FUNDAMENTOS DE VISIÓN POR COMPUTADOR UTILIZANDO APRENDIZAJE PROFUNDO

DIEGO RENZA TORRES - DORA MARÍA BALLESTEROS

INVESTIGACIÓN EDUCATIVA & PEDAGÓGICA IBEROAMERICANA

editorial redipe

editorial redipe © 2023

Título original: Fundamentos de visión por computador utilizando aprendizaje profundo

Autor: Diego Renza Torres & Dora María Ballesteros

ISBN: 978-1-957395-32-6

Primera edición, Diciembre 2023

SELLO Editorial

Editorial REDIPE (95857440), Nueva York – Cali
Red de Pedagogía S.A.S. NIT: 900460139-2

Coeditor: Fundación Yunis: Tejiendo nuevos sentidos con las personas con discapacidad

Red Iberoamericana de Pedaogía
editorial@rediberoamericanadepedagogia.com
www.redipe.org

Impreso en Cali, Colombia
Printed in Cali, Colombia

DIEGO RENZA

Docente Asociado de la Universidad Militar Nueva Granada, vinculado al Programa de Ingeniería en Telecomunicaciones desde el año 2014. Es Ingeniero Electrónico de la Universidad Surcolombiana, con Maestría en Ingeniería en Telecomunicaciones de la Universidad Nacional de Colombia y Doctor en Computación Avanzada para Ciencias e Ingeniería de la Universidad Politécnica de Madrid (España). Sus áreas de conocimiento son el procesamiento de imágenes, visión por computador y comunicaciones inalámbricas. Cuenta con más de 40 artículos publicados en revistas nacionales e internacionales, y es co-autor de dos libros académicos. Este libro titulado "Fundamentos de visión por computador utilizando aprendizaje profundo", es resultado de su ejercicio académico como profesor de la Universidad Militar Nueva Granada.

DORA MARIA BALLESTEROS

Docente Titular de la Universidad Militar Nueva Granada, vinculada al Programa de Ingeniería en Telecomunicaciones desde el año 2007. Es Ingeniera Electrónica de la Universidad Industrial de Santander, con Maestría en Ingeniería Electrónica de la Universidad de los Andes y Doctora en Ingeniería Electrónica de la Universidad Politécnica de Cataluña (España). Sus áreas de conocimiento son el procesamiento digital de señales, aprendizaje automático e ingeniería de datos. Cuenta con más de 60 artículos publicados en revistas nacionales e internacionales, y es co-autora de tres libros académicos. Este libro titulado "Fundamentos de visión por computador utilizando aprendizaje profundo", es resultado de su ejercicio académico como profesor de la Universidad Militar Nueva Granada.

PRÓLOGO

Día a día las imágenes se han convertido en uno de los contenidos multimedia más transmitidos, almacenados y utilizados tanto por usuarios de tecnologías digitales, como por empresas tecnológicas, por ejemplo, Facebook, Amazon, Google e IBM. La información que contienen las imágenes permite, entre otras, identificar (catalogar) las personas presentes en una foto, el lugar donde fue tomada (ej. bosque, ciudad, desierto), así como la cantidad y tipo de objetos. Hasta hace un poco más de una década, la visión por computador se basaba principalmente en técnicas tradicionales de procesamiento de imágenes (ej. filtros, operaciones morfológicas y ecualización), pero desde el año 2012 (aproximadamente), se utilizan soluciones basadas en el aprendizaje profundo para clasificar, identificar objetos, reconocer patrones, mejorar la calidad de la imagen y generar nuevas imágenes.

Este libro titulado "Fundamentos de visión por computador utilizando aprendizaje profundo" introducirá al lector en el aprendizaje profundo para resolver problemas de análisis de imágenes, con un enfoque tanto teórico, como práctico. Está dividido en cinco capítulos y dos anexos. En el primer capítulo, se presenta una breve introducción a la temática de la visión por computador. En el capítulo 2, se explican algunos conceptos básicos del aprendizaje profundo. En el capítulo 3 y 4, se explican las redes neuronales y convolucionales, respectivamente. En el capítulo 5 se abordan arquitecturas clásicas de CNNs, así como la transferencia de aprendizaje para el diseño de modelos de visión por computador. Finalmente, en los Anexos 1 y 2, se explica el entorno de trabajo con Colaboratory, así como algunas de las funcionalidades de la librería de OpenCV.

Contenido

Figuras

Tablas

1. INTRODUCCIÓN A LA VISIÓN POR COMPUTADOR

En este capítulo abordaremos dos temáticas. En la primera, se contextualiza la visión por computador dentro de la inteligencia artificial, así como se mencionan las principales empresas tecnológicas que investigan en este tema. En la segunda parte, se relacionan algunos de los trabajos más representativos de la visión por computador moderna.

VISIÓN POR COMPUTADOR EN LA ACTUALIDAD

Actualmente, gran parte de las compañías con mayor capitalización de mercado corresponden a empresas tecnológicas como Apple, Amazon, Microsoft, Alphabet, Facebook o NVIDIA. Estas empresas hacen parte del top 20 de los mayores proveedores de bienes y servicios a nivel mundial[1], por lo cual, sus temáticas de investigación son consideradas como tendencias en tecnología.

La Tabla I presenta un resumen de las áreas de investigación activas de algunas de estas empresas, en las que el procesamiento de señales (imagen, video, voz) y de datos, junto con la inteligencia artificial (aprendizaje de máquina, aprendizaje profundo), la visión por computador, y el procesamiento natural del lenguaje, son temáticas comunes entre ellas.

Tabla 1. Áreas de investigación activas en las mayores empresas tecnológicas actuales

Empresa	Algunas áreas de investigación activas
Apple[2]	Accesibility, Computer vision, Data science and annotation, Health, Privacy, Speech and natural language processing.

[1] https://companiesmarketcap.com/
[2] https://machinelearning.apple.com/research/

Amazon[3]	Cloud and systems, Computer vision, Natural language processing, Economics, Information and knowledge management, Machine learning, Operations research and optimization, Quantum technologies, Robotics.
Microsoft[4]	Artificial intelligence, Computer vision, Human computer interaction, Security, privacy and cryptography, Systems and networking.
Alphabet[5]	Algorithms and theory, Data mining and modeling, Information retrieval and the Web, Machine perception, Speech processing, Human-Computer Interaction and Visualization, Machine Intelligence, Natural Language Processing.
Facebook[6]	AR/VR, Artificial Intelligence, Computer vision, Data science, Databases, Machine learning, Natural Language Processing & speech, Security & Privacy.
NVIDIA[7]	3D, Deep learning, Artificial intelligence and machine learning, Computational photography and imaging, Computer graphics, Computer vision, Real-time rendering, VR, AR and display technology.

En el caso particular de la visión por computador, cabe resaltar que este es un campo interdisciplinario que puede involucrarse con diferentes áreas de la ciencia, ingeniería o tecnología. Por ejemplo, en la física, donde se estudia la óptica y la formación física de las imágenes; o en las ciencias biológicas y psicológicas, para entender cómo se procesa físicamente la información visual. Con las matemáticas e ingeniería de software, para el avance en la implementación de algoritmos de visión por computador.

En la más reciente década los principales avances en visión por computador se han alcanzado gracias a la aplicación de técnicas de inteligencia artificial, inicialmente con esquemas de aprendizaje de máquina y luego, específicamente,

[3] https://www.amazon.science/
[4] https://www.microsoft.com/en-us/research/
[5] https://research.google/research-areas/
[6] https://research.fb.com/research-areas/
[7] https://www.nvidia.com/en-us/research/

con el uso de herramientas de aprendizaje profundo. Es importante señalar que las técnicas de aprendizaje profundo son un subconjunto de técnicas de aprendizaje de máquina, y estas a su vez son un subconjunto de técnicas de inteligencia artificial, como lo muestra la Figura 1. Sin embargo, aunque se han presentado avances importantes en los últimos años, la inteligencia artificial no es un concepto nuevo, y sus principales desarrollos se han venido postulando desde mediados del siglo pasado.

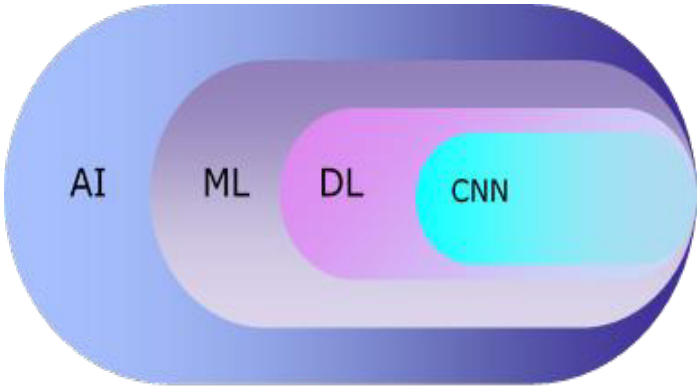


Figura 1. CNN y aprendizaje profundo en el contexto de aprendizaje automático e inteligencia artificial

Como contexto al aporte de la IA, se puede decir que la inteligencia artificial se define como cualquier técnica que permite a los computadores imitar el comportamiento humano. Por su parte, el aprendizaje de máquina permite alcanzar cierto grado de inteligencia artificial en sistemas que pueden aprender de la experiencia para reconocer patrones en un conjunto de datos. Por ejemplo, los sistemas basados en aprendizaje profundo son un caso específico de aprendizaje estadístico para extraer características o atributos de un conjunto de datos en bruto mediante el uso de múltiples capas ocultas, mientras que, los esquemas de aprendizaje de máquina, típicamente requieren que la extracción de características se realiza de forma manual, como etapa previa al modelamiento.

Por otra parte, es posible discriminar diferentes tipos de aprendizaje profundo: aprendizaje supervisado, aprendizaje no supervisado o aprendizaje por refuerzo. El aprendizaje supervisado se caracteriza principalmente por contar con un conjunto de datos que involucra tanto las características de los datos que serán procesados (datos de entrada), así como una etiqueta que los define (lo que tiene que aprender el algoritmo). En este caso, se habla de un conjunto de datos que permitirá tanto entrenar un algoritmo, como evaluarlo con datos correctamente etiquetados (*ground truth*). Como ejemplos de modelos que utilizan este tipo de

aprendizaje se tienen las redes neuronales convolucionales, las redes neuronales recurrentes o las arquitecturas tipo codificador-decodificador (*encoder-decoder*).

Por el contrario, en el caso de aprendizaje no supervisado, no se cuenta con un conjunto de datos etiquetados, por lo cual el algoritmo deberá identificar por su cuenta los patrones que caracterizan a los datos sin una referencia específica (*ground truth*). Un ejemplo de ello, son los algoritmos de clusterización. Por su parte, el aprendizaje por refuerzo involucra modelos que a partir de la representación de la información aprenden a realizar acciones a partir de una especie de realimentación (recompensas o penalizaciones), lo que a su vez puede modificar el estado del entorno8.

En cualquier caso, el aprendizaje deberá realizarse a partir de las características o atributos de los datos de entrada. Un ejemplo sencillo de atributos puede ser un clasificador de imágenes de frutas, en el cual las entradas son el peso y el color. Con un conjunto de datos reducido, esto permitirá distinguir algunas frutas entre sí, pero tal vez no será suficiente para generalizar el problema. En un contexto real, es probable que el conjunto de datos de entrada sea de una dimensionalidad mucho más alta y que a su vez el problema tenga un mayor número de datos. De hecho, algunos conjuntos de datos de reconocimiento de imágenes cuentan con millones de imágenes. Por esta razón, es importante tener en cuenta el tamaño del conjunto de datos, las entradas o características del modelo, la disponibilidad de los datos, al momento de trabajar en un proyecto de ciencia de datos.

Partiendo de lo anterior, se resaltan dos aspectos clave que permitieron la evolución del aprendizaje automático hacia el aprendizaje profundo. El primero, tiene que ver con una mayor cantidad de datos disponibles, propiciado por la masificación de tecnologías como Internet o la telefonía móvil, y sus respectivas aplicaciones. El segundo aspecto tiene que ver con las mejoras en el rendimiento de sistemas computacionales, por ejemplo, a partir de computación paralela o la disponibilidad a mayor escala de unidades de procesamiento gráfico (GPU). Estos dos aspectos permitieron no solo alcanzar un mayor rendimiento, si no también que este rendimiento se alcanzará a través de redes neuronales más profundas, tal como lo ilustra la Figura 2 (Saravia, 2021).

[8] https://deeplearning.mit.edu/

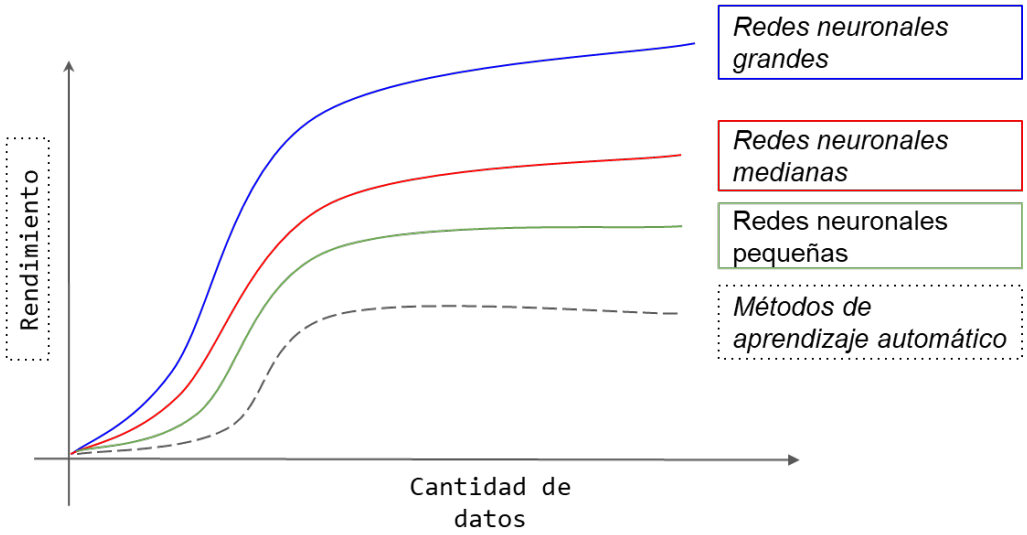


Figura 2. Rendimiento versus tamaño de un conjunto de datos en modelos de aprendizaje automático.

Respecto a los principales tipos de problemas, o las salidas que puede entregar un algoritmo de aprendizaje automático se tienen la regresión, clasificación, *clustering* o predicción de secuencia. La regresión se caracteriza por predecir un valor continuo, por ejemplo, predecir el valor de un bien inmueble a partir de sus características (antigüedad, tamaño, sector, etc.). Los modelos de clasificación se encargan de predecir un valor discreto (o categoría), por ejemplo, en la predicción de la raza de un perro a partir de una imagen. Por su parte, la función de los algoritmos de *clustering* consiste en realizar la segmentación o pertenencia a un grupo de casos similares, por ejemplo, en la segmentación de clientes de un supermercado a partir de su historial de compra. En cuanto a la predicción de secuencias, estas se encargan de estimar los valores siguientes de la misma, por ejemplo, en los sistemas de texto predictivo presentes en los dispositivos móviles.

TRABAJOS REPRESENTATIVOS EN VISIÓN POR COMPUTADOR

Aunque el auge actual de modelos y algoritmos basados en visión por computador ha presentado enormes y valiosos aportes en la última década, su desarrollo se ha venido construyendo décadas atrás. De hecho, las redes neuronales aparecen por primera vez a finales de la década del 50 del siglo pasado, en un trabajo donde Frank Rosenblatt del *Cornell Aeronautical Laboratory* propone el uso del perceptrón de una sola capa (Rosenblatt, 1958). Esta arquitectura fue posteriormente generalizada al conocido perceptrón multicapa (MLP), el cual

sigue siendo inspiración para actuales desarrollos (Liu, 2021), (Tolstikhin, 2021). Sin embargo, un aporte fundamental para el uso de esta arquitectura fue el planteamiento del proceso de aprendizaje por retro propagación (*backpropagation*) para redes neuronales, el cual ajusta de manera iterativa los pesos de las conexiones de la red para minimizar la diferencia entre el valor estimado por la red y su etiqueta o valor real (Rumelhart, 1986). Un trabajo posterior va un paso más allá, aplicando el algoritmo de retro propagación al reconocimiento de dígitos escritos a mano en documentos postales (LeCun Y. B., 1989), misma temática trabajada en la primera red neuronal convolucional propuesta por el mismo autor (LeCun Y. B., 1998).

Como un primer acercamiento para la comprensión de la visión a nivel biológico, el trabajo presentado por (Hubel, 1962) y realizado sobre gatos aplicó electrofisiología para estimular la corteza visual primaria mediante filtros orientados y describir así su respuesta. Dichos autores encontraron que las células simples responden a la orientación de la luz, las células complejas responden tanto a la orientación de luz como al movimiento y que células complejas responden a movimientos de luz en puntos específicos (Hubel, 1962).

Para la siguiente década, se presentaron dos iniciativas en el Instituto de Tecnología de Massachusetts (MIT). Una de ellas se considera la primera tesis de doctorado en visión por computador, y estuvo orientada hacia el modelado de sólidos en 3D para el reconocimiento y reconstrucción de formas geométricas simples del mundo real (Roberts, 1963). La segunda fue un proyecto de verano orientado a la solución de tareas concretas en visión por computador para el reconocimiento de patrones (Papert, 1966).

En un trabajo posterior desarrollado en este mismo instituto en la década de los 70, y publicado de manera póstuma (Marr D. , 2010), el autor describe un marco general para entender la percepción visual, abordando cuestiones sobre el estudio y comprensión del cerebro y sus funciones para la construcción de representaciones a partir de la descripción de una imagen y la descripción de objetos tridimensionales del entorno. Un aspecto importante es la introducción por parte de dicho autor sobre nociones para diferentes niveles de análisis: nivel computacional, nivel algorítmico y nivel de implementación de hardware.

De forma paralela, se abordó un problema fundamental en la representación de objetos: reducir la estructura compleja de un objeto a una colección de formas más simples y su configuración geométrica. En este sentido, se desarrollaron trabajos para la representación del cuerpo humano, obteniendo representaciones

mediante cilindros (Brooks, 1979) y mediante una estructura pictórica (Fischler, 1973). Estos trabajos constituyen la base de los modelos utilizados hoy en día.

Posteriormente se presentan trabajos enfocados en el reconocimiento de bordes y líneas, así como la segmentación de áreas de la imagen. Aquí se tienen trabajos pioneros en el área, como el reconocimiento de objetos 3D a partir de imágenes en 2D (Lowe D. G., 1987), la segmentación binaria de imágenes para resolver el problema de agrupación perceptiva en la visión (Shi, 2000), el reconocimiento de objetos a partir de detección de puntos clave mediante características SIFT (*Scale-Invariant Feature Transform*) (Lowe D. , 1999). También se tienen trabajos relacionados con el reconocimiento de categorías de escenas mediante la división de la imagen en subregiones y el correspondiente cálculo de histogramas de las características locales de cada subregión obteniendo una representación llamada pirámide espacial (Lazebnik, 2006). Otros importantes trabajos en esta temática involucran fundamentos de detección de bordes (Marr D. &., 1980), o algoritmos fundamentales y ampliamente conocidos para detección de bordes (Canny, 1986), segmentación de imágenes basada en regiones (Chan, 2001) y detección de ángulos (Harris, 1988).

También se tienen trabajos para la detección de personas o partes de estas (como el reconocimiento de rostros). Por ejemplo, los autores en (Dalal, 2005), proponen el uso de descriptores de gradientes orientados (HOG) para el reconocimiento robusto de objetos visuales (particularmente detección humana basada en SVM lineal). Por su parte, los autores en (Felzenszwalb, 2008), proponen la detección de personas mediante un modelo de detección de objetos entrenado para reconocer diferentes formas a diferentes escalas. Respecto a la detección de rostros, uno de los trabajos más relevantes está basado en la transformada wavelet con filtro Haar y un clasificador Adaboost (Viola, 2004).

Más allá de estos importantes aportes, uno de los contextos que impulsó en gran medida el desarrollo de nuevas soluciones en el área de visión por computador fue el establecimiento de desafíos abiertos a la comunidad científica. De forma particular, el desafío PASCAL para reconocimiento de objetos visuales, establecido con veinte categorías de objetos y alrededor de diez mil imágenes por clase, y el desafío de clasificación de imágenes, IMAGENET, establecido inicialmente con mil clases y mil imágenes por clase (1 millón de imágenes), se convirtieron en referentes para el desarrollo de nuevos modelos para reconocimiento de objetos, como por ejemplo nuevas arquitecturas de aprendizaje profundo.

Uno de los aspectos a resaltar aquí, radica en que el desafío IMAGENET permitió el surgimiento de las redes neuronales convolucionales, las cuales alcanzaron un rendimiento superior sobre las técnicas de visión por computador tradicionales, lo que llevaría al auge del aprendizaje profundo. Este punto de quiebre se presentó en la edición 2012 de este desafío, con una arquitectura de 8 capas de profundidad conocida como AlexNet (Krizhevsky, 2012). Para el año siguiente el primer puesto del desafío lo obtiene igualmente una arquitectura de 8 capas de profundidad, para la cual los autores exploraron técnicas de visualización que permitieron conocer la función de las capas de características intermedias y el funcionamiento del clasificador, superando el rendimiento de la arquitectura AlexNet (Zeiler, 2014).

Para el año 2014, surgen dos arquitecturas fundamentales en el desarrollo del aprendizaje profundo: VGG (Simonyan, 2014), una arquitectura de 19 capas para el reconocimiento de imágenes a gran escala y GoogLeNet (Szegedy, 2015), una arquitectura de 22 capas cuyo rendimiento fue superior al alcanzado por arquitecturas previas. Otra arquitectura fundamental en el estado del arte es ResNet, caracterizada por su gran profundidad (152 capas) y que se presentó en 2015, siendo premiada como la mejor ponencia en la conferencia sobre visión por computador y reconocimiento de patrones (CVPR) (He, Deep residual learning for image recognition, 2016).

La investigación y aportes relacionados hasta aquí, sumados a una gran cantidad de desarrollo y propuestas adicionales realizados por diferentes empresas, universidades e investigadores han permitido el avance de esta área, mostrando un avance significativo en la última década. Esto ha llevado a que el sector gubernamental, las universidades y los departamentos de investigación y desarrollo de las grandes empresas de tecnología hayan llevado su atención hacia temáticas relacionadas con este tema. Dada su importancia, esta es precisamente el área que se tratará en el presente libro.

2. FUNDAMENTOS DE APRENDIZAJE AUTOMÁTICO

Como contexto del concepto de redes neuronales, se partirá de la noción de regresión lineal. Una regresión implica modelar la relación entre variables independientes y una variable dependiente con el fin de predecir un valor numérico. Cuando se habla de regresión lineal, esta relación entre variables independientes y la variable dependiente es lineal, es decir se puede representar como una suma ponderada de las variables independientes hacia la variable dependiente.

Consideremos un conjunto de datos de entrenamiento, que incluye *n* ejemplos, donde el i-ésimo ejemplo se representará por el superíndice *i*. Las variables independientes o atributos de los datos se representarán por **x**, donde x_1, x_2, … corresponden al primer y segundo atributo, respectivamente. De esta forma, el i-ésimo ejemplo se representa de la forma:

$$x^i = \begin{bmatrix} x_1^i \\ x_2^i \end{bmatrix} \tag{1}$$

Y la variable independiente, o etiqueta del i-ésimo ejemplo se representa de la forma:

$$y^i \tag{2}$$

Utilizando esta representación, un conjunto de datos queda estructurado de la siguiente forma:

	Atributo 1	**Atributo 2**	⋯	**Atributo n**	**Etiqueta Salida (y)**
x^1	x_1^1	x_2^1	⋯	x_n^1	y^1
x^2	x_1^2	x_2^2	⋯	x_3^2	y^2
x^3	x_1^3	x_2^3	⋯	x_n^3	y^3
x^4	x_1^4	x_2^4	⋯	x_n^4	y^4

x^5	x_1^5	x_2^5	$\cdots$	x_n^5	y^5
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
x^m	x_1^m	x_2^m	$\cdots$	x_n^m	y^m

Siendo m el número de ejemplos o muestras del dataset, y n el número de atributos o variables independientes.

Para contextualizar lo anterior, tomemos un ejemplo relacionado con un modelo de regresión para estimar el valor de una vivienda en una ciudad determinada. Consideremos cuatro atributos de los datos entrada: el área, la antigüedad, el número de habitaciones y la zona donde está ubicada. Bajo este contexto, un modelo de predicción lineal del precio de la vivienda quedaría expresado bajo una suma ponderada, así:

$$precio = w_a \times área + w_e \cdot edad + w_h \cdot hab + w_z zona + b \quad (3)$$

En esta suma ponderada, los términos w_x corresponden al peso que tiene un determinado atributo en la suma, es decir qué tanto aporta en el cálculo de la variable independiente, que en este caso es el precio del bien inmueble. Por ejemplo, w_a representa el aporte o peso que tiene la superficie de la vivienda en el cálculo del precio.

Además, esta suma ponderada incluye un valor de sesgo o *bias* que permite ajustar el modelo. Para entender este concepto, podemos relacionarlo con el nivel DC que puede tener la respuesta de un circuito eléctrico, o de forma general podemos imaginar un modelo de regresión lineal con una sola variable independiente representado a través de la ecuación de una recta, expresada en su forma pendiente-intercepto:

$$Y = mx + b \quad (4)$$

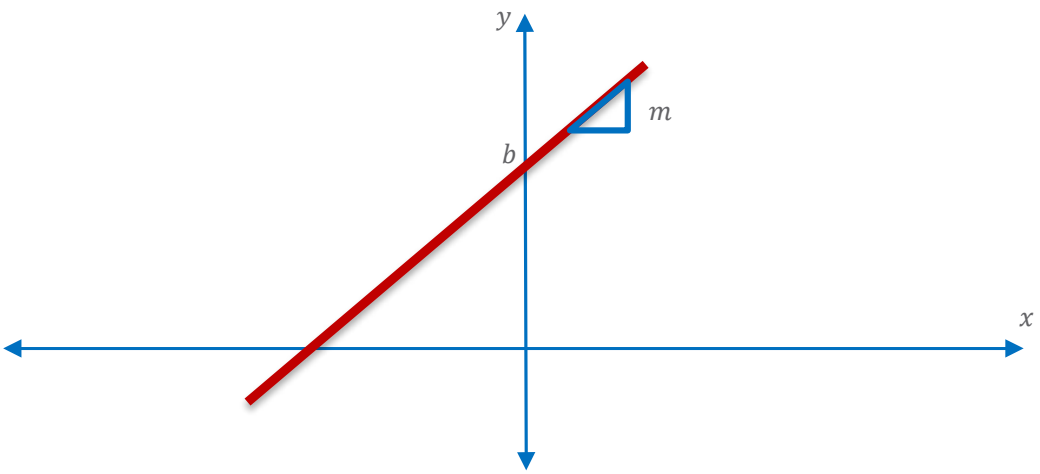


Figura 3. Función lineal representada en forma pendiente-intercepto

Si relacionamos la ecuación de la recta con el modelo de regresión lineal, la pendiente (m) corresponde al peso de la variable independiente (x) y el intercepto corresponde al sesgo o *bias* necesario para ajustar la recta en el eje vertical (y). De esta forma podríamos reescribir un modelo de regresión lineal de una sola variable independiente como:

$$y = wx + b \tag{5}$$

Donde w lo asociamos a la pendiente y b al intercepto de la recta.

Al expresar el modelo de regresión lineal como una suma ponderada, el objetivo será encontrar $\boldsymbol{w}$ y b a partir de los ejemplos del conjunto de datos, de tal forma que dichos valores se ajusten lo mejor posible a la etiqueta o valor de salida de los datos.

GENERALIZACIÓN

Considerando un caso que involucre n atributos (x_1, x_2, … x_n) en los datos de entrada, el modelo de regresión lineal para una muestra del conjunto de datos se puede expresar como:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b \tag{6}$$

Donde $\hat{y}$ es el valor de salida estimado por el modelo.

De esta forma, la dimensión del vector de atributos ***x*** de un ejemplo es *n*-dimensional al igual que la dimensión del vector de pesos, ***w***, es decir:

$$\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{w} \in \mathbb{R}^n \tag{7}$$

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + b \tag{8}$$

Al considerar el grupo de atributos de entrada como un vector ***x*** (x_1, x_2, … x_n) y el grupo de pesos como un vector **w** (w_1, w_2, … w_n), el modelo de regresión se puede expresar como:

Ahora, considerando que los datos de entrada están consolidados en un dataset que incluye *m* ejemplos, estructurados en una fila por cada ejemplo y una columna por cada atributo, es decir el conjunto de *n* atributos para *m* ejemplos será una matriz **X** de dimensiones:

$$\boldsymbol{X} \in \mathbb{R}^{m \times n} \tag{9}$$

Dado que la salida normalmente es un valor único, esta será un vector que involucre los *m* valores de salida (uno para cada ejemplo), es decir:

$$\hat{y} \in \mathbb{R}^m \tag{10}$$

Por su parte, las dimensiones del vector de pesos se mantienen con respecto al vector con un solo ejemplo, ya que el proceso de regresión lineal obtendrá un solo modelo o comportamiento para todos los ejemplos, es decir:

$$\boldsymbol{w} \in \mathbb{R}^n \tag{11}$$

De esta forma, el modelo de regresión lineal se puede generalizar para *m* ejemplos con *n* atributos así:

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w} + b \tag{12}$$

Donde $\hat{y}$ corresponde al valor estimado por el modelo (salida), **X** corresponde a una matriz que contiene los *n* atributos de entrada para *m* ejemplos y *b* corresponde al sesgo o ajuste del modelo.

El objetivo del entrenamiento del modelo generalizado sigue siendo el mismo: encontrar valores para ***w*** y b, que permitan realizar predicciones sobre nuevos datos (muestras a partir de la misma distribución de **X**) con el menor error posible.

FUNCIÓN DE PÉRDIDA

Dado que el objetivo del entrenamiento del modelo a partir de los datos involucra obtener los parámetros (***w*** y b) que presenten el menor error en la predicción, es necesario utilizar una función que permita medir dicho error. Esta función se conoce como función de pérdida, y se encarga de cuantificar la distancia entre el valor real (y) y el estimado por la función objetivo ($\hat{y}$).

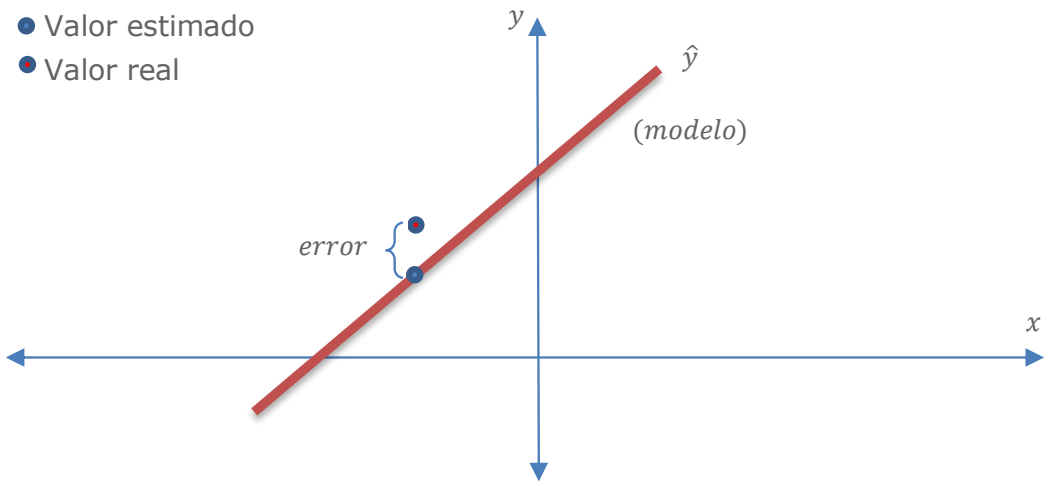


Figura 4. Error entre valor estimado y valor real

Típicamente, para calcular una función de error en aprendizaje automático es posible utilizar una métrica de distancia entre dos vectores, o alguna alternativa para calcular el tamaño de un vector. Por ejemplo, la función de error cuadrático o norma L_2:

$$l^i(\boldsymbol{w}, b) = \frac{1}{2}\left(\hat{y}^i - y^i\right)^2 \tag{13}$$

De esta forma, para cada uno de los ejemplos (i), la función calcula el cuadrado de la diferencia entre el valor estimado respecto al valor real. Dado el exponente de la expresión, esta función penaliza en mayor medida valores altos en la

diferencia, dada su naturaleza cuadrática. La expresión anterior también utiliza la constante ½, que permite alcanzar un valor unitario en el coeficiente cuando se derive la función al calcular el gradiente (lo que se verá reflejado con mayor claridad en las siguientes ecuaciones).

Dado que la ecuación anterior determina el valor de la función de pérdida para un ejemplo dado, y en aras de generalizar la función de pérdida para todos los ejemplos del conjunto de datos, se realiza un promedio sobre el valor de la pérdida para todos y cada uno de los ejemplos, así:

$$L(\boldsymbol{w}, b) = \frac{1}{m}\sum_{i=1}^{m} l^i(\boldsymbol{w}, b) = \frac{1}{m}\sum_{1}^{m}\left(\boldsymbol{w}^T\boldsymbol{x}^i + b - y^i\right)^2 \quad (14)$$

De esta forma, el objetivo del entrenamiento del modelo es minimizar la función de pérdida respecto a **w** y b. Es decir, encontrar los parámetros $\boldsymbol{w}$ y b en el dominio de la función L en los que el valor de esta se minimiza.

$$w^*, b^* = \underset{w,b}{argmin} L(\boldsymbol{w}, b) \quad (15)$$

REDUCCIÓN DEL ERROR

La reducción de error en la función de pérdida con respecto a los pesos ($\boldsymbol{w}$) y el sesgo o bias (b), involucra la actualización de los valores de w y b hacia la dirección del mínimo de la función.

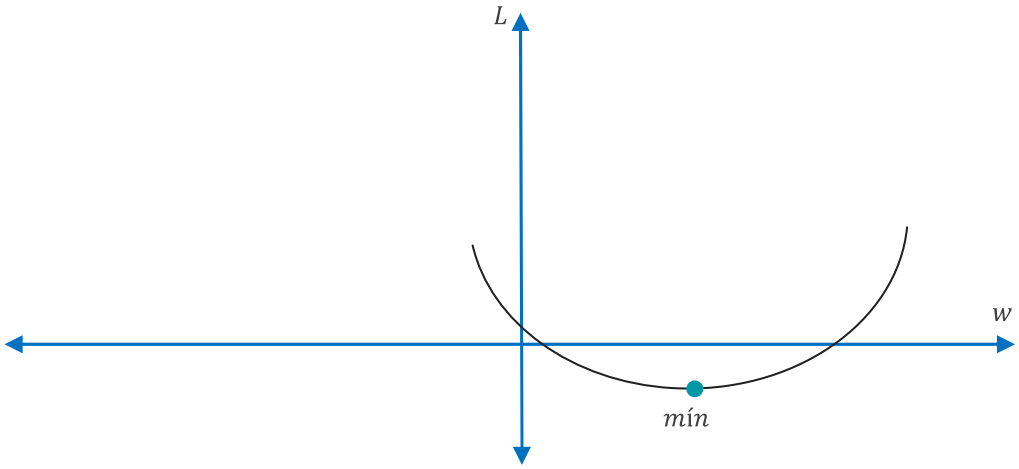


Figura 5. Ilustración del mínimo de una función

Este proceso se alcanza mediante un algoritmo de optimización como el algoritmo de gradiente descendente, el cual reduce iterativamente el error mediante la actualización de los parámetros en la dirección que disminuye progresivamente la función de pérdida. Con el objetivo de minimizar la función de pérdida, la actualización de los parámetros requiere el cálculo del gradiente de la función con respecto a las variables de pesos y *bias*, así:

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{m}\sum_{1}^{m}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial \boldsymbol{w}} \tag{16}$$

$$b = b - \frac{\alpha}{m}\sum_{1}^{m}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial b} \tag{17}$$

Donde, α representa un valor numérico conocido como tasa de aprendizaje. En estas dos últimas ecuaciones, el algoritmo de gradiente descendente opera sobre todo el conjunto de datos promediando su valor entre el número ejemplos (m).

Otro enfoque del algoritmo implica la conformación de un subconjunto de datos (conocido como minilote o minibatch), cada vez que se requiere actualizar los valores de los pesos y *bias*. Este enfoque se conoce como *Minibatch stochastic gradient descent.* En este caso, la actualización de los parámetros se realiza de la siguiente forma:

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial w} \tag{18}$$

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial b} \tag{19}$$

Donde *BS* representa el número de ejemplos de cada minilote (tamaño del lote o *batch size*) y α denota la tasa de aprendizaje.

Como paso previo a la aplicación del algoritmo por *Minibatch stochastic gradient descent,* es necesario inicializar los parámetros del modelo (w y b), lo cual por lo general se realiza de manera aleatoria. Posterior a esto, el algoritmo de forma iterativa toma minilotes aleatorios de los datos, y actualiza los parámetros en la

dirección del gradiente negativo. Tomando el gradiente de la función de pérdida con respecto a **w**, la actualización de los pesos se realiza de la siguiente forma:

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\hat{y}^i - y^i\right)^2\right)}{\partial w} \tag{20}$$

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\boldsymbol{w^T x^i} + b - y^i\right)^2\right)}{\partial w} \tag{21}$$

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\boldsymbol{x^i}\left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{22}$$

Por su parte, el valor de *bias* se actualiza restando al valor actual el término que incluye el gradiente de la función de pérdida respecto a b:

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\hat{y}^i - y^i\right)^2\right)}{\partial b} \tag{23}$$

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\boldsymbol{w^T x^i} + b - y^i\right)^2\right)}{\partial b} \tag{24}$$

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{25}$$

Como conclusión, el entrenamiento de los modelos consiste en realizar múltiples iteraciones sobre el conjunto de datos, tomando un minilote de ejemplos cada vez y actualizando los parámetros del modelo (*w, b*) en cada iteración. La iteración de todo el conjunto de datos se conoce como época, e involucra utilizar una vez cada lote del conjunto de datos de entrenamiento (siempre y cuando el número de ejemplos sea divisible por el tamaño del lote).

PARÁMETROS E HIPERPARÁMETROS

Durante el proceso de entrenamiento, el algoritmo de optimización permite actualizar de forma iterativa los parámetros del modelo, que para el modelo de regresión estudiado hasta aquí hemos denominado **w** y b. Sin embargo, el entrenamiento del modelo estipula valores adicionales como la tasa de aprendizaje, el tamaño del lote o el número de épocas, valores que típicamente no aprende el modelo durante el entrenamiento.

Estos valores que pueden ser ajustables pero que no se actualizan durante el entrenamiento se conocen como hiperparámetros. La selección de estos hiperparámetros se puede realizar con base en los resultados del entrenamiento, al evaluar el modelo sobre un conjunto de datos diferente al de entrenamiento que se conoce como conjunto de validación.

EJEMPLO DE IMPLEMENTACIÓN DE UN MODELO DE REGRESIÓN LINEAL

Para contextualizar un modelo de regresión lineal completo y aplicar los conceptos de este capítulo, será necesario considerar por los menos las siguientes fases:

- Datos de entrenamiento y lectura por lotes
- Creación del modelo
- Inicialización de parámetros
- Selección de función de pérdida
- Selección de algoritmo de optimización
- Entrenamiento del modelo

A continuación, se muestra el desarrollo de cada una de estas fases en un modelo de regresión lineal.

- Datos de entrenamiento y lectura por lotes

En este ejemplo se generarán 10000 datos de prueba que corresponden a las calificaciones obtenidas por 10000 estudiantes en una asignatura. Estas tres notas estarán entre 0.0 y 5.0, y por medio de ellas se generará la calificación

definitiva (etiqueta de salida) que se calcula considerando un 30% de la primera nota, 30% para la segunda nota y 40% para la tercera nota.

Lo anterior significa que los datos de entrenamiento se generarán de manera sintética, considerando tres atributos (x_1, x_2, x_3) con calificaciones entre 0.0 y 5.0. La nota definitiva será el valor a predecir, y para estos datos de entrenamiento se generan conociendo el peso de cada atributo, mediante la siguiente ecuación y código en Python:

$$y = 0.3x_1 + 0.3x_2 + 0.4x_3 \tag{26}$$

```
porcentajes = tf.constant([0.3, 0.3, 0.4])
b_ini = 0.0

atributos = tf.zeros((10000, 3))
atributos += tf.random.uniform(shape=atributos.sha
pe, minval=0.0, maxval=5.0)
etiqueta = tf.matmul(atributos, tf.reshape(porcent
ajes, (-1, 1))) + b_ini
etiqueta += tf.random.normal(shape=etiqueta.shape,
 stddev=0.2)
etiqueta = tf.reshape(etiqueta, (-1, 1))
```

En el código anterior, los pesos están definidos en la variable `porcentajes`, y no se incluye *bias* (b=0.0). La etiqueta de salida (`etiqueta`) se calculó utilizando la ecuación anterior, incluyendo cierto nivel de ruido en los datos mediante la adición de valores aleatorios obtenidos de una distribución normal con desviación estándar de 0.2.

Para el ejemplo, la dimensión de los atributos de entrada (en la variable `atributos`) tiene una dimensión de (10000, 3), mientras que la dimensión de la salida o etiqueta (variable `etiqueta`) es (10000, 1).

Para la lectura por minilotes, es preciso utilizar una función que permita crear un subconjunto de datos cuyos elementos sean extraídos de los tensores de entrada, conservando la estructura de estos. Esta operación se puede realizar por medio de la función `Dataset.from_tensor_slices` de tensorflow, junto con el método `batch` que permite combinar elementos consecutivos de un conjunto de

datos en lotes, de acuerdo con el tamaño del lote que se define al utilizar el método:

```
tam_lote=8

# Iteraciones para lectura del dataset en formato TensorFlow
dataset = tf.data.Dataset.from_tensor_slices((atributos, etiqueta))
dataset = dataset.shuffle(buffer_size=10000)
data_iter = dataset.batch(tam_lote)
```

Como buena práctica, también se han aleatorizado los datos de entrada, previo a la lectura por lotes (`dataset.shuffle`). A manera de prueba, se puede visualizar el contenido de una iteración, cuyo resultado serán 8 ejemplos del conjunto de datos, tanto para los atributos, como para la etiqueta:

```
next(iter(data_iter))
```

```
(<tf.Tensor: shape=(8, 3), dtype=float32, numpy=
 array([[2.2236247 , 1.8797708 , 3.2748823 ],
        [4.3598833 , 2.7310383 , 3.6872144 ],
        [1.310643  , 4.992275  , 4.745705  ],
        [4.1002517 , 3.413506  , 0.38547993],
        [4.4065337 , 1.3784158 , 1.4818233 ],
        [3.968305  , 0.14479339, 0.5596578 ],
        [4.7774534 , 2.0571613 , 2.067144  ],
        [3.0095048 , 2.2771108 , 3.656056  ]], dtype=float32)>,
 <tf.Tensor: shape=(8, 1), dtype=float32, numpy=
 array([[0.6800413],
        [2.6938043],
        [3.449986 ],
        [3.6250317],
        [1.6673617],
        [1.3380989],
        [1.7778645],
        [1.4929203]], dtype=float32)>)
```

En este caso, la respuesta muestra un tensor de dimensiones (8,3), lo que equivale a 8 ejemplos con 3 atributos cada uno. El segundo dato corresponde a un tensor de dimensiones (1,8), equivalente a la etiqueta de salida de cada uno de los 8 ejemplos.

- Creación del modelo

Como se ha comentado hasta aquí, el modelo de regresión lineal estudiado hasta aquí, que involucra la suma ponderada de los atributos de entrada (**X**) más el offset (b) para obtener el valor de salida estimado ($\hat{y}$) y se puede expresar mediante la Ecuación:

$$\hat{y} = \boldsymbol{X}\boldsymbol{w} + b \tag{27}$$

La implementación de este modelo en Python, se puede realizar mediante un producto punto entre los atributos de entrada (**X**) y los pesos del modelo (**w**) más el offset (b). Utilizando tensorflow, esta operación se puede realizar mediante la función `matmul` que permite multiplicar dos matrices. De esta forma, incluyendo el offset, el modelo puede definirse así:

```
def reg_lineal(X, w, b):
    return tf.matmul(X, w) + b
```

Sin embargo, este modelo también se puede formar mediante una relación como la mostrada en la siguiente gráfica (Saravia, 2021):

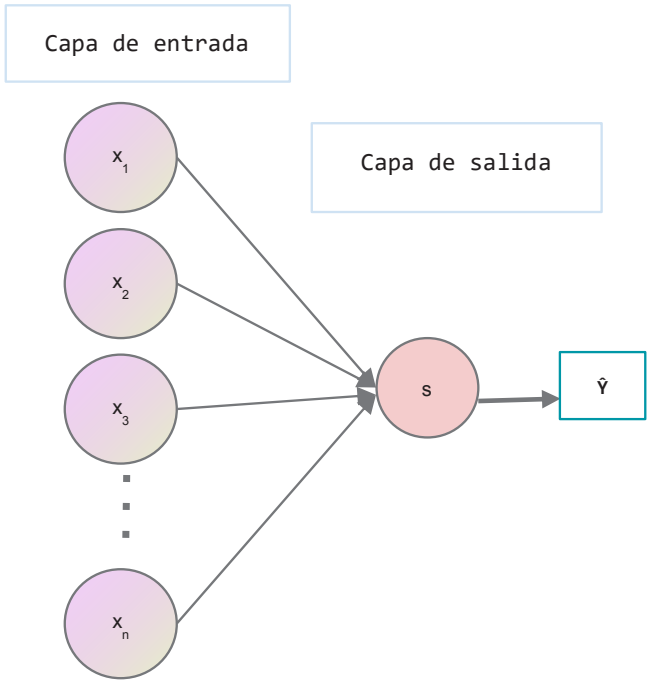


Figura 6. Ejemplo de regresión lineal mediante una red neuronal.

En la gráfica mostrada, los atributos de la entrada (x_1, x_2, …, x_n) están representados como una capa de entrada, cuyos valores aportan al cálculo del valor de salida con un peso determinado. Es decir, las líneas que unen cada atributo de entrada x con la salida, corresponden a los pesos de la regresión lineal. Al igual que en el modelo de regresión lineal, también se precisa de un valor de ajuste o sesgo (*offset* o *bias*).

El modelo mostrado corresponde a una red neuronal de una sola capa (la capa de salida, dado que la capa de entrada por lo general no se cuenta). Este tipo de capa en una red neuronal se conoce como densa o totalmente interconectada (*fully-connected (FC)*), dado que modela la relación entre las entradas y la salida como una suma ponderada y se implementa como una operación de multiplicación matriz/vector.

Para la implementación del modelo, se utilizará una API de alto nivel para `tensorflow`, conocida como `keras`. Las capas del modelo se enlazarán de manera secuencial, para lo cual `keras` dispone de la clase `Sequential` en la cual los datos de entrada se procesan en la primera capa, y su resultado será la entrada para la segunda capa y así sucesivamente.

Las capas tipo FC en keras se definen mediante la clase `Dense`, en la cual el argumento principal corresponde al número de neuronas o unidades de salida. Por su parte, el número de entradas de la capa `Dense` es calculado automáticamente por `keras` al ejecutar el modelo. De esta forma, al utilizar `tensorflow` para crear el modelo de regresión lineal quedaría así:

```
modelo = tf.keras.Sequential()
modelo.add(tf.keras.layers.Dense(1))
```

- Inicialización de parámetros

Dado que el algoritmo de optimización utilizado durante el aprendizaje involucra la actualización de los parámetros (**w**, b) a medida que avance el entrenamiento del algoritmo, es necesario definir un valor inicial previo a la primera iteración. Por lo general los pesos se inicializan con valores aleatorios. Este proceso en Python se puede realizar creando un tensor variable cuyos valores aleatorios se obtengan de una distribución normal. Asimismo, por lo general el *bias* se inicializa con cero, mediante un tensor variable:


```
w = tf.Variable(tf.random.normal(shape=(3, 1), mea
n=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)
```

Sin embargo, en la práctica, los valores iniciales se definen cuando el modelo se utiliza por primera vez (es decir que no es posible manipularlos desde el inicio). Para el caso específico del modelo con capa `Dense`, el método de inicialización se define como argumento de la capa. Para este ejemplo se utilizará el inicializador `RandomNormal`, que genera tensores con una distribución normal a partir de un valor de desviación estándar:

```
inicializador = tf.initializers.RandomNormal(stdde
v=0.01)
modelo_ini = tf.keras.Sequential()
modelo_ini.add(tf.keras.layers.Dense(1,kernel_init
ializer=inicializador))
```

Cabe hacer notar que tensorflow dispone de diversos algoritmos que han sido propuestos en la literatura para la inicialización de parámetros, cuyo diseño ha tenido en cuenta los requerimientos de los modelos actuales y que muchas veces ayudan a la convergencia del algoritmo, a la vez que evitan que parámetros diferentes converjan hacia un mismo valor, dado su carácter aleatorio. El listado de algoritmos de inicialización disponible en `tensorflow/keras` puede consultarse en la URL https://www.tensorflow.org/api_docs/python/tf/keras/initializers.

- Selección de función de pérdida

Tal como se comentó anteriormente, la función para calcular el error de un algoritmo en la predicción de los resultados puede programarse a partir de una métrica de distancia, como por ejemplo el error cuadrático:

```
Def perd_cuad(y_hat, y):
    return (y_hat - tf.reshape(y, y_hat.shape)) **
 2 / 2
```

Sin embargo, `keras/tensorflow` incluye el módulo `losses`, que incluye diversas clases y funciones para calcular el promedio de los errores. Por ejemplo,

la función `MeanSquaredError` permite calcular la media de los cuadrados de los errores entre las etiquetas y las predicciones. Su definición en tensorflow se puede realizar con una sola línea, así:

```
f_perdida = tf.keras.losses.MeanSquaredError()
```

El listado de funciones de pérdida disponible en `tensorflow/keras` puede consultarse en la URL https://www.tensorflow.org/api_docs/python/tf/keras/losses.

- Selección de algoritmo de optimización

La selección de un algoritmo de optimización como el *Minibatch stochastic gradient descent* comentado anteriormente, permite actualizar los valores de los parámetros con el fin de minimizar la función de pérdida. En este sentido, cuando se utilice el algoritmo de optimización durante el entrenamiento, los argumentos de entrada del algoritmo serán el conjunto de parámetros actual, la tasa de aprendizaje, y el tamaño del lote.

En este sentido, `Keras/tensorflow` dispone del módulo `optimizers`, que incluye diversas clases para realizar este proceso. Por ejemplo, la clase `SGD` permite disponer del optimizador de gradiente descendente (con lectura por minilotes, más un parámetro adicional conocido como impulso). Su selección en tensorflow se puede realizar con una sola línea, así:

```
optimizador = tf.keras.optimizers.SGD(learning_rat
e=0.002)
```

- Entrenamiento del modelo

Durante el entrenamiento del modelo, se realizarán iteraciones sobre el conjunto de datos (o subgrupos de este (minilotes)), donde un conjunto de iteraciones que involucre todo el conjunto de datos se conoce como época. Durante el entrenamiento, en cada una de estas iteraciones se realiza el siguiente procedimiento:

- Generar predicciones (mediante el modelo) y calcular la pérdida (proceso conocido como *forward propagation*).
- Calcular el valor de la pérdida y calcular los gradientes mediante el algoritmo (proceso conocido como *backpropagation*)
- Actualizar los parámetros del modelo mediante el optimizador

- Opcional: calcular y mostrar alguna métrica de rendimiento (para monitorear el progreso)

Como una aproximación de este proceso, se utilizará el siguiente bloque de código (Chollet, 2021; Zhang, 2021):

```
num_epochs = 4
for epoch in range(num_epochs):
    for X, y in data_iter:
        with tf.GradientTape() as tape:
            l = f_perdida(modelo_ini(X, training=T
rue), y)
        grads = tape.gradient(l, modelo_ini.traina
ble_variables)
        optimizador.apply_gradients(zip(grads, mod
elo_ini.trainable_variables))
    l = f_perdida(modelo_ini(atributos), etiqueta)
    print(f'época {epoch + 1}, pérdida {l:f}')
```

En este ejemplo se realizan cuatro lecturas completas del conjunto de datos (épocas), donde en cada época se realizan iteraciones del conjunto de datos con un tamaño de lote dado. Por ejemplo, si el conjunto de datos tiene 10000 datos, y se define 8 como el tamaño de lote, se tendrán 10000/8 = 1250 iteraciones en cada época. El método `GradientTape` utilizado en el código anterior facilita llevar un registro de los resultados relacionados con la diferenciación automática, cuyos gradientes serán utilizados por el optimizador para la actualización de los parámetros en cada iteración.

El resultado del entrenamiento, en este caso mostrará los valores de pérdida obtenidos para cada época:

```
época 1, pérdida 0.041337
época 2, pérdida 0.040648
época 3, pérdida 0.040433
época 4, pérdida 0.040402
```

Aunque para este ejemplo, se ha utilizado un bloque de código para el entrenamiento, el cual utiliza aspectos como la lectura por lotes, optimizador o la función de pérdida definidos previamente, *frameworks* como `tensforflow`

incluyen también soluciones para realizar el entrenamiento de los modelos. Esto se abarcará en más detalle en capítulos siguientes.

ALGORITMOS DE OPTIMIZACIÓN

Tal como se ha venido explicando, un algoritmo de optimización es una herramienta que permite la actualización recurrente de los parámetros de un modelo en aras de minimizar el valor de la función de pérdida (o función objetivo), cuando se evalúa en el conjunto de datos de entrenamiento. De aquí que, el rendimiento del algoritmo de optimización tiene una incidencia directa sobre la eficiencia del algoritmo de entrenamiento del modelo. Debido a esto, el estudio de este tipo de algoritmos y sus dependencias facilita un mejor ajuste de algunos hiperparámetros de un modelo.

En este sentido, hiperparámetros como la tasa de aprendizaje, la función de pérdida, el tamaño del lote o el número de épocas están directamente relacionados con el algoritmo de optimización, y cobran sentido una vez se comprende el principio de funcionamiento de dicho algoritmo.

Es importante tener en cuenta aquí que la función objetivo de un algoritmo de optimización corresponde a la función de pérdida. El diseño o selección de esta función debe estar orientada a medir el error de entrenamiento, entendido como la distancia entre el valor que predice un modelo respecto al valor real (obtenido a partir de las etiquetas de los datos de entrenamiento). Asimismo, el proceso de optimización debe propender por reducir el error de generalización del modelo, obtenido al evaluar un modelo sobre datos nuevos (datos que no fueron utilizados ni en entrenamiento ni en validación, pero que proceden de la misma distribución de datos).

Dado que el objetivo de algoritmo de optimización está orientado a minimizar la función de pérdida, este tipo de algoritmos presenta ciertos desafíos que dependen en gran medida del tipo de función objetivo. Particularmente, la búsqueda del mínimo de una función puede verse envuelta en mínimos locales, zonas de la señal con efecto silla o gradientes con tendencia a cero.

Los mínimos locales de una función se presentan cuando el valor de esta función objetivo en un punto dado es menor que los valores de dicha función en cualquier otro punto cercano (es decir, en una vecindad de dicho punto). En un determinado algoritmo de optimización, los mínimos locales podrían confundir a

un algoritmo de optimización, respecto al valor mínimo global que corresponde al mínimo de la función objetivo en todo el dominio.

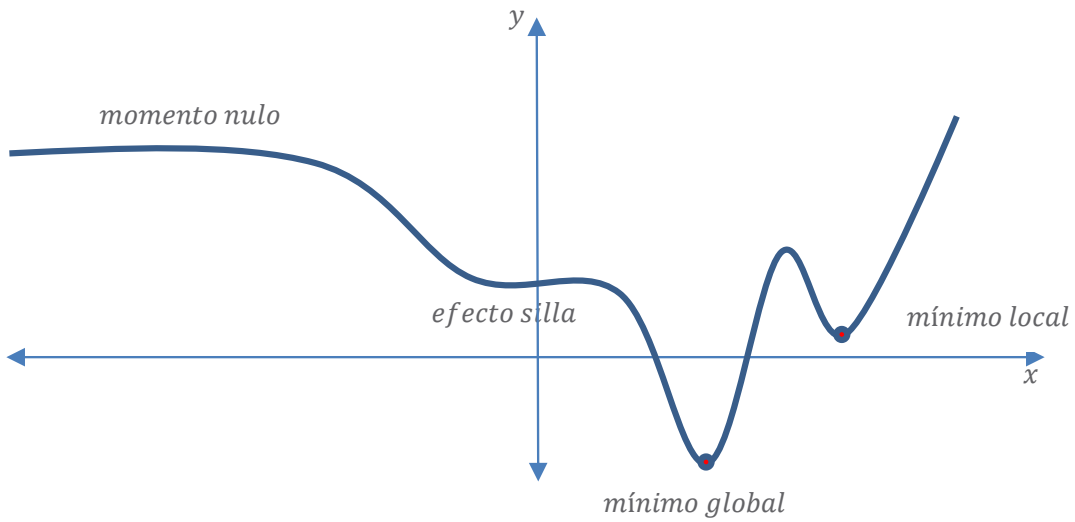


Figura 7. Mínimos de una función

La variación natural de los gradientes que se obtiene al utilizar pequeños subconjuntos del conjunto de datos de entrenamiento, como es el caso del algoritmo de gradiente descendente estocástico por minilotes (MSGD) facilita que el algoritmo sea capaz de sacar los parámetros de los mínimos locales.

En cuanto al efecto silla y gradientes nulos que pueden presentar algunas funciones, estos corresponden a segmentos de esta en los que el gradiente de la función parece atenuarse pero que no corresponden ni a un mínimo global ni a un mínimo local. De hecho, pueden existir zonas de una función (como por ejemplo aproximaciones asintóticas) en las cuales el algoritmo de optimización podría estancarse durante mucho tiempo.

Algoritmo de gradiente descendente

El gradiente descendente es un algoritmo de optimización que permite actualizar los parámetros de un modelo en la dirección que minimiza la función de pérdida, en la cual se involucra el cálculo del gradiente de la función ponderado por una constante conocida como tasa de aprendizaje.

Para entender por qué el algoritmo minimiza la función de pérdida, consideremos una función de costo unidimensional, *f(x)*, cuyo dominio y rango está definido por el conjunto de números reales. Dado que la función del algoritmo de optimización consiste en actualizar el parámetro del modelo (que para este ejemplo es *x*),

partamos de la expansión en serie de Taylor[9] de una función (infinitamente diferenciable)[10] alrededor de un punto $x=a$:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots, \tag{28}$$

Ahora consideremos un pequeño desplazamiento de la variable independiente, $x+\varepsilon$, y representemos la serie de Taylor en torno al punto x original. Es decir, utilizando la ecuación anterior reemplazamos *a* por *x*, y *x* en el término $(x-a)$ por $x+\varepsilon$:

$$f(x+\varepsilon) = f(x) + \varepsilon f'(x) + \varepsilon^2 \frac{f''(x)}{2} + \varepsilon^3 \frac{f^{(3)}(x)}{6} + \cdots, \tag{29}$$

Definiendo ahora Épsilon como el negativo del gradiente de la función multiplicado por una constante Alpha, es decir: $\varepsilon = -\alpha f'(x)$, se tiene:

$$f(x+\varepsilon) = f(x) - \alpha\big(f'(x)\big)^2 + \sum_{n=2}^{\infty} (-\alpha)^n \frac{f'(x) f^n(x)}{n!} \tag{30}$$

Si se utiliza una constante Alpha suficientemente pequeña, el tercer término del lado derecho de la ecuación se vuelve irrelevante. Además, si la derivada de la función es diferente de cero y la constante Alpha es positiva, el segundo término del lado derecho de la ecuación siempre será negativo. Esto implica que el valor de la función en el punto $(x+\varepsilon)$, es decir $f(x+\varepsilon)$, siempre será más pequeño que el valor de la función en el punto *x* (es decir, $f(x)$).

De acuerdo con lo anterior, con el objetivo de minimizar la función de costo, lo que busca el algoritmo de optimización de gradiente descendiente, es actualizar el parámetro o parámetros de la función de costo (es decir *x*), restándole al parámetro el gradiente de la función con respecto a dicho parámetro, es decir:

$$x = x - \alpha f'(x) \tag{31}$$

En términos generales, el algoritmo de gradiente descendente implica la selección tanto de un valor inicial para el parámetro *x* como el valor de una constante $\alpha>0$,

[9] https://en.wikipedia.org/wiki/Taylor_series
[10] https://mathworld.wolfram.com/TaylorSeries.html

para luego iterar continuamente el valor del parámetro *x* hasta una condición determinada (por ejemplo, el número de iteraciones o el número de épocas).

Un análisis similar al anterior se puede realizar para funciones multivariadas, llegando a la misma conclusión.

Gradiente descendente estocástico por minilotes

La aplicación del algoritmo de gradiente descendente puede considerarse desde dos enfoques, de acuerdo con el número de ejemplos del dataset utilizados en cada iteración del algoritmo de optimización. El primer enfoque (gradiente descendente) consiste en utilizar el conjunto de datos completo para calcular los gradientes y actualizar los parámetros. Sin embargo, el manejo de datos en este enfoque no es eficiente, dado que para una sola actualización de los parámetros es necesario leer y utilizar todo el dataset. Este aspecto puede tornarse de gran importancia al trabajar con conjuntos de datos muy grandes (por ejemplo, con millones de imágenes), lo que puede ralentizar el entrenamiento de un modelo.

En el otro extremo, el segundo enfoque (gradiente descendente estocástico) consiste en procesar cada vez un solo ejemplo del conjunto de datos, a partir de lo cual se actualizan los parámetros del modelo. Aunque este enfoque presenta un menor costo computacional en cada iteración comparado con el anterior, tampoco es eficiente desde el punto de vista de procesamiento de datos, dado que no aprovecha los recursos de paralelización y vectorización de los cuales disponen las unidades de procesamiento actuales (computacionales y gráficas).

En el medio de estos dos enfoques se encuentra el gradiente descendente estocástico por minilotes (*Minibatch stochastic gradient descent*), el cual puede ofrecer tanto un costo computacional bajo como una alta eficiencia estadística. Este enfoque consiste en leer subconjuntos o minilotes del conjunto de datos en lugar de ejemplos individuales ni tampoco leer todo el dataset. Una vez procesado el minilote, se calculan los gradientes y se actualizan los parámetros. De esta forma, el número de iteraciones que realiza el optimizador sobre el conjunto de datos está dado por:

$$\lceil DS/BS \rceil \tag{32}$$

donde *DS* corresponde al número de muestras del dataset completo, *BS* al tamaño (número de muestras) del minilote y el operador $\lceil\ \rceil$ corresponde a la función techo que devuelve el mínimo número entero no inferior al argumento.

Al utilizar el procesamiento por minilotes, el cálculo del gradiente se obtiene sobre el promedio de los datos:

$$\partial = \frac{\partial}{\partial \boldsymbol{w}}\left(\frac{1}{BS}\sum_{i\in BS} f(\boldsymbol{X}_i, \boldsymbol{W})\right) \tag{33}$$

Cuando se aplica el algoritmo de optimización por gradiente descendente estocástico por minilotes, el valor esperado del optimizador se mantiene sin cambios, mientras que la varianza se reduce significativamente (Zhang, 2021). En cuanto a tiempo de ejecución, en general la operación por minilotes es más rápida que la operación tanto por gradiente descendente estocástico como por gradiente descendente.

Ejemplo de algoritmo de optimización y tasa de aprendizaje

A manera de ejemplo, y como contexto de explicación del algoritmo de gradiente descendente, se tomará la siguiente función objetivo, que tiene su mínimo en *x=0*.

$$f(x) = \cosh\left({}^{\pi x}/_{4}\right) \tag{34}$$

Donde el cálculo del gradiente de *f(x)* da como resultado:

$$f'(x) = {}^{\pi}/_{4} * \sinh\left({}^{\pi x}/_{4}\right) \tag{35}$$

El cálculo de la función, sumado al cálculo del gradiente y respectiva actualización de *x* para 20 iteraciones arroja el siguiente resultado:

x	***α***	***f(x)***	***f'(x)***	***x-αf'(x)***
-1 (valor inicial)	0.05	1.3246	-0.6823	-0.9659 (valor siguiente)
-0.9659	0.5	1.3018	-0.6546	-0.6386
-0.6386	0.5	1.1284	-0.4106	-0.4333
-0.4333	0.5	1.0585	-0.2724	-0.2970
-0.2970	0.5	1.0273	-0.1849	-0.2046
-0.2046	0.5	1.0129	-0.1267	-0.1412
-0.1412	0.5	1.0062	-0.0873	-0.0976

-0.0976	0.5	1.0029	-0.0602	-0.0675
-0.0675	0.5	1.0014	-0.0416	-0.0466
-0.0466	0.5	1.0007	-0.0288	-0.0322
-0.0322	0.5	1.0003	-0.0199	-0.0223
-0.0223	0.5	1.0002	-0.0138	-0.0154
-0.0154	0.5	1.0001	-0.0095	-0.0107
-0.0107	0.5	1.0000	-0.0066	-0.0074
-0.0074	0.5	1.0000	-0.0046	-0.0051
-0.0051	0.5	1.0000	-0.0031	-0.0035
-0.0035	0.5	1.0000	-0.0022	-0.0024
-0.0024	0.5	1.0000	-0.0015	-0.0017
-0.0017	0.5	1.0000	-0.0010	-0.0012
-0.0012	0.5	1.0000	-0.0007	-0.0008

Tal como se aprecia en la tabla anterior, a medida que avanzan las iteraciones, el valor de *x* se aproxima gradualmente al valor en el cual la función objetivo es mínimo (*x=0*). Para comprobarlo, tomemos la función definida en Python:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline

# Función cosh y su gradiente
c = tf.constant(np.pi/4)
f = lambda x: tf.cosh(c * x)
gradf = lambda x: c * tf.sinh(c * x)
```

Como función de gradiente descendente, se utilizará la propuesta por (Zhang, 2021), donde es claro que, a partir del valor inicial, se actualiza el valor restando el gradiente de la función evaluada en ese punto.

```
def gd(alpha,valor_ini):
```



```
    x = valor_ini
    results = [x]
    for i in range(10):
        x -= alpha * gradf(x)
        results.append(float(x))
    return results
```

Los resultados del valor de *x* se almacenan en un vector para graficarlos sobre la función de costo, utilizando el siguiente código:

```
def grafica(valor_ini,parametros):
  eje_x = tf.range(valor_ini, -valor_ini, 0.01)
  y_curva = f(eje_x)
  y_puntos = f(parametros)

  # Plot the points using matplotlib
  plt.plot(eje_x, y_curva)
  plt.plot(parametros, y_puntos, marker = 'o', col
or = 'green')
  plt.xlabel('Argumento (x)')
  plt.ylabel('Función de costo')
  plt.title('Optimización')
  plt.legend(['Función de costo', 'Minimización it
erativa'])
  plt.show()
```

Para comprobar el comportamiento del algoritmo, se usará $x=-1$ como valor inicial, y $\alpha=0.05$ (tasa de aprendizaje).

```
valor_ini=-1
alpha=0.5
parametros = np.array(gd(alpha,valor_ini))
grafica(valor_ini,parametros)
```

De esta forma se obtiene el siguiente resultado, donde la curva azul corresponde a la función de costo a optimizar (*cosh*) y la curva verde corresponde a los valores de *x* para diferentes iteraciones. El número de valores de *x* mostrados en la gráfica es de 10, que corresponden a los primeros 10 valores de la tabla anterior.

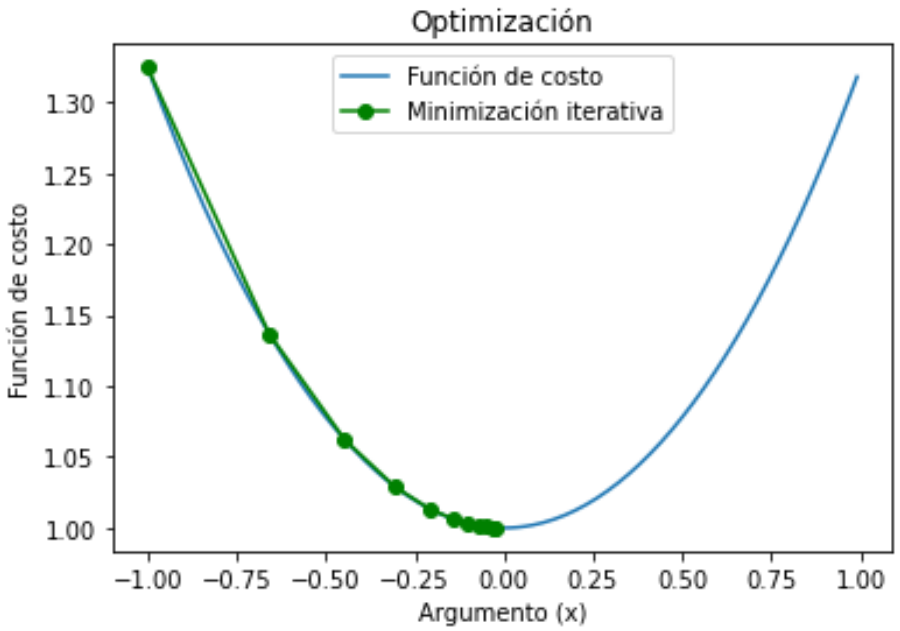


Figura 8. Ejemplo de algoritmo de optimización para función coseno hiperbólico y tasa de aprendizaje 0.5

A partir del valor inicial $x=-1$, el algoritmo de optimización muestra la aproximación del valor de x, hacia el punto mínimo de la función.

Además, al usar una tasa de aprendizaje muy pequeña puede hacer que la convergencia del algoritmo se ralentice, como se muestra a continuación:

```
valor_ini=-1
alpha=0.01
parametros = np.array(gd(alpha,valor_ini))
grafica(valor_ini,parametros)
```

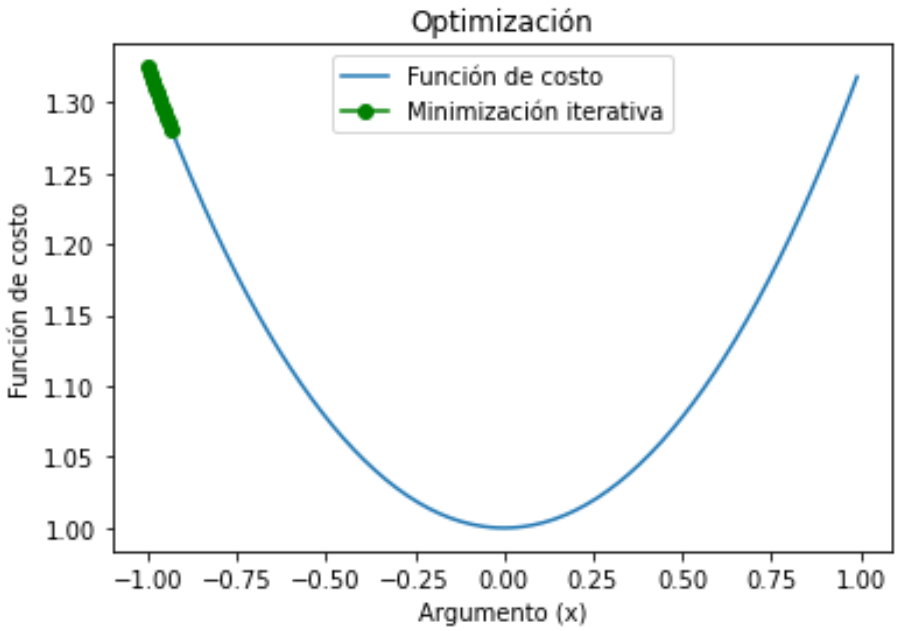



Figura 9. Ejemplo de algoritmo de optimización para función coseno hiperbólico y tasa de aprendizaje 0.01

Por el contrario, al usar una tasa de aprendizaje demasiado grande, no se garantiza que la iteración reduzca el valor de *x*. Para visualizar esto, se definirá una nueva función y su gradiente:

$$f(x) = {\sin(3\pi x)}/{x} \tag{36}$$

$$f'(x) = {(x\cos(3\pi x) - \sin(3\pi x))}/{x^2} \tag{37}$$

Para observar el comportamiento del algoritmo en esta nueva función, iniciaremos con una tasa de aprendizaje de 0.001 y un valor inicial de 0.1. Con estos valores, el comportamiento del algoritmo se aproxima gradualmente al mínimo global de la función:

```
c = tf.constant(3*np.pi)
f = lambda x: tf.sin(c * x)/x
gradf = lambda x: (x * tf.cos(c * x) - tf.sin(c *
x))/(x**2)

alpha=0.001
parametros = np.array(gd(alpha,0.1))
```

```
grafica(-2,parametros)
```

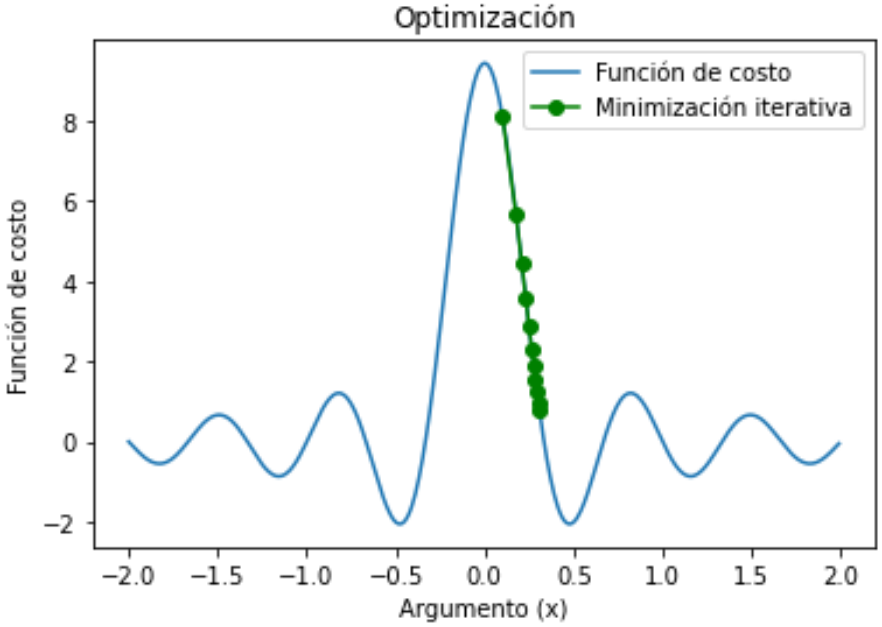


Figura 10. Ejemplo de algoritmo de optimización para función sinc y tasa de aprendizaje 0.001

Sin embargo, al aumentar significativamente el valor de la tasa de aprendizaje a 0.1 (e iniciar con un valor de 0.3), el algoritmo tendrá inconvenientes para reducir el valor del argumento:

```
alpha=1
parametros = np.array(gd(alpha,-1))
grafica(-4,parametros)
```

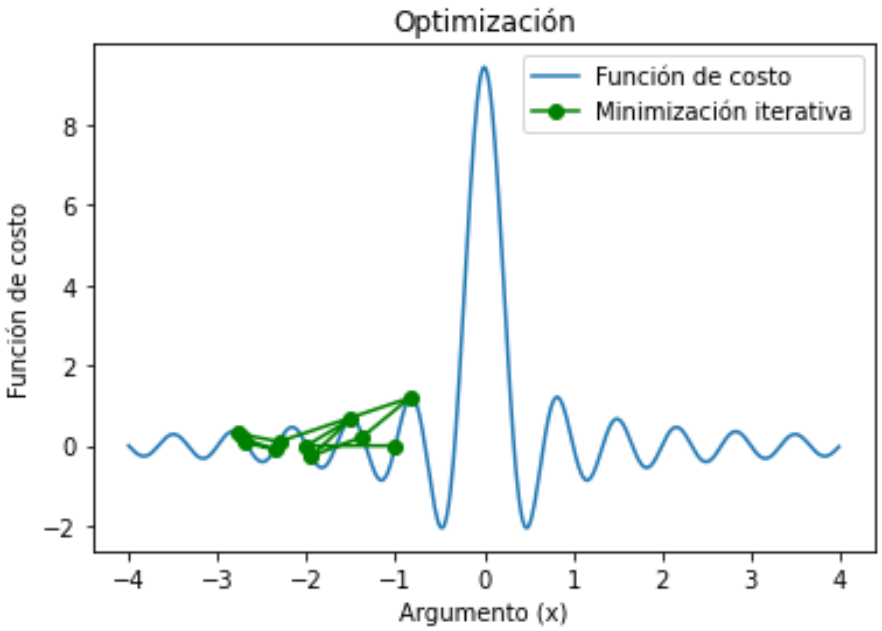



Figura 11. Ejemplo de algoritmo de optimización para función sinc y tasa de aprendizaje 1.0

Además, para una función que tenga muchos mínimos locales, el usar una tasa de aprendizaje grande, podría llevar también a una errónea localización del mínimo global:

```
alpha=0.1
parametros = np.array(gd(alpha,0.3))
grafica(-3,parametros)
```

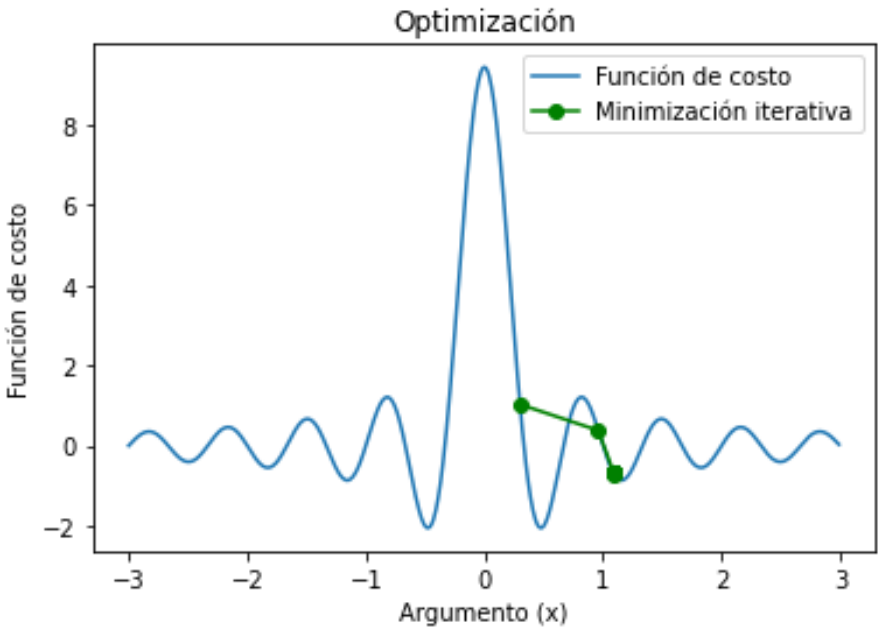



Figura 12. Ejemplo de algoritmo de optimización para función sinc y tasa de aprendizaje 0.1

Gradiente descendente estocástico

Como complemento al algoritmo de gradiente descendente, es importante considerar que normalmente la función objetivo se calcula como el promedio de la función de pérdida computada para todos y cada uno de los ejemplos del conjunto de datos de entrenamiento. Es decir,

$$f(x) = \frac{1}{m}\sum_{1}^{m} f_i(x) \tag{38}$$

De esta forma, el gradiente de la función objetivo se obtiene como el promedio del gradiente de la función de pérdida calculado para cada ejemplo del conjunto de datos de entrenamiento. Es decir:

$$\nabla f(x) = \frac{1}{m}\sum_{1}^{m} \nabla f_i(x) \tag{39}$$

El cálculo del gradiente como un promedio sobre el número de ejemplos, implica que el número de muestras es directamente proporcional al costo computacional del cálculo del gradiente. De aquí que el gradiente descendente estocástico (SGD) surge como una alternativa para reducir el costo computacional en cada iteración.

En particular, SGD propone en cada iteración seleccionar aleatoriamente un solo ejemplo del conjunto de datos mediante un muestreo uniforme. De esta forma, el costo computacional del cálculo del gradiente se reducirá, lo que es significativo para conjuntos de datos muy grandes. El precio a pagar por esta reducción involucra que la trayectoria de aproximación al mínimo de la función puede presentar mayores variaciones.

Tasa de aprendizaje dinámica

Como se comentó anteriormente, el tener una tasa de aprendizaje demasiado pequeña puede ralentizar el proceso de aprendizaje de tal manera que en cada iteración la aproximación hacia el mínimo de la función no sea significativo. Además, el tener una tasa de aprendizaje demasiado grande, por lo general no es una buena alternativa. Ante esta situación, el manejar una tasa de aprendizaje cuyo valor no sea constante para todas las iteraciones puede significar beneficios para el algoritmo de entrenamiento. Particularmente se puede iniciar con un valor determinado, e ir reduciendo su valor de acuerdo con una regla específica. De esta forma, es posible programar la tasa de aprendizaje para modular cómo cambia a lo largo del tiempo de entrenamiento. De esta forma, la sustitución de α por una tasa de aprendizaje variable en el tiempo, añade complejidad al control de la convergencia del algoritmo de optimización.

Actualmente, `Keras/tensorflow` dispone de las cuatro opciones listadas a continuación para programar la tasa de aprendizaje[11]:

- *Exponential Decay*: programa que aplica una función de decaimiento exponencial luego de cada paso del optimizador, de acuerdo con la siguiente Ecuación:

$$\alpha_i = \alpha_0 * r^{s_i/s} \tag{40}$$

Donde α_0 es la tasa de aprendizaje inicial, α_i es la tasa de aprendizaje en el paso i, r es la tasa de decaimiento, s_i es el número de paso y s es el número de pasos de decaimiento. De esta forma, la tasa de aprendizaje decrece gradualmente hasta alcanzar el r % de la tasa de aprendizaje original una vez transcurrido el número de pasos establecido.

[11] https://keras.io/api/optimizers/learning_rate_schedules/

Utilizando una tasa de aprendizaje inicial de 0.05, una tasa de decaimiento de 0.9 y 20000 pasos, el decaimiento exponencial se puede implementar de la siguiente manera:

```
alfa_inicial = 0.05
lr_schedule = tf.keras.optimizers.schedules.Exponenti
alDecay(
    alfa_inicial,
    num_pasos=20000,
    tasa_dec=0.9,
    staircase=True)
```

Así, la tasa de aprendizaje del optimizador se puede especificar como "`lr_schedule`" y no como un valor constante:

```
optimizer=tf.keras.optimizers.SGD(learning_rate=lr_sc
hedule)
```

- *Piecewise Constant Decay*: programa que calcula una tasa de aprendizaje constante por tramos luego del paso actual del optimizador, en función del número de paso actual y los límites de los tramos definidos, así:

$$\alpha = \begin{cases} \alpha[1] & cuando & s_i \leq límite[1] \\ \alpha[2] & cuando & límite[1] < s_i \leq límite[2] \\ \cdots & \cdots & \cdots \\ \alpha[n] & cuando & límite[n-1] < s_i \leq límite[n] \end{cases} \quad (41)$$

Donde $\alpha[j]$ es la tasa de aprendizaje definida para el tramo j, s_i es el número de paso y límite[k] corresponde al valor de paso que define la frontera de un determinado tramo. De esta forma, la tasa de aprendizaje para el primer tramo tendrá un valor de $\alpha[1]$ para el primer tramo, $\alpha[2]$ para el segundo tramo y así sucesivamente.

Con una tasa de aprendizaje inicial de 0.1 (en los primeros 1000 pasos), una tasa de aprendizaje de 0.01 en los siguientes 4000 pasos y una tasa de aprendizaje de 0.001 en los demás pasos, el decaimiento constante por tramos se puede utilizar de la siguiente manera:

```
step = tf.Variable(0, trainable=False)
limites = [1000, 5000]
tramos = [0.1, 0.01, 0.001]
learning_rate_fn = keras.optimizers.schedules.Piecewi
seConstantDecay(
```



```
    limites, tramos)
```

Así, la tasa de aprendizaje del optimizador se puede especificar en el optimizador:

```
learning_rate = learning_rate_fn(step)
optimizer=tf.keras.optimizers.SGD(learning_rate)
```

- *Polynomial Decay*: programa que aplica una función de decaimiento polinómico a un paso del optimizador, en función de la tasa de aprendizaje inicial, hasta alcanzar una tasa de aprendizaje final en un número determinado de pasos, de acuerdo con la siguiente Ecuación:

$$\alpha_i = \left((\alpha_0 - \alpha_{end})(1 - s_i/s)^p\right) + \alpha_{end} \quad (42)$$

Donde α_0 es la tasa de aprendizaje inicial, α_{end} es la tasa final de aprendizaje deseada, p es el orden del polinomio, s_i es el número de paso y s es el número de pasos de decaimiento. De esta forma, la tasa de aprendizaje decrece gradualmente hasta alcanzar el valor final una vez transcurrido el número de pasos establecido.

Utilizando una tasa de aprendizaje original de 0.05, una tasa final de decaimiento de 0.005 y 5000 pasos, el decaimiento polinómico se puede utilizar de la siguiente manera:

```
alfa_inicial = 0.05
alfa_final = 0.005
num_pasos = 5000
learning_rate_fn = tf.keras.optimizers.schedules.Poly
nomialDecay(
    alfa_inicial,
    num_pasos,
    alfa_final,
    power=0.5)
```

A partir de lo cual es posible especificar la tasa de aprendizaje del optimizador:

```
optimizer=tf.keras.optimizers.SGD(
                  learning_rate=learning_rate_fn)
```


- *Inverse Time Decay*: programa que aplica la función de decaimiento inverso a un paso del optimizador, en función de una tasa de aprendizaje inicial, de acuerdo con la siguiente Ecuación:

$$\alpha_i = \frac{\alpha_0}{(1 + r * s_i/s)} \tag{43}$$

Donde α_0 es la tasa de aprendizaje inicial, α_i es la tasa de aprendizaje en el paso i, r es la tasa de decaimiento, s_i es el número de paso y s es el número de pasos de decaimiento.

```
alfa_inicial = 0.01
num_pasos = 100
tasa_dec = 0.5
learning_rate_fn = keras.optimizers.schedules.Inverse
TimeDecay(
  alfa_inicial, num_pasos, tasa_dec)
```


3. INTRODUCCIÓN A LAS REDES NEURONALES

En el capítulo anterior se estableció que la función de pérdida se encarga de cuantificar la distancia entre el valor real y el valor estimado por la función objetivo. Más allá de evaluar este error, es importante en ciertos algoritmos de aprendizaje automático, como los de clasificación, interpretar la función de error de forma probabilística.

El objetivo aquí es mapear el vector de error entregado por el modelo hacia un vector de probabilidades. Por ejemplo, para un problema de clasificación, cada elemento de dicho vector contiene la probabilidad de pertenencia a cada una de las clases evaluadas. En cualquier caso, es importante garantizar que los valores de probabilidad estén normalizados entre 0 y 1 y que la suma de todos ellos sea 1. Esto se logra a través de la función *softmax*, que calcula una distribución de probabilidad sobre todas las clases (Zhang, 2021). Esta función se define como:

$$\hat{y} = softmax(o_i) = \frac{e^{o_i}}{\sum_j e^{o_j}} \tag{44}$$

Donde los elementos o_i corresponden a los valores entregados por el modelo.

Para ilustrar el funcionamiento de la operación *softmax*, se tiene un modelo cuyos pesos y *bias* ya han sido establecidos, y a partir de los cuales se calculará la salida estimada del modelo como $\hat{y} = \boldsymbol{w}x + b$.

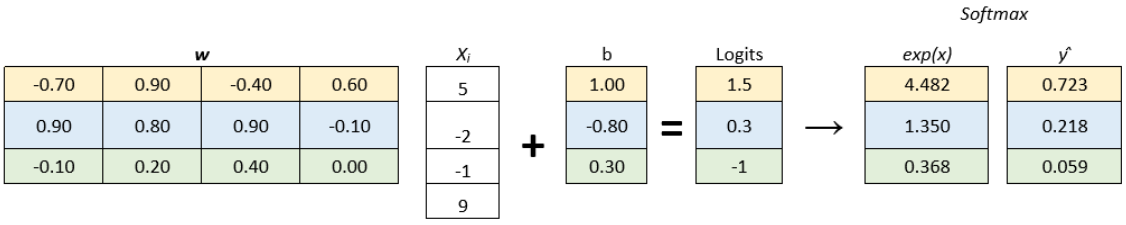


Figura 13. Ejemplo de operación softmax

El resultado de esta operación corresponde al vector de predicciones no normalizadas generado por el modelo de clasificación, y que se denomina en la

figura como el vector "*Logits*". Normalmente, este vector es la entrada a una función de normalización, que para el caso de un problema de clasificación multiclase suele ser la función *softmax*. De aquí que esta función genera el vector de probabilidades (normalizadas) con un valor para cada clase posible (vector $\hat{y}$ en la figura).

El resultado de la operación *softmax* también se usa en una de las funciones de pérdida más comunes en aprendizaje profundo, la pérdida de entropía cruzada (*cross entropy loss*). Esta función de pérdida evalúa la diferencia entre la probabilidad asignada por el modelo (dada por la operación *softmax*) y el valor esperado de la pérdida. Para un problema de clasificación multiclase, la pérdida de entropía cruzada está dada por:

$$L_{CE} = -\sum_{1}^{C} y_i \log(\hat{y}_i) \tag{45}$$

Donde *y* es el vector de etiquetas verdaderas (por ejemplo, con codificación *one-hot*), p_i es la probabilidad *softmax* para la *i*-ésima clase y *C* es el número de clases. Esto quiere decir que la entropía cruzada calcula el negativo de la función de verosimilitud logarítmica de la probabilidad predicha asignada a la etiqueta verdadera.

Para el caso binario, la función de pérdida se reduce a dos casos:

$$l = -\sum_{1}^{C} y_i \log(\hat{y}_i) \tag{46}$$

$$l = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{47}$$

La cual se puede generalizar para todos los ejemplos del conjunto de datos (N)[12]:

$$L = -\frac{1}{N}\sum_{1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{48}$$

[12] https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

PERCEPTRÓN MULTICAPA

La red neuronal más sencilla se puede construir apilando múltiples capas de neuronas, donde cada capa está totalmente conectada a la capa anterior. En esta arquitectura, por lo general, las primeras L-1 capas corresponden a la etapa de representación o extracción de características, y la última capa corresponde al predictor lineal. La incorporación de una o más capas ocultas está orientada a mejorar la generalización de los modelos, desde el punto de vista de linealidad. Este tipo de arquitectura se conoce como perceptrón multicapa (MLP), y fue inicialmente propuesto con una capa oculta por (Rosenblatt, 1958), a partir de lo cual evolucionó manteniéndose vigente en la actualidad. Esta arquitectura se muestra en la siguiente Figura (Saravia, 2021).

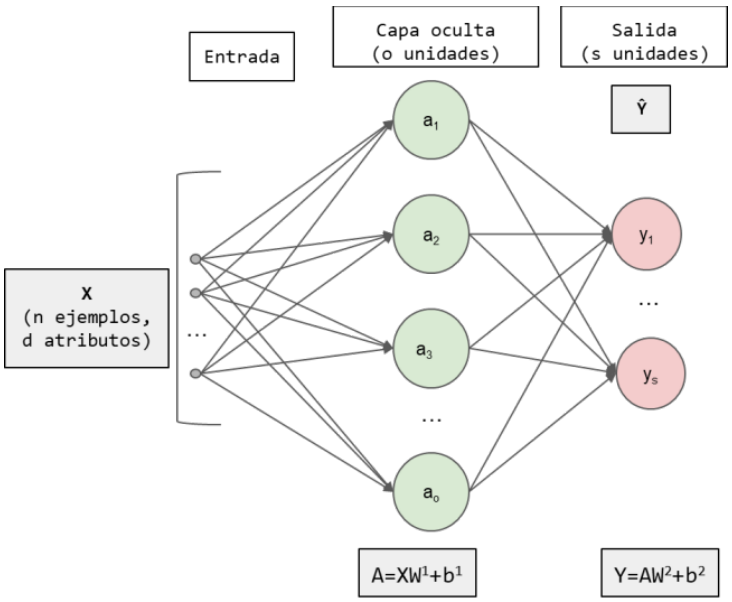


Figura 14. Ejemplo de estructura de red perceptrón.

En la arquitectura de la Figura se identifican tres capas: la de entrada, una capa oculta y la capa de salida.

Es común en este tipo de modelos, que, para definir la profundidad de la red o número de capas apiladas, no se considere la capa de entrada, por lo cual se dirá que este modelo tiene dos capas de procesamiento o profundidad. A continuación, se listan las características de los datos asociados a las tres capas mencionadas.

- Capa de entrada: corresponde al vector de datos de entrada (**x**), por lo cual el número de elementos o unidades de la capa corresponde al número de atributos de los datos (*d*). Para un conjunto de datos con *n* ejemplos y *d* atributos, el dataset se puede ver como una matriz **X** de *n* filas y *d* columnas.
- Capa oculta: esta capa involucra neuronas o unidades que se conectan a los datos de entrada por lo cual constituye una suma ponderada, donde los pesos ($\mathbf{w^1}$) y sesgo (b^1) de dicha suma son parámetros que aprende la red. El número de neuronas o unidades se puede definir al diseñar la red, y se denominará como *o*. En este sentido, tras la interconexión de la capa oculta con los datos de entrada, se tendrá una transformación del vector de datos de entrada (**x**) hacia un nuevo vector ***a***, dada por $\mathbf{a}=\mathbf{w}^1x+b^1$. Si se generaliza la operación de esta capa para todo el conjunto de datos de entrada, se tendrá una matriz de características **A**, dada por:

$$\boldsymbol{A} = \boldsymbol{X}\boldsymbol{W}^1 + \boldsymbol{b}^1 \tag{49}$$

  A será una matriz cuyo número de filas corresponde al número de ejemplos (*n*) del dataset y el número de columnas es el número de unidades de la capa (*o*).

  $\mathbf{W}^1$ y $\mathbf{b}^1$ corresponden a los parámetros (pesos y *bias*) de la primera capa. Las dimensiones de $\mathbf{W}^1$ son $d \times o$, mientras que $\mathbf{b}^1$ es un vector de *o* elementos.

- Capa de salida: esta capa involucra neuronas o unidades que se conectan a los datos ocultos (**A**) por lo cual constituye una suma ponderada, donde los pesos ($\mathbf{w^2}$) y sesgo (b^2) de dicha suma son parámetros que aprende la red. El número de neuronas o unidades se puede definir al diseñar la red, y se denominará *s*. En este sentido, tras la interconexión de la capa de salida con los datos de la capa oculta, se tendrá una transformación del vector (**a**) hacia un nuevo vector $\hat{\mathbf{y}}$, dado por $\hat{\mathbf{y}}=\mathbf{w}^2a+b^2$. Si se generaliza la operación de esta capa para todo el conjunto de datos de entrada, se tendrá un vector de salida $\hat{\mathbf{Y}}$, dado por:

$$\hat{\boldsymbol{Y}} = \boldsymbol{A}\boldsymbol{W}^2 + \boldsymbol{b}^2 \tag{50}$$

  Y será un vector cuyo número de elementos corresponde al número de unidades de la capa de salida (s).

$\mathbf{W}^2$ y $\mathbf{b}^2$ corresponden a los parámetros (pesos y *bias*) de la segunda capa. Las dimensiones de $\mathbf{W}^2$ son $o \times s$, mientras que $\mathbf{b}^2$ es un vector de s elementos.

Considerando el procesamiento de las dos capas mencionadas, la salida se puede calcular a partir de la entrada mediante (Zhang, 2021):

$$\widehat{\boldsymbol{Y}} = (\boldsymbol{X}\boldsymbol{W}^1 + \boldsymbol{b}^1)\boldsymbol{W}^2 + \boldsymbol{b}^2 \tag{51}$$

$$= \boldsymbol{X}\boldsymbol{W}^1\boldsymbol{W}^2 + \boldsymbol{b}^1\boldsymbol{W}^2 + \boldsymbol{b}^2 \tag{52}$$

Definiendo el producto $\boldsymbol{W}^1\boldsymbol{W}^2$ como una sola matriz $\boldsymbol{W}$ y $\boldsymbol{b}^1\boldsymbol{W}^2 + \boldsymbol{b}^2$ como un solo vector de sesgo, la salida se puede calcular como:

$$\widehat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b} \tag{53}$$

Lo anterior significa que, incluso agregando más capas al modelo, existe el riesgo de que el modelo se comporte de manera lineal, evitando obtener funciones más generales que puedan modelar de una mejor manera el comportamiento de los datos. Para evitar que la red presente un comportamiento lineal, los modelos han involucrado el uso de funciones no lineales, conocidas como funciones de activación.

FUNCIONES DE ACTIVACIÓN

A partir del valor de la suma ponderada más el sesgo en una capa, una función de activación decide si una neurona debe activarse o no. Es decir, al valor entregado por cada unidad oculta de una capa ($\boldsymbol{wx} + \boldsymbol{b}$), se le aplica una función no lineal ($f(x)$, función de activación). El valor resultante será la entrada para la siguiente capa. Para el caso de la arquitectura mostrada en la figura anterior, el cálculo de cada capa quedaría expresado como:

$$\boldsymbol{A} = f_1(\boldsymbol{X}\boldsymbol{W}^1 + \boldsymbol{b}^1) \tag{54}$$

$$\widehat{\boldsymbol{Y}} = f_2(\boldsymbol{A}\boldsymbol{W}^2 + \boldsymbol{b}^2) \tag{55}$$

Donde $f_1(x)$ y $f_2(x)$ corresponden a las funciones de activación de la primera y segunda capa respectivamente.

En la literatura se han propuesto una gran diversidad de funciones de activación, entre las cuales se resaltan las funciones sigmoide, tangente hiperbólica y ReLU (*Rectified Linear Unit*) por su amplia adopción. A continuación, se relacionan las principales características de estas funciones.

Función sigmoide

Función de activación que transforma el dominio de los datos de entrada hacia el intervalo (0, 1), Está definida como:

$$f_{sigmoide}(x) = \frac{1}{1+e^{-x}} \tag{56}$$

Esta función se utiliza comúnmente en las unidades de la capa de salida de clasificadores binarios, cuando se desea interpretar el valor de salida como una probabilidad (Zhang, 2021). Es decir, para problemas binarios realiza una tarea similar a la de la operación *softmax* en problemas multiclase. Esto se puede observar en la gráfica de la función, donde es fácil discriminar valores superiores/inferiores a 0.5.

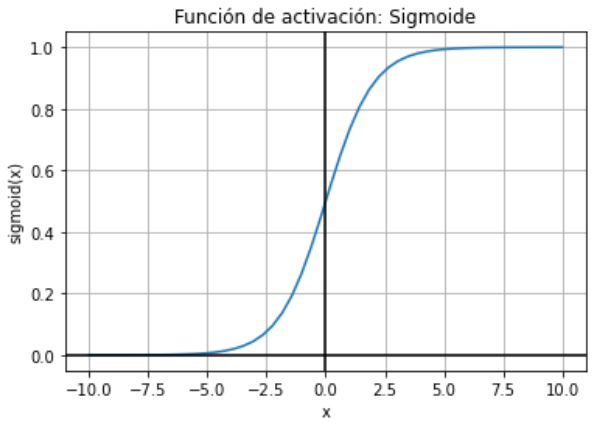


Figura 15. Función de activación sigmoide

Sin embargo, algunos inconvenientes de esta función están relacionados con el comportamiento asintótico que presenta la curva al aproximarse a 0 y a -1, y a que existen funciones más sencillas de implementar a nivel computacional. De aquí que, aunque inicialmente se consideró el uso de la función sigmoide para capas ocultas, actualmente esta ha sido reemplazada por la función de activación ReLU.

Función tangente hiperbólica

Función de activación similar a la *sigmoide*, pero que tiene la ventaja de estar centrada en cero. Esta función transforma el dominio de los datos de entrada hacia el intervalo $(-1, 1)$ y se define como:

$$f_{tanh}(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{57}$$

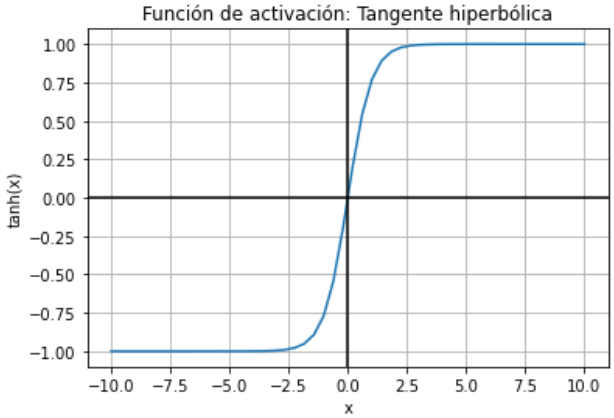


Figura 16. Función de activación tangente hiperbólica

La desventaja de esta función consiste en que sigue presentando un comportamiento asintótico en sus extremos, lo que ocasiona que prácticamente el gradiente de la función en esas zonas desaparezca.

Función ReLU (Rectified Linear Unit)

La función de activación ReLU (Rectified Linear Unit) se comporta como un rectificador de los valores de entrada, es decir que anula los valores negativos y mantiene los valores positivos. De aquí su sencillez de implementación, a la vez que se comporta como una función no lineal de bajo costo computacional. Esta función se define como:

$$f_{ReLU}(x) = max(0, x) \tag{58}$$

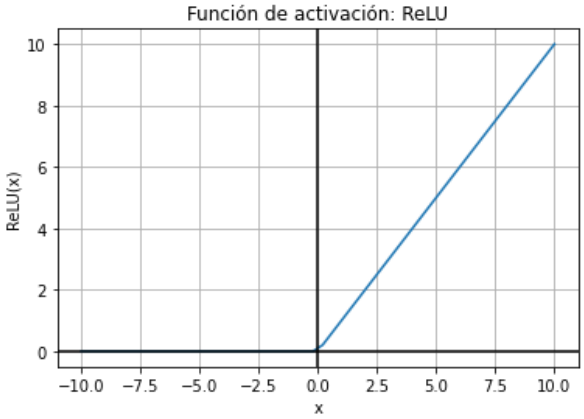


Figura 17. Función de activación ReLU

Ventajas adicionales de ReLU implican la no saturación en el semiplano derecho o una convergencia más rápida en comparación con las dos funciones descritas anteriormente. Como desventaja se tiene que la función no está centrada en cero y su comportamiento constante en el semiplano izquierdo.

Como variantes de la función ReLU se tiene LeakyReLU, la cual no anula la función cuando la unidad no está activa (semiplano izquierdo), sino que permite configurar una pequeña pendiente predefenida. Esto garantiza que la función no tenga saturación, mejora la eficiencia computacional y acelera la convergencia. Otra variante es PReLU (Parametric Rectified Linear Unit), la cual incluye también la inclusión de un pequeño gradiente en la parte negativa, que en este caso corresponde a un arreglo de las mismas dimensiones de la entrada y que es un parámetro que aprende la red.

Función ELU (Exponential Linear Unit)

Variante de la función ReLU definida por la siguiente ecuación:

$$f_{ELU}(x) = \begin{cases} x & si \quad x > 0 \\ \alpha(e^x - 1) & si \quad x < 0 \end{cases} \tag{59}$$

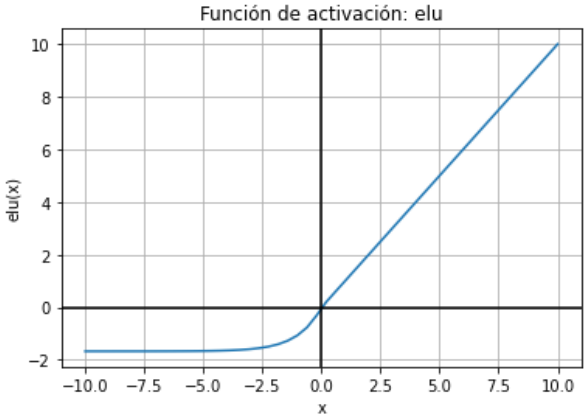


Figura 18. Función de activación ELU

Donde α es un hiperparámetro que controla el valor al que se satura la función para valores negativos, con el fin de disminuir el efecto del gradiente nulo o de fuga (Clevert, 2015). Esto es importante tenerlo en cuenta, ya que cuando hay saturación, la variación de la función y la información que se propaga a la siguiente capa disminuye. Asimismo, en ELU, al contar con valores negativos (en comparación con ReLU), la media de los valores es más cercana al gradiente natural (cero) lo que redunda en mayor velocidad de aprendizaje.

Función SELU (Scaled ELU)

Variante de la función ELU definida por la siguiente ecuación:

$$f_{SELU}(x) = \begin{cases} s * x & si \quad x > 0 \\ s * \alpha(e^x - 1) & si \quad x < 0 \end{cases} \tag{60}$$

Donde α y s (*scale*) son constantes predefinidas (α=1.67326324 y s=1.05070098 en Keras[13]). Estos valores deben elegirse de tal forma que la media y la varianza de las entradas se conserven entre dos capas consecutivas (Klambauer, 2017). La diferencia fundamental ente SELU y ELU consiste en la adición de un valor de escala que asegura una pendiente mayor a 1 en los valores positivos.

[13] https://keras.io/api/layers/activations/

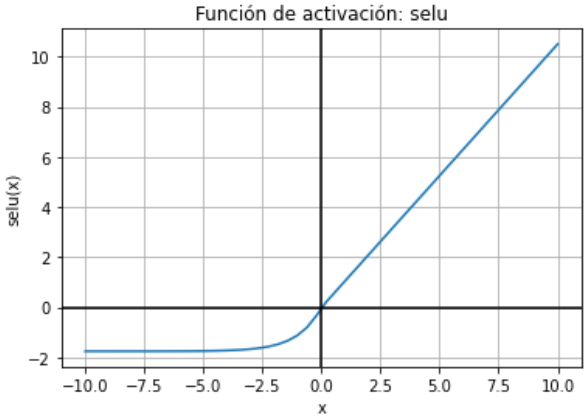


Figura 19. Función de activación SELU

Función softmax

Como se comentó al inicio de este capítulo, la función *softmax* convierte un vector de valores en una distribución de probabilidad, de tal forma que los elementos del vector de salida están en el rango (0, 1) y suman 1. La función *softmax* se utiliza comúnmente como la activación para la última capa de un modelo de clasificación multiclase, ya que su salida puede interpretarse como una distribución de probabilidad[14]. La función está definida tal como se explicó anteriormente.

$$f_{softmax}(x) = \frac{e^x}{\sum e^x} \tag{61}$$

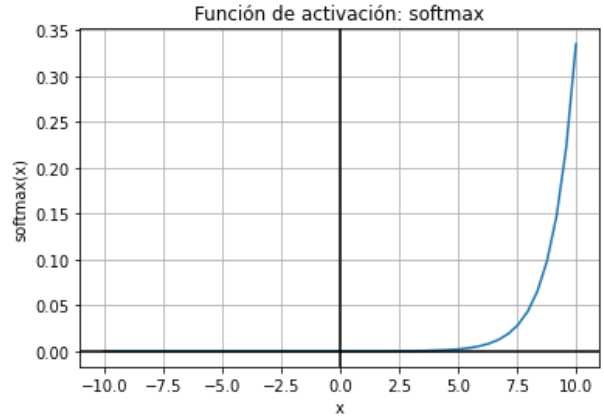


Figura 20. Función de activación Softmax

[14] https://keras.io/api/layers/activations/

PERCEPTRÓN MULTICAPA - EJEMPLO EN PYTHON

Una vez revisados los conceptos sobre redes neuronales y aspectos asociados al aprendizaje de estas, como la función de pérdida, los algoritmos de optimización, las funciones de activación y los hiperparámetros del modelo, se mostrará a continuación un ejemplo que involucra estos conceptos en un solo modelo programado en Python.

En primer lugar, será necesario importar las librerías necesarias para la ejecución del código. En este caso se trabajará con `numpy` para el manejo de datos, la selección aleatoria de imágenes mediante `random`, `tensorflow` para la creación del modelo y `matplotlib` para graficar el rendimiento en el entrenamiento del modelo.

```
# Carga de librerías
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt
```

Adicional a las citadas librerías, se importará la clase `Sequential` para crear un modelo secuencial que contiene solamente capas tipo `Flatten` para convertir datos matriciales (imágenes) en vectores y capas totalmente interconectadas (`Dense`) como la base del perceptrón. Por su parte, las etiquetas se representarán en formato *one-hot*, utilizando `to_categorical`.

```
# Carga de módulos
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.utils import to_categorical
```

Conjunto de datos

Como conjunto de datos del ejemplo se utilizará uno de los más tradicionales en el área de aprendizaje automático con imágenes: MNIST. Este dataset constituido a partir del trabajo de Yann LeCun en el reconocimiento con redes neuronales de dígitos escritos a mano en el código postal de EE. UU. (LeCun Y. B., 1989). MNIST cuenta con 70000 imágenes en escala de gris con una dimensión de 28×28 y

datos en formato de 8 bits sin signo (uint8), etiquetados del 0 al 9. Debido a su gran difusión y utilización, este dataset está disponible en *frameworks* como Keras/tensorflow, donde es posible cargarlo con una sola línea de código. En este caso, la carga involucra una separación de los datos con 60000 imágenes de entrenamiento y 10000 imágenes de prueba.

```
(x_train, y_train), (x_test, y_test) = tf.keras.da
tasets.mnist.load_data()
print(x_train.shape, x_test.shape)
print(y_train.shape, y_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
(60000, 28, 28) (10000, 28, 28)
(60000,) (10000,)
```

Tras ejecutar el código de carga, los datos se descargan desde el repositorio de tensorflow y se almacenan en tuplas de arreglos de Numpy. Las dos primeras variables corresponden a los 60000 datos de entrenamiento (`x_train`: imágenes, `y_train`: etiquetas) y las dos últimas a los 10000 datos de prueba (`x_test`: imágenes, `y_test`: etiquetas). Es posible corroborar las dimensiones del dataset, teniendo en cuenta que las dimensiones del arreglo corresponden al número de datos (60000 en entrenamiento y 10000 en prueba), número de filas y número de columnas (28×28). En el caso de las etiquetas, estas corresponden a un vector por lo cual solo se muestra una dimensión (número de ejemplos).

Para involucrar el número de canales o bandas de la imagen, a continuación, se realiza un redimensionado de los datos, agregando una cuarta dimensión que permite obtener tensores de 4 dimensiones: número de ejemplos, alto, ancho, canales.

```
x_train = x_train.reshape( (x_train.shape[0], 28,
28, 1))
print(x_train.shape)
```

```
(60000, 28, 28, 1)
```

Dado que los datos o intensidades de píxel de las imágenes están representados en uint8 (0-255), es recomendable normalizarlos, acotando su valor, por ejemplo, entre 0 y 1. Esto se debe a que trabajar con valores altos en una red neuronal puede hacer que los pesos de la red sean más difíciles de ajustar.


```
x_train = x_train.astype('float32') / 255.0
```

En el caso de las etiquetas, estas se representan de acuerdo con el dígito de la imagen. Es decir, la etiqueta de cada imagen corresponde a un valor entero entre 0 y 9. Esto se puede evidenciar visualizando los valores del arreglo:

```
y_train
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

Los valores de etiquetas con los cuales se trabaje en el algoritmo de clasificación pueden representarse fundamentalmente de dos formas: con valores enteros (como el de la variable y_train), o con una representación conocida como "*one hot encoding*", que consiste en convertir cada valor de etiqueta hacia un vector binario de n elementos, siendo n el número de clases del dataset; todos los elementos de este vector serán cero, excepto por el elemento que ocupa la posición del valor entero original. Por ejemplo, para diez clases, el "0" se representará como [1 0 0 0 0 0 0 0 0 0], o el "8" se representará como [0 0 0 0 0 0 0 0 1 0]. Al convertir las etiquetas de los datos de entrenamiento, es posible comprobar cómo el segundo elemento y el último elemento de `y_train` y corresponden a los ejemplos dados.

```
y_train = to_categorical(y_train)
y_train
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.]], dtype=float32)
```

Es importante tener en cuenta aquí que el tipo de representación de las etiquetas deberá estar acorde con el tipo de función de pérdida utilizada. En Keras[15], por ejemplo, para un problema de clasificación binario (dos clases), es posible utilizar la función de pérdida *BinaryCrossentropy*, con etiquetas representadas como 0 o 1, y valores de predicción ya sea en *logits* o en valores de probabilidad. Por su parte, para un problema de clasificación multiclase, la herramienta permite utilizar etiquetas representadas en entero, caso en el cual se utilizará una función

[15] https://keras.io/api/losses/

de pérdida tipo *Sparse* (*sparse_categorical_crossentropy*, por ejemplo), o etiquetas con representación *one-hot*, caso en el cual se utilizará una función de pérdida categórica (*categorical_crossentropy*, por ejemplo). En cualquier caso, es posible especificar si la predicción está como *logits* o como probabilidades.

Hiperparámetros

Como hiperparámetros, en este ejemplo se utilizará un tamaño de lote de 32, una tasa de aprendizaje de 0.1 para un optimizador SGD y se entrenará durante veinte épocas.

```
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 20
optimizerf = tf.keras.optimizers.SGD(learning_rate
=lr)
```

Modelo

En este ejemplo se crea un modelo secuencial que corresponde a un perceptrón de dos capas: una capa oculta de 256 unidades y una capa de salida cuyo número de unidades es igual al número de clases del problema (10). Dado que los datos de entrada tienen más de una dimensión (ancho, alto, canales), la primera capa aplana la entrada, convirtiéndola en un vector de w×h×c elementos, donde w es el número de columnas, h es el número de filas y c es el número de canales. Con este vector, es posible utilizar capas totalmente interconectadas.

```
modelo = Sequential([
                    Flatten(),  # 28×28×1=784
                    Dense(256, activation='sigmoid
'),
                    Dense(10, activation='softmax'
)])
```

En un modelo creado en Keras, es posible especificar la función de activación para cada capa. En este ejemplo se ha utilizado la función `'sigmoid'` para la capa oculta. En la capa de salida (clasificación), cuando el número de unidades corresponde al número de clases, es común utilizar la función `'softmax'`. Cuando se trata de un problema binario, es posible utilizar una capa `Dense` de 1 unidad y la función de activación `'sigmoid'`, o también utilizar una capa `Dense` de n unidades (clases) y la función de activación `'softmax'`.

Luego de crear el modelo, es necesario configurarlo para entrenamiento. Aquí es posible especificar hiperparámetros como el optimizador, la función de pérdida o métricas de rendimiento:

```
modelo.compile(
  optimizer=optimizerf,
  loss = 'categorical_crossentropy',
  metrics=['accuracy'])
```

A continuación, es posible entrenar el modelo durante un número establecido de épocas o iteraciones en el conjunto completo de datos[16]. Además, es necesario especificar datos y etiquetas de entrenamiento o el tamaño del lote si los datos no han sido ya separados. Argumentos adicionales y opcionales que pueden especificarse, involucran la segmentación del dataset en entrenamiento y validación, aleatorización de los datos, ponderación de clases, entre otros.

```
history = model.fit(x_train,y_train,epochs=num_epo
chs,batch_size=batch_s)
```

```
Epoch 1/20
1875/1875 [==============================] - 7s 3ms/step - loss: 0.5250 - accuracy: 0.8560
Epoch 2/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3090 - accuracy: 0.9100
Epoch 3/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2738 - accuracy: 0.9209
Epoch 4/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2454 - accuracy: 0.9290
Epoch 5/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2199 - accuracy: 0.9373
```

La ejecución del entrenamiento devuelve un diccionario con el registro de valores de pérdida de entrenamiento y valores de las métricas definidas en la compilación, y los correspondientes valores cuando se especifican datos de validación.

Adicionalmente, al obtener el modelo entrenado, es posible visualizar un resumen de la red, que muestra las capas del modelo, las dimensiones de los datos y el número de parámetros.

```
modelo.summary()
```

[16] https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

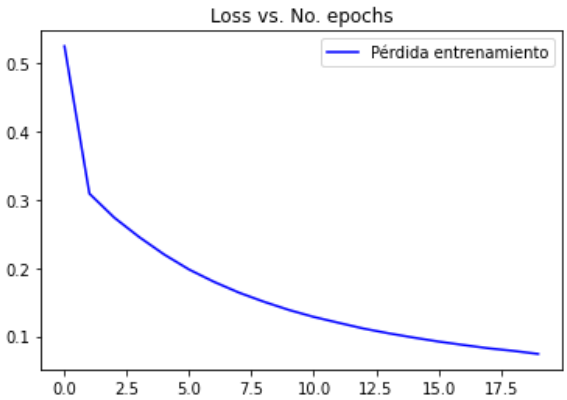

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (32, 784)                 0

 dense (Dense)               (32, 256)                 200960

 dense_1 (Dense)             (32, 10)                  2570

=================================================================
Total params: 203,530
Trainable params: 203,530
Non-trainable params: 0
```

Para cada una de las capas Dense, el número de parámetros equivale al número de unidades multiplicado por el número de atributos de entrada de la capa (w) más el número de unidades de la capa (b).

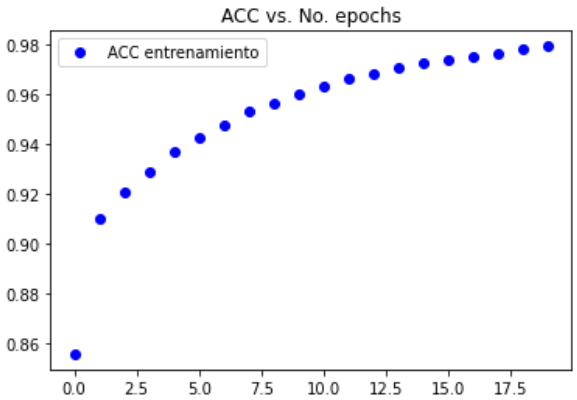
Visualizar evolución del entrenamiento

Utilizando el diccionario `history`, es posible graficar la evolución del valor de pérdida o de una métrica dada en función de la época de entrenamiento. La visualización de este comportamiento dependerá de la librería utilizada. A continuación, se muestran dos ejemplos básicos utilizando `matplotlib`. Sin embargo, se invita al lector a profundizar en estas herramientas.

```
# Gráfica de pérdida
loss = history.history['loss']
epochs = range(len(loss))
plt.plot(epochs, loss, 'b', label='Pérdida entrena
miento')
plt.title('Loss vs. No. epochs')
plt.legend()
plt.show()
```



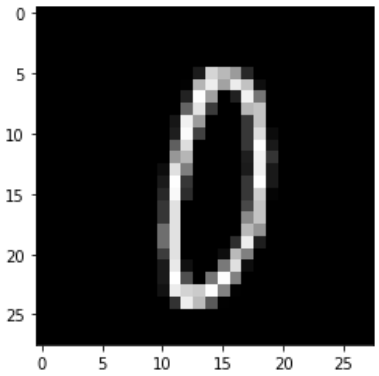

```
# Gráfica de métrica Accuracy
acc = history.history['accuracy']
epochs = range(len(acc))
plt.plot(epochs, acc, 'bo', label='ACC entrenamien
to')
plt.title('ACC vs. No. epochs')
plt.legend()
plt.show()
```



Prueba del modelo

Para evaluar el modelo se realizará una predicción sobre una nueva imagen utilizando el modelo entrenado. El resultado será la categoría que predice el modelo, es decir, el dígito. Como imagen de prueba, se seleccionará al azar una imagen del conjunto de datos de prueba, es decir de datos que no conoció el modelo cuando fue entrenado.

```
image = random.choice(x_test)
plt.imshow(image, cmap=plt.get_cmap('gray'))
plt.show()
```



A continuación, se realiza un pre-procesamiento similar al de los datos de entrenamiento (redimensionamiento y normalización):

```
image = (image.reshape((1, 28, 28, 1))).astype('float32') / 255.0
```

Finalmente, con el modelo entrenado se genera la predicción de salida para la imagen de entrada, utilizando el método `predict`. Aunque aquí se utiliza este método para una sola imagen, este método está diseñado para el procesamiento por lotes de un gran número de entradas.

```
modelo.predict(image)[0]
```

```
array([9.5489484e-01, 4.4818721e-06, 2.6543860e-03, 8.5353706e-04,
       3.6025063e-05, 3.5713132e-02, 1.8909280e-04, 3.4984839e-04,
       2.5580593e-03, 2.7466319e-03], dtype=float32)
```

La salida corresponde a un vector de predicciones tipo NumPy, que en este caso representan las probabilidades de pertenencia a cada una de las diez clases. Es decir, la suma de los diez valores es igual a 1, donde el mayor valor corresponde a la clase más probable. En el ejemplo mostrado, la clase más probable es la cero. Para solo mostrar el índice (clase) del valor máximo de un arreglo NumPy (a lo largo de un eje) se utiliza `argmax` de NumPy.

```
digit = np.argmax(modelo.predict(image)[0], axis=-1)
print("Prediction: ", digit)
```

```
Prediction:  0
```

MÉTRICAS DE EVALUACIÓN DE MODELOS DE CLASIFICACIÓN

El diseño de un algoritmo de clasificación por lo general está asociado a una métrica objetivo, la cual se puede evaluar durante el entrenamiento (con datos de entrenamiento y/o validación), y con datos de prueba. Durante el entrenamiento y validación, esta métrica permite evaluar no solamente el desempeño del algoritmo, sino también ajustar hiperparámetros del modelo y comparar diferentes versiones para facilitar la selección del modelo final. Durante

la fase de prueba, es posible evaluar la capacidad de generalización del modelo, con datos que no fueron utilizados ni en entrenamiento ni en validación.

Existe una gran diversidad de métricas de evaluación para modelos de clasificación, y la selección de la métrica o métricas objetivo dependerá del problema a resolver (teniendo en cuenta qué se espera del modelo) y su contexto de aplicación (por ejemplo, para efectos de comparación con el estado del arte). De cualquier forma, la gran mayoría de las métricas de evaluación de algoritmos de clasificación, como el *accuracy* o el F1-Score están basadas en la construcción de una matriz de confusión.

La matriz de confusión o matriz de error[17] corresponde a una tabla que permite visualizar el rendimiento de un algoritmo de clasificación, donde cada columna corresponde a la condición real de las clases y cada fila corresponde a la condición vaticinada en cada clase, o viceversa. La estructura de la matriz de confusión se muestra a continuación:

		Condición verdadera	
		Condición positiva	Condición negativa
Predicción	Predicción clase positiva		
	Predicción clase negativa		

El cruce de la condición real contra la condición predicha por el modelo facilita ver si el modelo etiqueta erróneamente una clase como otra, es decir si las confunde. Para ello, los resultados de clasificación se estructuran en cuatro grupos: Verdaderos positivos (TP), verdaderos negativos (TN), falsos negativos (FN) y falsos positivos (FP). Para realizar esta clasificación, es necesario establecer previamente cuál de las dos clases será la positiva (clase 1) y cuál la negativa (clase 0), donde la clase positiva por lo general se relaciona con el objetivo del modelo, es decir, corresponde a la clase que tiene mayor relevancia en la clasificación.

[17] https://en.wikipedia.org/wiki/Confusion_matrix

En una matriz de confusión binaria, TP y TN corresponden a las muestras clasificadas correctamente (para las clases 0 y 1, respectivamente), FN corresponde a muestras de la clase 1 que fueron clasificadas incorrectamente y FP corresponde a muestras de la clase 0 clasificadas como clase 1. La relación de TP, TN, FP y FN en una matriz de confusión se muestra a continuación:

		Condición verdadera	
		1	0
Predicción	1	**TP**	**FP**
	0	**FN**	**TN**

Para ilustrar el cálculo de algunas métricas basadas en una matriz de confusión, se utilizará el resultado de los dos modelos de clasificación que se muestran a continuación:

Tabla 2. Ejemplo de datos para matriz de confusión

	Modelo 1	**Modelo 2**
No. de ejemplos evaluados	200	200
TP	75	100
FN	25	50
FP	25	0
TN	75	50

Accuracy (ACC, exactitud)

Esta métrica evalúa el porcentaje de casos correctos (es decir solo TP & TN) en un clasificador. Es una de las métricas más utilizadas y se recomienda su uso cuando se tienen problemas balanceados (similar número de muestras en cada clase), o cuando la importancia de las clases es igual. Sin embargo, no se recomienda su utilización cuando el número de casos por clase difiere significativamente entre sí. El *accuracy* se define por la siguiente Ecuación:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{62}$$

El cálculo de *accuracy* para los dos modelos de ejemplo se muestra a continuación:

Modelo 1	Modelo 2
$ACC_1 = \frac{75 + 75}{200} = 0.75$	$ACC_2 = \frac{100 + 50}{200} = 0.75$

Desde el punto de vista de *accuracy*, ¿cuál de los dos modelos evaluados es el mejor? En este caso, aunque los dos modelos tienen diferentes valores en su matriz de confusión, su valor de *accuracy* es igual. Esto impide determinar cuál de ellos presenta un mejor rendimiento, y será necesario considerar otras métricas.

Precisión (P)

Esta métrica evalúa la precisión en las predicciones positivas, es decir sólo tiene obtiene la relación entre el número de veces donde acertó el modelo en la clase positiva respecto a la cantidad de casos vaticinados como positivos. Se calcula por medio de la siguiente Ecuación:

$$\mathrm{P} = \frac{TP}{TP + FP} \tag{63}$$

El cálculo de *la precisión* para los dos modelos de ejemplo se muestra a continuación:

Modelo 1	Modelo 2
$P_1 = \frac{75}{75 + 25} = 0.75$	$P_2 = \frac{100}{100 + 0} = 1.0$

En este caso, sí es posible discriminar una diferencia entre los dos modelos desde el punto de vista de precisión. En particular, el segundo modelo no presenta casos clasificados erróneamente como positivos, por lo cual la precisión es del 100%.

Recall (R)

El *recall* o *sensibilidad* determina el grado en el cual un modelo logra clasificar correctamente todas las muestras positivas del conjunto de datos evaluado. El recall está dado por:

$$\mathrm{R} = \frac{TP}{TP + FN} \tag{64}$$

Al calcular el *recall* para los dos modelos de ejemplo se obtiene lo siguiente:

Modelo 1	Modelo 2
$R_1 = \frac{75}{75+25} = 0.75$	$R_2 = \frac{100}{100+50} = 0.67$

En este caso, también es posible discriminar una diferencia entre los dos modelos. En particular, aunque el primer modelo clasifica correctamente un menor número de muestras como positivas, también presenta un menor número de casos clasificados erróneamente como positivos, por lo cual la precisión es mayor en relación con la del segundo modelo.

F1-Score

Otro tipo de métricas realizan una combinación de métricas para determinar su balance. En particular, el *F1-Score* calcula la media armónica entre P y R. Es decir, el cálculo del compromiso entre la precisión y el *recall* puede facilitar la selección de un modelo, dado que, en un caso puede ser mejor el modelo 1, mientras que desde el otro punto de vista puede ser mejor el otro. El *F1-Score* se define como:

$$\mathrm{F1} = 2\frac{P * R}{P + R} \tag{65}$$

El *F1_Score* para cada modelo del ejemplo es el siguiente:

Modelo 1	Modelo 2
$F1_1 = 2\frac{0.75}{0.75+0.75} = 0.75$	$F1_2 = 2\frac{1.0 * 0.67}{1.0+0.67} = 0.80$

Ya con el cálculo del F1-Score es posible determinar que el modelo 2 presenta un mejor rendimiento.

Otro tipo de métricas incluyen la curva ROC (*Receiver operation characteristic*) con aplicación en clasificación binaria, la cual relaciona la tasa de falsos positivos versus el *recall* (tasa de verdaderos positivos), o la métrica AUC (área bajo la curva) donde el mejor clasificador presenta un AUC de 1.0.

Como complemento a este tema, se sugiere al lector consultar información adicional relacionada con la generalización de la matriz de confusión en problemas multiclase o métricas diseñadas para problemas no balanceados.

CONJUNTOS DE DATOS

Como se ha comentado hasta aquí, los modelos de aprendizaje supervisado requieren un conjunto de datos cuyo patrón de comportamiento es modelado por el algoritmo durante la fase de entrenamiento. De aquí, que, la elección o construcción del conjunto de datos debe estar orientada de tal manera que su comportamiento refleje el de casos reales (datos similares a los que se esperan cuando el modelo esté implementado) (Ng, 2019).

Durante la fase de entrenamiento es común utilizar datos adicionales, conocidos como datos de validación, cuya función es orientar los cambios más importantes que se deben realizar al modelo para mejorar su rendimiento, es decir que facilitan el ajuste de los hiperparámetros. Por su parte, los datos de prueba permitirán evaluar el rendimiento del modelo con datos que nunca hayan sido utilizados ni en entrenamiento ni en validación, con el fin de evitar sesgos en el diseño del modelo. Lo recomendable es separar estos datos desde la fase inicial y sólo utilizarlos previo a la implementación del modelo o a la difusión de este.

También es recomendable que los datos de validación y prueba se extraigan de la misma distribución. Esto importante dado que durante el entrenamiento se puede correr el riesgo de que el modelo se sobreajuste a los datos de validación, o que lo datos de prueba sean mucho más complejos que los datos de validación o que los datos de prueba sean simplemente diferentes a los datos de validación. En cualquier caso, se tendría que, aunque el rendimiento del modelo presente un alto rendimiento en entrenamiento/validación, su rendimiento en con datos de prueba sea bajo. Es decir, que el modelo no logre generalizar el comportamiento de los datos.

De acuerdo con lo anterior, un conjunto de datos se distribuye en datos de entrenamiento, validación y prueba. El enfoque tradicional en aprendizaje

automático ha sido asignar alrededor de un 70% de los datos para entrenamiento, y 30% para validación y prueba. Sin embargo, estos porcentajes pueden variar en función del tamaño del dataset. Por ejemplo, se puede tener un 60% de datos para entrenamiento y 40% para validación/prueba cuando el número de muestras no es muy grande (ej. ~1000 muestras), o también el porcentaje para validación y prueba podría reducirse a menos del 10% cuando el número de ejemplos sea muy grande (ej. > 100.000 muestras). En cualquier caso, el número de muestras de validación y prueba se defina en función del grado de confianza deseado a la hora de evaluar el rendimiento del algoritmo (Ng, 2019).

En relación con la decisión de si construir un conjunto de datos desde cero o usar un dataset existente para un problema dado, cada alternativa tiene sus particularidades. En el caso de construir un dataset hay que tener en cuenta que la distribución de los datos sea similar a lo que se esperaría en la operación del sistema, evaluar que el número de muestras sea el adecuado para su rendimiento y que el tamaño del conjunto de datos no tenga incidencia directa sobre un posible sub-ajuste o sobreajuste del modelo (dependiendo de la complejidad de este). En el caso de utilizar un dataset existente, se tienen como ventajas que el número de muestras ya ha sido probado, una reducción en el tiempo asociada a la fase de preparación de datos y posibles resultados sobre los datos que ya han sido obtenidos por otros investigadores. Como reto se tendría que es necesario analizar el origen y la distribución de los datos con respecto al problema a resolver.

Fuentes de datos

En los últimos años la disponibilidad de datos para abordar problemas de aprendizaje automático ha crecido sustancialmente, con la posibilidad de contar con datos etiquetados y no etiquetados a nivel de imagen (ej. reconocimiento facial, reconocimiento de acciones, detección y reconocimiento de objetos, escenas naturales, datos geoespaciales), datos de texto (opiniones, noticias, mensajes de texto, redes sociales, chats, respuestas a preguntas), sonido (voz, música), video, sentimientos o sistemas de recomendación.

Este aumento en la disponibilidad de datos está asociado con el aumento de repositorios y buscadores de datasets. Por ejemplo, Google dispone tanto de su propio metabuscador (https://datasetsearch.research.google.com), como de un repositorio donde publican conjuntos de datos que han sido utilizados en sus investigaciones (https://research.google/tools/datasets/). Otra fuente reconocida de conjuntos de datos asociados a desafíos abiertos a la comunidad

es la plataforma Kaggle (https://www.kaggle.com/datasets); aquí es posible encontrar no solo los datos en sí, sino también soluciones y códigos que han sido utilizados para resolver un problema específico y que facilitan la comparación del rendimiento de un modelo respecto a resultados obtenidos por otros investigadores.

Por su parte, la plataforma *Papers with code* (https://paperswithcode.com/datasets) dispone también no solo de artículos de investigación y su implementación a nivel de código, sino también del respectivo conjunto de datos utilizado para obtener los resultados relacionados en los documentos científicos. De aquí que esta plataforma constituye un referente de gran importancia a la hora de evaluar los resultados de un sistema en relación con el estado del arte. Otros repositorios incluyen IEEE Dataport (https://ieee-dataport.org) o Mendeley (https://data.mendeley.com).

SELECCIÓN DE UN MODELO DE APRENDIZAJE AUTOMÁTICO

Configuración inicial del modelo

Como regla general a la hora de enfrentar un nuevo proyecto de aprendizaje automático por primera vez, es necesario definir una configuración inicial que involucra fundamentalmente tres aspectos: configuración del modelo, hiperparámetros del optimizador y número de pasos o épocas de entrenamiento (Godbole, Dahl , Gilmer, Shallue, & Nado, 2023).

Respecto a la configuración del modelo, se sugiere iniciar por una arquitectura simple y relativamente rápida que permita alcanzar un resultado razonable para el problema en cuestión. Otra alternativa, sugiere empezar con un modelo que funcione sobre ese tipo de problema, o un problema similar. Es decir, el modelo base se puede construir a partir de una revisión del estado del arte, con el fin de encontrar un trabajo previo que haya abordado un problema lo más similar posible (Godbole, Dahl , Gilmer, Shallue, & Nado, 2023).

De igual forma, la selección del optimizador sugiere iniciar con el optimizador más popular para el tipo de problema en cuestión y su tasa de aprendizaje predeterminada, de acuerdo con la revisión de trabajos previos. De igual forma, es importante considerar el ajuste de otros hiperparámetros del optimizador como el *momentum*.

En relación con el número de pasos de entrenamiento, su valor presenta un compromiso entre tiempo de ejecución (de cada época de entrenamiento) y rendimiento alcanzado. En este sentido, entrenar durante un mayor número de épocas puede permitir que se obtenga un modelo de mejores prestaciones, pero con un tiempo de ejecución más prolongado, lo que limita el número de experimentos (modelos o variaciones de hiperparámetros) que se pueden ejecutar en un tiempo dado. Aquí es importante también tener en cuenta el tamaño del lote, dado que este influye directamente en la velocidad de entrenamiento. Para el caso de imágenes, tradicionalmente se había sugerido un tamaño de lote igual o superior a 32 (Masters & Luschi, s.f.), sin embargo, estudios recientes sugieren que el tamaño ideal del lote usualmente es el mayor tamaño que soporte el hardware con el cual se entrenará el modelo (Godbole, Dahl , Gilmer, Shallue, & Nado, 2023). Es necesario tener en cuenta que para existen hiperparámetros que interactúan en gran medida con el tamaño del lote, como los específicos de cada optimizador o los de regularización.

Selección del modelo final

A la hora de seleccionar un modelo de aprendizaje automático, es importante tener en cuenta dos conceptos: error de entrenamiento y error de generalización. Como su nombre lo indica, el error de entrenamiento es el que se obtiene al usar los respectivos datos de entrenamiento, mientras que el error de generalización está relacionado con la evaluación del modelo sobre datos procedentes de la misma distribución de datos pero que no han sido conocidos por el modelo ni en entrenamiento ni en validación. Por lo general, la selección de un modelo se realiza luego de evaluar diferentes arquitecturas o modelos candidatos con base en la utilización de datos de validación que permiten escoger el modelo entre los candidatos.

Durante esta evaluación se realiza una evaluación de ajuste (*fitting*) del modelo, considerando si existe un sub-ajuste (*underfitting*) que se presenta cuando el modelo no es capaz de reducir el error de entrenamiento (aquí se dice también que el modelo presenta un alto *bias*), es decir, que el modelo se ajusta mal al conjunto de entrenamiento y sucede también que el error en el conjunto de validación es muy similar al error en entrenamiento. Por otro lado, puede existir sobreajuste (*overfitting*) que ocurre cuando el error de entrenamiento es mucho menor que el error de validación; aquí se dice que el modelo presenta una alta varianza, dado que el clasificador tiene un error de entrenamiento muy bajo, pero no logra generalizar en el conjunto de validación (Ng, 2019).

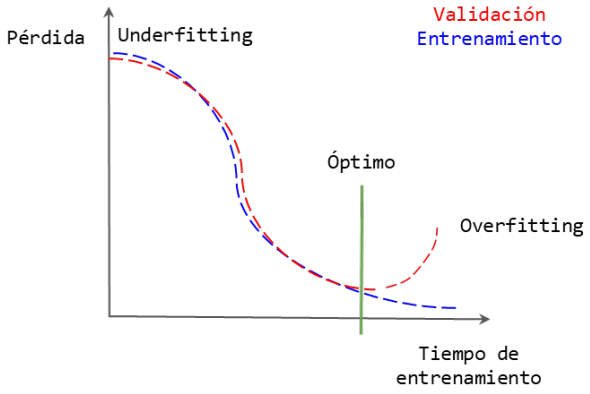


Figura 21. Ilustración de sub-ajuste (*underfitting*) y sobreajuste (*overfitting*) en modelos de aprendizaje automático.

Por lo general la complejidad de un modelo y el tamaño del conjunto de datos tienen una relación directa en el ajuste de este, donde a mayor complejidad del modelo se requiere un conjunto de datos más amplio. En este sentido, si el modelo es susceptible de mejora (es decir que existe posibilidad de reducir el *bias*) lo recomendable es aumentar el tamaño o complejidad del modelo (ej. no. de capas o neuronas); por su parte, si la varianza es alta, lo recomendable es agregar datos al conjunto de entrenamiento o aumentar su diversidad (Ng, 2019).

Sin embargo, existe un compromiso entre el *bias* y la varianza. Por ejemplo, incrementar la complejidad de un modelo puede mejorar el rendimiento (disminuir el *bias*), pero podría agregar *overfitting* (incrementar la varianza). A su vez, la aplicación de una técnica de regularización está orientada a reducir el *overfitting* (disminuir la varianza) pero puede afectar el rendimiento del modelo incrementando la varianza.

Las técnicas principales para mejorar el rendimiento de un modelo (reducir el *bias*), involucran aumentar el tamaño o complejidad de este sin descuidar lo relacionado con el *overfitting* (en caso de presentarse podría utilizare una técnica de regularización), modificar las características de entrada (a partir de un análisis de rendimiento), también en caso de que el modelo cuente con alguna técnica de regularización se puede reducir o eliminar (siempre teniendo en cuenta el compromiso entre *bias* y varianza), y también se puede considerar modificar la arquitectura del modelo (lo que también afecta *bias* y varianza).

Como técnicas de reducción de overfitting, se tiene en primer lugar el aumento de los datos de entrenamiento (en número y diversidad), agregar técnicas de regularización, agregar un criterio de detención temprana (*early stopping*) al

entrenamiento, reducir el número de atributos de entrada o reducir la complejidad del modelo. En cualquier caso, estas técnicas pueden ayudar a reducir la varianza, pero a su vez pueden aumentar el *bias*, por lo que se deben aplicar con precaución.

Regularización

Como se ha comentado hasta aquí, la reducción de overfitting se puede abordar en primera instancia aumentando el número de muestras de entrenamiento o reduciendo la complejidad del sistema ya sea limitando el número de atributos de los datos de entrada, o reduciendo el tamaño del modelo. También se ha comentado la posibilidad de incluir una técnica de regularización. En este aparatado se presentan tres técnicas comunes de regularización: *weight decay*, *dropout* y Batch normalization.

La técnica *Weight decay* consiste en agregar un término de penalización a la función de pérdida durante el entrenamiento con el fin de reducir la complejidad del modelo (Zhang, 2021). Esta técnica se conoce también como regularización L2.

Al aplicar *weight decay*, la Ecuación de pérdida presentada en el capítulo 4 se complementa con un término de penalización que depende de una constante de regularización (λ) que pondera la norma L2 del vector de pesos (por lo general no se regulariza el bias) (Zhang, 2021):

$$L(\boldsymbol{w}, b) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{66}$$

Una vez calculada la función de pérdida junto con el término de penalización, la actualización de los parámetros ($\mathbf{w}$) se realiza tal como se explicó en el capítulo 2.

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i \in BS} \boldsymbol{x^i}\left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{67}$$

$$b = b - \frac{\alpha}{BS}\sum_{i \in BS} \left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{68}$$

En cuanto al *Dropout*, corresponde a una técnica de regularización que fija en cero y de forma aleatoria *X* porcentaje de unidades de entrada en cada iteración durante el tiempo de entrenamiento[18] (es decir, una vez entrenado el modelo no se aplica el *dropout*).

Dado que se anulan ciertas unidades de entrada, las restantes se escalan a:

$$\frac{1}{1-X} \tag{69}$$

Con el fin de garantizar que el valor de todas las unidades no se altere. Un ejemplo de aplicación de dropout al 40% se muestra en la siguiente Figura (Saravia, 2021).

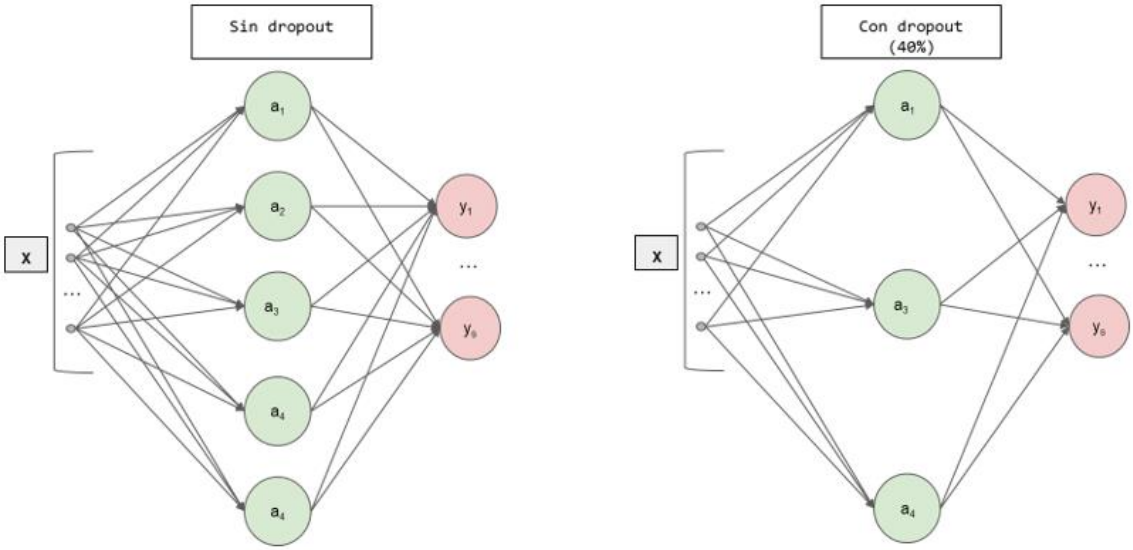


Figura 22. Ejemplo de redes neuronales con y sin regularización por *Dropout*.

Por su parte, *Batch normalization* (BN) es una técnica diseñada para acelerar la convergencia en redes profundas que pueden implicar un mayor desafío a nivel de complejidad y sobreajuste. BN se fundamenta en la importancia del pre-procesamiento de datos de entrada en las redes neuronales, especialmente los relacionados con normalización y estandarización. Dado que las funciones de activación en las capas intermedias de las redes neuronales pueden ocasionar que el rango dinámico de dichos datos sea muy amplio y en consecuencia puedan

[18] https://keras.io/api/layers/regularization_layers/dropout/

tomar valores muy altos, BN propone estandarizar estos datos, de tal forma que su media sea cero y su varianza sea unitaria.

La aplicación de BN en una capa involucra estandarizar los datos de entrada, substrayendo la media de estos y dividiendo por su desviación estándar. Además de aplicar un coeficiente de escalamiento (γ) y un valor offset (β) (Ecuación (71)). Estos dos valores corresponden a hiperparámetros que debe aprender la red durante su entrenamiento (Zhang, 2021).

$$BN(x)\gamma\frac{\boldsymbol{X}-\mu}{\sigma}+\beta \tag{70}$$

Es importante tener en cuenta que la aplicación de BN tiene relación directa con el tamaño del lote, dado que la media y la desviación estándar dependen de dicho valor. Además, su comportamiento presenta diferencias entre el entrenamiento (en función de las estadísticas de cada lote) y la predicción (en función de las estadísticas del dataset de prueba) (Zhang, 2021).

Ejemplo en python: *underfitting*, *overfitting* & regularización

Con el fin de contextualizar los conceptos de overfitting, underfitting y regularización se retomará el ejemplo de la sección anterior (perceptrón multicapa utilizando el dataset MNIST). En este caso, se evaluarán diferentes modelos que ilustran estos conceptos.

La definición de librerías y lectura de datos no tiene variaciones:

```
# Carga de librerías
import numpy as np
#import random
import tensorflow as tf
import matplotlib.pyplot as plt

# Carga de módulos
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.utils import to_categorical

# Lectura de datos
(x_train , y_train), (x_test , y_test) = tf.keras.dat
asets.mnist.load_data()
```



```
x_train = x_train.reshape( (x_train.shape[0], 28, 28,
 1))

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Modelo 1 (*underfitting*)

- 60 mil ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: 10
- Optimizador: SGD
- Capa oculta: 1 FC de 1 unidad

En primer lugar, se muestra un modelo muy simple (1 capa densa con una neurona), que presenta *underfitting*:

```
# Modelo 1 (Underfitting)
# Datos
x_train_Mod = x_train[1:60000,:,:,:]
y_train_Mod = y_train[1:60000,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 10
optimizerf = tf.keras.optimizers.SGD(learning_rate
=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(1, activation='sigmoid')
,
                    Dense(10, activation='softmax'
)])
```


Al entrenar este modelo se obtiene un *accuracy* de alrededor del 30% en entrenamiento y en validación. Es decir, el *bias* del modelo es alto (bajo rendimiento).

Modelo 2 (*overfitting*)

- **500** ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de **8 unidades**

Para entrenar el segundo modelo se utilizarán solo 500 ejemplos, pero se aumenta ligeramente la complejidad del modelo a una capa densa de 8 unidades y se utiliza el optimizador Adam.

```
# Modelo 2 (Mejora el rendimiento, pero presenta o
verfitting)
# Datos
x_train_Mod = x_train[1:500,:,:,:]
y_train_Mod = y_train[1:500,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(8, activation='sigmoid')
,
                    Dense(10, activation='softmax'
)])
```

Luego de entrenar este modelo, se alcanza un *accuracy* de 82% en entrenamiento, pero sólo 67% en validación. Es decir, el modelo no logra generalizar de forma óptima, por lo cual existe overfitting.

Modelo 3 (Reducción de overfitting)

- **60000** ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 8 unidades

Para el tercer modelo se utilizan de nuevo los 60000 ejemplos (es decir se aumenta el número de muestras respecto al modelo 2, con el fin de reducir el overfitting).

```
# Modelo 3 (Combatir overfitting reduciendo el tam
año del dataset)
# Datos
x_train_Mod = x_train[1:60000,:,:,:]
y_train_Mod = y_train[1:60000,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(8, activation='sigmoid')
,
                    Dense(10, activation='softmax'
)])
```

En este caso, se alcanzan valores similares de *accuracy* para entrenamiento y validación (alrededor del 86%), lo que equivale a una reducción de overfitting en el sistema.

Modelo 4 (Reducción de overfitting)

- 60000 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.1
- Número de épocas: **20**
- Optimizador: Adam
- Capa oculta: 2 FC de **256 unidades, ReLU**

En cuarto lugar, se trata de mejorar el rendimiento del modelo, aumentando su complejidad y con un mayor tiempo de entrenamiento:

```
# Modelo 4 (Aumentar complejidad del modelo)
# Datos
x_train_Mod = x_train[1:60000,:,:,:]
y_train_Mod = y_train[1:60000,:]

# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.1, 20
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(256, activation='sigmoid
'),
                    Dense(10, activation='softmax'
)])
```

Con este último modelo, el rendimiento mejora y se aproxima al 90% tanto entrenamiento como en validación.

Modelo 5 (*overfitting*)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10

- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU

Para evaluar el impacto de aplicar técnicas de regularización, como punto de partida se tomará un bajo número de muestras de entrenamiento (500) y una complejidad media del modelo:

```
# Modelo 5
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu'),
                    Dense(10, activation='softmax'
)])
```

Al entrenar el modelo 5, se obtiene un *accuracy* en entrenamiento del 99% y 84% en validación, por lo cual existe overfitting.

Modelo 6 (regularización L2)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU y regularización L2 (*weight decay* de 0.01)

```
# Modelo 6 (Weight Decay)
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
```

```
weight_decay1 = 0.02
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(weight
_decay1)),
                    Dense(10, activation='softmax'
)])
```

Los resultados del entrenamiento del modelo entregan un *accuracy* de entrenamiento de 97% y en validación de 83%. Es decir, aunque los resultados son sutiles, se presenta un acercamiento entre el resultado de entrenamiento y el de validación.

Modelo 7 (dropout de 0.2)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU y dropout de 0.2

```
# Modelo 7 (Dropout: 0.2)
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu'),
                    Dropout(0.2),
```



```
                    Dense(10, activation='softmax'
)])
```

Para el modelo 7 se alcanza una pequeña reducción de overfitting respecto al modelo 5. En este caso, se reduce el *accuracy* en entrenamiento al 98% en entrenamiento, manteniendo el 84% de *accuracy* en validación.

Modelo 8 (dropout de 0.5)

- 500 ejemplos
- Batch size: 32
- Tasa de aprendizaje: 0.001
- Número de épocas: 10
- Optimizador: Adam
- Capa oculta: 1 FC de 128 unidades con ReLU y dropout de 0.5

```
# Modelo 8 (Dropout: 0.5)
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.001, 10
optimizerf = tf.keras.optimizers.Adam(learning_rat
e=lr)

# Modelo
model = Sequential([
                    Flatten(),  # 28*28=784
                    Dense(128, activation='relu'),
                    Dropout(0.5),
                    Dense(10, activation='softmax'
)])
```

En este último ejemplo, el grado de dropout es mayor, lo cual impacta principalmente en el *accuracy* de entrenamiento (92%), acercando su valor al rendimiento del algoritmo en datos desconocidos (datos de validación), que para este caso están alrededor del 83%.

4. REDES NEURONALES CONVOLUCIONALES

En teoría, una red neuronal artificial con una sola capa oculta como las estudiadas en el capítulo anterior puede aproximar o modelar arbitrariamente bien una función continua de *n* variables reales, siempre y cuando tenga un número suficiente de unidades o neuronas a las cuales se les haya asignado una ponderación adecuada (Hornik, 1989). Es decir que en principio una red neuronal poco profunda, pero muy amplia puede modelar cualquier función por compleja que sea (Csáji, 2001).

Sin embargo, una red neuronal de dichas características tenderá a sobre ajustarse rápidamente a los datos de entrenamiento. Entre mayor sea la amplitud de la red (mayor número de neuronas en la capa oculta), mayor será la capacidad de memorización de la red, por lo cual se comportará muy bien con los datos de entrenamiento, pero no muy bien con datos que no haya visto, como los datos de validación o prueba.

Como soluciones a este problema de sobreajuste, se podría pensar en aplicar alguna de las reglas para reducción de varianza, como aumentar el número de ejemplos de entrenamiento (que no en todos los casos es fácil de realizar), o también aumentar la complejidad del modelo.

Para aumentar la complejidad del modelo, sin aumentar el número de neuronas de la capa oculta, la opción sería aumentar el número de capas ocultas del modelo. Esto permitiría a la red aprender a caracterizar los datos a diferentes niveles de abstracción, con estructuras más diversas. Precisamente el incremento en la complejidad (profundidad) ayuda a que el modelo en este caso generalice el comportamiento de los datos de entrada, en lugar de aprenderlos de memoria. A su vez, tener un modelo con una mayor profundidad puede reducir la amplitud del modelo, lo que equivale a un menor número de neuronas en cada capa oculta. La importancia de esto último radica en que el número de parámetros de la red puede reducirse, acelerando su entrenamiento. En conclusión, un balance entre profundidad y amplitud (priorizando profundidad contra amplitud) puede ser provechoso en cuanto a evitar un sobreajuste excesivo y controlar el número de parámetros de la red.

De cualquier forma, es importante considerar aquí que las redes tipo perceptrón multicapa procesan datos que estén estructurados como un vector. En caso de alimentar una red de este tipo con datos de una dimensión mayor, como imágenes, será necesario aplanar los datos para adaptarlos a la capa de entrada de la red (por ejemplo, mediante una capa *Flatten* en *Keras/tensorflow*). Respecto a la forma de procesar imágenes con este tipo de redes, se pueden precisar dos observaciones importantes. La primera tiene que ver con luego de aplanar los datos, la información espacial o de textura de la imagen se pierde, al igual que su información espectral o de color. La segunda tiene que ver con el costo computacional, asociado al número de neuronas que utilizan las capas totalmente interconectadas de estas arquitecturas. A partir de estas dos observaciones se ha propuesto como alternativa el uso de redes neuronales convolucionales, que se describen a continuación.

Operaciones de correlación cruzada y convolución en 2D

Como se comentó anteriormente, una red basada en capas totalmente interconectadas se caracteriza por una gran cantidad de parámetros entrenables, dado que el número de parámetros entre capas se determina por el producto entre los tamaños (o número de neuronas) de las dos capas. De aquí que, si el número de parámetros de la red es considerablemente alto, su aprendizaje puede resultar inviable o requerir una gran cantidad de datos.

Como alternativa, se tiene el uso de capas convolucionales que procesan la información de entrada como si de un filtro espacial se tratara y mantienen la estructura de la información de entrada (por ejemplo, sin aplanar los datos), con dos objetivos en mente. El primero consiste en que la información de entrada se procese por pequeñas áreas o vecindades (principio de localización), mientras que el segundo involucra que todas las zonas o áreas de la imagen sean tratadas de la misma forma (invariancia a traslaciones). Por ejemplo, si en una imagen se desea identificar un determinado objeto, la red debe ser de capaz de identificarlo independientemente de la posición o tamaño que tenga.

Partiendo de la estructura de una capa totalmente interconectada ($A=XW+b$, Ecuación (49)), estudiada en el capítulo anterior, se procederá a definir una capa convolucional. En este caso, para cumplir los dos criterios anteriores, la imagen de entrada X se procesaría en una vecindad alrededor de un píxel (principio de localización) y dicha vecindad se desplazaría en toda la imagen a lo largo de filas y columnas (invarianza a traslaciones). A su vez, la operación en cada región o

vecindad alrededor de un píxel se aplicaría con ciertos valores o pesos que corresponden a los valores del filtro (*K*), sumados a valores de ajuste o *bias* (*b*). De esta forma, el valor de salida para cada píxel o valor de entrada procesado en una posición específica (*i,j*) corresponde a:

$$[\boldsymbol{A}]_{\boldsymbol{i,j}} = \sum_{-m}^{m}\sum_{-n}^{n}[K]_{mnij}\,[\boldsymbol{X}]_{i+m,j+n} + [b]_{ij} \tag{71}$$

Donde la dimensión de la ventana a procesar es (*2m+1* × *2n+1*). Sin embargo, si se desea que todas las zonas de la imagen se procesen de la misma forma, tanto el *bias* como los valores de ponderación del filtro o *kernel* (*K*) deberían ser iguales para toda la imagen. De esta forma, la ecuación se simplifica de la siguiente forma:

$$[\boldsymbol{A}]_{\boldsymbol{i,j}} = \sum_{-m}^{m}\sum_{-n}^{n}[K]_{mn}\,[\boldsymbol{X}]_{i+m,j+n} + b \tag{72}$$

Lo anterior significa que mediante una operación de este tipo es posible que un modelo aprenda patrones que estén presentes en cualquier parte de los datos de entrada, lo que no sucede con operaciones tipo FC. Esta característica hace que las redes convolucionales funcionen mejor en el modelamiento de imágenes respecto a arquitecturas basadas en perceptrón multicapa. Para dar cumplimento al principio de localización, la zona procesar, es decir, las dimensiones de la ventana (*2m+1* × *2n+1*) se tienen que limitar a un valor pequeño en comparación con las dimensiones de los datos de entrada. Es típico utilizar en redes convolucionales recientes, dimensiones de filtro de (3×3) o (5×5). De esta forma, los valores que debe aprender la red se limitan a los valores del filtro, y su número es reducido en comparación con el número de parámetros que se tienen para una capa FC.

Pero ¿por qué se habla de capas convolucionales? Recordando la Ecuación de convolución discreta entre dos funciones que se estudia en un curso de señales, se tiene que:

$$y[i,j] = f[i,j] * g[i,j] = \sum_{-\infty}^{\infty}\sum_{-\infty}^{\infty} f[a,b]\,g[i-a,j-b] \tag{73}$$

Si comparamos la Ecuación (72) con la de convolución en 2D, vemos que tienen una estructura similar. Sin embargo, existen dos diferencias. La primera es la presencia del *bias* como parámetro de ajuste del aprendizaje, y la segunda es que no se invierte la segunda señal. Esto último implica, que la operación mostrada en la Ecuación (72), no corresponde estrictamente a una operación de convolución, sino a una operación de correlación cruzada. Más allá de esto, en la literatura se popularizó el nombre de redes convolucionales, que hoy en día se mantiene.

Para ilustrar la operación de correlación cruzada (es decir, la operación que realizan las capas convolucionales), se creará en *python* la siguiente matriz de entrada:

```
import numpy as np
import cv2
Entrada = np.array([
    [0.0, 5.0, 0.0, 0.0, 9.0, 4.0, 0.0],
    [1.0, 6.0, 0.0, 0.0, 8.0, 3.0, 0.0],
    [2.0, 7.0, 0.0, 0.0, 7.0, 2.0, 0.0],
    [3.0, 8.0, 0.0, 0.0, 6.0, 1.0, 0.0],
    [4.0, 9.0, 0.0, 0.0, 5.0, 0.0, 0.0]
])
Entrada
```

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

Y se operará con el siguiente filtro o *kernel*:

```
kernel = np.array([
    [1.0, -2.0, 3.0],
    [4.0, -5.0, 6.0],
    [-7.0, 8.0, -9.0],
    ])
kernel
```



```
array([[ 1., -2.,  3.],
       [ 4., -5.,  6.],
       [-7.,  8., -9.]])
```

Como ejemplo se muestra la operación de los datos del recuadro rojo, con los datos del *kernel*, multiplicando elemento por elemento y sumando sus resultados. Es decir, 0*1 + 5*(-2) + 0*3 + 1*4 + 6*(-5) + 0*6 + 2*(-7) + 7*(8) + 0*(-9) = 6. Para operar sobre los bordes se realiza una operación conocida como *padding*, que se explicará más adelante. El resultado de la operación completa se muestra a continuación.

```
Salida = cv2.filter2D(src=Entrada, ddepth=-
1, kernel=kernel)
Salida
```

```
array([[-16.,   5., -16.,   6.,   9., -14.,   4.],
       [-21.,   6., -20.,  12.,  10., -15.,  14.],
       [-22.,   5., -22.,  12.,   9., -14.,  16.],
       [-23.,   4., -24.,  12.,   8., -13.,  18.],
       [ -8.,   1., -12.,  -6.,   5., -10., -12.]])
```

En conclusión, la convolución realiza un producto punto entre el filtro y la imagen de entrada en una posición específica, sumando el resultado para obtener un único valor. Luego desplaza el filtro una posición y repite el mismo procedimiento. Para ilustrar un ejemplo de convolución para una imagen, se utilizarán tres tipos de filtros que permiten extraer los bordes de una imagen. Particularmente, los bordes horizontales (0º), bordes verticales (90º) y los bordes diagonales (45º). Los filtros se han definido manualmente de la siguiente forma:

```
filtro_horizontal = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1]])

filtro_vertical =np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]])
```



```
filtro_diagonal = np.array([
    [-1, -2, 0],
    [-2, 0, 2],
    [0, 2, 1]])
```

La imagen de prueba y las tres imágenes filtradas se muestran a continuación:

```
image = cv2.imread("umng.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

horizontal = cv2.filter2D(image, -
1, filtro_horizontal)
vertical = cv2.filter2D(image, -
1, filtro_vertical)
diagonal_edges2 = cv2.filter2D(image, -
1, filtro_diagonal)
```

Imagen original

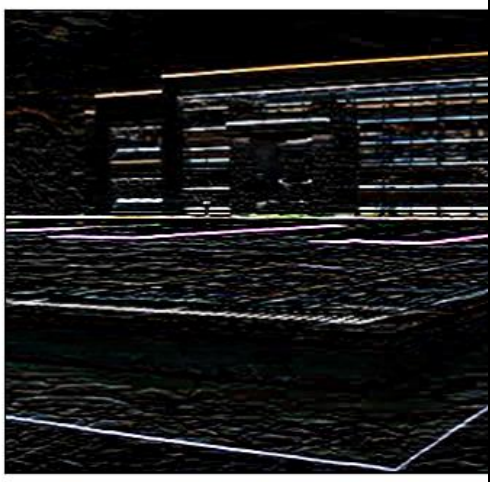


Imagen filtrada (bordes horizontales)

Imagen filtrada (bordes verticales)

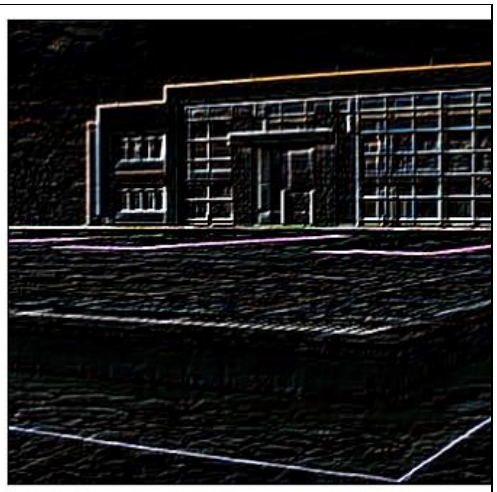


Imagen filtrada (bordes diagonales)

Figura 23. Ejemplo de aplicación de filtros espaciales en una imagen con máscaras predefinidas.

El ejemplo anterior involucra el uso de tres filtros pre-diseñados para extraer una característica específica. En la práctica, para capas convolucionales la idea es utilizar de manera simultánea más de un filtro, de tal forma que cada uno de ellos extraiga características específicas. Además, el valor que tendrán los coeficientes del filtro será aprendido durante el entrenamiento de la red, por lo cual no es necesario diseñarlos previamente, pero tampoco se tiene la posibilidad de definir sus características específicas. Para que la red aprenda los coeficientes del filtro, se realiza un procedimiento como el que se ha explicado en capítulos anteriores, es decir, se construye la capa convolucional, se inicializan los valores del filtro, se define una función de pérdida sobre la cual se calcula el gradiente y se actualizan los valores del filtro.

Para crear una capa convolucional en *frameworks* como *tensorflow*, los principales argumentos que hay especificar al instanciar la clase serán el número de filtros, el tamaño del *kernel* o filtro (vecindad), el algoritmo de inicialización y dos características adicionales: *strides* y *padding* que se explicarán a continuación.

```
#
https://keras.io/api/layers/convolution_layers/convolution2d/
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
```



```
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs)
```

En el caso de no especificar algunos argumentos, la herramienta tomará su valor predeterminado. Para agregar una capa convolucional a un modelo secuencial en keras/tensorflow, se realiza de forma similar a la adición de una capa Densa. Es importante tener en cuenta la estructura de los datos que ingresarán a la capa convolucional, ya que, si se utiliza una convolución en 2D, los datos ingresarán con estructura de filas y columnas, por lo cual no es necesario realizar un aplanamiento (*flatten*) previo.

```
model = Sequential([Conv2D(32, (3, 3),
                           activation= 'relu',
                           input_shape=(28, 28, 1)
                           ),
                    MaxPooling2D((2, 2)),
                    Flatten(),
                    Dense(10, activation='softmax')])
```

Al ser la primera capa del modelo, por cuestiones prácticas en este ejemplo se ha incluido la dimensión de los datos de entrada. Posterior a la capa convolucional se ha incluido una capa de *pooling*, que se explicará en este mismo capítulo. Luego de las capas convolucionales, es común que el modelo incluya capas Densas, por loscual, antes de dichas capas se incluye una capa *Flatten*.

Padding y *Stride*

Como se comentó anteriormente, para poder realizar la convolución entre el filtro y la imagen de entrada, es necesario realizar una operación adicional, de tal forma que los pixeles de los bordes puedan coincidir con el píxel central del *kernel*, ya que es común que los filtros en capas convolucionales sean de dimensión impar (1×1, 3×3, 5×5, etc.). Para esto, se agregan píxeles de relleno alrededor del límite de la imagen de entrada, lo que se conoce como *padding*. Para completar estos valores, es posible agregar, por ejemplo, ceros, o repetir el mismo valor de los pixeles del borde. Tomando el ejemplo anterior, la imagen con *padding* agregando ceros quedaría de la siguiente forma:

0	5	0	0	9	4	0
1	6	0	0	8	3	0
2	7	0	0	7	2	0
3	8	0	0	6	1	0
4	9	0	0	5	0	0

0	0	0	0	0	0	0	0	0
0	0	5	0	0	9	4	0	0
0	1	6	0	0	8	3	0	0
0	2	7	0	0	7	2	0	0
0	3	8	0	0	6	1	0	0
0	4	9	0	0	5	0	0	0
0	0	0	0	0	0	0	0	0

Figura 24. Ejemplo de *padding* de dos filas y dos columnas en una matriz

En este ejemplo, se han agregado dos filas y dos columnas, por lo cual al realizar la convolución entre la imagen y el filtro 3×3, el resultado será de las mismas dimensiones que la imagen de entrada. Más allá de este caso, es importante decir que el valor de *padding* tendrá incidencia en las dimensiones del dato de salida, de acuerdo con la siguiente Ecuación:

$$(I_x - k_x + p_x + 1) \times (I_y - k_y + p_y + 1) \tag{74}$$

Donde *x* es el número de filas, *y* es el número de columnas, I es la imagen de entrada, *k* es el *kernel* y *p* representa padding. Si los datos del ejemplo se procesan con el *padding* señalado (dos filas y dos columnas añadidas) y un filtro 3×3, las dimensiones del dato de salida serán:

$$(5 - 3 + 2 + 1) \times (7 - 3 + 2 + 1) = 5 \times 7 \tag{75}$$

Que en este caso corresponde a la misma dimensión de la imagen original, dado que se ha utilizado $p_x = k_x - 1$ y $p_y = k_y - 1$.

En cuanto al *stride*, este corresponde al número de filas y número de columnas que se desplaza el kernel en cada operación. Hasta aquí, los ejemplos mostrados involucraron un *stride* 1, sin embargo, en la práctica es posible utilizar un desplazamiento mayor. Para ilustrar este concepto, la siguiente figura muestra el desplazamiento del *kernel* con un valor de 1, y un valor de 2 (Saravia, 2021).

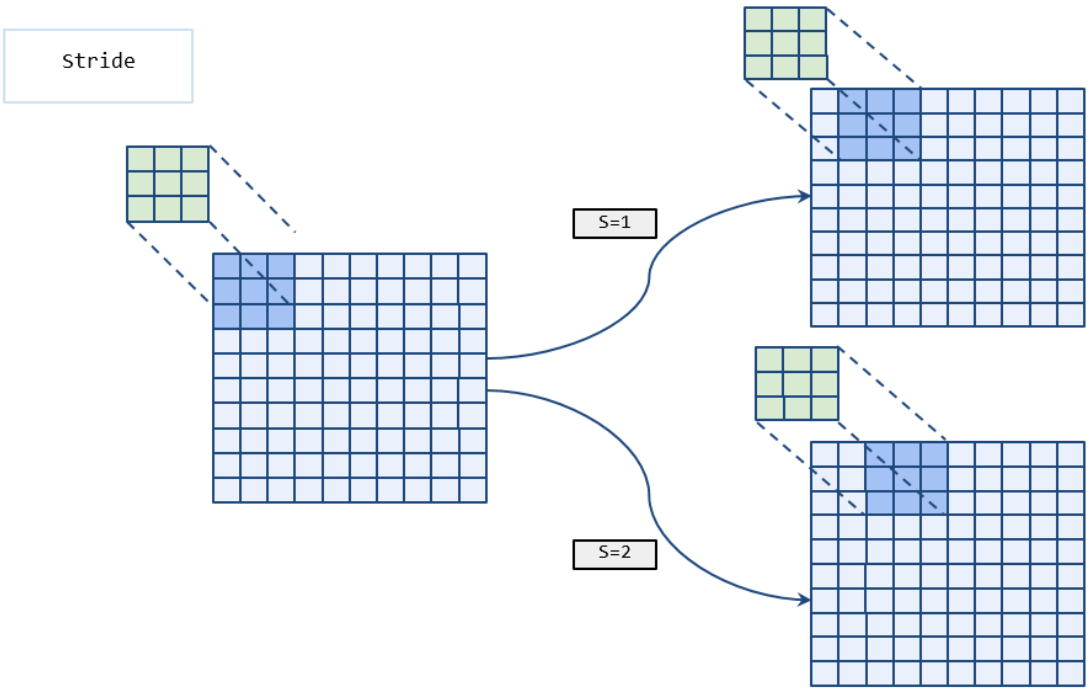


Figura 25. Ejemplo de dos valores de *stride* diferentes en una operación de convolución.

Al igual que para el *padding*, los valores que se definan en el *stride* inciden directamente en las dimensiones de los datos de salida, que están dadas por:

$$\lfloor (I_x - k_x + p_x + s_x)/s_x \rfloor \times \lfloor (I_y - k_y + p_y + s_y)/s_y \rfloor \quad (76)$$

Donde s corresponde a *stride*, y los corchetes medios corresponden a una operación de valor entero. Continuando con el ejemplo, se utilizará un *stride* 2 tanto en filas como en columnas, por lo cual las dimensiones de los datos de salida serán:

$$\lfloor (5 - 3 + 2 + 2)/2 \rfloor \times \lfloor (7 - 3 + 2 + 2)/2 \rfloor = 3 \times 4 \quad (77)$$

Para definir el *stride* en *keras/tensorflow* es necesario especificar un valor entero (si el *stride* en las dos dimensiones es igual) o una tupla o lista de 2 enteros (si el *stride* en filas y columnas es diferente). Estos valores, por lo tanto, detallan el desplazamiento del filtro de convolución a lo alto y ancho. En el caso del padding, se dispone de dos opciones: "*valid*" y "*same*". La primera opción significa que no aplica *padding*; por su parte, "*same*" entrega como resultado un *padding* uniforme ya sea a izquierda/derecha o arriba/abajo de tal forma que garantiza que la salida tenga la misma dimensión que la entrada[19].

Ejemplo de red neuronal convolucional en Python

A continuación, se muestra un modelo de red neuronal convolucional que se entrenará con el dataset MNIST, y que involucra tres capas convolucionales y una capa FC.

En primer lugar, se importan los módulos y librerías necesarias:

```
# Carga de librerías
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Carga de módulos
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
, Conv2D, MaxPooling2D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
```

Conjunto de datos MNIST, normalizando datos entre 0 y 1, redimensionando a un tensor de 4 dimensiones (muestras, alto, ancho, canales), y convirtiendo las etiquetas a una representación categórica (*one-hot encoding*).

```
# Dataset
```

[19] https://keras.io/api/layers/convolution_layers/convolution2d/


```
(x_train, y_train ), (x_test , y_test) = mnist.loa
d_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = x_train.reshape( (x_train.shape[0], 28,
28, 1))
x_test = x_test.reshape( (x_test.shape[0], 28, 28,
 1))

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Hiperparámetros: tamaño de lote de 32, tasa de aprendizaje 0.01 y entrenamiento por 10 épocas. Modelo secuencial con 3 capas convolucionales con filtros 3×3, un *stride* 1×1, y *padding* "*valid*". Para las tres capas convolucionales, el primer argumento que se especifica es el número de filtros que tendrá la capa, valor que usualmente se define como una potencia de 2 y que por lo general aumenta capa tras capa.

```
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.01, 10
optimizerf = tf.keras.optimizers.SGD(learning_rate
=lr)

# Modelo CNN
model = Sequential([
                    Conv2D(32, (3, 3),
                           activation= 'relu',
                           input_shape=(28, 28, 1)
,
                           strides=(1, 1),
                           padding="valid"),
                    MaxPooling2D(),
                    Conv2D(64, (3, 3),
                           activation= 'relu',
                           strides=(1, 1),
                           padding="valid"),
```

```
                    MaxPooling2D(),
                    Conv2D(128, (3, 3),
                           activation= 'relu',
                           strides=(1, 1),
                           padding="valid"),
                    MaxPooling2D(),

                    Flatten(),
                    Dense(10, activation='softmax'
)])
```

Una vez se ha creado el modelo, es necesario compilarlo (`compile`) previo a su entrenamiento (`fit`):

```
model.compile(
  optimizer=optimizerf,
  loss= 'categorical_crossentropy',
  metrics=['CategoricalAccuracy']
)

history = model.fit(
    x_train,
    y_train,
    epochs=1,
    batch_size=batch_s,
    validation_data=(x_test, y_test))
```

```
1875/1875 [==============================] - 17s 5ms/step - loss: 0.5368 - categorical_accuracy: 0.8567
```

Al observar el resumen del modelo, se percibe el bajo número de parámetros que involucran las capas convolucionales en comparación con las capas FC:

```
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_19 (Conv2D)          (None, 26, 26, 32)        320

 max_pooling2d_16 (MaxPoolin  (None, 13, 13, 32)       0
 g2D)

 conv2d_20 (Conv2D)          (None, 11, 11, 64)        18496

 max_pooling2d_17 (MaxPoolin  (None, 5, 5, 64)         0
 g2D)

 conv2d_21 (Conv2D)          (None, 3, 3, 128)         73856

 max_pooling2d_18 (MaxPoolin  (None, 1, 1, 128)        0
 g2D)

 flatten_8 (Flatten)         (None, 128)               0

 dense_9 (Dense)             (None, 10)                1290

=================================================================
Total params: 93,962
Trainable params: 93,962
Non-trainable params: 0
```

Una vez se tiene el modelo entrenado, es posible realizar predicción como se explicó en el capítulo anterior.

Inicialización de parámetros

Tal como se explicó en el capítulo 2, es necesario que los parámetros que se empiezan a actualizar en la primera iteración de un ciclo de entrenamiento tengan un valor definido, al cual se le pueda restar un término que depende del gradiente. Sin embargo, esta no es la única razón o característica involucrada en el uso de un método de inicialización. Particularmente, los métodos de inicialización en aprendizaje automático deben tratar de acotar o prevenir que el gradiente de un parámetro tome un valor excesivamente alto[20], pero tampoco

[20] https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

que tomen un valor excesivamente pequeño que impida que el gradiente identifique una tendencia de cambio en los datos[21].

Adicionalmente, es necesario que los algoritmos de inicialización eviten que los parámetros dentro de una capa sean iguales, dado que los mismos parámetros con unas mismas entradas podrían producir un mismo valor de activación hacia la siguiente capa. Es decir, que diferentes unidades o filtros de una misma capa podrían aprender lo mismo de los datos. Esto ocasionaría que el gradiente respecto a los parámetros sea igual para todas las unidades o filtros, y que al final el modelo se comporte como si tuviera una sola neurona o un solo filtro. Este comportamiento se evita con un método de inicialización que asigne valores diferentes o valores aleatorios, particularmente en los pesos o en los filtros de la capa (no es necesario para el *bias*).

Los métodos de inicialización disponibles en *keras/tensorlfow* se pueden consultar en https://www.tensorflow.org/api_docs/python/tf/keras/initializers.

El método de inicialización general asigna valores de pesos aleatorios con una distribución normal y valores de *bias* en cero. Métodos alternativos como el propuesto por Xavier (Glorot, 2010), permite utilizar una distribución normal de media cero y desviación estándar dependiente del número de unidades del tensor de entrada y de salida, o también propone el uso de una distribución uniforme, donde los valores límite también dependen de los mismos dos argumentos.

Otro método de inicialización popular es el propuesto por He (He K. Z., 2015), propone alternativas para definir la desviación estándar de la distribución uniforme (manteniendo la desviación estándar en cero), y también para el valor límite de la distribución uniforme. En este caso, sólo utiliza el número de unidades del tensor de entrada.

Convolución para múltiples canales de entrada

Al utilizar redes neuronales convolucionales, es común que la entrada presente más de un canal, por ejemplo, al utilizar imágenes RGB. Este tipo de imágenes se define por su dimensión espacial mediante alto y ancho, y por su profundidad o resolución espectral que equivale al número de canales (ej. 3 para imágenes RGB).

[21] https://en.wikipedia.org/wiki/Vanishing_gradient_problem

Es importante señalar que os ejemplos mostrados hasta aquí, trataron con imágenes de un solo canal, por lo cual el *kernel* para procesar dicho dato es de un canal también. Al tener imágenes de más de un canal, el *kernel* necesario para procesar los datos tendrá el mismo número de canales de los datos de entrada. De esta forma, cada canal del *kernel* procesa el respectivo canal de los datos de entrada, para finalmente sumar sus resultados y entregar un único valor, como lo muestra la siguiente figura (Saravia, 2021).

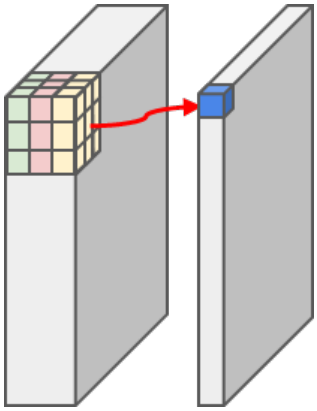


Figura 26. Ilustración de proceso de convolución para datos de entrada de varios canales y *kernel* del mismo número de canales.

En la Figura, los datos de entrada (en color gris) poseen tres canales, por lo cual el filtro (3×3) presenta también tres canales (ilustrados en verde, rojo y amarillo). Además, el filtro en cada posición entregará un único valor de salida (ilustrado en color azul), por lo cual típicamente el dato de salida contendrá solo un canal. Es decir que el resultado en cada canal de salida se calcula a partir del *kernel* de convolución correspondiente a ese canal de salida calculado sobre todos los canales del tensor de entrada (Zhang, 2021). En caso de definir un número de filtros mayor a 1, el número de canales del dato de salida será igual al número de filtros, tal como sucede en las capas convolucionales 1×1.

Capas de convolución 1×1

Algunos modelos de aprendizaje profundo utilizan capas convolucionales de dimensión 1. Debido a esto, este tipo de capa procesará los datos solo en la dimensión de los canales. Es decir, no tiene en cuenta la información espacial de una vecindad alrededor de un píxel, solo tiene en cuenta su profundidad. Una de sus principales aplicaciones es modificar el número de canales de los datos, dado que cada capa convolucional 1×1 determinará un número de canales de salida.

De aquí que el número de parámetros de la capa está dado solamente por el producto entre el número de canales de entrada multiplicado por el número de canales de salida. La ilustración de la operación de una capa convolucional 1×1 de un canal se ilustra en la siguiente Figura (Saravia, 2021).

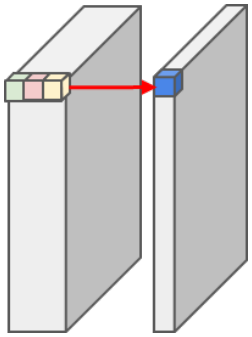


Figura 27. Ilustración de proceso de convolución 1×1.

Es importante tener en cuenta que este tipo de capa permite implementar una capa FC, dado que se aplica sobre un solo píxel.

Capas de *pooling*

Con el fin de operar capas convolucionales a diferentes resoluciones e impedir que los patrones que aprende la red sean sensibles a la ubicación y dimensiones de los datos, se suele utilizar capas de agrupación o *pooling*, que realizan una especie de submuestreo de los datos, por lo cual entregan datos con menor número de filas y de columnas.

Estos operadores son deterministas, es decir, que ante una misma entrada entregan una misma salida. Por lo general involucran una operación de selección del valor máximo de un área (*max pooling*) o del valor medio de los datos en el área (*average pooling*). Su operación se ilustra a continuación para una operación de *max pooling* (Saravia, 2021):

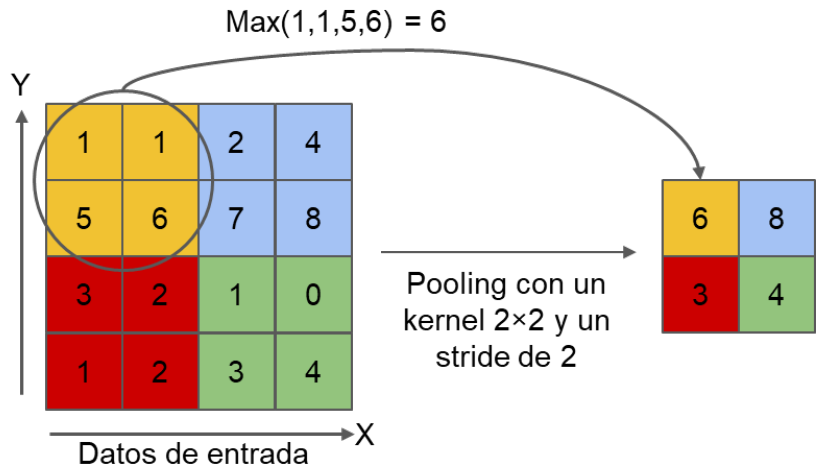


Figura 28. Ilustración de proceso de *max pooling* con *stride* de 2.

Para el ejemplo mostrado, es posible especificar también un valor de *padding* y *stride*. Es común utilizar filtros 2×2, con *padding* "*same*" y con un *stride* de 2. Estos valores garantizan que el ancho y alto de los datos de salida sean la mitad respecto a los datos de entrada. En esta operación, el número de canales de salida equivale al número de canales de los datos de entrada.

Finalmente, los tipos de capas de *pooling* disponibles en *Keras/tensorflow* se listan a continuación.

- MaxPooling1D layer
- MaxPooling2D layer
- MaxPooling3D layer
- AveragePooling1D layer
- AveragePooling2D layer
- AveragePooling3D layer
- GlobalMaxPooling1D layer
- GlobalMaxPooling2D layer
- GlobalMaxPooling3D layer
- GlobalAveragePooling1D layer

Guardar y cargar modelos

Un modelo creado en Keras/tensorflow abarca la arquitectura (capas y su interconexión), parámetros (valores de pesos y *bias*), optimizador y el conjunto

de pérdidas y métricas, los cuales se definen al compilar el modelo. De esta forma, una vez un modelo ha sido entrenado, es posible exportar tanto su arquitectura como sus parámetros o su configuración, para su posterior utilización o implementación. Una de las alternativas en keras/tensorflow es el método `save`[22] que que permite tanto exportar en formato json o en formato h5. El formato h5, genera un archivo de estado con claves de directorio para las capas y sus pesos.

Posteriormente, el comando `load`[23] permite cargar ya sea el modelo o sólo sus pesos (en formato hdf5). El siguiente ejemplo ilustra la utilización de estos dos comandos.

```
from keras.models import load_model

# Guardar el modelo y los pesos
model.save('alexnet.h5')
model.save_weights('alexnetw.hdf5')

# Cargar el modelo y los pesos
model = load_model('alexnet.h5')
model.load_weights('alexnetw.hdf5')
```

Puntos de control de modelos (Model ckeck point)

Durante el entrenamiento de un modelo, bajo un criterio dado (un número de época, un valor de métrica, etc.), es posible realizar un determinado tipo de acción conocida como *callback*. Este tipo de acción es útil para procesos como registrar logs o estadísticas del modelo o sus métricas, guardar versiones del modelo, detener el entrenamiento de forma anticipada (*early stopping*).

La opción que permite guardar periódicamente el modelo o sus pesos durante el entrenamiento se conoce como *model ckeckpoint*[24]. Como criterio para guardar el modelo o sus pesos en disco, puede definirse que sea al final de cada época, o transcurridos un número dado de lotes, o también, definir una métrica a monitorear, de tal forma que el modelo solo se exporta cuando el rendimiento

[22] https://www.tensorflow.org/guide/keras/serialization_and_saving
[23] https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

[24] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

mejora. Un ejemplo de utilización se ilustra en el siguiente código, en donde, luego de importar el método, se define una ruta donde se guardará el modelo y el nombre de archivo que puede configurarse para utilizar variables del entrenamiento. En ese caso, la métrica a monitorear es "*accuracy*". El *checkpoint* creado se instancia durante el entrenamiento como un *callback*.

```
# Callbacks - ModelCheckPoint
from tensorflow.keras.callbacks import
ModelCheckpoint
filepath="weights-improvement-{epoch:02d}-
{accuracy:.2f}.h5"
checkpoint = ModelCheckpoint(filepath,
                             monitor='accuracy',
                             verbose=1,
                             save_best_only=True,
                             mode='max')
callbacks_list = [checkpoint]

history = model.fit(train_ds,
                    validation_data=val_ds,
                    epochs=65,
                    verbose=1,
                    callbacks=callbacks_list)
```

Predicción y evaluación con modelos entrenados

Como se ha comentado hasta aquí, una vez creado el modelo, es posible configurarlo especificando una función de pérdida y métricas de evaluación a través del método *compile()*, o entrenarlo mediante el método *fit()*. Con el método entrenado, se pueden realizar predicciones sobre nuevos datos mediante el método *predict()*. En este caso, se genera predicciones de salida para las muestras de entrada, es decir el modelo realiza una inferencia sobre estos nuevos datos y entrega una salida, que por ejemplo corresponde a la clase a la cual pertenece la imagen de entrada. Los métodos aplicables a un modelo en keras/tensorflow pueden revisarse en https://www.tensorflow.org/api_docs/python/tf/keras/Model.

Al aplicar el método *predict(),* el cálculo se realiza por lotes, por lo cual no es necesario aplicarlo dentro de bucles o lazos de lectura de datos. Esto significa que el método está diseñado para funcionar con entradas a gran escala, puede funcionar desde una sola imagen, hasta un conjunto o lote de imágenes. Cuando se instancia el método de predicción, es necesario especificar como mínimo los datos de entrada (tensor, arreglo, etc). De manera complementaria, es posible especificar aspectos como el tamaño del lote, el número de pasos o *callbacks*. La forma general de aplicación del método de predicción en clasificación, depende del tipo de clasificador: binario o multiclase. En el primer caso, por lo general se tendrá una función de activación sigmoide en la última capa FC, por lo cual discriminar los valores de salida comparándolos con 0.5 servirá para determinar la clase (0, 1).

```
result = (model.predict(x) > 0.5).astype("int32")
```

Al tener un clasificador multiclase, donde la función de activación de la última capa FC es de tipo *softmax*, se suele utilizar la función `argmax` de `NumPy`, que devuelve los índices de los valores máximos a lo largo de un eje. Este valor máximo corresponde a la clase del dato de entrada.

```
result = np.argmax(model.predict(img), axis=-1
```

Más allá de la predicción, es posible evaluar el rendimiento de un modelo mediante el método `evaluate()`. Este método devuelve los valores de pérdida y métricas del modelo en modo de prueba. Al igual que en la predicción, este método opera por lotes. Los principales argumentos a especificar cuando se evalúa un modelo corresponden a los datos a evaluar y a sus etiquetas. Es posible especificar también el tamaño del lote o los pesos de las muestras de prueba para ponderar la función de pérdida, entre otros. Estos argumentos y su valor predeterminado se muestran a continuación.

```
evaluate(x=None, y=None, batch_size=None,
verbose=1, sample_weight=None, steps=None,
callbacks=None, max_queue_size=10, workers=1,
use_multiprocessing=False, return_dict=False)
```


Arquitectura LeNet

LeNet es una arquitectura de aprendizaje automático que utiliza capas convolucionales y que fue propuesta por Yann LeCun, Léon Bottou, Yoshua Bengio, y Patrick Haffner en 1998, consolidando un gran trabajo desarrollado en los años anteriores a su publicación. Es importante destacar que esta arquitectura se propuso mucho antes de la explosión del aprendizaje profundo iniciada a partir del 2012. En su momento, este modelo presentó resultados competitivos con respecto a métodos de aprendizaje de máquina consolidados como lo son las máquinas de soporte vectorial (SVM).

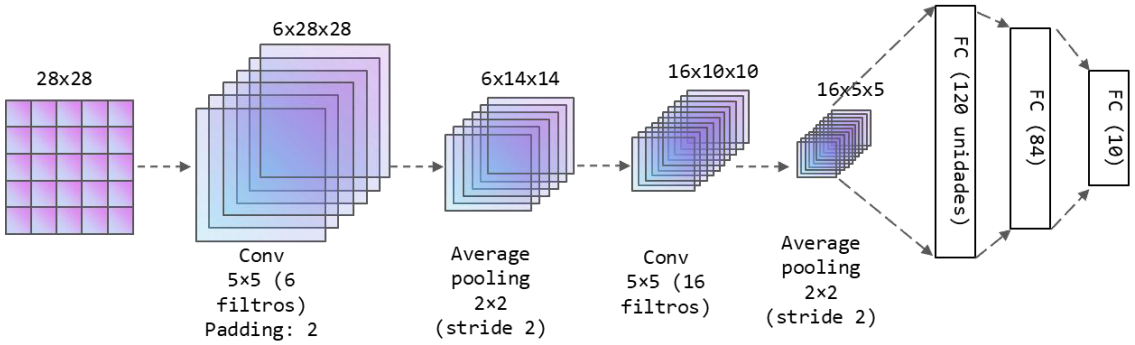


Figura 29. Arquitectura LeNet.

Como se aprecia en la Figura, la arquitectura LeNet implementa dos capas convolucionales con filtros 5×5 que utilizan función de activación sigmoide (LeCun Y. B., 1998). Estas capas convolucionales aumentan la profundidad de los datos, primero con seis filtros y luego con dieciséis filtros. A la par se reducen las dimensiones de los datos en alto y ancho, mediante capas de *pooling* que promedia los datos de una vecindad 2×2 y que reduce el alto y ancho a la mitad (reducción de dimensiones a la cuarta parte). Estas primeras capas permiten la extracción de características de los datos.

En la segunda parte del modelo se tienen capas tipo *Fully connected*, que reducen la dimensionalidad de los datos hasta llegar al número de clases del problema (10), dado que esta arquitectura se evaluó sobre el dataset MNIST. Por consiguiente, las capas FC realizan la clasificación.

La implementación de la arquitectura LeNet como modelo secuencial en *keras/tensorflow* se puede implementar de la siguiente forma:

```
model = Sequential([
```



```
                    Conv2D(filters=6, kernel_size=5,
                           activation='sigmoid', paddi
ng='same'),
                    AvgPool2D(pool_size=2, strides=2)
,
                    Conv2D(filters=16, kernel_size=5,
 activation='sigmoid'),
                    AvgPool2D(pool_size=2, strides=2)
,
                    Flatten(),
                    Dense(120, activation='sigmoid'),
                    Dense(84, activation='sigmoid'),
                    Dense(10)])
```

Se resalta que dada la fecha en la cual se presentó esta arquitectura, las funciones de activación fueron sigmoide y las capas de *pooling* son de promedio. Hoy en día es más común encontrar arquitecturas que utilicen función de activación ReLU y *max pooling*.

Aumento de datos

Como se mencionó en el capítulo anterior, una de las primeras alternativas para combatir el sobre ajuste de un modelo es aumentar la cantidad de muestras del conjunto de datos de entrenamiento. Sin embargo, no en todos los casos es fácil recolectar más datos, ni tampoco aumentar su diversidad.

Es aquí donde se habla de técnicas de *data augmentation*, es decir, literalmente aumento de datos, aunque no hay que confundirse con el concepto literal de esta frase. Para esto, las técnicas de *data augmentation* no están enfocadas a aumentar el número de muestras del conjunto de datos, sino de aumentar la diversidad de estas. Este aumento de diversidad puede facilitar que el modelo aprenda a reconocer de una mejor forma los patrones de los datos de entrenamiento, lo que conlleva a una mejor generalización del modelo, y en consecuencia, una reducción en el sobre ajuste.

Para aumentar la diversidad de los datos es posible aplicar un pre-procesamiento a los datos de entrenamiento que modifiquen ciertas características de las imágenes. Por ejemplo, es posible aplicar un zoom a la imagen, para que los

objetos de interés se vean más grandes, o también se pueden rotar o desplazar las imágenes haciendo que los datos presenten diversos contextos que siguen siendo válidos. En este caso, la imagen original se reemplaza por la imagen modificada, conservando el número de ejemplos de entrenamiento.

Operaciones típicas de pre-procesamiento disponibles en los frameworks de aprendizaje profundo incluyen las siguientes:

- Rotación
- Desplazamiento (a lo ancho o a lo largo)
- Modificación de brillo
- Inclinación
- Zoom
- Efecto espejo (a nivel vertical u horizontal)

En cada una de estas operaciones es posible especificar el grado de modificación de la imagen. Para la rotación, por ejemplo, el rango de grados en los cuales de forma aleatoria se modificará la imagen.

La implementación de técnicas de *data augmentation* suele realizarse desde el momento de la lectura de datos de entrenamiento, con opciones como el *Image Data Generator* de *Keras*. Esta opción permite implementar operaciones de pre-procesamiento de datos, o leer datos desde un dataframe, o desde una carpeta a los cuales se les aplicarán de forma aleatoria las técnicas de aumento de datos comentadas anteriormente. En el caso de leer los datos desde un directorio, lo deseable es que el directorio principal contenga subcarpetas con imágenes de cada una de las clases.

Como ejemplo se mostrará la implementación de *data augmentation* con el *Image Data Generator* de *keras*, aplicado al dataset MNIST. Para su aplicación, se leerá el dataset desde un directorio de entrenamiento que contiene diez subcarpetas correspondientes a cada clase, y que cada subcarpeta contiene imágenes de ejemplo de cada clase, como se muestra a continuación.

Figura 30. Estructura de conjuntos de datos MNIST (10 clases) con estructura por carpetas

En primer lugar, se define la ruta que contiene los datos:

```
import pathlib
data_dir = "/content/MNIST/trainingSet"
data_dir = pathlib.Path(data_dir)
```

A partir de lo anterior, es posible leer los datos por lotes especificando las operaciones de pre-procesamiento que se aplicarán a los datos, en el generador (`train_datagen`). Para el ejemplo, se aplicarán operaciones de inclinación (shear), zoom. Se ha desactivado la operación de reflejo horizontal dado que no es viable aplicarla para un conjunto de datos que contienen los dígitos del cero al nueve. Aquí se especifica cómo se realizará la distribución de los datos (en entrenamiento y validación). En este caso se utilizan el 80% de los datos para entrenamiento y el 20% para validación. En consecuencia, con el generador creado, es posible generar los contenedores de los datos de entrenamiento (`train_generator`) y de validación (`val_generator`). Para poder crearlos, se especifica el directorio done están los datos, las dimensiones que tendrán las imágenes leídas, el tamaño del lote, el tipo de etiqueta (categórica para el ejemplo) y se especifica cuál es el subconjunto que se está generando.


```
from tensorflow.keras.preprocessing.image import I
mageDataGenerator
img_height = 28
img_width = 28

train_datagen = ImageDataGenerator(
        #rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        #horizontal_flip=True,
        validation_split=0.2)

train_generator = train_datagen.flow_from_director
y(
        'MNIST/trainingSet',
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='categorical',
        subset='training')

val_generator = train_datagen.flow_from_directory(
        'MNIST/trainingSet',
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='categorical',
        subset='validation')
```

```
Found 33604 images belonging to 10 classes.
Found 8396 images belonging to 10 classes.
```

Al ejecutar el script de aumento de datos, se observa el número de imágenes utilizado en cada subconjunto (entrenamiento y validación) y el número de clases identificado que corresponde al número de subcarpetas del directorio. Estos contenedores pueden utilizarse para procesos posteriores como el entrenamiento, de la siguiente forma:

```
model.fit(
        train_generator,
        epochs=5,
        validation_data=val_generator,)
```


5. ARQUITECTURAS CNN DE REFERENCIA

En el capítulo anterior se presentó una de las arquitecturas de aprendizaje profundo que ayudó a sentar las bases de las redes neuronales modernas: LeNet. En términos generales, esta arquitectura desarrollada en los años 90 del siglo pasado presenta características similares a las arquitecturas secuenciales propuestas en la década del 2010, época en la cual se revolucionó el mundo del aprendizaje profundo. La pregunta aquí sería, por qué si ya se habían desarrollado las bases de las CNN, fue dos décadas después que empezó su aplicación e implementación a gran escala.

La respuesta a esta pregunta se fundamentó en dos aspectos: capacidad de cómputo y cantidad de datos. En dos décadas, la capacidad de cómputo creció exponencialmente de acuerdo con la ley de Moore, y permitió no solo aumentar la velocidad en el procesamiento de los datos, sino también disponer de unidades de procesamiento gráfico (GPU) para procesar datos en paralelo de manera eficaz. Este aspecto facilita la implementación de modelos con mayor número de capas y mayor profundidad, mejorando a su vez el rendimiento de los algoritmos.

Para entrenar este tipo de modelos un aspecto clave es contar con una gran cantidad de datos etiquetados que permitan en el entrenamiento ajustar los parámetros de la arquitectura. Mientras que la arquitectura LeNet fue entrenada y validada con un dataset de 60000 ejemplos con 10 clases, arquitecturas recientes han contado con datasets más complejos y cercanos a datos reales como lo son Pascal o ImageNet. La construcción y uso de estos datasets se potencializó a través de desafíos que invitaban a la comunidad a usar estos datos y resolver problemas relacionados con clasificación y reconocimiento de objetos.

En este sentido, ImageNet es el dataset de referencia más utilizado para clasificación de imágenes y reconocimiento de objetos a gran escala. Este dataset cuenta con más de 14 millones de imágenes etiquetadas manualmente con 1.000 categorías de objetos. Su amplia difusión radica en que este fue el dataset utilizado en el Reto ImageNet de reconocimiento visual a gran escala (ImageNet Large Scale Visual Recognition Challenge, ILSVRC). El ILSVRC fue una competencia anual que evaluó el progreso de los algoritmos de visión por

computador en cuanto a clasificación de imágenes y detección de objetos. La primera versión de este reto se realizó por primera vez en 2010[25].

Este reto se dividió en dos tareas: clasificación de imágenes y detección de objetos. En la tarea de clasificación de imágenes, los algoritmos se evalúan en función de su capacidad para clasificar correctamente las imágenes en una de las 1.000 categorías de objetos. En la tarea de detección de objetos, los algoritmos se evalúan por su capacidad para identificar y localizar objetos en las imágenes. El ILSVRC ha sido decisivo para impulsar el progreso de la visión por ordenador. El reto ha contribuido a impulsar el desarrollo de nuevos algoritmos de aprendizaje automático y ha dado lugar a mejoras significativas en la precisión de la clasificación de imágenes y la detección de objetos.

Este tipo de retos facilitó la comparación de nuevos modelos propuestos desde la investigación, de tal forma que en el año 2012 una arquitectura de aprendizaje profundo conocida como AlexNet superó a los modelos tradicionales de aprendizaje de máquina en el ILSVRC. En este año, AlexNet ganó el desafío de clasificación de imágenes con una tasa de error del 26,2%, superando ampliamente el resultado del año anterior que era del 37,5%. En 2014, de nuevo se presenta una mejora importante con la arquitectura VGG que obtuvo una tasa de error del 16,4%, demostrando la competitividad de los modelos de aprendizaje profundo. Ya en 2016, el ganador fue ResNet con una tasa de error del 3,5 %, lo que supuso un rendimiento similar al de un experto humano. En términos generales, las principales arquitecturas que significaron un aporte fundamental en esta área fueron las siguientes[26]:

1. AlexNet, que significó el gran avance en la implementación de arquitecturas de aprendizaje profundo (Krizhevsky, 2012)
2. VGG, a partir de la cual las redes neuronales se vuelven muy profundas mediante la implementación por bloques (Simonyan, 2014)
3. GoogLeNet (Szegedy, 2015)
4. ResNet, Redes neuronales de gran profundidad (He K. Z., 2016)

Estas arquitecturas que han sido pilares en el desarrollo del aprendizaje profundo se explicarán a continuación.

[25] https://www.image-net.org/challenges/LSVRC/
[26] https://github.com/floodsung/Deep-Learning-Papers-Reading-Roadmap

AlexNet

AlexNet es una arquitectura de CNN desarrollada por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton en 2012, que utiliza *kernels* convolucionales más grandes de lo usual hoy en día, lo que le permite aprender características complejas de las imágenes. Para su entrenamiento una de las novedades en su momento fue la utilización de GPUs, lo que facilitó un entrenamiento mucho más rápido, en comparación con modelos anteriores. Lo anterior, dado que, operaciones como la convolución o la multiplicación de tensores pueden paralelizarse en hardware. En particular, para su entrenamiento se utilizaron convoluciones agrupadas para ajustar el modelo en dos GPU.

Esta red tiene ocho capas, cinco de ellas convolucionales y tres FC (Ver Figura 31). Las capas convolucionales y las dos capas FC previas a la clasificación utilizan funciones de activación ReLU, siendo una de las primeras arquitecturas que utilizaron esta función de activación. La última capa FC utiliza función de activación *softmax*.

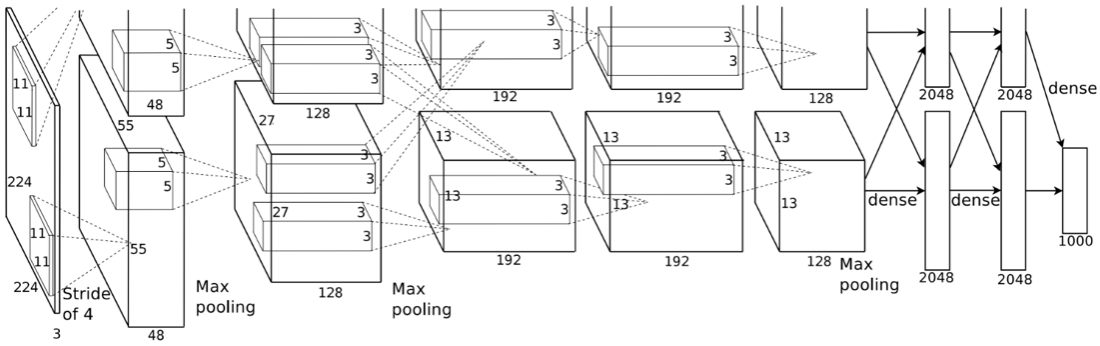


Figura 31. Ilustración de la arquitectura AlexNet y la operación entre dos GPUs para el entrenamiento (Figura tomada del artículo original (Krizhevsky, 2012))

Además de superar a los métodos de visión por computador de la época, esta arquitectura se caracteriza por tener alrededor de 60 millones de parámetros, que permitieron que los filtros de extracción de características obtenidos mediante el aprendizaje presentaran un rendimiento superior a los métodos diseñados manualmente. Este aspecto se puede ilustrar visualizando los 96 *kernels* convolucionales (11× 11×3) aprendidos por la primera capa convolucional de la red para imágenes de entrada de 224×224×3. Los 48 filtros superiores permiten extraer información espacial de la imagen y corresponden a

la primera GPU, mientras que los 48 filtros inferiores permiten extraer información espectral y de texturas y corresponden a la segunda GPU.

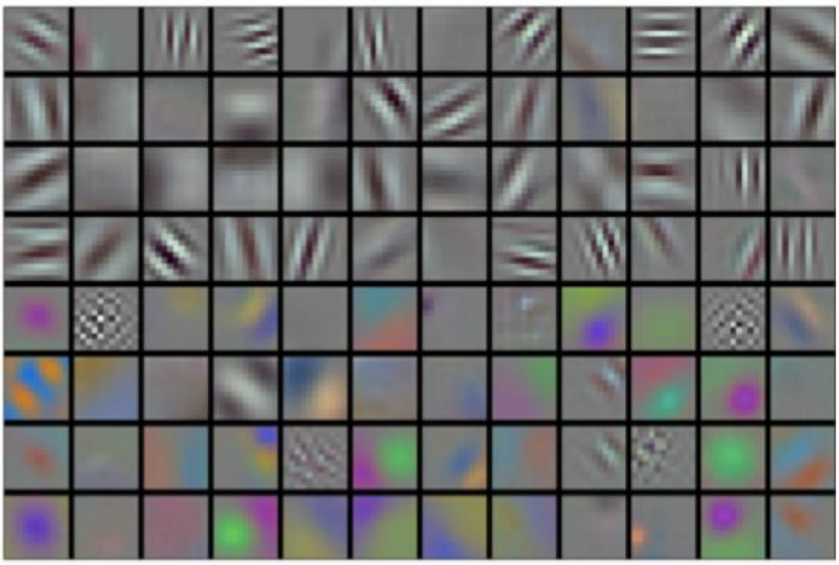


Figura 32. Filtros de imagen aprendidos por la primera capa de AlexNet con la GPU 1 (48 filtros superiores) y con la GPU 2 (48 filtros inferiores)
(Figura tomada del artículo original (Krizhevsky, 2012))

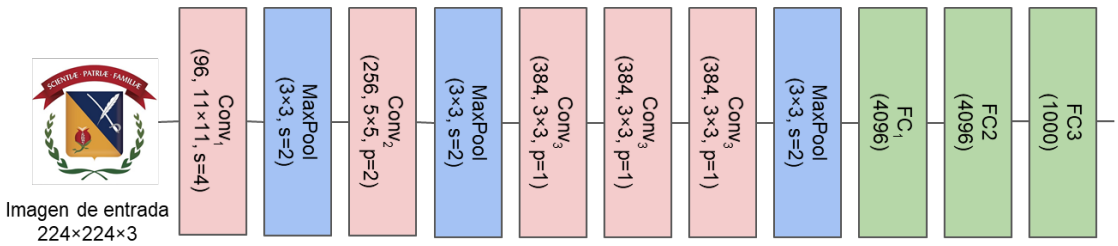


Figura 33. Esquema general de arquitectura AlexNet.

Un ejemplo de implementación de la arquitectura AlexNet se muestra a continuación. Aspectos característicos de esta red incluyen en la primera capa un filtro de gran dimensión (11×11) articulado con una reducción de dimensiones significativa (*padding* de 4). Igualmente, el número de canales aumenta en comparación con LeNet. Las capas convolucionales posteriores utilizan un tamaño de filtro menor (5×5 y 3×3). Para las dos capas densas previas a la clasificación, esta arquitectura puede involucrar el uso de *dropout* con fines de reducción de *overfitting*.

```
model = Sequential([
        Conv2D(filters=96, kernel_size=11,strides=4,
activation='relu'),
        MaxPool2D(pool_size=3, strides=2),
```

```
        Conv2D(filters=256, kernel_size=5,padding='sa
me', activation='relu'),
        MaxPool2D(pool_size=3, strides=2),
        Conv2D(filters=384, kernel_size=3,padding='sa
me', activation='relu'),
        Conv2D(filters=384, kernel_size=3,padding='sa
me', activation='relu'),
        Conv2D(filters=256, kernel_size=3,padding='sa
me', activation='relu'),
        MaxPool2D(pool_size=3, strides=2),
        Flatten(),
        Dense(4096, activation='relu'),
        Dense(4096, activation='relu'),
        Dense(10)])
```

VGG

VGG es una arquitectura propuesta por el grupo de investigación en geometría visual (VGG) de la universidad de Oxford en 2014. Esta arquitectura conserva la estructura de las dos arquitecturas presentadas anteriormente, en cuanto a que está estructurada en dos partes: extracción de características (capas conv. + capas *pooling*) y clasificación (capas FC). Una de las principales novedades de esta arquitectura radica en que se puede construir a partir de bloques compuestos por una secuencia de capas convolucionales seguidas de una capa de *pooling*. Para cada bloque, es posible especificar el número de capas convolucionales y el número de filtros o canales de salida del bloque. La estructura general del bloque VGG se muestra en la Figura 34 (Simonyan, 2014).

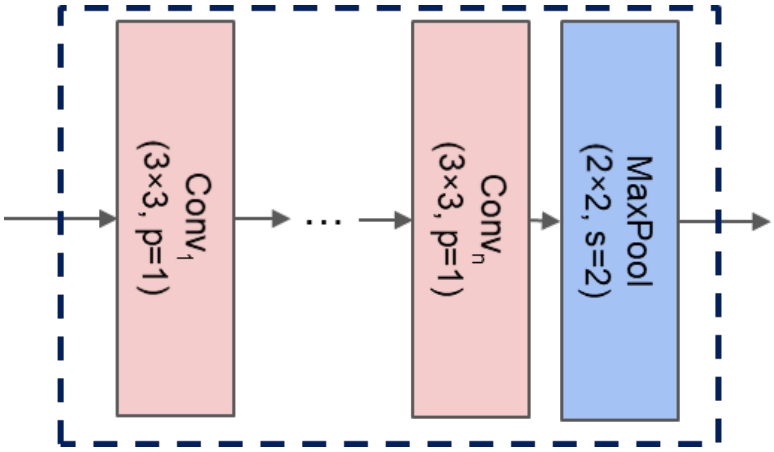


Figura 34. Esquema de bloque en arquitectura VGG con n capas convolucionales y número de canales variable.

La construcción de la arquitectura a partir de bloques reutilizables permite representar de manera compacta una red profunda. Dos de las arquitecturas VGG más difundidas corresponden a VGG16 y a VGG19, donde el número representa la profundidad del modelo. La representación de una arquitectura VGG se ilustra en la Figura 35 (Simonyan, 2014).

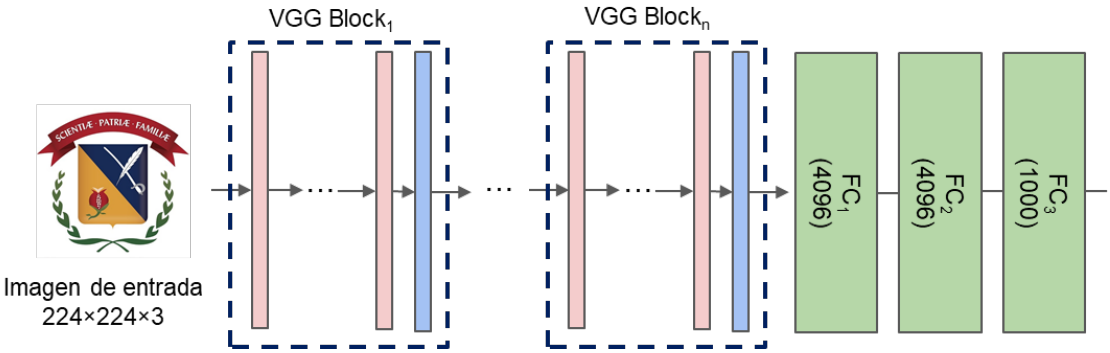


Figura 35. Arquitectura VGG conformada a partir de bloques.

GoogLeNet

Las arquitecturas presentadas hasta aquí están diseñadas de tal forma que extraen las características espaciales y espectrales de la imagen a través de una serie de capas de convolucionales y de pooling para reducir dimensiones. Estas características extraídas se procesan mediante un perceptrón multicapa, con una secuencia de capas de FC, que finalizan con la clasificación de los datos de entrada. Adicionalmente, este tipo de estructura fue compactada mediante bloques a partir de la propuesta de la arquitectura VGG.

Un enfoque alternativo propuso representar de una manera más amplia la estructura espacial de la imagen mediante el uso de capas convolucionales combinadas con capas FC (Lin, 2013) desde los primeros bloques de la arquitectura. De esta forma, los bloques Network in Network (NiN) están conformados por una capa convolucional seguida de múltiples capas convolucionales 1×1, que se comportan como capas totalmente interconectadas.

Una vez se implementa una arquitectura a partir de bloques NiN, el número de canales de salida se reduce al número de clases del problema por medio de las capas convolucionales 1×1. Esto implica que no sea necesario utilizar las capas FC de salida de los modelos anteriores, solo basta con utilizar una capa de *pooling* promedio global (*Global average pooling*). Su ventaja radica en un menor número

de parámetros y el precio a pagar puede implicar un mayor tiempo de entrenamiento.

A partir de estos fundamentos principales: la filosofía por bloques utilizada tanto en VGG como en NiN, y el uso de representaciones de diferentes dimensiones unida al *pooling* global de salida en NiN, Google propuso la arquitectura GoogLeNet. La idea en esta arquitectura es utilizar una combinación de filtros de distintos tamaños, que se articulan en un solo bloque conocido como *Inception*. Este bloque permite extraer información en paralelo mediante capas convolucionales de diferentes dimensiones que incluso involucran una capa de *max pooling*. La arquitectura completa se conforma a partir de la conexión de estos bloques con otras capas en serie. La arquitectura del bloque Inception se muestra en la Figura 36 (Szegedy, 2015).

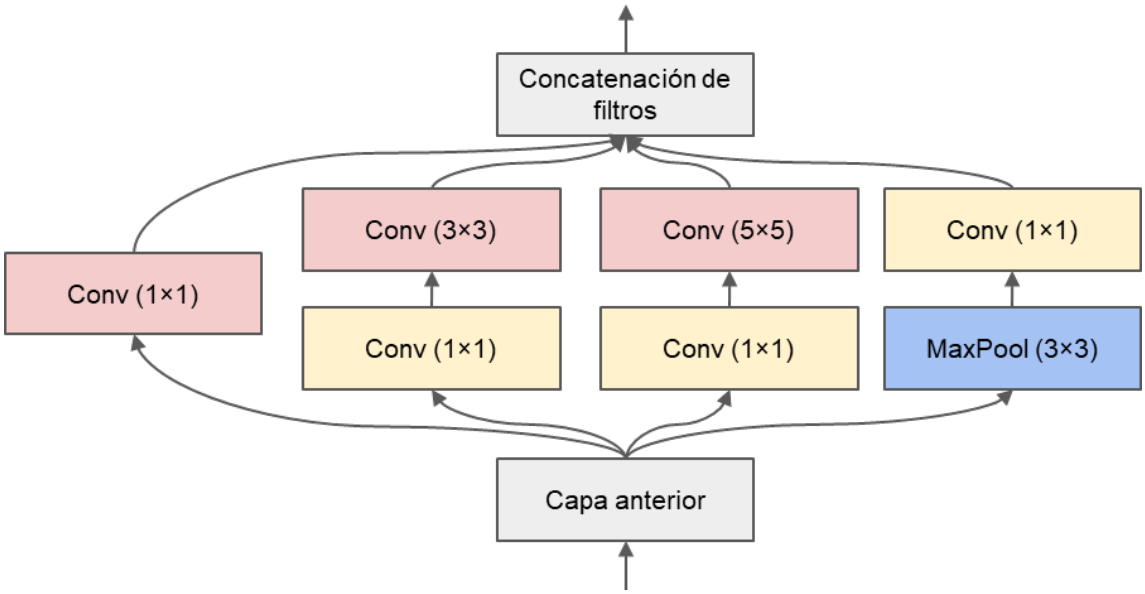


Figura 36. Esquema de bloque Inception para arquitectura GoogleNet.

El esquema general de la arquitectura GoogLeNet conformado a partir de bloques *Inception*, algunas capas convolucionales y *pooling* de entrada, más la capas de *global average pooling* y capa FC de clasificación se muestra en la Figura 37. En el intermedio de algunos bloques Inception se utilizan capas de *max pooling* para reducir la resolución de los datos a nivel de filas y columnas. Al interior de los bloques Inception se utilizan capas convolucionales 1×1 que permiten reducir la dimensión, pero en este caso a nivel de canales (Szegedy, 2015).

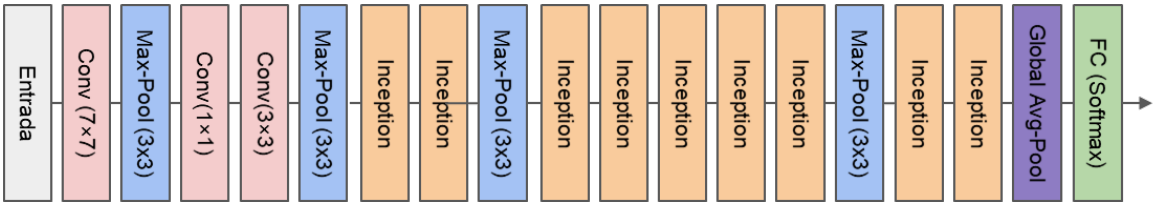
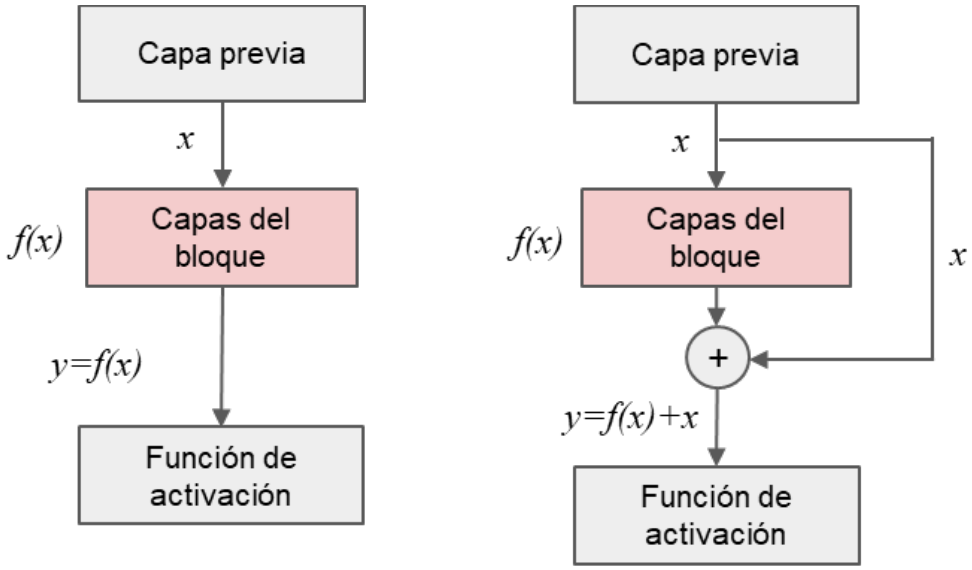


Figura 37. Diagrama general arquitectura GoogleNet.

ResNet

Las arquitecturas estudiadas hasta aquí conectan de manera secuencial los bloques definidos en cada una de ellas, tal como lo ilustra la Figura 38(a).



(a) Sin conexión residual (b) Con conexión residual

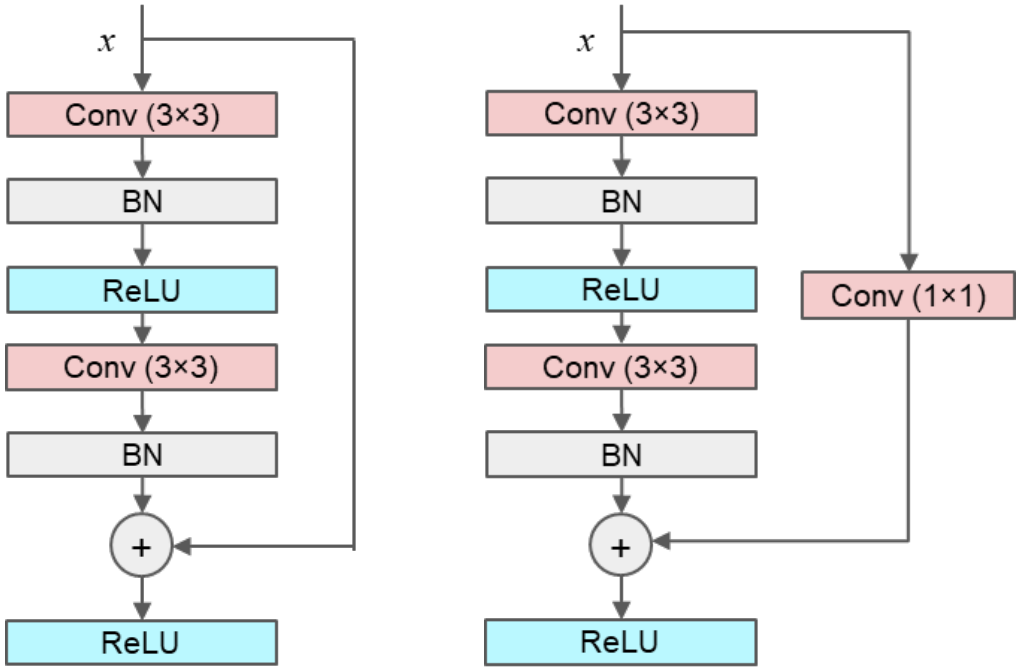
Figura 38. Esquema de conexión de bloques en arquitecturas CNN.

La arquitectura ResNet va un paso más allá y propone combinar la función no lineal entregada por un bloque ($f(x)$) con el componente lineal de entrada al bloque (x) (Ecuación (78)) (Zhang, 2021). Esta combinación se conecta posteriormente a la función de activación (ver Figura 38(b)).

$$g(x) = x + f(x) \tag{78}$$

En ResNet (He K. Z., 2016), el bloque de la arquitectura está conformado por dos capas convolucionales 3×3 con igual número de canales, seguidas cada una por

un proceso de *batch normalization* y una función de activación ReLU. Previo a la conexión de esta última función de activación se realiza la conexión residual. En este punto, la arquitectura contempla dos posibilidades. La primera consiste en conservar el número de canales de la entrada, por lo cual el número de canales de las dos capas convolucionales será igual al número de canales de entrada. De esta forma, la señal de salida del bloque (*f(x)*) puede adicionarse a esta misma entrada (*x*). La segunda alternativa permite variar el número de canales de salida, por lo cual es necesario adicionar en la conexión residual una capa convolucional 1×1 que permita ajustar el número de canales de la entrada *x* al número de canales configurados en las dos capas convolucionales del bloque (He K. Z., 2016).



(a) Conexión conservando el número de canales de entrada

(b) Conexión variando el número de canales de entrada

Figura 39. Esquemas de conexión residual en arquitectura ResNet.

DenseNet

En la arquitectura ResNet se estableció el uso de conexiones residuales que implican una adición entre el dato de entrada y una combinación no lineal de este. Tomando este tipo de conexión como referencia, la arquitectura DenseNet

propone el uso de conexiones similares, con la diferencia que los datos no se suman, sino que se combinan mediante concatenación (Huang, 2017). De acuerdo con sus autores, este tipo de conexión está orientado a mejorar el flujo de información y los gradientes a través de la red, lo que facilita su entrenamiento, además de tener un efecto de regularización sobre la red.

La conexión *feedforward* de la arquitectura DenseNet se realiza hacia varios dentro de un bloque, a diferencia de las conexiones residuales en ResNet. La concatenación de los datos y la conexión en diversos puntos hace que la cantidad de datos se incremente, por lo cual es necesario utilizar una red tipo perceptrón multicapa en la salida con el fin de reducir el número de atributos. Esto significa que la última capa del bloque está densamente conectada a todas las capas anteriores (Huang, 2017). El bloque fundamental de la arquitectura general de conexiones densas de DenseNet se ilustra en la Figura 40 y en la Ecuación (79). La idea de este tipo de bloque es permitir una conexión directa dese cualquier capa a todas las capas subsecuentes del bloque. De esta forma, la salida es una función de diferentes procesos previos.

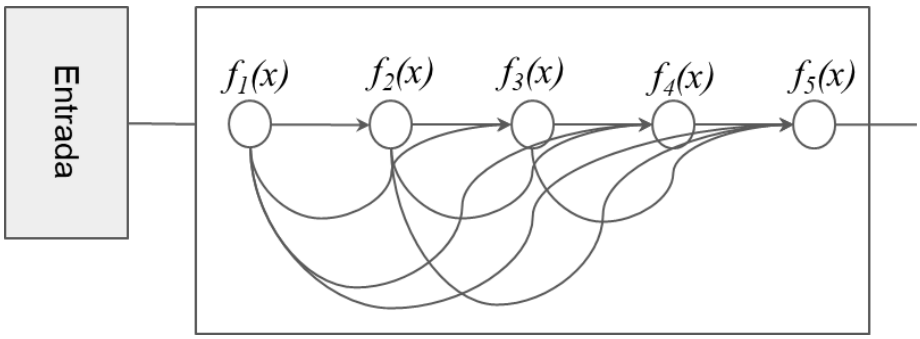


Figura 40. Ejemplo de un bloque denso de 5 capas de una red DenseNet. Cada capa toma como entradas las salidas de todas las capas que la preceden en el bloque.

Como lo ilustra la Figura 40, en el bloque DenseNet se conectan las salidas de todas las capas intermedias (con tamaños de mapa de características coincidentes) hacia las siguientes capas, para garantizar el máximo flujo de información entre ellas. Para ello, las unidades de convolución al interior de los bloques Dense tienen el mismo número de canales de salida, con el fin de realizar la concatenación a lo largo de esta dimensión. De acuerdo con lo anterior, la última capa del bloque recibe los mapas de características de todas las capas anteriores. En consecuencia, el mapa de características de salida del bloque DenseNet estará dado por la concatenación de las salidas de cada una de las capas intermedias del bloque, así:

$$x \to \left[x, f_1(x), f_2\left(x, f_1(x)\right), f_3\left(x, f_1(x), f_2\left(x, f_1(x)\right), \cdots\right)\right] \quad (79)$$

De acuerdo a la estructura de la red DenseNet, cada una de las funciones compuestas $f_n(x)$ involucran tres capas: *Batch normalization*, función de activación (ej. ReLU) y una capa de convolución 3x3. Para interconectar los bloques densos, la red incluye capas de transición conformadas por una capa convolucional 1x1 que permite controlar el número de canales y una capa de *pooling* promedio 2x2 para realizar *downsampling*. En conclusión, las principales unidades que componen DenseNet corresponden a los bloques densos y las capas de transición como se muestra a continuación.

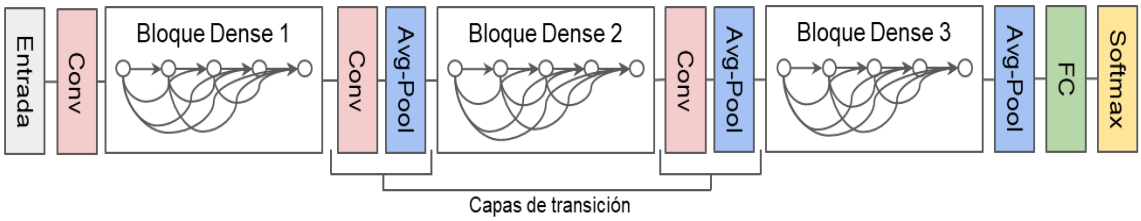


Figura 41. Arquitectura DenseNet.

Transferencia de aprendizaje

El aprendizaje por transferencia es una estrategia de diseño en la que la información de un modelo pre-entrenado para una tarea específica se utiliza en un nuevo problema, es decir, es posible aprovechar los conocimientos de dicho modelo. Los dos modelos pueden diferenciarse ya sea en la complejidad de las dos tareas, o también, es posible realizar la transferencia desde un modelo entrenado con más datos a otro con menos datos (utilizando un conjunto de datos diferente). En este sentido, la transferencia de aprendizaje puede ser útil para modelos de última generación y también cuando no hay suficientes datos de entrenamiento.

A nivel de clasificación de imágenes, es posible encontrar modelos top en el estado del arte, que por lo general han sido entrenados con ImageNet y están disponibles para descarga tanto en repositorios como para importarlos directamente en los *frameworks* de desarrollo de modelos de aprendizaje profundo. Algunas opciones que permiten obtener estos modelos se listan a continuación:

- TensorFlow Hub: https://www.tensorflow.org/hub/
- PyTorch Hub: https://pytorch.org/hub/
- HuggingFace: https://huggingface.co
- Keras applications: https://keras.io/api/applications/

A septiembre de 2023, *keras applications* dispone los siguientes modelos para realizar transferencia de aprendizaje:

- Xception
- VGG (16, 19)
- ResNet (50 (v1, v2), 101 (v1, v2), 152 (v1, v2))
- Inception v3
- InceptionResNetv2
- MobileNet (v1, v2)
- DenseNet (121, 169, 201)
- NASNetMobile
- NAsNetLarge
- EfficientNet (B0-B7)
- EfficientNetV2 (B0-B3, S, M, L)
- ConvNext (Tiny, small, base, Large, Xlarge)

Estos modelos que han sido entrenados en un problema sirven como punto de partida en un problema relacionado, reduciendo el tiempo de entrenamiento y por lo general disminuyendo el error de generalización. Al importarlos pueden aplicarse directamente en la predicción, para extraer características o también para realizar un ajuste fino del nuevo modelo sobre un problema diferente.

Predicción

Los modelos pre-entrenados pueden utilizarse directamente para realizar predicciones sobre las clases del dataset en las cuales fue entrenado. En el caso de ImageNet, el modelo por transferencia de aprendizaje puede utilizarse para realizar predicciones sobre datos de algunas de las 1000 clases de este dataset.

Para importar un modelo por transferencia de aprendizaje en Colaboratory, es necesario importar el modelo específico desde keras/tensorflow. También es necesario importar los métodos que permiten pre-procesar y adaptar las imágenes a los requerimientos del modelo y su predicción.

Al instanciar la clase que importa el modelo (ej. VGG16) es posible especificar si se importa solamente la estructura del modelo, o también se importan los

parámetros (pesos) del modelo pre-entrenado con ImageNet. Si se incluyen estos pesos, el modelo estará listo para realizar predicciones sobre nuevos datos. El siguiente bloque de código muestra el proceso para realizar estas tareas. En este caso se lee una imagen cargada en el entorno de ejecución, la cual se pre-procesa y se estructura a las dimensiones de entrada de la arquitectura (224×224×3).

```
from tensorflow.keras.applications.vgg16 import
 VGG16
from tensorflow.keras.preprocessing import imag
e
from tensorflow.keras.applications.vgg16 import
 preprocess_input, decode_predictions
import numpy as np

model = VGG16(weights='imagenet')

img_path = 'Gato1.jpg'
img = image.load_img(img_path, target_size=(224
, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
print('Predicted:', decode_predictions(preds, t
op=3)[0])
```

El parámetro `top` especificado en la decodificación de predicciones (`decode_predictions`), determina el número de clases más probables que pueden corresponder a los datos de entrada.

Extracción de características

Un modelo pre-entrenado puede utilizarse también como bloque de extracción de características hacia otro clasificador. Para esto es necesario especificar al importar el modelo, que no se utilizarán las capas top o capas de clasificación de la arquitectura (últimas capas del modelo), esto mediante el argumento "`include_top=False`".


```
from tensorflow.keras.applications.vgg16 import
 VGG16
from tensorflow.keras.preprocessing import imag
e
from tensorflow.keras.applications.vgg16 import
 preprocess_input
import numpy as np

model = VGG16(weights='imagenet', include_top=F
alse)
```

Al realizar la predicción sobre un modelo importado sin sus capas top, la salida no entrega un vector de probabilidades de pertenencia a la clase, sino que entrega un mapa de características. Esto se ilustra en el siguiente ejemplo que da como resultado un mapa de características 7×7 por cada canal de salida.

```
img_path = 'Gato1.jpg'
img = image.load_img(img_path, target_size=(224
, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)


features = model.predict(x)

# Ver las dimensiones de salida del modelo en
el summary
print(features.shape)
#print(features)
print(features[0,0:7,0:7,128])
```



```
[[ 0.         5.7264395  0.         0.         0.         0.
   4.35953  ]
 [ 0.         8.568925  41.75738   50.764385   0.         0.
   0.       ]
 [ 0.         5.896989   0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         0.
   0.       ]
 [ 0.         0.         0.         0.         0.         1.1514584
   8.352236 ]]
```

En este sentido, la salida de un modelo de este tipo puede concatenarse con un modelo de aprendizaje tradicional, o también es posible especificar una nueva estructura de capas top que permita realizar un ajuste fino del modelo sobre un problema diferente (con un conjunto de datos diferente) y la posterior clasificación sobre el nuevo conjunto de datos.

Ajuste fino (*fine tuning*)

Como se ha comentado hasta aquí, los modelos pre-entrenados pueden importarse tanto en estructura como en parámetros, con la posibilidad de excluir la capa de salida (la que realiza la clasificación) incluida en las capas top del modelo. En este caso será necesario agregar al nuevo modelo una capa de salida cuyo tamaño corresponda al número de clases del nuevo conjunto de datos de destino. Una vez realizado este ajuste, es posible entrenar el nuevo modelo con el nuevo conjunto de datos.

Además de poder entrenar la capa de salida, es posible entrenar capas anteriores del modelo. De hecho, las capas de un modelo de keras/tensorflow cuentan un parámetro denominado "*Trainable*", que define si los parámetros de esa capa se conservan constantes (conocido también como congelar, con la opción False), o si por el contrario se actualizan durante cada iteración del ciclo de entrenamiento (se descongelan con la opción True). En este sentido, los parámetros de las capas nuevas que se han agregado al modelo se inicializan de forma aleatoria y se entrenan desde cero. Los parámetros de las capas que se descongelan inician con los valores que traían del modelo original, y se empiezan a ajustar a los nuevos datos durante el entrenamiento en un proceso conocido como ajuste fino (*fine tuning*).

La importación de un modelo para *fine tuning* es similar a la importación para extracción de características (es decir, no se importan las capas top):

```
from tensorflow.keras.applications.vgg16 import V
GG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Global
AveragePooling2D

base_model = VGG16(weights='imagenet', include_to
p=False)
```

Luego de importar el modelo, se agregan las capas top para la clasificación. Para este ejemplo, se agregará una capa de *average pooling,* una capa densa de 1024 unidades y una capa densa de 2 unidades. Este último valor corresponde al número de clases del nuevo problema a abordar, que para el ejemplo es un caso binario.

```
# Agregar una capa de average pooling
x = base_model.output
x = GlobalAveragePooling2D()(x)

# Agregar una capa Fully Connected
x = Dense(1024, activation='relu')(x)

# Agregar una capa FC del número de clases
predictions = Dense(2, activation='softmax')(x)

# Modelo final para entrenar
model = Model(inputs=base_model.input,
outputs=predictions)
```

Una vez se ha consolidado el modelo (modelo base + nuevas capas top) es posible especificar cuáles de las capas se entrenarán y cuáles no. Una alternativa, por ejemplo, consiste en congelar todas las capas que se importaron del modelo original (modelo base) y entrenar solamente las nuevas capas top, como lo muestra el siguiente bloque de código.

```
For layer in base_model.layers:
    layer.trainable = False

model.compile(...)
```



```
model.fit(...)
```

La segunda alternativa consiste en definir cuáles capas se entrenarán y cuáles no. Para ello, primero se pueden visualizar los nombres de las capas y sus índices para determinar cuáles capas congelar:

```
for i, layer in enumerate(base_model.layers):
   print(i, layer.name)
```

A continuación, se definen las capas que no se entrenarán con el parámetro en `False` y las capas que sí se entrenarán con el parámetro en `True`. En el siguiente ejemplo, se congelan las primeras quince capas y se entrenarán las capas restantes.

```
for layer in model.layers[:15]:
  layer.trainable = False
for layer in model.layers[15:]:
  layer.trainable = True
```

A partir de esto, es posible compilar y entrenar el modelo de la forma usual.

ANEXO 1. ENTORNO DE EJECUCIÓN

El lenguaje de programación que se utiliza en el presente libro es Python, el cual puede ser ejecutado en diferentes entornos de ejecución como distribuciones basadas en Conda[27] y notebooks tipo Jupyter[28]. Estos entornos de ejecución permiten la inclusión de frameworks como Tensorflow[29], OpenCV[30] o Keras[31], que permiten la definición de modelos de aprendizaje automático y el procesamiento de imágenes.

Miniconda es una distribución libre y liviana para conda. Es una versión sencilla de Anaconda que incluye sólo conda, Python, los paquetes de los que dependen, y un pequeño número de otros paquetes útiles, incluyendo pip, zlib, entre otros. Sin embargo, mediante el comando "*conda install"* es posible instalar más de 720 paquetes adicionales desde el repositorio de Anaconda[32].

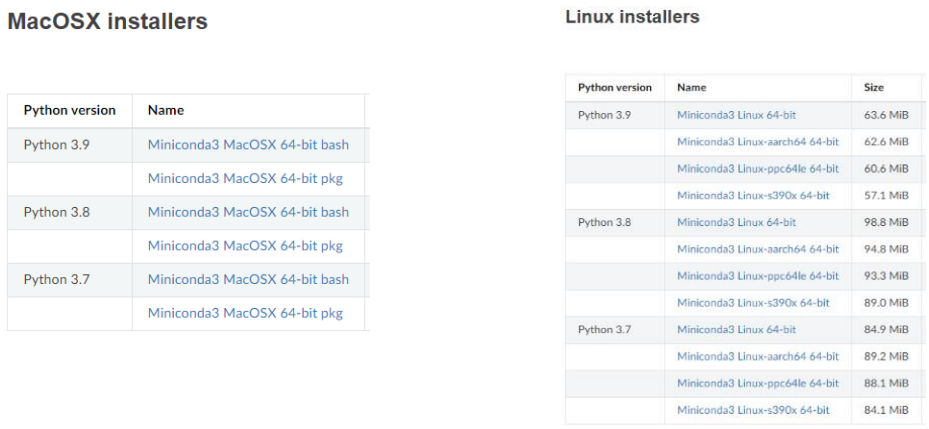
Latest Miniconda Installer Links

Platform	Name	SHA
Windows	Miniconda3 Windows 64-bit	b33
	Miniconda3 Windows 32-bit	24f
MacOSX	Miniconda3 MaxOSX 64-bit bash	786
	Miniconda3 MaxOSX 64-bit pkg	8fa
Linux	Miniconda3 Linux 64-bit	1ea
	Miniconda3 Linux-aarch64 64-bit	487
	Miniconda3 Linux-ppc64le 64-bit	fa9
	Miniconda3 Linux-s390x 64-bit	1fa

Windows installers

Python version	Name	Size
Python 3.9	Miniconda3 Windows 64-bit	58.1 MiB
Python 3.8	Miniconda3 Windows 64-bit	57.3 MiB
Python 3.7	Miniconda3 Windows 64-bit	55.8 MiB
Python 3.9	Miniconda3 Windows 32-bit	55.3 MiB
Python 3.8	Miniconda3 Windows 32-bit	54.5 MiB
Python 3.7	Miniconda3 Windows 32-bit	55.3 MiB

[27] https://github.com/conda/conda
[28] https://jupyter.org
[29] https://www.tensorflow.org
[30] https://opencv.org
[31] https://keras.io
[32] https://conda.io/en/latest/miniconda.html



MacOSX installers

Python version	Name
Python 3.9	Miniconda3 MacOSX 64-bit bash
	Miniconda3 MacOSX 64-bit pkg
Python 3.8	Miniconda3 MacOSX 64-bit bash
	Miniconda3 MacOSX 64-bit pkg
Python 3.7	Miniconda3 MacOSX 64-bit bash
	Miniconda3 MacOSX 64-bit pkg

Linux installers

Python version	Name	Size
Python 3.9	Miniconda3 Linux 64-bit	63.6 MiB
	Miniconda3 Linux-aarch64 64-bit	62.6 MiB
	Miniconda3 Linux-ppc64le 64-bit	60.6 MiB
	Miniconda3 Linux-s390x 64-bit	57.1 MiB
Python 3.8	Miniconda3 Linux 64-bit	98.8 MiB
	Miniconda3 Linux-aarch64 64-bit	94.8 MiB
	Miniconda3 Linux-ppc64le 64-bit	93.3 MiB
	Miniconda3 Linux-s390x 64-bit	89.0 MiB
Python 3.7	Miniconda3 Linux 64-bit	84.9 MiB
	Miniconda3 Linux-aarch64 64-bit	89.2 MiB
	Miniconda3 Linux-ppc64le 64-bit	88.1 MiB
	Miniconda3 Linux-s390x 64-bit	84.1 MiB

Figura 42. Ejemplos de opciones de instalación de *miniconda* para diferentes sistemas operativos.

En su repositorio[33], es posible descargar instaladores para Windows (32 y 64 bits), MacOSX (64 bits) y Linux (64 bits), con la posibilidad de seleccionar entre las últimas versiones de Python. Una vez se instala y ejecuta la aplicación, es posible ejecutar también *jupyter notebook* (Figuras A1.2 y A1.3).

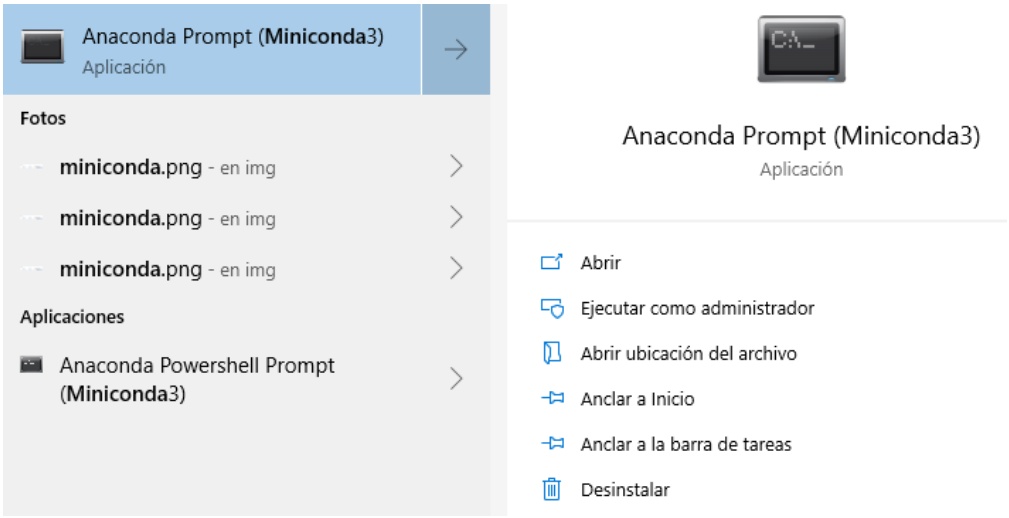


Figura 43. Ejecución de *miniconda* en Windows.

[33] https://conda.io/en/latest/miniconda.html

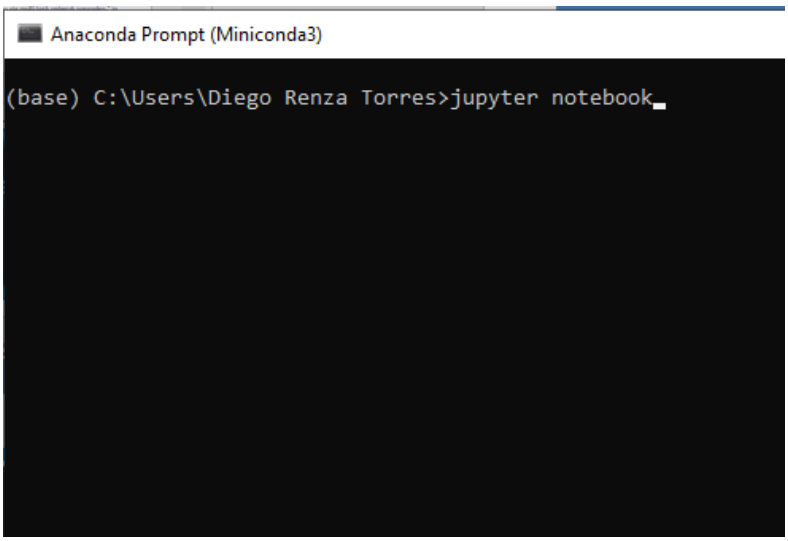


Figura 44. Ejecución de *jupyter notebook* en Windows.

Al ejecutar *jupyter notebook* se lanza el navegador web predeterminado del sistema accediendo al computador local a través de la interfaz de red loopback (localhost) en el puerto predeterminado 8888 y mostrando el árbol de carpetas y archivos del sistema (Figura A1.4). En el contexto de ejecución del *jupyter notebook*, los archivos principales tendrán extensión ipynb, como se muestra en la Figura A1.5.

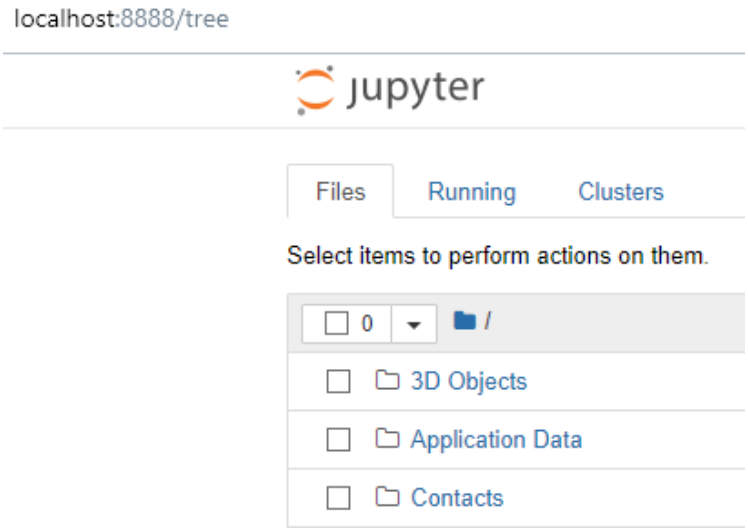


Figura 45. Ambiente de trabajo en *jupyter notebook.*

Figura 46. Ejemplos de archivos creados con *jupyter notebook.*

Los archivos ipynb se caracterizan por incluir dos tipos de bloques principales: bloques de texto y bloques de código (Figura A1.6). En los bloques de texto es posible incluir explicaciones, ecuaciones, figuras, divisiones de sección, entre otras que faciliten el contexto del código contenido en el bloc de notas. En cuanto a los bloques de código, su característica principal es la de posibilitar la ejecución tanto de líneas seleccionadas, como de bloques independientes (bloque a bloque) o ejecutar todo el contenido del bloc de notas (Figura A1.7).

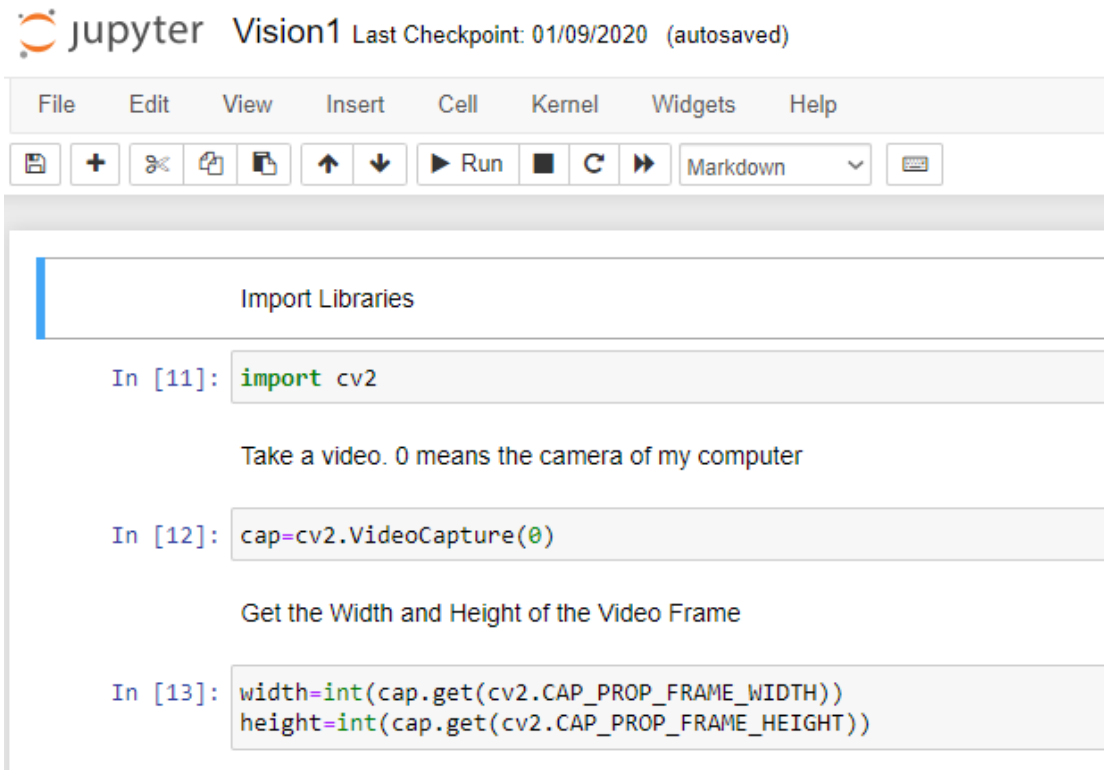


Figura 47. Ejemplos de bloques de código en un *jupyter notebook.*

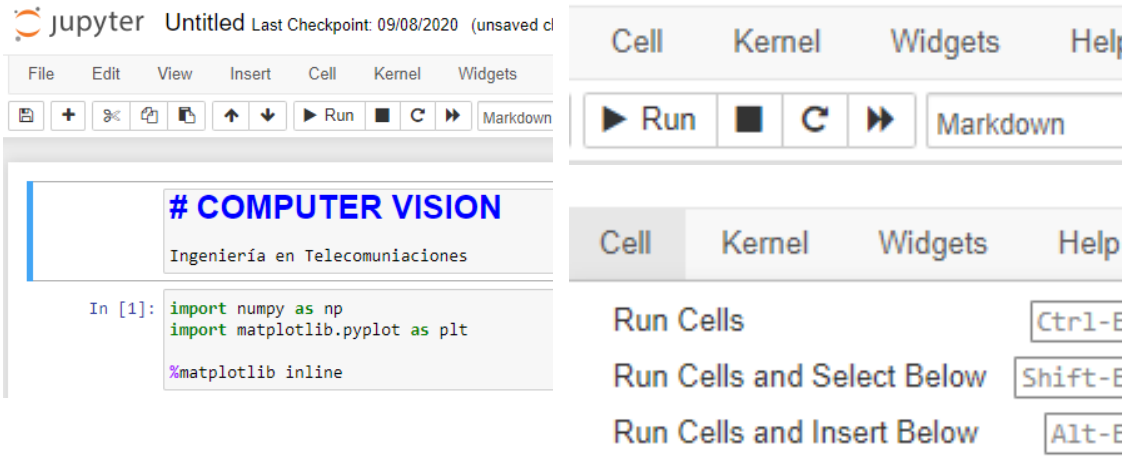


Figura 48. Opciones de ejecución de código en un *jupyter notebook.*

A manera de ejemplo, la Figura A.1.8 muestra varios bloques de código en el cual se importa Tensorflow, se crea un vector de 12 elementos con los números del 0 al 11, y luego se aplica un operador de multiplicación. Cada bloque se ejecutó de manera independiente, donde los números entre corchetes indicados en la parte izquierda indican el orden de ejecución de estos. Adicionalmente, y de acuerdo al tipo de operación realizada, es posible visualizar el resultado de las líneas ejecutadas, como lo muestra las salidas de los bloques 3 y 4.

```
In [2]: import tensorflow as tf

In [3]: x = tf.range(12)
        x

Out[3]: <tf.Tensor: shape=(12,), dtype=int32, numpy=array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])>

In [4]: x*2

Out[4]: <tf.Tensor: shape=(12,), dtype=int32, numpy=array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22])>

In [ ]:
```

Figura 49. Bloques de código en un *jupyter notebook.*

Existen herramientas para ejecución de *jupyter notebook* en servicios alojados en la nube. Una de las más utilizadas en la actualidad es Google CoLaboratory, el cual es un proyecto de investigación de Google creado para ayudar a difundir la educación y la investigación del aprendizaje automático. Es un entorno de *Jupyter notebook* que no requiere ninguna configuración para su uso y se ejecuta

completamente en la nube[34]. Por lo tanto, Colaboratory (conocido simplemente como Colab), permite escribir y ejecutar código Python en un navegador web, incluyendo además acceso gratuito a una GPU y facilitando el trabajo compartido de estos archivos[35].

Para el uso de Colab, es necesario agregarlo a las aplicaciones del entorno G-suite (https://colab.research.google.com). Posterior a su instalación y ejecución, se agrega una carpeta en Google drive diseñada para repositorio de estos blocs de notas, aunque es posible almacenarlos y ejecutarlos desde cualquier carpeta de este drive.

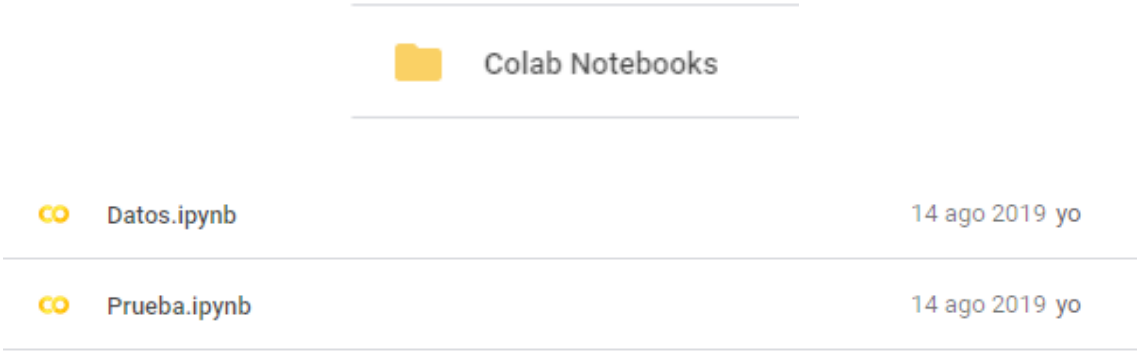


Figura 50. Estructura de archivos de CoLaboratory en Google drive.

Figura 51. Entorno de ejecución de un notebook en *CoLaboratory.*

[34] https://research.google/tools/
[35] https://colab.research.google.com/notebooks/intro.ipynb#scrollTo=5fCEDCU_qrC0

Para ampliar la comprensión de las funcionalidades de *jupyter notebook*, se sugiere al lector la consulta del siguiente enlace:

https://nbviewer.jupyter.org/github/jupyter/notebook/tree/master/docs/source/examples/Notebook/

ANEXO 2. INTRODUCCIÓN A PYTHON Y OPENCV

En este capítulo se realizará una introducción al lenguaje de programación Python, incluyendo variables, operadores, condicionales, iteraciones, algunos tipos de datos y funciones.

Como primer acercamiento al lenguaje, se visualiza en pantalla el mensaje "*Hola mundo*" mediante la función `print()`, y se define como argumento de la función la cadena de caracteres.

```
print("Hola mundo")
```

Indentación: Algo importante a tener en cuenta en Python es el uso obligatorio de indentación para definir el alcance de un determinado entorno. Por ejemplo, las instrucciones asociadas a un ciclo `for` deberán tener un nivel de indentación respecto a esta función, así:

```
for i in range(3): # Imprimir de 0-2
  print(i)
```

Variables

En Python, no es necesario asignar el tipo de una variable cuando se utiliza por primera vez. Los principales tipos de variables son *integer, float, string, boolean, NoneType*. En cualquier caso, siempre es posible verificar el tipo de una variable usando `type()`, o el estado de una variable lógica utilizando `bool()`. Para convertir entre tipos de datos, es posible utilizar funciones como `str()`, `int()` o `float()`. Es posible también ingresar cadenas texto mediante la función `input()`.

```
# Integer
num1 = 2021

# float
num2 = 20.21
```



```
# String
mensaje = "Teleco UMNG"

# Boolean
var_logica = False
#Al utilizar True, False (la primera letra deberá
estar en mayúscula)

# None
num3 = None
```

```
# Verificar contenido y tipo de una variable
print(num1)
type(num1)

# Prueba de variable lógica
bool("valor") # True
bool("") # False
```

```
# Ingreso de datos por usuario
name = input("Enter your name: ") # name es de tip
o String
print("El nombre ingresado es: " + name)
```

Operadores

Python dispone de operadores aritméticos, de comparación y lógicos como cualquier lenguaje de programación. Los principales operadores aritméticos incluidos son (entre paréntesis se especifica el operador):

> Suma (+), resta (-), multiplicación (*), división (/), módulo de la división (%), parte entera de la división (//), asignación (=), suma y asignación (+=), resta y asignación (-=), multiplicación y asignación (*=), división y asignación (/=).

Como operadores de comparación, se tiene comprobación de igualdad (==), de diferencia (!=), mayor que (>), menor que (<), mayor o igual (>=) y menor o igual (<=).

En cuanto a operadores lógicos, se cuenta con operadores booleanos tipo `and`, `or` y `not`.

El operador de concatenación de cadenas de texto en python es "+", como lo muestra la siguiente línea de código:

```
var = 5.0
print("Obtendré un " + str(var) + " en Computer Vi
sion")
```

Condicionales

El uso de condicionales en Python (`if`, `else`, `elif`) tiene la versatilidad de la mayoría de lenguajes de programación, tal como se relaciona a continuación:

Uso básico:

```
var = True
if var:
  print("Condición verdadera")
else:
  print("No cumple la condición")
```

Evaluación de una única condición:

```
a = 5
b = 10

if a == 5:
  print('Ok')
```

Evaluación de diferentes condiciones:

```
if var_mes == 4:
  print('Estamos en febrero')
elif var_mes == 6:
  print('Estamos en junio')
elif var_mes == 9:
  print('Estamos en septiembre')
```



```
elif var_mes == 11:
  print(' Estamos en noviembre')
else:
  print('Este mes no es de 30 días')
```

Evaluación utilizando comparación y operadores lógicos :

```
# AND
if a == 1 and b == 2:
  print('Los dos números son enteros positivos
menores a 3')

# OR
if a == 0 or a == 1:
  print('El factorial del número es 1')

# NOT
if not a == 0:
  print('El número es diferente de cero')
```

Existe también la posibilidad de evaluar un ocndicional en una sola línea:

```
# única condición
if var1 >= var2: print("La primera variable es
mayor o igual a la segunda")

# If + Else
print('A vale
cero ') if a == 0 else print('A es diferente a
cero')
```

Para el uso de if anidados, es importante tener en cuenta la indentación:

```
num = 123
if num > 0:
  print("El número es positivo")
  if num > 5:
    print("El número es positivo mayor a 5")
```



```
  if num > 55:
    print("El número es positivo mayor a 55")
else:
  print("El número es negativo o cero")
```

Iteraciones (loops)

Uso de ciclo `for` para iterar sobre elementos de un arreglo, o una cadena de texto:

```
programa = "Teleco"
for char in programa:
  print(char) # Imprime cada carácter de la cadena
de texto

vigencia = ['2', '0', '2', '1']
for ele in vigencia: # Imprime cada elemento del a
rreglo
  print(ele)
```

Es posible también generar una secuencia de números utilizando `range`. Aquí se especifica el valor de inicio (opcional, si no se especifica se utiliza 0), el valor final (obligatorio) y el valor de incremento (opcional, si no se especifica se utiliza 1).

```
# Ciclo for y Range para visualizar los números
impares entre 1 y 10
for i in range(1, 10, 2):
  print("Número " + str(i))
else:
  print("Fin")
```

Además, los valores generados por `range` pueden ser utilizados para indexar los elementos en un arreglo, por ejemplo:

```
arr = ['2', '0', '2', '1']
for index in range(0, len(arr)):
  print(arr[index])
```


Al igual que en los condicionales, es posible anidar ciclos `for` (con un manejo adecuado de indentación):

```
# Ciclos For anidados
for i in range(5):
  for j in range(10):
    print("Conteo interno" + str(j))
  print("Conteo externo..." + str(i))
```

Por su parte, para el uso de la sentencia `while`, es necesario realizar el incremento de la variable utilizada en la condición:

```
i = 2
while i <= 10:
  print(i)
  i += 2
```

Hay que tener en cuenta que tanto `for` como `while` admiten el uso de `break`, `continue` y `else`:

```
while i <= 25:
  print(i)
  i += 1
  if i == 18:
    break
else:
  print('Completado')
```

Contenedores de datos

Listas: las listas son un tipo de arreglo ordenado en Python, cuyos elementos pueden ser de diferente tipo. Por ejemplo, la siguiente lista contiene valores enteros, cadenas de texto y valores booleanos especificados entre corchetes [ ]:

```
lista = [1, 5, "xyz", True, "Telec0", 2021]
```


Al ser un arreglo ordenado, es posible acceder, adicionar o borrar elementos mediante índices. Aquí hay que tener en cuenta que el índice del primer elemento es 0, el segundo tendrá un índice 1, y así sucesivamente. Es posible utilizar números negativos como índices, ante lo cual el índice -1 corresponde al último elemento de la lista, el -2 al penúltimo, y así sucesivamente. Por su parte, es posible rangos del arreglo mediante el operador dos puntos (:). Al utilizar este operador como índice único, se seleccionará el arreglo completo, mientras que si acompaña de valores que lo precedan o que lo sucedan, estos harán las veces de inicio y fin de rango, respectivamente.

```
# Acceso a los elementos de una lista por índices
lista[0]
lista[-1]
lista[-2]
lista[2:4] # inicio : fin-1
lista[:3] # 0... fin-1
lista[3:] # inicio... fin
lista[:] # Arreglo completo
```

Para cambiar un valor de la lista, bastará con asignar el nuevo valor en la posición o índice adecuado:

```
#Cambiar un valor de una lista especificando su
posición
lista[4] = "Teleco"
```

Para agregar elementos a una lista, es posible utilizar tanto el método `append()`, como el método `insert()`. El primero de ellos agrega el nuevo elemento al final de la lista, mientras que el segundo inserta el objeto antes del índice especificado.

```
lista.append("UMNG")
lista
lista.insert(4, "Computer vision") # Insertar en u
na posición específica
lista
```


En cualquier caso, es posible obtener el número de elementos de un contenedor, utilizando `len()`.

```
len(lista)
```

Para eliminar objetos de una lista, basta con utilizar la sentencia `del`, el contenedor y la posición a eliminar. Para eliminar y devolver un elemento específico (posición) de una lista, el método `pop()` es el indicado. Sin embargo, si en este método no se especifica el índice, por defecto se eliminará el último elemento. Además, es posible especificar elementos a eliminar de una lista, de acuerdo con su valor. Para ello se utiliza el método `remove()` y el valor que debe tener el elemento a eliminar (el comando eliminará el primer elemento que encuentre en la lista con dicho valor).

```
# Borrar un elemento específico
del lista[4]   # opción 1
lista

# Eliminar el último elemento de la lista
elemento_eliminado = lista.pop()
print(elemento_eliminado)
print(lista)

# Eliminar elementos por su valor (el primer eleme
nto que encuentre)
lista.remove(5)
lista
```

Dado que una lista contiene elementos en posiciones específicas, es posible cambiar los elementos de posición, de acuerdo con un criterio de ordenamiento. Para esto, el método `sort()`, permite ordenar de manera ascendente (predeterminado) o descendente (especificando el parámetro `reverse=True`). Además, cuenta con la posibilidad de suministrar una función que especifique los criterios para ordenar los elementos.

```
# Ordenar (sort es una operación que no se puede d
eshacer)
```



```
letras = ["u", "m", "n", "g"]
letras.sort()
print(letras)

letras.sort(reverse=True)
print(letras)
```

Es importante tener en cuenta que esta operación no es reversible, por lo cual se deberá realizar una duplicación de la variable si se desea conservar el orden original. Para solamente devolver una nueva lista con todos los elementos en orden ascendente (sin cambiar la lista original), se puede utilizar la función `sorted()`:

```
# Imprimir datos ordenados sin cambiar el orden de
 los datos originales
letras = ["u", "m", "n", "g"]
print(sorted(letras))
print(alpha)
```

De forma complementaria, el método `reverse()`, permite ordenar los elementos de un arreglo en sentido inverso. Es decir, el último elemento de la lista se ubicará en primera posición, el penúltimo en segunda posición, etc.

```
# Invertir un arreglo
numeros = [10, 1, 5]
numeros.reverse()
print(numeros)
```

Para duplicar una lista, es importante tener en cuenta el método para realizarlo, dado que en algunos casos puede existir referenciación entre las dos variables, y un cambio en una de ellas puede verse reflejado en la otra variable, como en los siguientes ejemplos:

```
# Copia referenciado
letras = ["u", "m", "n", "g"]
letras2 = letras # Cualquier cambio en la variable
 original o en la segunda afecta la variable enlaz
ada
```



```
letras[1]="eme"
letras2[3]="gege"
print(letras)
print(letras2)
```

```
['u', 'eme', 'n', 'gege']
['u', 'eme', 'n', 'gege']
```

En el anterior código, un cambio en cualquiera de las dos variables posterior a la copia de la variable ocasionará que el contenido de las dos variables se vea alterado.

Para evitar esta situación, se puede realizar una copia independiente de todos los elementos de la lista, mediante el operador dos puntos [:]. De esta forma, al realizar un cambio en cualquiera de las dos variables posterior a su duplicación, solo afectará a la variable sobre la cual se realizó dicha operación:

```
# Copia de un arreglo no referenciado
letras = ["u", "m", "n", "g"]
letras1 = letras[:] # Copia independiente. Cualqui
er cambio en la variable original o en la segunda
NO afecta la variable enlazada
letras[1]="eme"
letras1[3]="gege"
print(letras)
print(letras1)
```

```
['u', 'eme', 'n', 'g']
['u', 'm', 'n', 'gege']
```

Otra alternativa para crear una copia independiente de una lista es utilizar el método `copy()`, el cual devuelve una copia superficial de la lista.

```
# Referenciado 2
letras = ["u", "m", "n", "g"]
letras3 = letras.copy() # Cualquier cambio en la v
ariable original o en la segunda NO afecta la vari
able enlazada
letras[1]="eme"
letras3[3]="gege"
```



```
print(letras)
print(letras3)
```

```
['u', 'eme', 'n', 'g']
['u', 'm', 'n', 'gege']
```

Tuplas: este tipo de contenedor es similar a una lista, con la diferencia de que nos es posible cambiarlas posterior a su creación. Es decir, no es posible agregar elementos ni actualizar su contenido. Por ejemplo, la siguiente tupla contiene valores enteros, cadenas de texto y valores booleanos especificados entre paréntesis ( ):

```
tupla = [1, 5, "xyz", True, "Telec0", 2021]
```

El acceso a los elementos de la tupla se puede realizar de manera similar a los de una lista:

```
print(tupla)
print(tupla[3])
print(len(tupla))
```

```
(1, 5, 'xyz', True, 'Telec0', 2021)
True
6
```

Aunque es posible crear tuplas vacías, la adición o modificación de sus valores generará una excepción:

```
tupla[0] = "Diego"
```

```
TypeError: 'tuple' object does not support item assignment
```

```
tupla_blanco = ()  # Tupla vacía
tupla_blanco[0] = "Diego"
```

```
TypeError: 'tuple' object does not support item assignment
```


Conjuntos: este tipo de contenedor corresponde a una colección no ordenada de elementos, es decir que no existen elementos duplicados dentro del conjunto y que a su vez pueden ser de diferente tipo. Por ejemplo, la creación del siguiente conjunto con valores enteros y caracteres especificados entre llaves { }, devolverá un solo elemento por tipo:

```
conjunto = {'u', 'm', 'n', 'g', 2, 0, 2, 2}
print(conjunto)
```

```
{0, 2, 'm', 'n', 'g', 'u'}
```

Al igual que en las listas, es posible iterar sobre los elementos del conjunto (teniendo en cuenta las particularidades de este contenedor). Asimismo, el número de elementos del conjunto se puede obtener de forma similar a una lista o a una tupla:

```
# Acceder a elementos en un conjunto (a través de
un ciclo)
for elemento in conjunto:
  print(elemento)

# Longitud de un conjunto
len(conjunto)
```

```
0
2
m
n
g
u
```

Para adicionar elementos a una conjunto, es posible utilizar el étodo add() (para un elemento) o el método update para más de un elemento:

```
# Agregar 1 elemento
conjunto.add('a')
print(conjunto)

# Agregar más de un elemento
conjunto.update(['b', 'c', 'd'])
```



```
print(conjunto)
```

```
{0, 2, 'm', 'n', 'g', 'a', 'u'}
{0, 2, 'd', 'm', 'n', 'g', 'a', 'c', 'b', 'u'}
```

Para eliminar un elemento de la lista, Python dispone del método `remove()` y del método `discard()`, siendo más recomendable este último.

```
# Eliminar un elemento de la lista
conjunto.remove('a')
print(conjunto)

conjunto.discard('b') # Opción preferida
print(conjunto)
```

```
{0, 2, 'd', 'm', 'n', 'g', 'c', 'b', 'u'}
{0, 2, 'd', 'm', 'n', 'g', 'c', 'u'}
```

Diccionarios: tipo de contenedor en Python que corresponde a una colección desordenada que puede actualizarse. La forma de definir un diccionario en Python requiere definir para cada elemento una llave y su respectivo valor ():

```
diccionario = {
  "key" : "value"
}
```

Por ejemplo, el siguiente diccionario contiene cadenas de texto para tres elementos:

```
empleado = {
    "nombre": "Diego",
    "apellido": "Renza",
    "cargo": "Docente"
  }
```

El acceso a los valores del diccionario puede realizarse indexando la llave del elemento (entre corchetes []):

```
# Acceder a valores
```



```
empleado["nombre"]      # Opción 1
```

```
'Diego'
```

Sin embargo, al realizarlo de esta forma se genera un error cuando no existe la llave especificada. En su lugar, puede ser más adecuado utilizar el método `get()`:

```
empleado.get("nombre")
```

Al igual que en los casos anteriores, el número de elementos de la lista se obtiene con la función `len()`. Sin embargo, para los diccionarios es posible listas independientes para las llaves y para los valores:

```
# Obtener todas las llaves y valores de un diccion
ario (en una lista)
llaves = empleado.keys()
print(llaves)
# Valores
valores = empleado.values()
print(valores)
```

```
dict_keys(['nombre', 'apellido', 'cargo'])
dict_values(['Diego', 'Renza', 'Docente'])
```

Asimismo, es posible iterar sobre los elementos de un diccionario utilizando el método `items()` con la diferencia de que en este caso se tendrán variables tanto para la llave como para su valor:

```
for key, value in empleado.items():
  print(key + " corresponde a " + value)
```

```
nombre corresponde a Diego
apellido corresponde a Renza
cargo corresponde a Docente
```

Al utilizar el método `enumerate()` se tendrán variables tanto para el índice de la llave como para su valor:

```
for index, value in enumerate(empleado):
```



```
  print("El índice " + str(index) + " del diccionario corresponde a la llave " + value)
```

```
El índice 0 del diccionario corresponde a la llave nombre
El índice 1 del diccionario corresponde a la llave apellido
El índice 2 del diccionario corresponde a la llave cargo
```

Funciones

La creación de funciones en Python involucra la utilización del comando `def`, el nombre de la función y especificar los argumentos necesarios para la ejecución de la función. Al crear la función, es preciso utilizar la debida indentación y a diferencia de otros lenguajes, no es obligatorio especificar el valor retornado por la función (en caso tal se utilizará `return` y el valor a retornar por la función). Para ejecutar la función, basta con especificar los argumentos de la misma como una tupla. En el siguiente código de ejemplo, se crea una función para la suma de una lista de números, donde se ha utilizado * para especificar que el número de argumentos de la función es variable:

```
def suma2(*num):
  print(sum(num))

suma2(1,3,7,9,15)
```

```
35
```

Al ejecutar una función en Python, es posible también ingresar los argumentos en desorden, con la condición de especificar el nombre y valor de cada argumento al llamar la función. Asimismo, es posible establecer el valor predeterminado de un argumento.

```
def visua_empleado(nombre, apellido, cargo='Docente'):
  print(nombre + " - " + apellido + " - " + cargo)

visua_empleado(apellido='Renza', nombre='Diego')
```

```
Diego - Renza - Docente
```


INTRODUCCIÓN A OPENCV

En esta sección del libro, se utilizarán algunas librerías y módulos para el manejo y visualización de datos e imágenes. En primera instancia, se importarán los módulos de extensión para Python NumPy y Matplotlib. NumPy (Numeric Python) está orientado a acelerar la ejecución de operaciones con estructuras de datos, con base en representación de matrices y arreglos multidimensionales. En el segundo caso, este permite activar el soporte interactivo de gráficas con `matplotlib` en cualquier momento de una sesión.

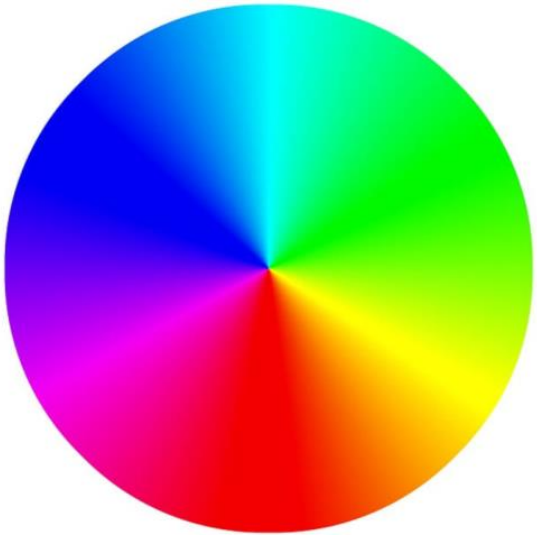
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
from PIL import Image
```

Para el manejo de imágenes en Python, en primera instancia se utilizará la librería Pillow (PIL), en particular con el módulo *Image*. Al utilizar este módulo se dispone de una clase para representar imágenes, además de la posibilidad de realizar operaciones sobre ellas (ej. carga desde archivo, creación de nuevas imágenes, etc.).

https://pillow.readthedocs.io/en/stable/reference/Image.html

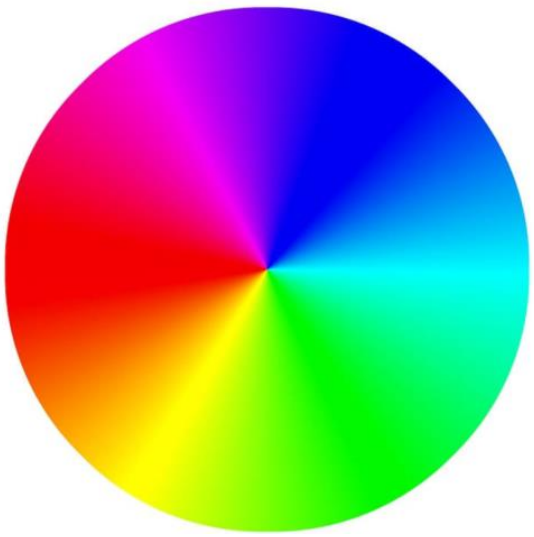
Para abrir una imagen desde un archivo con PIL/Image se puede utilizar el método open, especificando la ruta del archivo (de no existir el archivo en la ruta especificada, se generará un error). Para visualizarla, bastará con llamar a la variable que la contiene:

```
img = Image.open('images/Test.jpg')
img
```

Como ejemplo de otro tipo de operaciones que se pueden realizar con PIL/Image, está la rotación en un ángulo específico. Para más información de esta librería, se remite al lector a la documentación oficial: https://pillow.readthedocs.io/en/stable/reference/Image.html.

```
img.rotate(-90)
```



Una vez leída la imagen, es posible comprobar el tipo de objeto al cual corresponde, que para el caso del ejemplo es una imagen tipo JPEG en formato PIL.

```
type(img)
```

```
PIL.JpegImagePlugin.JpegImageFile
```

Dado que la imagen está en formato PIL, para su procesamiento puede ser conveniente representarla como un arreglo tridimensional, utilizando el módulo NumPy. Para esto, es posible utilizar el método `asarray` sobre la variable de la imagen. Esto devolverá una variable tipo NumPy (arreglo multidimensional):

```
img_array = np.asarray(img)
type(img_array)
```

```
numpy.ndarray
```

Para obtener las dimensiones del arreglo, el método `shape` devuelve en este caso el número de filas, columnas y canales de la imagen:

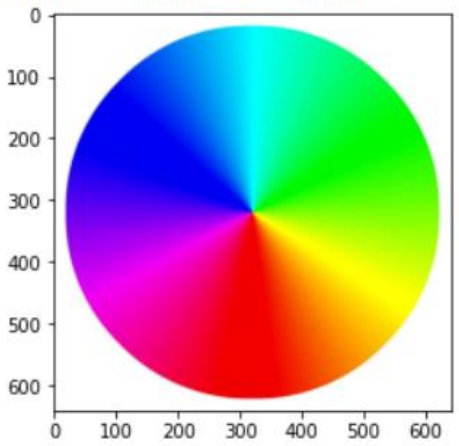
```
img_array.shape
```

```
(640, 640, 3)
```

Para visualizar la imagen en formato NumPy, una alternativa es utilizar Matplolib:

```
plt.imshow(img_array)
```

```
<matplotlib.image.AxesImage at 0x7fefcdfe4a20>
```



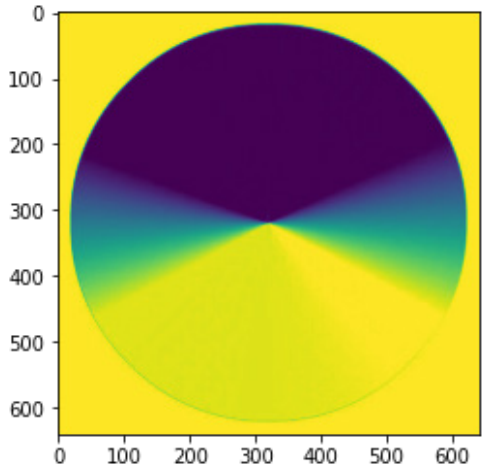
Esta imagen de ejemplo corresponde a una imagen a color (dado que tiene 3 canales), y el valor de cada píxel en cada canal está almacenado como un entero de 8 bits sin signo (*unsigned integer* de 8 bits, uint8). Es decir, el valor de intensidad de cada píxel en un canal dado oscila entre 0 (mínima intensidad o negro) y 255 (máxima intensidad o blanco).

En el caso de los canales para una imagen a color, el espacio de color o representación comúnmente usada es RGB. En este espacio de color, el primer canal o canal 0 representa las intensidades en la longitud de onda del rojo, el

segundo canal o canal 1 representa el verde y el tercer canal o canal 2 representa las intensidades del color azul. Al tener la imagen a color, cada píxel de la imagen tendrá 24 bits, es decir 8 bits/pixel/canal.

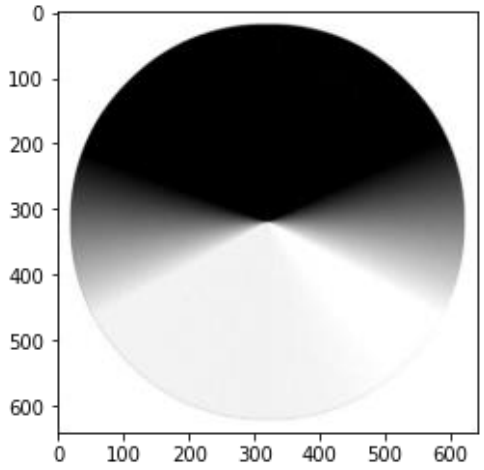
Para visualizar Para visualizar uno de los canales de la imagen, se realizará una copia de la variable que contiene el arreglo de la imagen, y en la función `imshow` se especificarán los índices respectivos a nivel de filas, columnas y canales. En este caso se mostrarán todas las filas, todas las columnas y solo el canal rojo:

```
img_test = img_array.copy()

plt.imshow(img_test[:, :, 0])
```



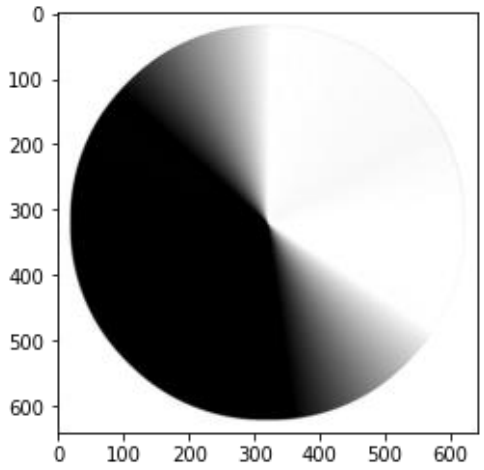
En la imagen anterior se aprecian dos aspectos fundamentales. El primero tiene que ver con las intensidades de mayor valor (las que están en color más claro, amarillo en este caso). Se puede observar que las zonas de la imagen de color rojo son las que tienen mayor intensidad. Por otra parte, aunque se especificó que `imshow` mostrara solo un canal de la imagen, la visualización se realizó como una imagen a color; esto se debe al mapa de color predeterminado de *matplolib*. Para asegurarse que la imagen se muestre en escala de grises, es necesario especificar el argumento `cmap` en gris.

```
plt.imshow(img_test[:, :, 0], cmap ='gray')
```

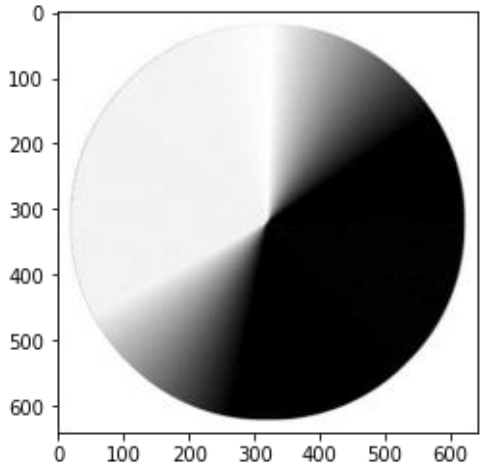
De forma similar es posible mostrar el canal verde de la imagen en escala de grises:

```
plt.imshow(img_test[:, :, 1], cmap ='gray')
```



Y el canal azul también en escala de grises:

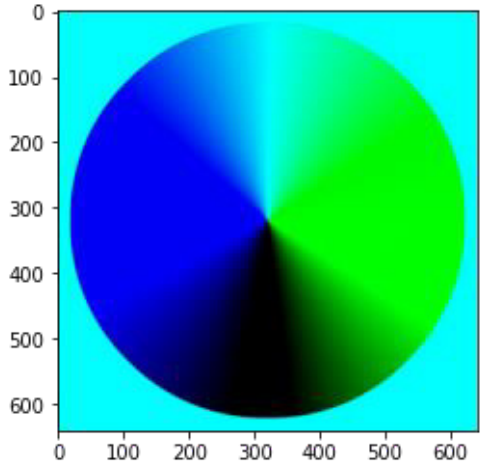
```
plt.imshow(img_test[:, :, 2], cmap ='gray')
```

Se invita al lector a analizar las imágenes anteriores, comparando las zonas de mayor intensidad respecto al canal visualizado.

De forma complementaria, y a manera de prueba, se eliminará un canal determinado, estableciendo todos los valores de píxel del canal en cero. De esta forma, al visualizar la imagen las tonalidades relacionadas con el color eliminado no se mostrarán.

```
img_test = img_array.copy()
img_test[:,:,0]=0
plt.imshow(img_test)
```



Al comparar la imagen anterior con la original, se percibe que la zona que antes era roja ahora se muestra oscura (en color negro), dado que las intensidades de píxel de dicho canal se establecieron en cero. Asimismo, el fondo de la imagen que antes era blanco (el cual se forma con el valor máximo en cada canal, es decir Rojo=255, Verde=255 y azul=255), ahora se muestra en color cian, color

que se obtiene al mezclar verde y azul (sin ningún aporte del canal rojo). Utilizando este ejemplo, se invita al lector a analizar la imagen eliminando el color verde, y luego eliminando solo el color azul.

OpenCV

OpenCV (Open Source Computer Vision Library) es una librería de software de visión por ordenador y aprendizaje automático de código abierto[36], que está disponible a través de una licencia BSD. Esta librería dispone de más de 2500 algoritmos para tareas de visión por computador orientadas principalmente a aplicaciones en tiempo real como detección de rostros, identificación de objetos, clasificación, seguimiento y rastreo de objetos, modelos 3D, o fusión de datos,

OpenCV tiene interfaces en diferentes lenguajes como C++, Python, Java y MATLAB por lo cual es compatible con diversos sistemas operativos, a saber, Windows, Linux, Android y Mac OS. Para su utilización en Python, al igual que otras librerías es necesario importarla con el comando `import cv2`:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import cv2
```

Una vez importada, la librería dispone de métodos para operación con imágenes. Por ejemplo, para cargar una imagen bastará con utilizar el método `imread` y especificar la ruta y el nombre de archivo de la imagen. Es importante verificar que la ruta y el archivo existan, dado que OpenCV no mostrará ningún error o excepción si el archivo no existe.

```
img = cv2.imread('logo_umng.jpg')
```

La imagen leída con OpenCV se almacena como un arreglo multidimensional tipo NumPy, lo que se puede corroborar con el comando `type`:

```
type(img)
```

[36] https://opencv.org


```
numpy.ndarray
```

Dado que es un arreglo multidimensional, es pertinente comprobar las dimensiones de la imagen a través del método `shape`:

```
img.shape
```

```
(625, 516, 3)
```

De esta forma, el archivo del ejemplo corresponde a una imagen de 322500 píxeles, estructurados en 625 filas y 516 columnas, y es una imagen a color, dado que tiene 3 canales.

Para visualizar la imagen, es posible utilizar el comando `imshow` de matplolib. Este comando muestra una imagen a color (RGB), desplegando en el primer canal el componente rojo (R), en el segundo canal el componente verde (G), y en el tercer canal el componente azul (B). El resultado para la imagen del ejemplo se muestra a continuación:

```
plt.imshow(img)
```

Al realizar este proceso, lo que se observa es que la imagen mostrada en pantalla se presenta en falso color. Esto sucede porque al trabajar con OpenCV, la variable que contiene el arreglo de la imagen se estructura ubicando en el primer canal el componente azul, en el segundo canal el componente verde y en el tercer canal el componente rojo, es decir con una distribución BGR. Para solventar esta situación, OpenCV permite convertir la imagen hacia RGB, y así poder utilizar un visualizador estándar como matplolib:


```
img_fix = cv2.cvtColor(img,
                       cv2.COLOR_BGR2RGB)
plt.imshow(img_fix)
```

OpenCV también permite realizar diversas operaciones de procesamiento de imagen, desde el mismo momento de su lectura. Por ejemplo, es posible leer una imagen a color en escala de grises, lo que arrojará que nuestro arreglo multidimensional solo tenga un canal (es decir solo está conformado por filas y columnas):

```
img_gray = cv2.imread('logo_umng.jpg',
                      cv2.IMREAD_GRAYSCALE)
img_gray.shape
```

```
(625, 516)
```

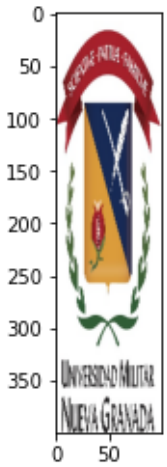
```
plt.imshow(img_gray, cmap = 'gray')
```

Algunos ejemplos de operaciones comunes en procesamiento de imagen disponibles en OpenCV se muestran a continuación:

Redimensionado de imagen:

Opción 1: especificar la variable de la imagen a operar y las dimensiones deseadas:

```
img_new = cv2.resize(img_fix,
                     (100,400))
plt.imshow(img_new)
```



Opción 2: especificar la relación de la dimensión de la imagen tanto en alto como en ancho:

```
width_ratio = 2
height_ratio = 2
img2 = cv2.resize(img_fix,
                  (0,0),
                  img_fix,
                  width_ratio,
                  height_ratio)
plt.imshow(img2)
```


Reflejo de imagen

Con respecto al eje X (eje 0):

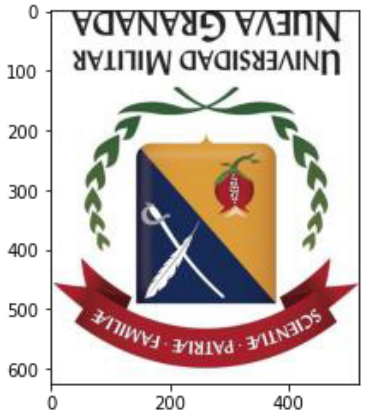
```
img_3 = cv2.flip(img_fix, 0)
plt.imshow(img_3)
```

Con respecto al eje Y (eje 1):

```
img_3 = cv2.flip(img_fix, 1)
plt.imshow(img_3)
```

Con respecto a los dos ejes:

```
img_3 = cv2.flip(img_fix, -1)
plt.imshow(img_3)
```



Referencias

Brooks, R. A. (1979). The ACRONYM model-based vision system. *Proceedings of the 6th international joint conference on Artificial intelligence*, *1*, págs. 105-113.

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence, 6*, 679-698.

Chan, T. F. (2001). Active contours without edges. *IEEE Transactions on image processing, 10(2)*, 266-277.

Chollet, F. (2021). *Deep learning with Python.* Simon and Schuster.

Clevert, D. A. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289.*

Csáji, B. C. (2001). *Approximation with artificial neural networks.* Hungary: Faculty of Sciences, Etvs Lornd University.

Dalal, N. &. (2005). Histograms of oriented gradients for human detection. *IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, *1*, págs. 886-893.

Felzenszwalb, P. M. (2008). A discriminatively trained, multiscale, deformable part model. *IEEE conference on computer vision and pattern recognition*, (págs. 1-8).

Fischler, M. A. (1973). The representation and matching of pictorial structures. *IEEE Transactions on computers, 100(1)*, 67-92.

Glorot, X. &. (2010). Understanding the difficulty of training deep feedforward neural networks. *Thirteenth international conference on artificial intelligence and statistics* (págs. 249-256). JMLR Workshop and Conference proceedings.

Godbole, V., Dahl , G., Gilmer, J., Shallue, C., & Nado, Z. (1 de 2023). *Deep Learning Tuning Playbook*. Obtenido de http://github.com/google/tuning_playbook

Harris, C. &. (1988). A combined corner and edge detector. *Alvey vision conference*, *15(50)*, págs. 10-5244.

He, K. Z. (2015). Delving deep into rectifiers. Surpassing human-level performance on imagenet classification. *IEEE international conference on computer vision* (págs. 1026-1034). IEEE.

He, K. Z. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, (págs. 770-778).

Hornik, K. M. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 359-366.

Huang, G. a. (2017). Densely connected convolutional networks., (págs. 4700--4708).

Hubel, D. H. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology, 160(1)*, 106-154.

Klambauer, G. U. (2017). Self-normalizing neural networks. *Advances in neural information processing systems, 30*.

Koṫlawī, A. Y. (s.f.). *Akhlāq-uṣ-Ṣāliḥīn.* Karachi, Pakistan: Maktaba-tul-Madīnaĥ.

Krizhevsky, A. S. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems, 25*, 1097-1105.

Lazebnik, S. S. (2006). Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, *2*, pág. 2169.

LeCun, Y. B. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation, 1(4)*, 541-551.

LeCun, Y. B. (1998). Gradient-based learning applied to document recognition. *86(11)*, 2278-2324.

Lin, M. a. (2013). Network in network. *arXiv:1312.4400*.

Liu, H. D. (2021). Pay Attention to MLPs. *arXiv preprint arXiv:2105.08050.*

Lowe, D. (1999). Object recognition from local scale invariant features. *Proceedings of the seventh IEEE international conference on computer vision*, *2*, págs. 1150-1157.

Lowe, D. G. (1987). Three-dimensional object recognition from single two-dimensional images. *Artificial intelligence, 31(3)*, 355-395.

Marr, D. &. (1980). Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, *207(1167)*, págs. 187-217.

Marr, D. (2010). *Vision, A Computational Investigation into the Human Representation and Processing of Visual Information.* The MIT Press.

Masters, D., & Luschi, C. (s.f.). *Revisiting small batch training for deep neural networks*. Obtenido de https://arxiv.org/pdf/1804.07612.pdf

Ng, A. (2019). *Machine learning yearning: Technical strategy for ai engineers in the era of deep learning.* Obtenido de Machine learning yearning: https://www.mlyearning.org

Papert, S. A. (1966). *The summer vision project.*

Roberts, L. G. (1963). Machine perception of three-dimensional solids (Doctoral dissertation. *Massachusetts Institute of Technology*.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review, 65(6)*, 386.

Rumelhart, D. E. (1986). Learning representations by back-propagating errors. *Nature, 323(6088)*, 533-536.

Saravia, E. (2021). ML Visuals. *https://github.com/dair-ai/ml-visuals*.

Shi, J. &. (2000). Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence, 22(8)*, 888-905.

Simonyan, K. &. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Szegedy, C. L. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition*, (págs. 1-9).

Tolstikhin, I. H. (2021). Mlp-mixer: An all-mlp architecture for vision. *arXiv preprint arXiv:2105.01601*.

Viola, P. &. (2004). Robust real-time face detection. *International journal of computer vision, 57(2)*, 137-154.

Zeiler, M. D. (2014). Visualizing and understanding convolutional networks. *European conference on computer vision* (págs. 818-833). Cham: Springer.

Zhang, A. L. (2021). *Dive into deep learning.* arXiv preprint arXiv:2106.11342.

