto.

El último tipo de función que veremos son los generadores. Estas funciones son muy útiles porque nos permiten crear nuestros propios iteradores. Para ello, crearemos una función que devuelva valores usando la palabra reservada `yield`. Cada vez que llamemos a la función, esta devolverá el siguiente valor del iterador, hasta que no haya más valores que devolver. El siguiente ejemplo es un generador que itera sobre los números pares.

Python

```python
def f(n=10):
    for i in range(n):
        if i % 2 == 0:
            yield i

gen = f()

for i in gen:
    print(i)
```

Devuelve: *0 2 4 6 8*

También puedes crear generadores con la sintaxis de `list comprehension`, usando los paréntesis en vez de los corchetes (para listas) o llaves (para diccionarios).

Python

```python
gen = (x for x in range(10) if x % 2 == 0)

for i in gen:
    print(i)
```

Devuelve: *0 2 4 6 8*

En nuestro ejemplo de red neuronal, hemos creado una función que se encarga de calcular la precisión de un modelo en un conjunto de datos determinado. Dados como parámetros de entrada el modelo y los datos, la función calcula la precisión del modelo en los datos y la devuelve. Esta función encapsula una funcionalidad determinada que podríamos reutilizar en diferentes partes de nuestro programa, o incluso en otros gracias a los módulos (que veremos en la siguiente sección).

Python

```python
def evaluate(model, X_test, y_test):
    acc = 0
    with torch.no_grad():
        for b in range(num_batches):
            x = X_test[b*bs:(b+1)*bs]
            y = y_test[b*bs:(b+1)*bs]
            y_hat = model(x)
            acc += torch.sum(torch.argmax(y_hat, dim=1) == y).item()
    return acc

acc = evaluate(model, X_test, y_test)
print(f"Accuracy: {acc} / {len(X_test)}")
```

Módulos

Si bien hasta ahora hemos hecho ejemplos pequeñitos en el que todo nuestro código está en el mismo archivo, cuando hacemos aplicaciones grandes con miles de líneas de código es muy común separarlas en diferentes archivos. Esto nos permite organizar mejor nuestro código y reutilizarlo en diferentes proyectos. En Python, cada archivo terminado en **.py** se considera un **módulo**. Podemos importar código desde un módulo en Python usando la palabra reservada `import`, igual que importamos librerías externas. Por ejemplo, la librería estándar de Python, contiene diversos módulos que vienen preinstalados por defecto con muchísima funcionalidad que podemos usar. Un ejemplo es el módulo `math` que contiene funciones matemáticas.

Python

```python
import math # importar un módulo con todas sus funciones

math.sqrt(2) # invocamos la función sqrt definida en el módulo math
```

Devuelve: *1.4142135623730951*

> Puedes aprender más sobre la librería estándar de Python en este post. También puedes consultar la documentación oficial. Esta librería contiene módulos para todo tipo de tareas, desde manejo de archivos, hasta generación de números aleatorios, pasando por matemáticas, criptografía, procesamiento de imágenes, computación paralela, etc.

La librería estándar contiene mucha funcionalidad, pero no toda. Por suerte, existen miles de librerías de terceros que podemos instalar y usar en nuestros proyectos. La comunidad de Python es una de las más grandes y activas, y existen módulos para prácticamente todo. Esto es una gran ventaja, ya que te ahorrará muchísimo tiempo al no tener que implementar de nuevo funcionalidad existente.

Es posible crear tus propios módulos que puedas importar desde otros archivos. Veamos un ejemplo.

Python

```
# Supongamos que tenemos un archivo module.py al lado de este notebook
    con el siguiente contenido:

#

# def f(x):

#   return 2*x
```

Python

```
import module

module.f(2)
```

Devuelve: *4*

Si no tienes el archivo `module.py` al mismo nivel, obtendrás un error de importación.

Python

```
import module2
```

Devuelve un error:

ModuleNotFoundError Traceback (most recent call last)

/Users/sensio/Desktop/libro/03_python.ipynb Cell 264 in 1 —-> 1 import module2

ModuleNotFoundError: No module named ‘module2’

Ya hemos visto que podemos asignar un alias a un módulo cuando lo importamos (útil para acortar el nombre o evitar colisiones entre módulos con el mismo nombre). También

podemos importar solo una parte del módulo, usando la palabra reservada `from` seguida del nombre del módulo, la palabra reservada `import` y el nombre de la función que queremos importar. Esto nos permite usar la función directamente sin tener que escribir el nombre del módulo.

Python

```python
from module import f # importar solo la función f del módulo module

f(2)
```

Devuelve: *4*

A esta también podemos darle un alias:

Python

```python
from module import f as f2 # importar solo la función f del módulo module
     con el alias f2

f2(2)
```

Devuelve: *4*

Y, por último, puedes importar todo el contenido de un módulo con el símbolo `*` (no recomendado, ya que puede causar colisiones entre funciones con el mismo nombre).

Python

```python
from module import *

f(2)
```

Devuelve: *4*

En nuestro ejemplo de red neuronal, hemos importado multitud de módulos de diferentes maneras:

Python

```python
from sklearn.datasets import fetch_openml
import numpy as np


# ...

from torch.nn import Sequential as S
from torch.nn import Linear as L
from torch.nn import ReLU as R


# ...

from tqdm import tqdm


# ...
```

Programación Orientada a Objetos

El último concepto que veremos en este capítulo es el de la programación orientada a objetos. Este paradigma de programación no es único de Python, puedes usarlo en muchos otros lenguajes de programación. A diferencia de la programación funcional, en la que el programa se compone de funciones que aceptan entradas y devuelven salidas, la programación orientada a objetos se basa en la creación de objetos, con sus métodos y atributos. Estos objetos son instanciados a partir de **clases**, que son como plantillas que definen la estructura de los objetos. De esta manera podemos encapsular la lógica de nuestro programa en objetos que interactúan entre sí.

Para crear una clase, usaremos la palabra reservada `class` seguida del nombre de la clase (normalmente en *CamelCase*).

Python

```python
class MiClase:
    a = "hola" # atributo

    def f(self): # método
        print(self.a)
```

Una clase está compuesta por métodos y atributos. Los métodos son funciones de la clase, mientras que los atributos son variables. Tanto los métodos como los atributos son solo accesibles por los objetos instanciados a partir de la clase, usando la sintaxis `objeto.metodo` o `objeto.atributo`.

Python

```
x = MiClase() # instanciar la clase

x.f()
```

Devuelve: *hola*

Como puedes ver, dentro de la clase usamos `self` para referirnos al objeto instanciado. Esto es necesario para poder acceder a los atributos y métodos del objeto.

Una vez instanciado el objeto, podemos modificar sus métodos y atributos libremente (no es recomendable, y está prohibido en la mayoría de lenguajes de programación, pero demuestra la flexibilidad de Python).

Python

```
x.a = "hello"

x.f()
```

Devuelve: *hello*

El Constructor

A la hora de definir clases existen métodos especiales que nos permiten controlar el comportamiento de los objetos instanciados a partir de la clase. Uno de estos métodos es el constructor, que se ejecuta automáticamente cuando instanciamos un objeto. El constructor se define con el nombre `__init__` y acepta como argumentos los atributos que queremos inicializar.

Python

```
class MiClase:
    def __init__(self, a):
        self.a = a

    def f(self):
        print(self.a)
```

Python

```
x = MiClase("hola")
x.f()
```

Devuelve: *hola*

Python

```
y = MiClase("hello")
y.f()
```

Devuelve: *hello*

Sobrecarga de Operadores

También podemos sobrecargar operadores para que funcionen con nuestros objetos. Por ejemplo, podemos sobrecargar el operador + de la siguiente manera:

Python

```
class MiClase:
    def __init__(self, a):
        self.a = a

    def __add__(self, other): # sobrecarga del operador +
        return self.a - other.a # aquí podríamos hacer lo que quisiéramos
```

Python

```
x = MiClase(1)
y = MiClase(2)

x + y # invoca a x.__add__(y), que en verdad está restando
```

Devuelve: *-1*

Un operador que es conveniente sobrecargar es el operador `__repr__`, que nos permite definir la representación en forma de *string* de nuestro objeto. Esto es útil para poder imprimir el objeto en la consola.

Python

```python
class MiClase:
    def __init__(self, a):
        self.a = a

    def __repr__(self):
        return f"El valor de `a` es {self.a}"
```

Python

```python
x = MiClase(1)

print(x)
```

a tiene el valor: *1*

Y otro operador muy interesante que podemos sobrecargar es el operador `__call__`, que nos permite llamar a un objeto como si fuera una función.

Python

```python
class MiClase:
    def __init__(self, a):
        self.a = a

    def __call__(self, b):
        return self.a ** b
```

Python

```python
x = MiClase(2)

x(3) # invoca a x.__call__(3)
```

Devuelve: *8*

Este es un patrón muy usado por las librerías de Inteligencia Artificial, que veremos más adelante, y que difumina la línea entre la programación funcional y la programación orien-

tada a objetos, usando lo mejor de ambos paradigmas.

Herencia

Cerramos este capítulo presentando el concepto de la herencia, que nos permite definir clases a partir de otras clases, heredando sus métodos y atributos, pudiendo añadir más o modificando los existentes. Este mecanismo es muy útil para la reutilización de código, ya que nos permite definir clases genéricas que luego podemos especializar.

En el siguiente ejemplo definimos a la clase base `MiAlgoritmoBase` con sus métodos y atributos. Luego, la clase `MiAlgoritmo` hereda de `MiAlgoritmoBase`, añadiendo un nuevo atributo y modificando un método. Para inicializar la clase madre desde la clase hija, usamos la función `super().__init__()`. Puedes acceder a cualquier método de la clase madre usando la sintaxis `super().metodo()`.

Python

```python
class MiAlgoritmoBase:
    def __init__(self, a):
        self.a = a

    def __call__(self):
        return 2*self.a

class MiAlgoritmo(MiAlgoritmoBase):
    def __init__(self, a, b):
        super().__init__(a) # invocamos al constructor de la clase base
        self.b = b

    def __call__(self): # modificamos el método __call__
        return 2*self.a + self.b # self.a existe porque lo hemos
            inicializado en la clase base
```

Python

```python
x = MiAlgoritmoBase(a=1)
x()
```

Devuelve: *2*

Python

```python
x = MiAlgoritmo(a=1, b=2)
x()
```

Devuelve: *4*

Existen libros enteros dedicados a la programación orientada a objetos. Aquí no hemos hecho más que rascar la superficie, pero los conceptos básicos presentados nos serán útiles en los siguientes capítulos a la hora de crear nuestras redes neuronales.

Nuestra primera red neuronal

Ahora que ya conoces el lenguaje de programación Python, deberías ser capaz de entender mejor el ejemplo presentado en el capítulo anterior de nuestra primera red neuronal. Antes de pasar al siguiente capítulo, vamos a diseccionarlo, añadiendo comentarios en las diferentes partes del código para entender qué está pasando.

Python

```
# Importamos la librería de numpy y la función `fetch_openml` de sklearn

from sklearn.datasets import fetch_openml
import numpy as np


# descargamos los datos y los asignamos a la variable `mnist`

mnist = fetch_openml('mnist_784', version=1)


# extraemos los datos y las etiquetas de los datos y los asignamos a las
    variables `X` e `y`

X, y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)


# De la librería `torch``, importamos varias clases

from torch.nn import Sequential as S
from torch.nn import Linear as L
from torch.nn import ReLU as R


# definimos el modelo y lo asignamos a la variable `model`

model = S(L(784,128),R(),L(128,10))


# importamos la librería `tqdm`

from tqdm import tqdm
```

```
import torch


# Separamos los datos en entrenamiento y test, usando los primeros 60000
    para entrenar y el resto para test

X_train, X_test = torch.from_numpy(X[:60000] / 255.), torch.from_numpy(X
    [60000:] / 255.)
y_train, y_test = torch.from_numpy(y[:60000]), torch.from_numpy(y
    [60000:])


# definimos el tamaño del batch y el número de batches

bs = 32
num_batches = len(X_train) // bs


# instanciamos objetos de las clases `CrossEntropyLoss` y `Adam`

loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)


# doble bucle para entrenar el modelo

for epoch in range(10):
    for b in tqdm(range(num_batches)):
        x = X_train[b*bs:(b+1)*bs] # usamos los datos en lotes (batches)
        y = y_train[b*bs:(b+1)*bs]
        y_hat = model(x) # obtenemos las predicciones del modelo, pasando
             los datos como parámetro
        loss = loss_fn(y_hat, y) # calculamos la función de pérdida,
            pasando las predicciones y las etiquetas como parámetros
        # ejecutamos diferentes métodos de los objetos `loss` y `
            optimizer`
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} loss: {loss.item():.3f}") # imprimimos
        resultados


# definimos una función para evaluar el modelo

def evaluate(model, X_test, y_test):
    acc = 0
    with torch.no_grad():
        for b in range(num_batches):
            x = X_test[b*bs:(b+1)*bs]
            y = y_test[b*bs:(b+1)*bs]
            y_hat = model(x)
            acc += torch.sum(torch.argmax(y_hat, dim=1) == y).item()
    return acc
```

```
# ejecutamos la función `evaluate` para obtener la precisión del modelo y
     mostramos resultados

acc = evaluate(model, X_test, y_test)
print(f"Accuracy: {acc} / {len(X_test)}")
```

Devuelve:

```
/Users/sensio/miniconda3/lib/python3.9/site-packages/sklearn/datasets/
    _openml.py:1002: FutureWarning: The default value of `parser` will
    change from `'liac-arff'` to `'auto'` in 1.4. You can set `parser='
    auto'` to silence this warning. Therefore, an `ImportError` will be
    raised from 1.4 if the dataset is dense and pandas is not installed.
     Note that the pandas parser may return different data types. See
    the Notes Section in fetch_openml's API doc for details.
  warn(
100%|| 1875/1875 [00:01<00:00, 1458.68it/s]

Epoch 1 loss: 0.047

100%|| 1875/1875 [00:01<00:00, 1793.50it/s]

Epoch 2 loss: 0.029

100%|| 1875/1875 [00:01<00:00, 1641.34it/s]

Epoch 3 loss: 0.049

100%|| 1875/1875 [00:01<00:00, 1847.32it/s]

Epoch 4 loss: 0.051

100%|| 1875/1875 [00:00<00:00, 1937.88it/s]

Epoch 5 loss: 0.036

100%|| 1875/1875 [00:01<00:00, 1849.13it/s]

Epoch 6 loss: 0.027

100%|| 1875/1875 [00:01<00:00, 1867.21it/s]

Epoch 7 loss: 0.007

100%|| 1875/1875 [00:00<00:00, 1938.87it/s]

Epoch 8 loss: 0.003
```

```
100%|| 1875/1875 [00:00<00:00, 1956.67it/s]

Epoch 9 loss: 0.002

100%|| 1875/1875 [00:01<00:00, 1874.38it/s]

Epoch 10 loss: 0.001
Accuracy: 9754 / 10000
```

Más allá de las peculiaridades del uso de librerías como `Pytorch` o `Numpy` deberías ser capaz de entender a alto nivel qué es lo que está haciendo el código antes de seguir adelante. Si no es así, te recomiendo que vuelvas a leer este capítulo y hagas los ejercicios propuestos.

Proyecto

Antes de cerrar el capítulo con las últimas palabras, me gustaría proponerte un proyecto con el que poner en práctica los conceptos vistos y así afianzarlos antes de seguir. Se trata de implementar el conocido juego de la vida, creado por el matemático John Horton Conway en 1970. El juego de la vida es un autómata celular, es decir, un modelo matemático que se ejecuta en una cuadrícula de celdas, en la que cada celda puede estar en un estado u otro. En el caso del juego de la vida, cada celda puede estar viva o muerta. El estado de cada celda depende del estado de sus vecinas, y se actualiza en cada iteración. Las reglas son las siguientes:

- Si una célula está viva y tiene 2 o 3 vecinas vivas, sobrevive.
- Si una célula está muerta y tiene exactamente 3 vecinas vivas, nace.
- Si una célula está viva y tiene menos de 2 o más de 3 vecinas vivas, muere.

Para llevar a cabo este proyecto deberás usar variables, bucles, condicionales, listas y funciones (y, opcionalmente, clases). Puedes usar la librería estándar para aprovechar alguna de la funcionalidad que ya trae implementada, como la generación de números aleatorios, así como Matplotlib para la generación de visualizaciones. Te animo a que intentes resolver el problema por tu cuenta antes de mirar soluciones en internet. Aunque, llegado el caso, puedes ver un ejemplo de solución aquí. Este fue el primer programa que hice cuando estaba aprendiendo a programar, en mi caso con el lenguaje `C++`, por lo que le tengo un especial cariño. Además, es una introducción perfecta al mundo de la inteligencia artificial, ya que los patrones generados con este (aparentemente) simple algoritmo con reglas simples pueden llegar a resultar hipnóticos, incluso hay quien diría que inteligentes.

¿Estás listo para lo que se viene?

Todo lo que haremos a partir de ahora asume que tienes un buen manejo de los conceptos presentados en este capítulo. En caso de que no los domines, te recomiendo que te tomes un tiempo para practicarlos antes de seguir. Tal y como te he sugerido al principio de este capítulo, puedes hacer los dos primeros módulos. Son gratuitos y podrás practicar todo lo que has aprendido hasta ahora con diferentes ejercicios y proyectos, y así afrontar el resto del libro con mayor seguridad.

3. Introducción al Aprendizaje Profundo con Pytorch

El capítulo anterior nos ha servido para aprender los conceptos esenciales de la programación con Python, sin embargo, a la hora de entrenar redes neuronales nos apoyaremos en librerías que nos faciliten el trabajo. De entre las alternativas, aquí aprenderemos a usar `Pytorch`. Además, por el camino veremos los aspectos fundamentales necesarios para aprender cómo y por qué funciona el Aprendizaje Profundo, básicamente estos son el álgebra lineal, el cálculo diferencial y la probabilidad. Si, como yo, no eres un *crack* de las matemáticas, no te preocupes, ya que iremos aprendiendo estos conceptos a partir de principios básicos, usando Python para visualizarlos y ver cómo se aplican a la hora de diseñar y entrenar redes neuronales. A grandes rasgos:

- El `Álgebra Lineal` es el campo de las matemáticas que se centra en el estudio de vectores, matrices y las operaciones que se pueden realizar con ellos. Como veremos más adelante, tanto los parámetros de las redes neuronales como los datos usados para entrenarlas se representan como matrices y vectores, por lo que ya puedes intuir su importancia.
- La `Probabilidad` es la rama de las matemáticas que estudia los fenómenos aleatorios. En el contexto del Aprendizaje Profundo, la probabilidad nos permite modelar la incertidumbre que existe en los datos de entrenamiento y en los parámetros de la red neuronal.
- El `Cálculo Numérico` nos permite diseñar los algoritmos que usamos para entrenar las redes neuronales, permitiéndonos actualizar de manera iterativa los parámetros de un modelo durante el entrenamiento para ir reduciendo progresivamente su error.

¿Preparado? ¡Vamos a ello!

Frameworks de Aprendizaje Profundo

Como ya te comenté en el capítulo anterior, el ecosistema de `Python` es probablemente su mejor rasgo a día de hoy. Esto incluye, como no, librerías, o *frameworks*, para el diseño, entrenamiento y puesta en producción de modelos de Inteligencia Artificial. De entre ellas, las que más nos interesan en este punto son los *frameworks* de Aprendizaje Profundo. Los más populares son:

- Tensorflow: Este *framework* fue desarrollado por Google y gozó de gran popularidad durante muchos años. Podríamos decir que fue gracias a Tensorflow que el aprendizaje profundo se popularizó, permitiendo a investigadores y desarrolladores de *software* crear y entrenar modelos de manera relativamente sencilla. Sin embargo, en su primera versión, Tensorflow era un *framework* bastante complejo de utilizar, por lo que su popularidad disminuyó a medida que otros *frameworks* más sencillos fueron apareciendo. Aun así, su uso está muy extendido, especialmente en aplicaciones en producción. En su versión más actual, *Tensorflow* ha simplificado y mejorado su facilidad de uso, convergiendo en su diseño con el de *Pytorch* y otros *frameworks* disponibles en la actualidad.
- Pytorch: Probablemente el *framework* más popular en el momento de escribir estas líneas. Desarrollado por Facebook (ahora Meta) simplificó en gran medida el diseño y entrenamiento de redes neuronales gracias a su interfaz *Pythónica*. Este es el *framework* que aprenderás a usar en este libro.
- Jax: Gracias al aprendizaje resultante del desarrollo y uso de *Tensorflow* y el resto de *frameworks*, Google desarrolló `JAX` con el objetivo de acelerar el entrenamiento de modelos de aprendizaje profundo. Para ello, `JAX` utiliza `XLA`, un compilador de alto rendimiento que permite ejecutar código Python en GPUs y TPUs. Aunque `JAX` es un *framework* relativamente nuevo, su uso está creciendo rápidamente gracias a su rendimiento, aunque es mucho más complejo que el de `Pytorch` y `Tensorflow`, y actualmente está reservado para centros de investigación.
- Otros *frameworks* que merecen ser mencionados son: Tinygrad, desarrollado por la empresa TinyCorp, que pretende crear el *framework* más pequeño en términos de líneas de código a la vez que sencillo de soportar en cualquier *hardware*; MLX, el *framework* de Apple optimizado para sus CPUs y GPUs; y muchos otros.

Por otro lado, existen los conocidos como *metaframeworks*, o librerías que se construyen por encima de alguno o varios de los *frameworks* anteriores con el objetivo de facilitar su

uso y extender su rango de aplicación. Los más populares son:

- Keras, que en su versión 3 es capaz de usar tanto `Tensorflow` como `Pytorch` o `JAX` en el *backend*. Esta es una librería muy popular, debido a que permite, con muy pocas líneas de código, diseñar y entrenar modelos de aprendizaje profundo. Sin embargo, esto lo consigue gracias a una sintaxis propia, por lo que no es muy amigable a la hora de resolver problemas o crear modelos más complejos.
- Pytorch Lightning, una librería implementada por encima de `Pytorch` que nos facilita la vida a la hora de entrenar modelos de aprendizaje profundo de manera más avanzada (entrenamiento distribuido en varias GPUs, *trackeado* de experimentos, etc.). Esta es la librería que yo uso en mi día a día, por lo que te recomiendo que le eches un vistazo aunque no la usaremos en este libro.

Finalmente, existen otras librerías en el ecosistema que nos permiten ejecutar modelos de manera eficiente en diferentes dispositivos una vez han sido entrenados. De entre ellas, destacar ONNX, un formato interoperable que te permitirá ejecutar tus modelos entrenados en cualquier *framework* en diferentes dispositivos (desde un navegador web hasta un dispositivo móvil) y TensorRT, una librería para optimizar modelos en GPUs de NVIDIA especialmente interesante para dispositivos *edge* como cámaras, sensores, robots, vehículos autónomos, etc.

Como puedes ver, estamos en un campo muy activo en el que existen muchas alternativas (y cada día aparecen más). En este libro nos centraremos en Pytorch, que es, en mi opinión, el más sencillo de utilizar y útil a la hora de aprender. Sin embargo, una vez hayas aprendido los conceptos básicos, te recomiendo que inviertas tiempo en aprender Keras o Pytorch Lightning, ya que te permitirán crear modelos de manera más sencilla y rápida.

Introducción a Pytorch

¿Qué es Pytorch?

`Pytorch` es un *framework* de `redes neuronales`, un conjunto de librerías y herramientas que nos hacen la vida más fácil a la hora de diseñar, entrenar y poner en producción nuestros modelos de Aprendizaje Profundo. Una forma sencilla de entender qué es `Pytorch` es la siguiente:

$$Pytorch = Numpy + Autograd + GPU$$

Una de las características más importantes de Pytorch es su facilidad de uso. Esto es debido a que su interfaz es muy similar a la de `NumPy`, por lo que si estás familiarizado con esta librería no debería costarte mucho usar `Pytorch`. `NumPy` es una librería de Python que nos permite trabajar con vectores y matrices de manera eficiente, pero para entrenar redes neuronales necesitamos *vitaminar* `NumPy` con las dos otras características que ofrece `Pytorch`: `Autograd` y ejecución en `GPU`. `Autograd` es un mecanismo que nos permite calcular de manera automática las derivadas de las funciones que usamos para entrenar las redes neuronales con respecto a sus parámetros, mientras que la ejecución en `GPU` nos permite acelerar el entrenamiento de los modelos. En las siguientes secciones veremos todo esto en detalle.

Instalación

Puedes instalar `Pytorch` de diferentes maneras, te recomiendo que eches un vistazo a su web para obtener instrucciones concretas y actualizadas. En el momento de escribir estas líneas, el siguiente comando es el recomendado para instalar Pytorch en Linux con soporte CUDA.

```
pip3 install torch
```

Si quieres instalarlo en otro sistema operativo, como Mac o Windows, simplemente usa el comando indicado en la web. Si estás trabajando en entornos *cloud*, como Google Colab, es muy probable que ya tengas `Pytorch` instalado. Puedes verificarlo con el siguiente bloque de código:

Python

```
import torch

torch.__version__
```

Devuelve: *2.0.1*

Numpy

Pues empezamos por el primer elemento que le da forma a Pytoch: Numpy. Si bien Pytorch no usa Numpy, su API es muy similar (esto hace referencia al nombre de las funciones y clases así como sus entradas y salidas), por lo que aprender Numpy no solo te servirá para entender Pytorch, sino que también te será útil para otras tareas de programación científica, como el análisis de datos, procesamiento de imágenes o cálculo numérico.

Numpy es muy usado para todo lo que tiene que ver con el `álgebra lineal`, una rama de las matemáticas utilizada en muchos campos de la ciencia y la ingeniería, incluyendo la Inteligencia Artificial. Un buen entendimiento de álgebra lineal es esencial para trabajar con la mayoría de algoritmos de Machine Learning, y sobretodo algoritmos de Aprendizaje Profundo. En esta sección vamos a ver los conceptos de álgebra lineal más importantes para el desarrollo de algoritmos de IA, omitiendo otros conceptos importantes pero no esenciales para nuestros objetivos, y los veremos a través de ejemplos de código en Python usando Numpy (y que podrás aplicar a Pytorch).

Escalares

Un `escalar` es simplemente un número único, a diferencia de la mayoría del resto de objetos matemáticos estudiados en álgebra lineal que normalmente son colecciones de múltiples números. Suelen denotarse mediante letras en minúscula, por ejemplo, cuando decimos que una red neuronal tiene un número n de neuronas estamos usando un valor escalar.

Python

```
# un valor escalar
x = 1
x
```

Devuelve: *1*

Vectores

Un `vector` es una colección de números dispuestos en una secuencia. Podemos identificar cada elemento individual mediante su posición en la secuencia. Suelen denotarse mediante letras en minúscula y en negrita, por ejemplo $\mathbf{x}$, mientras que cada elemento se denota

con la misma letra que el vector, pero sin negrita y con un subíndice indicando su posición en el vector, por ejemplo x_1 es el primer elemento en el vector $\mathbf{x}$.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

En Python, podemos representar un vector como una lista de números:

Python

```
v = [1, 2, 3, 4]
v
```

Devuelve: *[1, 2, 3, 4]*

Python

```
# primer valor del vector

v[0]
```

Devuelve: *1*

> Recuerda que en Python los índices de las diferentes estructuras de datos empiezan por el valor 0.

Utilizaremos vectores para representar un conjunto de puntos en el espacio, secuencias de valores en series temporales (como, por ejemplo, la evolución de la temperatura en un lugar determinado), texto en las que cada elemento corresponde a una palabra o carácter (ya veremos más adelante cómo transformar texto en números), etc.

Matrices

Una `matriz` es un vector bidimensional, en el que cada elemento se identifica mediante dos índices en vez de uno. Suelen denotarse mediante letras en mayúscula y en negrita,

por ejemplo $\mathbf{A}$. Como en el caso de los vectores, cada elemento se identifica con la misma letra que la matriz, pero sin negrita y con dos subíndices indicando su posición en la matriz. Por ejemplo, el valor $A_{1,1}$ es el elemento en la primera fila y primera columna.

$$\mathbf{A} = \begin{bmatrix} A_{1,1} A_{1,2} \\ A_{2,1} A_{2,2} \end{bmatrix}$$

En Python, podemos representar una matriz como una lista de listas.

Python

```
# una matriz

A = [[1, 2], [3, 4]]
A
```

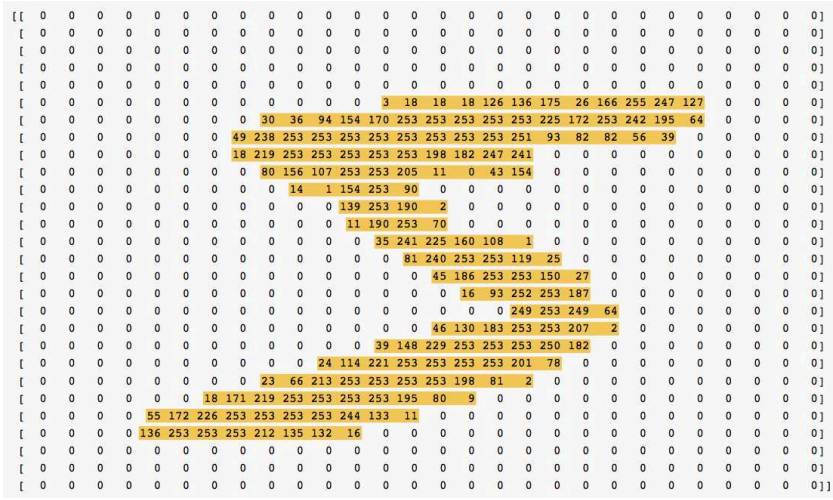
Devuelve: *[[1, 2], [3, 4]]*

Python

```
# primer valor

A[0][0]
```

Devuelve: *1*

Utilizaremos matrices para representar imágenes en blanco y negro, los parámetros de una capa en una red neuronal, etc. En la siguiente figura puedes ver un ejemplo de una imagen con un número 5. Como puedes ver se trata de una matriz en la que cada elemento indica la intensidad de luz para cada píxel.



Tensores

Un `tensor` es una secuencia de valores dispuesta en una malla regular con un número arbitrario de dimensiones. Podemos ver un `escalar` como un tensor con 0 dimensiones, un `vector` como un tensor de 1 dimensión y una `matriz` como un tensor de 2 dimensiones. Así pues, hablamos de tensores en general como la estructura de datos que engloba las estructuras vistas anteriormente y cualquier otra con mayor número de dimensiones.

Python

```
# un tensor de 3 dimensiones

# puedes interpretarlo como dos matrices

A = [[[1, 2], [3, 4]],[[1, 2], [3, 4]]]
A
```

Devuelve: *[[[1, 2], [3, 4]], [[1, 2], [3, 4]]]*

Usaremos tensores de tres dimensiones para representar imágenes en color (canales RGB), tensores de cuatro dimensiones para representar vídeos (secuencia de imágenes en color), etc.

NumPy

Hemos visto que podemos usar listas de Python para representar tensores. No obstante, a la hora de hacer operaciones con ellos, el lenguaje no proporciona un buen soporte. Es en este punto en el que entra en juego `NumPy` (*Numerical Python*). Sus principales características son:

- Ofrece el objeto *ndarray*, similar a una lista de Python pero optimizada para el cálculo numérico. Nos referiremos a este objeto como *array* de NumPy, o simplemente *array*.
- Implementa funciones matemáticas que pueden trabajar directamente sobre *arrays* sin tener que implementar bucles.
- Proporciona funciones para leer/escribir datos a archivos de manera optimizada.
- Permite aplicaciones de álgebra lineal, generación de números aleatorios y transformadas de Fourier.

El núcleo de Numpy está implementado en C, ofreciendo *bindings* en Python para interactuar con él. Esto se traduce en que NumPy es más rápido que el equivalente en puro Python. Muchas otras librerías para el análisis de datos están construidas sobre NumPy, utilizando los *ndarrays* como la estructura de datos básica debido a su eficiencia. Para ilustrar esta propiedad vamos a calcular el tiempo necesario para llevar a cabo la misma operación en puro Python en comparación con *arrays* de NumPy.

Python

```
l = [i for i in range(10000000)]

%time l2 = [2*i for i in l]
```

Devuelve:
CPU times: user 254 ms, sys: 206 ms, total: 460 ms
Wall time: 540 ms

Python

```
import numpy as np

a = np.array(l)

%time a2 = 2*a
```

Devuelve:
CPU times: user 6.87 ms, sys: 18.5 ms, total: 25.4 ms
Wall time: 28.8 ms

Podemos ver que Numpy es mucho más rápido que Python llevando a cabo la misma operación. No te preocupes por los detalles, en las siguientes secciones aprenderás lo necesario para crear *arrays*, hacer operaciones, etc. Lo más importante es destacar que en Python necesitamos iterar por cada valor en la lista aplicando la operación en concreto (lo cual es lento) mientras que en NumPy podemos simplemente aplicar la operación al *array* entero confiando en la implementación subyacente para llevar a cabo la operación de la manera más eficiente posible.

> Numpy es un módulo externo de Python, por lo que tendrás que instalarlo con el comando `pip install numpy`.

El objeto `ndarray`

Como hemos comentado en la sección anterior, NumPy está basado en el objeto *ndarray*. Puedes ver este objeto como una lista de Python con súperpoderes. El objeto *ndarray* es multidimensional, lo que implica que nos permite representar tanto valores escalares como vectores, matrices y tensores (o matrices multidimensionales).

Para poder trabajar con NumPy, primero tenemos que importarlo. Es común importarlo con el nombre `np`.

Python

```python
import numpy as np
```

Tenemos varias maneras de crear un *array*. Una de ellas es utilizar funciones implementadas en NumPy para la creación de *arrays*, indicando el número de elementos en cada dimensión.

Python

```
# crear un vector de ceros

np.zeros(5)
```

Devuelve: *array([0., 0., 0., 0., 0.])*

Python

```
# crear una matriz de ceros

np.zeros((3, 4))
```

Devuelve:
array([
[0., 0., 0., 0.],
[0., 0., 0., 0.],
[0., 0., 0., 0.]])

De la misma manera podemos crear `arrays` de 1s, con un valor determinado o sin inicializar con las funciones `np.one()`, `np.full()` o `np.empty()` respectivamente. Estas son algunas de las propiedades de un `array`.

Python

```
# tensor de unos

a = np.ones((3, 4, 2))

a
```

Devuelve:
array([[[1., 1.],
[1., 1.],
[1., 1.],
[1., 1.]],

[[1., 1.],
[1., 1.],
[1., 1.],

[1., 1.]],

[[1., 1.],
[1., 1.],
[1., 1.],
[1., 1.]]])

Python

```
# número elementos en cada dimensión

a.shape
```

Devuelve: *(3, 4, 2)*

Python

```
# longitud del array

a.ndim
```

Devuelve: *3*

Python

```
# elementos totales en el array

a.size
```

Devuelve: *24*

Otra manera muy común de inicializar `arrays` de `Numpy` es mediante listas de `Python`. Para ello usamos la función `np.array()`.

Python

```
np.array([[1, 2, 3],[4, 5, 6]])
```

Devuelve:
array(
[[1, 2, 3],
[4, 5, 6]])

Por último, también podemos crear *arrays* mediante funciones secuenciales o con valores aleatorios de la siguiente manera.

Python

```
# vector de `int` en rango

np.arange(1, 5)
```

Devuelve: *array([1, 2, 3, 4])*

Python

```
# vector de `float` en rango

np.linspace(0, 1, 10)
```

Devuelve:
array([0. , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.])

Python

```
# matriz de números aleatorios

np.random.rand(3,4)
```

Devuelve:
array([[0.54136795, 0.07058591, 0.68007413, 0.50998075],
[0.18159313, 0.09477728, 0.70835038, 0.97899667],
[0.34458957, 0.00462217, 0.3875978 , 0.01316419]])

Tipos de datos

A diferencia de las estructuras de datos de Python, en las que podemos mezclar cualquier tipo de datos, los *arrays* de NumPy tienen que tener siempre datos del mismo tipo (este es uno de los motivos que nos permiten optimizar sus operaciones).

Python

```
a = np.arange(1, 5)
a
```

Devuelve: *array([1, 2, 3, 4])*

Python

```
# acceder al tipo de datos

a.dtype
```

Devuelve: *dtype('int64')*

Podemos indicarle a NumPy el tipo de dato con el que queremos trabajar al crear nuestro *array*.

Python

```
a = np.arange(1, 5, dtype=np.uint8)
a
```

Devuelve: *array([1, 2, 3, 4], dtype=uint8)*

Los tipos disponibles son `int8|16|32|64`, `uint8|16|32|64`, `float16|32|64` y `complex64|128`. Puedes encontrar una lista completa en la documentación.

Indexado y troceado

NumPy adopta la misma lógica de indexado y troceado que Python, algo que ya conocemos y que puedes refrescar en el capítulo anterior.

Python

```
a = np.array([1, 5, 3, 19, 13, 7])
a
```

Devuelve: *array([1, 5, 3, 19, 13, 7])*

Python

```
# acceder a un valor por su índice

a[3]
```

Devuelve: *19*

> Al igual que las listas de Python, el primer elemento de un *array* tiene índice 0.

Python

```
# troceado

a[2:5]
```

Devuelve: *array([3, 19, 13])*

Python

```
# usamos índices negativos para indexar desde el final

a[2:-1]
```

Devuelve: *array([3, 19, 13])*

Podemos indexar *arrays* multidimensionales con diferentes índices para cada dimensión, separados por comas.

Python

```
a = a.reshape(2, 3)
a
```

Devuelve:
array([[1, 5, 3],
[19, 13, 7]])

Python

```
# valor en segunda fila, tercera columna
a[1, 2]
```

Devuelve: *7*

Python

```
# segunda fila
a[1, :]
```

Devuelve: *array([19, 13, 7])*

Python

```
# última columna
a[:, -1]
```

Devuelve: *array([3, 7])*

El indexado *fancy* nos permite indexar un *array* mediante una lista con los índices de interés.

Python

```
# primera y segunda fila, desde la segunda columna a la tercera
a[(0,1),1:4]
```

Devuelve:
array([[5, 3],
[13, 7]])

El indexado *booleano* es muy útil para trabajar con máscaras.

Python

```
a = np.array([1, 2, 3, 4])
mask = np.array([True, False, True, False])

a[mask]
```

Devuelve: *array([1, 3])*

Iteración

Podemos iterar sobre un *array* de NumPy de la misma manera que iteramos cualquier otra estructura de datos en Python.

Python

```
a = np.arange(5)
a
```

Devuelve: *array([0, 1, 2, 3, 4])*

Python

```
for i in a:
    print(i)
```

Devuelve:
0
1
2
3
4

Al trabajar con *arrays* multidimensionales, necesitaremos un bucle para cada dimensión.

Python

```
a = np.arange(9).reshape((3,3))
a
```

Devuelve:

array([[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])

Python

```
for fila in a:
    for i in fila:
        print(i)
```

Devuelve:

0
1
2
3
4
5
6
7
8

Guardado y cargado

Podemos guardar nuestros *arrays* en archivos que más tarde podemos cargar de nuevo, muy útil para guardar un modelo entrenado que luego cargamos para hacer predicciones.

Python

```
a = np.random.rand(2,3)
a
```

Devuelve:

array([[0.03451408, 0.06039806, 0.16211501],
[0.50103188, 0.3333689 , 0.21666378]])

Python

```
# guardar array en archivo

np.save("mi_array", a)
```

Por defecto un *array* se guarda con la extensión `.npy`. Para cargar de nuevo el *array*:

Python

```
b = np.load("mi_array.npy")
b
```

Devuelve:

array([[0.03451408, 0.06039806, 0.16211501],
[0.50103188, 0.3333689 , 0.21666378]])

Las funciones que hemos visto guardan los *arrays* en formato binario para maximizar la velocidad de lectura. Sin embargo, podemos guardar nuestros *arrays* en formato texto para utilizarlos en otras aplicaciones.

Python

```
# guardar array en formato csv

np.savetxt("mi_array.csv", a, delimiter=",")
```

También podemos guardar varios *arrays* en un solo archivo comprimido en formato `.npz`.

Python

```
b = np.arange(24, dtype=np.uint8).reshape(2, 3, 4)
b
```

Devuelve:

array([[[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]],

[[12, 13, 14, 15],
[16, 17, 18, 19],

[20, 21, 22, 23]]], dtype=uint8)

Python

```
# guardar arrays

np.savez("mis_arrays", a=a, b=b)


# cargar arrays

mis_arrays = np.load("mis_arrays.npz")
mis_arrays
```

Devuelve: *<numpy.lib.npyio.NpzFile at 0x135c82cd0>*

Python

```
mis_arrays["a"]
```

Devuelve:
array([[0.03451408, 0.06039806, 0.16211501],
[0.50103188, 0.3333689 , 0.21666378]])

Python

```
mis_arrays["b"]
```

Devuelve:
array([[[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]],

[[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23]]], dtype=uint8)

Programación orientada a *arrays*: Vectorización y *Broadcasting*

Como hemos visto, la principal ventaja que ofrece NumPy es la velocidad superior respecto a Python en aplicaciones que requieren estructuras de datos numéricos que podemos

expresar en forma de vectores, matrices o tensores. Parte de esta mejora viene del hecho de que usando NumPy podemos llevar a cabo muchos tipos de procesado de datos mediante expresiones concisas aplicadas sobre *arrays* que de otra forma requerirían bucles. La práctica de sustituir bucles por expresiones basadas en *arrays* se conoce como **vectorización**, que junto al *broadcasting* forman la base de la programación orientada a *arrays* (Array-Oriented Programming).

Vectorización Queremos evaluar la función $z = \sqrt{x^2 + y^2}$ sobre una malla regular de valores. Una manera simple de llevar a cabo esta operación es la siguiente:

Python

```
import numpy as np
import math

def slow(n=1000):
    x = np.linspace(-1,1,n)
    y = np.linspace(-1,1,n)
    z = np.empty((n,n))
    for i, _y in enumerate(y):
        for j, _x in enumerate(x):
            z[i, j] = math.sqrt(_x**2 + _y**2)
    return x, y, z

%time x, y, z = slow()
```

Devuelve:

CPU times: user 268 ms, sys: 2.94 ms, total: 271 ms
Wall time: 275 ms

Como puedes ver en la implementación de la función la evaluación de la expresión requiere de dos bucles `for` para iterar sobre todos los valores de `x` e `y` calculando el resultado. Esta es una implementación ineficiente que puede acelerarse si aplicamos la vectorización, sustituir bucles por expresiones basadas en *arrays*.

Python

```
def fast(n=1000):
    p = np.linspace(-1,1,n)
    x, y = np.meshgrid(p, p)
    z = np.sqrt(x**2 + y**2)
    return x, y, z

%time x, y, z = fast()
```

Devuelve:
CPU times: user 4.2 ms, sys: 10.5 ms, total: 14.7 ms
Wall time: 16.9 ms

El resultado que obtenemos es el mismo, pero mucho más rápido. Esto es gracias al uso de la vectorización. NumPy pone a nuestra disposición multitud de funciones que podemos aprovechar para acelerar nuestros cálculos.

Python

```
a = np.array([[2.5, 3.1, 7], [10, 11, 12]])
a
```

Devuelve:
array([[2.5, 3.1, 7.],
[10. , 11. , 12.]])

Python

```
for func in (np.abs, np.sqrt, np.exp, np.log, np.sign, np.ceil, np.modf,
    np.isnan, np.cos):
    print("\n", func.__name__)
    print(func(a))
```

Devuelve:
absolute
[[2.5, 3.1, 7.],
[10. , 11. , 12.]]

sqrt
[[1.58113883, 1.76068169, 2.64575131],
[3.16227766, 3.31662479, 3.46410162]]

exp
[[1.21824940e+01, 2.21979513e+01, 1.09663316e+03],
[2.20264658e+04, 5.98741417e+04, 1.62754791e+05]]

log
[[0.91629073, 1.13140211, 1.94591015],
[2.30258509, 2.39789527, 2.48490665]]

sign
[[1., 1., 1.],
[1., 1., 1.]]

ceil
[[3., 4., 7.],
[10., 11., 12.]]

modf
(array([[0.5, 0.1, 0.],
[0. , 0. , 0.]]),
array([[2., 3., 7.],
[10., 11., 12.]]))

isnan
[[False, False, False],
[False, False, False]]

cos
[[-0.80114362, -0.99913515, 0.75390225],
[-0.83907153, 0.0044257 , 0.84385396]]

También tenemos funciones para calcular estadísticos de un *array*, como la media, la desviación estándar, la mediana, etc.

Python

```
for func in (a.min, a.max, a.sum, a.prod, a.std, a.var):
    print(func.__name__, "=", func())
```

Devuelve: *min = 2.5*
max = 12.0
sum = 45.6
prod = 71610.0
std = 3.7260345319566395
var = 13.883333333333333

Es posible aplicar estas operaciones en dimensiones concretas.

Python

```
# calcular valor medio de cada columna
a.mean(axis=0)
```

Devuelve: *array([6.25, 7.05, 9.5])*

Python

```
# suma todos los valores de cada fila
a.sum(axis=1)
```

Devuelve: *array([12.6, 33.])*

Broadcasting El `broadcasting` es la segunda propiedad (después de la `vectorización`) que le proporciona a `NumPy` su versatilidad y potencia a la hora de llevar a cabo operaciones con `arrays`. Esta nos permitirá llevar a cabo operaciones con *arrays* que, por sus dimensiones, no se deberían poder llevar a cabo estrictamente hablando desde un punto de vista matemático (por ejemplo, sumar una matriz y un vector, o un vector y un escalar). Como norma general, cuando `Numpy` espera `arrays` de la misma forma, pero encuentra que esto no se cumple, aplica las reglas del `broadcasting`. Dominar estas reglas nos permitirá implementar operaciones mucho más concisas y rápidas. Vamos a verlas en detalle.

La primera regla nos dice que si dos *arrays* no tienen el mismo rango, entonces se añadirá una dimensión de `1` al principio del *array* con menor rango hasta que estos coincidan.

> El rango de un *array* es lo mismo que su número de dimensiones.

Python

```
a = np.arange(5).reshape(1, 1, 5)
a
```

Devuelve: *array([[[0, 1, 2, 3, 4]]])*

Python

```
a.shape
```

Devuelve: *(1, 1, 5)*

Vamos a intentar sumar este tensor al siguiente vector:

Python

```
b = np.arange(5)
b
```

Devuelve: *array([0, 1, 2, 3, 4])*

Python

```
b.shape
```

Devuelve: *(5,)*

Python

```
a + b
```

Devuelve: *array([[[0, 2, 4, 6, 8]]])*

Como puedes ver, NumPy no se ha quejado ni nos ha dado ningún error pese a que formalmente esta operación no es válida. Lo que ha ocurrido es que la primera regla del *broadcasting* ha sido aplicada, añadiendo dimensiones extra a nuestro *array* b para convertirlo en un *array* con dimensiones (1, 1, 5) y así poder llevar a cabo la operación. Esta es la misma regla que nos permite sumar un valor escalar a un vector o una matriz.

Python

```
1 + a
```

Devuelve: *array([[[1, 2, 3, 4, 5]]])*

La segunda regla nos dice que si un *array* tiene un `1` en alguna dimensión en particular actuará como si tuviese la longitud del *array* con mayor longitud en aquella dimensión.

Esto implica que el valor en esta dimensión se repetirá hasta coincidir con la longitud. Por ejemplo:

Python

```
a = np.arange(6).reshape(2, 3)
a
```

Devuelve:
array([[0, 1, 2],
[3, 4, 5]])

Python

```
a.shape
```

Devuelve: *(2, 3)*

Python

```
b = np.array([[100], [200]])
b
```

Devuelve:
array([[100],
[200]])

Python

```
b.shape
```

Devuelve: *(2, 1)*

Si intentamos sumar estas dos matrices, NumPy repetirá los valores de `b` hasta crear tres columnas iguales y así poder llevar a cabo la operación.

Python

```
a + b
```

Devuelve:
array([[100, 101, 102],
[203, 204, 205]])

Las reglas del `broadcasting` pueden combinarse.

Python

```
c = np.array([100, 200, 300])
c.shape
```

Devuelve: *(3,)*

Python

```
a + c
```

Devuelve:
array([[100, 201, 302],
[103, 204, 305]])

NumPy ha añadido una dimensión extra al principio de `c` para obtener una forma de `(1, 3)` y luego ha repetido todos los valores en una nueva fila para conseguir una forma de `(2, 3)` y así poder llevar a cabo la suma.

La tercera regla nos dice que después de aplicar las reglas 1 y 2, las dimensiones de los *arrays* deben coincidir. Si no es así, la operación no se puede llevar a cabo.

Python

```
a + [1, 2]
```

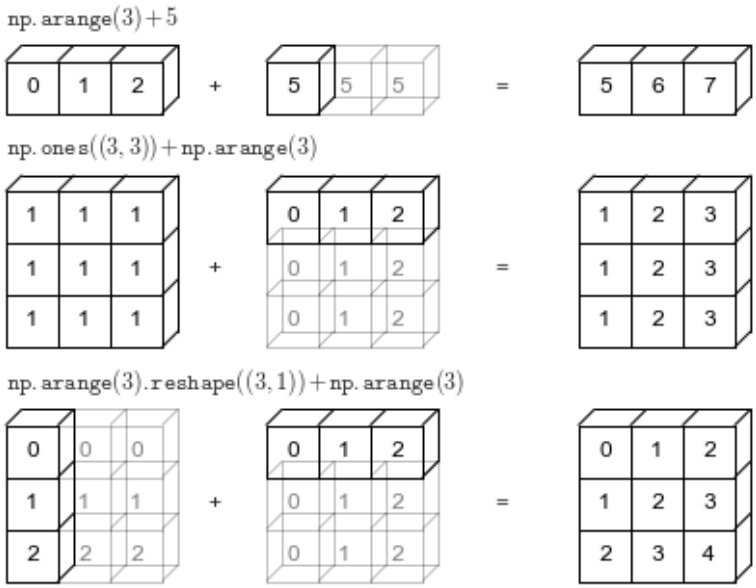
Devuelve un error:

ValueError Traceback (most recent call last)

/Users/sensio/Desktop/libro/04_fundamentos.ipynb Cell 127 in 1 —-> 1 a + [1, 2]

ValueError: operands could not be broadcast together with shapes (2,3) (2,)

Aquí tienes un ejemplo visual de estas reglas aplicadas sobre varios *arrays*.



A la hora de diseñar y entrenar redes neuronales haremos un uso extensivo de estas reglas para poder llevar a cabo muchas de las operaciones envueltas en este proceso.

Operaciones

Una vez introducido el objeto matemático fundamental del álgebra lineal, el tensor, y su equivalente en código Python, el `ndarray`, vamos a ver qué operaciones podemos llevar a cabo con ellos.

Trasposición

Una operación importante cuando trabajamos con matrices es la `traspuesta`. Consiste en intercambiar las filas por las columnas, y suele denotarse con el superíndice $(\cdot)^T$, por ejemplo $\mathbf{A}^T$ es la matriz traspuesta de $\mathbf{A}$.

Python

```
A = np.arange(10).reshape(2,5)
A
```

Devuelve:
array([[0, 1, 2, 3, 4],
[5, 6, 7, 8, 9]])

Python

```
A.T
```

Devuelve:
array([[0, 5],
[1, 6],
[2, 7],
[3, 8],
[4, 9]])

Un vector es una matriz de una sola columna, por lo que su transpuesta es simplemente una matriz con una fila con los mismos valores. En cuanto a un valor escalar, él mismo es su traspuesta. Como veremos más adelante, algunas operaciones con tensores requieren que estos tengan unas dimensiones concretas, por lo que a veces necesitaremos trasponer tensores para poder llevar a cabo dichas operaciones.

Suma y resta

Podemos sumar y restar matrices entre ellas siempre que tengan la misma forma, añadiendo cada elemento de manera independiente.

Python

```
A = np.random.randn(3,3)
B = np.random.randn(3,3)

A, B
```

Devuelve:
(array([[0.90803849, -0.9685911 , -0.28821889],
[-0.21972879, 0.51057993, -0.17264292],
[-0.77155575, -0.75616334, -1.77097818]]),

array([[1.60500904, -1.39148164, -0.05244577],

[-1.84133471, -1.00416671, 0.02791049],
[-0.61384255, -0.00430097, -0.71224058]]))

Python

```
A + B
```

Devuelve:
array([[2.51304753, -2.36007274, -0.34066466],
[-2.06106351, -0.49358677, -0.14473243],
[-1.3853983 , -0.76046431, -2.48321876]])

Aunque tal y como hemos visto en la sección anterior podemos saltarnos esta restricción al trabajar con NumPy gracias al `broadcasting`.

Multiplicación de matrices

Una de las operaciones más importantes en álgebra lineal es la multiplicación de matrices. Para poder multiplicar dos matrices, $\mathbf{C} = \mathbf{AB}$, necesitamos que $\mathbf{A}$ tenga el mismo número de columnas que filas tiene $\mathbf{B}$. El resultado, $\mathbf{C}$, será una matriz con el mismo número de filas que $\mathbf{A}$ y el mismo número de columnas que $\mathbf{B}$.

Python

```
A = np.array([[1,2,1],[0,1,0],[2,3,4]])
B = np.array([[2,5],[6,7],[1,8]])

A, B
```

Devuelve:
(array([[1, 2, 1],
[0, 1, 0],
[2, 3, 4]]),

array([[2, 5],
[6, 7],
[1, 8]]))

Python

```
C = A.dot(B)
C
```

Devuelve:
array([[15, 27],
[6, 7],
[26, 63]])

Python

```
# notación alternativa

C = A @ B
C
```

Devuelve:
array([[15, 27],
[6, 7],
[26, 63]])

Python

```
C.shape
```

Devuelve: *(3, 2)*

Esta operación es distributiva, $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC}$, y asociativa, $\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$, pero no commutativa, $\mathbf{AB} \neq \mathbf{BA}$.

Ten en cuenta que esta operación no es el resultado de multiplicar cada elemento de las matrices por separado. Este tipo de multiplicación se conoce como `Hardamard product` o `element-wise product` en inglés. En este caso ambas matrices deberán tener la misma forma.

Python

```
A = np.random.randn(2,2)
B = np.random.randn(2,2)

A, B
```

Devuelve:

(array([[0.9516984 , 0.56901991],
[1.84130937, -0.35349111]]),

array([[-1.06685798, 0.25000438],
[1.05519757, 0.78566518]]))

Python

```
# multiplicamos cada elemento de manera independiente

A*B
```

Devuelve:

array([[-1.01532703, 0.14225747],
[1.94294517, -0.27772565]])

Identidad y matriz inversa

La matriz `identidad` es una matriz con 1 en todos los valores de su diagonal y 0 en el resto de valores.

Python

```
# matriz identidad

I = np.eye(3)
I
```

Devuelve:

array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]])

Esta matriz tiene la propiedad de no alterar ningún vector por el que se multiplique, y nos permite definir la matriz `inversa` como aquella matriz que cumple la condición $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, donde $\mathbf{A}^{-1}$ es la matriz inversa de $\mathbf{A}$ y $\mathbf{I}$ es la matriz identidad. Esta matriz inversa nos permite resolver sistemas de ecuaciones lineales de la forma $\mathbf{Ax} = \mathbf{b}$, donde $\mathbf{A}$ y $\mathbf{b}$ son una matriz y un vector, respectivamente, de valores conocidos y $\mathbf{x}$ es un vector de incógnitas. La

solución a este sistema es $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. El problema con la matriz inversa es que no siempre existe y, cuando lo hace, calcularla puede requerir de un tiempo de cálculo considerable, el cual aumenta con el tamaño de $\mathbf{A}$ requiriendo de métodos alternativos de resolución en la gran mayoría de ocasiones.

Python

```
A = np.array([[1,2,3],[5,7,11],[21,29,31]])
A
```

Devuelve:

array([[1, 2, 3],
[5, 7, 11],
[21, 29, 31]])

Python

```
import numpy.linalg as linalg

linalg.inv(A)
```

Devuelve:

array([[-2.31818182, 0.56818182, 0.02272727],
[1.72727273, -0.72727273, 0.09090909],
[-0.04545455, 0.29545455, -0.06818182]])

> Puedes encontrar toda la funcionalidad que ofrece NumPy para álgebra lineal en el paquete `numpy.linalg`.

Podemos resolver el siguiente sistema de ecuaciones lineales:

$$2x + 6y = 6$$

$$5x + 3y = -9$$

De esta manera:

Python

```
A  = np.array([[2, 6], [5, 3]])
b = np.array([6, -9])

x = linalg.inv(A).dot(b)
x
```

Devuelve: *array([-3., 2.])*

Python

```
# comprobar solución

A.dot(x) == b
```

Devuelve: *array([True, True])*

Ya que hemos reescrito el problema como $\mathbf{Ax} = \mathbf{b}$, donde

$$\mathbf{A} = \begin{bmatrix} 2, 6 \\ 5, 3 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 6 \\ -9 \end{bmatrix}$$

Otros conceptos y recursos adicionales

Un tipo especial de matriz es la `diagonal`, con elementos diferentes de 0 en su diagonal y 0 en el resto de elementos (similar a la matriz identidad). Otros tipos de matrices interesantes son las matrices `simétricas`, cuya traspuesta es ella misma, $\mathbf{A} = \mathbf{A}^T$, y las matrices `ortogonales`, cuya inversa es igual a su traspuesta, $\mathbf{A}^{-1} = \mathbf{A}^T$, muy interesante ya que en este caso calcular la inversa es una operación sencilla. También existen otras operaciones interesantes como la descomposición de matrices, muy usadas en algoritmo de Machine Learning como la reducción de la dimensionalidad. En cualquier caso, si quieres aprender más sobre álgebra lineal y cómo se aplica en el campo de la IA, te recomiendo las siguientes referencias:

- *The Matrix Cookbook* (Petersen y Pedersen, 2006)
- *Deep Learning* (Goodfellow, Bengio y Courville, 2016)

- MIT, 2010 curso *online* gratis
- FastAI curso *online* gratis
- 3Blue1Brown Youtube
- Post

El objeto `Tensor` de Pytorch

¿Pero no estábamos aprendiendo Pytorch? Así es, y es que como comentaba antes, Pytorch está inspirado en la interfaz de Numpy. La principal diferencia es que en lugar de usar el `ndarray` como estructura de datos fundamental, usaremos el objeto `Tensor` de Pytorch.

Python

```
# array de numpy

x = np.array([1, 2, 3])
x
```

Devuelve: *array([1, 2, 3])*

Python

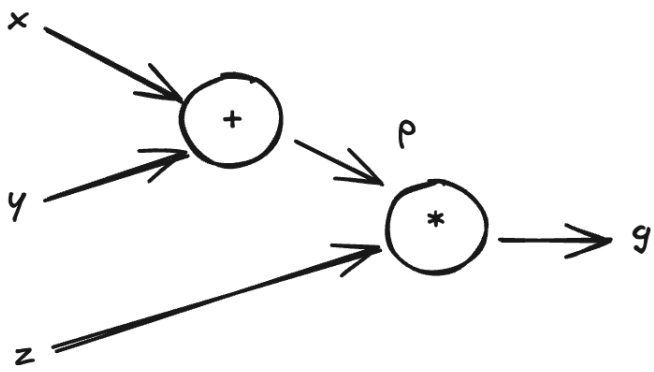
```
# tensor de pytorch

x = torch.tensor([1, 2, 3])
x
```

Devuelve: *tensor([1, 2, 3])*

No solo podremos hacer prácticamente todo lo que hemos visto anteriormente con `Numpy`, sino que además el `Tensor` de Pytorch añade *superpoderes* adicionales a nuestros tensores para facilitar el entrenamiento de redes neuronales: el cálculo de gradientes automático y la aceleración por GPU.

Autograd

Vamos a ver un ejemplo de `autograd` en acción para el cálculo de derivadas automáticas. Para ello, considera el siguiente grafo computacional sencillo:



Tenemos tres `tensores`, x, y y z, los cuales combinamos en diferentes operaciones para calcular g. ¿Cómo podemos encontrar la derivada de g con respecto a cada uno de los tensores a la entrada? Es en este punto en el que entra en juego el `cálculo numérico`, y en el que tendrás que desenpolvar tus conocimientos sobre derivadas que aprendiste en el instituto. Para el caso de z esto es sencillo:

$$\frac{dg}{dz} = p = x + y$$

En el caso de x e y es un poco más complicado, ya que tenemos que aplicar la regla de la cadena de la derivada:

$$\frac{dg}{dx} = \frac{dg}{dp}\frac{dp}{dx} = z$$
$$\frac{dg}{dy} = \frac{dg}{dp}\frac{dp}{dy} = z$$

Si bien en este ejemplo sencillo lo hemos podido calcular a mano, imagina tener que hacer esto en redes neuronales con miles de millones de parámetros… imposible. `Autograd` nos permite calcular estas derivadas de manera automática.

Python

```
x = torch.tensor(1., requires_grad=True)
y = torch.tensor(2., requires_grad=True)
p = x + y

z = torch.tensor(3., requires_grad=True)
g = p * z
```

Para ello, marcaremos los `tensores` de los cuales queremos calcular derivadas con el parámetro `requires_grad=True`. Llamado posteriormente a la función `backward` sobre el `tensor` resultante, `autograd` calculará las derivadas como hemos hecho antes manualmente, pero de manera automática, y las almacenará en el atributo `grad` de cada `tensor`.

Python

```
g.backward()
```

Python

```
z.grad # x + y
```

Devuelve: *tensor(3.)*

Python

```
x.grad # z
```

Devuelve: *tensor(3.)*

Python

```
y.grad # z
```

Devuelve: *tensor(3.)*

Como puedes ver, el `grafo computacional` es una herramienta extraordinaria para diseñar `redes neuronales` de complejidad arbitraria. Con una simple función podemos calcular todas las derivadas de manera sencilla (cada nodo que representa una operación solo necesita calcular su propia derivada de manera local) y optimizar el modelo con nuestro algoritmo de gradiente preferido.

Añadiendo `autograd` encima de `NumPy`, `Pytorch` nos ofrece todo lo que necesitamos para diseñar y entrenar `redes neuronales`. Puedes aprender más sobre `autograd` aquí. Sin embargo, si queremos entrenar redes muy grandes o utilizar *datasets* muy grandes (o ambas cosas a la vez), el proceso de entrenamiento será muy lento. Es aquí donde entra en juego el último elemento que hace de `Pytorch` lo que es.

Aceleración por GPU

La última pieza que nos falta explorar es la posibilidad de ejecutar nuestro código en `GPU`. Puedes comprobar si `Pytorch` tiene acceso a `GPU` con el siguiente código:

Python

```
torch.cuda.is_available()
```

Devuelve: *True*

En caso de no tener acceso (y asumiendo que efectivamente tu ordenador disponga de una GPU compatible), es probable que necesites reinstalar `Pytorch` asegurándote que usas una versión compatible con `GPU` o bien tengas que instalar o actualizar los *drivers* de tu tarjeta gráfica. `Pytorch` tiene soporte para `GPUs` de las marcas `Nvidia` y `AMD`, así como las GPUs integradas en `Mac` y otros dispositivos. Aun así, el soporte para `NVIDIA` suele ser el más estable y recomendado.

En el caso de tener acceso `GPU`, será tan sencillo como crear nuestros `tensores` en la `GPU` y ejecutar las operaciones de la misma manera. ¡Super sencillo!

Python

```
x = torch.randn(3, 3, device="cuda")
y = torch.randn(3, 3, device="cuda")

x * y
```

Devuelve:
tensor([[9.8747e-02, 8.7117e-05, -2.5572e+00],
[-3.9884e-01, 1.0041e+00, -7.7360e-01],
[-6.9971e-01, 2.5536e+00, 7.4995e-01]], device='cuda:0')

Las siguientes son todas formas válidas de crear un `tensor` en la `GPU`.

Python

```
device = torch.device("cuda")           # device = "cuda" también sirve

x = torch.randn(3, 3, device=device)    # crea el tensor en la GPU

x = torch.randn(3, 3)
x = x.to(device)                        # mueve el tensor a la GPU (menos
     eficiente)
x = x.cuda()                            # mueve el tensor a la GPU (menos
     eficiente)

device = "cuda:0"                       # selecciona la primera GPU, si
    hay más de una - "cuda:1", "cuda:2", etc.
x = torch.randn(3, 3, device=device)
x
```

Devuelve:

tensor([[0.7451, -0.2976, -0.8867],
[-0.3100, -1.1565, 1.5185],
[-0.0100, -0.8342, -0.3056]], device='cuda:0')

Puedes copiar un tensor de la `GPU` a la `CPU` con la función `cpu`.

Python

```
device = torch.device("cpu")

x = x.cpu()
x = x.to("cpu")
x = x.to(device)
```

El siguiente ejemplo ilustra por qué es importante ejecutar nuestro código en `GPU`. En este caso, vamos a calcular el tiempo que tarda en ejecutarse la multiplicación de dos matrices grandes.

Python

```
# en cpu

x = torch.randn(10000,10000)
y = torch.randn(10000,10000)

%time z = x*y
```

Devuelve:
CPU times: user 89.2 ms, sys: 250 ms, total: 339 ms
Wall time: 35.7 ms

Python

```
# en gpu

x = torch.randn(10000,10000).cuda()
y = torch.randn(10000,10000).cuda()

%time z = x*y
```

Devuelve:
CPU times: user 0 ns, sys: 18.9 ms, total: 18.9 ms Wall time: 18.9 ms

Como veremos más adelante, durante el entrenamiento de una red neuronal, la mayoría del tiempo se pasa en las multiplicaciones de matrices, por lo que acelerar este proceso es fundamental para poder entrenar redes neuronales grandes en tiempos razonables. Este es el motivo por el que los modelos más grandes que se entrenan hoy en día utilizan superordenadores con cientos de miles de GPUs (lo cual, a su vez, cuesta millones de euros).

Por otro lado, la `GPU` no es el único acelerador *hardware* que se utiliza para entrenar redes neuronales. Google desarrolla sus propios chips de entrenamiento, llamados `TPU`, más rápidos y eficientes que las `GPUs`, pero que solo sirven para entrenar redes neuronales (las `GPUs` se utilizan también, entre otras cosas, para renderizar gráficos en videojuegos). Otras empresas desarrollando sus propios chips de entrenamiento, además de `Intel` y `AMD`, incluyen `Tesla`, con su chip `Dojo`, así como multitud de `startups` proclamando las bondades de sus chips de IA, como `Groq` o `Cerebras`. Esto se debe a la inmensa demanda por capacidad computacional que existe y la limitada oferta de `GPUs` y `TPUs` que existen en el mercado. Como suele decirse: «Durante una fiebre del oro, vende palas».

> Nota del autor: si sabes fabricar chips, por favor, ponte en contacto conmigo. Es por un tema…

Redes Neuronales

Una vez introducidos los conceptos básicos de `Pytorch`, vamos a ver otras de las funcionalidades que esta librería nos ofrece para diseñar y entrenar `redes neuronales`. En esta sección veremos el módulo `nn`, que contiene gran parte de la funcionalidad para definir nuestros modelos y utilidades para el entrenamiento.

Volviendo a nuestro ejemplo inicial, las siguientes líneas de código nos servían para crear un modelo.

Python

```
from torch.nn import Sequential as S
from torch.nn import Linear as L
from torch.nn import ReLU as R


model = S(L(784,128),R(),L(128,10))
model
```

Devuelve:

Sequential(
(0): Linear(in_features=784, out_features=128, bias=True)
(1): ReLU()
(2): Linear(in_features=128, out_features=10, bias=True)
)

Como puedes ver, hemos importado las clases `Sequential`, `Linear` y `ReLU` del módulo `torch.nn` y las hemos combinado para crear una red neuronal. Por un lado, la clase `Linear` nos ofrece una implementación de un tipo de capa muy común en redes neuronales, la capa `fully-connected` o `densa` (y que veremos en detalle en el siguiente capítulo, ¡en el que incluso harás tu propia implementación!). Por otro lado, la clase `ReLU` nos ofrece una implementación de la función de activación `ReLU`, un elemento esencial en las redes neuronales (que, de nuevo, aprenderás en el siguiente capítulo). Por último, la clase `Sequential` nos permite combinar varias capas en una única capa, de manera que la salida de una capa es la entrada de la siguiente, de forma secuencial. A la hora de instanciar nuestras capas, en especial la capa lineal, deberemos indicar una serie de argumentos para definir las dimensiones y que representarán en última instancia la capacidad de nuestro modelo, cuántos parámetros tiene.

Una vez definido el modelo, podemos pasarle un tensor de entrada para obtener la salida.

Python

```python
import torch

x = torch.randn(10, 784)
y_hat = model(x)
preds = torch.argmax(y_hat, dim=1)

y_hat.shape, preds
```

Devuelve: *(torch.Size([10, 10]), tensor([5, 9, 7, 6, 5, 1, 9, 0, 9, 5]))*

En función de las capas que formen nuestro modelo, necesitaremos pasarle un tipo u otro de entradas, obteniendo a su vez un tipo en concreto de salidas. En este caso, la capa lineal de `Pytorch` espera un tensor de entrada de dos dimensiones, con la primera dimensión representando el número de muestras y la segunda dimensión el número de características de cada muestra (que en nuestro ejemplo inicial es el valor de cada pixel en una imagen). La salida de la capa lineal será un tensor de dos dimensiones, con la primera dimensión representando el número de muestras y la segunda dimensión el número de neuronas, o dimensión oculta (128 en la primera capa y 10 en la segunda). Una vez obtenida la salida, aplicamos la función `argmax` para obtener el índice del valor más alto en cada fila, que será nuestra predicción final. No te preocupes si no entiendes del todo estos conceptos, en el siguiente capítulo profundizaremos en ellos.

Modelos secuenciales

Como hemos visto, la forma más sencilla de definir una `red neuronal` en `Pytorch` es utilizando la clase `Sequential`. Esta clase nos permite definir una secuencia de capas, que se aplicarán de manera secuencial (las salidas de una capa serán la entrada de la siguiente). El siguiente es un ejemplo de implementación alternativa al modelo de nuestro ejemplo:

Python

```
    D_in, H, D_out = 784, 100, 10

    model = torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out),
    )

    model
```

Devuelve:
Sequential(
(0): Linear(in_features=784, out_features=100, bias=True)
(1): ReLU()
(2): Linear(in_features=100, out_features=10, bias=True)
)

El modelo anterior se conoce como Perceptrón Multicapa, o `MLP` por sus siglas en inglés. Tiene 784 entradas, una capa oculta de 100 neuronas y 10 salidas. Para *ejecutar* el modelo, podemos llamarlo como de si una función se tratase, pasando como argumento el tensor con los *inputs*.

> La capa de tipo `Linear` espera un tensor de 2 dimensiones, donde la primera es la dimensión del `batch` que puede ser arbitraria y la segunda tiene que coincidir con el número de neuronas especificado, en nuestro ejemplo 784 en la primera capa y 100 en la segunda.

Python

```
    outputs = model(torch.randn(64, 784))
    outputs.shape
```

Devuelve: *torch.Size([64, 10])*

De la misma manera que hemos visto antes con los tensores, podemos enviar nuestro modelo a la `GPU` para acelerar las operaciones internas.

Python

```
model.cuda()
x = torch.randn(64, 784).cuda()

outputs = model(x)
outputs.shape, outputs.device
```

Devuelve: *(torch.Size([64, 10]), device(type='cuda', index=0))*

Modelos personalizados

Si bien los modelos secuenciales son útiles para definir redes neuronales sencillas, en la práctica casi siempre necesitaremos definir redes más complejas. Para ello, podemos definir nuestras propias clases que hereden de la clase `Module` de `Pytorch`. Esta clase nos permite definir modelos de manera más flexible, ya que nos permite diseñar la lógica de ejecución del modelo a nuestro gusto.

Python

```
class Model(torch.nn.Module):

    # constructor
    def __init__(self, D_in=784, H=100, D_out=10):

        # llamamos al constructor de la clase madre
        super(Model, self).__init__()

        # definimos nuestras capas
        self.fc1 = torch.nn.Linear(D_in, H)
        self.relu = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(H, D_out)

    # lógica para calcular las salidas de la red
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

En primer lugar, necesitamos definir una nueva clase que herede de la clase `torch.nn.Module`. Esta clase madre aportará toda la funcionalidad esencial que necesita una `red neuronal` (soporte GPU, iterar por sus parámeteros, etc.). Luego, en esta clase necesitamos definir como mínimo dos funciones:

- `__init__`: en el constructor definiremos todas las capas que querramos usar en la red, así como cualquier otro parámetro necesario.
- `forward`: en esta función definimos toda la lógica que aplicaremos desde que recibimos los *inputs* hasta que devolvemos los *outputs*.

En el ejemplo anterior simplemente hemos replicado la misma red (puedes conseguir el mismo efecto usando la clase `Sequential`).

Python

```
model = Model(784, 100, 10)
outputs = model(torch.randn(64, 784))
outputs.shape
```

Devuelve: *torch.Size([64, 10])*

Datasets y *Dataloaders*

A la hora de entrenar una red neuronal, necesitaremos un conjunto de datos sobre el que entrenar. Para ello, `Pytorch` nos ofrece funcionalidad para su creación e iteración de manera optimizada. Vamos a ver un ejemplo usando el conjunto de datos `MNIST`, que podemos descargar usando `Scikit-Learn`, como hemos visto en nuestro ejemplo.

Python

```
from sklearn.datasets import fetch_openml
import numpy as np

mnist = fetch_openml('mnist_784', version=1)
X, Y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

En nuestro ejemplo inicial, iterábamos directamente sobre los tensores de `Pytorch` durante el entrenamiento.

Python

```
X_train, X_test = torch.from_numpy(X[:60000] / 255.), torch.from_numpy(X
    [60000:] / 255.)
Y_train, Y_test = torch.from_numpy(Y[:60000]), torch.from_numpy(Y
    [60000:])

bs = 32
num_batches = len(X_train) // bs

for epoch in range(1):
    for b in range(num_batches):
        x = X_train[b*bs:(b+1)*bs] # usamos los datos en lotes (batches)
        y = Y_train[b*bs:(b+1)*bs]
        y_hat = model(x)

        print(x.shape, y.shape, y_hat.shape)

        break
```

Devuelve: *torch.Size([32, 784]) torch.Size([32]) torch.Size([32, 10])*

Si bien esto es totalmente factible, `Pytorch` ofrece los objetos `Dataset` y `DataLoader` para facilitar la creación y el acceso a los datos de manera optimizada (por ejemplo, creando los `batches` en paralelo y cargando los datos en la `GPU` de manera automática).

El `DataLoader`

El `DataLoader` es un objeto que nos permite iterar nuestro *dataset* en `batches` de manera eficiente. Podemos pasarle como argumento cualquier iterador, desde una lista de `Python` hasta un `array` de `NumPy` o un `tensor` de `Pytorch`.

Python

```
dataloader = torch.utils.data.DataLoader(X, batch_size=100)

for batch in dataloader:
    print(batch.shape)
    break
```

Devuelve: *torch.Size([100, 784])*

Tiene varias opciones interesantes que nos permitirán mejorar la eficiencia de nuestro entrenamiento.

Python

```
dataloader = torch.utils.data.DataLoader(
    X,                      # datos
    batch_size=100,         # tamaño del batch, número de imágenes por
        iteración
    shuffle=True,           # barajamos los datos antes de cada epoch
    num_workers=4,          # número de procesos que se lanzan para
        cargar los datos (número de cores de la CPU para carga en
        paralelo)
    pin_memory=True,        # si tenemos una GPU, los datos se cargan en
        la memoria de la GPU
    collate_fn=None,        # función para combinar los datos de cada
        batch
)
```

El `Dataset`

Si bien el `dataloader` es capaz de trabajar con cualquier iterador, `Pytorch` nos ofrece una clase base para crear nuestros propios *datasets*:

Python

```
class Dataset(torch.utils.data.Dataset):

    # constructor
    def __init__(self, X, Y):
        self.X = torch.tensor(X).float()
        self.Y = torch.tensor(Y).long()

    # cantidad de muestras en el dataset
    def __len__(self):
        return len(self.X)

    # devolvemos el elemento `ix` del dataset
    def __getitem__(self, ix):
        return self.X[ix], self.Y[ix]

    # opcionalmente, podemos definir una función para generar un batch
    def collate_fn(self, batch):
        x, y = [], []
        for _x, _y in batch:
            x.append(_x)
            y.append(_y)
        return torch.stack(x).view(-1,28,28), torch.stack(y) #
            convertimos x en una imagen de 28x28
```

Para ello crearemos una nueva clase que hereda de `torch.utils.data.Dataset` y definiremos estas tres funciones:

- `__init__`: el constructor
- `__len__`: devuelve el número de muestras en el *dataset*
- `__getitem__`: devuelve una muestra en concreto del *dataset*

Cada vez que nuestro `dataloader` necesite una nueva muestra, llamará a la función `__getitem__` pasándole el índice de la muestra que necesita. Aquí podremos definir cualquier lógica de carga y procesado de datos (por ejemplo, leer imágenes y aplicar transformaciones).

Python

```
dataset = Dataset(X, Y)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=100)

for batch in dataloader:
    x, y = batch
    print(x.shape, y.shape)
    break
```

Devuelve: *torch.Size([100, 784]) torch.Size([100])*

La función `collate_fn` es opcional, pero nos dará control sobre cómo combinar las muestras en un `batch`.

Python

```
dataset = Dataset(X, Y)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=100,
    collate_fn=dataset.collate_fn)

for batch in dataloader:
    x, y = batch
    print(x.shape, y.shape)
    break
```

Devuelve: *torch.Size([100, 28, 28]) torch.Size([100])*

Entrenamiento

Como te puedes imaginar, `Pytorch` nos oferece funcionalidad para poder entrenar redes neuronales. Para empezar, el módulo `nn` ofrece funciones de pérdida que utilizaremos para medir el error de nuestro modelo durante el entrenamiento, calcular las derivadas de

este error con respecto a los parámetros de nuestro modelo y actualizar dichos parámetros en la dirección que nos permita minimizar el error. Por otro lado, el módulo `optim` nos ofrece implementaciones de diferentes algoritmos de optimización que podemos utilizar para actualizar los parámetros de nuestro modelo.

Con estos conceptos podemos entender el bucle de entrenamiento de nuestro ejemplo inicial.

Python

```python
from tqdm import tqdm


# preparamos los datos
X_train, X_test = torch.from_numpy(X[:60000] / 255.), torch.from_numpy(X
    [60000:] / 255.)
y_train, y_test = torch.from_numpy(Y[:60000]), torch.from_numpy(Y
    [60000:])
bs = 32
num_batches = len(X_train) // bs

loss_fn = torch.nn.CrossEntropyLoss() # función de pérdida
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3) # optimizador

for epoch in range(10):
    for b in tqdm(range(num_batches)):
        x = X_train[b*bs:(b+1)*bs]
        y = y_train[b*bs:(b+1)*bs]
        y_hat = model(x)                # calculamos la salida del modelo
        loss = loss_fn(y_hat, y)        # calculamos la función de pé
            rdida
        optimizer.zero_grad()           # ponemos a cero los gradientes (
            si no se acumulan)
        loss.backward()                 # calculamos los gradientes
        optimizer.step()                # actualizamos los parámetros
    print(f"Epoch {epoch+1} loss: {loss.item():.3f}")
```

Devuelve:

```
100%|▓▓▓▓▓▓▓▓▓▓| 1875/1875 [00:01<00:00, 1328.97it/s]

Epoch 1 loss: 0.047

100%|▓▓▓▓▓▓▓▓▓▓| 1875/1875 [00:01<00:00, 1274.10it/s]
```

```
Epoch 2 loss: 0.034

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1259.36it/s]

Epoch 3 loss: 0.026

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1253.15it/s]

Epoch 4 loss: 0.020

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1259.32it/s]

Epoch 5 loss: 0.020

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1258.17it/s]

Epoch 6 loss: 0.015

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1256.40it/s]

Epoch 7 loss: 0.011

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1250.27it/s]

Epoch 8 loss: 0.005

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1255.98it/s]

Epoch 9 loss: 0.005

100%| ▓▓▓▓▓▓▓▓▓▓ | 1875/1875 [00:01<00:00, 1255.01it/s]
Epoch 10 loss: 0.002
```

Si todo va bien, deberíamos ver cómo el error de nuestro modelo disminuye a medida que entrenamos.

Un ejemplo más completo

Vamos a mejorar nuestro ejemplo introduciendo los conceptos que hemos aprendido hasta ahora.

En primer lugar, definiremos nuestro modelo como una clase que hereda de `torch.nn.Module` y que implementa la lógica de nuestro modelo en la función `forward`.

Python

```python
class Model(torch.nn.Module):

    # constructor
    def __init__(self, D_in=784, H=100, D_out=10):

        # llamamos al constructor de la clase madre
        super(Model, self).__init__()

        # definimos nuestras capas
        self.fc1 = torch.nn.Linear(D_in, H)
        self.relu = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(H, D_out)

    # lógica para calcular las salidas de la red
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

En segundo lugar, iteraremos sobre nuestro *dataset* usando un `DataLoader` para cargar los datos en `batches` de manera eficiente a partir de nuestro `Dataset`.

Python

```python
class Dataset(torch.utils.data.Dataset):

    # constructor
    def __init__(self, X, Y):
        self.X = torch.tensor(X).float()
        self.Y = torch.tensor(Y).long()

    # cantidad de muestras en el dataset
    def __len__(self):
        return len(self.X)

    # devolvemos el elemento `ix` del dataset
    def __getitem__(self, ix):
        return self.X[ix], self.Y[ix]
```

Poniendo todo junto, tenemos el siguiente código.

Python

```
# instanciamos nuestro dataset y dataloader
dataset = {
    "train": Dataset(X[:60000], Y[:60000]),
    "val": Dataset(X[60000:], Y[60000:])
}
dataloader = {
    'train': torch.utils.data.DataLoader(dataset['train'], batch_size
        =100),
    'val': torch.utils.data.DataLoader(dataset['val'], batch_size=100)
}


# instanciamos nuestro modelo
model = Model(784, 100, 10)


# definimos la función de pérdida y el optimizador
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())


# bucle de entrenamiento
epochs = 5
for e in range(1, epochs+1):
    print(f"epoch: {e}/{epochs}")

    # entrenamiento
    model.train()
    for batch_ix, (x, y) in enumerate(dataloader['train']):
        optimizer.zero_grad()
        outputs = model(x)
        loss = criterion(outputs, y)
        loss.backward()
        optimizer.step()
        if batch_ix % 100 == 0:
            loss, current = loss.item(), (batch_ix + 1) * len(x)
            print(f"loss: {loss:.4f} [{current:>5d}/{len(dataset['train
                ']):>5d}]")

    # validación
    model.eval()
    val_loss, val_acc = [], []
    with torch.no_grad():
        for batch_ix, (x, y) in enumerate(dataloader['val']):
            outputs = model(x)
            loss = criterion(outputs, y)
            val_loss.append(loss.item())
            val_acc.append((outputs.argmax(1) == y).float().mean().item()
                )
```

```
        print(f"val_loss: {np.mean(val_loss):.4f} val_acc: {np.mean(val_acc)
            :.4f}")
```

```
    epoch: 1/5
    loss: 31.3980 [  100/60000]
    loss: 0.8569 [10100/60000]
    loss: 0.4440 [20100/60000]
    loss: 0.3492 [30100/60000]
    loss: 0.2278 [40100/60000]
    loss: 0.3071 [50100/60000]
    val_loss: 0.2355 val_acc: 0.9393
    epoch: 2/5
    loss: 0.1880 [  100/60000]
    loss: 0.3158 [10100/60000]
    loss: 0.3040 [20100/60000]
    loss: 0.3370 [30100/60000]
    loss: 0.1464 [40100/60000]
    loss: 0.2595 [50100/60000]
    val_loss: 0.2108 val_acc: 0.9460
    epoch: 3/5
    loss: 0.1863 [  100/60000]
    loss: 0.3656 [10100/60000]
    loss: 0.2138 [20100/60000]
    loss: 0.2807 [30100/60000]
    loss: 0.1452 [40100/60000]
    loss: 0.1479 [50100/60000]
    val_loss: 0.1884 val_acc: 0.9525
    epoch: 4/5
    loss: 0.1262 [  100/60000]
    loss: 0.2856 [10100/60000]
    loss: 0.1984 [20100/60000]
    loss: 0.2441 [30100/60000]
    loss: 0.1706 [40100/60000]
    loss: 0.1525 [50100/60000]
    val_loss: 0.1892 val_acc: 0.9553
    epoch: 5/5
    loss: 0.1330 [  100/60000]
    loss: 0.2545 [10100/60000]
    loss: 0.1669 [20100/60000]
    loss: 0.0918 [30100/60000]
    loss: 0.1125 [40100/60000]
    loss: 0.2416 [50100/60000]
    val_loss: 0.1926 val_acc: 0.9537
```

Donde, además, hemos añadido funcionalidad para evaluar nuestro modelo en los datos de validación durante el entrenamiento. Como puedes ver, tenemos que poner el modelo en modo entrenamiento con `model.train()` o en modo evaluación con `model.eval()`, ya que, en función de las capas que usemos, el comportamiento de nuestro modelo puede variar.

Además, durante la evaluación, hemos incluido el contexto `torch.no_grad()` para indicarle a `Pytorch` que no calcule las derivadas de las operaciones que se lleven a cabo dentro de este contexto, porque no las necesitaremos para evaluar el modelo y esto nos permitirá ahorrar tiempo de computación.

Consejos para mejorar las prestaciones

El siguiente ejemplo aprovecha otras de las caracterísitcas de `Pytorch` para mejorar las prestaciones de nuestro entrenamiento.

Python

```
dataset = {
    "train": Dataset(X[:60000], Y[:60000]), # 60.000 imágenes para
        entrenamiento
    "val": Dataset(X[60000:], Y[60000:])    # 10.000 imágenes para
        validación
}

# aumentamos el tamaño del batch para aprovechar la GPU, carga en
    paralelo y movemos los datos a la GPU
dataloader = {
    'train': torch.utils.data.DataLoader(dataset['train'], batch_size
        =1000, shuffle=True, num_workers=4, pin_memory=True),
    'val': torch.utils.data.DataLoader(dataset['val'], batch_size=1000,
        num_workers=4, pin_memory=True)
}

model = Model(784, 100, 10)

# compilamos el modelo

# model = torch.compile(model)

# torch.backends.cuda.matmul.allow_tf32 = True # allow tf32 on matmul

# torch.backends.cudnn.allow_tf32 = True # allow tf32 on cudnn

# movemos el modelo a la GPU
model.cuda()

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())
epochs = 5
for e in range(1, epochs+1):
    print(f"epoch: {e}/{epochs}")

    # entrenamiento
    model.train()
```

```
    for batch_ix, (x, y) in enumerate(dataloader['train']):
        # movemos los datos a la GPU
        x, y = x.cuda(), y.cuda()
        optimizer.zero_grad()
        # automatic mixed precision
        with torch.autocast(device_type='cuda', dtype=torch.bfloat16):
            outputs = model(x)
            loss = criterion(outputs, y)
        loss.backward()
        optimizer.step()
        if batch_ix % 10 == 0:
            loss, current = loss.item(), (batch_ix + 1) * len(x)
            print(f"loss: {loss:.4f} [{current:>5d}/{len(dataset['train
                '']):>5d}]")

    # validación
    model.eval()
    val_loss, val_acc = [], []
    with torch.no_grad():
        for batch_ix, (x, y) in enumerate(dataloader['val']):
            # movemos los datos a la GPU
            x, y = x.cuda(), y.cuda()
            # automatic mixed precision
            with torch.autocast(device_type='cuda', dtype=torch.bfloat16)
                :
                outputs = model(x)
                loss = criterion(outputs, y)
            val_loss.append(loss.item())
            val_acc.append((outputs.argmax(1) == y).float().mean().item()
                )
    print(f"val_loss: {np.mean(val_loss):.4f} val_acc: {np.mean(val_acc)
        :.4f}")
```

```
epoch: 1/5


loss: 34.7140 [ 1000/60000]
loss: 1.8635 [11000/60000]
loss: 1.2503 [21000/60000]
loss: 0.8488 [31000/60000]
loss: 0.5767 [41000/60000]
loss: 0.5361 [51000/60000]
val_loss: 0.4539 val_acc: 0.8898
epoch: 2/5
loss: 0.4240 [ 1000/60000]
loss: 0.4112 [11000/60000]
loss: 0.4003 [21000/60000]
loss: 0.3724 [31000/60000]
loss: 0.2522 [41000/60000]
loss: 0.3241 [51000/60000]
val_loss: 0.3059 val_acc: 0.9148
```

```
epoch: 3/5
loss: 0.3350 [ 1000/60000]
loss: 0.2639 [11000/60000]
loss: 0.2016 [21000/60000]
loss: 0.2233 [31000/60000]
loss: 0.2759 [41000/60000]
loss: 0.2234 [51000/60000]
val_loss: 0.2445 val_acc: 0.9324
epoch: 4/5
loss: 0.2122 [ 1000/60000]
loss: 0.1563 [11000/60000]
loss: 0.1945 [21000/60000]
loss: 0.1995 [31000/60000]
loss: 0.1573 [41000/60000]
loss: 0.1716 [51000/60000]
val_loss: 0.2098 val_acc: 0.9421
epoch: 5/5
loss: 0.1725 [ 1000/60000]
loss: 0.1717 [11000/60000]
loss: 0.2167 [21000/60000]
loss: 0.1672 [31000/60000]
loss: 0.2144 [41000/60000]
loss: 0.1525 [51000/60000]
val_loss: 0.1907 val_acc: 0.9476
```

Usar la `GPU` acelerará de manera dramática nuestro entrenamiento, pero no es la única manera de mejorar las prestaciones. Otras técnicas incluyen: generar lo *batches* en paralelo (usando el parámtero `num_workers` del `DataLoader` al número de cores de tu `CPU`), compilar el modelo con `torch.compile` (introducido en la versión 2 de Pytorch), entrenar el modelo usando precisión mixta… En resumen, hay muchas técnicas que podemos usar para mejorar las prestaciones de nuestro entrenamiento, y que veremos en más detalle en el capítulo 6. Aun así, estas optimizaciones no harán más que complicar nuestro código, incrementando las posibilidades de introducir errores. Es por este motivo que utilizar librerías como `Keras` o `Pytorch Lightning` son muy útiles para abstraernos de estos detalles y centrarnos en el diseño de nuestro modelo.

Modelos en producción

Una vez entrenado tu modelo es posible que quieras exportarlo para usarlo en otro proyecto o incluso en producción. Para ello, `Pytorch` nos ofrece diferentes opciones para exportar nuestro modelo. La forma más sencilla es usar la función `torch.save`, que nos permite guardar el modelo en un fichero.

Python

```
    torch.save(model, 'model.pth')
```

Python

```
    loaded = torch.load('model.pth')
    loaded
```

Devuelve:

Model(
(fc1): Linear(in_features=784, out_features=100, bias=True)
(relu): ReLU()
(fc2): Linear(in_features=100, out_features=10, bias=True)
)

Una alternativa más eficiente y flexible consiste en explorar el `state_dict`.

Python

```
    torch.save(model.state_dict(), 'model.ckpt')

    model = Model(784, 100, 10)
    model.load_state_dict(torch.load('model.ckpt'))
    model
```

Devuelve:

Model(
(fc1): Linear(in_features=784, out_features=100, bias=True)
(relu): ReLU()
(fc2): Linear(in_features=100, out_features=10, bias=True)
)

Además del modelo podemos guardar el `state_dict` del optimizador para poder continuar el entrenamiento en otro momento, por lo que es muy útil para guardar `checkpoints`.

En ambos casos vas a necesitar el mismo código usado a la hora de exportar el modelo para poder cargarlo. Esto puede ser un inconveniente en muchos casos, por lo que una buena alternativa es usar `torch.script` para serializar el modelo.

Python

```
    model_scripted = torch.jit.script(model)
    model_scripted.save('model_scripted.pt')
```

Python

```
    model = torch.jit.load('model_scripted.pt')
    model
```

Devuelve:

RecursiveScriptModule(
original_name=Model
(fc1): RecursiveScriptModule(original_name=Linear)
(relu): RecursiveScriptModule(original_name=ReLU)
(fc2): RecursiveScriptModule(original_name=Linear)
)

Esta aproximación permite desacoplar el código de los pesos del modelo, sin embargo, vamos a seguir necesitando `Pytorch` para poder cargar el modelo (lo cual no siempre será posible en función del entorno de producción del modelo). Para ello, podemos usar `ONNX` para exportar el modelo a un formato estándar que puede ser usado por cualquier *framework*.

Python

```
    torch.onnx.export(
        model.cpu(),
        torch.randn(10, 784),
        'model.onnx',
        opset_version=11,
        input_names=['input'],
        output_names=['output'],
        dynamic_axes={'input': {0: 'batch_size'}, 'output': {0: 'batch_size'}
    })
```

Python

```
    import onnx

    onnx_model = onnx.load('model.onnx')
    onnx.checker.check_model(onnx_model)
```

En `Python` podemos usar la librería `onnxruntime` para cargar el modelo y usarlo en producción sin necesidad de que `Pytorch` esté instalado (funciona directamente con `NumPy`).

Python

```python
import onnxruntime as ort

ort_session = ort.InferenceSession('model.onnx')
ort_inputs = {ort_session.get_inputs()[0].name: torch.randn(16, 784).
    numpy()}
ort_outs = ort_session.run(None, ort_inputs)
ort_outs[0].shape
```

Devuelve: *(16, 10)*

Además, existen *runtimes* para otros lenguajes de programación, así como para dispositivos móviles y microcontroladores que te permitirán ejectuar tus modelos en cualquier entorno.

El Ecosistema de Pytorch

`Pytorch` es un proyecto muy activo, con una gran comunidad de desarrolladores que contribuyen a su mejora. Además, cuenta con una gran cantidad de librerías que nos permiten extender sus funcionalidades. Algunos ejemplos son:

- `Torchvision`: librería de visión por computador para `Pytorch`, puedes aprender más sobre ella en el siguiente enlace.
- `Torchtext`: librería de procesamiento de lenguaje natural para `Pytorch`, puedes aprender más sobre ella en el siguiente enlace.
- `Huggingface`: gran ecosistema conocido por su librería de `transformers`, pero que poco a poco ha evolucionado incluyendo muchas otras funcionalidades, puedes aprender más sobre ella en el siguiente enlace.
- `Torhcmetrics`: librería de métricas para `Pytorch`.
- Pytorch Lightning recomendada para entrenar modelos de una manera más sencilla y modular.

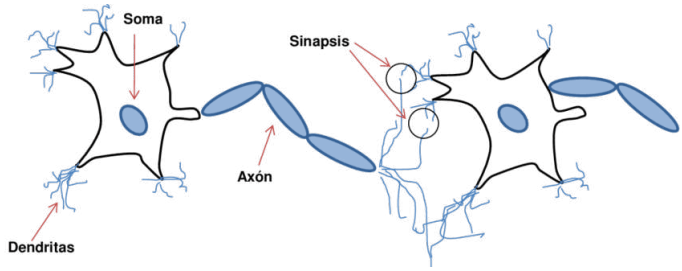
Además, si estás interesado en una implementación de algún modelo en concreto, un rápida búsqueda en `Google` te llevará a multitud de implementaciones de modelos en `Pytorch` que puedes usar como punto de partida.

4. Fundamentos del Aprendizaje Profundo

Bienvenido a mi capítulo favorito del libro. Hasta ahora nos hemos centrado en aprender las principales herramientas que nos permiten diseñar, entrenar y poner en producción modelos de Aprendizaje Profundo, pero ahora toca ponerse en serio y entender cómo y por qué aprenden las redes neuronales a llevar a cabo una tarea.

El Perceptrón

El `Perceptrón` es la unidad de computación básica utilizada en las redes neuronales, aunque también puede usarse por sí solo como algoritmo de IA en algunos casos. Está inspirado en el funcionamiento de las neuronas biológicas, y fue propuesto en 1957 por Frank Rosenblatt.



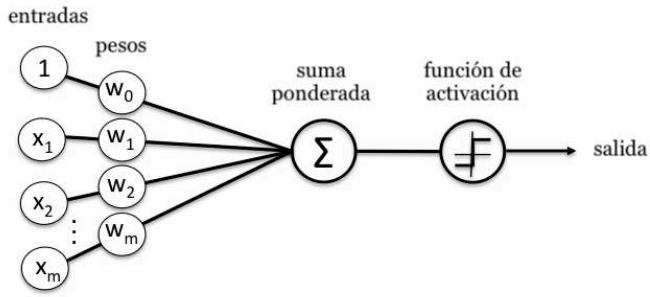
Una neurona está compuesta de un cuerpo celular que contiene el núcleo y la mayoría de los componentes complejos de la célula, muchas extensiones ramificadas llamadas dendritas y una extensión muy larga llamada axón. Cerca de su extremidad, el axón se divide en ramas llamadas telodendria, y en la punta de estas ramas encontramos estructuras minúsculas llamadas terminales sinápticas (o simplemente sinapsis), que están conectadas a las dendritas o cuerpos celulares de otras neuronas. Las neuronas biológicas producen impulsos eléctricos cortos llamados potenciales de acción (o simplemente señales) que viajan a lo largo de los axones y hacen que las sinapsis liberen señales químicas llamadas

neurotransmisores. Cuando una neurona recibe una cantidad suficiente de estos neurotransmisores en unos pocos milisegundos, dispara sus propios impulsos eléctricos. Aunque las neuronas individuales parecen comportarse de manera bastante simple, pueden conectarse a miles de otras neuronas formando redes complejas que pueden realizar tareas complejas.

El `Perceptrón` es una versión simplificada de la neurona que calcula la suma ponderada de todas sus entradas y después aplica una `función de activación` para dar el resultado:

$$\hat{y} = f(\mathbf{w} \cdot \mathbf{x}) = f(w_0 + w_1 x_1 + ... + w_m x_m)$$

Dónde $\hat{y}$ es la salida del `Perceptrón`, $\mathbf{w}$ son sus parámetros (también llamados los pesos), $\mathbf{x}$ son las entradas y f es la función de activación. A continuación, puedes ver un esquema del `Perceptrón`, puedes compararlo con el esquema de la neurona biológica para evaluar sus parecidos y diferencias.

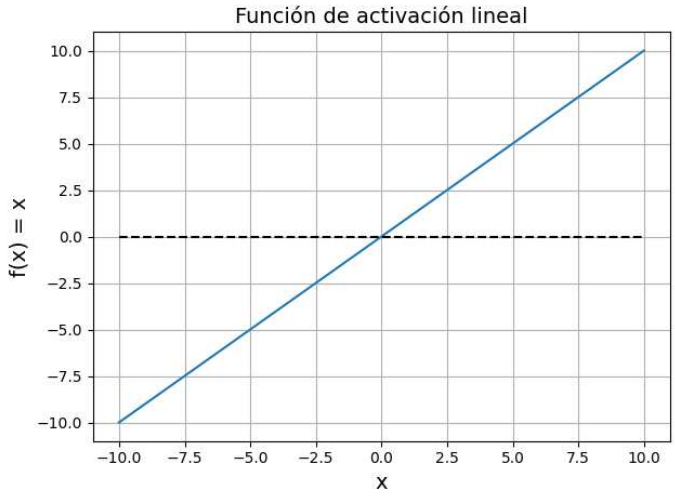


Regresión lineal

Existen varias tareas que podemos llevar a cabo con un `Perceptrón`. La primera que se propuso es la `regresión lineal`. En este tipo de tarea, como en cualquier tarea de regresión, queremos obtener un modelo que se ajuste de la mejor forma posible a un conjunto de datos determinado. En el caso de la regresión lineal, este modelo será una línea recta y el `Perceptrón`, utilizando una función de activación lineal, $f(x) = x$, es de hecho capaz de llevar a cabo esta tarea.

Python

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-10, 10)
y = x

plt.plot(x, y)
plt.grid(True)
plt.xlabel('x', fontsize=14)
plt.ylabel('f(x) = x', fontsize=14)
plt.title('Función de activación lineal', fontsize=14)
plt.plot(x, np.zeros(len(x)), '--k')
plt.tight_layout()
plt.savefig('resources/reg_lin1.png', bbox_inches='tight', pad_inches
    =0.1)
plt.close()
```

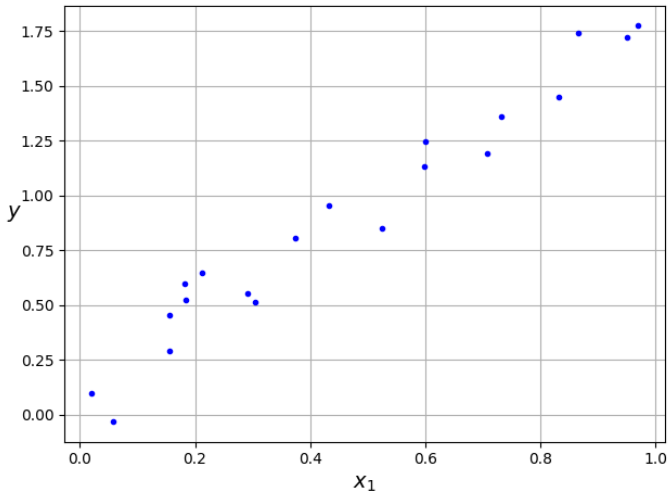


> Utilizar una función de activación lineal es como no utilizar ninguna función de activación, ya que la salida del Perceptrón será directamente el producto de las entradas por los pesos, $\hat{y} = \mathbf{w} \cdot \mathbf{x}$.

Para ilustrar cómo funciona el `Perceptrón` para tareas de regresión lineal, vamos a utilizar el siguiente conjunto de datos sintético.

Python

```
np.random.seed(42)

x = np.random.rand(20)
y = 2*x + (np.random.rand(20)-0.5)*0.5

plt.plot(x, y, "b.")
plt.xlabel("$x_1$", fontsize=14)
plt.ylabel("$y$", rotation=0, fontsize=14)
plt.grid(True)
plt.tight_layout()
plt.savefig('resources/reg_lin2.png', bbox_inches='tight', pad_inches
    =0.1)
plt.close()
```



Para empezar de manera sencilla, vamos a considerar que solo tenemos una característica que usaremos como entrada para nuestro `Perceptrón`, x_1. El objetivo es que el `Perceptrón`, al recibir cada uno de estos valores, nos dé como salida un valor lo más cercano posible a y. En este caso, al tener una sola característica por elemento, nuestro `Perceptrón` sigue la siguiente expresión:

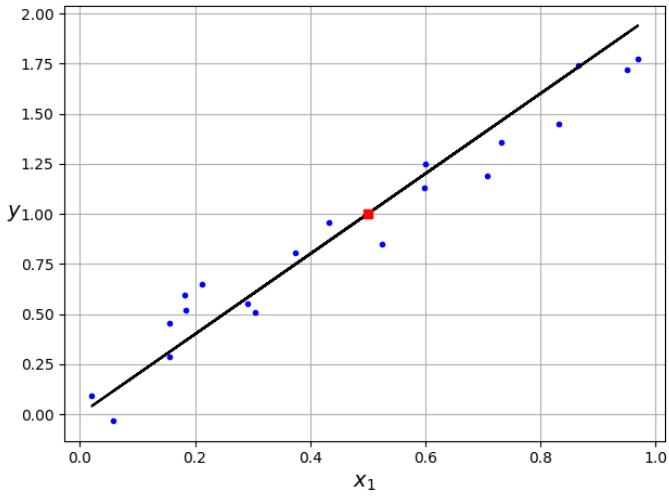
$$\hat{y} = \mathbf{w} \cdot \mathbf{x} = w_0 + w_1 x_1$$

En el caso en que $w_0 = 0$ y $w_1 = 2$, $\hat{y} = 2x$, lo cual podemos representar como una recta. En el caso de querer conocer el valor de y para un nuevo elemento, por ejemplo $x_1 = 0{,}5$,

simplemente tenemos que calcular su valor haciendo regresión a nuestra recta, en este caso $\hat{y} = 1$. Un ejemplo de aplicación de este tipo de modelos de regresión es la predicción del precio de un inmueble ($\hat{y}$) dadas una serie de características como su localización (x_1), metros cuadrados (x_2), número de habitaciones (x_3), etc. Tener un buen modelo de regresión para esta aplicación nos puede ayudar a saber si una casa en venta está por encima o por debajo de su valor real, ayudándonos a tomar mejores decisiones de inversión.

Python

```
plt.plot(x, y, "b.")
plt.plot(x, 2*x, 'k')
plt.plot(0.5, 2*0.5, 'sr')
plt.xlabel("$x_1$", fontsize=14)
plt.ylabel("$y$", rotation=0, fontsize=14)
plt.grid(True)
plt.tight_layout()
plt.savefig('resources/reg_lin3.png', bbox_inches='tight', pad_inches
    =0.1)
plt.close()
```



Sin embargo, aquí hemos hecho algo de trampa... En este ejemplo sabemos que los pesos de nuestro modelo son $w_0 = 0$ y $w_1 = 2$, ya que son los mismos utilizados para generar los datos. Nuestro objetivo será el de encontrar estos valores, y para ello utilizamos el famoso algoritmo de `descenso por gradiente`.

Descenso por Gradiente

Para poder aplicar este algoritmo necesitaremos, por un lado, un modelo paramétrico que podamos ir actualizando y, por otro lado, una medida de lo bien o mal que funciona (el error o función de pérdida). El algoritmo de descenso por gradiente para el `Perceptrón`, usando como función de pérdida el error medio cuadrático (*MSE*), se implementa de la forma siguiente:

1. Calcular la salida del modelo, $\hat{y}$.
2. Calcular la derivada de la función de pérdida con respecto a los parámetros del modelo, $\frac{\partial MSE}{\partial w} = \frac{2}{N}\frac{\partial \hat{y}}{\partial w}(\hat{y} - y)$ dónde $\frac{\partial \hat{y}}{\partial w} = x$.
3. Actualizar los parámetros, $w \leftarrow w - \eta \frac{\partial MSE}{\partial w}$, dónde η es el *learning rate*, o ratio de aprendizaje.
4. Repetir hasta converger.

Inicializaremos nuestro modelo con unos valores aleatorios para los pesos e iremos actualizando sus valores de manera iterativa en la dirección de pendiente negativa de la función de pérdida. Como puedes ver hay varios `hyperparámetros` en el proceso de optimización que van a afectar el proceso de optimización. Estos son:

- Inicialización de los pesos.
- Valor escogido del *learning rate*.
- Datos utilizados para el cálculo del gradiente.

Vamos a ver cómo afecta la elección de diferentes valores para estos parámetros.

> Llamamos `hyperparámetros` a todos aquellos parámetros que influyen en el resultado de la optimización, pero que no son resultados de la misma (como los valores iniciales, *learning rate*, etc.), mientras que los `parámetros` serían los resultados de la optimización (los pesos del modelo).

Python

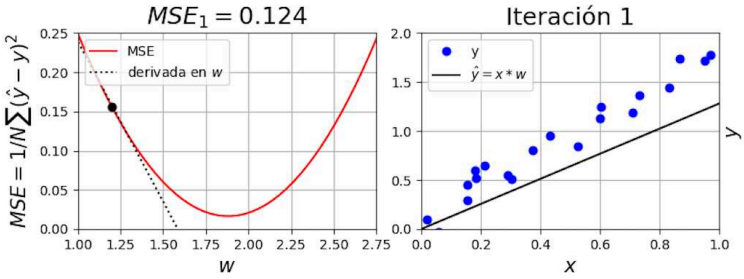
```python
def gradient(w, x, y):
    # calculamos la derivada de la función de pérdida
    # con respecto a los parámteros `w`
    dldw = x*w - y
    dydw = x
    dldw = dldw*dydw
    return np.mean(2*dldw)

def cost(y, y_hat):
    # calculamos la función de pérdida
    return ((y_hat - y)**2).mean()

def solve(epochs = 29, w = 1.2, lr = 0.2):
    # iteramos un número determinado de `epochs`
    # por cada epoch, calculamos gradientes y
    # actualizamos los pesos
    weights = [(w, gradient(w, x, y), cost(x*w, y))]
    for i in range(1,epochs+1):
        dw = gradient(w, x, y)
        w = w - lr*dw
        weights.append((w, dw, cost(x*w, y)))
    return weights
```

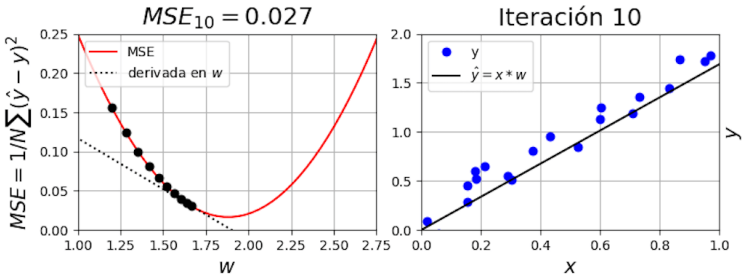
Efecto de la inicialización de los pesos

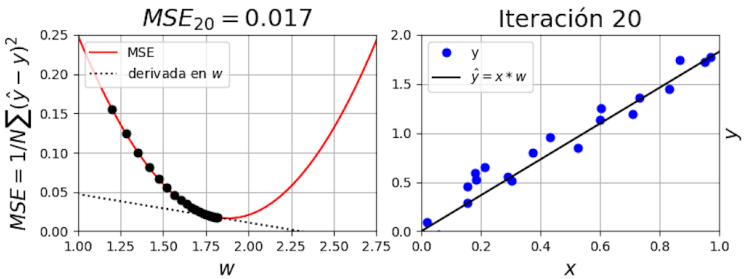
En este primer ejemplo utilizamos un valor inicial de $w_1 = 1{,}2$ y un ratio de aprendizaje de $\eta = 0{,}2$. En cada iteración calculamos la derivada de la función de pérdida con respecto al peso del modelo, la cual nos indicará la dirección en la que tenemos que actualizar su valor para reducir el error. Repetimos el proceso por un número determinado de iteraciones, lo que conocemos como `épocas`, o `epochs` en inglés.

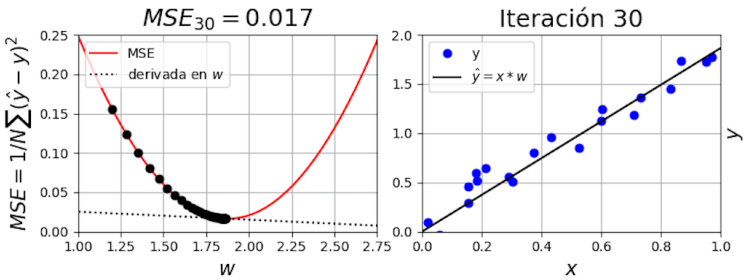
Python

```python
weights = solve(epochs = 30, w = 1.2, lr = 0.2)
```


$MSE_1 = 0.124$
Iteración 1
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_{10} = 0.027$
Iteración 10
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_{20} = 0.017$
Iteración 20
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y

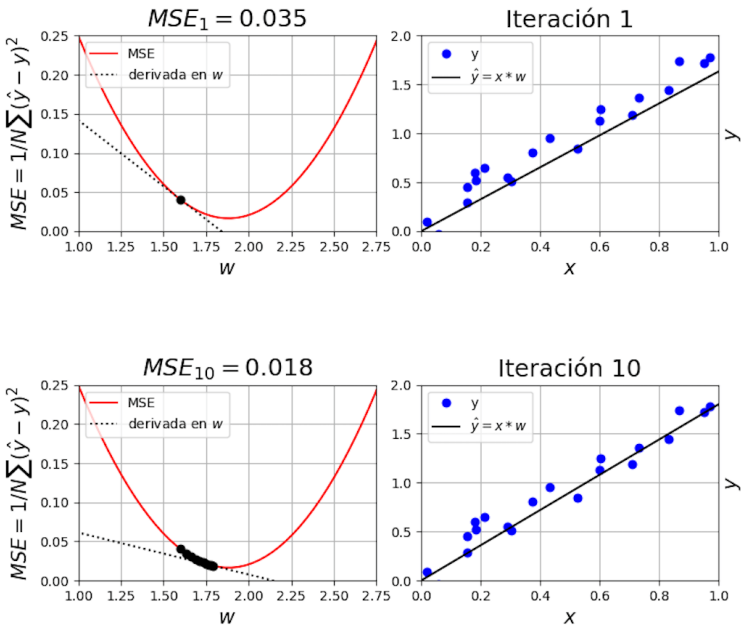

$MSE_{30} = 0.017$
Iteración 30
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y

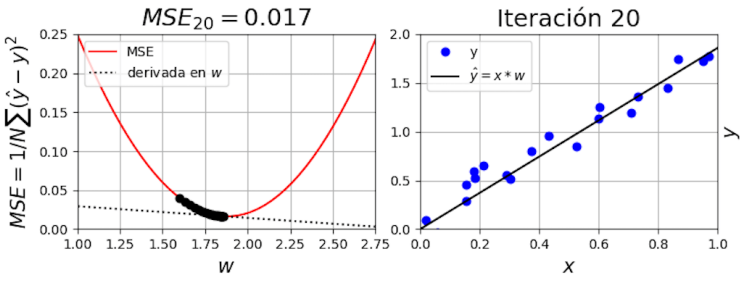
Como puedes ver en cada paso de iteración el valor de w_1 se va actualizando en la dirección de pendiente negativa de la función de pérdida, como si de una pelota bajando por una colina se tratara. Este es el motivo por el que llamamos a este algoritmo `descenso por gradiente`. En este caso, el valor de w_1 converge lentamente hacia el valor óptimo, dando al principio pasos grandes y luego más pequeños (debido a que el valor del gradiente es cada vez más pequeño y el ratio de aprendizaje es constante).

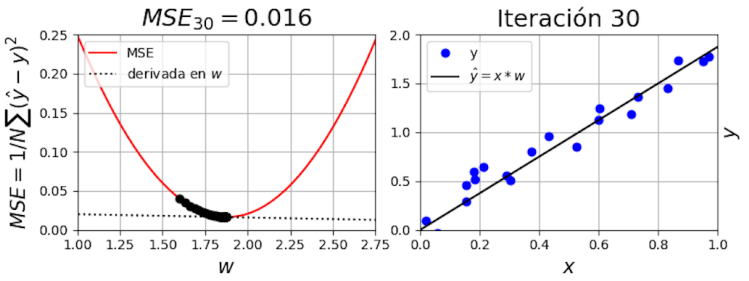
En el siguiente ejemplo vemos el efecto que tiene utilizar un valor de inicialización más cercano al valor óptimo. Como podemos ver, en este caso llegamos al valor óptimo más rápido, en menos *epochs*. Es común inicializar estos pesos de manera aleatoria (quizás siguiendo una distribución de probabilidad determinada). Así pues, si tenemos suerte y los valores iniciales están cercanos al óptimo el proceso de optimización necesitará menos iteraciones y por lo tanto será más rápido (o para el mismo número de iteraciones tendremos un mejor resultado).

Python

```
weights = solve(epochs = 30, w = 1.6, lr = 0.2)
```



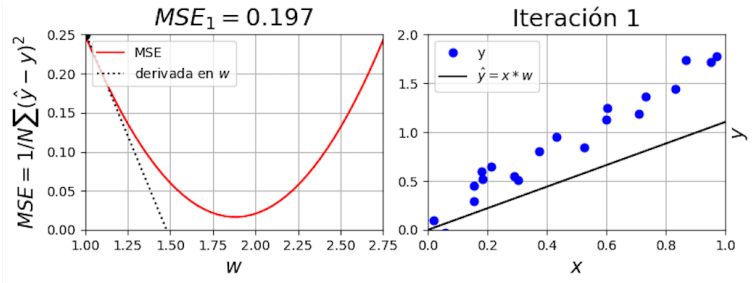


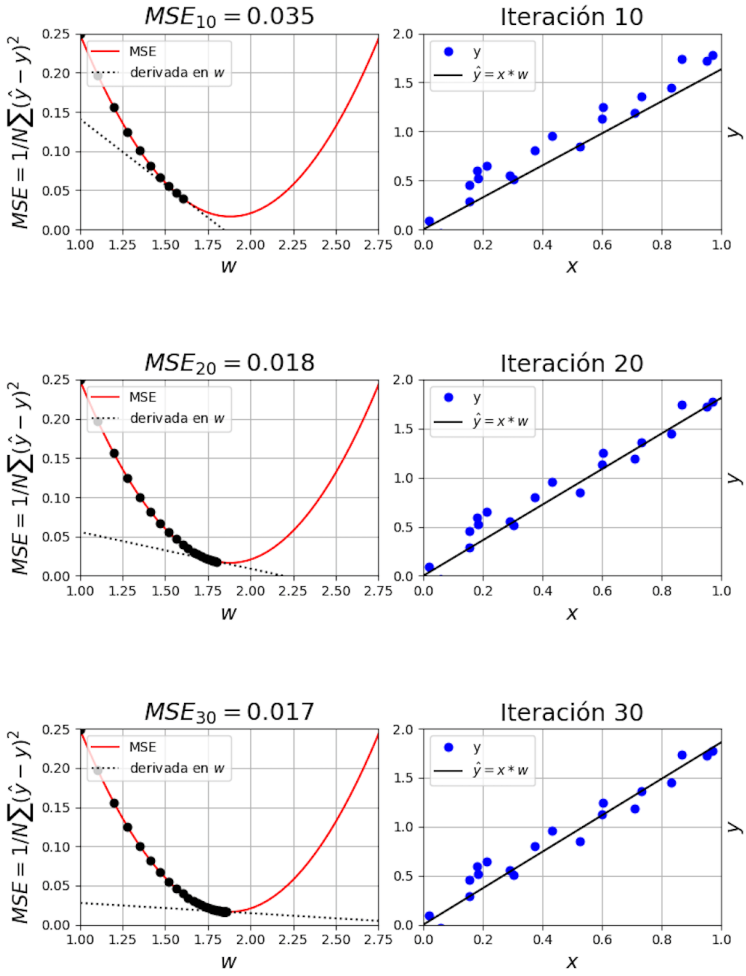


Por otro lado, si no disponemos de una buena inicialización y el valor inicial está más alejado del óptimo, necesitamos más iteraciones para obtener un buen resultado.

Python

```
weights = solve(epochs = 30, w = 1, lr = 0.2)
```





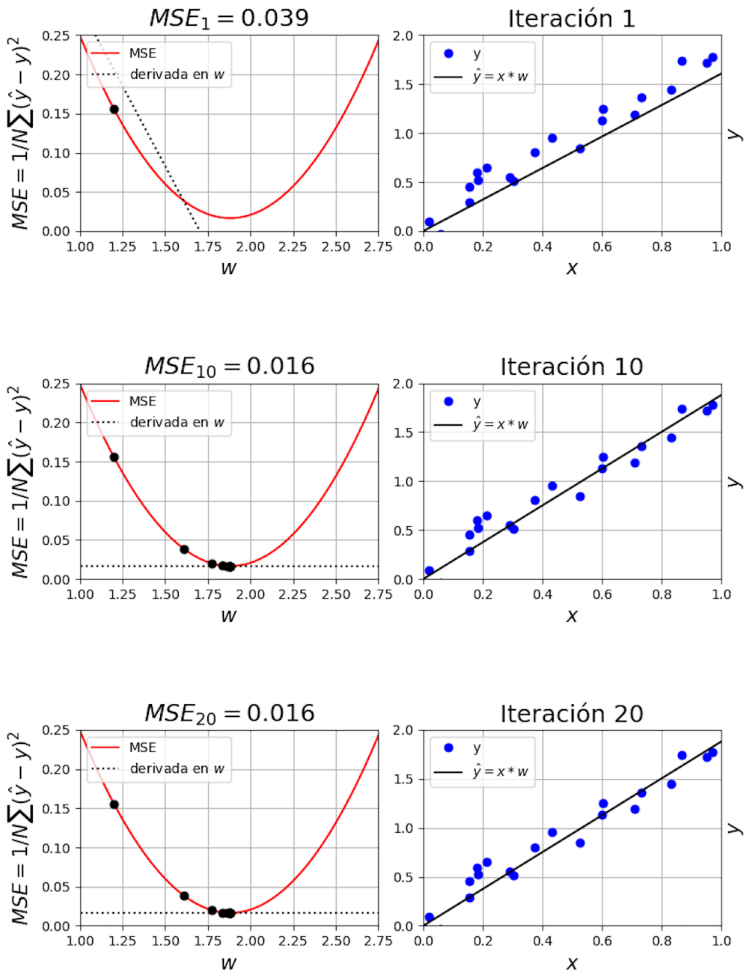
Como hemos podido ver, la inicialización de nuestro modelo tendrá un efecto en el proceso de optimización, principalmente en el número de iteraciones que necesitaremos para llegar al valor óptimo. Sin embargo, debido a que el algoritmo de descenso por gradiente no nos garantiza encontrar el valor óptimo global y puede quedarse «atascado» en un valor óptimo local, es común realizar múltiples optimizaciones con diferentes valores iniciales.

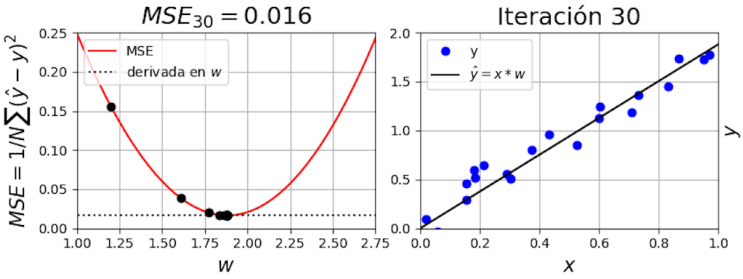
Efectos del *learning rate*

En esta sección veremos cómo afecta el valor escogido del *learning rate* al proceso de optimización. Para ello, usaremos el mismo valor inicial en todos los casos. En el primer caso, utilizamos un valor de $\eta = 1$, mucho más grande del que hemos usado en los ejemplos anteriores.

Python

```
weights = solve(epochs = 30, w = 1.2, lr = 1)
```



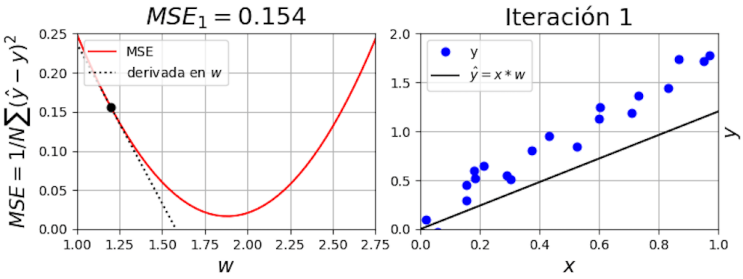


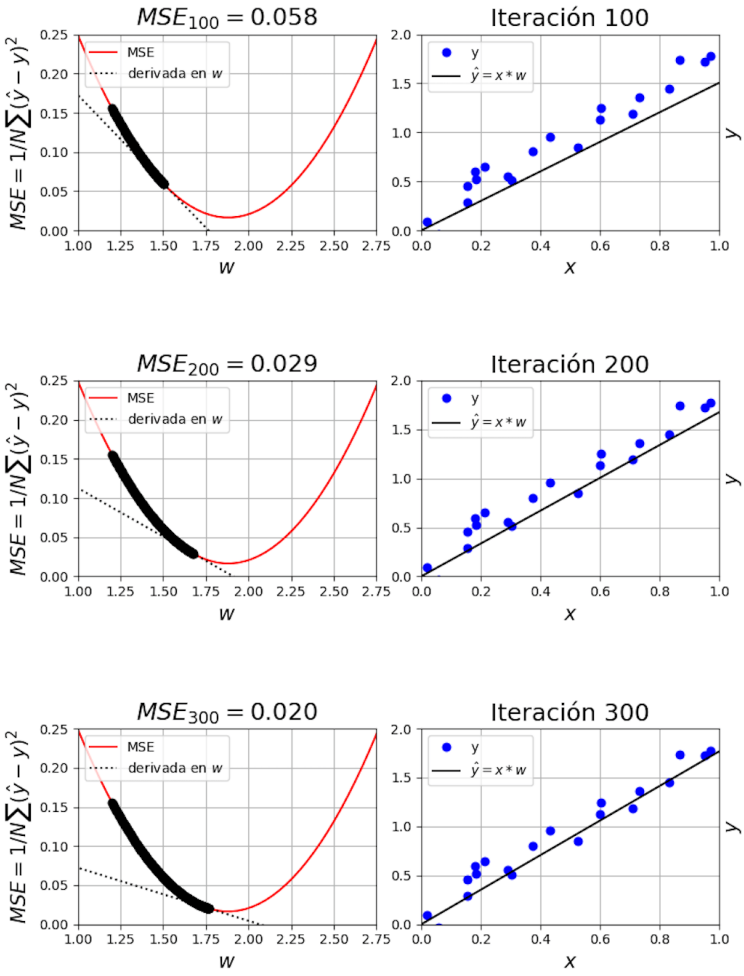
Como puedes observar, hemos llegamos a la solución óptima mucho más rápido, en apenas 10 iteraciones, mientras que en el caso anterior necesitábamos hasta 30. Recuerda que los pesos los actualizamos siguiendo la ecuación $w \leftarrow w - \eta \frac{\partial MSE}{\partial w}$, por lo que el *learning rate* determinará cómo de grande será el paso que daremos en cada iteración en dirección al valor óptimo.

Por el contrario, usar un valor muy pequeño nos sigue llevando a la solución óptima, pero requiriendo de muchas iteraciones.

Python

```
weights = solve(epochs = 300, w = 1.2, lr = 0.01)
```

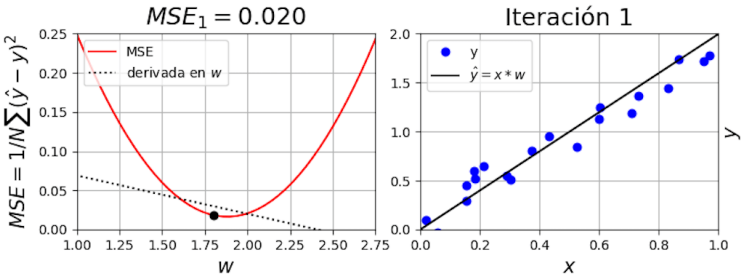


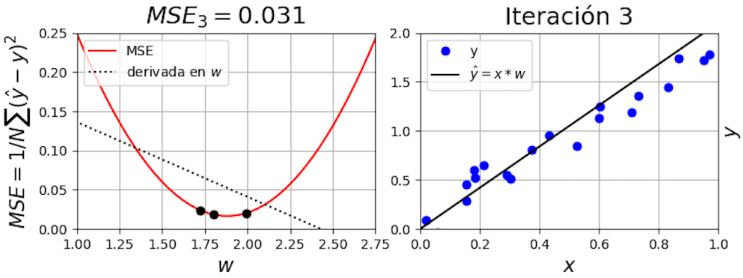


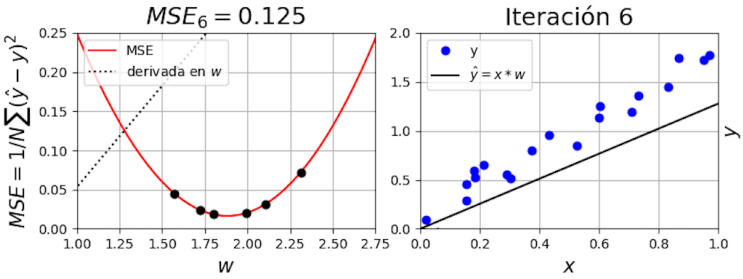
Sin embargo, si utilizamos un valor demasiado grande el proceso de optimización puede no converger y el valor de los pesos puede oscilar alrededor del valor óptimo sin llegar a él o incluso diverger (incluso si la inicialización de los pesos es cercana al valor óptimo).

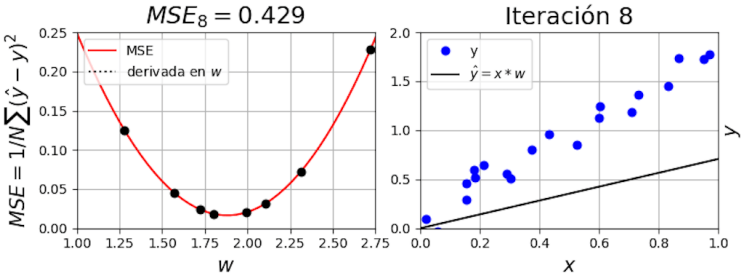
Python

```
weights = solve(epochs = 8, w = 1.8, lr = 4)
```


$MSE_1 = 0.020$
Iteración 1
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_3 = 0.031$
Iteración 3
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_6 = 0.125$
Iteración 6
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_8 = 0.429$
Iteración 8
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y

Hemos visto el efecto que tiene el *learning rate* en el entrenamiento de nuesto Perceptrón. Un valor muy pequeño resulta en un proceso de optimización lento, mientras que un valor muy grande hará que el proceso sea divergente. Así pues, tendremos que utilizar valores adecuados para tener una optimización rápida y convergente. Lamentablemente, un valor «muy pequeño», «muy grande» o «adecuado» va a depender de la función de pérdida y modelo utilizados, por lo que cada caso requerirá de varias pruebas para encontrar el mejor valor de *learning rate*.

Variantes del descenso por gradiente

En todos los casos anteriores hemos utilizado todos los elementos del *dataset* en el cálculo del gradiente. Esto se conoce como `Batch Gradient Descent` y su principal limitación es aquellos casos en los que el conjunto de datos es tan grande que no cabe en la memoria del ordenador. En estos casos tenemos dos alternativas. En el caso diametralmente opuesto, `Stochastic Gradient Descent` calcula la derivada por cada elemento del *dataset* de manera independiente. Aun así, la opción más utilizada es `Mini-Batch Gradient Descent` , método en el que usaremos un pequeño conjunto de los datos para calcular la derivada de la función de pérdida en cada iteración.

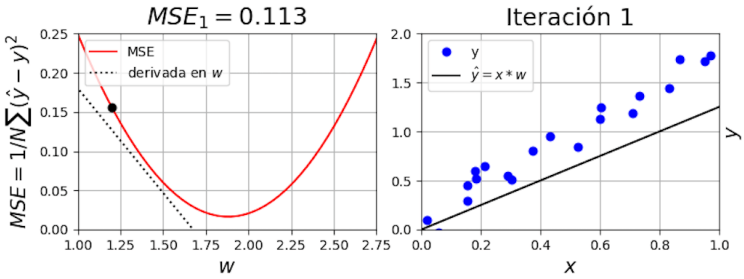
Stochastic Gradient Descent

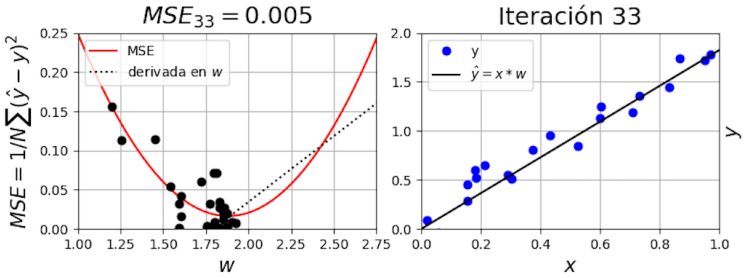
Python

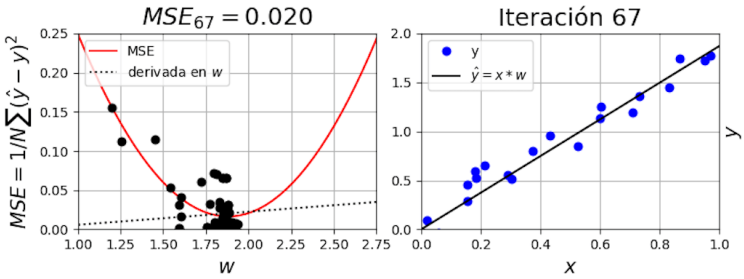
```python
def solve_sgd(epochs = 5, w = 1.2, lr = 0.2):
    # stochastic gradient descent
    weights = [(w, gradient(w, x, y), cost(x*w, y))]
    for i in range(1,epochs+1):
        # un update por cada elemento del dataset
        for _x, _y in zip(x, y):
            dw = gradient(w, _x, _y)
            w = w - lr*dw
            weights.append((w, dw, cost(_x*w, _y)))
    return weights
```

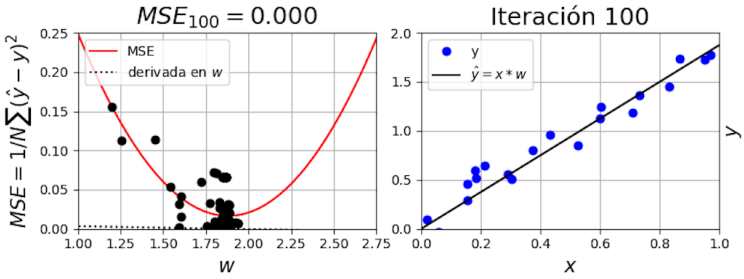
Python

```python
weights = solve_sgd(epochs = 5, w = 1.2, lr = 0.2)
```


$MSE_1 = 0.113$
Iteración 1
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_{33} = 0.005$
Iteración 33
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_{67} = 0.020$
Iteración 67
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y


$MSE_{100} = 0.000$
Iteración 100
MSE
derivada en w
$MSE = 1/N\sum(\hat{y} - y)^2$
w
y
$\hat{y} = x * w$
x
y

Este algoritmo es capaz de converger, pero en esta variante el proceso de optimización es errático, ya que la estimación del gradiente varía mucho de un punto a otro.

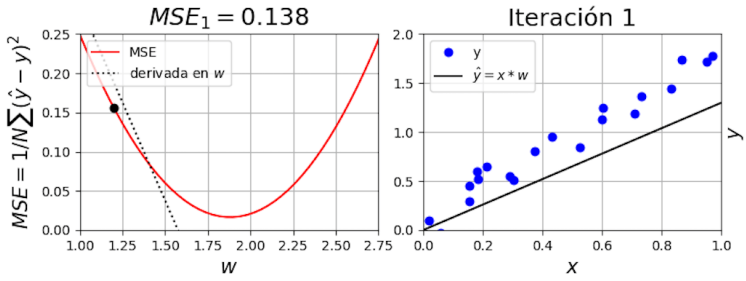
Mini-Batch Gradient Descent

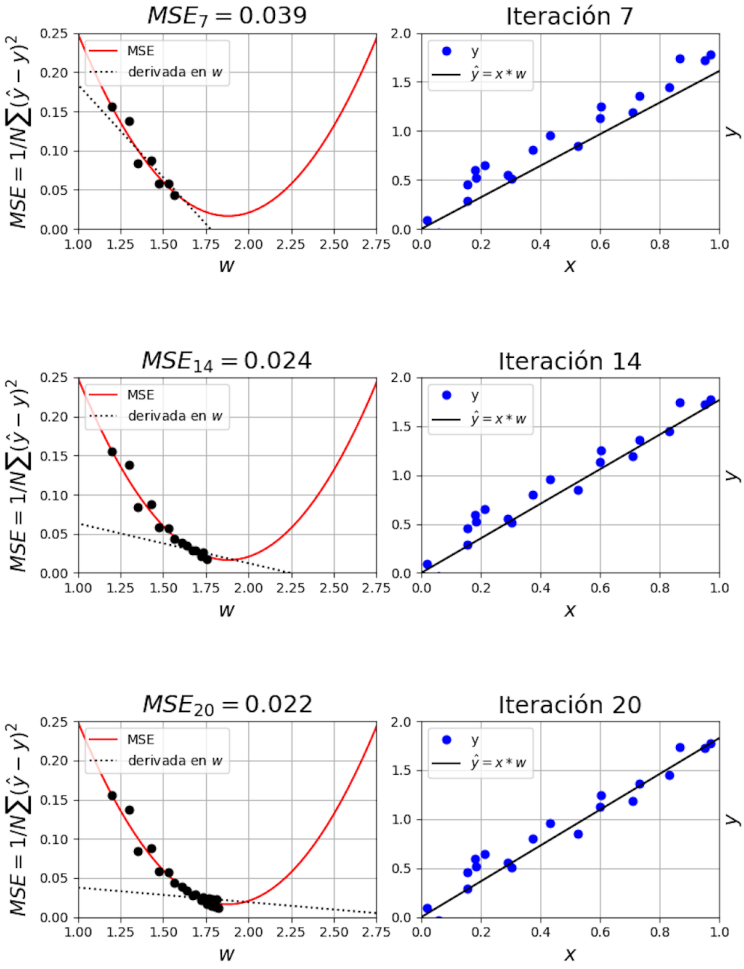
Python

```
def solve_mbgd(epochs = 20, w = 1.2, lr = 0.2, batch_size=10):
    # mini-batch gradient descent
    batches = len(x) // batch_size
    weights = [(w, gradient(w, x, y), cost(x*w, y))]
    for i in range(1,epochs+1):
        # un update por cada `batch`
        for j in range(batches):
            _x = x[j*batch_size:(j+1)*batch_size]
            _y = y[j*batch_size:(j+1)*batch_size]
            dw = gradient(w, _x, _y)
            w = w - lr*dw
            weights.append((w, dw, cost(x*w, y)))
    return weights
```

Python

```
weights = solve_mbgd(epochs = 10, w = 1.2, lr = 0.2)
```





En este caso observamos un comportamiento intermedio entre utilizar todos los datos para hacer un paso de optimización o un solo elemento del *dataset*. El proceso sigue convergiendo, no de manera tan suave como en la variante `Batch Gradient Descent`, sigue siendo un poco errático, pero no tanto como la variante `Stochastic Gradient Descent`. Este hecho hace que sea la opción por defecto en la mayoría de procesos de optimización.

> Una implementación en modo *mini batch* siempre nos permitirá utilizar la variante *batch* o *stochastic* simplemente usando un *batch size* igual al número de elementos del *dataset* para el primer caso o igual a 1 para el segundo.

Regresión y Clasificación

El `Perceptrón` es un algoritmo de IA muy simple, pero es capaz de llevar a cabo tareas no solo de regresión sino también de clasificación. Para ello, utilizaremos diferentes funciones de activación. En el caso de la regresión lineal, ejemplo visto hasta ahora, utilizamos la función de activación lineal, $f(x) = x$ (que es como si no hubiese función de activación), pero en el caso de la clasificación utilizaremos diferentes funciones de activación dependiendo del problema que queramos resolver.

Regresión Logística

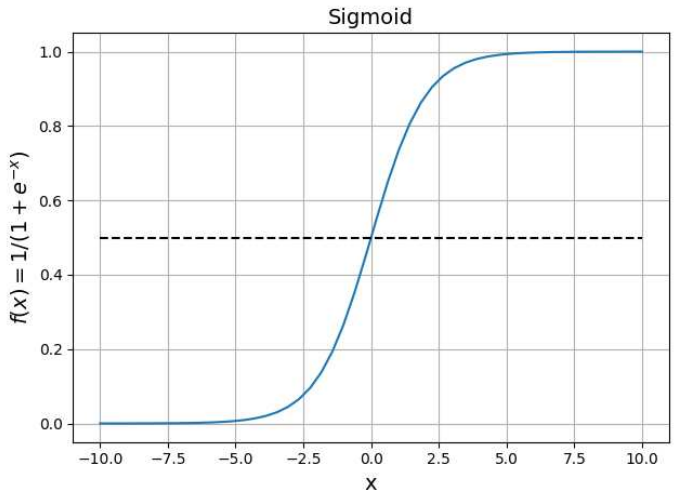
Si bien el nombre puede llevar a confusión, la regresión logística es un algoritmo de clasificación. En este caso, la función de activación que utilizamos es la función sigmoide, $f(x) = \frac{1}{1+e^{-x}}$. Esta función nos da un valor entre 0 y 1, que podemos interpretar como la probabilidad de que un elemento pertenezca a una clase. En el caso de la regresión logística, el `Perceptrón` nos dará como salida la probabilidad de que un elemento pertenezca a una clase, y si esta probabilidad es mayor que un umbral determinado, diremos que el elemento pertenece a esa clase.

Python

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

Python

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10)
y = 1. / (1. + np.exp(-x))

plt.plot(x, y)
plt.grid(True)
plt.xlabel('x', fontsize=14)
plt.ylabel('$f(x) = 1/(1 + e^{-x})$', fontsize=14)
plt.title('Sigmoid', fontsize=14)
plt.plot(x, np.full(len(x), 0.5), '--k')
plt.tight_layout()
plt.savefig("resources/reg_log1.png", bbox_inches='tight', pad_inches
    =0.1)
plt.close()
```



Como podemos ver, los valores muy negativos saturan a un valor de 0 mientras que los valores muy positivos saturan a un valor de 1. Cerca de 0 la función sigmoid nos dará una transición suave entre los dos valores límite.

Podríamos intentar entrenar nuestro modelo de `regresión logísitca` con la función de pérdida que ya conocemos, *MSE*, y de hecho funcionaría sin problemas ya que estamos forzando la salida a tomar el valor 0 o 1 en función del *ground truth*. Sin embargo, vamos a introducir una nueva función de pérdida muy utilizada cuando trabajamos con modelos

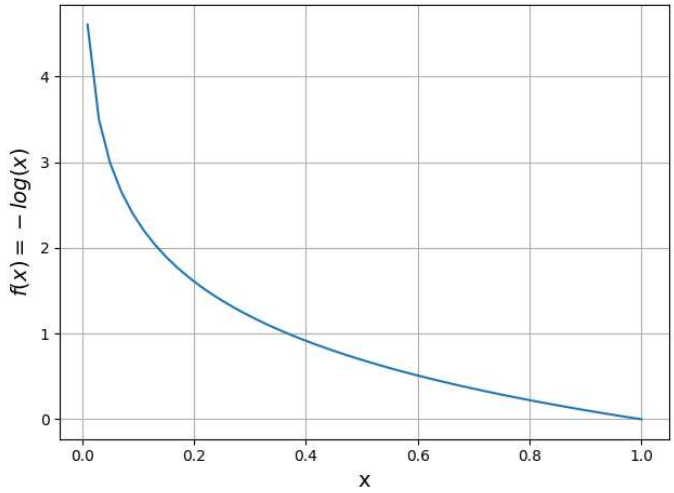
probabilísticos, ya que dan como resultado un mejor proceso de optimización. Esta función es conocida como *Binary Cross Entropy*, o también la puedes encontrar por el nombre de *log loss*.

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{j=1}^{N}\left[y^{(j)}log(\hat{y}^{(j)}) + (1-y^{(j)})log(1-\hat{y}^{(j)})\right]$$

Como puedes ver, cuando el *ground truth* sea 0 solo intervendrá el segundo término, $log(1-\hat{y})$. En este caso, si la salida del modelo es correcta y nos da un valor cercano a 0 obtendremos un valor de la función de pérdida pequeño. Sin embargo, si el modelo se equivoca y da una probabilidad alta, el valor de la función de pérdida será muy grande indicando un alto error (fíjate en el signo negativo al principio de la expresión, necesario porque el límite cuando x tiende a 0 del logaritmo es $-\infty$, pero necesitamos valores positivos para indicar error). Lo mismo se aplica al caso contrario en el que el *ground truth* sea 1, pero con el primer término.

Python

```
x = np.linspace(0.01, 1)
y =  -np.log(x)

plt.plot(x, y)
plt.grid(True)
plt.xlabel('x', fontsize=14)
plt.ylabel('$f(x) = - log(x)$', fontsize=14)
plt.tight_layout()
plt.savefig("resources/reg_log2.png", bbox_inches='tight', pad_inches
    =0.1)
plt.close()
```



El motivo por el que usamos logaritmos en la función de pérdida está relacionado con el concepto de `entropía`, una medida del grado de confianza asociado a una distribución de probabilidad.

Como bien sabrás ya, en este punto la función de pérdida no solo sirve para calcular el error de nuestro modelo, sino también para su entrenamiento. Para ello, necesitamos calcular la derivada de la función de pérdida con respecto a los pesos de nuestro modelo. En este caso, es un poco más complicado que para la *MSE*, pero con un poco de matemáticas llegamos a la siguiente expresión:

$$\frac{\partial J}{\partial w_i} = \frac{1}{N}\sum_{j=1}^{N}(\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}) - y^{(i)})x_i^{(j)}$$

¿Te resulta familiar? Es exactamente la misma expresión que obtenemos al calcular la derivada de la *MSE* en el caso del `Perceptrón` con función de activación lineal que ya utilizamos para tareas de `regresión`. Vamos a mejorar la implementación del `Perceptrón` de la sección anterior para que ahora también sea capaz de funcionar como un modelo de `regresión logística`.

Python

```python
# funciones de pérdida

def mse(y, y_hat):
        return np.mean((y_hat - y)**2)

def bce(y, y_hat):
        return - np.mean(y*np.log(y_hat) - (1 - y)*np.log(1 - y_hat))


# funciones de activación

def linear(x):
        return x

def step(x):
        return x > 0

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


# Perceptrón

class Perceptron():
    def __init__(self, size, activation, loss):
        self.w = np.random.randn(size)
        self.ws = []
        self.activation = activation
        self.loss = loss

    def __call__(self, w, x):
        return self.activation(np.dot(x, w))

    def fit(self, x, y, epochs, lr):
        x = np.c_[np.ones(len(x)), x]
        for epoch in range(epochs):
                # Batch Gradient Descent
                y_hat = self(self.w, x)
                # función de pérdida
                l = self.loss(y, y_hat)
                # derivadas
                dldh = (y_hat - y)
                dhdw = x
                dldw = np.dot(dldh, dhdw)
                # actualizar pesos
                self.w = self.w - lr*dldw
                # guardar pesos para animación
                self.ws.append(self.w.copy())
```

Como puedes ver hemos definido diferentes funciones para cada una de las funciones de pérdida y de activación que conocemos, y simplemente le diremos al `Perceptrón` cuáles utilizar al inicializarlo. De esta manera, nuestro modelo será capaz de resolver tanto tareas de `regresión` como de `clasificación binaria`.

Ahora que ya tenemos todas las piezas en su lugar, vamos a utilizar nuestro `Perceptró n` para la tarea de clasificación binaria en el dataset Iris, que contiene 150 muestras de tres especies de Iris (Iris setosa, Iris virginica e Iris versicolor). Cada muestra tiene cuatro características (longitud y ancho del sépalo y del pétalo) y una etiqueta que indica a qué especie pertenece. Para poder visualizar el proceso de optimización vamos a utilizar solo dos de las características. Además, al tratarse de un problema de clasificación binaria, vamos a intentar distinguir una de las clases de las otras dos.

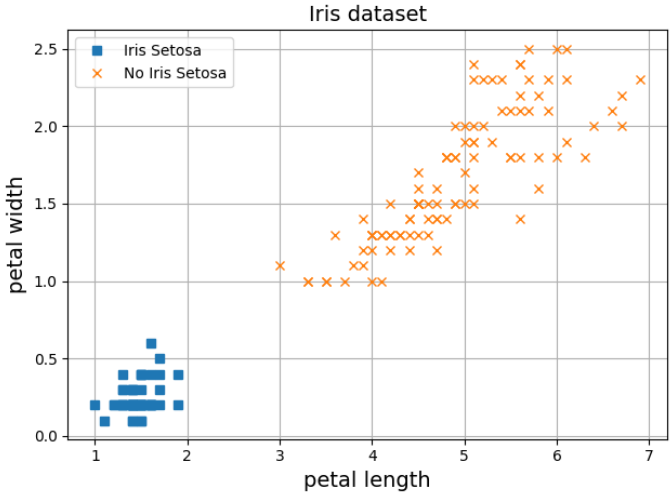
Python

```python
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(int) # clsificación binaria

X.shape, y.shape
```

Devuelve: *((150, 2), (150,))*

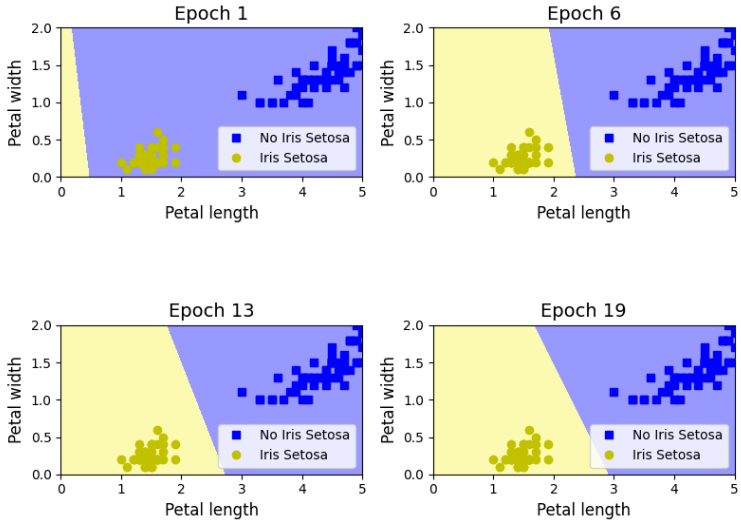
Python

```python
plt.plot(X[y==1, 0], X[y==1, 1], 's', label="Iris Setosa")
plt.plot(X[y==0, 0], X[y==0, 1], 'x', label="No Iris Setosa")
plt.grid()
plt.legend()
plt.xlabel('petal length', fontsize=14)
plt.ylabel('petal width', fontsize=14)
plt.title("Iris dataset", fontsize=14)
plt.tight_layout()
plt.savefig("resources/reg_log3.png", bbox_inches='tight', pad_inches
    =0.1)
plt.close()
```



Python

```
np.random.seed(42)

perceptron = Perceptron(3, sigmoid, bce)
epochs, lr = 20, 0.01
perceptron.fit(X, y, epochs, lr)
```



En cada imagen podemos ver la frontera de decisión que nuestro modelo usará para asignar una clase u otra. Como podemos ver, nuestro modelo es capaz de converger a una solución óptima que separa completamente las dos clases. Ahora podemos usar nuestro modelo entrenado para asignar probabilidades de que una flor sea del tipo Iris Setosa a partir de la longitud y el ancho de sus pétalos.

Python

```
w = perceptron.ws[-1]
w
```

Devuelve: *array([3.1121193 , -1.14317047, -0.70370935])*

Python

```
x_new = [1, 2, 0.5]
y_pred = perceptron(w, x_new)
y_pred # Iris Setosa
```

Devuelve: *0.6163120194730394*

Python

```
x_new = [1, 1, 0.5]
y_pred = perceptron(w, x_new)
y_pred # Iris Setosa
```

Devuelve: *0.8343939887777257*

Python

```
x_new = [1, 3, 0.5]
y_pred = perceptron(w, x_new)
y_pred # No Iris Setosa
```

Devuelve: *0.33866551968712205*

Python

```
x_new = [1, 4, 0.5]
y_pred = perceptron(w, x_new)
y_pred # No Iris Setosa
```

Devuelve: *0.14034623285805609*

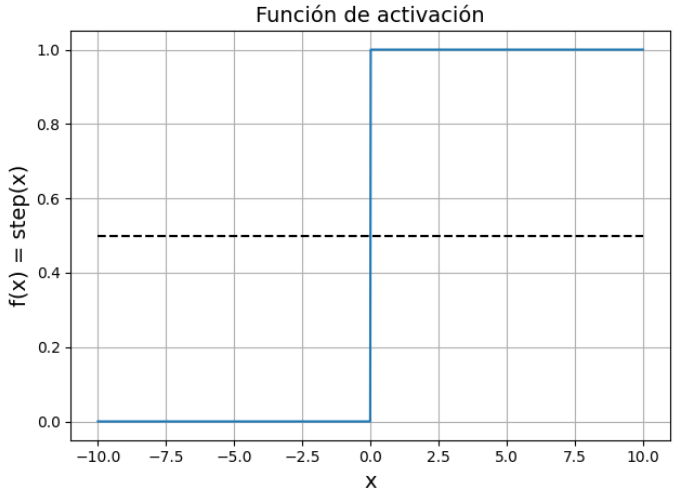
Como puedes observar, el modelo está más seguro cuanto más nos alejamos de la frontera de decisión (valores más cercanos a 0 o 1). Sin embargo, cuanto más cerca de la frontera de decisión nos encontramos, el modelo está menos seguro dando resultados más cercanos a $0{,}5$. ¿Cómo asignamos una clase u otra entonces? Lo más común es decidir un valor umbral, `threshold`, a partir del cual asignaremos una clase u otra. El valor más común es el de $0{,}5$ (cualquier valor por encima será asignado a la clase en cuestión). El valor de este `threshold` puede modificarse en cualquier caso para adaptarnos a los criterios de diseño buscados, el ratio entre `falsos positivos` y `falsos negativos`. Esto es muy importante sobre todo en aplicaciones sensibles como sistemas de ayuda al diagnóstico médico, por ejemplo, en el que es más importante no tener `falsos negativos` que `falsos positivos` (y, por lo tanto, se escogerá un `threshold` acorde, aunque ello dé como resultado un modelo peor según otras métricas).

Clasificación Binaria

Otra manera en la que podemos llevar a cabo la tarea de clasificación binaria es usando una función de activación diferente, la función de activación de paso, o `step` en inglés, $f(x) = 1$ si $x > 0$ y 0 en caso contrario. En este caso, el `Perceptrón` nos dará como salida un valor de 0 o 1, que podemos interpretar como la clase a la que pertenece un elemento.

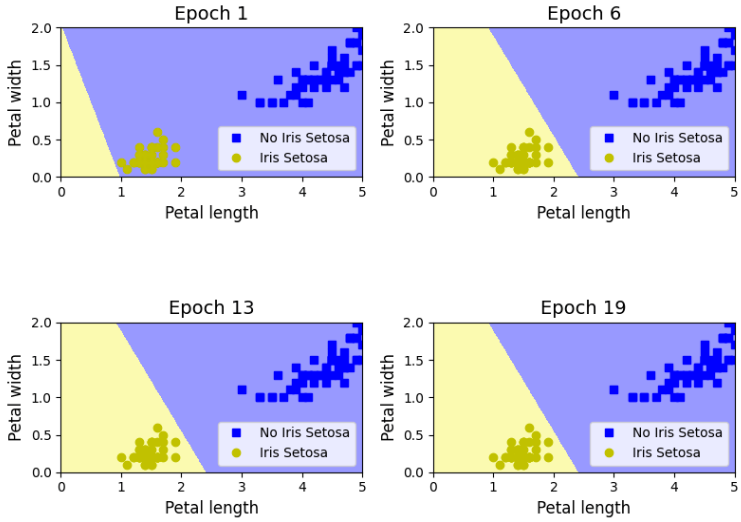
Python

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

x = np.linspace(-10, 10, 1000)
y = x > 0

plt.plot(x, y)
plt.grid(True)
plt.xlabel('x', fontsize=14)
plt.ylabel('f(x) = step(x)', fontsize=14)
plt.title('Función de activación', fontsize=14)
plt.plot(x, np.full(len(x), 0.5), '--k')
plt.tight_layout()
plt.savefig("resources/bin_clas1.png", bbox_inches='tight', pad_inches
    =0.1)
plt.close()
```



Python

```
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(int) # clsificación binaria

perceptron = Perceptron(3, step, mse)
epochs, lr = 20, 0.01
perceptron.fit(X, y, epochs, lr)
```



De nuevo, nuestro modelo ha sido capaz de encontrar una buena frontera de decisión. La principal diferencia es que no tenemos una probabilidad de que un elemento pertenezca a una clase u otra, sino que simplemente nos da una clase u otra.

Python

```
# últimos pesos encontrados

w = perceptron.ws[-1]
w
```

Devuelve: *array([2.03302986, -0.84815337, -0.63213696])*

Python

```
x_new = [1, 2, 0.5]
y_pred = perceptron(w, x_new)
y_pred # Iris Setosa
```

Devuelve: *True*

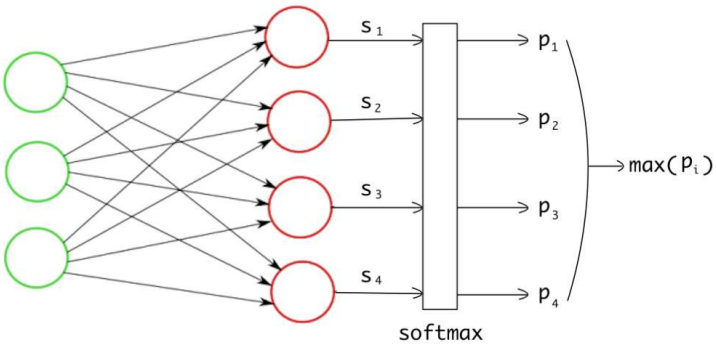
Python

```
x_new = [1, 4, 0.5]
y_pred = perceptron(w, x_new)
y_pred # No Iris Setosa
```

Devuelve: *False*

Clasificación Multiclase

Los ejemplos que hemos visto hasta ahora nos han permitido llevar a cabo dos tipos de tareas: regresión lineal y clasificación binaria. En esta sección vamos a mejorar la implementación de nuestro modelo para ser capaces de utilizarlo en la tarea de clasificación en varias clases. En primer lugar, vamos a revisar la arquitectura del `Perceptrón`. La idea es la siguiente: si un `Perceptrón` es capaz de clasificación binaria (identificar una clase en particular de entre el resto), ¿podemos utilizar tantos perceptrones como clases tengamos? En esta configuración, cada `Perceptrón` tendrá el objetivo de identificar una de las clases en particular del resto.



En esta nueva implementación, tendremos tantas salidas como clases (podemos usarlo también para regresión, cada salida correspondería a cada una de las magnitudes de interés). Calculamos la salida del modelo de la misma manera que en el caso del `Perceptrón` de una sola salida. La diferencia es que en la última capa aplicaremos una función de activación de tipo `softmax`. Esto nos dará como resultado una distribución de probabilidad sobre todas las clases. Finalmente, asignaremos aquella clase que tenga la probabilidad más alta.

$$\hat{y} = \arg\max_k \ \sigma(\mathbf{w} \cdot \mathbf{x})_k$$

En el caso del `Perceptrón` con una sola salida, nuestros pesos estaban representados por un vector de longitud igual al número de entradas (más uno si tenemos en cuenta el *bias*). Ahora, al tener varias salidas, los pesos estarán representados por una matriz (puedes ver el vector anterior como una matriz de una sola columna). Este modelo también se conoce por el nombre de `Softmax Regression`.

Python

```python
def softmax(x):
    return np.exp(x) / np.exp(x).sum(axis=-1,keepdims=True)
```

Para entrenar el modelo con el algoritmo de `descenso por gradiente` necesitamos una función de pérdida, una medida del error que nos sirva para optimizar los pesos. En este caso, utilizamos la función de pérdida conocida como `Cross Entropy`.

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{j=1}^{N}\sum_{k=1}^{K} y_k^{(j)} log(\hat{y}_k^{(j)})$$

donde $y_k^{(j)}$ es la probabilidad de que el elemento j pertenezca a la clase k (normalmente 1 ó 0). Como puedes ver, en el caso de que tengamos solo dos clases esta expresión es equivalente a la función de pérdida que utilizamos para clasificación binaria en el modelo de regresión logística, la función `Binary Cross Entropy`.

Como siempre, para poder aplicar el algoritmo, no solo necesitamos la función de pérdida, sino también su derivada con respecto a los pesos del modelo que queremos optimizar. Calculando la derivada de la función anterior obtenemos la siguiente expresión.

$$\frac{\partial J_k}{\partial \mathbf{w}} = \frac{1}{N}\sum_{j=1}^{N}(\hat{y}_k^{(j)} - y_k^{(j)})\mathbf{x}^{(j)}$$

Por motivos de estabilidad numérica, es común combinar la función `softmax` con la función `cross entropy` de la siguiente manera.

Python

```
# aplica softmax + cross entropy de manera estable

def crossentropy(y, y_hat):
    logits = y_hat[np.arange(len(y_hat)),y]
    entropy = - logits + np.log(np.sum(np.exp(y_hat),axis=-1))
    return entropy.mean()


# solo si usamos softmax
def grad_crossentropy(y, y_hat):
    answers = np.zeros_like(y_hat)
    answers[np.arange(len(y_hat)),y] = 1
    return (- answers + softmax(y_hat)) / y_hat.shape[0]
```

Vamos a mejorar ahora nuestra implementación del `Perceptrón` para ser capaz de llevar a cabo la tarea de clasificación multiclase (además de todas las que ya es capaz de hacer).

Python

```python
class Perceptron():
    def __init__(self, inputs, outputs, activation, loss, grad_loss):
        inputs = inputs + 1
        self.w = np.random.normal(loc=0.0,
                    scale = np.sqrt(2/(inputs+outputs)),
                    size = (inputs, outputs))
        self.ws = []
        self.activation = activation
        self.loss = loss
        self.grad_loss = grad_loss

    def __call__(self, w, x):
        return self.activation(np.dot(x, w))

    def fit(self, x, y, epochs, lr, batch_size=None, verbose=True,
        log_each=1):
        if batch_size == None:
                batch_size = len(x)
        x = np.c_[np.ones(len(x)), x]
        batches = len(x) // batch_size
        for epoch in range(1,epochs+1):
                # Mini-Batch Gradient Descent
                for b in range(batches):
                        _x = x[b*batch_size:(b+1)*batch_size]
                        _y = y[b*batch_size:(b+1)*batch_size]
                        y_hat = self(self.w, _x)
                        #print(y_hat.shape)
                        # función de pérdida
                        l = self.loss(_y, y_hat)
                        # derivadas
                        dldh = self.grad_loss(_y, y_hat)
                        dhdw = _x
                        dldw = np.dot(dhdw.T, dldh)
                        # actualizar pesos
                        self.w = self.w - lr*dldw
                # guardar pesos para animación
                self.ws.append(self.w.copy())
                # print loss
                if verbose and not epoch % log_each:
                        print(f"Epoch {epoch}/{epochs} Loss {l}")

    def predict(self, x):
        x = np.c_[np.ones(len(x)), x]
        return self(self.w, x)
```

Python

```python
# Mean Square Error -> usada para regresión (con activación lineal)
def mse(y, y_hat):
        return np.mean((y_hat - y.reshape(y_hat.shape))**2)


# Binary Cross Entropy -> usada para clasificación binaria (con sigmoid)
def bce(y, y_hat):
        return - np.mean(y.reshape(y_hat.shape)*np.log(y_hat) - (1 - y.
            reshape(y_hat.shape))*np.log(1 - y_hat))


# solo si usamos activación lineal
def grad_mse(y, y_hat):
        return y_hat - y.reshape(y_hat.shape)


# solo si utilizamos sigmoid
def grad_bce(y, y_hat):
        return y_hat - y.reshape(y_hat.shape)

class LinearRegression(Perceptron):
    def __init__(self, inputs, outputs=1):
        super().__init__(inputs, outputs, linear, mse, grad_mse)

class LinearClassification(Perceptron):
    def __init__(self, inputs, outputs=1):
        super().__init__(inputs, outputs, step, mse, grad_mse)

class LogisticRegression(Perceptron):
    def __init__(self, inputs, outputs=1):
        super().__init__(inputs, outputs, sigmoid, bce, grad_bce)

class SoftmaxRegression(Perceptron):
    def __init__(self, inputs, outputs):
        # usamos activación lineal porque `crossentropy` ya incluye la
            softmax
        super().__init__(inputs, outputs, linear, crossentropy,
            grad_crossentropy)
```

Ahora vamos a probar nuestro nuevo modelo para clasificación multiclase con el dataset Iris, ahora sí, usando las 3 clases.

Python

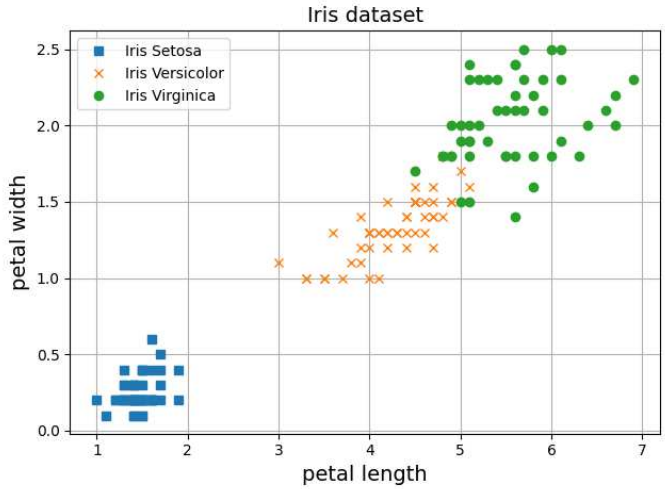
```
iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
X_mean, X_std = X.mean(axis=0), X.std(axis=0)
X_norm = (X - X_mean) / X_std # normalización

y = iris.target

X.shape, y.shape
```

Devuelve: *((150, 2), (150,))*

Adicionalmente, hemos normalizado los datos para que tengan media 0 y desviación estándar 1. Esto es importante para que el proceso de optimización sea más rápido y estable, como veremos en un par de capítulos.

Python

```
plt.plot(X[y==0, 0], X[y==0, 1], 's', label="Iris Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], 'x', label="Iris Versicolor")
plt.plot(X[y==2, 0], X[y==2, 1], 'o', label="Iris Virginica")
plt.grid()
plt.legend()
plt.xlabel('petal length', fontsize=14)
plt.ylabel('petal width', fontsize=14)
plt.title("Iris dataset", fontsize=14)
plt.tight_layout()
plt.savefig("resources/clas_multi1.png")
plt.close()
```



A diferencia de los ejemplos anteriores en los que intentábamos separar una de las tres clases del resto, ahora vamos a intentar ser capaces de clasificar las tres clases a la vez.

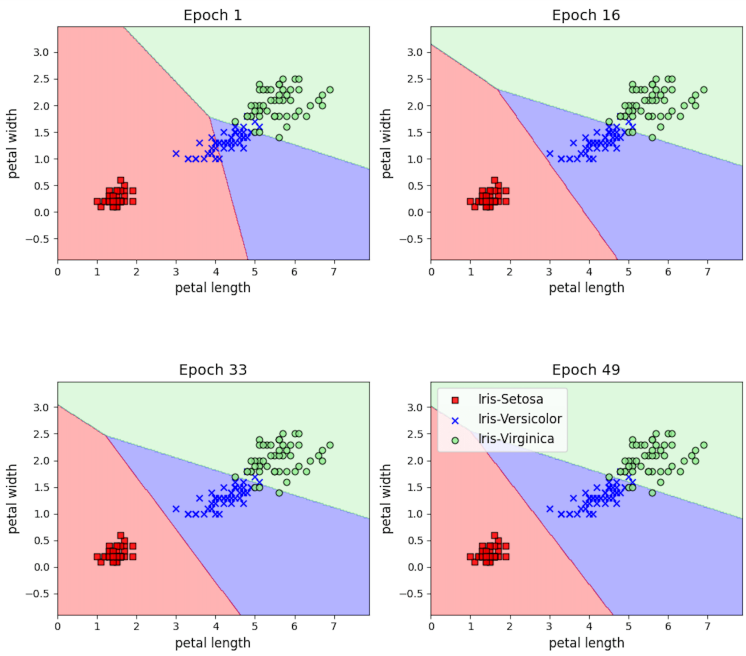
Python

```
perceptron = SoftmaxRegression(2, 3)
epochs, lr = 50, 1
perceptron.fit(X_norm, y, epochs, lr, log_each=10)
```

Devuelve:

Epoch 10/50 Loss 0.3759239255356093
Epoch 20/50 Loss 0.27886494936667544
Epoch 30/50 Loss 0.2313267940856377
Epoch 40/50 Loss 0.20239520361161742
Epoch 50/50 Loss 0.18276675650576815

/tmp/ipykernel_1141597/1661452277.py:33: UserWarning: You passed a edgecolor/edgecolors ('black') for an unfilled marker ('x'). Matplotlib is ignoring the edgecolor in favor of the facecolor. This behavior may change in the future. ax.scatter(x=X[y == cl, 0],



Como puedes ver ahora, nuestro modelo es capaz de separar todo el espacio de longitud y ancho del pétalo de una flor en tres clases. Dada una nueva flor, podemos saber su clase de la siguiente forma:

Python

```
#  nuevo punto
X_new = [[2, 0.5]]

# normalizamos
X_new_norm = (X_new - X_mean) / X_std

# salida del perceptron
y = perceptron.predict(X_new_norm)
y
```

Devuelve: *array([[4.61587824, 2.70828904, -4.21233636]])*

A la salida del perceptrón aplicamos la función `softmax` para convertirla en una distribución de probabilidad.

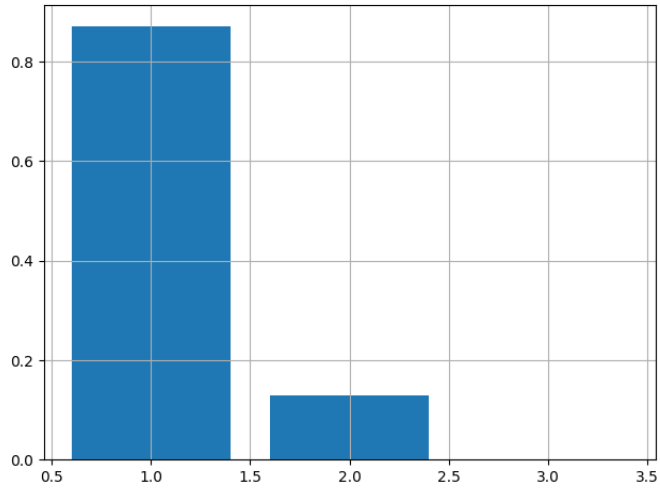
Python

```
y_probas = softmax(y)
y_probas
```

Devuelve: *array([[8.70636972e-01, 1.29235446e-01, 1.27582826e-04]])*

Python

```
plt.bar(list(range(1,4)), y_probas[0])
plt.grid(True)
plt.tight_layout()
plt.savefig("resources/clas_multi3.png")
plt.close()
```



En una distribución de probabilidad, la suma de todas las probabilidades asignadas a cada una de las clases tiene que sumar 1.

Python

```
y_probas.sum(axis=1)
```

Devuelve: *array([1.])*

Ahora asignaremos aquella clase con la mayor probabilidad.

Python

```
y_pred = np.argmax(y_probas, axis=1)
y_pred
```

Devuelve: *array([0])*

Si abstraemos la lógica en una función podemos probar varios puntos.

Python

```
def evaluate(x):
    x = (x - X_mean) / X_std
    y = perceptron.predict(x)
    y_probas = softmax(y)
    return np.argmax(y_probas, axis=1)
```

Python

```
X = [[5, 1],[2, 0.5],[7, 3]]
evaluate(X)
```

Devuelve: *array([1, 0, 2])*

Métricas de Clasificación

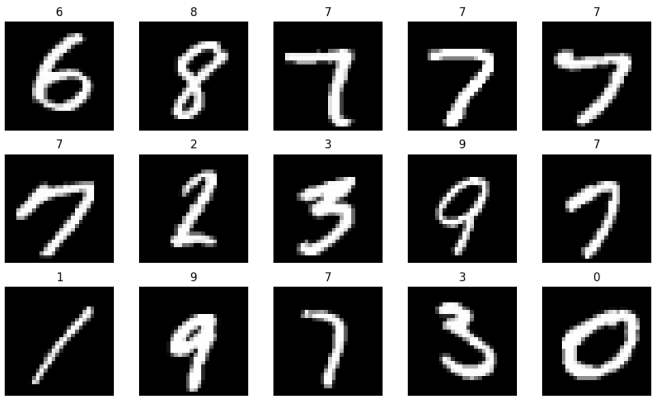
En este punto ya hemos entrenado varios Preceptrones con diferentes funciones de activación, funciones de pérdida e incluso diferentes algoritmos de optimización. Ahora, ¿cómo lo podemos evaluar? ¿Cómo saber si un modelo es un buen modelo o un mal modelo? ¿Si entrenamos otro modelo diferente, cómo podemos saber si es mejor o peor que el primero? ¿Cómo definimos qué es un buen modelo? Todas estas preguntas las podemos responder con las `métricas`. En esta sección veremos diferentes métricas para evaluar modelos entrenados con el dataset MNIST.

Python

```
from sklearn.datasets import fetch_openml
import numpy as np

mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

Python

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import random

r, c = 3, 5
fig = plt.figure(figsize=(2*c, 2*r))
for _r in range(r):
    for _c in range(c):
        plt.subplot(r, c, _r*c + _c + 1)
        ix = random.randint(0, len(X)-1)
        img = X[ix]
        plt.imshow(img.reshape(28,28), cmap='gray')
        plt.axis("off")
        plt.title(y[ix])
plt.tight_layout()
plt.savefig("resources/clas_met1.png")
plt.close()
```



Es importante tener en cuenta que para evaluar correctamente un modelo lo tenemos que hacer con datos que no hayan sido usados durante el entrenamiento.

Python

```
X_train, X_test = X[:60000] / 255, X[60000:] / 255.
y_train, y_test = y[:60000], y[60000:]
```

Empezaremos entrenando un clasificador binario que distinga las imágenes de 5 de las demás.

Python

```
y_train_5 = (y_train == 5).astype(int)
y_test_5 = (y_test == 5).astype(int)
```

Python

```
perceptron = LogisticRegression(784)
epochs, lr = 10, 1e-5
perceptron.fit(X_train, y_train_5, epochs, lr, log_each=1, batch_size
    =1000)
```

Devuelve:

Epoch 1/10 Loss 0.07726661859839262
Epoch 2/10 Loss 0.09473705303350866
Epoch 3/10 Loss 0.08710146104565634
Epoch 4/10 Loss 0.07832594460460084
Epoch 5/10 Loss 0.0713883444624499
Epoch 6/10 Loss 0.06619097612474398
Epoch 7/10 Loss 0.06228509906438874
Epoch 8/10 Loss 0.05930533400485463
Epoch 9/10 Loss 0.056996863039314914
Epoch 10/10 Loss 0.05518540189220763

Python

```
def evaluate(x, th = 0.5):
    y_pred = perceptron.predict(x) > th
    return y_pred.astype(int).ravel()
```

Precisión

La primera métrica que vamos a ver, también la más común, es la *accuracy* (o precisión). En esta métrica simplemente contamos todos los elementos del dataset que nuestro modelo ha sido capaz de clasificar correctamente y lo dividimos entre el número total de elementos. Esto implica que la precisión será un valor entre 0 y 1, significando 0 que nuestro modelo no ha acertado ningún resultado y 1 que los ha acertado todos. También es común dar este valor en porcentaje (simplemente multiplicando por 100 el resultado anterior).

Python

```python
def accuracy(y_pred, y):
    return np.sum(y_pred == y) / len(y)
```

Python

```python
accuracy(evaluate(X_train), y_train_5)
```

Devuelve: *0.92155*

Python

```python
accuracy(evaluate(X_test), y_test_5)
```

Devuelve: *0.9236*

Como podemos ver, nuestro modelo tiene una precisión cercana al $92\,\%$ tanto para el conjunto de entrenamiento como para el de evaluación. ¿Es este un buen modelo? A priori podríamos pensar que sí, ya que nuestro modelo acierta 9 de cada 10 imágenes… Sin embargo, la realidad es que nuestro modelo no está haciendo prácticamente nada. Vamos a ver por qué analizando nuestros datos. Como ya hemos comentado, estamos haciendo un clasificador binario para detectar el número 5. ¿Cuántas imágenes diferentes de 5s tenemos en nuestro dataset?

Python

```python
1 - y_train_5.mean(), 1 - y_test_5.mean()
```

Devuelve: *(0.90965, 0.9108)*

Aquí tenemos la respuesta, nuestro dataset está formado en un $91\,\%$ por imágenes que no son 5. Esto significa que un modelo *naive* que siempre diga que NO tenemos un 5 va a tener una *accuracy* del $91\,\%$. Sin embargo, en el mundo real, nunca detectaremos este dígito y nuestra aplicación fallará estrepitosamente. Nuestro modelo tiene una precisión del $92\,\%$, mejorando ligeramente este valor, pero no por mucho.

La métrica de precisión es muy útil, pero puede llevar a engaños. Esto se hace especialmente patente en aplicaciones en las que tenemos muy pocas muestras de la clase que queremos capturar. Un ejemplo claro es el de detección de transacciones bancarias fraudulentas, que representan cerca del 0.001 % del total de operaciones realizadas. Un modelo

de clasificación binaria que siempre diese como resultado que una transacción NO es fraudulenta tendría una precisión del 99.999 %, sin embargo sería un modelo totalmente inútil en el mundo real.

Siempre es recomendable tener unas primeras métricas `baseline` con las que comparar nuestros modelos. Estas métricas pueden obtenerse con un modelo aleatorio (antes de ser entrenado) o en el caso de la clasificación podemos utilizar directamente la distribución de clases en el dataset. Cualquier modelo que hagamos debería mejorar estas métricas.

Todas las métricas que vamos a ver están implementadas en la librería `Scikit Learn`, por lo que te recomiendo que las uses directamente para evitar errores en la implementación.

Python

```
from sklearn.metrics import accuracy_score

accuracy_score(y_test_5, evaluate(X_test))
```

Devuelve: *0.9236*

Python

```
perceptron = LogisticRegression(784)
epochs, lr = 200, 1e-3
perceptron.fit(X_train, y_train_5, epochs, lr, log_each=20, batch_size
    =1000)
```

Devuelve:

Epoch 20/200 Loss 0.06317891682335086
Epoch 40/200 Loss 0.06218961662104983
Epoch 60/200 Loss 0.06161661936506813
Epoch 80/200 Loss 0.061127063035041194
Epoch 100/200 Loss 0.06072296826516242
Epoch 120/200 Loss 0.06039388169206788
Epoch 140/200 Loss 0.06012566554261699
Epoch 160/200 Loss 0.05990659878826801
Epoch 180/200 Loss 0.05972774593149956
Epoch 200/200 Loss 0.05958217656443273

Python

```
y_pred = evaluate(X_test)
accuracy_score(y_test_5, y_pred)
```

Devuelve: *0.9769*

La matriz de confusión

Una matriz de confusión nos va a indicar exactamente cuáles son los puntos fuertes y débiles de nuestro clasificador. Para cada clase, vamos a calcular cuántos elementos han sido bien clasificados por nuestro modelo y cuántos han sido confundidos con otras clases (esta matriz nos será muy útil en clasificación en varias clases también).

Python

```
TP = np.sum((y_pred == 1) & (y_test_5 == 1))
TN = np.sum((y_pred == 0) & (y_test_5 == 0))
FP = np.sum((y_pred == 1) & (y_test_5 == 0))
FN = np.sum((y_pred == 0) & (y_test_5 == 1))

CM = [[TN, FP],
      [FN, TP]]
CM
```

Devuelve: *[[9017, 91], [140, 752]]*

En esta matriz podemos observar:

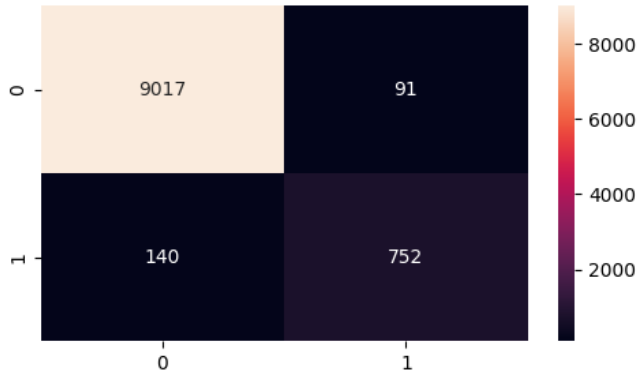
- Verdaderos Positivos (TP, *True Positives*): elementos que nuestro modelo clasifica correctamente como la clase que nos interesa, la clase positiva (fila 1, columna 1).
- Verdaderos Negativos (TN, *True Negatives*): elementos que nuestro modelo clasifica correctamente como la clase negativa (fila 0, columna 0).
- Falsos Positivos (FP, *False Positives*): elementos que nuestro modelo clasifica erróneamente como la clase positiva (fila 0, columna 1).
- Falsos Negativos (FN, *False Negatives*): elementos que nuestro modelo clasifica erróneamente como la clase negativa (fila 1, columna 0).

Python

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test_5, y_pred)
```

Devuelve:

array([[9017, 91],
[140, 752]])



Precision* y *Recall

Hemos visto la confusión a la que nos puede llevar una métrica simple como la *accuracy*. Ahora veremos dos métricas que son un poco más informativas: *precision* (no confundir con la precisión, en castellano, equivalente a la *accuracy*) y *recall*. Para definir estas métricas primero necesitamos definir los siguiente conceptos:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

Precision nos da una idea de cómo de propenso es nuestro modelo a dar falsos positivos. Un valor cercano a 1 indicará que nuestro modelo apenas da falsos positivos (aunque puede seguir fallando dando falsos negativos) mientras que un valor cercano a 0 indicará que

nuestro modelo da muchos falsos positivos. Lo mismo se aplica al *Recall*, pero en este caso aplicada a los falsos negativos. Un valor cercano a 1 indicará que nuestro modelo apenas da falsos negativos (aunque puede seguir fallando dando falsos positivos) mientras que un valor cercano a 0 indicará que nuestro modelo da muchos falsos negativos.

Python

```
def precision(y_pred, y):
    TP = np.sum((y_pred == 1) & (y == 1))
    FP = np.sum((y_pred == 1) & (y == 0))
    return TP / (TP + FP)

def recall(y_pred, y):
    TP = np.sum((y_pred == 1) & (y == 1))
    FN = np.sum((y_pred == 0) & (y == 1))
    return TP / (TP + FN)
```

Python

```
precision(y_pred, y_test_5), recall(y_pred, y_test_5)
```

Devuelve: *(0.8920521945432978, 0.8430493273542601)*

Python

```
from sklearn.metrics import precision_score, recall_score

precision_score(y_test_5, y_pred), recall_score(y_test_5, y_pred)
```

Devuelve: *(0.8920521945432978, 0.8430493273542601)*

En función de la aplicación en la que estemos trabajando vamos a querer optimizar una métrica u otra. En aplicaciones en las que tener falsos positivos sea perjudicial (por ejemplo, aplicaciones de seguridad) querremos modelos con un buena *precision*, mientras que en aplicaciones en las que tener falsos negativos sea perjudicial (por ejemplo, sistemas de diagnóstico médico) querremos modelos con buen *recall*. Una vez nuestro modelo ha sido entrenado, podemos ajustar estas métricas variando el valor del *threshold* utilizado en la evaluación. A esto se le conoce como el *precision-recall trade off*.

Python

```python
for t in np.linspace(0.1,0.9,20):
    y_pred = evaluate(X_test, t)
    print(f"Threshold: {t:.3f} Precision {precision(y_pred, y_test_5):.4f
        } Recall {recall(y_pred, y_test_5):.4f}")
```

Devuelve:

```
Threshold: 0.100 Precision 0.5630 Recall 0.9372
Threshold: 0.142 Precision 0.6342 Recall 0.9193
Threshold: 0.184 Precision 0.6914 Recall 0.9092
Threshold: 0.226 Precision 0.7322 Recall 0.9013
Threshold: 0.268 Precision 0.7669 Recall 0.8924
Threshold: 0.311 Precision 0.7976 Recall 0.8879
Threshold: 0.353 Precision 0.8234 Recall 0.8834
Threshold: 0.395 Precision 0.8557 Recall 0.8711
Threshold: 0.437 Precision 0.8746 Recall 0.8599
Threshold: 0.479 Precision 0.8847 Recall 0.8520
Threshold: 0.521 Precision 0.8993 Recall 0.8408
Threshold: 0.563 Precision 0.9067 Recall 0.8285
Threshold: 0.605 Precision 0.9180 Recall 0.8161
Threshold: 0.647 Precision 0.9216 Recall 0.7904
Threshold: 0.689 Precision 0.9332 Recall 0.7679
Threshold: 0.732 Precision 0.9420 Recall 0.7466
Threshold: 0.774 Precision 0.9497 Recall 0.7197
Threshold: 0.816 Precision 0.9591 Recall 0.6839
Threshold: 0.858 Precision 0.9694 Recall 0.6390
Threshold: 0.900 Precision 0.9754 Recall 0.5785
```

Como se puede observar, incrementar el valor de una de las métricas implica que la otra va a disminuir (es por esto que se le llama *trade-off*, ya que tenemos que llegar a un compromiso). Así pues, escogeremos un valor del *threshold* que se ajustó a los criterios de diseño de nuestra aplicación en particular (en función de la cantidad de falsos positivos y falsos negativos que estemos dispuestos a asumir).

F1-Score

Una métrica rápida que nos va a decir si nuestro modelo tiene buena *precision* y *recall* es el *F1-score*.

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

Es una métrica muy usada, ya que aglutina la información de ambas métricas. Sin embargo, va a favorecer mucho aquellos modelos con un valor similar de *precision* y *recall*, que no es siempre lo que vamos a querer.

Python

```
p = precision_score(y_test_5, y_pred)
r = recall_score(y_test_5, y_pred)

f1 = 2*(p*r)/(p+r)
f1
```

Devuelve: *0.7262491203377902*

Python

```
from sklearn.metrics import f1_score

f1_score(y_test_5, y_pred)
```

Devuelve: *0.7262491203377902*

La curva ROC

Podemos visualizar rápidamente el comportamiento de un modelo para varios *thresholds* con la llamada curva ROC (*Receiver Operating Characteristic*). En ella, representamos el ratio de verdaderos positivos (TPR) contra el ratio de falsos positivos (FPR), definidos de la siguiente manera:

$$TPR = \frac{TP}{TP + FN}$$

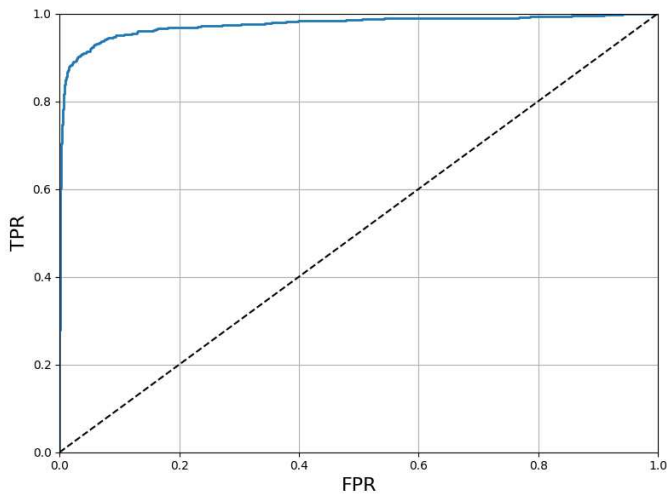
$$FPR = \frac{FP}{FP + TN}$$

Python

```
from sklearn.metrics import roc_curve

y_pred2 = perceptron.predict(X_test)
fpr, tpr, thresholds = roc_curve(y_test_5, y_pred2)
```

Python

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('FPR', fontsize=16)
    plt.ylabel('TPR', fontsize=16)
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)
plt.tight_layout()
plt.savefig("resources/roc.png")
plt.close()
```



Una métrica muy utilizada para comparar clasificadores es el área bajo la curva ROC, ya que nos indica cómo de robusto es un modelo. La línea recta representa un modelo *naive* que podemos usar como *baseline*. Como puedes ver, el área bajo la curva ROC de este modelo es de $0{,}5$, por lo que cualquier modelo que hagamos debería superar este valor. Cuánto más se acerque la curva a la esquina superior izquierda, mayor será el área y mejor será el modelo.

Python

```python
from sklearn.metrics import roc_auc_score

roc_auc_score(y_test_5, y_pred2)
```

Devuelve: *0.9752180362801341*

Resumen

- *Accuracy* es la métrica de clasificación más simple, consistente en contar cuántos elementos del dataset nuestro clasifica correctamente. Puede llevarnos a engaños si no tenemos en cuenta la distribución de nuestras clases, poniendo en relevancia la necesidad de tener métricas `baseline` para comprar nuestros primeros modelos.
- La matriz de confusión nos va indicar el número de elementos de nuestro dataset que el modelo clasifica bien, así como todos los que confunda con otras clases. De esta manera, sabremos cómo mejorar nuestro modelo.
- *Precision* y *Recall* nos indican cómo de robusto será nuestro modelo, cuantificando los falsos positivos y falsos negativos. En función de nuestra aplicación y su sensibilidad preferiremos una mejor métrica u otra (como hemos visto, mejorar una de ella implica empeorar la otra).
- El *F1-Score* nos dará una idea rápida de la robustez del modelo combinando las dos métricas anteriores.
- La curva *ROC*, y en especial el área debajo de la curva, será una muy buena manera de comparar modelos para poder elegir el que más se ajuste a nuestras necesidades.

Existen muchas otras métricas en función del problema que estemos tratando. Por ejemplo, en problemas de regresión podemos utilizar el *Mean Absolute Error* (MAE) o el *Mean Squared Error* (MSE). Como veremos más adelante, es una buena práctica monitorizar estas métricas durante el entrenamiento para poder detectar problemas en el modelo sin tener que esperar al final del proceso de optimización (que ahora mismo es rápido, pero en

los modelos más complejos puede durar horas o incluso días o semanas). También es importante recalcar que estas métricas pueden aplicarse a la clasificación en varias clases, no solo a la binaria.

Python

```
perceptron = SoftmaxRegression(784, 10)
epochs, lr = 100, 1e-3
perceptron.fit(X_train, y_train, epochs, lr, log_each=10, batch_size
    =1000)
```

Devuelve:

Epoch 10/100 Loss 1.8153115152383779
Epoch 20/100 Loss 1.431267954625735
Epoch 30/100 Loss 1.1828646698795682
Epoch 40/100 Loss 1.0153474282102097
Epoch 50/100 Loss 0.8968663138730597
Epoch 60/100 Loss 0.809409369809988
Epoch 70/100 Loss 0.7425111368155134
Epoch 80/100 Loss 0.6898185944081827
Epoch 90/100 Loss 0.6473008610741742
Epoch 100/100 Loss 0.6122962792103395

Python

```
y_pred = perceptron.predict(X_test)
y_pred = np.argmax(y_pred, axis=1)
```

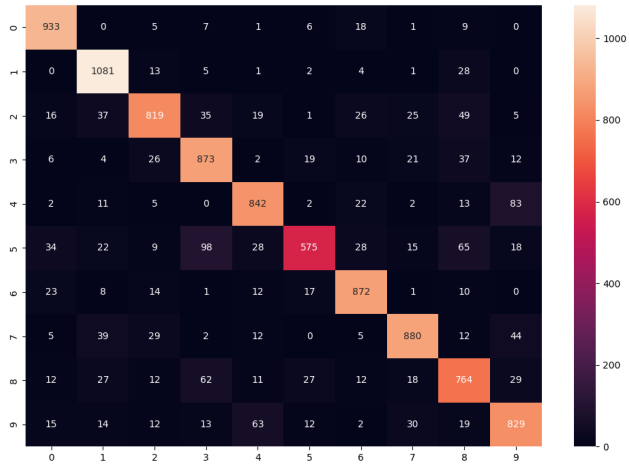
Python

```
accuracy_score(y_test, y_pred)
```

Devuelve: *0.8468*

Python

```python
import pandas as pd
import seaborn as sn

cm = confusion_matrix(y_test, y_pred)
df_cm = pd.DataFrame(cm, index = [i for i in range(0,10)], columns = [i
    for i in range(0,10)])
plt.figure(figsize = (10,7))
sn.heatmap(df_cm, annot=True, fmt='d')
plt.tight_layout()
plt.savefig("resources/cm2.png")
plt.close()
```



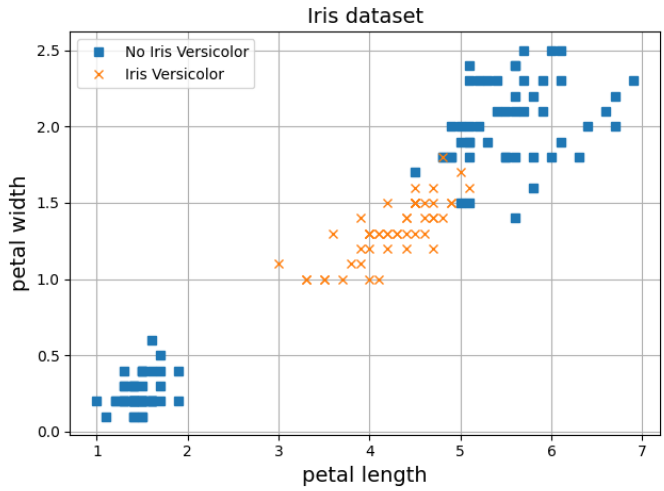
En la matriz de confusión podemos ver claramente qué dígitos se confunden entre sí (como, por ejemplo, el 3 y el 5), lo cual nos puede dar pistas de cómo mejorar nuestro modelo.

Limitaciones del Perceptrón

Si bien el `Perceptrón` es un algoritmo muy simple y capaz de llevar a cabo tareas de regresión y clasificación, tiene una serie de limitaciones que lo hacen poco útil en la práctica. La principal limitación es que no es capaz de resolver problemas que no sean linealmente separables. Esto significa que no es capaz de encontrar una frontera de decisión que separe las clases en el espacio de las características. En el caso de la regresión, no es capaz de encontrar una línea recta que se ajuste a los datos. En el caso de la clasificación, no es capaz de encontrar una frontera de decisión que separe las clases.

Python

```
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 1).astype(int)

plt.plot(X[y==0, 0], X[y==0, 1], 's', label="No Iris Versicolor")
plt.plot(X[y==1, 0], X[y==1, 1], 'x', label="Iris Versicolor")
plt.grid()
plt.legend()
plt.xlabel('petal length', fontsize=14)
plt.ylabel('petal width', fontsize=14)
plt.title("Iris dataset", fontsize=14)
plt.tight_layout()
plt.savefig("resources/perc_lims1.png")
plt.close()
```

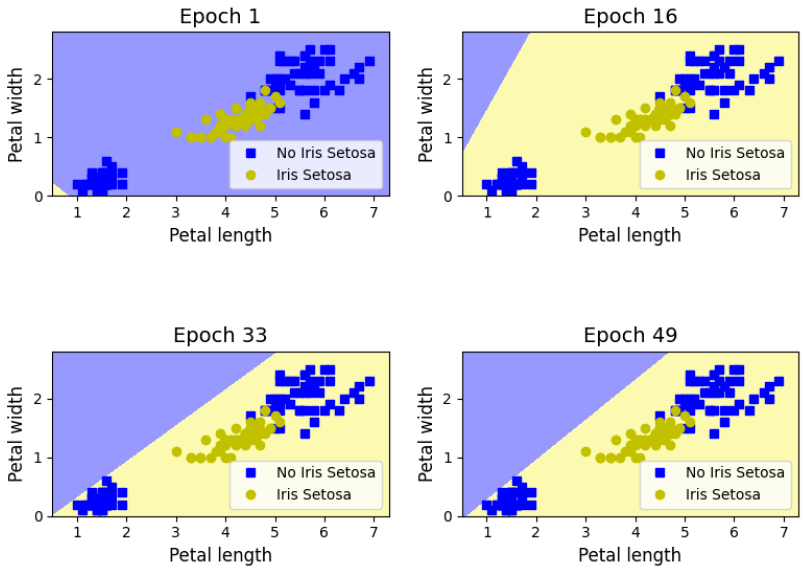


Python

```
perceptron = LogisticRegression(2, 1)
epochs, lr = 50, 0.1
perceptron.fit(X_norm, y, epochs, lr, log_each=10)
```

Devuelve:

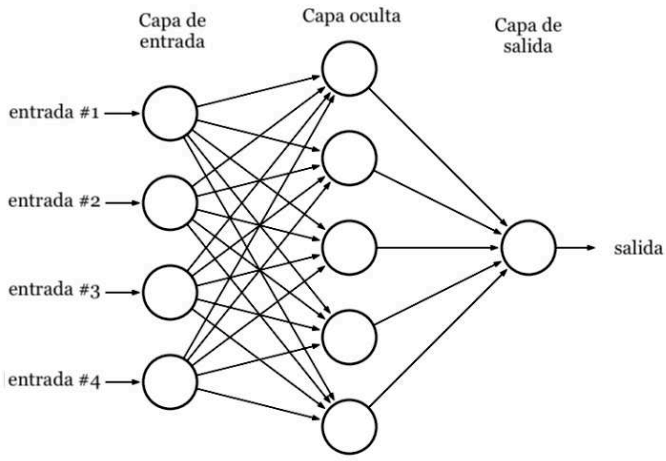
Epoch 10/50 Loss -0.4594576235232987 Epoch 20/50 Loss -2.2102337175122635
Epoch 30/50 Loss -0.34755302350903916 Epoch 40/50 Loss -1.4066474484457576
Epoch 50/50 Loss -1.009388048053112



Para sobreponerse a esta limitación necesitaremos utilizar modelos más complejos, que sí que sean capaces de encontrar fronteras de decisión no lineales. Ya conocemos uno de estos modelos, ya que lo hemos usado en los capítulos anteriores, el `Perceptrón Multicapa`. En la siguiente sección veremos cómo podemos implementar este modelo desde cero así como el algoritmo de entrenamiento.

El Perceptrón Multicapa

Como hemos visto, el Perceptrón es un algoritmo cuya pincipal limitación es la de no ser capaz de resolver problemas que no sean linealmente separables. Esto restringe su rango de aplicación a problemas simples o aquellos en los que un buen trabajo de ingeniería de características, o *feature engineering* en inglés, permita transformar los datos en un espacio en el que sí sean linealmente separables. Uno de los grandes avances en el campo de la IA es el desarrollo de modelos más complejos que son capaces de llevar a cabo este *feature engineering* de manera automática, aprendiendo a partir de ejemplos. Uno de estos modelos es el `Perceptrón Multicapa`, o *Multilayer Perceptron* (MLP), que nos permite resolver problemas que no sean linealmente separables, sobreponiéndose a la principal limitación del `Perceptrón`.



Como puedes ver en la figura, un MLP no es más que una secuencia de `Perceptrones`, las salidas de los cuales son a la vez las entradas para la siguiente capa (así hasta llegar a la salida). En el caso de un MLP de dos capas, podemos calcular la salida de la primera capa como:

$$\mathbf{h}_1 = f_1(\mathbf{w}_1 \cdot \mathbf{x})$$

Donde $\mathbf{h}_1$ es el estado oculto, *hidden state* en inglés, de la primera capa. Como puedes ver la expresión es la misma que la que usamos en el modelo `Perceptrón`, en el que $\mathbf{w}_1$ son los pesos de la primera capa y $\mathbf{x}$ las entradas. Si ahora consideramos $\mathbf{h}_1$ como las entradas de la siguiente capa, podemos encontrar la salida como:

$$\hat{y} = f_2(\mathbf{w}_2 \cdot \mathbf{h}_1) = f_2(\mathbf{w}_2 \cdot f_1(\mathbf{w}_1 \cdot \mathbf{x}))$$

En la que simplemente hemos vuelto a aplicar la misma expresión. Puedes intuir que si tenemos más capas, simplemente repetiremos la expresión para cada capa, usando las salidas de una como entradas de la siguiente de manera recurrente hasta llegar a la salida.

El algoritmo de *Backpropagation*

Una vez definida la arquitectura del MLP vamos a ver cómo entrenarlo. Para ello, vamos a utilizar el mismo algoritmo que ya conocemos para el `Perceptrón`: el algoritmo de `descenso por gradiente`. Primero, necesitamos una `función de pérdida`, de la cual calcularemos la derivada con respecto a los pesos del modelo. Para ilustrar el algoritmo vamos a asumir que utilizamos una función de activación de tipo `relu` en la primera capa, $f_1 = max(0, x)$, y una función de activación lineal para la segunda capa, ya que estamos interesados en utilizar nuestro MLP para regresión. En cuanto a la función de activación, como ya sabemos para problemas de regresión usamos el error medio cuadrático (MSE).

$$MSE(\hat{y}, y) = \frac{1}{N}\sum_{j=1}^{N}(\hat{y}^{(j)} - y^{(j)})^2$$

> Utilizamos funciones de activación no lineales en las capas ocultas de un MLP, ya que una combinación lineal de funciones lineales sigue siendo una función lineal, lo cual significaría que nuestro MLP no sería mejor que un `Perceptrón` simple. Por ejemplo, en el caso de un MLP con una capa oculta de una sola neurona, una entrada y una salida:
>
> $$\hat{y} = w_2^1 h_1 + w_2^0 = w_2^1(w_1^1 x_1 + w_1^0) + w_2^0 = w^1 x + w^0$$

Recordemos el algoritmo de `descenso por gradiente`:

1. Calcular la salida del modelo, $\hat{y}$.
2. Calcular la derivada de la función de pérdida con respecto a los parámetros del modelo, $\frac{\partial MSE}{\partial w}$.
3. Actualizar los parámetros, $w \leftarrow w - \eta\frac{\partial MSE}{\partial w}$, donde η es el *learning rate*.
4. Repetir hasta converger.

En el caso del `Perceptrón` vimos que podemos calcular la derivada de la función de pérdida con respecto a los pesos de la siguiente manera:

$$\frac{\partial MSE}{\partial w} = \frac{\partial MSE}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w} = \frac{2}{N}(\hat{y} - y)x$$

En el caso del MLP, esta expresión sigue siendo válida, pero en este caso solo nos sirve para encontrar la derivada de la función de pérdida con respecto a los pesos de la última capa. Para encontrar la derivada con respecto a los pesos de las capas anteriores tenemos que utilizar el algoritmo de `backpropagation`, que básicamente consiste en aplicar la regla de la cadena de la derivada hacia atrás en el MLP hasta llegar a la primera capa.

$$\frac{\partial MSE}{\partial w_2} = \frac{\partial MSE}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} = \frac{2}{N}(\hat{y} - y)h_1$$

$$\frac{\partial MSE}{\partial w_1} = \frac{\partial MSE}{\partial h_1} \frac{\partial h_1}{\partial w_1} = \frac{\partial MSE}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial w_1} = \frac{2}{N}(\hat{y} - y)\mathbf{w}_2 f_1'(\mathbf{w_1} \cdot \mathbf{x})x$$

Y de nuevo, si tenemos más de dos capas, simplemente repetimos el razonamiento para cada capa hasta llegar a la primera.

> Los *frameworks* de `redes neuronales`, como Pytorch o Tensorflow, se van guardando todas las operaciones que tienen lugar en una red neuronal en lo que se llama un `grafo computacional` de manera que cuando queremos derivar cualquier valor con respecto a los pesos de la red simplemente recorremos el grafo hacia atrás aplicando la regla de la cadena, o `backpropagation`. Esto nos permite diseñar arquitecturas de manera arbitraria sin tener que preocuparnos por calcular todas las derivadas de manera manual.

Vamos a implementar este algoritmo en `Python`.

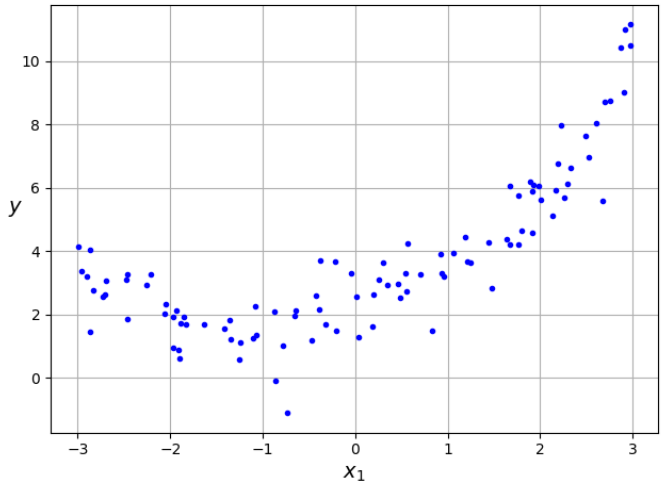
Python

```
def relu(x):
    return np.maximum(0, x)

def reluPrime(x):
    return x > 0

class MLP():
    def __init__(self, D_in, H, D_out):
        self.w1, self.b1 = np.random.normal(loc=0.0, scale=np.sqrt(2/(
            D_in+H)), size=(D_in, H)), np.zeros(H)
        self.w2, self.b2 = np.random.normal(loc=0.0, scale=np.sqrt(2/(H+
            D_out)), size=(H, D_out)), np.zeros(D_out)
        self.ws = []
```

```
        self.loss = mse
        self.grad_loss = grad_mse

    def __call__(self, x):
        self.h_pre = np.dot(x, self.w1) + self.b1
        self.h = relu(self.h_pre)
        y_hat = np.dot(self.h, self.w2) + self.b2
        return y_hat

    def fit(self, X, Y, epochs = 100, lr = 0.001, batch_size=None,
        verbose=True, log_each=1):
        batch_size = len(X) if batch_size == None else batch_size
        batches = len(X) // batch_size
        l = []
        for e in range(1,epochs+1):
                # Mini-Batch Gradient Descent
                _l = []
                for b in range(batches):
                        x = X[b*batch_size:(b+1)*batch_size]
                        y = Y[b*batch_size:(b+1)*batch_size]
                        y_pred = self(x)
                        loss = self.loss(y, y_pred)
                        _l.append(loss)
                        # Backprop
                        dldy = self.grad_loss(y, y_pred)
                        grad_w2 = np.dot(self.h.T, dldy)
                        grad_b2 = dldy.mean(axis=0)
                        dldh = np.dot(dldy, self.w2.T)*reluPrime(self.
                            h_pre)
                        grad_w1 = np.dot(x.T, dldh)
                        grad_b1 = dldh.mean(axis=0)
                        # Update (GD)
                        self.w1 = self.w1 - lr * grad_w1
                        self.b1 = self.b1 - lr * grad_b1
                        self.w2 = self.w2 - lr * grad_w2
                        self.b2 = self.b2 - lr * grad_b2
                l.append(np.mean(_l))
                self.ws.append((self.w1.copy(), self.b1.copy(), self.w2.
                    copy(), self.b2.copy()))
                if verbose and not e % log_each:
                        print(f'Epoch: {e}/{epochs}, Loss: {np.mean(l):.5
                            f}')

    def predict(self, ws, x):
        w1, b1, w2, b2 = ws
        h = relu(np.dot(x, w1) + b1)
        y_hat = np.dot(h, w2) + b2
        return y_hat
```

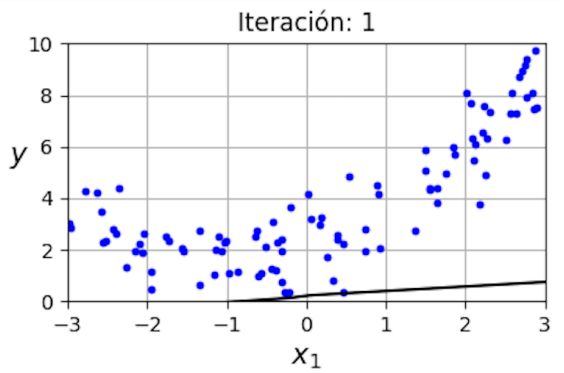
Python

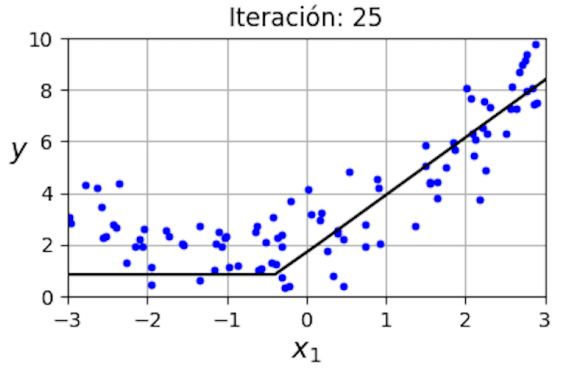
```python
import numpy as np
import matplotlib.pyplot as plt

m = 100
x = 6 * np.random.rand(m, 1) - 3
y = 0.5 * x**2 + x + 2 + np.random.randn(m, 1)

plt.plot(x, y, "b.")
plt.xlabel("$x_1$", fontsize=14)
plt.ylabel("$y$", rotation=0, fontsize=14)
plt.grid(True)
plt.tight_layout()
plt.savefig("resources/mlp_bp1.png")
plt.close()
```

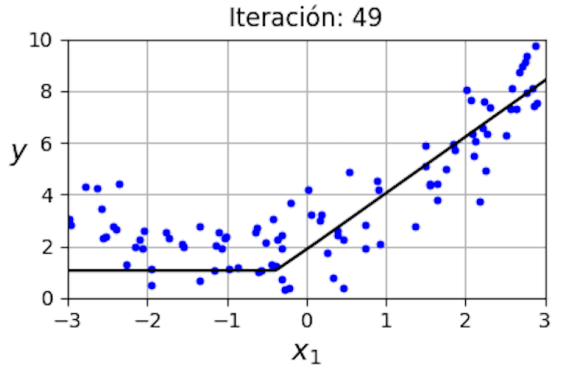


Python

```python
model = MLP(D_in=1, H=3, D_out=1)
epochs, lr = 50, 0.0002
model.fit(x, y, epochs, lr, batch_size=1, log_each=10)
```


Iteración: 1
y
x_1


Iteración: 25
y
x_1


Iteración: 49
y
x_1

Nuestro modelo ahora es capaz de ajustarse a distribuciones de datos no lineales. Esto es debido a que la arquitectura del MLP es capaz de encontrar fronteras de decisión no lineales en el espacio de las características. Esto se puede visualizar en el ejemplo anterior en el que hemos podido ajustar una distribución polinómica con lo que parecen 3 rectas, y no es casualidad que el modelo en su capa oculta tenga 3 neuronas. Como ya sabrás, las redes neuronales más grandes pueden tener del orden de millones de neuronas en sus múltiples capas ocultas, lo que les permite ajustarse a distribuciones de datos muy complejas tales como imágenes, texto, audio, etc.

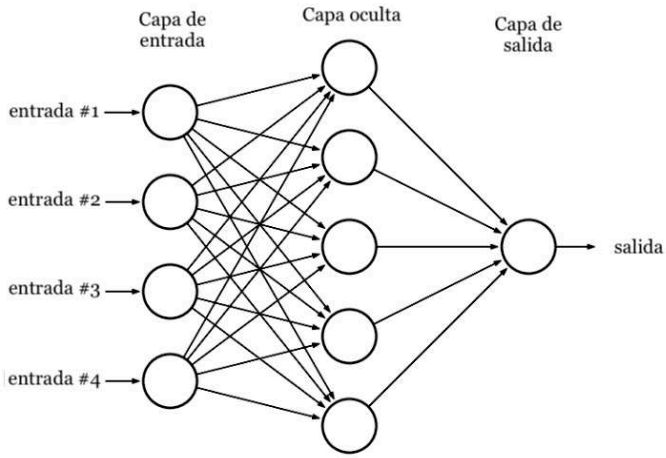
En el siguiente capítulo veremos diferentes tipos de redes neuronales, diseñadas para diferentes tipos de problemas. Aun así, todas ellas se basan en los conceptos presentados en este capítulo. Gracias a las librerías de redes neuronales, como Pytorch, podemos diseñar redes neuronales de manera arbitraria, ya que es el mismo *framework* el que se encarga de calcular todas las derivadas necesarias para el entrenamiento de la red.

5. Arquitecturas de Redes Neuronales

Si has llegado hasta este punto, permíteme felicitarte. Sé que el capítulo anterior ha sido quizás algo complicado, pero créeme que entender todos los conceptos presentados marcará la diferencia en tu carrera en Inteligencia Artificial. Encaramos ahora la recta final del libro, en la que en primer lugar veremos diferentes tipos de redes neuronales para posteriormente terminar con consejos prácticos para su entrenamiento y aplicación.

El Perceptrón Multicapa

Uno de los modelos más simples y conocidos de redes neuronales es el Perceptrón Multicapa (MLP, por sus siglas en inglés). En el capítulo anterior ya presentamos este modelo en detalle, haciendo una implementación del mismo desde cero con Python así como también del algoritmo de entrenamiento (por lo que te recomiendo que le eches un vistazo si no lo has hecho y quieres entender cómo funciona).



El MLP es un tipo de red neuronal formado por múltiples capas de neuronas consecutivas, de manera secuencial, en la que todas las neuronas de una capa están conectadas con todas las neuronas de la capa siguiente. La primera capa de la red es la capa de entrada, la última capa es la de salida y las capas intermedias se conocen como capas ocultas.

Este modelo es muy usado en la práctica cuando la dimensionalidad de los datos no es muy elevada, ya que al ser un modelo denso en el que todas las neuronas de una capa están conectadas con todas las neuronas de la capa siguiente, el número de parámetros a entrenar crece de manera exponencial y puede rápidamente volverse inmanejable cuando trabajamos con datos de alta dimensionalidad como imágenes o texto.

El ejemplo con el que hemos ido trabajando a lo largo del libro, usando el dataset de MNIST, es un ejemplo de aplicación de un MLP.

Python

```python
from sklearn.datasets import fetch_openml
import numpy as np


# descarga de datos

mnist = fetch_openml('mnist_784', version=1)
X, Y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

Podemos definir un modelo MLP con Pytorch de la siguiente manera:

Python

```python
from torch.nn import Sequential as S
from torch.nn import Linear as L
from torch.nn import ReLU as R

model = S(L(784,128),R(),L(128,10))
model
```

Devuelve: *Sequential((0): Linear(in_features=784, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=10, bias=True)*

Este MLP tiene una capa de entrada de 784 neuronas (el número de píxeles de las imágenes de MNIST), dos capas ocultas de 128 neuronas cada una y una capa de salida de 10 neuronas (una por cada dígito del 0 al 9). La capa `Sequential` nos permite definir una se-

cuencia de capas de manera muy sencilla mientras que la capa `Linear` nos permite definir una capa densa de neuronas. Es importante el uso de funciones de activación no lineales entre capas, en este caso usamos la función `ReLU`.

Python

```python
import torch

class Dataset(torch.utils.data.Dataset):

    # constructor
    def __init__(self, X, Y):
        self.X = torch.tensor(X).float()
        self.Y = torch.tensor(Y).long()

    # cantidad de muestras en el dataset
    def __len__(self):
        return len(self.X)

    # devolvemos el elemento `ix` del dataset
    def __getitem__(self, ix):
        return self.X[ix], self.Y[ix]
```

Python

```python
def train(model, epochs = 5, batch_size=1000):
    dataset = {
        "train": Dataset(X[:60000], Y[:60000]), # 60.000 imágenes para
            entrenamiento
        "val": Dataset(X[60000:], Y[60000:])    # 10.000 imágenes para
            validación
    }
    dataloader = {
        'train': torch.utils.data.DataLoader(dataset['train'], batch_size
            =batch_size, shuffle=True, num_workers=4, pin_memory=True),
        'val': torch.utils.data.DataLoader(dataset['val'], batch_size=
            batch_size, num_workers=4, pin_memory=True)
    }
    model.cuda()
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters())
    for e in range(1, epochs+1):
        print(f"epoch: {e}/{epochs}")
        # entrenamiento
        model.train()
        for batch_ix, (x, y) in enumerate(dataloader['train']):
            x, y = x.cuda(), y.cuda()
            optimizer.zero_grad()
            with torch.autocast(device_type='cuda', dtype=torch.bfloat16)
                :
                outputs = model(x)
                loss = criterion(outputs, y)
```

```
            loss.backward()
            optimizer.step()
            if batch_ix % 10 == 0:
                loss, current = loss.item(), (batch_ix + 1) * len(x)
                print(f"loss: {loss:.4f} [{current:>5d}/{len(dataset['
                    train']):>5d}]")
        # validación
        model.eval()
        val_loss, val_acc = [], []
        with torch.no_grad():
            for batch_ix, (x, y) in enumerate(dataloader['val']):
                x, y = x.cuda(), y.cuda()
                with torch.autocast(device_type='cuda', dtype=torch.
                    bfloat16):
                    outputs = model(x)
                    loss = criterion(outputs, y)
                val_loss.append(loss.item())
                val_acc.append((outputs.argmax(1) == y).float().mean().
                    item())
        print(f"val_loss: {np.mean(val_loss):.4f} val_acc: {np.mean(
            val_acc):.4f}")
```

Python

```
train(model)
```

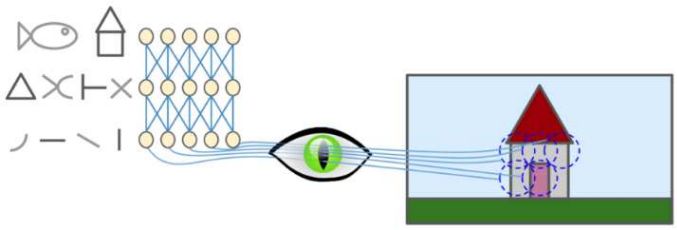
Devuelve: *epoch: 1/5 loss: 29.5178 [1000/60000] loss: 2.0781 [11000/60000] loss: 1.3114 [21000/60000] loss: 0.8793 [31000/60000] loss: 0.6140 [41000/60000] loss: 0.6180 [51000/60000] val_loss: 0.4771 val_acc: 0.8916 epoch: 2/5 loss: 0.4765 [1000/60000] loss: 0.4008 [11000/60000] loss: 0.3830 [21000/60000] loss: 0.3384 [31000/60000] loss: 0.3624 [41000/60000] loss: 0.2999 [51000/60000] val_loss: 0.2965 val_acc: 0.9247 epoch: 3/5 loss: 0.1929 [1000/60000] loss: 0.2449 [11000/60000] loss: 0.2013 [21000/60000] loss: 0.2444 [31000/60000] loss: 0.2131 [41000/60000] loss: 0.1776 [51000/60000] val_loss: 0.2341 val_acc: 0.9381 epoch: 4/5 loss: 0.1505 [1000/60000] loss: 0.1616 [11000/60000] loss: 0.2334 [21000/60000] loss: 0.1522 [31000/60000] loss: 0.1798 [41000/60000] loss: 0.1417 [51000/60000] val_loss: 0.2024 val_acc: 0.9455 epoch: 5/5 loss: 0.0856 [1000/60000] loss: 0.1345 [11000/60000] loss: 0.1565 [21000/60000] loss: 0.1449 [31000/60000] loss: 0.1473 [41000/60000] loss: 0.1335 [51000/60000] val_loss: 0.1922 val_acc: 0.9490*

Redes Neuronales Convolucionales

El siguiente tipo de red neuronal que vamos a ver son las redes convolucionales (CNNs, por sus siglas en inglés, *Convolutional Neural Networks*). Esta familia de modelos surgieron del estudio del córtex visual del cerebro y se han utilizado en el reconocimiento de imágenes desde la década de los 80. En los últimos años, gracias al aumento de la potencia computacional, la cantidad de datos de entrenamiento disponibles y algunos trucos que veremos en el siguiente capítulo para entrenar redes profundas, las CNNs han logrado un rendimiento sobrehumano en algunas tareas visuales complejas. Este tipo de red se puede encontrar en servicios de búsqueda de imágenes, coches autónomos, sistemas de clasificación de vídeo automático y muchas otras aplicaciones. Además, las CNNs no se limitan a la percepción visual, también tienen éxito en tareas como el reconocimiento de voz y el procesamiento de datos tridimensionales.

El córtex visual

Las neuronas del córtex visual tienen un pequeño *campo receptivo local*, lo que significa que reaccionan solo a los estímulos visuales ubicados en una región limitada del campo visual. Los campos receptivos de diferentes neuronas pueden superponerse y juntos forman el campo visual completo.

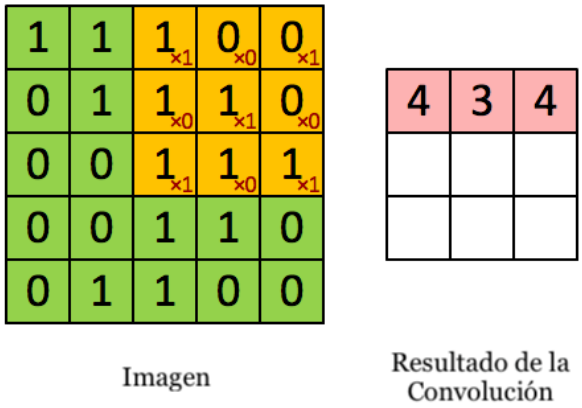


Las neuronas de las primeras capas del córtex visual reaccionan solo ante ciertos patrones simples, como, por ejemplo, líneas horizontales, mientras que otras reaccionan a líneas verticales (dos neuronas pueden tener el mismo campo receptivo pero reaccionan a diferentes orientaciones de línea). En las capas siguientes, las neuronas tienen campos receptivos más grandes y reaccionan a patrones más complejos, que pueden ser combinaciones de patrones de nivel inferior, construyendo de esta manera una jerarquía en diferentes capas que resultan finalmente en las formas y colores que vemos.

Basándose en estos conceptos, Yann LeCunn introdujo las CNNs en 1998 en la famosa arquitectura LeNet-5, utilizada por los bancos para reconocer de manera automática los números manuscritos en cheques para un procesamiento más rápido.

La Capa Convolucional

Las `redes convolucionales` están formadas por varias capas con diferentes responsabilidades. De entre estas capas, la más importante es la `capa convolucional` que es la responsable de identificar y construir las diferentes formas, colores y texturas de manera similar al córtex visual. Para llevar a cabo esta tarea usaremos un conjunto de *filtros* (también llamados *kernels*) los cuales deslizaremos por toda la imagen aplicando la operación *convolución*. Esta operación consiste en aplicar el producto escalar entre el filtro y los píxeles de la imagen cubiertos por este, lo que se conoce como el *campo receptivo* (o *receptive field*). En la siguiente imagen puedes ver esta operación en acción, en la que tenemos un filtro de 3x3 que deslizamos por nuestra imagen, la cual tiene una resolución de 5x5. Para cada posible posición del filtro dentro de la imagen, calculamos el producto de cada píxel por el valor del filtro correspondiente y guardamos el resultado en el mapa de salida.



En el caso general, aplicaremos esta operación en imágenes con varios canales (por ejemplo, una imagen en color RGB). En este caso, nuestros filtros también tendrán 3 canales. Además, se suelen aplicar más de un filtro, lo cual resulta en número de canales en el mapa de salida igual al número de filtros utilizados.

Vamos a ver un ejemplo de aplicación con imágenes reales del dataset CIFAR10 con el que ya hemos trabajado anteriormente.

Python

```python
import torchvision

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
    download=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
    download=True)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Python

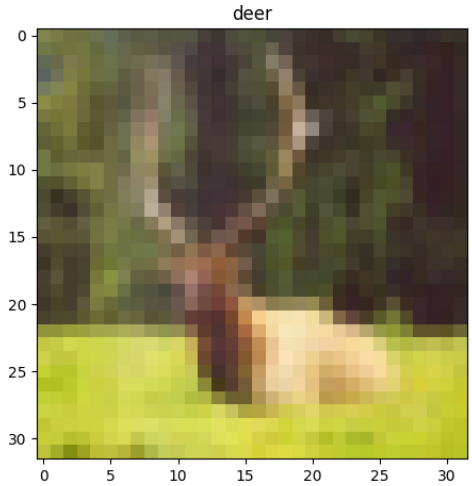
```python
import numpy as np

train_imgs, train_labels = np.array([np.array(i[0]) for i in trainset]),
    np.array([i[1] for i in trainset])
test_imgs, test_labels = np.array([np.array(i[0]) for i in testset]), np.
    array([i[1] for i in testset])

train_imgs.shape, test_imgs.shape
```

Devuelve: *((50000, 32, 32, 3), (10000, 32, 32, 3))*

Una vez convertidas todas las imágenes a *arrays* de `Numpy`, vamos a visualizar un ejemplo aleatorio.

Python

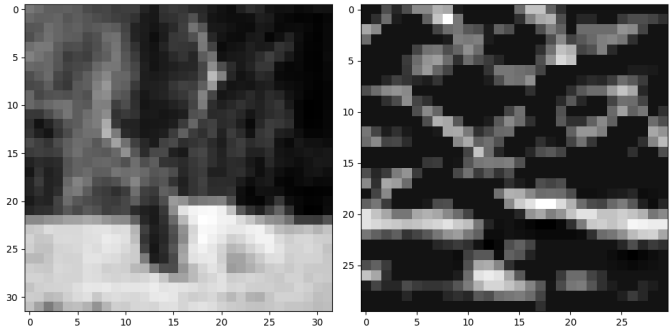
```python
import random
import matplotlib.pyplot as plt

ix = random.randint(0, len(train_imgs))
img, label = train_imgs[ix], train_labels[ix]

plt.imshow(img)
plt.title(classes[label])
plt.tight_layout()
plt.savefig('resources/cnns1.png')
plt.close()
```



Ahora, vamos a aplicar un filtro definido manualmente a esta imagen. En este caso, aplicaremos un filtro de 3x3 con valores de 1 en la primera fila, 0 en la central y -1 en la última. Como puedes ver en el resultado, este filtro es útil para identificar líneas horizontales.

Python

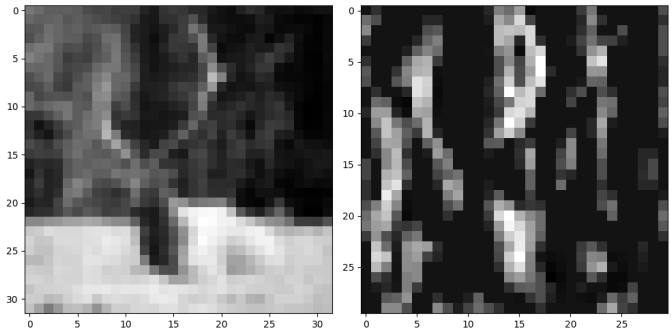
```python
import numpy as np
import scipy.signal
from skimage import color
from skimage import exposure

img = color.rgb2gray(img)

kernel = np.array([[1,1,1],
                   [0,0,0],
                   [-1,-1,-1]])

edges = scipy.signal.convolve2d(img, kernel, 'valid')
edges = exposure.equalize_adapthist(edges/np.max(np.abs(edges)),
    clip_limit=0.03)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))
ax1.imshow(img, cmap=plt.cm.gray)
ax2.imshow(edges, cmap=plt.cm.gray)
plt.tight_layout()
plt.savefig('resources/cnns2.png')
plt.close()
```



Si aplicamos ahora el mismo filtro, pero transpuesto, obtenemos un detector de líneas verticales.

Python

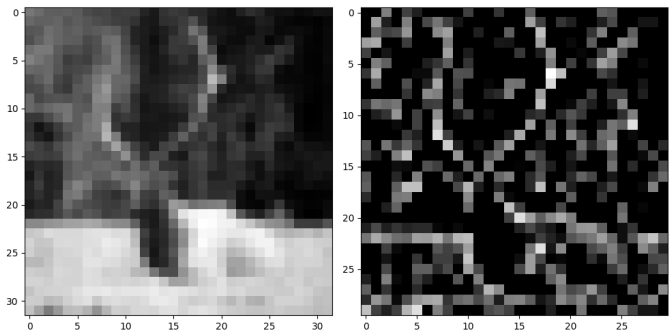
```
kernel = np.array([[1,0,-1],
                   [1,0,-1],
                   [1,0,-1]])

edges = scipy.signal.convolve2d(img, kernel, 'valid')
edges = exposure.equalize_adapthist(edges/np.max(np.abs(edges)),
    clip_limit=0.03)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))
ax1.imshow(img, cmap=plt.cm.gray)
ax2.imshow(edges, cmap=plt.cm.gray)
plt.tight_layout()
plt.savefig('resources/cnns3.png')
plt.close()
```



De la misma manera, el siguiente filtro será útil para detectar bordes en cualquier orientación.

Python

```
kernel = np.array([[0,-1,0],
                   [-1,4,-1],
                   [0,-1,0]])

edges = scipy.signal.convolve2d(img, kernel, 'valid')
edges = exposure.equalize_adapthist(edges/np.max(np.abs(edges)),
    clip_limit=0.03)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))
ax1.imshow(img, cmap=plt.cm.gray)
ax2.imshow(edges, cmap=plt.cm.gray)
plt.tight_layout()
plt.savefig('resources/cnns4.png')
plt.close()
```



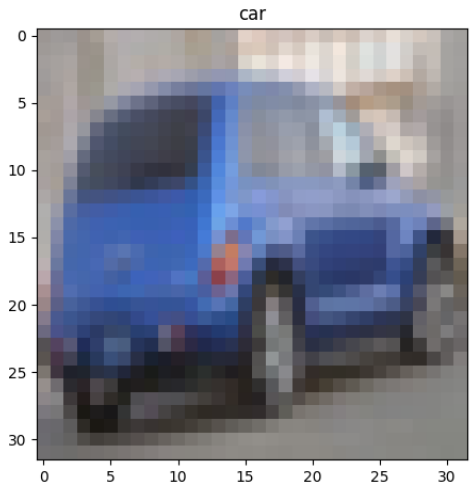
Como puedes ver, al aplicar varios filtros a nuestra imagen podemos obtener información relevante a la hora de llevar a cabo tareas tales como la clasificación de la imagen, detectar varios objetos en ella o generar una descripción textual de la misma. La pregunta ahora es: ¿Y cómo decidimos qué filtros utilizar? La respuesta es fácil, dejaremos que sea la propia red neuronal quien aprenda estos valores a través del proceso de entrenamiento, de manera que sea ella misma quien decida qué patrones son los más importantes a la hora de llevar a cabo su tarea. Así pues, los filtros serán ahora los parámetros de nuestra red.

Implementación en Pytorch

En `Pytorch` tenemos implementada la capa convolucional en el clase `torch.nn.Conv2D`. A esta capa le pasamos como parámetros el número de canales de la imagen a la entrada, el número de filtros, el tamaño del filtro y otros parámetros relevantes de los que hablaremos más adelante. Si miras en la documentación, esta capa espera un tensor a la entrada con dimensiones (N, C_{in}, H, W), dónde N es el tamaño del *batch*, C_{in} es el número de canales del mapa de entrada, H es el alto y W el ancho del mapa.

Python

```
ix = random.randint(0, len(train_imgs))
img, label = train_imgs[ix], train_labels[ix]

plt.imshow(img)
plt.title(classes[label])
plt.tight_layout()
plt.savefig('resources/cnns5.png')
plt.close()
```



Python

```
# convertir la imágen en tensor con dimensiones (N, C_in, H, W)

img_tensor = torch.from_numpy(img / 255.).unsqueeze(0)
img_tensor = img_tensor.permute(0, 3, 1, 2).float()

img_tensor.shape, img_tensor.dtype
```

Devuelve: *(torch.Size([1, 3, 32, 32]), torch.float32)*

Python

```
# aplicamos 10 filtros de tamaño 3x3

conv = torch.nn.Conv2d(in_channels = 3, out_channels = 10, kernel_size =
    3)

output = conv(img_tensor)


# dimensiones: (N, #filtros, H', W')
output.shape
```

Devuelve: *torch.Size([1, 10, 30, 30])*

Como puedes ver las dimensiones del tensor de salida son diferentes al tensor de entrada. En primer lugar, el número de canales del mapa de entrada ahora es el número de filtros aplicados (el resultado de aplicar cada filtro se guarda en un canal). En cuanto al ancho y alto, dependerá de la relación entre el tamaño de la imagen y el del filtro. Puedes calcular estas dimensiones de la siguiente manera:

$$o = \lfloor \frac{n + 2p - m}{s} \rfloor + 1$$

Donde o es la dimensión de salida, n la de entrada, m es el tamaño del filtro y p y s son dos parámetros con los que podemos jugar para ajustar el tamaño de salida. p es el *padding* y consiste en el número de valores extra que añadimos en los bordes para aumentar el tamaño de la entrada. s es el *stride*, y controla el número de píxeles que el filtro salta en cada paso. $\lfloor \cdot \rfloor$ es el operador *floor*, que redondea cualquier resultado a la baja.

Usar $p = 1$ y $s = 1$ con un filtro de 3x3 no cambiará el tamaño de la imagen.

Python

```
conv = torch.nn.Conv2d(in_channels = 3, out_channels = 10, kernel_size =
    3, padding = 1, stride = 1)

output = conv(img_tensor)


# dimensiones: (N, #filtros, H', W')
output.shape
```

Devuelve: *torch.Size([1, 10, 32, 32])*

Aplicar un salto de 2 píxeles en la aplicación del filtro, $s = 2$, reducirá el tamaño a la mitad.

Python

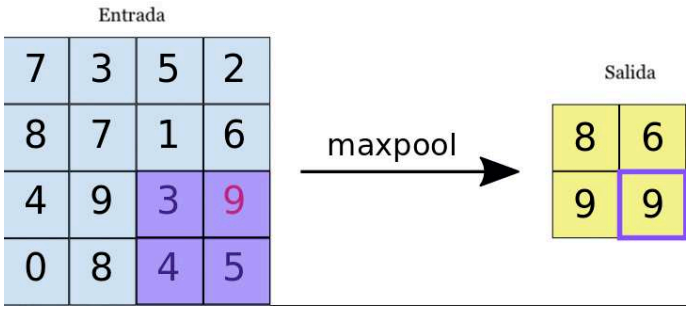
```
conv = torch.nn.Conv2d(in_channels = 3, out_channels = 10, kernel_size =
    3, padding = 1, stride = 2)

output = conv(img_tensor)


# dimensiones: (N, #filtros, H', W')
output.shape
```

Devuelve: *torch.Size([1, 10, 16, 16])*

En una `red convolucional` tendremos varias de estas `capas convolucionales` en diferentes capas consecutivas, de manera que las entradas a unas capas serán las salidas de las anteriores. De esta manera, la red será capaz de construir patrones cada vez más elaborados a partir de patrones más sencillos. Puedes ver una animación del funcionamiento de esta capa en el siguiente vídeo.

Capas de *Pooling*

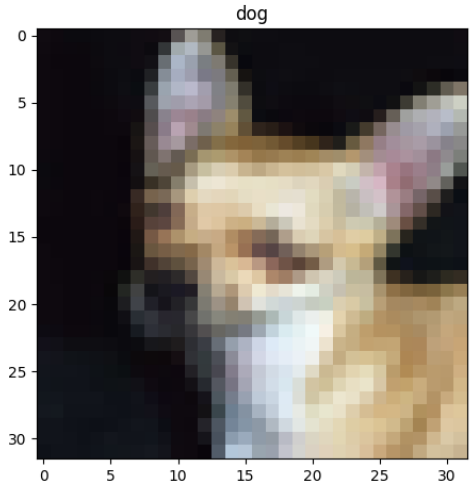
Si bien hemos visto que jugando con los tamaños del filtro, *stride* y *padding* podemos controlar el tamaño de los mapas generados por las capas convolucionales, es también común el uso de capas *pooling* para reducir los mapas de características.



Estas capas también aplican un filtro sobre su entrada, pero en este caso es un solo filtro que además no tiene parámetros, sino que aplica una función predeterminada en su campo receptivo (mínimo, máximo, promedio, etc.). La idea detrás del uso de este tipo de capas es la de reducir la resolución de los mapas de características, reduciendo así el coste computacional para entrenar la `red neuronal`, pero manteniendo las características importantes para el reconocimiento de patrones.

Python

```
ix = random.randint(0, len(train_imgs))
img, label = train_imgs[ix], train_labels[ix]

plt.imshow(img)
plt.title(classes[label])
plt.tight_layout()
plt.savefig('resources/cnns6.png')
plt.close()
```



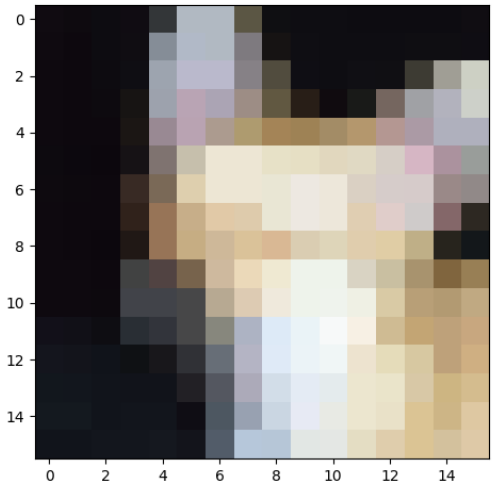
Python

```
pool = torch.nn.MaxPool2d(3, padding=1, stride=2)

img_tensor = torch.from_numpy(img / 255.).unsqueeze(0).permute(0, 3, 1,
    2).float()
output = pool(img_tensor)
output.shape
```

Devuelve: *torch.Size([1, 3, 16, 16])*

Python

```
plt.imshow(output.squeeze(0).permute(1,2,0))
plt.tight_layout()
plt.savefig('resources/cnns7.png')
plt.close()
```



Redes Convolucionales

Una vez hemos visto los componentes principales de las `redes convolucionales`, vamos a ver un ejemplo de cómo podemos implementar una red convolucional completa para, en este caso, la clasificación de las imágenes en el dataset MNIST. Vamos a definir una `red convolucional` con varias `capas convolucionales` y `capas de pooling`. Para poder clasificar las imágenes, conectaremos las salidas de la última capa de la red convolucional con un `MLP` para obtener las predicciones finales.

Python

```
    def block(c_in, c_out, k=3, p=1, s=1, pk=2, ps=2):
        return torch.nn.Sequential(
            torch.nn.Conv2d(c_in, c_out, k, padding=p, stride=s),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(pk, stride=ps)
        )

    def block2(c_in, c_out):
        return torch.nn.Sequential(
            torch.nn.Linear(c_in, c_out),
            torch.nn.ReLU()
        )

    class CNN(torch.nn.Module):
      def __init__(self, n_channels=1, n_outputs=10):
        super().__init__()
        self.conv1 = block(n_channels, 64)
```

```
    self.conv2 = block(64, 128)
    self.fc = torch.nn.Linear(128*7*7, n_outputs)

  def forward(self, x):
    print("Dimensiones:")
    print("Entrada: ", x.shape)
    x = self.conv1(x)
    print("conv1: ", x.shape)
    x = self.conv2(x)
    print("conv2: ", x.shape)
    x = x.view(x.shape[0], -1)
    print("pre fc: ", x.shape)
    x = self.fc(x)
    print("Salida: ", x.shape)
    return x
```

Python

```
model = CNN()

output = model(torch.randn(64, 1, 28, 28))
```

Devuelve: *Dimensiones: Entrada: torch.Size([64, 1, 28, 28]) conv1: torch.Size([64, 64, 14, 14]) conv2: torch.Size([64, 128, 7, 7]) pre fc: torch.Size([64, 6272]) Salida: torch.Size([64, 10])*

Utilizando capas convolucionales con filtros de 3x3, *padding* 1 y *stride* 1, mantenemos las dimensiones constantes, dejando la responsabilidad de reducir la resolución a las capas *pooling*. Tras cada capa convolucional, el número de canales corresponde con el número de filtros aplicados. Para conectar la última salida de la capa convolucional a la capa lineal, tenemos que estirar el último mapa de características en un vector de una sola dimensión.

Python

```
class CNN(torch.nn.Module):
  def __init__(self, n_channels=1, n_outputs=10):
    super().__init__()
    self.conv1 = block(n_channels, 64)
    self.conv2 = block(64, 128)
    self.fc = torch.nn.Linear(128*7*7, n_outputs)

  def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = x.view(x.shape[0], -1)
    x = self.fc(x)
    return x
```

Ahora, podemos entrenar la red utilizando nuestro bucle de entrenamiento, aunque para que todo funcione vamos a necesitar darle las imágenes con las dimensiones correctas: número de canales, alto y ancho.

Python

```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, X, Y):
        self.X = torch.tensor(X).float()
        self.Y = torch.tensor(Y).long()

    def __len__(self):
        return len(self.X)

    # devolvemos las imágenes con dimensiones (C, H, W)
    def __getitem__(self, ix):
        return self.X[ix].reshape(1, 28, 28), self.Y[ix]
```

Python

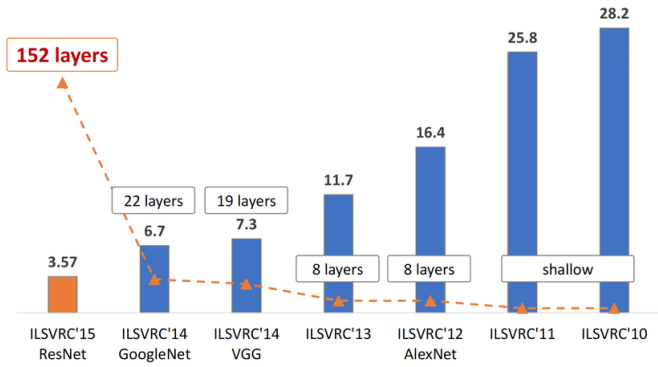
```
model = CNN()

train(model)
```

Devuelve:

epoch: 1/5 loss: 20.1746 [1000/60000] loss: 1.8033 [11000/60000] loss: 1.0030 [21000/60000] loss: 0.4555 [31000/60000] loss: 0.2780 [41000/60000] loss: 0.2519 [51000/60000] val_loss: 0.1671 val_acc: 0.9450 epoch: 2/5 loss: 0.1712 [1000/60000] loss: 0.1634 [11000/60000] loss: 0.1506 [21000/60000] loss: 0.1099 [31000/60000] loss: 0.1224 [41000/60000] loss: 0.1071 [51000/60000] val_loss: 0.0963 val_acc: 0.9671 epoch: 3/5 loss: 0.0904 [1000/60000] loss: 0.1178 [11000/60000] loss: 0.0892 [21000/60000] loss: 0.0723 [31000/60000] loss: 0.1058 [41000/60000] loss: 0.0750 [51000/60000] val_loss: 0.0751 val_acc: 0.9770 epoch: 4/5 loss: 0.0738 [1000/60000] loss: 0.0869 [11000/60000] loss: 0.0904 [21000/60000] loss: 0.0529 [31000/60000] loss: 0.0797 [41000/60000] loss: 0.0793 [51000/60000] val_loss: 0.0646 val_acc: 0.9794 epoch: 5/5 loss: 0.0802 [1000/60000] loss: 0.0586 [11000/60000] loss: 0.0677 [21000/60000] loss: 0.0872 [31000/60000] loss: 0.0470 [41000/60000] loss: 0.0687 [51000/60000] val_loss: 0.0651 val_acc: 0.9774

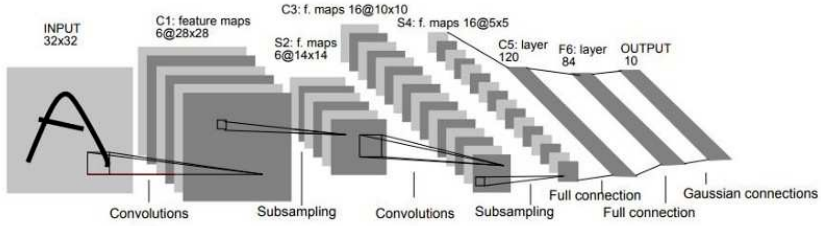
Arquitecturas de Redes Convolucionales

En las secciones anteriores hemos introducido la arquitectura de `red neuronal convolucional`, un modelo de `red neuronal` especialmente diseñado para trabajar con imágenes en tareas de visión artificial. Entre estas tareas, la más común es la de clasificación de imágenes, que consiste simplemente en asignar una categoría de entre varias a una imagen en particular. Una buena manera para medir el progreso en el campo del `Machine Learning` es a través de `benchmarks` y competiciones en los que investigadores proponen diferentes soluciones a un mismo problema, que se evalúan sobre un mismo conjunto de datos de test oculto. En el campo del `deep learning` y `computer vision`, una de estas competiciones, que hoy en día ya es un `benchmark` establecido, es la clasificación de imágenes en el dataset Imagenet. En 2012, la primera `red neuronal convolucional` entró en esta competición consiguiendo una mejora de 10 puntos sobre la mejor solución hasta la fecha, lo que supuso un detonante para el campo del `deep learning`. A partir de ese momento, todas las soluciones ganadoras presentadas durante los años siguientes han sido redes convolucionales, convirtiendo a Imagenet una estupenda fuente de información para conocer los últimos avances en el campo (cuando una nueva arquitectura es diseñada, es práctica común reportar sus prestaciones en Imagenet para poder compararla con otras redes). En esta sección vamos a revisar algunas de las arquitecturas más conocidas que han aparecido durante los años, las cuales podremos usar para nuestras aplicaciones. Puedes conocer en detalle del progreso de este `benchmark` en *papers with code*, donde también encontrarás artículos y código en muchísimos otros campos de aplicación.



LeNet-5

Empezamos revisando la primera arquitectura de CNN desarrollada. Esta red es conocida por el nombre de LeNet-5 y, si bien no participó en la competición de Imagenet, es interesante conocer esta primera arquitectura.



Esta CNN es alimentada por imágenes de 32x32 píxeles de dígitos manuscritos, y está formada por tres capas convolucionales, dos de las cuales reducen la dimensionalidad de los mapas de características mediante *average pooling*. La salida de la última capa convolucional es conectada un perceptrón multicapa de dos capas, que tienen la responsabilidad de dar la predicción final. Puedes ver más detalles en el artículo original. A continuación, puedes ver un ejemplo de implementación en Pytorch.

Python

```python
import torch

def block(c_in, c_out, k=3, p=1, s=1):
    return torch.nn.Sequential(
        torch.nn.Conv2d(c_in, c_out, k, padding=p, stride=s),
        torch.nn.Tanh(),
        torch.nn.AvgPool2d(2, stride=2)
    )

def block2(c_in, c_out):
    return torch.nn.Sequential(
        torch.nn.Linear(c_in, c_out),
        torch.nn.ReLU()
    )

class LeNet5(torch.nn.Module):
  def __init__(self, n_channels=1, n_outputs=10):
    super().__init__()
    #self.pad = torch.nn.ConstantPad2d(2, 0.)
    self.conv1 = block(n_channels, 6, 5, p=0)
    self.conv2 = block(6, 16, 5, p=0)
    self.conv3 = torch.nn.Sequential(
```

```
            torch.nn.Conv2d(16, 120, 5, padding=0),
            torch.nn.Tanh()
        )
        self.fc1 = block2(120, 84)
        self.fc2 = torch.nn.Linear(84, 10)

    def forward(self, x):
        #x = self.pad(x)
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.shape[0], -1)
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

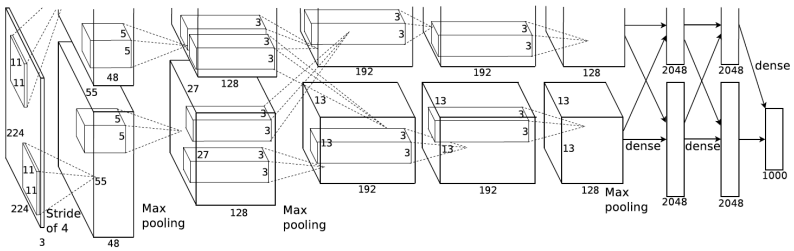
Python

```
lenet5 = LeNet5()
output = lenet5(torch.randn(64, 1, 32, 32))
output.shape
```

Devuelve: *torch.Size([64, 10])*

AlexNet

AlexNet ganó la competición de Imagenet en 2012. Es una arquitectura similar a LeNet5, pero más profunda. Fue la primera red convolucional que ganó la competición, y también la primera en ser entrenada en GPUs.



Tanto esta red como muchas otras las podemos encontrar ya implementadas en varios paquetes. En `Pytorch` podemos descargar redes convolucionales con el paquete `torchvision`.

Python

```
import torchvision

alexnet = torchvision.models.AlexNet()
alexnet
```

Devuelve: *AlexNet((features): Sequential((0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2)) (1): ReLU(inplace=True) (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False) (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2)) (4): ReLU(inplace=True) (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False) (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (7): ReLU(inplace=True) (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (9): ReLU(inplace=True) (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (11): ReLU(inplace=True) (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)) (avgpool): AdaptiveAvgPool2d(output_size=(6, 6)) (classifier): Sequential((0): Dropout(p=0.5, inplace=False) (1): Linear(in_features=9216, out_features=4096, bias=True) (2): ReLU(inplace=True) (3): Dropout(p=0.5, inplace=False) (4): Linear(in_features=4096, out_features=4096, bias=True) (5): ReLU(inplace=True) (6): Linear(in_features=4096, out_features=1000, bias=True)))*

Python

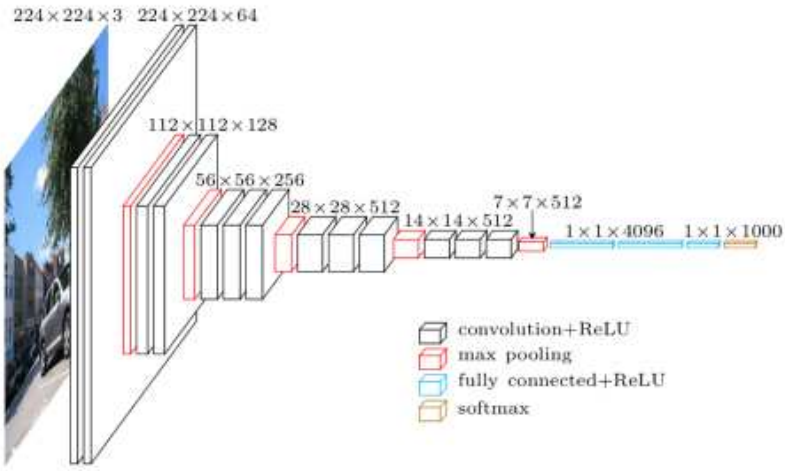
```
output = alexnet(torch.randn(64, 3, 224, 224))
output.shape
```

Devuelve: *torch.Size([64, 1000])*

> El dataset Imagenet contiene imágenes comunes descargadas de internet, y el objetivo es el de clasificar imágenes entre 1000 clases diferentes. Es por este motivo que verás que los diferentes modelos en `torchvision` tienen una última capa lineal con 1000 salidas.

VGG

Si bien VGG no ganó, tuvo mucho éxito ya que sentó un precedente en el patrón del diseño de las redes convolucionales. Este patrón consiste en usar siempre capas convolucionales con filtros de 3x3, *stride* y *padding* 1 (manteniendo el tamaño de las entradas constante) y usar *max pool* para reducir a la mitad los mapas de características. Además, después de cada capa se dobla el número de filtros.



Python

```
# existen dos variantes: vgg16 y vgg19, con 16 y 19 capas respectivamente

vgg16 = torchvision.models.vgg16()
vgg16
```

Devuelve: *VGG((features): Sequential((0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (1): ReLU(inplace=True) (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (3): ReLU(inplace=True) (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (6): ReLU(inplace=True) (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (8): ReLU(inplace=True) (9): MaxPool2d(kernel_size=2, stride=2,*

padding=0, dilation=1, ceil_mode=False) (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (11): ReLU(inplace=True) (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (13): ReLU(inplace=True) (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (15): ReLU(inplace=True) (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (18): ReLU(inplace=True) (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (20): ReLU(inplace=True) (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (22): ReLU(inplace=True) (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (25): ReLU(inplace=True) (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (27): ReLU(inplace=True) (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (29): ReLU(inplace=True) (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)) (avgpool): AdaptiveAvgPool2d(output_size=(7, 7)) (classifier): Sequential((0): Linear(in_features=25088, out_features=4096, bias=True) (1): ReLU(inplace=True) (2): Dropout(p=0.5, inplace=False) (3): Linear(in_features=4096, out_features=4096, bias=True) (4): ReLU(inplace=True) (5): Dropout(p=0.5, inplace=False) (6): Linear(in_features=4096, out_features=1000, bias=True)))

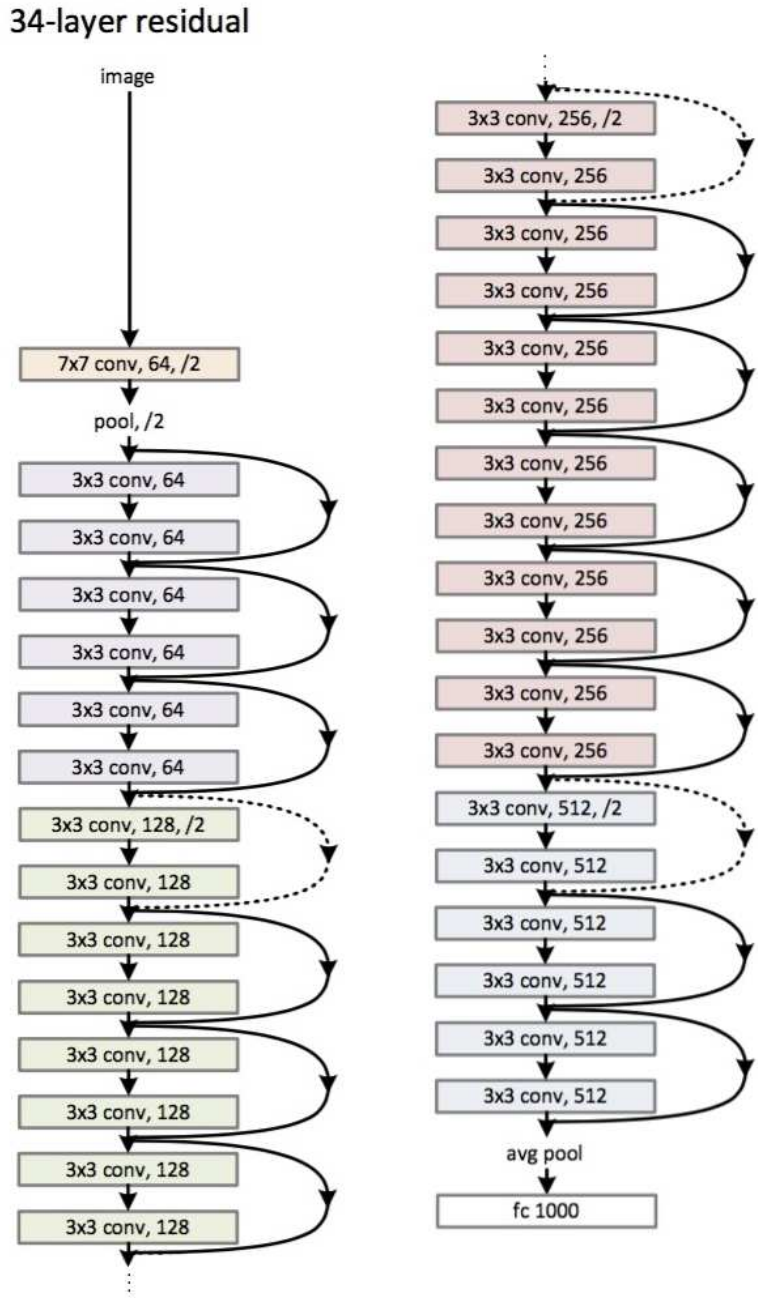
Python

```
output = vgg16(torch.randn(64, 3, 224, 224))
output.shape
```

Devuelve: *torch.Size([64, 1000])*

ResNet

ResNet fue el campeón en 2015. La variante ganadora tenía 152 capas y confirmó la tendencia de profundizar con menos parámetros. La clave para entrenar redes tan profundas es el uso de *skip connections* donde la señal que entra a una capa también se agrega a la salida. Esto proporciona una ruta limpia para que el gradiente se propague desde la salida a la entrada. También se puede interpretar como una forma de dejar a la red la decisión de qué capas usar (si alguna capa no es necesaria, simplemente se la puede saltar).


34-layer residual
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000

Python

```
# existen múltiples variantes: resnet18, resnet35, resnet50, resnet101,
    resnet152

resnet34 = torchvision.models.resnet34()
resnet34
```

Devuelve:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
      3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
      ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
```

```
        track_running_stats=True)
  )
)
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding
        =(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=
          False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
  (3): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
```

```
)
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=
          False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
  (3): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
  (4): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
```

```
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
  (5): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
)
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=
          False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
```

```
            track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
              padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
        )
        (3): BasicBlock(
          (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
              padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
              padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
        )
        (4): BasicBlock(
          (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
              padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
              padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
        )
        (5): BasicBlock(
          (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
              padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
              padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
        )
      )
      (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
      (fc): Linear(in_features=512, out_features=1000, bias=True)
    )
```

Python

```
output = resnet34(torch.randn(64, 3, 224, 224))
output.shape
```

Devuelve: *torch.Size([64, 1000])*

Otras arquitecturas

Existen una infinidad de arquitecturas de redes convolucionales, cada una con sus propias particularidades y ventajas. En el momento de escribir este libro algunas de las redes CNNs más potentes son EfficientNet y ConvNext, que puedes usarlas en `Pytorch` a través del paquete timm. Sin embargo, pese a la popularidad de estas arquitecturas, el rey indiscutible en el momento de escribir este libro es el `Tranformer`, una arquitectura que ha revolucionado el campo del `deep learning` y que ha demostrado ser muy eficiente no solo en tareas de visión artificial sino en otros campos como el procesamiento de lenguaje natural.

El Transformer

Desde que esta arquitectura de red neuronal fue introducida en 2017 en el artículo Attention is all you need nuevas aplicaciones aparecen cada día en diferentes campos, mejorando el estado del arte gracias al uso de `Transformers`. Si bien su mayor impacto se ha visto en el campo del procesado de lenguaje natural, su aplicación en otros dominios (como el de la visión artificial) no hace más que crecer. En esta sección aprenderemos qué es un `Transformer`, cómo funciona y cómo podemos implementarlo en `Pytorch`.

Attention is all you need

Como comentaba en el párrafo anterior, la arquitectura *Transformer* hizo su aparición en escena en 2017 gracias al artículo *Attention is all you need*. El *paper* empieza así...

> *The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely.*

Lo que traducido sería algo así como...

> *Los modelos secuenciales de transducción dominantes están basados en redes neuronales recurrentes o convolucionales complejas, las cuales incluyen un codificador y un decodificador. Los mejores modelos, además, conectan el codificador y el decodificador mediante un mecanismo de atención. Proponemos una nueva arquitectura simple, el Transformer, basado solamente en mecanismos de atención, haciendo innecesario el uso de recurrencia o convoluciones.*

A grandes rasgos, podemos ver que efectivamente se trata de una nueva arquitectura basada en algo llamado `mecanismos de atención`, y que no utiliza ni convoluciones ni recurrencia. Vamos ahora a entrar en detalle sobre lo que significa `atención` y cómo se implementa.

¿Qué es la atención?

Un mecanismo de atención, en el contexto de las redes neuronales, consiste en una operación matemática que recibe como *inputs* un conjunto de vectores (que ya sabemos que pueden representar texto, imágenes o cualquier tipo de datos con el que trabajemos) y nos da como resultado otro conjunto de vectores. Este resultado dependerá, obviamente, del tipo de mecanismo de atención que utilicemos. Vamos a ver algunos ejemplos.

Hard attention

El mecanismo de atención más sencillo de entender es el conocido como *hard attention mechanism*. En este tipo de atención generaremos un número de vectores a la salida igual a los de la entrada (de ahora en adelante asumiremos que este es siempre el caso, a no ser que se indique lo contrario) en el que cada *output* atenderá únicamente a su correspondiente vector a la entrada. O dicho de otra forma, este mecanismo de atención no produce nada nuevo, produce a la salida lo mismo que recibe a la entrada. Considera el siguiente conjunto de vectores:

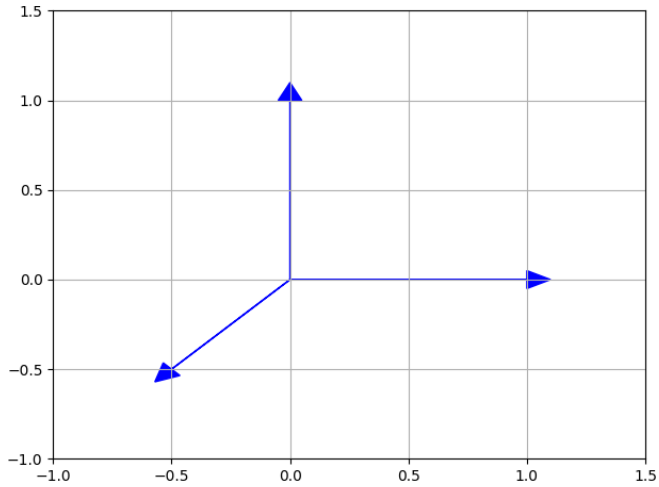
Python

```python
import numpy as np
import torch
import matplotlib.pyplot as plt
import numpy as np
```

Python

```
X = torch.tensor([[1, 0],[0, 1], [-0.5, -0.5]])
X
```

Devuelve: *tensor([[1.0000, 0.0000], [0.0000, 1.0000], [-0.5000, -0.5000]])*



Un mecaniso de *hard attention* prestará atención a un único vector. Podemos representarlo como un vector en el que, en cada posición, tenemos el peso relativo de cada vector a la entrada. En este caso, todos los valores serán 0, excepto el que se encuentre en la misma posición del vector al que queremos prestar atención.

Python

```
# hard attention (a es one hot)

# atendemos al primer vector
a = torch.tensor([1, 0, 0])
a
```

Devuelve: *tensor([1, 0, 0])*

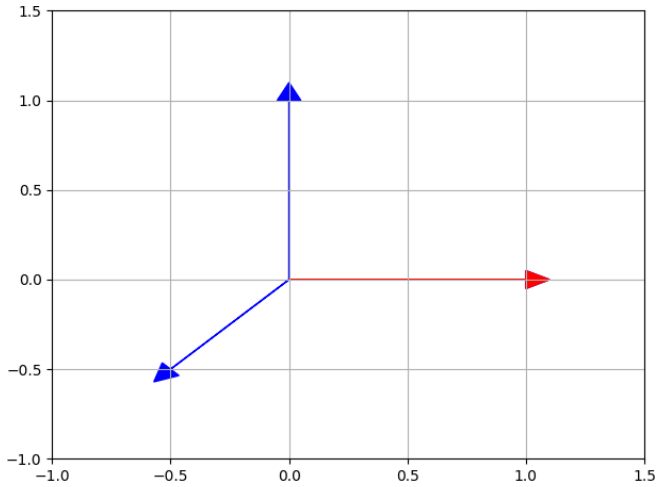
Para aplicar nuestro mecanismo de atención, simplemente multiplicamos nuestro conjunto de vectores por el vector de atención.

Python

```
# todos los vectores en la salida son 0, excepto al que hemos prestado
    atención

y = a.unsqueeze(1) * X
y
```

Devuelve: *tensor([[1.0000, 0.0000], [0.0000, 0.0000], [-0.0000, -0.0000]])*

Podemos ver en rojo la salida de nuestro mecanismo de atención fuerte cuando atendemos al primer vector (es, efectivamente, igual al primer vector). Hemos prestado atención únicamente a un elemento del conjunto, deshechando el resto.



Podemos aplicar este mecanismo en una sola operación a todos los vectores generando una matriz de atención. En el caso de *hard attention*, esta matriz es la identidad.

Python

```
A = torch.eye(3)
A
```

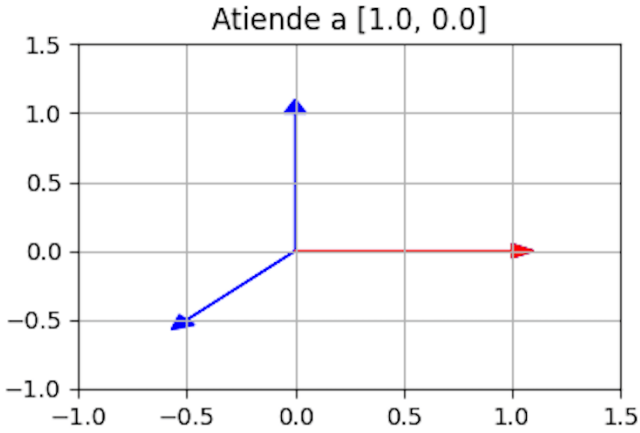
Devuelve: *tensor([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])*

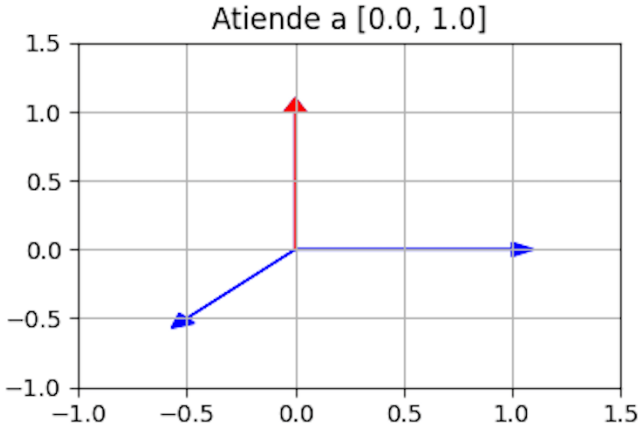
Multiplicando nuestra matriz de atención por la matriz que contiene todos los vectores de entrada, obtenemos los vectores de salida. En este caso, repetimos, obtendremos exactamente el mismo conjunto de vectores, ya que cada vector a la salida atiene únicamente a un vector a la entrada, aquel que está en su misma posición.

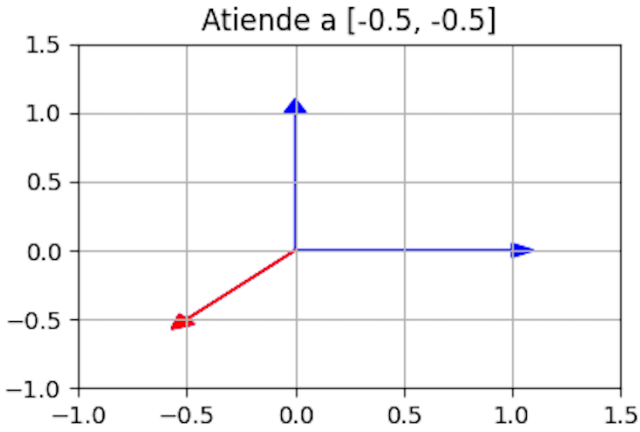
Python

```
Y = A @ X
Y
```

Devuelve: *tensor([[1.0000, 0.0000], [0.0000, 1.0000], [-0.5000, -0.5000])*







Soft attention

Ahora que hemos explicado qué es un mecanismo de atención y visto un ejemplo sencillo, vamos a ver otro tipo de mecanismo un poco más flexible. Si en el caso de la atención fuerte, cada vector generado presta atención simplemente a un único vector en la entrada, en el caso de la atención débil vamos a permitir prestar atención a todos los vectores a la entrada. Así pues, cada vector generado será una combinación de los *inputs*. En el siguiente ejemplo, cada vector generado presta un 80 % de atención al vector en la entrada en su misma posición y un 10 % al resto.

Python

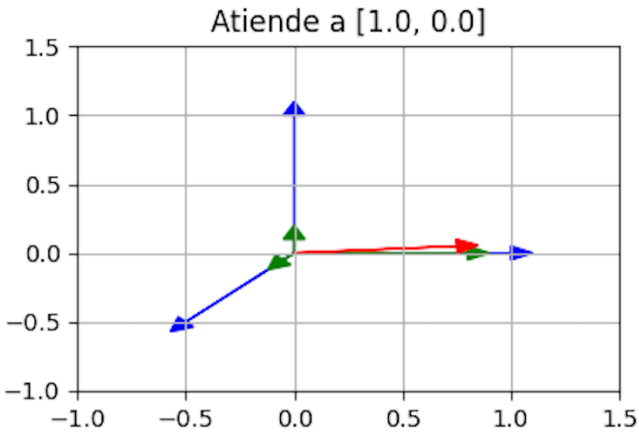
```
# soft attention (cada fila suma 1)

A = torch.ones((3, 3))*0.1
A.fill_diagonal_(0.8)
A
```

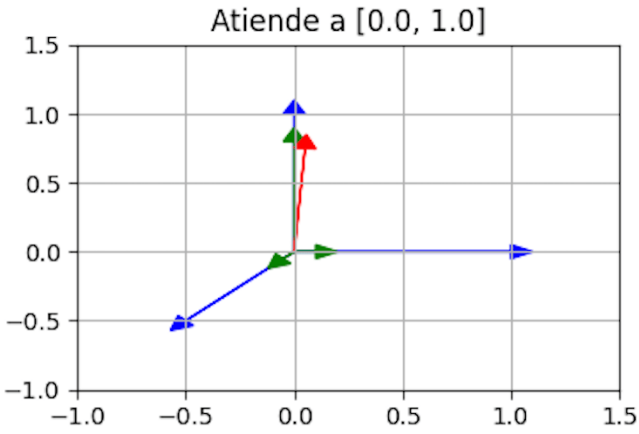
Devuelve: *tensor([[0.8000, 0.1000, 0.1000], [0.1000, 0.8000, 0.1000], [0.1000, 0.1000, 0.8000])*

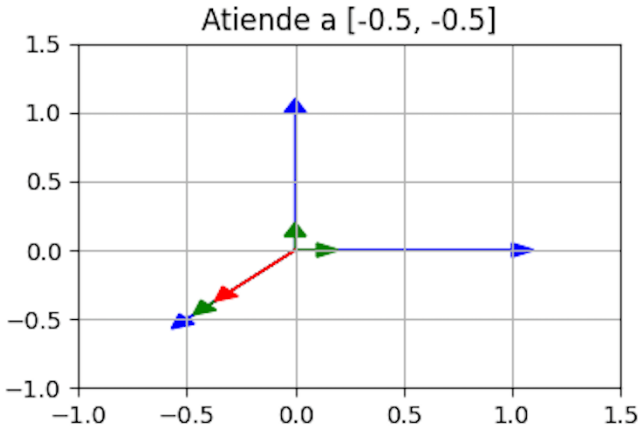
Python

```
Y = A @ X
Y
```

Devuelve: *tensor([[0.7500, 0.0500], [0.0500, 0.7500], [-0.3000, -0.3000])*







En verde puedes ver la contribución de cada vector al resultado, mientras que en rojo puedes ver los vectores generados por nuestro mecanismo de atención. En este caso, son muy similares a los originales, ya que estamos prestando mucha atención a estos mismos. Sin embargo, el resto de vectores pueden influir en el resultado. Vamos ahora a prestar igual atención a todos los vectores.

Python

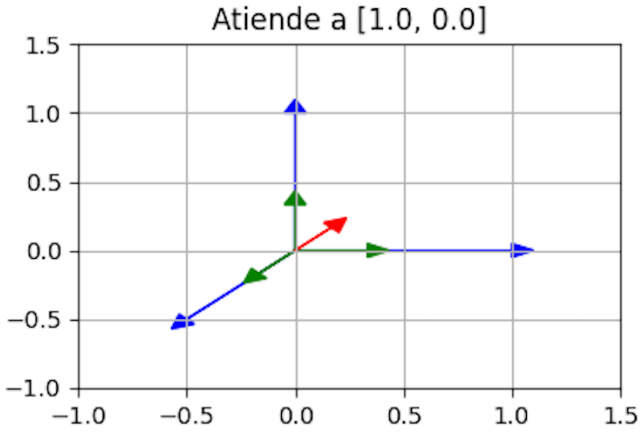
```
A = torch.ones((3, 3))*(1./3.)
A
```

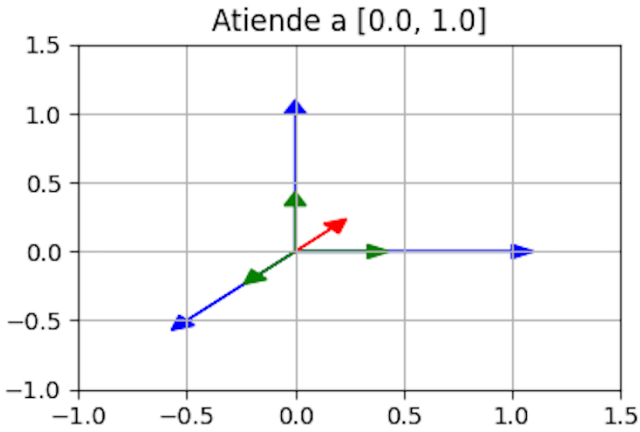
Devuelve: *tensor([[0.3333, 0.3333, 0.3333], [0.3333, 0.3333, 0.3333], [0.3333, 0.3333, 0.3333])*

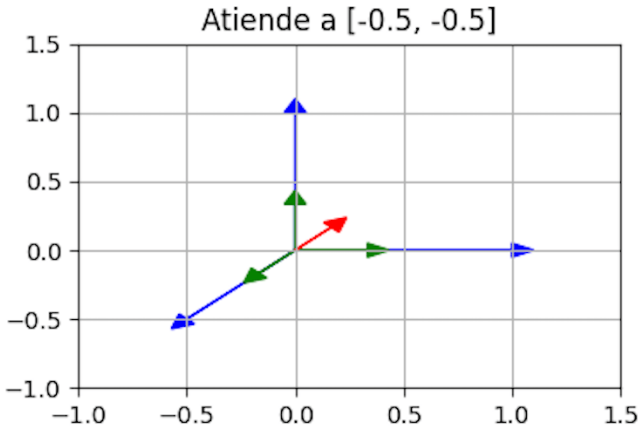
Python

```
Y = A @ X
Y
```

Devuelve: *tensor([[0.1667, 0.1667], [0.1667, 0.1667], [-0.1667, -0.1667])*


Atiende a [1.0, 0.0]
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5


Atiende a [0.0, 1.0]
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5



En este caso, todos los vectores generados son iguales. La pregunta ahora es, ¿cuánta atención debe un vector prestar al resto? La respuesta es sencilla… ¡*Self attention* !

Self attention

En este último mecanismo de atención que veremos, cada vector es responsable de decidir por sí mismo cuánta atención prestar al resto. Para ello, calcularemos la similitud entre vectores. Cuanto más parecidos sean dos vectores, más atención habrá entre ellos y viceversa. En una tarea de traducción de texto, por ejemplo, a la hora de generar una palabra un mecanismo de *self attention* permitiría prestar más atención a aquellas palabras más relacionadas a la entrada, y no desperdiciar computación con aquellas que no tienen importancia. Esta similitud la calculamos multiplicando los vectores por sí mismos y aplicando una función *softmax*.

Python

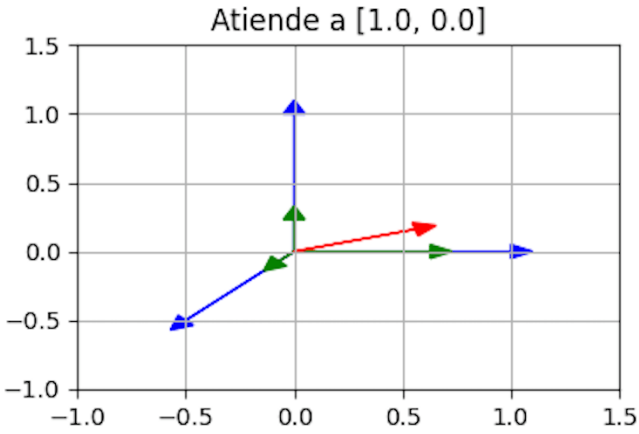
```
# self attention -> similitud de cada vector con el resto

A = torch.softmax(X @ X.T, 1)
A
```

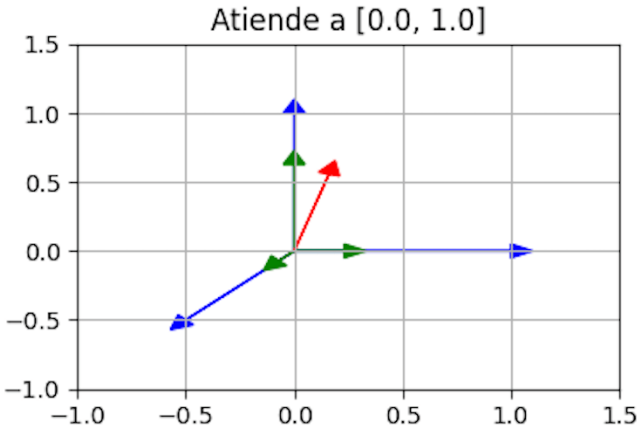
Devuelve: *tensor([[0.6285, 0.2312, 0.1402], [0.2312, 0.6285, 0.1402], [0.2119, 0.2119, 0.5761])*

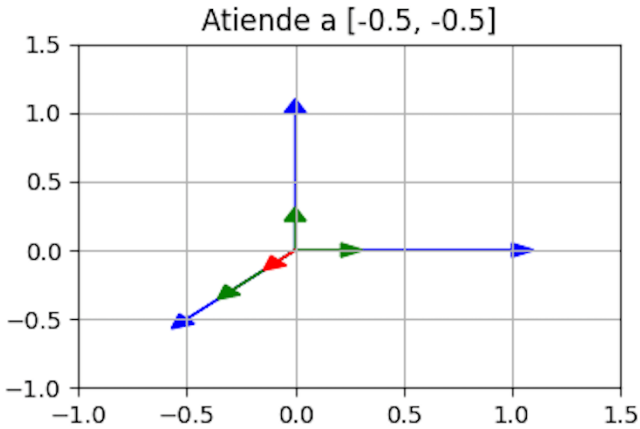
Python

```
Y = A @ X
Y
```

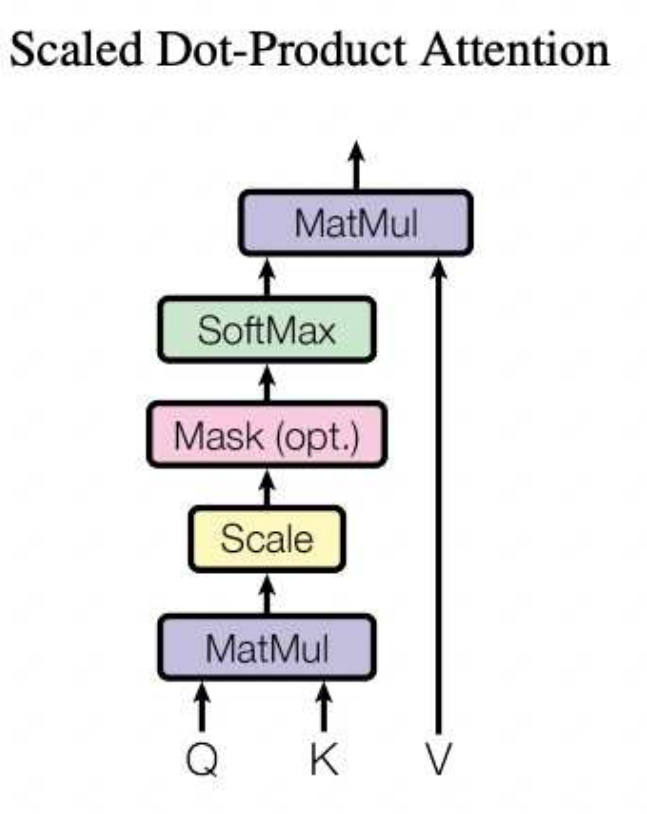
Devuelve: *tensor([[0.5584, 0.1611], [0.1611, 0.5584], [-0.0761, -0.0761])*







Este es el mecanismo de atención en el que se basa el `Transformer`, aunque introduce una variante conocida como el `scaled dot-product attention`.



Este mecanismo de atención consiste en tres conjuntos de vectores K, Q y V, llamados respectivamente *keys*, *queries* y *values*. Utilizaremos K y Q para calcular la matriz de atención, la cual aplicaremos a V.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Si te fijas, una formulación muy similar a la que hemos usado anteriormente, con la diferencia de que en este caso escalamos el producto de Q y K con la raíz cuadrada de su dimensión, d_k (detalles técnicos). ¿De dónde vienen estas Q, K y V? Aquí es donde entra en juego el componente de aprendizaje. En los ejemplos que hemos visto anteriormente hemos trabajado con conjuntos de vectores definidos por nosotros mismos, y hemos calculado matrices de atención basándonos en sus valores. Sin embargo, para que este

sistema sea capaz de aprender, calcularemos los diferentes vectores utilizando capas lineales: $Q = W_q X$, $K = W_k X$ y $V = W_v X$. De esta manera, nuestro sistema será capaz de calcular la mejor representación de X para obtener un alineamiento óptimo en el mecanismo de atención.

> ¿Qué obtenemos si utilizamos un mecanismo de *hard-attention* en esta formulación?
>
> $$Attention(Q, K, V) = 1V = W_v X$$
>
> Un mecanismo de atención, con *hard-attention*, no es más que un perceptrón de toda la vida. Esto significa que un perceptrón, hasta ahora visto como la unidad básica de cómputo en redes neuronales a partir de la cual se han desarrollado arquitecturas como redes recurrentes y convolucionales, es en realidad un caso particular de un mecanismo más general y potente. Esto abre un nuevo abanico de arquitectura, más allá del *transformer*, basado en mecanismos de atención. A grandes rasgos, puedes entender el mecanismo de atención como un perceptrón cuyos pesos dependen de los datos, en vez de ser siempre los mismos (resultado del entrenamiento).

Vamos a ver un ejemplo de implementación y caso de uso de este mecanismo de atención en la tarea de clasificación de imágenes con el dataset MNIST.

Python

```python
from sklearn.datasets import fetch_openml
import numpy as np

mnist = fetch_openml('mnist_784', version=1)
X, Y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

Python

```
class Dataset(torch.utils.data.Dataset):
  def __init__(self, X, y, patch_size=(7, 7)):
    self.X = X
    self.y = y
    self.patch_size = patch_size

  def __len__(self):
    return len(self.X)

  def __getitem__(self, ix):
    image = torch.from_numpy(self.X[ix]).float().view(28, 28) # 28 x 28
    h, w = self.patch_size
    patches = image.unfold(0, h, h).unfold(1, w, w) # 4 x 4 x 7 x 7
    patches = patches.contiguous().view(-1, h*w) # 16 x 49
    return patches, torch.tensor(self.y[ix]).long()
```

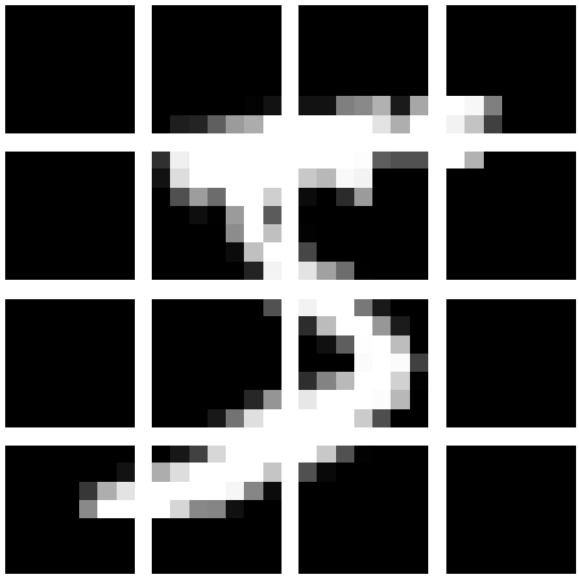
Python

```
attn_dm = Dataset(X, Y)
imgs, labels = attn_dm[0]
imgs.shape, labels.shape
```

Devuelve: *(torch.Size([16, 49]), torch.Size([]))*

Python

```
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(5,5))
for i in range(4):
    for j in range(4):
        ax = plt.subplot(4, 4, i*4 + j + 1)
        ax.imshow(imgs[i*4 + j].view(7, 7), cmap="gray")
        ax.axis('off')
plt.tight_layout()
plt.savefig('resources/attn_patches.png')
plt.close()
```



Para ello separamos las imágenes en *patches* de 7x7 que usaremos como vectores de entrada para el mecanismo de atención.

Python

```python
# basado en: https://github.com/karpathy/minGPT/blob/master/mingpt/model.
    py

import math

class ScaledDotSelfAttention(torch.nn.Module):

    def __init__(self, n_embd):
        super().__init__()

        # key, query, value projections
        self.key = torch.nn.Linear(n_embd, n_embd)
        self.query = torch.nn.Linear(n_embd, n_embd)
        self.value = torch.nn.Linear(n_embd, n_embd)

    def forward(self, x):
        B, L, F = x.size()

        # calculate query, key, values
        k = self.key(x) # (B, L, F)
        q = self.query(x) # (B, L, F)
        v = self.value(x) # (B, L, F)
```

```
            # attention (B, L, F) x (B, F, L) -> (B, L, L)
            att = (q @ k.transpose(1, 2)) * (1.0 / math.sqrt(k.size(-1)))
            att = torch.nn.functional.softmax(att, dim=-1)
            y = att @ v # (B, L, L) x (B, L, F) -> (B, L, F)

            return y

    class Model(torch.nn.Module):

        def __init__(self, n_embd=7*7, seq_len=4*4):
            super().__init__()
            self.attn = ScaledDotSelfAttention(n_embd)
            self.actn = torch.nn.ReLU(inplace=True)
            self.fc = torch.nn.Linear(n_embd*seq_len, 10)

        def forward(self, x):
            x = self.attn(x)
            y = self.fc(self.actn(x.view(x.size(0), -1)))
            return y
```

Nuestro modelo basado en atención tiene muchos menos parámetros. Esto es debido a que en el MLP todas las neuronas en la capa oculta están conectadas a todos los píxeles de la imagen. Ahora, sin embargo, reutilizamos conexiones a nivel de *patch*, de manera similar a cómo funcionan las redes convolucionales.

Python

```
    model = Model()

    train(model)
```

Devuelve:

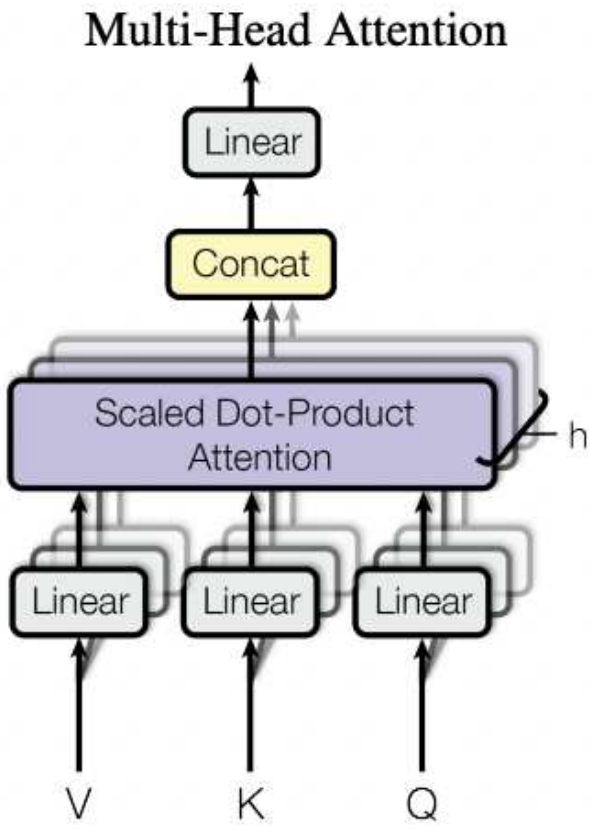
epoch: 1/5

loss: 43.0019 [1000/60000] loss: 11.2572 [11000/60000] loss: 5.9748 [21000/60000] loss: 4.1359 [31000/60000] loss: 3.1597 [41000/60000] loss: 2.6025 [51000/60000] val_loss: 2.1969 val_acc: 0.5880 epoch: 2/5 loss: 2.1482 [1000/60000] loss: 1.9276 [11000/60000] loss: 1.6801 [21000/60000] loss: 1.5558 [31000/60000] loss: 1.5691 [41000/60000] loss: 1.5915 [51000/60000] val_loss: 1.4548 val_acc: 0.6477 epoch: 3/5 loss: 1.3585 [1000/60000] loss: 1.1923 [11000/60000] loss: 1.2928 [21000/60000] loss: 1.2913 [31000/60000] loss: 1.1982 [41000/60000] loss: 1.1966 [51000/60000] val_loss: 1.1661 val_acc: 0.6891 epoch: 4/5 loss: 1.0600 [1000/60000] loss: 1.0634 [11000/60000] loss: 1.1305 [21000/60000] loss: 0.9593 [31000/60000]

loss: 1.0436 [41000/60000] loss: 0.9852 [51000/60000] val_loss: 1.0172 val_acc: 0.7078 epoch: 5/5 loss: 0.8851 [1000/60000] loss: 0.9909 [11000/60000] loss: 0.8428 [21000/60000] loss: 0.8948 [31000/60000] loss: 0.9575 [41000/60000] loss: 0.9572 [51000/60000] val_loss: 0.9131 val_acc: 0.7329

Multi-head Attention

La capacidad de representación del mecanismo de atención implementado es limitada. Para resolver este problema, los autores de *Attention is all you need* proponen una mejora conocida como *Multi-head attention*.



Este mecanismo toma inspiración en el uso de múltiples filtros en una red convolucional

para mejorar la capacidad de representación de datos. En el contexto de atención, esto se traduce en repetir un número determinado de veces (*heads* o cabezas) el mecanismo de *scaled-dot product attention* que conocemos del *post* anterior.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^o$$

Donde:

$$head_i = Attention(QW_q^i, KW_k^i, VW_v^i)$$

A grandes rasgos, repetimos el mecanismo de atención aplicando diferentes proyecciones a la hora de obtener nuestras *queries*, *keys* y *values*. Una vez aplicada la atención a cada cabeza, concatenamos los resultados y aplicamos una nueva capa lineal para obtener el resultado final.

Vamos a ver esta implementación en el mismo ejemplo anterior.

Python

```python
# basado en: https://github.com/karpathy/minGPT/blob/master/mingpt/model.
    py

import math

class MultiHeadAttention(torch.nn.Module):

    def __init__(self, n_embd, n_heads):
        super().__init__()
        self.n_heads = n_heads

        # key, query, value projections
        self.key = torch.nn.Linear(n_embd, n_embd*n_heads)
        self.query = torch.nn.Linear(n_embd, n_embd*n_heads)
        self.value = torch.nn.Linear(n_embd, n_embd*n_heads)

        # output projection
        self.proj = torch.nn.Linear(n_embd*n_heads, n_embd)

    def forward(self, x):
        B, L, F = x.size()

        # calculate query, key, values for all heads in batch and move
            head forward to be the batch dim
        k = self.key(x).view(B, L, F, self.n_heads).transpose(1, 3) # (B,
             nh, L, F)
```

```
            q = self.query(x).view(B, L, F, self.n_heads).transpose(1, 3) # (
                B, nh, L, F)
            v = self.value(x).view(B, L, F, self.n_heads).transpose(1, 3) # (
                B, nh, L, F)

            # attention (B, nh, L, F) x (B, nh, F, L) -> (B, nh, L, L)
            att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
            att = torch.nn.functional.softmax(att, dim=-1)
            y = att @ v # (B, nh, L, L) x (B, nh, L, F) -> (B, nh, L, F)
            y = y.transpose(1, 2).contiguous().view(B, L, F*self.n_heads) #
                re-assemble all head outputs side by side

            return self.proj(y)

    class Model(torch.nn.Module):

        def __init__(self, n_embd=7*7, seq_len=4*4, n_heads=4*4):
            super().__init__()
            self.attn = MultiHeadAttention(n_embd, n_heads)
            self.actn = torch.nn.ReLU(inplace=True)
            self.fc = torch.nn.Linear(n_embd*seq_len, 10)

        def forward(self, x):
            x = self.attn(x)
            y = self.fc(self.actn(x.view(x.size(0), -1)))
            return y
```

Python

```
    model = Model()

    train(model)
```

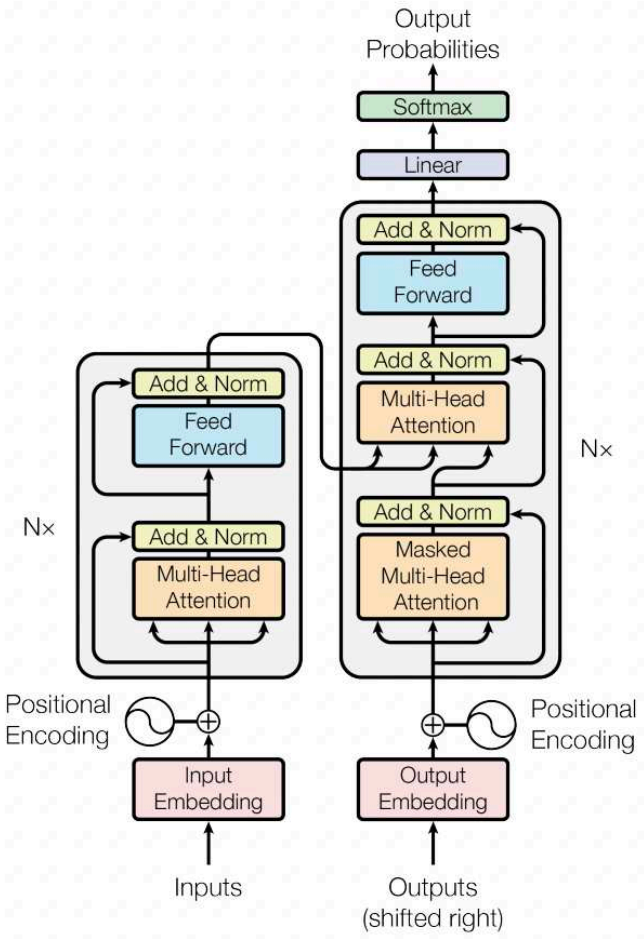
Devuelve:

epoch: 1/5 loss: 19.9184 [1000/60000] loss: 3.1439 [11000/60000] loss: 1.5333 [21000/60000] loss: 1.1029 [31000/60000] loss: 0.9131 [41000/60000] loss: 0.6673 [51000/60000] val_loss: 0.5506 val_acc: 0.8532 epoch: 2/5 loss: 0.4652 [1000/60000] loss: 0.4102 [11000/60000] loss: 0.4313 [21000/60000] loss: 0.4003 [31000/60000] loss: 0.2878 [41000/60000] loss: 0.3517 [51000/60000] val_loss: 0.3538 val_acc: 0.9022 epoch: 3/5 loss: 0.3783 [1000/60000] loss: 0.2623 [11000/60000] loss: 0.2535 [21000/60000] loss: 0.2704 [31000/60000] loss: 0.2309 [41000/60000] loss: 0.2742 [51000/60000] val_loss: 0.2809 val_acc: 0.9218 epoch: 4/5 loss: 0.1665 [1000/60000] loss: 0.1707 [11000/60000] loss: 0.1813 [21000/60000] loss: 0.1900 [31000/60000] loss: 0.1898 [41000/60000] loss: 0.1753 [51000/60000] val_loss: 0.2421 val_acc: 0.9332 epoch: 5/5 loss: 0.1295 [1000/60000] loss: 0.2090

[11000/60000] loss: 0.1384 [21000/60000] loss: 0.1359 [31000/60000] loss: 0.1697 [41000/60000] loss: 0.1545 [51000/60000] val_loss: 0.2189 val_acc: 0.9353

Transformers

Una vez entendido el mecanismo de atención, podemos entrar en la arquitectura completa del `Transformer`.



Como puedes ver el modelo tiene dos partes bien diferenciadas: un *encoder* y un *decoder*.

El *encoder* se encarga de procesar la entrada y generar una representación de la misma, mientras que el *decoder* se encarga de generar la salida a partir de esta representación. Ambos están formados por capas de *multi-head attention* seguidas de capas *feed-forward* (MLPs) y capas de normalización. La salida del *encoder* es la entrada del *decoder*, y la salida del *decoder* es la predicción final. Si bien esta es la formulación original, es perfectamente posible utilizar solo el *encoder* o el *decoder*. Modelos como GPT-4 están basados únicamente en el *decoder*, otros modelos como `Vision Transformer` están basados en el *encoder*.

Transformer Encoder

Python

```python
# basado en: https://github.com/karpathy/minGPT/blob/master/mingpt/model.
    py

import math

class MultiHeadAttention(torch.nn.Module):

    def __init__(self, n_embd, n_heads):
        super().__init__()
        self.n_heads = n_heads

        # key, query, value projections
        self.key = torch.nn.Linear(n_embd, n_embd*n_heads)
        self.query = torch.nn.Linear(n_embd, n_embd*n_heads)
        self.value = torch.nn.Linear(n_embd, n_embd*n_heads)

        # output projection
        self.proj = torch.nn.Linear(n_embd*n_heads, n_embd)

    def forward(self, x):
        B, L, F = x.size()

        # calculate query, key, values for all heads in batch and move
            head forward to be the batch dim
        k = self.key(x).view(B, L, F, self.n_heads).transpose(1, 3) # (B,
             nh, L, F)
        q = self.query(x).view(B, L, F, self.n_heads).transpose(1, 3) # (
            B, nh, L, F)
        v = self.value(x).view(B, L, F, self.n_heads).transpose(1, 3) # (
            B, nh, L, F)

        # attention (B, nh, L, F) x (B, nh, F, L) -> (B, nh, L, L)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = torch.nn.functional.softmax(att, dim=-1)
```

```
            y = att @ v # (B, nh, L, L) x (B, nh, L, F) -> (B, nh, L, F)
            y = y.transpose(1, 2).contiguous().view(B, L, F*self.n_heads) #
                re-assemble all head outputs side by side

            return self.proj(y)

    class TransformerBlock(torch.nn.Module):
        def __init__(self, n_embd, n_heads):
            super().__init__()
            self.ln1 = torch.nn.LayerNorm(n_embd)
            self.ln2 = torch.nn.LayerNorm(n_embd)
            self.attn = MultiHeadAttention(n_embd, n_heads)
            self.mlp = torch.nn.Sequential(
                torch.nn.Linear(n_embd, 4 * n_embd),
                torch.nn.ReLU(),
                torch.nn.Linear(4 * n_embd, n_embd),
            )

        def forward(self, x):
            x = self.ln1(x + self.attn(x))
            x = self.ln2(x + self.mlp(x))
            return x

    class Model(torch.nn.Module):

        def __init__(self, n_input=7*7, n_embd=7*7, seq_len=4*4, n_heads=4*4,
             n_layers=1):
            super().__init__()
            self.pos_emb = torch.nn.Parameter(torch.zeros(1, seq_len, n_embd)
                )
            self.inp_emb = torch.nn.Linear(n_input, n_embd)
            self.tranformer = torch.nn.Sequential(*[TransformerBlock(n_embd,
                n_heads) for _ in range(n_layers)])
            self.fc = torch.nn.Linear(n_embd*seq_len, 10)

        def forward(self, x):
            # embedding
            e = self.inp_emb(x) + self.pos_emb
            # transformer blocks
            x = self.tranformer(e)
            # classifier
            y = self.fc(x.view(x.size(0), -1))
            return y
```

Python

```
model = Model()

train(model)
```

Devuelve:

epoch: 1/5 loss: 2.5067 [1000/60000] loss: 0.7082 [11000/60000] loss: 0.4032 [21000/60000] loss: 0.3576 [31000/60000] loss: 0.2820 [41000/60000] loss: 0.2612 [51000/60000] val_loss: 0.2208 val_acc: 0.9328 epoch: 2/5 loss: 0.2287 [1000/60000] loss: 0.1764 [11000/60000] loss: 0.2463 [21000/60000] loss: 0.2214 [31000/60000] loss: 0.2052 [41000/60000] loss: 0.1439 [51000/60000] val_loss: 0.1612 val_acc: 0.9530 epoch: 3/5 loss: 0.1845 [1000/60000] loss: 0.1377 [11000/60000] loss: 0.1562 [21000/60000] loss: 0.1302 [31000/60000] loss: 0.1433 [41000/60000] loss: 0.1399 [51000/60000] val_loss: 0.1392 val_acc: 0.9580 epoch: 4/5 loss: 0.1449 [1000/60000] loss: 0.1330 [11000/60000] loss: 0.1029 [21000/60000] loss: 0.1298 [31000/60000] loss: 0.1134 [41000/60000] loss: 0.1276 [51000/60000] val_loss: 0.1144 val_acc: 0.9645 epoch: 5/5 loss: 0.0797 [1000/60000] loss: 0.0934 [11000/60000] loss: 0.1074 [21000/60000] loss: 0.0780 [31000/60000] loss: 0.1364 [41000/60000] loss: 0.1065 [51000/60000] val_loss: 0.1052 val_acc: 0.9670

Encoder-Decoder

Para este ejemplo vamos a cambiar ligeramente nuestra tarea, y en vez de clasificar los dígito de MNIST en una de las diez clases posibles, queremos que nuestro modelo produzca a la salida el nombre del dígito en texto. Para ello convertiremos nuestras clases (dígitos del 0 al 9) en vectores de números enteros, en el que cada valor en el vector corresponde al índice de la letra correspondiente en el vocabulario (que a su vez contiene todas las letras de la a a la z).

Python

```python
from sklearn.datasets import fetch_openml
import numpy as np

mnist = fetch_openml('mnist_784', version=1)
X, Y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

Python

```
vocab = 'abcdefghijklmnopqrstuvwxyz'
len_vocab = len(vocab) + 3

def number2caption(ix):
    if ix == 0: return 'cero'
    if ix == 1: return 'uno'
    if ix == 2: return 'dos'
    if ix == 3: return 'tres'
    if ix == 4: return 'cuatro'
    if ix == 5: return 'cinco'
    if ix == 6: return 'seis'
    if ix == 7: return 'siete'
    if ix == 8: return 'ocho'
    if ix == 9: return 'nueve'

def caption2ixs(caption):
    return [vocab.index(c) + 3 for c in caption]

def ixs2caption(ixs):
    return (''.join([vocab[ix - 3] for ix in ixs if ix not in [0, 1,
        2]]))
```

Al codificar los nombres añadimos también dos *tokens* especiales para indicar el inicio y final del nombre, 1 y 2 respectivamente. El *token* 0 lo reservamos para más adelante. Es por este motivo que en las funciones `caption2ixs` y `ixs2caption` sumamos o restamos 3 a los índices.

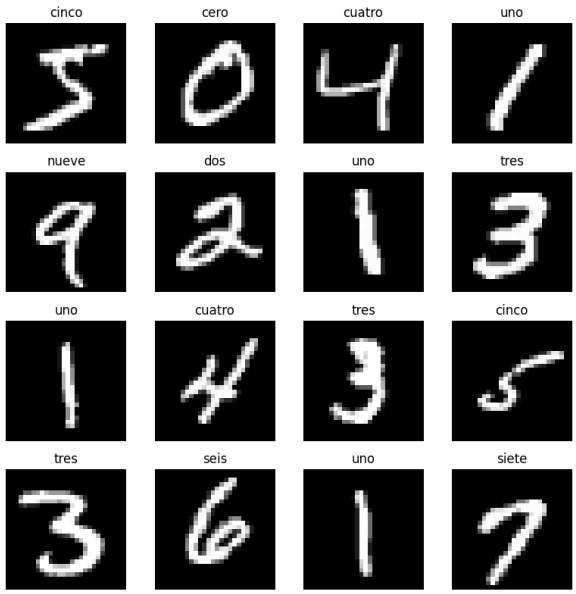
Python

```
captions = [number2caption(ix) for ix in Y]


# cada letra tiene su número (índice en el vocab)
encoded = [[1] + caption2ixs(caption) + [2] for caption in captions]

captions[:3], encoded[:3]
```

Devuelve: *(['cinco', 'cero', 'cuatro'], [[1, 5, 11, 16, 5, 17, 2], [1, 5, 7, 20, 17, 2], [1, 5, 23, 3, 22, 20, 17, 2]])*

Python

```python
import matplotlib.pyplot as plt

r, c = 4, 4
fig = plt.figure(figsize=(c*2, r*2))
for _r in range(r):
    for _c in range(c):
        ix = _r*c + _c
        ax = plt.subplot(r, c, ix + 1)
        img, caption = X[ix], captions[ix]
        ax.axis("off")
        ax.imshow(img.reshape(28,28), cmap="gray")
        ax.set_title(caption)
plt.tight_layout()
plt.savefig('resources/transformer_patches.png')
plt.close()
```



Para el modelo usaremos la implementación de *Transformer* que nos ofrece Pytorch. Esta implementación ya incluye un *encoder* y *decoder*, por lo que podremos trabajar con ella de manera muy sencilla. El único detalle que tenemos que tener en cuenta es el procesado de las imágenes y el texto a la entrada del *transformer*. Para las imágenes optamos por la misma solución que vimos anteriormente (separar en *patches*).

Python

```
import torch

class Dataset(torch.utils.data.Dataset):
  def __init__(self, X, y, patch_size=(7, 7)):
    self.X = X
    self.y = y

  def __len__(self):
    return len(self.X)

  def __getitem__(self, ix):
    image = torch.from_numpy(self.X[ix]).float().view(1, 28, 28)
    return image, torch.tensor(self.y[ix]).long()
```

Python

```
import torch.nn as nn


# https://github.com/jankrepl/mildlyoverfitted/blob/master/
    github_adventures/vision_transformer/custom.py

class PatchEmbedding(nn.Module):
    def __init__(self, img_size, patch_size, in_chans, embed_dim):
        super().__init__()
        self.img_size = img_size
        self.patch_size = patch_size
        self.n_patches = (img_size // patch_size) ** 2
        self.patch_size = patch_size
        self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size
            , stride=patch_size)

    def forward(self, x):
        x = self.proj(x)  # (B, E, P, P)
        x = x.flatten(2)  # (B, E, N)
        x = x.transpose(1, 2)  # (B, N, E)
        return x
```

En cuanto al *embedding* del texto, utilizaremos la capa de `embedding` que `Pytorch` nos ofrece. Lo mismo haremos para el *positional embedding* del texto en el *decoder*. Finalmente, usaremos un MLP para obtener una distribución de probabilidad sobre el conjunto del vocabulario para cada salida del *decoder*. Estos valores los compararemos con el *ground truth* para construir nuestra *loss function* y entrenar el modelo completo. Un aspecto importante es el hecho de que todo esto ocurre en un solo paso. Este es el principal motivo que hace que los *Transformers* sean más eficientes que alternativas como las redes neuronales recurrentes. Por contra, debemos asegurarnos de que, en el generador de texto, en cada paso el modelo solo pueda atender a los *tokens* pasados. Esto lo conseguimos con una

máscara causal (matriz triangular de ceros y unos) que aplicamos a la matriz de atención.

A la hora de predecir nuevos valores, utilizaremos un sistema autoregresivo, añadiendo la salida en cada paso a los *inputs* (que siempre empiezan con un *token* especial que indica el inicio de la frase, *start of sentence* o *SOS*) hasta que el modelo dé a la salida el valor del *token* que indica el final de la frase (*token end of sentence* o *EOS*) o lleguemos a una longitud de secuencia máxima (para evitar que nuestro generador se quede atrapado en un bucle infinito).

Python

```
    import torch.nn.functional as F

    class Model(torch.nn.Module):

        def __init__(self,
                     len_vocab,
                     img_size=28,
                     patch_size=7,
                     in_chans=1,
                     embed_dim=100,
                     max_len=8,
                     nhead=2,
                     num_encoder_layers=3,
                     num_decoder_layers=3,
                     dim_feedforward=400,
                     dropout=0.1
                    ):
            super().__init__()

            self.patch_embed = PatchEmbedding(img_size, patch_size, in_chans,
                 embed_dim)
            self.pos_embed = nn.Parameter(torch.zeros(1, self.patch_embed.
                n_patches, embed_dim))

            self.trg_emb = nn.Embedding(len_vocab, embed_dim)
            self.trg_pos_emb = nn.Embedding(max_len, embed_dim)
            self.max_len = max_len

            self.transformer = torch.nn.Transformer(
                embed_dim, nhead, num_encoder_layers, num_decoder_layers,
                    dim_feedforward, dropout
            )

            self.l = nn.LayerNorm(embed_dim)
            self.fc = nn.Linear(embed_dim, len_vocab)

        def forward(self, images, captions):
            # embed images
            embed_imgs = self.patch_embed(images)
            embed_imgs = embed_imgs + self.pos_embed  # (B, N, E)
```

```
        # embed captions
        B, trg_seq_len = captions.shape
        trg_positions = (torch.arange(0, trg_seq_len).expand(B,
            trg_seq_len).to(images.device))
        embed_trg = self.trg_emb(captions) + self.trg_pos_emb(
            trg_positions)
        trg_mask = self.transformer.generate_square_subsequent_mask(
            trg_seq_len).to(images.device)
        tgt_padding_mask = captions == 0
        # transformer
        y = self.transformer(
            embed_imgs.permute(1,0,2),  # S, B, E
            embed_trg.permute(1,0,2),  # T, B, E
            tgt_mask=trg_mask, # T, T
            tgt_key_padding_mask = tgt_padding_mask
        ).permute(1,0,2) # B, T, E
        # head
        return self.fc(self.l(y))

    def predict(self, image, device):
        self.eval()
        self.to(device)
        with torch.no_grad():
            image = image.to(device)
            B = 1
            # start of sentence
            eos = torch.tensor([1], dtype=torch.long, device=device).
                expand(B, 1)
            trg_input = eos
            for _ in range(self.max_len):
                preds = self(image.unsqueeze(0), trg_input)
                preds = torch.argmax(preds, axis=2)
                trg_input = torch.cat([eos, preds], 1)
            return preds
```

Diferentes nombres tienen diferentes longitudes, así que para poder crear *batches* necesitaremos rellenar los nombres más cortos con el *token PAD*, que suele tener el valor 0. Es habitual definir una longitud de secuencia máxima y simplemente rellenar todas las muestras con menor longitud.

Python

```
MAX_LEN = 8

def collate_fn(batch):
    images, captions = zip(*batch)
    images = torch.stack(images)
    captions = [torch.nn.functional.pad(caption, (0, MAX_LEN - len(
        caption)), value=0) for caption in captions]
    captions = torch.stack(captions)
    return images, captions
```

Python

```python
def train(model, epochs=5, batch_size=1000):
    dataset = {
        "train": Dataset(X[:60000], encoded[:60000]), # 60.000 imágenes
            para entrenamiento
        "val": Dataset(X[60000:], encoded[60000:])    # 10.000 imágenes
            para validación
    }
    dataloader = {
        'train': torch.utils.data.DataLoader(dataset['train'], batch_size
            =batch_size, shuffle=True, num_workers=4, pin_memory=True,
            collate_fn=collate_fn),
        'val': torch.utils.data.DataLoader(dataset['val'], batch_size=
            batch_size, num_workers=4, pin_memory=True, collate_fn=
            collate_fn)
    }
    model.cuda()
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters())
    for e in range(1, epochs+1):
        print(f"epoch: {e}/{epochs}")
        # entrenamiento
        model.train()
        for batch_ix, (x, y) in enumerate(dataloader['train']):
            x, y = x.cuda(), y.cuda()
            optimizer.zero_grad()
            outputs = model(x, y[:,:-1])
            loss = criterion(outputs.permute(0,2,1), y[:,1:])
            loss.backward()
            optimizer.step()
            if batch_ix % 10 == 0:
                loss, current = loss.item(), (batch_ix + 1) * len(x)
                print(f"loss: {loss:.4f} [{current:>5d}/{len(dataset['
                    train']):>5d}]")
        # validación
        model.eval()
        val_loss, val_acc = [], []
        with torch.no_grad():
            for batch_ix, (x, y) in enumerate(dataloader['val']):
                x, y = x.cuda(), y.cuda()
                outputs = model(x, y[:,:-1])
                loss = criterion(outputs.permute(0,2,1), y[:,1:])
                val_loss.append(loss.item())
                acc = (torch.argmax(outputs, axis=2) == y[:,1:]).sum().
                    item() / (y[:,1:].shape[0]*y[:,1:].shape[1])
                val_acc.append(acc)
        print(f"val_loss: {np.mean(val_loss):.4f} val_acc: {np.mean(
            val_acc):.4f}")
```

Python

```
    model = Model(len(vocab) + 3, max_len=MAX_LEN)

    train(model)
```

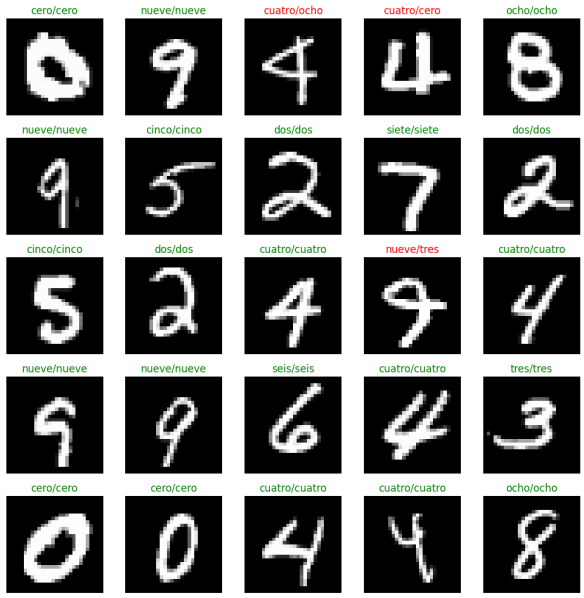
Devuelve:

epoch: 1/5

loss: 3.6873 [1000/60000] loss: 1.4449 [11000/60000] loss: 0.7053 [21000/60000] loss: 0.4560 [31000/60000] loss: 0.3702 [41000/60000] loss: 0.3264 [51000/60000] val_loss: 0.2598 val_acc: 0.9023 epoch: 2/5 loss: 0.2835 [1000/60000] loss: 0.2569 [11000/60000] loss: 0.2381 [21000/60000] loss: 0.2046 [31000/60000] loss: 0.1964 [41000/60000] loss: 0.1780 [51000/60000] val_loss: 0.1413 val_acc: 0.9511 epoch: 3/5 loss: 0.1667 [1000/60000] loss: 0.1491 [11000/60000] loss: 0.1538 [21000/60000] loss: 0.1438 [31000/60000] loss: 0.1167 [41000/60000] loss: 0.1162 [51000/60000] val_loss: 0.0923 val_acc: 0.9697 epoch: 4/5 loss: 0.1170 [1000/60000] loss: 0.1159 [11000/60000] loss: 0.1034 [21000/60000] loss: 0.1055 [31000/60000] loss: 0.0901 [41000/60000] loss: 0.0949 [51000/60000] val_loss: 0.0725 val_acc: 0.9760 epoch: 5/5 loss: 0.0944 [1000/60000] loss: 0.0929 [11000/60000] loss: 0.0842 [21000/60000] loss: 0.0917 [31000/60000] loss: 0.0696 [41000/60000] loss: 0.0695 [51000/60000] val_loss: 0.0599 val_acc: 0.9807

Python

```python
import random

r, c = 5,5
fig = plt.figure(figsize=(c*2, r*2))
for _r in range(r):
    for _c in range(c):
        # ix = _r*c + _c
        ix = random.randint(0, len(X))
        ax = plt.subplot(r, c, _r*c + _c + 1)
        img, caption = torch.from_numpy(X[ix]).float().view(1, 28, 28),
            captions[ix]
        ax.axis("off")
        ax.imshow(img.reshape(28,28), cmap="gray")
        pred = model.predict(img, device='cuda')
        pred = ixs2caption(pred.squeeze().tolist())
        ax.set_title(f'{caption}/{pred}', color="green" if caption ==
            pred else 'red')
plt.tight_layout()
plt.savefig('resources/transformer_predictions.png')
plt.close()
```



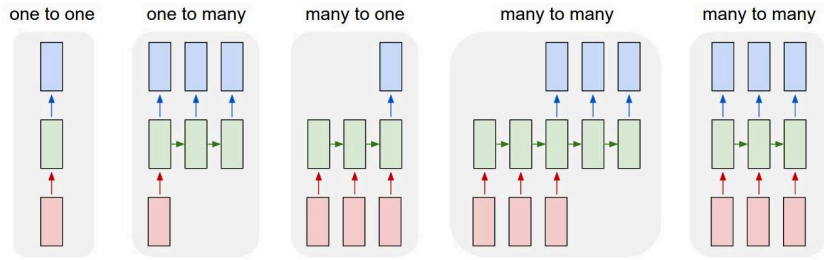
De la misma manera que cuando hablamos de redes convolucionales, existen muchísimas variantes del *Transformer* para diferentes tareas. En el campo del NLP cada día aparecen nuevas empresas con sus propios *Transformers*, como OpenAI, Meta, Mixtral y un larguísimo etcétera. Esta tecnología promete una nueva revolución en todos los ámbitos, desde la educación hasta la medicina, pasando por la industria y la investigación. Hay quienes incluso opinan que usando `Transformers` cada vez más grandes, entrenados con más datos, y algún que otro ajuste algorítmico (como por ejemplo añadir capacidades de planificación), nos llevarán a la AGI antes de 2030. ¿Será cierto? Solo el tiempo lo dirá.

Otros tipos de redes neuronales

Cerramos este capítulo mencionando otras arquitectura de redes no tan populares o usadas, pero que pueden ser útiles en ciertas situaciones.

Redes neuronales recurrentes

Las redes neuronales recurrentes eran la arquitectura de red neuronal más usada para procesar secuencias de datos, como texto, hasta la aparición del *Tranformer*. De hecho, muchas de las ideas fundamentales de las redes recurrentes se ven reflejadas en los `Transformers`.



Estas redes pueden ser útiles en aplicaciones como la predicción de series temporales, aunque como ya hemos comentado, en la mayoría de los casos los `Transformers` son una mejor opción. Variantes a destacar dentro de las redes recurrentes son las redes LSTM y GRU, que permiten un mejor manejo de las dependencias a largo plazo.

Graph Neural Networks

Las redes neuronales convencionales están diseñadas para trabajar con datos estructurados en forma de vectores o matrices. Sin embargo, en muchos casos los datos que queremos procesar tienen una estructura más compleja, como los grafos. En estos casos, las `Graph Neural Networks` (GNNs) son una buena opción. Estas redes son capaces de trabajar con nodos y aristas, y son muy útiles en aplicaciones como la recomendación de productos, el análisis de redes sociales o la química computacional.

State Space Models

Una familia de modelos que está ganando popularidad recientemente son las redes basadas en modelos de espacio de estados, popularizados por la arquitectura llamada Mamba. Estos modelos tienen como objetivo mejorar a los `Transformers` y se basan en la idea de poder comportarse como redes convolucionales durante el entrenamiento (lo cual es eficiente) y como redes recurrentes en la fase de inferencia. Esto permite superar la principal limitación de los `Transformers`, y es su elevada complejidad a la hora de trabajar con secuencias de longitud elevada, ya que necesitan procesar la secuencia entera en todo momento. Sin embargo, es un tipo de modelo nuevo y solo el tiempo dirá si realmente es capaz de superar a los `Transformers` en tareas de procesamiento de lenguaje natural (quién sabe, quizás cuando leas este libro ya lo haya hecho).

6. Entrenando Redes Neuronales

¡Bienvenido al último capítulo de este libro! En este punto deberías tener una comprensión sólida de los fundamentos de las redes neuronales y cómo utilizar `Python`, `Pytorch` y otras herramientas del ecosistema para diseñarlas, entrenarlas y evaluarlas. En este capítulo, vamos a poner en práctica todo lo que hemos aprendido hasta ahora y también te voy a dar algunos consejos por el camino sobre cómo entrenar tus redes neuronales de manera efectiva.

Conjuntos de datos

La primera regla para entrenar redes neuronales es que tus datos deben ser representativos de la tarea que estás tratando de resolver. Si tus datos no son representativos, tu modelo no podrá generalizar bien a nuevos datos. Esto significa que tus datos de entrenamiento tienen que ser lo más parecidos posible a los datos que el modelo encontrará en la vida real, en la aplicación en la que esté funcionando. Obviamente, si no estás desarrollando ninguna aplicación o solo quieres evaluar un modelo para un *benchmark*, este punto no es tan importante. La segunda regla es que cuantos más datos tengas, mejor. Esto se debe a que las redes neuronales son modelos muy complejos y necesitan una gran cantidad de datos para poder generalizar bien. Si tienes pocos datos, tu modelo puede sobreajustarse a ellos y no generalizar bien a nuevos datos. Por lo tanto, es importante tener un conjunto de datos lo suficientemente grande para que tu modelo pueda aprender de manera efectiva. Por último, la calidad de los datos también es importante. Si tus datos están mal etiquetados o contienen errores, tu modelo aprenderá de esos errores y no podrá generalizar bien. Por lo tanto, es importante tener muchos datos, pero de alta calidad para entrenar tus modelos. Cuanto peor sea la calidad, más datos necesitarás para compensarla, y viceversa.

Dicho esto, vamos a recuperar nuestro ejemplo para entrenar un MLP en el dataset MNIST.

Python

```
from sklearn.datasets import fetch_openml
import numpy as np

mnist = fetch_openml('mnist_784', version=1)
X, Y = mnist["data"].values.astype(np.float32), mnist["target"].values.
    astype(int)
```

Python

```
import torch

class Dataset(torch.utils.data.Dataset):

    # constructor
    def __init__(self, X, Y):
        self.X = torch.tensor(X).float()
        self.Y = torch.tensor(Y).long()

    # cantidad de muestras en el dataset
    def __len__(self):
        return len(self.X)

    # devolvemos el elemento `ix` del dataset
    def __getitem__(self, ix):
        return self.X[ix], self.Y[ix]
```

Tal y como hemos dicho, cuantos más datos mejor, por lo que vamos a entrenar nuestro modelo con todos los datos disponibles.

Python

```
from sklearn.metrics import accuracy_score

def softmax(x):
    return torch.exp(x) / torch.exp(x).sum(axis=-1,keepdims=True)

def train(model, epochs = 30, batch_size=1000, log_each = 10):
    dataset = Dataset(X, Y)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=
        batch_size, shuffle=True, num_workers=4, pin_memory=True)
    model.cuda()
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters())
    l, acc = [], []
    for e in range(1, epochs+1):
        _l, _acc = [], []
        # entrenamiento
        model.train()
        for batch_ix, (x, y) in enumerate(dataloader):
```

```
            x, y = x.cuda(), y.cuda()
            optimizer.zero_grad()
            with torch.autocast(device_type='cuda', dtype=torch.bfloat16)
                :
                y_pred = model(x)
                loss = criterion(y_pred, y)
            _l.append(loss.item())
            y_probas = torch.argmax(softmax(y_pred), axis=1)
            _acc.append(accuracy_score(y.cpu().numpy(), y_probas.cpu().
                detach().numpy()))
            loss.backward()
            optimizer.step()
        l.append(np.mean(_l))
        acc.append(np.mean(_acc))
        loss, current = loss.item(), (batch_ix + 1) * len(x)
        if e % log_each == 0:
            print(f"epoch: {e}/{epochs} loss: {loss:.4f} acc {acc[-1]:.5f
                }")
    return {'epoch': list(range(1, epochs+1)), 'loss': l,  'acc': acc}
```

Python

```
from torch.nn import Sequential as S
from torch.nn import Linear as L
from torch.nn import ReLU as R

def build_model(H=128):
    return S(L(784,H),R(),L(H,10))
```

Python

```
model = build_model()

hist = train(model, log_each=1)
```

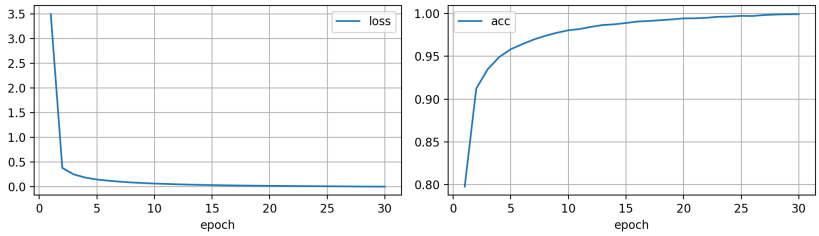
Devuelve:

epoch: 1/30 loss: 0.5655 acc 0.79763 epoch: 2/30 loss: 0.3177 acc 0.91229 epoch: 3/30 loss: 0.2303 acc 0.93497 epoch: 4/30 loss: 0.1534 acc 0.94921 epoch: 5/30 loss: 0.1493 acc 0.95831 epoch: 6/30 loss: 0.1516 acc 0.96420 epoch: 7/30 loss: 0.1461 acc 0.96966 epoch: 8/30 loss: 0.0655 acc 0.97400 epoch: 9/30 loss: 0.0796 acc 0.97754 epoch: 10/30 loss: 0.0728 acc 0.98046 epoch: 11/30 loss: 0.0879 acc 0.98191 epoch: 12/30 loss: 0.0653 acc 0.98451 epoch: 13/30 loss: 0.0427 acc 0.98663 epoch: 14/30 loss: 0.0524 acc 0.98743 epoch: 15/30 loss: 0.0313 acc 0.98896 epoch: 16/30 loss: 0.0279 acc 0.99064 epoch: 17/30 loss: 0.0304 acc 0.99130 epoch: 18/30 loss: 0.0337 acc 0.99227 epoch: 19/30 loss: 0.0381 acc 0.99316 epoch: 20/30 loss: 0.0312 acc

0.99433 epoch: 21/30 loss: 0.0169 acc 0.99447 epoch: 22/30 loss: 0.0178 acc 0.99507 epoch: 23/30 loss: 0.0190 acc 0.99619 epoch: 24/30 loss: 0.0155 acc 0.99644 epoch: 25/30 loss: 0.0216 acc 0.99733 epoch: 26/30 loss: 0.0119 acc 0.99714 epoch: 27/30 loss: 0.0092 acc 0.99824 epoch: 28/30 loss: 0.0131 acc 0.99884 epoch: 29/30 loss: 0.0054 acc 0.99917 epoch: 30/30 loss: 0.0044 acc 0.99927

Python

```
import matplotlib.pyplot as plt
import pandas as pd

fig = plt.figure(dpi=200, figsize=(10,3))
ax = plt.subplot(121)
pd.DataFrame(hist).plot(x='epoch', y='loss', grid=True, ax=ax)
ax = plt.subplot(122)
pd.DataFrame(hist).plot(x='epoch', y='acc', grid=True, ax=ax)
plt.tight_layout()
plt.savefig('resources/training1.png')
plt.close()
```



¡Uau! Llegamos a alcanzar prácticamente el $100\,\%$ de precisión, nuestro modelo clasifica correctamente casi todas la imágenes en el dataset. ¿Es posible que hayamos entrenado el mejor modelo de la historia? Lamentablemente, la respuesta es NO. El procedimiento que hemos seguido tiene un fallo, y es que cuando este modelo esté trabajando en el mundo real, las imágenes que va a recibir serán diferentes a las utilizadas durante el entrenamiento, y ahora mismo no tenemos ni idea de cómo se va a comportar. Es por esto que necesitamos evaluar nuestro modelo en un conjunto de imágenes que no hayamos usado para entrenar para hacernos una idea de la *performance* de nuestro modelo en el mundo real.

El conjunto de *validación*

Para evaluar un modelo mientras lo estamos entrenando, usamos el conjunto de datos de *validación*. Para ello, simplemente separaremos un conjunto de imágenes de nuestro dataset para este objetivo.

Python

```
    def train(model, epochs = 30, batch_size=1000, dataloader = None,
        log_each = 10, optimizer = None, scheduler=None):
        if dataloader is None:
            dataset = {
                "train": Dataset(X[:60000], Y[:60000]), # 60.000 imágenes
                    para entrenamiento
                "val": Dataset(X[60000:], Y[60000:])    # 10.000 imágenes
                    para validación
            }
            dataloader = {
                'train': torch.utils.data.DataLoader(dataset['train'],
                    batch_size=batch_size, shuffle=True, num_workers=4,
                    pin_memory=True),
                'val': torch.utils.data.DataLoader(dataset['val'], batch_size
                    =batch_size, num_workers=4, pin_memory=True)
            }
        model.cuda()
        criterion = torch.nn.CrossEntropyLoss()
        if optimizer is None:
            optimizer = torch.optim.Adam(model.parameters())
        l, acc, lr = [], [], []
        val_l, val_acc = [], []
        for e in range(1, epochs+1):
            _l, _acc = [], []
            for param_group in optimizer.param_groups:
                lr.append(param_group['lr'])
            # entrenamiento
            model.train()
            for batch_ix, (x, y) in enumerate(dataloader['train']):
                x, y = x.cuda(), y.cuda()
                optimizer.zero_grad()
                with torch.autocast(device_type='cuda', dtype=torch.bfloat16)
                    :
                    y_pred = model(x)
                    loss = criterion(y_pred, y)
                _l.append(loss.item())
                y_probas = torch.argmax(softmax(y_pred), axis=1)
                _acc.append(accuracy_score(y.cpu().numpy(), y_probas.cpu().
                    detach().numpy()))
                loss.backward()
                optimizer.step()
            l.append(np.mean(_l))
            acc.append(np.mean(_acc))
            loss, current = loss.item(), (batch_ix + 1) * len(x)
```

```
        # validación
        model.eval()
        _l, _acc = [], []
        with torch.no_grad():
            for batch_ix, (x, y) in enumerate(dataloader['val']):
                x, y = x.cuda(), y.cuda()
                with torch.autocast(device_type='cuda', dtype=torch.
                    bfloat16):
                    y_pred = model(x)
                    loss = criterion(y_pred, y)
                    _l.append(loss.item())
                    y_probas = torch.argmax(softmax(y_pred), axis=1)
                    _acc.append(accuracy_score(y.cpu().numpy(), y_probas.
                        cpu().numpy()))
        val_l.append(np.mean(_l))
        val_acc.append(np.mean(_acc))
        if scheduler:
            scheduler.step()
        if e % log_each == 0:
            print(f"Epoch {e}/{epochs} loss {l[-1]:.5f} acc {acc[-1]:.5f}
                 val_loss {val_l[-1]:.5f} val_acc {val_acc[-1]:.5f}")
    return {'epoch': list(range(1, epochs+1)), 'loss': l, 'acc': acc, '
        val_loss': val_l, 'val_acc': val_acc, 'lr': lr}
```

Python

```
model = build_model()
hist = train(model, log_each=1)
```

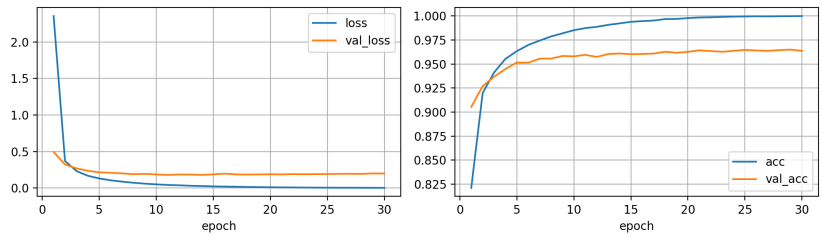
Devuelve:

Epoch 1/30 loss 2.35522 acc 0.82115 val_loss 0.49402 val_acc 0.90540 Epoch 2/30 loss 0.37029 acc 0.91983 val_loss 0.32225 val_acc 0.92700 Epoch 3/30 loss 0.23295 acc 0.94123 val_loss 0.26950 val_acc 0.93680 Epoch 4/30 loss 0.16752 acc 0.95538 val_loss 0.23549 val_acc 0.94490 Epoch 5/30 loss 0.13100 acc 0.96345 val_loss 0.21500 val_acc 0.95160 Epoch 6/30 loss 0.10550 acc 0.96985 val_loss 0.20819 val_acc 0.95140 Epoch 7/30 loss 0.08739 acc 0.97443 val_loss 0.20069 val_acc 0.95560 Epoch 8/30 loss 0.07154 acc 0.97878 val_loss 0.18907 val_acc 0.95570 Epoch 9/30 loss 0.05938 acc 0.98197 val_loss 0.19343 val_acc 0.95840 Epoch 10/30 loss 0.05007 acc 0.98522 val_loss 0.18539 val_acc 0.95800 Epoch 11/30 loss 0.04165 acc 0.98742 val_loss 0.17849 val_acc 0.95960 Epoch 12/30 loss 0.03712 acc 0.98873 val_loss 0.18533 val_acc 0.95740 Epoch 13/30 loss 0.03108 acc 0.99078 val_loss 0.18387 val_acc 0.96040 Epoch 14/30 loss 0.02682 acc 0.99232 val_loss 0.17895 val_acc 0.96100 Epoch 15/30 loss 0.02238 acc 0.99395 val_loss 0.18630 val_acc 0.96020 Epoch 16/30 loss 0.01960 acc 0.99467 val_loss 0.19699 val_acc 0.96050 Epoch 17/30 loss 0.01764 acc 0.99530 val_loss 0.18579

val_acc 0.96090 Epoch 18/30 loss 0.01405 acc 0.99682 val_loss 0.18387 val_acc 0.96280 Epoch 19/30 loss 0.01239 acc 0.99697 val_loss 0.18613 val_acc 0.96170 Epoch 20/30 loss 0.01017 acc 0.99780 val_loss 0.18761 val_acc 0.96270 Epoch 21/30 loss 0.00873 acc 0.99845 val_loss 0.18594 val_acc 0.96430 Epoch 22/30 loss 0.00763 acc 0.99867 val_loss 0.19025 val_acc 0.96350 Epoch 23/30 loss 0.00619 acc 0.99907 val_loss 0.18842 val_acc 0.96280 Epoch 24/30 loss 0.00503 acc 0.99938 val_loss 0.18986 val_acc 0.96370 Epoch 25/30 loss 0.00429 acc 0.99955 val_loss 0.19143 val_acc 0.96470 Epoch 26/30 loss 0.00358 acc 0.99972 val_loss 0.19426 val_acc 0.96410 Epoch 27/30 loss 0.00383 acc 0.99960 val_loss 0.19508 val_acc 0.96370 Epoch 28/30 loss 0.00299 acc 0.99975 val_loss 0.19330 val_acc 0.96450 Epoch 29/30 loss 0.00269 acc 0.99982 val_loss 0.19944 val_acc 0.96500 Epoch 30/30 loss 0.00214 acc 0.99992 val_loss 0.19853 val_acc 0.96370

Python

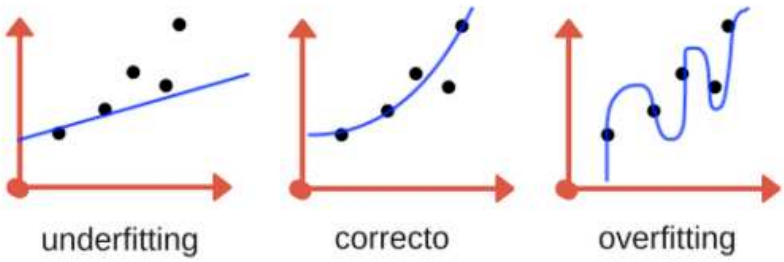
```
def plot_hist(hist, name):
    fig = plt.figure(dpi=200, figsize=(10,3))
    ax = plt.subplot(121)
    pd.DataFrame(hist).plot(x='epoch', y=['loss', 'val_loss'], grid=True,
         ax=ax)
    ax = plt.subplot(122)
    pd.DataFrame(hist).plot(x='epoch', y=['acc', 'val_acc'], grid=True,
        ax=ax)
    plt.tight_layout()
    plt.savefig(name)
    plt.close()

plot_hist(hist, 'resources/training2.png')
```



Oh, oh… Nuestro modelo es perfecto en los datos de entrenamiento, sin embargo, al evaluarlo en los datos de *validación* (recuerda, datos no usados para entrenar) la precisión se reduce. Este fenómeno se conoce por el nombre de *overfitting*, y se da cuando un modelo se ajusta muy bien a los datos de entrenamiento pero luego falla en datos nuevos, no vistos en el entrenamiento.

Capacidad del modelo

Hablamos de la capacidad de un modelo para referirnos a su potencia a la hora de representar un conjunto de datos.



Si un modelo tiene poca capacidad, observaremos *underfitting*. El modelo no tiene suficientes parámetros para representar correctamente un dataset y tendremos una mala precisión.

Python

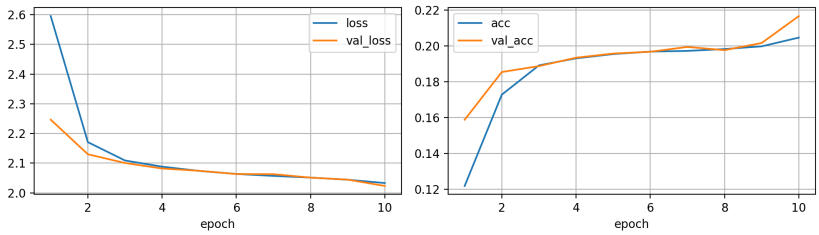
```
model = build_model(H=3)

hist = train(model, epochs=10, log_each=1)
```

Devuelve:

Epoch 1/10 loss 2.59526 acc 0.12185 val_loss 2.24648 val_acc 0.15880 Epoch 2/10 loss 2.17082 acc 0.17277 val_loss 2.13006 val_acc 0.18540 Epoch 3/10 loss 2.10906 acc 0.18912 val_loss 2.10064 val_acc 0.18870 Epoch 4/10 loss 2.08828 acc 0.19303 val_loss 2.08202 val_acc 0.19340 Epoch 5/10 loss 2.07397 acc 0.19542 val_loss 2.07430 val_acc 0.19570 Epoch 6/10 loss 2.06367 acc 0.19680 val_loss 2.06342 val_acc 0.19670 Epoch 7/10 loss 2.05706 acc 0.19718 val_loss 2.06300 val_acc 0.19940 Epoch 8/10 loss 2.05150 acc 0.19822 val_loss 2.05128 val_acc 0.19760 Epoch 9/10 loss 2.04433 acc 0.19973 val_loss 2.04459 val_acc 0.20160 Epoch 10/10 loss 2.03308 acc 0.20460 val_loss 2.02334 val_acc 0.21660

Python

```
plot_hist(hist, 'resources/training3.png')
```



Nuestro modelo simple no es capaz de alcanzar la máxima precisión en los datos de entrenamiento, esta es la clara señal de *underfitting* y en este caso simplemente tendremos que aumentar el número de parámetros.

En el caso contrario, si nuestro modelo tiene mucha capacidad (más parámetros de los necesarios) se ajustará muy bien a los datos de entrenamiento pero no será capaz de generalizar a datos no vistos durante el entrenamiento. Esto es lo que llamamos *overfitting*, y es el fenómeno que estamos observando en nuestro ejemplo.

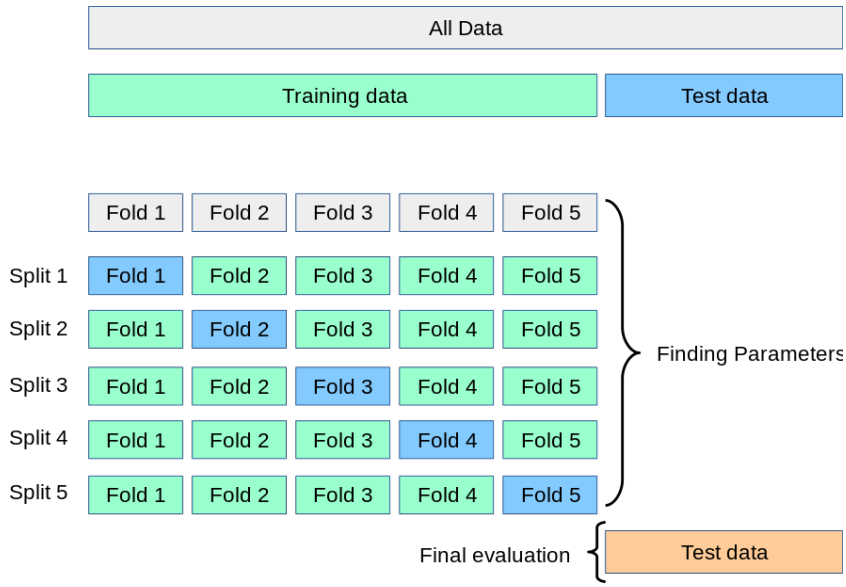
En este punto, podrías estar tentado a probar diferentes arquitecturas, modificando el número de capas del `MLP` por ejemplo, y utilizar la mejor combinación de parámetros que maximice la métrica en el conjunto de validación. Sin embargo, esta aproximación no es la correcta, ya que lo único que conseguiríamos sería hacer *overfitting* al conjunto de validación. Es por esto que utilizamos los datos de *validación* para probar y comparar diferentes modelos, así como para buscar el conjunto de mejores hiperparámetros, pero a la hora de dar las métricas finales siempre utilizamos un tercer conjunto de datos, el conjunto de *test*.

El conjunto de *test*

Si bien usar un conjunto de entrenamiento y otro de validación es una buena práctica, la realidad es que si usamos el conjunto de validación para probar diferentes modelos o hiperparámetros, corremos el riesgo de hacer *overfitting* a este conjunto. Para evitar esto, utilizamos un tercer conjunto de datos, el conjunto de *test*. Este conjunto de datos no se utiliza en el entrenamiento ni en la validación, y solo se utiliza al final del proceso para evaluar el modelo final. De esta manera, podemos estar seguros de que nuestro modelo generaliza bien a datos no vistos durante el entrenamiento y la validación.

Validación cruzada

Hemos visto que para evaluar correctamente un modelo necesitamos un conjunto de datos de *test*. Además, para poder iterar diferentes versiones de un modelo, necesitamos un conjunto de *validación*. Este es el procedimiento más utilizado cuando trabajamos con modelos o *datasets* muy grandes que requieren muchos recursos computacionales. No obstante, existe una mejor aproximación que, si nos la podemos permitir, nos dará muchos mejores resultados: la validación cruzada.



Esta técnica consiste en dividir nuestro conjunto de datos de entrenamiento en diferentes paquetes, o *folds* en inglés, y entrenar tantos modelos como *folds* tengamos, utilizando un *fold* diferente para validar en cada caso y entrenando con el resto de *folds*. De esta manera, habremos entrenado y validado con todos los datos de entrenamiento. Esta técnica permite, además, utilizar todos los modelos entrenados para generar las predicciones finales en lo que se conoce como un ensamblado de modelos. La idea es que las predicciones generadas por este ensamblado serán mejores que cualquier predicción hecha por un modelo individual, ya que las debilidades de un modelo serán compensadas por el resto. Esta técnica es muy utilizada en competiciones para sacar ese extra de precisión final que puede marcar la diferencia.

Python

```
from sklearn.model_selection import KFold

FOLDS = 5
kf = KFold(n_splits=FOLDS)


# separamos en train y test
X_train, X_test, y_train, y_test = X[:60000] / 255., X[60000:] / 255., Y
    [:60000], Y[60000:]

X_train.shape, X_test.shape, kf.get_n_splits(X)
```

Devuelve: *((60000, 784), (10000, 784), 5)*

Python

```
train_accs, val_accs = [], []
models = []

# hacemos validación cruzada con datos de train
for k, (train_index, val_index) in enumerate(kf.split(X_train)):
    print("Fold:", k+1)
    X_train_fold, X_val_fold = X_train[train_index], X_train[val_index]
    y_train_fold, y_val_fold = y_train[train_index], y_train[val_index]
    dataset = {
        'train': Dataset(X_train_fold, y_train_fold),
        'val': Dataset(X_val_fold, y_val_fold)
    }
    dataloader = {
        'train': torch.utils.data.DataLoader(dataset['train'], batch_size
            =100, shuffle=True),
        'val': torch.utils.data.DataLoader(dataset['val'], batch_size
            =1000, shuffle=False)
    }
    model = build_model()
    models.append(model)
    hist = train(model, dataloader=dataloader)
    train_accs.append(hist['acc'][-1])
    val_accs.append(hist['val_acc'][-1])
```

Devuelve:

Fold: 1 Epoch 10/30 loss 0.03855 acc 0.98883 val_loss 0.08359 val_acc 0.97633 Epoch 20/30 loss 0.00876 acc 0.99833 val_loss 0.09467 val_acc 0.97633 Epoch 30/30 loss 0.00120 acc 0.99996 val_loss 0.10376 val_acc 0.97867 Fold: 2 Epoch 10/30 loss 0.03729 acc 0.98948 val_loss 0.09162 val_acc 0.97183 Epoch 20/30 loss 0.00743 acc 0.99881 val_loss 0.09673 val_acc 0.97500 Epoch 30/30 loss 0.00113 acc 0.99996

val_loss 0.11296 val_acc 0.97650 Fold: 3 Epoch 10/30 loss 0.03670 acc 0.98971 val_loss 0.09226 val_acc 0.97258 Epoch 20/30 loss 0.00671 acc 0.99896 val_loss 0.09864 val_acc 0.97525 Epoch 30/30 loss 0.00118 acc 0.99998 val_loss 0.11357 val_acc 0.97617 Fold: 4 Epoch 10/30 loss 0.03984 acc 0.98865 val_loss 0.10516 val_acc 0.96942 Epoch 20/30 loss 0.00712 acc 0.99904 val_loss 0.11672 val_acc 0.97158 Epoch 30/30 loss 0.00101 acc 1.00000 val_loss 0.13117 val_acc 0.97317 Fold: 5 Epoch 10/30 loss 0.03753 acc 0.99019 val_loss 0.09023 val_acc 0.97233 Epoch 20/30 loss 0.00772 acc 0.99873 val_loss 0.09824 val_acc 0.97650 Epoch 30/30 loss 0.00104 acc 0.99998 val_loss 0.11030 val_acc 0.97617

Hacer validación cruzada nos permite dar un intervalo de confianza en las métricas, lo que nos permite conocer cómo de seguro está nuestro modelo en sus predicciones (información que podemos tener en cuenta a la hora de escoger un modelo sobre otro).

Python

```
np.mean(train_accs), np.std(train_accs)
```

Devuelve: *(0.9999750000000001, 1.5590239111529667e-05)*

Python

```
np.mean(val_accs), np.std(val_accs)
```

Devuelve: *(0.9761333333333333, 0.0017524585904126753)*

Finalmente, podemos calcular las métricas finales sobre el conjunto de test.

Python

```
dataset = { 'test': Dataset(X_test, y_test) }
dataloader = { 'test': torch.utils.data.DataLoader(dataset['test'],
    batch_size=100) }
test_accs = []
with torch.no_grad():
    for model in models:
        model.eval()
        preds = torch.tensor([]).cuda()
```

```
            for batch_ix, (x, y) in enumerate(dataloader['test']):
                x, y = x.cuda(), y.cuda()
                with torch.autocast(device_type='cuda', dtype=torch.bfloat16)
                    :
                    y_pred = model(x)
                    preds = torch.cat([preds, y_pred])
            preds = torch.mean(preds, axis=0)
            y_probas = torch.argmax(softmax(y_pred), axis=1)
            test_accs.append(accuracy_score(y.cpu().numpy(), y_probas.cpu().
                numpy()))

    np.mean(test_accs), np.std(test_accs)
```

Devuelve: *(0.986, 0.01019803902718558)*

Optimización

Vamos a ver ahora diferentes técnicas para acelerar el proceso de optimización y, potencialmente, obtener mejores resultados de los que hemos sido capaces de obtener hasta ahora.

Optimizadores

En el capítulo 4 aprendimos el algoritmo de `stochastic gradient descent`, el cual actualiza los pesos del modelo siguiente la regla $w \leftarrow w - \eta \ \frac{dl}{dw}$, donde w son los parámetros de la red neuronal, η es el factor de aprendizaje (o *learning rate*) y $\frac{dl}{dw}$ es la derivada de la función de pérdida con respecto a los parámetros. Si bien este algoritmo funciona y en ocasiones es la mejor elección, existen otros algoritmos de optimización que pueden acelerar el entrenamiento requiriendo muchas menos *epochs* para alacanzar el mismo resultado, o para el mismo número de *epochs* encontrar una mejor solución.

Momentum

La primera mejora que podemos hacer sobre el algoritmo *SGD* es añadir *momentum*. De la misma manera que una pelota rodando por una superficie inclinada acelera poco a poco debido a la fuerza de la gravedad (a esto se le llama ganar *momentum*) podemos hacer que nuestros pasos de `descenso por gradiente` sean más grandes si el signo del gradiente es el mismo durante *updates* consecutivos.

$$m \leftarrow \beta m - \eta \frac{dl}{dw}$$

$$w \leftarrow w + m$$

En `Pytorch` podemos controlar el valor de β (cuanto *momentum* queremos añadir) con el parámetro `momentum`.

RMSProp

El algoritmo del `descenso por gradiente` funciona intentando descender rápidamente en la dirección en la que el gradiente es mayor. Sin embargo, esta dirección no tiene por qué ser siempre la que indica el camino hacia el óptimo global. El algoritmo de optimización `RMSProp` intenta tener este factor en cuenta escalando el gradiente en la dimensión con mayor pendiente:

$$s \leftarrow \beta s + (1 - \beta) \frac{dl}{dw} \frac{dl}{dw}$$

$$w \leftarrow w - \eta \frac{dl}{dw} \sqrt{s + \epsilon}$$

Donde β es el *decay rate* (normalmente 0.9) y ϵ es un valor pequeño para evitar división entre 0. En `Pytorch` podemos usar este optimizador con la clase `torch.optim.RMSProp`.

Adam

Adam combina las ideas de *momentum* y *RMSProp*, y es la opción elegida por defecto por la mayoría de desarrolladores.

$$m \leftarrow \beta_1 m - (1 - \beta_1) \frac{dl}{dw}$$
$$s \leftarrow \beta_2 s + (1 - \beta_2) \frac{dl}{dw} \frac{dl}{dw}$$
$$\hat{m} \leftarrow \frac{m}{1 - \beta_1^T}$$

$$\hat{s} \leftarrow \frac{s}{1-\beta_2^T}$$
$$w \leftarrow w + \eta \hat{m} \sqrt{\hat{s} + \epsilon}$$

Comparativa

Vamos a comparar los diferentes optimizadores presentados.

Python

```
model = build_model()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
hist_sgd = train(model, optimizer=optimizer)
```

Devuelve:

Epoch 10/30 loss 0.24939 acc 0.93285 val_loss 0.27315 val_acc 0.92670 Epoch 20/30 loss 0.16529 acc 0.95342 val_loss 0.21386 val_acc 0.94100 Epoch 30/30 loss 0.12747 acc 0.96308 val_loss 0.18771 val_acc 0.94730

Python

```
model = build_model()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
hist_momentum = train(model, optimizer=optimizer)
```

Devuelve:

Epoch 10/30 loss 0.14080 acc 0.95803 val_loss 0.17144 val_acc 0.95250 Epoch 20/30 loss 0.09878 acc 0.96992 val_loss 0.15510 val_acc 0.95700 Epoch 30/30 loss 0.07690 acc 0.97663 val_loss 0.15185 val_acc 0.95910

Python

```
model = build_model()
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.001)
hist_rms = train(model, optimizer=optimizer)
```

Devuelve:

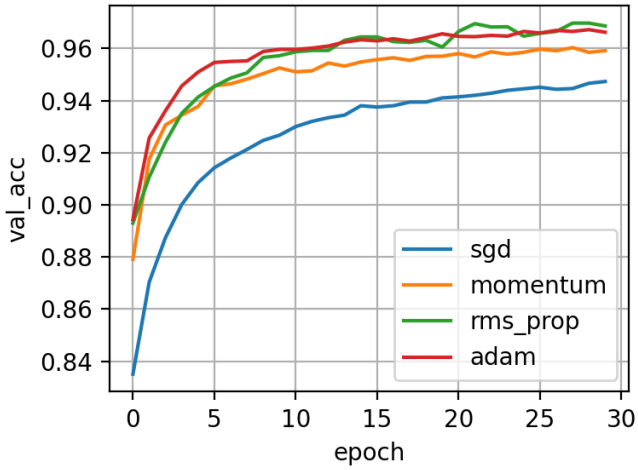
Epoch 10/30 loss 0.10426 acc 0.96815 val_loss 0.18289 val_acc 0.95720 Epoch 20/30 loss 0.66525 acc 0.94103 val_loss 0.20484 val_acc 0.96050 Epoch 30/30 loss 0.01923 acc 0.99393 val_loss 0.18819 val_acc 0.96860

Python

```
model = build_model()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
hist_adam = train(model, optimizer=optimizer)
```

Devuelve:

Epoch 10/30 loss 0.04418 acc 0.98670 val_loss 0.16219 val_acc 0.95960 Epoch 20/30 loss 0.00670 acc 0.99920 val_loss 0.16578 val_acc 0.96560 Epoch 30/30 loss 0.00151 acc 0.99998 val_loss 0.17708 val_acc 0.96620



Como puedes ver, las variantes de *SGD* son capaces de converger mucho más rápido. Aun así, si le damos suficiente tiempo, *SGD* es capaz de llegar a los mismos resultados, e incluso superarlos. Existen muchos otros optimizadores, sin embargo, los vistos aquí son los más comunes, siendo Adam un buen punto de partida.

Otras técnicas que pueden influir en el proceso de optimización son:

- *Learning Rate Scheduling*: Ir variando el *learning rate* a medida que avanzamos en el proceso de optimización es una práctica común que ayuda a entrenar modelos más rápido.
- Normalización: Normalizar los datos de entrada así como las activaciones de diferentes capas puede acelerar el proceso de optimización, por ejemplo con *Batch Nor-*

malization o diferentes variantes de *Layer Normalization*.

- Inicialización: Inicializar los pesos de la red de manera adecuada puede acelerar el proceso de optimización, lo cual incluye el *Transfer Learning*, técnica en la que usamos una red preentrenada en otro conjunto de datos como punto de partida puede acelerar el proceso de optimización (y que veremos más en detalle en la siguiente sección por su importancia).

Transfer learning

Una de las técnicas más importantes que debes conocer es cómo utilizar redes neuronales que ya han sido pre-entrenadas en algún conjunto de datos. Esto se debe a que para la mayoría de aplicaciones no es común disponer de suficientes datos como para entrenar de manera efectiva un modelo desde cero. Vamos a ver un ejemplo de cómo podemos utilizar una red pre-entrenada en Imagenet y adaptarla a un nuevo caso con pocos datos.

Python

```
import wget

wget.download('https://mymldatasets.s3.eu-de.cloud-object-storage.
    appdomain.cloud/flowers.zip')
```

Python

```
import zipfile

with zipfile.ZipFile('flowers.zip', 'r') as zip_ref:
    zip_ref.extractall('.')
```

Una vez extraído el dataset, podemos ver que tenemos 5 clases de flores diferentes, distribuidas en 5 carpetas diferentes. Cada carpeta contiene varios ejemplos de flores de la categoría en cuestión.

Python

```
import os

PATH = 'flowers'

classes = os.listdir(PATH)
classes
```

Devuelve: *['tulip', 'dandelion', 'rose', 'sunflower', 'daisy']*

Python

```
imgs, labels = [], []

for i, lab in enumerate(classes):
  paths = os.listdir(f'{PATH}/{lab}')
  print(f'Categoría: {lab}. Imágenes: {len(paths)}')
  paths = [p for p in paths if p[-3:] == "jpg"]
  imgs += [f'{PATH}/{lab}/{img}' for img in paths]
  labels += [i]*len(paths)
```

Devuelve:

Categoría: tulip. Imágenes: 984 Categoría: dandelion. Imágenes: 1055 Categoría: rose. Imágenes: 784 Categoría: sunflower. Imágenes: 734 Categoría: daisy. Imágenes: 769

Podemos visualizar algunas imágenes en el dataset.

Vamos a crear también un subconjunto de test para evluar nuestros modelos.

Python

```python
from sklearn.model_selection import train_test_split

train_imgs, test_imgs, train_labels, test_labels = train_test_split(imgs,
     labels, test_size=0.2, stratify=labels)

len(train_imgs), len(test_imgs)
```

Devuelve: *(3458, 865)*

Y, por último, creamos nuestros objetos `Dataset` y `DataLoader` para poder darle las imágenes a nuestros modelos.

Python

```python
import torch
device = "cuda" if torch.cuda.is_available() else "cpu"

class Dataset(torch.utils.data.Dataset):
  def __init__(self, X, y, trans, device):
    self.X = X
    self.y = y
    self.trans = trans
    self.device = device

  def __len__(self):
    return len(self.X)

  def __getitem__(self, ix):
    # cargar la imagen
    img = io.imread(self.X[ix])
    # aplicar transformaciones
    if self.trans:
      img = self.trans(image=img)["image"]
    return torch.from_numpy(img / 255.).float().permute(2,0,1), torch.
        tensor(self.y[ix])
```

Nos aseguraremos de que todas las imágenes del dataset tengan las mismas dimensiones: 224x224 píxeles. Para ello, usaremos la librería `albumentations`, la cual te recomiendo a la hora de trabajar con imágenes.

Python

```python
import albumentations as A

trans = A.Compose([
    A.Resize(224, 224)
])

dataset = {
    'train': Dataset(train_imgs, train_labels, trans, device),
    'test': Dataset(test_imgs, test_labels, trans, device)
}

len(dataset['train']), len(dataset['test'])
```

Devuelve: *(3458, 865)*

Python

```python
fig, axs = plt.subplots(3,5, figsize=(10,6))
for _ax in axs:
  for ax in _ax:
    ix = random.randint(0, len(dataset['train'])-1)
    img, lab = dataset['train'][ix]
    ax.imshow(img.permute(1,2,0))
    ax.axis('off')
    ax.set_title(classes[lab])
plt.tight_layout()
plt.savefig('resources/transfer_learning2.png')
plt.close()
```

Python

```
dataloader = {
    'train': torch.utils.data.DataLoader(dataset['train'], batch_size=64,
         shuffle=True, pin_memory=True),
    'test': torch.utils.data.DataLoader(dataset['test'], batch_size=256,
        shuffle=False)
}

imgs, labels = next(iter(dataloader['train']))
imgs.shape
```

Devuelve: *torch.Size([64, 3, 224, 224])*

Escogemos la arquitectura `resnet`, de la que ya hablamos en el capítulo anterior, para hacer nuestro clasificador. De este modelo usaremos todas las capas excepto la última, que sustituiremos por una nueva capa lineal para llevar a cabo la clasificación en 5 clases.

Python

```
import torchvision

resnet = torchvision.models.resnet18()
resnet
```

Devuelve:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
      3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
      ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
          =(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
    )
    (1): BasicBlock(
```

```
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
        =(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding
        =(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
)
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding
        =(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=
          False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
          track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
  )
)
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
        padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=
```

```
            False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
            padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=
              False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=1000, bias=True)
  )
```

Python

```python
class Model(torch.nn.Module):
  def __init__(self, n_outputs=5, pretrained=False, freeze=False):
    super().__init__()
    # descargamos resnet
    resnet = torchvision.models.resnet18(pretrained=pretrained)
    # nos quedamos con todas las capas menos la última
    self.resnet = torch.nn.Sequential(*list(resnet.children())[:-1])
    if freeze:
      for param in self.resnet.parameters():
        param.requires_grad=False
    # añadimos una nueva capa lineal para llevar a cabo la clasificación
    self.fc = torch.nn.Linear(512, 5)

  def forward(self, x):
    x = self.resnet(x)
    x = x.view(x.shape[0], -1)
    x = self.fc(x)
    return x

  def unfreeze(self):
    for param in self.resnet.parameters():
        param.requires_grad=True
```

Python

```python
model = Model()
outputs = model(torch.randn(64, 3, 224, 224))
outputs.shape
```

Devuelve: *torch.Size([64, 5])*

Python

```python
from tqdm import tqdm
import numpy as np

def fit(model, dataloader, epochs=5, lr=1e-2):
    model.to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    criterion = torch.nn.CrossEntropyLoss()
    for epoch in range(1, epochs+1):
        model.train()
        train_loss, train_acc = [], []
        bar = tqdm(dataloader['train'])
        for batch in bar:
            X, y = batch
            X, y = X.to(device), y.to(device)
            optimizer.zero_grad()
            y_hat = model(X)
            loss = criterion(y_hat, y)
            loss.backward()
            optimizer.step()
            train_loss.append(loss.item())
            acc = (y == torch.argmax(y_hat, axis=1)).sum().item() / len(y
                )
            train_acc.append(acc)
            bar.set_description(f"loss {np.mean(train_loss):.5f} acc {np.
                mean(train_acc):.5f}")
        bar = tqdm(dataloader['test'])
        val_loss, val_acc = [], []
        model.eval()
        with torch.no_grad():
            for batch in bar:
                X, y = batch
                X, y = X.to(device), y.to(device)
                y_hat = model(X)
                loss = criterion(y_hat, y)
                val_loss.append(loss.item())
                acc = (y == torch.argmax(y_hat, axis=1)).sum().item() /
                    len(y)
                val_acc.append(acc)
                bar.set_description(f"val_loss {np.mean(val_loss):.5f}
                    val_acc {np.mean(val_acc):.5f}")
        print(f"Epoch {epoch}/{epochs} loss {np.mean(train_loss):.5f}
            val_loss {np.mean(val_loss):.5f} acc {np.mean(train_acc):.5f
            } val_acc {np.mean(val_acc):.5f}")
```

En primer lugar, vamos a entrenar nuestro modelo desde cero para ver qué métricas podemos obtener.

Python

```
model = Model()
fit(model, dataloader, epochs=15)
```

Devuelve:

```
loss 1.34495 acc 0.42670: 100%| ▓▓▓▓▓▓ | 55/55 [00:09<00:00,  5.83it/s]
val_loss 3.17634 val_acc 0.24244: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:01<00:00,
     2.10it/s]


Epoch 1/15 loss 1.34495 val_loss 3.17634 acc 0.42670 val_acc 0.24244


loss 1.14354 acc 0.53125: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:09<00:00,  5.76
    it/s]
val_loss 5.15191 val_acc 0.17022: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.97it/s]


Epoch 2/15 loss 1.14354 val_loss 5.15191 acc 0.53125 val_acc 0.17022


loss 1.03202 acc 0.58409: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.43
    it/s]
val_loss 8.90796 val_acc 0.18053: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.80it/s]


Epoch 3/15 loss 1.03202 val_loss 8.90796 acc 0.58409 val_acc 0.18053


loss 0.96974 acc 0.62727: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.20
    it/s]
val_loss 6.71880 val_acc 0.21154: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.76it/s]


Epoch 4/15 loss 0.96974 val_loss 6.71880 acc 0.62727 val_acc 0.21154


loss 0.93328 acc 0.62670: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.09
    it/s]
val_loss 1.89438 val_acc 0.33017: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.69it/s]


Epoch 5/15 loss 0.93328 val_loss 1.89438 acc 0.62670 val_acc 0.33017
```

```
loss 0.84877 acc 0.67244: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.10
    it/s]
val_loss 5.09534 val_acc 0.30923: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.74it/s]


Epoch 6/15 loss 0.84877 val_loss 5.09534 acc 0.67244 val_acc 0.30923


loss 0.81787 acc 0.68466: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.04
    it/s]
val_loss 1.54116 val_acc 0.47102: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.68it/s]


Epoch 7/15 loss 0.81787 val_loss 1.54116 acc 0.68466 val_acc 0.47102


loss 0.80202 acc 0.69801: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.97
    it/s]
val_loss 5.12972 val_acc 0.29783: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.75it/s]


Epoch 8/15 loss 0.80202 val_loss 5.12972 acc 0.69801 val_acc 0.29783


loss 0.72754 acc 0.72898: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.96
    it/s]
val_loss 1.22316 val_acc 0.53254: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.70it/s]


Epoch 9/15 loss 0.72754 val_loss 1.22316 acc 0.72898 val_acc 0.53254


loss 0.68346 acc 0.73693: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.94
    it/s]
val_loss 1.40439 val_acc 0.56523: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.69it/s]


Epoch 10/15 loss 0.68346 val_loss 1.40439 acc 0.73693 val_acc 0.56523


loss 0.64693 acc 0.75540: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.92
    it/s]
val_loss 4.63574 val_acc 0.32802: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.68it/s]


Epoch 11/15 loss 0.64693 val_loss 4.63574 acc 0.75540 val_acc 0.32802
```

```
loss 0.64474 acc 0.77102: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.86
    it/s]
val_loss 2.71527 val_acc 0.34349: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.58it/s]


Epoch 12/15 loss 0.64474 val_loss 2.71527 acc 0.77102 val_acc 0.34349


loss 0.61885 acc 0.77670: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.89
    it/s]
val_loss 1.58861 val_acc 0.52828: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.59it/s]


Epoch 13/15 loss 0.61885 val_loss 1.58861 acc 0.77670 val_acc 0.52828


loss 0.56614 acc 0.79318: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.82
    it/s]
val_loss 3.42326 val_acc 0.43450: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.64it/s]


Epoch 14/15 loss 0.56614 val_loss 3.42326 acc 0.79318 val_acc 0.43450


loss 0.58348 acc 0.79545: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.79
    it/s]
val_loss 2.48764 val_acc 0.36169: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.62it/s]

Epoch 15/15 loss 0.58348 val_loss 2.48764 acc 0.79545 val_acc 0.36169
```

Como puedes ver, es complicado conseguir buenas métricas, ya que nuestro dataset es muy pequeño. Ahora vamos a entrenar el mismo modelo, pero en este caso, utilizando los pesos pre-entrenados de `resnet`. Lo haremos *congelando* la mayor parte de la red, es decir, no actualizaremos los pesos de las capas pre-entrenadas, solo los de la última capa que hemos añadido. Esto se conoce como *transfer learning* ya que estamos transfiriendo todo el conocimiento que tiene un modelo de una tarea a otra diferente.

Python

```python
model = Model(pretrained=True, freeze=True)

fit(model, dataloader)
```

Devuelve:

```
/home/juan/miniconda3/lib/python3.11/site-packages/torchvision/models/
    _utils.py:223: UserWarning: Arguments other than a weight enum or `
    None` for 'weights' are deprecated since 0.13 and may be removed in
    the future. The current behavior is equivalent to passing `weights=
    ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=
    ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
loss 1.06549 acc 0.62983: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.44
    it/s]
val_loss 0.99367 val_acc 0.63991: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.62it/s]


Epoch 1/5 loss 1.06549 val_loss 0.99367 acc 0.62983 val_acc 0.63991


loss 0.64418 acc 0.81506: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.25
    it/s]
val_loss 0.72195 val_acc 0.73537: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.60it/s]


Epoch 2/5 loss 0.64418 val_loss 0.72195 acc 0.81506 val_acc 0.73537


loss 0.54951 acc 0.81847: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.39
    it/s]
val_loss 0.80100 val_acc 0.64585: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.41it/s]


Epoch 3/5 loss 0.54951 val_loss 0.80100 acc 0.81847 val_acc 0.64585


loss 0.50014 acc 0.83750: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.19
    it/s]
val_loss 0.60357 val_acc 0.77720: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.56it/s]


Epoch 4/5 loss 0.50014 val_loss 0.60357 acc 0.83750 val_acc 0.77720


loss 0.46140 acc 0.85085: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.43
    it/s]
val_loss 0.67027 val_acc 0.74478: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.56it/s]

Epoch 5/5 loss 0.46140 val_loss 0.67027 acc 0.85085 val_acc 0.74478
```

Como puedes ver no solo obtenemos un mejor modelo en menos *epochs* sino que además cada *epoch* tarda menos en completarse. Esto es debido a que, al no estar entrenando gran parte de la red, los requisitos computacionales se reducen considerablemente. Mejores modelos y entrenados más rápido. Aun así, podemos mejorar todavía más los resultados si entrenamos toda la red, no solo la última capa. Esto se conoce como *fine-tuning*, o ajuste fino del modelo.

Python

```
model = Model(pretrained=True, freeze=False)

fit(model, dataloader)
```

Devuelve:

```
loss 0.80210 acc 0.72784: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.60
    it/s]
val_loss 0.57471 val_acc 0.79728: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.53it/s]


Epoch 1/5 loss 0.80210 val_loss 0.57471 acc 0.72784 val_acc 0.79728


loss 0.35230 acc 0.89943: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.60
    it/s]
val_loss 0.43203 val_acc 0.83271: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.61it/s]


Epoch 2/5 loss 0.35230 val_loss 0.43203 acc 0.89943 val_acc 0.83271


loss 0.24936 acc 0.92784: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.51
    it/s]
val_loss 0.35053 val_acc 0.87977: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.51it/s]


Epoch 3/5 loss 0.24936 val_loss 0.35053 acc 0.92784 val_acc 0.87977


loss 0.19491 acc 0.95000: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.42
    it/s]
```

```
val_loss 0.33975 val_acc 0.87817: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.50it/s]


Epoch 4/5 loss 0.19491 val_loss 0.33975 acc 0.95000 val_acc 0.87817


loss 0.14396 acc 0.96733: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.57
    it/s]
val_loss 0.32238 val_acc 0.88376: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.48it/s]

Epoch 5/5 loss 0.14396 val_loss 0.32238 acc 0.96733 val_acc 0.88376
```

Es común entrenar primero el modelo sin entrenar la red pre-entrenada durante varias *epochs* y después seguir entrenando, pero permitiendo ahora la actualización de pesos también en la red pre-entrenada (usualmente con un *learning rate* más pequeño).

Python

```python
model = Model(pretrained=True, freeze=True)
fit(model, dataloader)
model.unfreeze()
fit(model, dataloader, lr=1e-4)
```

Devuelve:

```
/home/juan/miniconda3/lib/python3.11/site-packages/torchvision/models/
    _utils.py:208: UserWarning: The parameter 'pretrained' is deprecated
     since 0.13 and may be removed in the future, please use 'weights'
    instead.
  warnings.warn(
/home/juan/miniconda3/lib/python3.11/site-packages/torchvision/models/
    _utils.py:223: UserWarning: Arguments other than a weight enum or `
    None` for 'weights' are deprecated since 0.13 and may be removed in
    the future. The current behavior is equivalent to passing `weights=
    ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=
    ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
loss 1.09279 acc 0.61903: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.23
    it/s]
val_loss 1.17811 val_acc 0.47687: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.49it/s]


Epoch 1/5 loss 1.09279 val_loss 1.17811 acc 0.61903 val_acc 0.47687
```

```
loss 0.65803 acc 0.80028: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.97
    it/s]
val_loss 0.93755 val_acc 0.63183: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.57it/s]


Epoch 2/5 loss 0.65803 val_loss 0.93755 acc 0.80028 val_acc 0.63183


loss 0.55450 acc 0.82131: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:11<00:00,  4.99
    it/s]
val_loss 0.86775 val_acc 0.60874: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.52it/s]


Epoch 3/5 loss 0.55450 val_loss 0.86775 acc 0.82131 val_acc 0.60874


loss 0.48154 acc 0.84176: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.02
    it/s]
val_loss 0.61070 val_acc 0.74443: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.37it/s]


Epoch 4/5 loss 0.48154 val_loss 0.61070 acc 0.84176 val_acc 0.74443


loss 0.43795 acc 0.85994: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:10<00:00,  5.04
    it/s]
val_loss 0.45480 val_acc 0.86583: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.55it/s]


Epoch 5/5 loss 0.43795 val_loss 0.45480 acc 0.85994 val_acc 0.86583


loss 0.48036 acc 0.83608: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.27
    it/s]
val_loss 0.45279 val_acc 0.86130: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.45it/s]


Epoch 1/5 loss 0.48036 val_loss 0.45279 acc 0.83608 val_acc 0.86130


loss 0.44145 acc 0.85540: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.49
    it/s]
val_loss 0.44807 val_acc 0.85810: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.48it/s]


Epoch 2/5 loss 0.44145 val_loss 0.44807 acc 0.85540 val_acc 0.85810
```

```
loss 0.43213 acc 0.85511: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.46
    it/s]
val_loss 0.41750 val_acc 0.87462: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.53it/s]


Epoch 3/5 loss 0.43213 val_loss 0.41750 acc 0.85511 val_acc 0.87462


loss 0.43444 acc 0.85256: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.53
    it/s]
val_loss 0.41720 val_acc 0.87462: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.43it/s]


Epoch 4/5 loss 0.43444 val_loss 0.41720 acc 0.85256 val_acc 0.87462


loss 0.42273 acc 0.85568: 100%| ▓▓▓▓▓▓▓▓▓▓ | 55/55 [00:12<00:00,  4.27
    it/s]
val_loss 0.41057 val_acc 0.88821: 100%| ▓▓▓▓▓▓▓▓▓▓ | 4/4 [00:02<00:00,
     1.41it/s]

Epoch 5/5 loss 0.42273 val_loss 0.41057 acc 0.85568 val_acc 0.88821
```

Otra alternativa de *fine tuning* es la de entrenar el modelo con diferentes *learning rates*, uno para la red pre-entrenada y otro para las capas nuevas. Esto lo puedes conseguir en `Pytorch` de la siguiente manera:

Python

```python
optimizer = torch.optim.Adam([
    {'params': model.resnet.parameters(), 'lr': 1e-4},
    {'params': model.fc.parameters(), 'lr': 1e-3}
])
```

Regularización

El último concepto del que vamos a hablar que puede afectar al entrenamiento de tu modelo es el de la regularización. La regularización es una técnica que se utiliza para evitar el *overfitting* en los modelos de *machine learning*. Vamos a ver diferentes técnicas de regularización usando el dataset CIFAR10.

Python

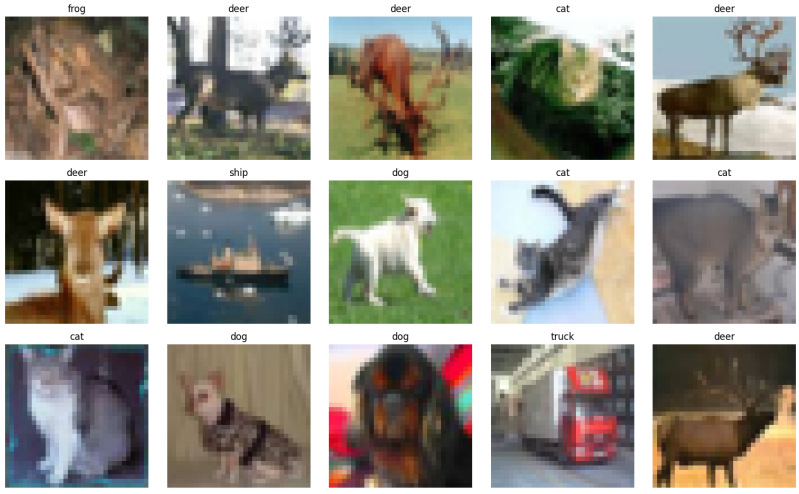
```python
import torchvision

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
    download=True)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
    download=True)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
     'ship', 'truck')

len(trainset), len(testset)
```

Devuelve:

Files already downloaded and verified *Files already downloaded and verified*

(50000, 10000)

Este dataset está formado por imágenes en color de baja resolución, tenemos 50000 imágenes de entrenamiento y 10000 de test y el objetivo es el clasificar en 10 clases diferentes. Veamos algunos ejemplos.



Vamos a transformar las imágenes en `arrays` de `NumPy` para poder manejarlas de la misma manera que hemos hecho hasta ahora en los post anteriores. También separaremos un conjunto de imágenes para validación así como un subset de imágenes para entrenar más rápido.

Python

```python
import numpy as np

train_images = np.array([np.array(img) for img, label in trainset])
X_test = np.array([np.array(img) for img, label in testset])

train_labels = np.array([label for img, label in trainset])
y_test = np.array([label for img, label in testset])

X_train, X_val, X_subset = train_images[:40000], train_images[40000:],
    train_images[:5000]
y_train, y_val, y_subset = train_labels[:40000], train_labels[40000:],
    train_labels[:5000]

X_train.shape, X_val.shape, X_test.shape, X_subset.shape
```

Devuelve: *((40000, 32, 32, 3), (10000, 32, 32, 3), (10000, 32, 32, 3), (5000, 32, 32, 3))*

Ahora, podemos entrenar un `MLP` en esta tarea y ver las curvas de entrenamiento.

Python

```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, X, Y):
        self.X = torch.from_numpy(X / 255.).float().cuda().view(-1,
            32*32*3)
        self.Y = torch.from_numpy(Y).long().cuda()
    def __len__(self):
        return len(self.X)
    def __getitem__(self, ix):
        return self.X[ix], self.Y[ix]

dataset = {
    'train': Dataset(X_subset, y_subset),
    'val': Dataset(X_val, y_val),
}

dataloader = {
    'train': torch.utils.data.DataLoader(dataset['train'], batch_size=32,
         shuffle=True),
    'val': torch.utils.data.DataLoader(dataset['val'], batch_size=1000,
        shuffle=False)
}

len(dataset['train']), len(dataset['val'])
```

Devuelve: *(5000, 10000)*

Python

```
from sklearn.metrics import accuracy_score

def softmax(x):
    return torch.exp(x) / torch.exp(x).sum(axis=-1,keepdims=True)

def build_model(D_in=32*32*3, H=100, D_out=10):
    return torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out)
    ).cuda()

def fit(model, dataloader, epochs=100, log_each=10, weight_decay=0):
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1, weight_decay=
        weight_decay)
    l, acc = [], []
    val_l, val_acc = [], []
    for e in range(1, epochs+1):
        _l, _acc = [], []
        model.train()
```

```
        for x_b, y_b in dataloader['train']:
            y_pred = model(x_b)
            loss = criterion(y_pred, y_b)
            _l.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            y_probas = torch.argmax(softmax(y_pred), axis=1)
            _acc.append(accuracy_score(y_b.cpu().numpy(), y_probas.cpu().
                detach().numpy()))
        l.append(np.mean(_l))
        acc.append(np.mean(_acc))
        model.eval()
        _l, _acc = [], []
        with torch.no_grad():
            for x_b, y_b in dataloader['val']:
                y_pred = model(x_b)
                loss = criterion(y_pred, y_b)
                _l.append(loss.item())
                y_probas = torch.argmax(softmax(y_pred), axis=1)
                _acc.append(accuracy_score(y_b.cpu().numpy(), y_probas.
                    cpu().numpy()))
        val_l.append(np.mean(_l))
        val_acc.append(np.mean(_acc))
        if not e % log_each:
            print(f"Epoch {e}/{epochs} loss {l[-1]:.5f} acc {acc[-1]:.5f}
                val_loss {val_l[-1]:.5f} val_acc {val_acc[-1]:.5f}")
    return {'epoch': list(range(1, epochs+1)), 'loss': l, 'acc': acc, '
        val_loss': val_l, 'val_acc': val_acc}
```

Python

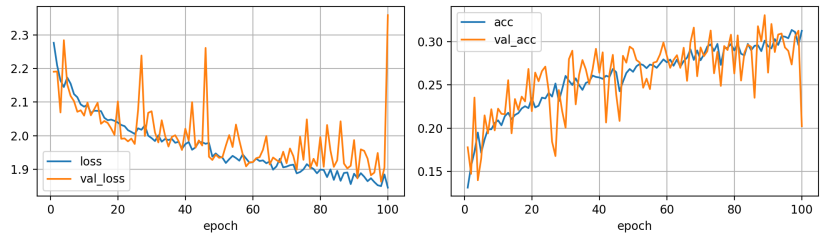
```
model = build_model()
hist = fit(model, dataloader)
```

Devuelve:

Epoch 10/100 loss 1.70773 acc 0.38376 val_loss 2.03552 val_acc 0.28730 Epoch 20/100 loss 1.48095 acc 0.45900 val_loss 2.81445 val_acc 0.24080 Epoch 30/100 loss 1.28839 acc 0.53065 val_loss 2.27806 val_acc 0.32300 Epoch 40/100 loss 1.13749 acc 0.58439 val_loss 3.08324 val_acc 0.26360 Epoch 50/100 loss 1.00819 acc 0.63137 val_loss 2.86893 val_acc 0.33720 Epoch 60/100 loss 0.87316 acc 0.69029 val_loss 3.59225 val_acc 0.28960 Epoch 70/100 loss 0.73331 acc 0.73846 val_loss 2.83442 val_acc 0.34650 Epoch 80/100 loss 0.67347 acc 0.76393 val_loss 6.86735 val_acc 0.24730 Epoch 90/100 loss 0.50357 acc 0.81986 val_loss 4.08476 val_acc 0.32820 Epoch 100/100 loss 0.57434 acc 0.80693 val_loss 3.87901 val_acc 0.37890

Python

```python
import pandas as pd

fig = plt.figure(dpi=200, figsize=(10,3))
ax = plt.subplot(121)
pd.DataFrame(hist).plot(x='epoch', y=['loss', 'val_loss'], grid=True, ax=
    ax)
ax = plt.subplot(122)
pd.DataFrame(hist).plot(x='epoch', y=['acc', 'val_acc'], grid=True, ax=ax
    )
plt.tight_layout()
plt.savefig('resources/reg2.png')
plt.close()
```



Como podemos observar de las curvas de entrenamiento, las métricas para el conjunto de datos de entrenamiento van mejorando *epoch* tras *epoch*. Sin embargo, las métricas de validación rápidamente se estancan y empiezan a empeorar. Esta es la señal clara de que nuestro modelo está sufriendo de *overfitting*, y en las siguientes secciones veremos diferentes técnicas de **regularización**, el término técnico utilizado para referirnos a las diferentes técnicas de reducción de *overfitting*.

Regularización L2

Una técnica de regularización muy utilizada es la regularización L2. Esta técnica consiste en restringir la magnitud de los pesos de la `red neuronal` forzándolos a ser valores pequeños. A efectos prácticos, la regularización L2 se implementa como un término extra en la función de pérdida:

$$l = CE(\hat{y}, y) + \alpha \frac{1}{2} ||\mathbf{w}||$$

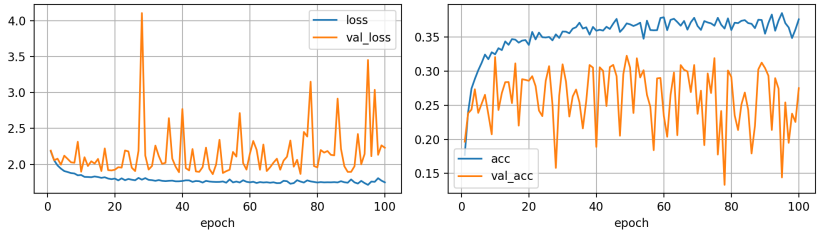
Donde $CE(\hat{y}, y)$ es la función *cross entropy* (o cualquier función de pérdida que uses dependiendo de tu caso), $\mathbf{w}$ son los pesos de la red y α es un parámetro indicando cuánto queremos regularizar el modelo. En `Pytorch` podemos asignar el valor de α mediante el parámetro `wieght_decay` directamente en el optimizador (por defecto está a 0). Valores típicos utilizados se encuentran en el rango 0.001 - 0.01.

Python

```
model = build_model()
hist = fit(model, dataloader, weight_decay=0.01)
```

Devuelve:

Mejor modelo guardado con acc 0.19660 en epoch 1 Mejor modelo guardado con acc 0.23820 en epoch 2 Mejor modelo guardado con acc 0.24370 en epoch 3 Mejor modelo guardado con acc 0.27350 en epoch 4 Mejor modelo guardado con acc 0.32080 en epoch 10 Epoch 10/100 loss 1.85092 acc 0.32484 val_loss 1.89685 val_acc 0.32080 Epoch 20/100 loss 1.80120 acc 0.33857 val_loss 1.92479 val_acc 0.28600 Epoch 30/100 loss 1.78338 acc 0.35828 val_loss 1.92711 val_acc 0.31010 Epoch 40/100 loss 1.76777 acc 0.35947 val_loss 2.77026 val_acc 0.18890 Mejor modelo guardado con acc 0.32270 en epoch 49 Epoch 50/100 loss 1.75399 acc 0.36883 val_loss 1.99816 val_acc 0.30530 Epoch 60/100 loss 1.74859 acc 0.37918 val_loss 2.14679 val_acc 0.23450 Epoch 70/100 loss 1.77016 acc 0.36644 val_loss 2.05172 val_acc 0.23750 Epoch 80/100 loss 1.74625 acc 0.36047 val_loss 1.95698 val_acc 0.29160 Epoch 90/100 loss 1.78897 acc 0.35510 val_loss 1.89476 val_acc 0.30460 Epoch 100/100 loss 1.74833 acc 0.37619 val_loss 2.22997 val_acc 0.27510



Como puedes observar, ahora las curvas de entrenamiento y validación están más próximas, por lo que hemos disminuido el *overfitting*. Es importante remarcar que reducir el *overfitting* no implica mejorar las métricas.

Early Stopping

Otra técnica muy utilizada para regularizar un modelo es el *early stopping*. Esta técnica consiste en llevar un registro de las métricas de validación durante el entrenamiento, guardar los pesos del modelo cada vez que mejoramos las mejores métricas obtenidas y, una vez terminado el entrenamiento, cargar los mejores pesos en vez de quedarnos con los últimos. De forma opcional también podemos detener el entrenamiento si no mejoramos nuestras métricas durante un número determinado de *epochs* seguidas, lo cual puede traducirse en un ahorro de tiempo y cómputo.

Python

```
def fit(model, dataloader, epochs=100, log_each=10, weight_decay=0,
    early_stopping=0):
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1, weight_decay=
        weight_decay)
    l, acc = [], []
    val_l, val_acc = [], []
    best_acc, step = 0, 0
    for e in range(1, epochs+1):
        _l, _acc = [], []
        model.train()
        for x_b, y_b in dataloader['train']:
            y_pred = model(x_b)
            loss = criterion(y_pred, y_b)
            _l.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            y_probas = torch.argmax(softmax(y_pred), axis=1)
            _acc.append(accuracy_score(y_b.cpu().numpy(), y_probas.cpu().
                detach().numpy()))
        l.append(np.mean(_l))
        acc.append(np.mean(_acc))
        model.eval()
        _l, _acc = [], []
        with torch.no_grad():
            for x_b, y_b in dataloader['val']:
                y_pred = model(x_b)
                loss = criterion(y_pred, y_b)
                _l.append(loss.item())
                y_probas = torch.argmax(softmax(y_pred), axis=1)
```

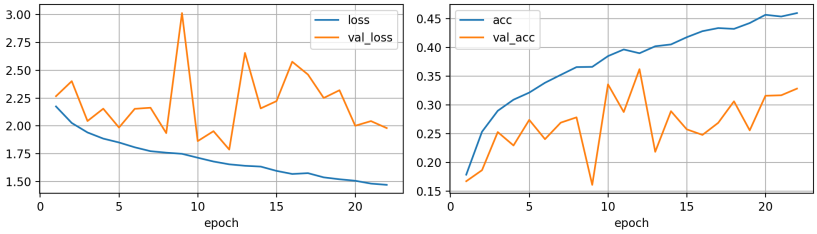
```
                _acc.append(accuracy_score(y_b.cpu().numpy(), y_probas.
                    cpu().numpy()))
        val_l.append(np.mean(_l))
        val_acc.append(np.mean(_acc))
        # guardar mejor modelo
        if val_acc[-1] > best_acc:
            best_acc = val_acc[-1]
            torch.save(model.state_dict(), 'ckpt.pt')
            step = 0
            print(f"Mejor modelo guardado con acc {best_acc:.5f} en epoch
                {e}")
        step += 1
        # parar
        if early_stopping and step > early_stopping:
            print(f"Entrenamiento detenido en epoch {e} por no mejorar en
                {early_stopping} epochs seguidas")
            break
        if not e % log_each:
            print(f"Epoch {e}/{epochs} loss {l[-1]:.5f} acc {acc[-1]:.5f}
                val_loss {val_l[-1]:.5f} val_acc {val_acc[-1]:.5f}")
    # cargar mejor modelo
    model.load_state_dict(torch.load('ckpt.pt'))
    return {'epoch': list(range(1, len(l)+1)), 'loss': l, 'acc': acc, '
        val_loss': val_l, 'val_acc': val_acc}
```

Python

```
model = build_model()
hist = fit(model, dataloader, early_stopping=10)
```

Devuelve:

Mejor modelo guardado con acc 0.16750 en epoch 1 Mejor modelo guardado con acc 0.18670 en epoch 2 Mejor modelo guardado con acc 0.25300 en epoch 3 Mejor modelo guardado con acc 0.27420 en epoch 5 Mejor modelo guardado con acc 0.27860 en epoch 8 Mejor modelo guardado con acc 0.33600 en epoch 10 Epoch 10/100 loss 1.71431 acc 0.38515 val_loss 1.86365 val_acc 0.33600 Mejor modelo guardado con acc 0.36240 en epoch 12 Epoch 20/100 loss 1.50740 acc 0.45701 val_loss 2.00147 val_acc 0.31620 Entrenamiento detenido en epoch 22 por no mejorar en 10 epochs seguidas



Si comparas esta figura con la primera, podrás observar que si seguimos entrenando más *epochs* no vamos a encontrar un modelo mejor, por lo tanto no es necesario seguir entrenando, ya que en este punto hemos encontrado el mejor modelo posible dadas las circunstancias. *Early stopping* es una técnica muy sencilla y efectiva, pero en función del modelo, datos e hiperparámetros puede que nos dé problemas. Me ha pasado más de una vez que si dejas un modelo entrenar suficiente tiempo, acaba mejorando, incluso si no lo ha hecho en las 100 epochs anteriores. Por lo tanto, mi recomendación es no usar *early stopping* a nos ser que tus curvas de aprendizaje sean extremadamente suaves.

Dropout

Otra técnica muy popular para reducir el *overfitting* es el uso de *Dropout*. Esta técnica se implementa como una capa extra en nuestra `red neuronal` cuyo objetivo es, durante el entrenamiento, «apagar» de manera aleatoria algunas neuronas para forzar a nuestro modelo a aprender diferentes caminos dentro de la arquitectura para representar los mismos datos.

Python

```
    def build_model(D_in=32*32*3, H=100, D_out=10, p=0):
        return torch.nn.Sequential(
            torch.nn.Linear(D_in, H),
            torch.nn.ReLU(),
            torch.nn.Dropout(p),
            torch.nn.Linear(H, H),
            torch.nn.ReLU(),
            torch.nn.Dropout(p),
            torch.nn.Linear(H, D_out)
        ).cuda()
```

Puedes añadir una capa de *Dropout* con la clase `torch.nn.Dropout`. El parámetro `p` controla la probabilidad de «apagar» una neurona. Este valor coincide con la proporción (en promedio) de neuronas que serán anuladas en una capa determinada.

> Cuando no estamos entrenando, generalmente no queremos que la capa *Dropout* esté activa de manera que utilicemos todas las neuronas en el modelo. Podemos controlar el modo en el que trabaja la capa llamando a las funciones `model.train()` y `model.eval()`, las cuales se encargarán de asignar el modo adecuado a la capa *Dropout* (y cualquier otra capa de tu red que también tenga diferente comportamiento según estemos entrenando o evaluado el modelo). Esto es muy importante y fuente común de errores al trabajar con `Pytorch`.

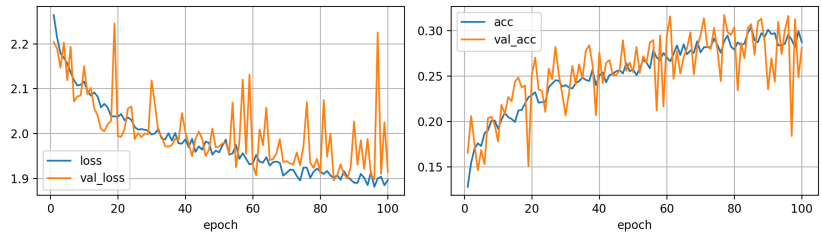
Python

```
model = build_model(p=0.5)
hist = fit(model, dataloader)
```

Devuelve:

Mejor modelo guardado con acc 0.16550 en epoch 1 Mejor modelo guardado con acc 0.20600 en epoch 2 Epoch 10/100 loss 2.11585 acc 0.19208 val_loss 2.15042 val_acc 0.17790 Mejor modelo guardado con acc 0.21830 en epoch 11 Mejor modelo guardado con acc 0.22690 en epoch 13 Mejor modelo guardado con acc 0.24380 en epoch 15 Mejor modelo guardado con acc 0.24850 en epoch 16 Mejor modelo guardado con acc 0.25140 en epoch 20 Epoch 20/100 loss 2.03782 acc 0.22910 val_loss 1.99427 val_acc 0.25140 Mejor modelo guardado con acc 0.27040 en epoch 21 Mejor modelo guardado con acc 0.28220 en epoch 27 Epoch 30/100 loss 1.99778 acc 0.24005 val_loss 2.11850 val_acc 0.20690 Mejor modelo guardado con acc 0.28400 en epoch 37 Epoch 40/100 loss 1.98718 acc 0.25060 val_loss 2.00350 val_acc 0.27620 Mejor modelo guardado con acc 0.28950 en epoch 47 Epoch 50/100 loss 1.95807 acc 0.25756 val_loss 1.97111 val_acc 0.25440 Mejor modelo guardado con acc 0.28980 en epoch 56 Mejor modelo guardado con acc 0.29480 en epoch 58 Epoch 60/100 loss 1.93355 acc 0.27070 val_loss 1.92971 val_acc 0.29440 Mejor modelo guardado con acc 0.31580 en epoch 61 Epoch 70/100 loss 1.91263 acc 0.27627 val_loss 1.93859 val_acc 0.29500 Mejor modelo guardado con acc 0.31730 en epoch 77 Epoch 80/100 loss 1.91551 acc 0.27906 val_loss 1.91222 val_acc 0.30380 Epoch 90/100 loss 1.89049 acc 0.30115 val_loss 2.02714 val_acc 0.23550

Epoch 100/100 loss 1.89647 acc 0.28742 val_loss 1.91496 val_acc 0.28130



De nuevo, observamos que las curvas de entrenamiento ya no presentan *overfitting*. Puedes jugar con la probabilidad de *Dropout* para controlar su efecto en el resultado final. De nuevo, date cuenta de que reducir el *overfitting* no implica necesariamente que nuestro modelo tenga mejores prestaciones, simplemente estamos forzando al modelo a no «aprender de memoria» el dataset de entrenamiento.

Usar más datos

La mejor manera de reducir el *overfitting* es usando más datos. Esto es un hecho obvio, pero no sencillo de conseguir, ya que obtener un mayor dataset no es siempre factible y normalmente ya utilizamos todos los datos a nuestra disposición. Sin embargo, a diferencia del resto de estrategias presentadas hasta ahora, esta no solo reducirá el *overfitting*, sino que también mejorará las prestaciones de nuestro modelo.

> Si te fijas en las empresas más TOP en Inteligencia Artificial, todas ellas han conseguido su ventaja competitiva gracias a la cantidad (y calidad) de los datos que han podido generar. Es posible caer en el error de creer que por descargar un dataset de internet vamos a poder entrenar modelos que marquen la diferencia, pero la realidad es que sin una fuerte inversión en recogida, generación y etiquetado de datos, no se puede competir.

Python

```
# ahora sí que usamos todos los datos

dataset = {
    'train': Dataset(X_train, y_train),
    'val': Dataset(X_val, y_val),
}

dataloader = {
    'train': torch.utils.data.DataLoader(dataset['train'], batch_size=32,
         shuffle=True),
    'val': torch.utils.data.DataLoader(dataset['val'], batch_size=1000,
        shuffle=False)
}

len(dataset['train']), len(dataset['val'])
```

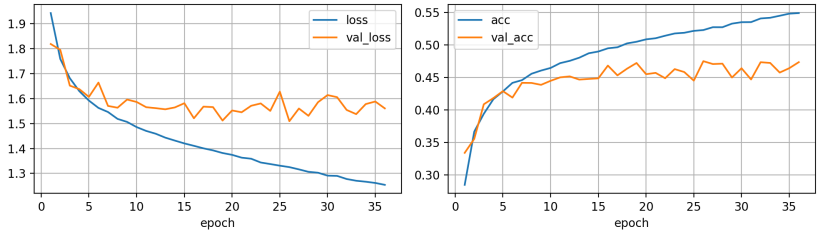
Devuelve: *(40000, 10000)*

Python

```
model = build_model()
hist = fit(model, dataloader, early_stopping=10)
```

Devuelve:

Mejor modelo guardado con acc 0.33400 en epoch 1 Mejor modelo guardado con acc 0.35550 en epoch 2 Mejor modelo guardado con acc 0.40870 en epoch 3 Mejor modelo guardado con acc 0.41820 en epoch 4 Mejor modelo guardado con acc 0.42890 en epoch 5 Mejor modelo guardado con acc 0.44170 en epoch 7 Mejor modelo guardado con acc 0.44480 en epoch 10 Epoch 10/100 loss 1.48558 acc 0.46460 val_loss 1.58620 val_acc 0.44480 Mejor modelo guardado con acc 0.45010 en epoch 11 Mejor modelo guardado con acc 0.45160 en epoch 12 Mejor modelo guardado con acc 0.46820 en epoch 16 Mejor modelo guardado con acc 0.47220 en epoch 19 Epoch 20/100 loss 1.37447 acc 0.50857 val_loss 1.55218 val_acc 0.45500 Mejor modelo guardado con acc 0.47500 en epoch 26 Epoch 30/100 loss 1.29126 acc 0.53497 val_loss 1.61378 val_acc 0.46410 Entrenamiento detenido en epoch 36 por no mejorar en 10 epochs seguidas



Como puedes observar, la precisión de nuestro modelo ha aumentado considerablemente gracias al uso de más datos. De esta manera, aprovechamos más capacidad del modelo.

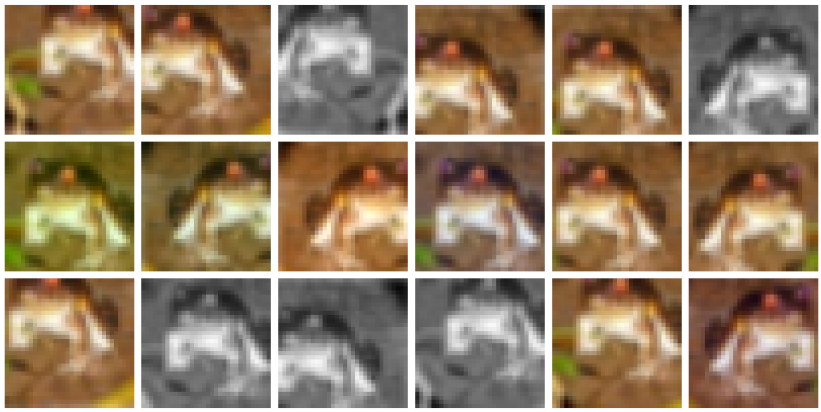
Data Augmentation

Si bien conseguir más datos no siempre es factible, podemos llevar a cabo una simple idea: en vez de darle al modelo siempre nuestras imágenes en la misma manera, vamos a aplicar transformaciones de manera aleatoria de forma que, efectivamente, nuestro modelo no vea nunca la misma imagen dos veces consiguiendo así un dataset potencialmente infinito. Estas transformaciones deben alterar la imagen lo suficiente como para poder considerarla como una muestra diferente, pero no tanto como para alterar su etiqueta. Algunos ejemplos de estas transformaciones incluyen: giros, recortes, alteraciones de color, brillo o tamaño, etc. Si bien es cierto que esta técnica está enfocada en aplicaciones de visión por computador su uso no está tan extendido en otros campos como el del procesamiento de lenguaje natural.

Existen muchas herramientas disponibles para aplicar transformaciones a imágenes (por ejemplo, la misma librería de `torchvision` incluye algunas). Una elección popular es la librería `albumentations`.

Python

```python
from albumentations import Compose, RandomCrop, Resize, HorizontalFlip,
    ToGray, RGBShift, OneOf

trans = Compose([
    RandomCrop(24,24),
    Resize(32, 32),
    HorizontalFlip(),
    OneOf([
        ToGray(p=0.2),
        RGBShift(p=0.3)
    ])
])

idx = 0
r, c = 3, 6
plt.figure(figsize=(c*3, r*3))
for row in range(r):
    for col in range(c):
        ix = c*row + col
        plt.subplot(r, c, ix + 1)
        img, label = trainset[idx]
        # apply transformation
        img = trans(image=np.array(img))["image"]
        plt.imshow(img)
        plt.axis('off')
plt.subplots_adjust(wspace=0.1, hspace=0.2)
plt.tight_layout()
plt.savefig('resources/data_aug1.png')
plt.close()
```



En la figura anterior estás viendo una única muestra de nuestro dataset a la que hemos aplicado una serie de transformaciones que han alterado su apariencia sin modificar su etiqueta. Aplicando estas transformaciones aleatorias a cada muestra antes de dársela a la `red neuronal` obligará a nuestro a modelo a generalizar mucho más, reduciendo el *overfitting*.

Python

```
    class Dataset(torch.utils.data.Dataset):
        def __init__(self, X, Y, trans=None):
            self.X = X
            self.Y = Y
            self.trans = trans
        def __len__(self):
            return len(self.X)
        def __getitem__(self, ix):
            img = self.X[ix]
            if self.trans:
                img = trans(image=img)["image"]
            img = torch.from_numpy(img / 255.).float().cuda().view(-1)
            label = torch.tensor(self.Y[ix]).long().cuda()
            return img, label

    dataset = {
        'train': Dataset(X_train, y_train, trans=trans),
        'val': Dataset(X_val, y_val),
    }

    dataloader = {
        'train': torch.utils.data.DataLoader(dataset['train'], batch_size=32,
             shuffle=True),
        'val': torch.utils.data.DataLoader(dataset['val'], batch_size=1000,
            shuffle=False)
    }

    len(dataset['train']), len(dataset['val'])
```

Devuelve: *(40000, 10000)*

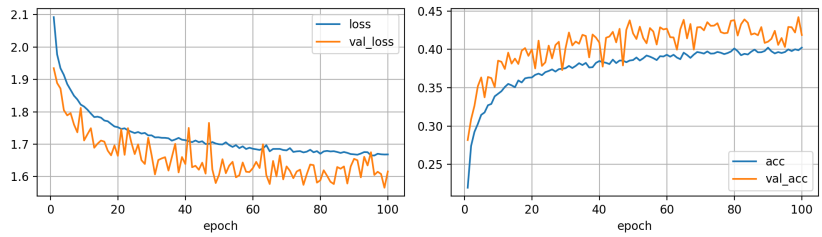
Python

```
    model = build_model()
    hist = fit(model, dataloader)
```

Devuelve:

Mejor modelo guardado con acc 0.28170 en epoch 1 Mejor modelo guardado con acc 0.30870 en epoch 2 Mejor modelo guardado con acc 0.32640 en epoch 3 Mejor modelo

guardado con acc 0.35190 en epoch 4 Mejor modelo guardado con acc 0.36340 en epoch 5 Mejor modelo guardado con acc 0.36400 en epoch 7 Mejor modelo guardado con acc 0.38530 en epoch 10 Epoch 10/100 loss 1.81602 acc 0.34210 val_loss 1.71155 val_acc 0.38530 Mejor modelo guardado con acc 0.39570 en epoch 13 Mejor modelo guardado con acc 0.39830 en epoch 17 Mejor modelo guardado con acc 0.40170 en epoch 18 Epoch 20/100 loss 1.75249 acc 0.36335 val_loss 1.66397 val_acc 0.39990 Mejor modelo guardado con acc 0.41140 en epoch 22 Epoch 30/100 loss 1.72727 acc 0.37543 val_loss 1.66641 val_acc 0.40180 Mejor modelo guardado con acc 0.42200 en epoch 31 Epoch 40/100 loss 1.71244 acc 0.38480 val_loss 1.63871 val_acc 0.40900 Mejor modelo guardado con acc 0.42290 en epoch 44 Mejor modelo guardado con acc 0.42680 en epoch 46 Mejor modelo guardado con acc 0.43790 en epoch 49 Epoch 50/100 loss 1.69970 acc 0.38610 val_loss 1.60588 val_acc 0.42090 Epoch 60/100 loss 1.68590 acc 0.39300 val_loss 1.62601 val_acc 0.42710 Mejor modelo guardado con acc 0.43860 en epoch 65 Epoch 70/100 loss 1.68091 acc 0.39575 val_loss 1.63135 val_acc 0.42910 Epoch 80/100 loss 1.67046 acc 0.40125 val_loss 1.58859 val_acc 0.43820 Mejor modelo guardado con acc 0.43910 en epoch 83 Epoch 90/100 loss 1.66857 acc 0.40227 val_loss 1.65470 val_acc 0.41010 Mejor modelo guardado con acc 0.44210 en epoch 99 Epoch 100/100 loss 1.66833 acc 0.40232 val_loss 1.61562 val_acc 0.41880



El uso de *data augmentation* requerirá de ciclos de entrenamiento más largos para poder aprovechar toda la capacidad que las transformaciones nos ofrecen. Una característica interesante de esta técnica es que podemos conseguir que las curvas de aprendizaje de validación estén por encima de las de entrenamiento, indicando que nuestro modelo realmente trabaja mejor en datos no vistos (lo cual no significa necesariamente que el modelo sea mejor. En este caso, por ejemplo, nuestras métricas no son tan buenas como en el caso anterior).

> Si crees que los modelos entrenados en esta sección no son muy buenos (lo cual es cierto), te reto a utilizar las técnicas de *transfer learning* explicadas anteriormente para mejorar drásticamente los resultados.

Receta de entrenamiento

En este capítulo hemos introducido diferentes técnicas para entrenar tus modelos de manera efectiva, pero si quieres una receta paso a paso para entrenar tus modelos, te recomiendo que le eches un vistazo al siguiente post: `juansensio.com/blog/033_receta_entrenamiento`.

7. Epílogo

Pues hasta aquí mi libro. ¡Menuda odisea! En el primer capítulo hemos aprendido qué es la IA, qué aplicaciones tiene y por qué es importante aprender a trabajar con ella. Para ello, saber programar, especialmente con el lenguaje Python, es esencial y lo que hemos aprendido en el capítulo 2. Si bien podríamos usar solo Python para hacer nuestros modelos, existen librerías que ya implementan todo lo necesario para diseñar, entrenar y desplegar modelos de Aprendizaje Profundo de forma óptima. De entre ellas, Pytorch es mi opción preferida, y la que hemos aprendido a usar en el tercer capítulo. En el cuarto capítulo, nos hemos adentrado en las entrañas del Aprendizaje Profundo para entender cómo y por qué funciona, y hemos terminado presentando las arquitecturas de redes neuronales más comunes hoy en día (capítulo 5), de entre las cuales el *Transformer* es sin duda el rey indiscutible en el momento de escribir estas páginas, así como técnicas esenciales para entrenar modelos de Aprendizaje Profundo de manera efectiva (capítulo 6).

Espero de verdad que todo el contenido te sea de utilidad. Como escribí al principio, he escrito este libro en parte para tener recogido en un solo lugar todo lo que me hubiese gustado saber cuando me adentré por primera vez en este mundo tan apasionante que es la Inteligencia Artificial y el Aprendizaje Profundo, así como para tener una referencia a mano, la cual pueda consultar en cualquier momento para refrescar la memoria. Estos conceptos me han sido útiles durante mi carrera profesional, tanto como desarrollador de IA como emprendedor, y creo que también te lo serán a ti.

¿Y ahora qué?

Que este libro lleve la palabra *Introducción* en el título no es casualidad, y es que el objetivo es precisamente el de presentar los conceptos básicos y fundamentos de la IA y el Aprendizaje Profundo de una manera accesible para que cualquiera pueda entenderlos. Por ello, es muy probable que se te quede algo corto en cuanto te adentres en profundidad en aspectos más avanzados. Estos son algunos de los temas que te recomiendo para seguir aprendiendo:

- **Aprendizaje Profundo en Visión por Computador**: Si bien hemos hablado sobre redes neuronales convolucionales y su uso en clasificación de imágenes, la visión por computador es un campo más amplio que abarca desde la clasificación de imágenes hasta la detección de objetos, la segmentación, generación de imágenes y videos, etc. Además, el uso de la arquitectura *Transformer* en este campo es cada vez más común, dando lugar a nuevos modelos como *ViT* o *Swin Transformer*. Este campo es vital en aplicaciones como la conducción autónoma y robótica (un campo que personalmente me apasiona y que tiene pinta que en un futuro cercano vaya a revolucionarlo todo en nuestras vidas).
- **Aprendizaje Profundo en Procesamiento de Lenguaje Natural**: De nuevo, en este libro hemos introducido la arquitectura *Transformer* e implementado un simple ejemplo para generación de texto, pero hay mucha más tela que cortar con respecto al NLP. Si bien la aparición de los grandes modelos de languaje (*LLMs* por sus siglas en inglés) han simplificado en gran medida la ingeniería de modelos de lenguaje, entrenar este tipo de modelos está al alcanze de pocos. Aún así, es posible usarlos para crear modelos más pequeños y ajustados a nuestras necesidades, lo cual parece ser una buena estrategia durante los próximos años.
- **Aprendizaje no supervisado**: Todos los ejemplos presentados aquí han usado conjuntos de datos etiquetados para entrenar modelos de manera supervisada. Sin embargo la cantidad de datos etiquetados disponibles palidece en comparación con la cantidad de datos no etiquetados en el mundo. Poder usarlos para generar buenos modelos pre-entrenados, conocidos como modelos fundacionales o *foundation models*, ha sido parte de la calve del éxito y auge de los *LLMs*. Además, el aprendizaje no supervisado puede usarse para la creación de modelos del mundo (*world models*), en los cuales la IA puede aprender de manera autónoma simplemente observando su entorno para luego ser ajustada con muy pocos datos a nuevas tareas de manera efectiva.
- **Aprendizaje por refuerzo y robótica**: El aprendizaje por refuerzo es un campo de la IA que tiene mucho que oferecer. Su combinación con el Aprendizaje Profundo dió lugar a avances espectaculares como los presentados por DeepMind con sus sistema AlphaGo y sigue haciendolo hoy en día con aplicaciones científicas como el plegado de proteínas para el desarrollo de nuevos medicamentos o el control de plasma en reactor de fusión nuclear. Además, es un paso clave en el desarrollo de los asistentes más avanzado como ChatGPT y compañía. Su uso en robótica también es esencial a la hora de desarrollar sistemas de conducción autónoma, robots humanoides, etc. Un campo apasionante, del que no hemos hablado en este libro, pero que te recomiendo

que tengas en tu radar.

- **Aprendizaje Profundo en producción (MLOps)**: Todo lo que hemos presentado en este libro ha estado centrado en el desarrollo y entrenamiento de modelos de Aprendizaje Profundo, pero una vez estos modelos están listos para trabajar en el mundo real se abre una nueva puerta a un mundo lleno de complejidades e imprevistos. ¿Cómo servimos un modelo de manera efectiva?, ¿cómo podemos estar seguros que el modelo está funcionando como es debido?, ¿cómo podemos mejorar el modelo y desplegar nuevas versiones de manera contínua? Todos estos conceptos se engloban dentro del término *MLOps*, y se podrían escribir libros enteros solo sobre ellos.
- **Full Stack Deep Learning**: Otro tema muy interesante es el del desarrollo de aplicaciones alrededor de los modelos de Aprendizaje Profundo. Páginas webs, aplicaciones móviles, videojuegos… han ido evolucionando bajo un paradigma que la IA está cambiando. Las interfaces de tipo chat, como ChatGPT, son un ejemplo de disrupción en el que quizás es tan importante el modelo que hay detrás como la interfaz de usuario para interactuar con el modelo. El desarrollo de productos de IA es un tema en sí mismo y saber aprovechar las nuevas capacidades que la IA trae puede suponer una gran ventaja en el panorama actual.

Quién sabe, si este libro va bien quizás me animo a adentrarme en alguno de estos temas en futuras entregas. Si es algo que te interesa, asegúrate de hacerme llegar tu petición a través de los diferentes canales disponibles:

- Twitter/X: `https://twitter.com/juansensio`
- Discord: `https://discord.gg/aTeEKXzKbs`
- Youtube: `https://www.youtube.com/@juansensio`
- Web: `https://juansensio.com/`

En cualquier caso, te recomiendo que le eches un vistazo a https://www.elcursodeia.com/ donde encontrás mucho más conocimiento para seguir avanzando en tu carrera en Inteligencia Artificial. ¡Un abrazo!

8. Agradecimientos

En primer lugar, me gustaría agradecer al equipo de la editorial Savvily por la confianza depositada en mí. Si no fuese por su iniciativa y apoyo durante el proceso, este libro no hubiese sido posible. En especial a María y Carlos por contactar conmigo y darme la oportunidad de escribir este libro. También quiero agradecer a Liset por su ayuda en la edición y revisión del manuscrito, así como a Maxi por su ayuda encontrando los últimos detalles para corregir.

Gran parte del contenido de este libro es fruto del trabajo realizado desde 2020 y que he ido publicando en mi blog y canal de Youtube, por lo que en última instancia este libro está dedicado a todos aquellos que me han seguido, acompañado y apoyado en este camino.

Savvily

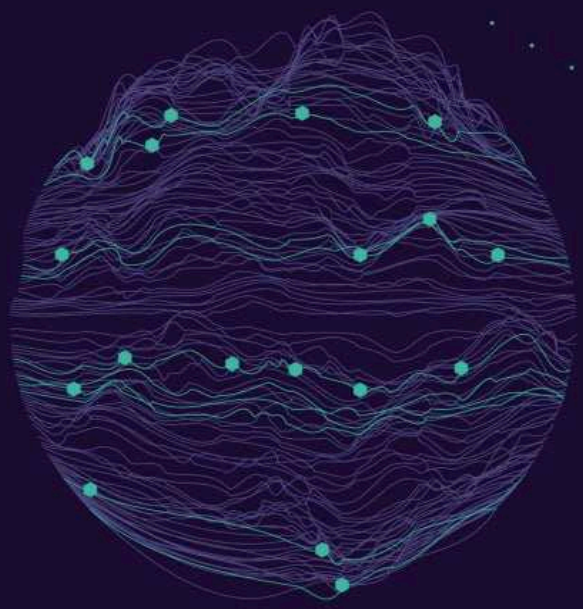
¡Enhorabuena por leer hasta aquí! Agradecemos el tiempo que te has tomado estudiando este libro y que lo recomiendes a cualquier persona que pienses que le puede ser de ayuda. Escanea el código QR y accede a la web de la editorial, donde podrás encontrar información sobre nosotros y nuestros datos de contacto. Una vez nos escribas un *email* y adjuntes un justificante de compra, podremos darte de alta en la comunidad y disfrutarás de todos los beneficios.

Estaría genial que envíes una valoración del libro por *email*, una reseña de lo que te ha parecido. Si has encontrado erratas, por favor envíalas también por *email* y con gusto las arreglaremos en futuras revisiones y ediciones del libro. Para publicar libros como este, se necesita todo un equipo multidisciplinar, desde las autoras y autores, al equipo de revisión, pasando por diseñadoras y especialistas en marketing digital. Por todo ello, te pedimos que, si el libro te ha gustado y no lo habías comprado, te animes a comprarlo. Es el mejor reconocimiento a la labor de todas las personas que lo han hecho posible. Si eres docente en una institución académica y quieres utilizar este libro en tus clases, Savvily ofrece acuerdos para docentes y estudiantes con precios exclusivos en todos nuestros libros y cursos. Por favor, escríbenos y cuéntanos tu caso, preséntanos a tu organización y te enviaremos más información.

¿Te has planteado alguna vez escribir tu propio libro? Seguro que tienes mucho que contar, ¿te animas? Hacen falta buenos libros en castellano. Visita nuestra web y descubre cómo puedes convertirte en autora o autor, estaremos encantados de acompañarte.

Muchas gracias de parte de todo el equipo de Savvily.

Contacto: contacto@savvily.es



Este libro es una introducción a la Inteligencia Artificial con Python. Está pensado para estudiantes, desarrolladores y otros profesionales que quieran adentrarse en el campo de la IA, y en el Aprendizaje Profundo (Deep Learning) en concreto. Para ello, en el libro se aprende, desde cero, cómo programar con Python y, a partir de ahí, cómo usar la librería Pytorch para crear y entrenar redes neuronales. Se presentan arquitecturas de redes neuronales populares y consejos para su entrenamiento de manera efectiva.

savvily

Conocimiento que compartir