Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

18 de abril de 2024

Versión Preliminar

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro

ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: https://github.com/facundobatista/libro-pyciencia

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Parte I
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

1 | Machine learning

En el contexto de la inteligencia artificial coexisten diversas metodologías tales como sistemas expertos, agentes racionales, procesamiento del lenguaje natural, visión de computadoras, robótica, y aprendizaje de máquinas o *machine learning* (ML). Particularmente esta ultima metodología ha mostrado un desarrollo impresionante de aplicaciones en los últimos años, debido principalmente al avance en los procesadores de cálculo especializados (GPUs[1] y TPUs[2]) y a la existencia sistematizada de grandes cantidades de información (*big data*).

Módulo	Versión
Keras	3.0.5
Matplotlib	3.5.3
NumPy	1.24.2
scikit-learn	1.4.1.post1
tqdm	4.64.1

Código disponible

Entre las diferentes metodologías de ML se encuentran las redes neuronales (o NN por su sigla en inglés), cuyos orígenes se remontan al trabajo de McCulloch y Pitts de 1943 [**McCulloch1943**], quienes intentaron modelar las redes de neuronas en el cerebro mediante redes de cálculo computacional. Existen numerosas topologías o *arquitecturas*[3] de estas redes, que se especializan para realizar diferentes tareas de como ajuste de funciones, clasificación, identificación de patrones, detección de agregados ("*clustering*"), etc.

Analizaremos aquí un modelo simple de red del tipo *feedforward* (o de avance o propagación directa), en las cuales las conexiones entre distintas neuronas son solo en una dirección, formando de este modo un grafo acíclico entre las entradas y las salidas. Cada nodo de la red computa una función de sus entradas y pasa el resultado a las entradas de los nodos siguientes, sin generar ciclos. Por otra parte, existen otros tipos de redes como las recurrentes, en las cuales las entradas de los nodos se retroalimentan con sus propias salidas (este tipo de redes es apropiado para el ajuste de series temporales, por ejemplo).

1.1. Nodos, pesos y funciones de activación

Una red neuronal constituye un grafo en el que los nodos representan las *neuronas*, que son unidades de cálculo inspiradas en las neuronas biológicas. Estas últimas están conectadas unas a otras mediante axones y dendritas por medio de enlaces sinápticos que realizan la transmisión de impulsos nerviosos. Este proceso es el que se simula en las redes neuronales artificiales, en las que los nodos de cada capa están conectados a los nodos de la capa siguiente de la red a través de *pesos* que representan la intensidad de la conexión sináptica entre los nodos. Una red neuronal artificial calcula una función de los valores de entrada hacia los nodos de salida, utilizando estos pesos como parámetros de dicha función. Un esquema de esta estructura se ve en la Figura 1.1, en la que se muestra la capa de nodos de entrada (en azul), varias capas intermedias (u *ocultas*,

[1] *Graphics processing unit*, unidad de procesamiento gráfico.
[2] *Tensor processing unit*, unidad de procesamiento tensorial.
[3] Ver *The Neural Network Zoo* (en inglés).

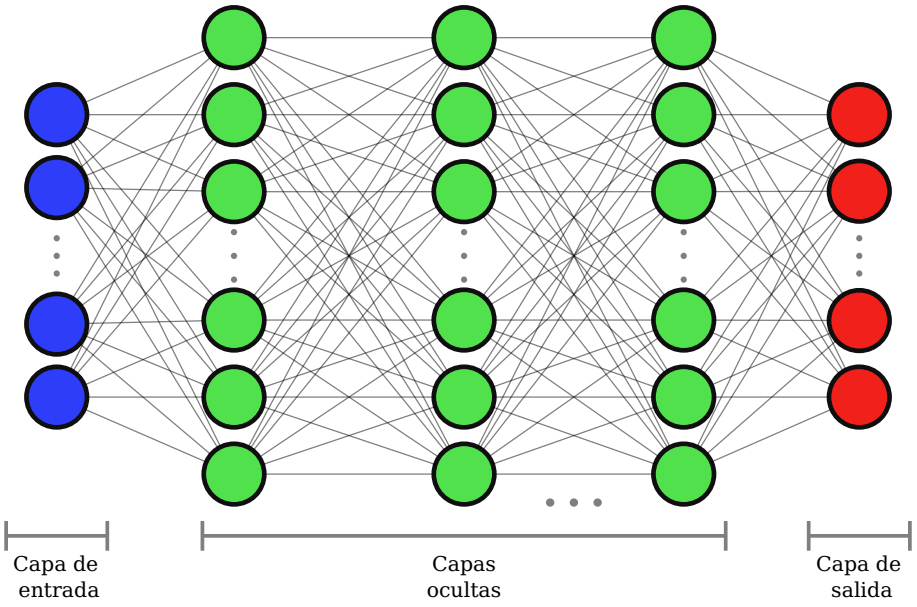


FIGURA 1.1: Representación de la estructura de una red *feedforward* multicapa.

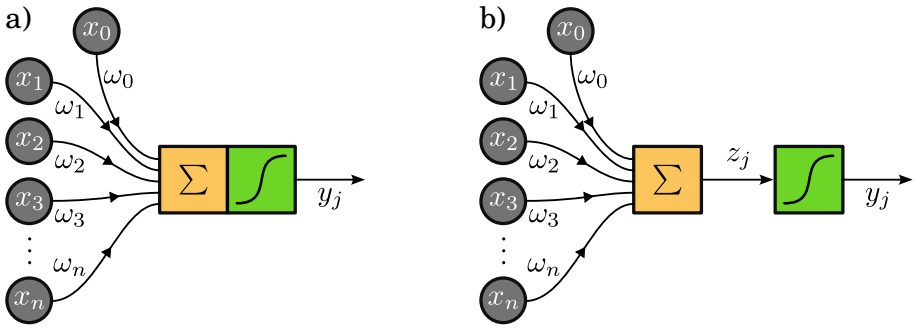


FIGURA 1.2: a) Esquema de un nodo o "neurona". b) Esquema con la separación de operaciones indicando explícitamente el valor de preactivación z_j.

en verde), y una capa de nodos de salida (en rojo). Las neuronas de cada capa están densamente conectadas con las capas vecinas.

Cada nodo (o neurona) en la red computa una suma ponderada de las entradas de sus nodos predecesores, y a este resultado le aplica una función no lineal para producir la salida, tal como se representa en la Figura 1.2. Si denotamos por y_j la salida del nodo j, y $\omega_{i,j}$ el peso correspondiente a la conexión entre el nodo predecesor i y el j, la salida del nodo j da como resultado:

$$y_j = g_j\left(\sum_{i=0}^{n} \omega_{ij} x_i\right) = g_j(z_j)$$

donde $z_j = \sum_{i=0}^{n} \omega_{ij} x_i$ es la suma ponderada de los n nodos precedentes conectados con el nodo j_i, denominado *valor de activación*. Aquí, $g_j(\cdot)$ es alguna función no lineal denominada *función de activación*. Es usual asumir que $x_0 = 1$ por lo que ω_{0j} es un término de *bias* o sesgo, y es un parámetro que permite que los nodos de la red aprendan aún cuando todas las entradas sean nulas.

Si bien tanto la transformación lineal como la posterior evaluación de la función de activación son cómputos que se realizan en una neurona, es posible considerar un nodo con su correspondiente función de activación no lineal como dos nodos computacionales, uno que realiza la

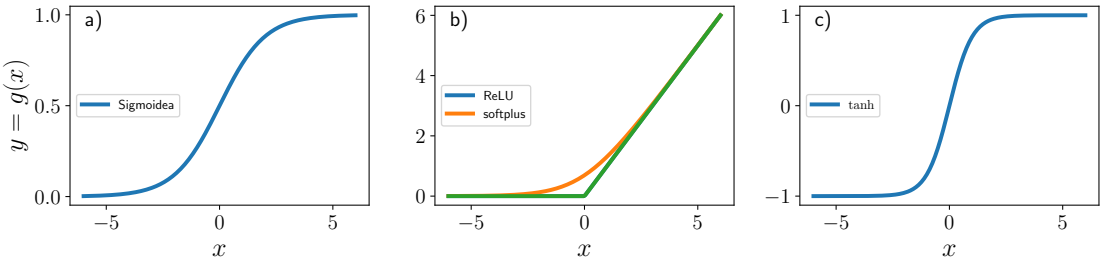


FIGURA 1.3: Funciones de activación: a) sigmoidea, b) ReLU y softplus y c) tangente hiperbólica.

transformación lineal $z_j = \boldsymbol{\omega}^\intercal \cdot \boldsymbol{x}$, siendo $\boldsymbol{\omega}$ el vector que representa los pesos que conectan las entradas $\boldsymbol{x} = [x_0, x_1, \ldots, x_n]$ del nodo j (aquí denotamos las cantidades vectoriales con letras en negrita). En el ejemplo que desarrollaremos a continuación, usaremos este esquema desacoplado de cálculo, lo que simplificará la representación computacional de los nodos.

El hecho de que la función de activación sea no lineal es de central importancia, porque si fuese lineal, cualquier composición de nodos solo podría representar una función lineal. La no linealidad de g_j es lo que permite que redes suficientemente grandes puedan representar funciones arbitrarias[4]. Existe una variedad de funciones de activación que se usan con diferentes propósitos. Adicionalmente, tienen un impacto significativo en la velocidad de aprendizaje, factor que es uno de los principales criterios para su elección. Entre las más utilizadas están:

- La función logística o sigmoidea:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- La función lineal rectificada (o ReLU, por su sigla en inglés *rectified linear unit*):

$$\text{ReLU}(x) = \text{máx}(0, x)$$

- La función *softplus*, que es una versión suavizada de la ReLU:

$$\text{softplus}(x) = \log(1 + e^x)$$

  y cuya derivada es la función sigmoidea.

- La función tangente hiperbólica:

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

La Figura 1.3 muestra las representaciones gráficas de estas funciones. Se puede apreciar que tanto la sigmoidea como la tangente hiperbólica acotan el recorrido a un intervalo finito: $(0, 1)$ para el caso de la sigmoidea y $(-1, 1)$ para la tanh. Un comporamiento diferente tienen ReLU y softplus, ya que estas funciones no están acotadas. En particular, la función softplus tiene derivada continua en $x = 0$, a diferencia de ReLU. En todos los casos, las funciones son monótonamente crecientes, por lo que en todos sus dominios las derivadas correspondientes son positivas.

Una función de activación particular, dado que se utiliza casi exclusivamente en la capa de salida, es la *softmax* o función exponencial normalizada. Esta función es una generalización de

[4]Ver el teorema de aproximación universal [**cybenko1989**, **hornik1991**, **leshno1993**].

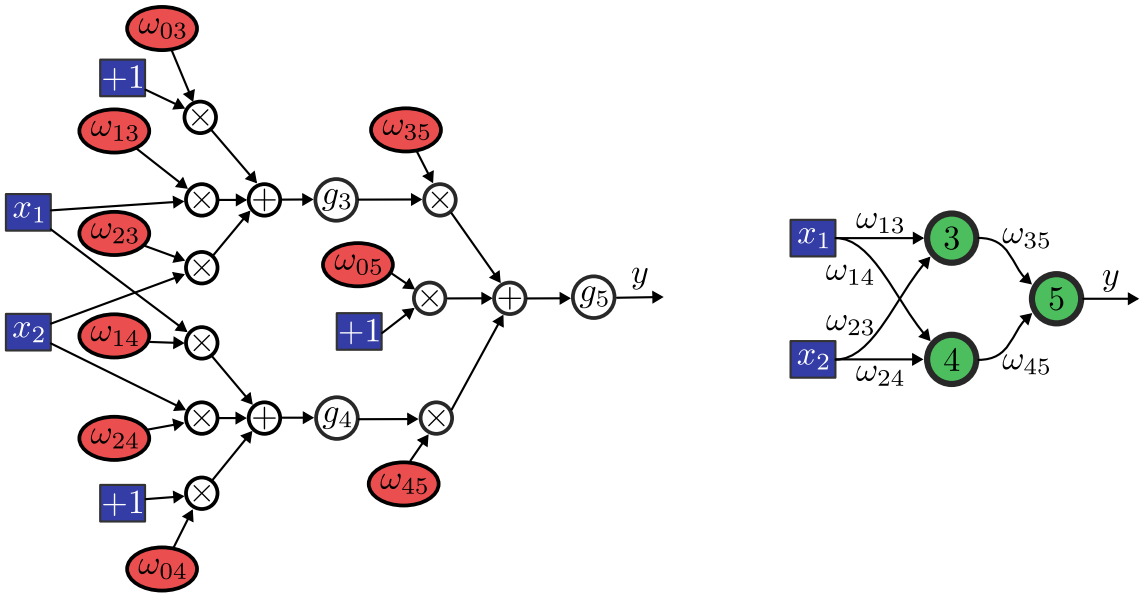


FIGURA 1.4: Flujo del cálculo de operaciones matemáticas en una red "*feedforward*". A la izquierda, el detalle para una red con dos neuronas de entrada (x_1, x_2), dos neuronas en la capa oculta, con funciones de activación g_3 y g_4, y una neurona en la capa de salida con función de activación g_5 que produce la predicción y. A la derecha, la forma simplificada usual de representación.

la función logística que convierte un vector de k números reales en una distribución de probabilidad de k eventos discretos. Específicamente, si la entrada es un vector $\boldsymbol{z} = [z_1, z_2, \ldots, z_k]$ que representa los valores de activación de los k nodos en una capa dada, la función de activación *softmax* para la salida i-ésima es:

$$\sigma(\boldsymbol{z}) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}, \quad \forall i \in \{1, 2, \ldots, k\}$$

Al utilizar un esquema desacoplado de cálculo, una capa que implementa solo la función *softmax* no requiere pesos que deban ajustarse, ya que solo transforma un vector de valores reales en probabilidades.

Una forma general de abordar el tratamiento matemático que realiza una red *feedforward* es la de considerar la red como grafo de cálculo o de "flujo de datos", esto es, un grafo donde cada nodo representa un cálculo elemental y las uniones entre los nodos representan la relación de entrada/salida de los diferentes nodos. La Figura 1.4 muestra, a la izquierda, el flujo de cálculos que se realiza a partir de las entradas x_1 y x_2 (en azul) y que realizan los nodos 3, 4 y 5 utilizando los pesos ω_{ij} (en rojo), dando como resultado el valor y. A la derecha de la Figura 1.4 se representa la red en su forma simplificada habitual.

1.2. Aprendizaje

El procedimiento por el cual se modifican los valores de los pesos de una red neuronal, con el propósito de minimizar el error en sus predicciones, se denomina *aprendizaje*. Este procedimiento se inicia con un conjunto de datos de entrada a los que se expone la red, y a partir de las predicciones que obtiene de estas entradas se genera una retroalimentación que pueden clasificarse, en general, en tres tipos:

- **Aprendizaje supervisado**. Cada valor de entrada se acompaña de la correspondiente salida conocida (que usualmente se denomina *etiqueta*), y la red aprende una función que mapea

la entrada con la salida. Este será el caso que abordaremos a continuación, en la que cada entrada es una imagen y la correspondiente salida es un dígito decimal.

- **Aprendizaje no supervisado**. En este caso, la red aprende a detectar patrones en las entradas sin la especificación de la retroalimentación. Uno de los ejemplos más usuales de este caso el el agrupamiento o *clustering*.
- **Aprendizaje por refuerzo**. Aquí se utiliza un sistema de refuerzo, por lo general un premio y castigo. Esta forma de aprendizaje es la típica de redes que aprenden a jugar, obteniendo un premio al ganar el juego o un castigo al perderlo. De este modo, la red aprende de qué forma puede actuar para ganar más premios en el futuro.

Dado que este aprendizaje representa formalmente un proceso de optimización, podría usarse en principio cualquier algoritmo con ese propósito (por ejemplo, un algoritmo genético [**gupta1999**, **mirjalili2019**, **lanham2023**]). Sin embargo, en la práctica el método predominante para el entrenamiento de redes neuronales es el del descenso del gradiente (o alguna variante). En particular, examinaremos el descenso estocástico del gradiente (SGD por su sigla en inglés), que es un método iterativo que permite optimizar una función suficientemente suave (como las que mencionamos como funciones de activación y transformaciones lineales) por medio de una aproximación en la que se reemplaza el gradiente de la función calculado sobre el conjunto completo de datos por una estimación obtenida a partir de un subconjunto elegido al azar de dichos datos. Esta estrategia de optimización permite abordar problemas de alta dimensión (como el conjunto $\boldsymbol{\omega}$ de parámetros de una red neuronal) disminuyendo el costo computacional, pagando el precio de una menor velocidad de convergencia. Una forma muy utilizada de implementar el método SGD es a través del algoritmo de propagación hacia atrás, o *backpropagation*, que describiremos a continuación.

Formalmente, el entrenamiento para el caso del aprendizaje supervisado se puede proponer del siguiente modo: dado el conjunto de entrenamiento con N muestras

$$X = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2), \cdots, (\boldsymbol{x}_n, \boldsymbol{y}_N)\}$$

donde cada par fue generado por una función desconocida $\boldsymbol{y} = f(\boldsymbol{x})$, determinar una función $\hat{f}(X, \boldsymbol{\omega})$ que aproxime a f. Naturalmente, tanto $\boldsymbol{x}_i$ como $\boldsymbol{y}_i$ son usualmente cantidades vectoriales. Con este fin, se define una *función costo*[5], C, que resulta de alguna medida de la diferencia entre los valores $\hat{\boldsymbol{y}}_i = (y_1^L, y_2^L, \ldots, y_{m_L}^L)$ que predice la red para una determinada entrada $\boldsymbol{x}_i$ y parámetros de pesos y sesgos $\boldsymbol{\omega}$. Aquí, L es el número de capas de la red (L es el índice de la capa de salida) y m_L es la cantidad de nodos que tiene la capa L. Claramente, la idea es que durante el aprendizaje de la red, los valores de esta función disminuyan progresivamente, dado que idealmente si las predicciones coinciden con los valores reales, la "distancia" entre ellas será nula. Según el problema que abordamos con la red (regresión, clasificación, salidas discretas o continuas), pueden elegirse distintas opciones para C, por ejemplo el valor absoluto de la diferencia entre el valor predicho y el valor verdadero $\left\|\boldsymbol{y} - \hat{\boldsymbol{y}}\right\|$ (conocida usualmente como la norma L^1), el error cuadrático $\left\|\boldsymbol{y} - \hat{\boldsymbol{y}}\right\|^2$, o la función de costo $0/1$ que devuelve un valor 1 si la predicción es incorrecta y es útil para salidas con valores discretos 0 si $y = \hat{y},$ si no 1.

La función $\hat{f}$ que mencionamos anteriormente depende los valores de las entradas del conjunto de entrenamiento, x, y también de los valores de los pesos y sesgos $\boldsymbol{\omega}$. No obstante, durante el aprendizaje los valores tanto de las entradas como las etiquetas están fijos, por lo que solo tenemos la posibilidad de minimizar la función costo a través de $\boldsymbol{\omega}$. En lo siguiente, adoptaremos como C la función error cuadrático medio[6]:

$$C(X, \boldsymbol{\omega}) = \frac{1}{2n} \sum_{i=1}^{N} \left\|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i\right\|^2 \tag{1.1}$$

[5]Es usual referirse a la función costo también como función de pérdida (*loss*) o función objetivo.

[6]En el ejemplo de aplicación utilizaremos como función costo la función de entropía cruzada o *crossentropy*.

La suma se realiza sobre cada caso individual del conjunto de entrenamiento y $\|\cdot\|$ es la norma L^2 o euclídea. Esta elección de la función costo según la expresión (1.1) cumple con una condición necesaria para poder utilizar el algoritmo de *backpropagation*, esto es, que C debe ser *aditiva*, ya que de este modo podemos escribir la función costo como un promedio de funciones costo para muestras de entrenamiento individuales $C(\boldsymbol{x}_i)$, lo que nos permitirá calcular las derivadas parciales, y obtener luego el gradiente $\partial C/\partial \boldsymbol{\omega}$ a partir de promedios sobre las muestras de entrenamiento.

Si bien esta función depende de los valores de $\boldsymbol{y}_i$, recordemos que estos valores están fijos durante el entrenamiento, al igual que los valores de las entradas correspondientes $\boldsymbol{x}_i$, por lo que estrictamente los valores de C quedan determinados por la elección de los pesos $\boldsymbol{\omega}$. De este modo, minimizar la función consiste en utilizar el descenso del gradiente para que iterativamente generemos una sucesión de valores de $\boldsymbol{\omega}$ que disminuyan el valor de C:

$$\boldsymbol{\omega}^{t+1} = \boldsymbol{\omega}^t - \eta \frac{\partial C(X, \boldsymbol{\omega}^t)}{\partial \boldsymbol{\omega}} \tag{1.2}$$

donde $\boldsymbol{\omega}^t$ denota los parámetros de la red neuronal en la iteración t del descenso del gradiente. η se denomina *tasa de aprendizaje* y determina el tamaño del paso que se da en la dirección del descenso del gradiente y por lo tanto condiciona la forma en que converge la minimización. El valor de η debe realizarse cuidadosamente para evitar inestabilidades en el proceso de aprendizaje, y usualmente está definido en el rango $0 < \eta < 1$.

Dado que la función costo se puede descomponer en una suma sobre términos de costo o error para cada para individual de entrada salida $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, la derivada se puede calcular para cada par entrada/salida individualmente y luego combinar estos términos al final, ya que la derivada de una suma de funciones es la suma de las derivadas de cada función:

$$\frac{\partial C(X, \boldsymbol{\omega})}{\partial \omega_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial}{\partial \omega_{ij}^k} \left(\frac{1}{2} (\hat{\boldsymbol{y}}_d - \boldsymbol{y}_d)^2 \right) = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial C_d}{\partial \omega_{ij}^k}$$

donde ω_{ij}^k el el peso (o sesgo) para el nodo j de la capa l_k del nodo de entrada i en la capa l_{k-1}. De este modo, con el propósito de derivar el algortimo de *backpropagation* vamos a considerar solo un par de entrada/salida. Luego de esta derivación, la forma general para todos los pares de entrada/salida del conjunto X puede ser generada combinando los gradientes individuales. Así, la función costo que utilizaremos es:

$$C = \frac{1}{2} \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|^2$$

donde omitimos el subíndice d en C_d, $\hat{\boldsymbol{y}}_d$ y $\boldsymbol{y}$ por simplicidad.

La derivación del algoritmo de *backpropagation* comienza aplicando la regla de la cadena a la derivada parcial de la función costo

$$\frac{\partial C}{\partial \omega_{ij}^k} = \frac{\partial C}{\partial z_j^k} \frac{\partial z_j^k}{\partial \omega_{ij}^k}$$

donde z_j^k es el valor de activación del nodo j de la capa k antes de pasar por la función de activación para generar la salida y_j^k. Esta descomposición de la derivada parcial establece que el cambio en la función costo debido a un peso es el producto del cambio de la función C debido a la activación z_j^k multiplicado por el cambio en la activación z_j^k debido al peso ω_{ij}^k. Generalmente se llama *error* al primer factor de la expresión anterior, y se denota por

$$\delta_j^k \equiv \frac{\partial C}{\partial z_j^k}$$

El segundo factor se puede calcular a partir de la definición del valor de activación:

$$\frac{\partial z_j^k}{\partial \omega_{ij}^k} = \frac{\partial}{\partial \omega_{ij}^k}\left(\sum_{l=0}^{m_{k-1}} \omega_{lj}^k y_l^{k-1}\right) = y_i^{k-1}$$

siendo m_{k-1} la cantidad de nodos en la capa $k-1$. Entonces, la derivada parcial de la función costo C con respecto de un peso ω_{ij}^k es

$$\frac{\partial C}{\partial \omega_{ij}^k} = \delta_j^k y_i^{k-1}$$

Se puede ver entonces que la derivada parcial respecto a un peso es el producto del término de error δ_j^k del nodo j en la capa k, y de la salida y_i^{k-1} del nodo i de la capa $k-1$. Esto resulta bastante intuitivo porque ω_{ij}^k conecta la salida del nodo i de la capa $k-1$ con la entrada del nodo j de la capa k en el grafo computacional.

Comenzamos con el cálculo de errores de la última capa (L), que pueden ser evaluados directamente a partir de la "distancia" entre los valores predichos por la red y los correspondientes del conjunto de entrenamiento X. Para el nodo j en la capa de salida cuya función de activación es $g_o(z_j^L)$, tenemos:

$$C_j = \frac{1}{2}(\hat{y}_j - y_j)^2 = \frac{1}{2}\left(g_o(z_j^L) - y_j\right)^2$$

Aplicando la derivada parcial y usando la regla de la cadena resulta:

$$\delta_j^L = (g_o(z_j^L) - y_j) g_o'(z_j^L) = (\hat{y}_j - y_j) g_o'(z_j^L)$$

donde como es usual, g' es la derivada de g. La derivada parcial de C con respecto a un peso de la capa final es, entonces,

$$\frac{\partial C_j}{\partial \omega_{ij}^L} = \delta_j^L z_i^{L-1} = (\hat{y}_j - y_j) g_o'(z_i^L) y_i^{L-1}$$

Ahora podemos empezar a propagar hacia atrás los errores de la última capa, y para ello volvemos a utilizar la regla de la cadena para la derivada de funciones multivariadas. El término del error δ_k^k en la capa $1 \leq k \leq L$ es

$$\delta_j^k = \sum_{l=1}^{m_{k+1}} \frac{\partial C}{\partial z_l^{k+1}} \frac{\partial z_l^{k+1}}{\partial z_j^k}$$

donde l va desde 1 hasta m_{k+1} (el número de nodos de la capa siguiente). Notar que debido a que la entrada de sesgo y_0^k correspondiente a ω_{0j}^{k+1} es fijo, su valor no depende de las salidas de las capas previas, y por lo tanto l no toma el valor cero. Insertando el término de error δ_l^{k+1} obtenemos la siguiente ecuación:

$$\delta_j^k = \sum_{l=1}^{m_{k+1}} \delta_l^{k+1} \frac{\partial z_l^{k+1}}{\partial z_j^k}$$

A partir de la definición del valor de activación:

$$z_l^{k+1} = \sum_{j=1}^{m_k} \omega_{jl}^{k+1} g(z_j^k)$$

donde $g(\cdot)$ es la función de activación de las capas ocultas, tenemos

$$\frac{\partial z_l^{k+1}}{\partial z_j^k} = \omega_{jl}^{k+1} g'(z_j^k)$$

Reemplazando esta expresión en la ecuación previa obtenida para δ_j^k, resulta la derivada parcial de la función costo con respecto a un peso en las capas ocultas ω_{ij}^k para $1 \leq k < L$:

$$\frac{\partial C}{\partial \omega_{ij}^k} = \delta_j^k y_i^{k-1} = g'(z_j^k) y_i^{k-1} \sum_{l=1}^{m_{k+1}} \omega_{jl}^{k+1} \delta_l^{k+1}$$

Esta ecuación es el motivo del nombre del algoritmo de aprendizaje. El error δ_j^k en la capa k depende de los errores δ_l^{k+1} de la capa siguiente $k+1$. De este modo, los errores se propagan hacia atras desde la última capa hacia la primera. El algoritmo comienza entonces calculando los errores de la última capa a partir de la salida predicha por la red $\hat{y}_j = g_o(z_j^L)$ y del valor de entrenamiento y_j. A partir de aquí, los términos de error de la capa previa son calculados realizando una suma de los errores recién calculados ponderada por los pesos ω_{jl}^{k+1} de los términos de error de la capa siguiente y escalados por $g'(z_j^k)$, y esto se repite hasta que se alcanza la primera capa.

Esta propagación hacia atrás de los errores es muy similar a la propagación hacia adelante que calcula la predicción de la red. Por esto, el cálculo de la salida de la red se suele llamar la *fase de propagación hacia adelante*, mientras que el cálculo de los errores y derivadas se denomina usualmente *fase de propagación hacia atrás*, o *backpropagation*. Durante la fase de avance, las entradas se combinan repetidamente desde la primer capa hasta la última por medio de las sumas ponderadas por los pesos ω_{ij}^k y las funciones de activación $g(x)$ y $g_o(x)$. En la fase de retroceso, las "entradas" son los términos de error de la capa final, que son repetidamente recombinados desde la última capa hacia la primera mediante sumas ponderadas por los pesos ω_{ij}^{k+1} y transformados por los factores de escala no lineales $g_o'(z_j^L)$ y $g'(z_j^k)$.

Con las expresiones obtenidas para el cálculo del gradiente de C con respecto de $\boldsymbol{\omega}$, podemos realizar la secuencia de actualizaciones de los pesos dada por la ecuación (1.2). Esta actualización no se realiza calculando el gradiente completo sobre todo el conjunto de entrenamiento X simultáneamente, sino que se realiza una estimación del gradiente utilizando un subconjunto de X elegido aleatoriamente, en lo que se denomina un *minibatch*. El uso de *minibatchs* otorga eficiencia al método, dado que permiten un entrenamiento más rápido al calcular el gradiente y actualizar los pesos solo con una parte de los datos. Además, al promediar los gradientes de múltiples ejemplos, reduce el ruido y mejora la estabilidad del proceso de aprendizaje. No obstante, este método puede afectar a la precisión final de la red, ya que la actualización de los pesos se basa en una aproximación del gradiente total. El tamaño de la muestra que compone el minibatch es un parámetro importante que se debe ajustar para obtener un buen rendimiento.

Una vez que el conjunto X se particiona en subconjuntos aleatorios de muestras en *minibatch*, se llama una *era* a la secuencia de actualizaciones de los pesos en cada *minibatch*, hasta que se recorre el conjunto completo de aprendizaje. Luego, se iteran sucesivas eras hasta que la función costo alcanza un valor suficientemente bajo. En el ejemplo siguiente analizaremos una implementación práctica del algoritmo de *backpropagation* para el entrenamiento de una red *feedforward*.

1.3. Ejemplo de red usando NumPy

Vamos a poner en código los conceptos desarrollados para la red de propagación hacia adelante, entrenando una red para que pueda reconocer dígitos escritos a mano. Utilizaremos para

FIGURA 1.5: Ejemplo de imágenes del conjunto MNIST. Fuente: Wikimedia Commons.

ello el conjunto de datos MNIST[7] que consiste en una colección de 60 000 imágenes de entrenamiento y 10 000 imágenes de prueba. Cada imagen tiene una resolución de 28×28 píxeles en valores que representan una escala de grises en el rango $[0, 255]$. Una muestra de las imágenes que componen el conjunto se puede ver en la Figura 1.5.

Como es usual, en las primeras líneas importamos los módulos necesarios para crear clases abstractas (`abc`); para intepretar los argumentos de la línea de comandos (`argparse`); para serializar y guardar objetos (`pickle`); para realizar conteos de frecuencia (`Counter`); los usuales para graficar (`matplotlib`) y para operaciones numéricas (`numpy`); y finalmente para visualizar una barra de progres (`trange`). A continuación establecemos el valor de la semilla del generador de números aleatorios de modo de poder reproducir los resultados.[8]

```
3  import abc
4  import argparse
5  import pickle
6  from collections import Counter
7
8  import matplotlib.pyplot as plt
9  import numpy as np
10 from tqdm import trange
11
12 # predictibilidad durante el desarrollo
13 np.random.seed(216091)
```

Comenzamos definiendo la función `load_dataset`, que como su nombre indica será la responsable de leer los datos desde un archivo csv (ver subsección ??). Este archivo contiene en cada fila una imagen, la primer columna de la fila es el dígito que representa dicha imagen (valor entre 0 y 9), y las restantes 784 columnas contiene el valor en escala de gris de cada píxel. Esta función devuelve dos arrays de `(N, 1)` y `(N, 784)` en `Y_train` y `X_train`, respectivamente, siendo `N` la cantidad de datos que contiene el *dataset*. El conjunto de entrada (`X_train`) se normaliza antes de devolver ambos arrays.

```
16 def load_dataset(datasource_path):
17     """Carga los datos de trabajo."""
```

[7] Ver entrada en Wikipedia. El conjunto MNIST se puede descargar desde acá.

[8] Usamos para esto el trigésimo primer número perfecto.

```
    # Lectura de datos
    train_data = np.genfromtxt(datasource_path, delimiter=',', skip_header=1, dtype=float)
    Y_train = train_data[:, 0].astype(int)
    X_train = train_data[:, 1:]

    # Normalizamos X
    X_train = X_train / 255

    return X_train, Y_train
```

La estructura de nuestra red consiste en una secuencia de capas que contienen a los nodos separados en dos unidades de cálculo: la que calcula los valores de activación z_i y la que que representa las correspondientes funciones de activación. En este contexto, definimos primero una clase abstracta `Layer` que define los métodos que luego cada clase derivada debe implementar: `forward`, que realiza un paso de propagación hacia adelante, y `backward` que utilizaremos durante la fase de entrenamiento para implementar el algoritmo de *backpropagation*:

```
class Layer(abc.ABC):
    """Capa abstracta.

    Cada capa puede realizar dos tareas:

      - Procesar la entrada para generar la salida: output = layer.forward(inputdata)

      - Implementar backpropagation: grad_input = layer.backward(inputdata, grad_output)

    Algunas capas tienen parámetros que se actualizan durante backpropagation (w, b).
    """

    @abc.abstractmethod
    def forward(self, inputdata):
        """Realiza un paso de propagación para adelante.

        Recibe una entrada de shape [batch, input_nodes], y devuelve la salida con
        shape [batch, output_nodes].
        """

    @abc.abstractmethod
    def backward(self, inputdata, grad_output):
        """Realiza un paso de backpropagation a través de la capa, respecto de la entrada.

        Si la capa tiene parámetros (capa densa), se actualiza en este paso.
        """
```

Para todas las capas ocultas utilizaremos ReLU como función de activación. Los nodos de esta capa no contienen información sobre los pesos de las conexiones sinápticas, ya que solamente toman como entrada el valor de activación calculado en la capa previa y transforma este valor según la función ReLU. Esto es lo que implementa el método `forward`. Para el cálculo de la propagación hacia atrás del error, esto simplemente es el error de la capa siguiente multiplicado por la derivada de la función ReLU (que es 1 si $x > 0$, y 0 para $x \leq 0$). Este es el cálculo que realiza el método `forward`.

```
class ReLU(Layer):
    """Capa ReLU que aplica una unidad lineal rectificada elemento por elemento."""

    def forward(self, inputdata):
        """Aplica ReLU elemento por elemento a la matriz [batch, input_nodes]."""
```

```
        relu_forward = np.maximum(0, inputdata)
        return relu_forward

    def backward(self, inputdata, grad_output):
        """Calcula el gradiente de la función de costo respecto de la entrada ReLU."""
        relu_grad = inputdata > 0
        return grad_output * relu_grad
```

La clase siguiente permite instanciar capas cuyos nodos están todos conectados con todos los nodos de la capa anterior, y por eso se denomina `Dense`. A diferencia de la capa `ReLU`, esta capa contiene los valores ω_{ij}^{l} (`self.W`) que representan los pesos de las conexiones entre los nodos i de la capa $l-1$ con los nodos j de la capa l. Por lo tanto esta capa contiene un método adicional (`__init__`) que permite inicializar estos valores (utilizamos para ello el método de inicialización de Xavier[**glorot2010**], que asigna valores en un rango razonable de valores obtenidos de una distribución normal con media cero y varianza que depende de la cantidad de conexiones de entrada y salida). Además, en esta capa separamos explícitamente el valor de los sesgos o *bias* (`self.bias`). El método `forward` realiza el cálculo del valor de activación por medio del producto matricial del array con los valores de entrada y los pesos, sumando los valores de los sesgos. Por su parte, el método `backward` no solo calcula el gradiente de la función costo respecto de los pesos, sino que también implementa la actualización de los mismos con el método del descenso del gradiente incorporando el correspondiente parámetro η (`self.learning_rate`).

```
class Dense(Layer):
    """Capa densa; la salida es W * x + b = z."""

    def __init__(self, input_nodes, output_nodes, learning_rate=0.1):
        self.learning_rate = learning_rate
        self.W = np.random.normal(
            loc=0.0,
            scale=np.sqrt(2 / (input_nodes + output_nodes)),
            size=(input_nodes, output_nodes))
        self.bias = np.zeros(output_nodes)

    def forward(self, inputdata):
        """Calcula el producto matricial entre la entrada y los pesos, más los bias."""
        return np.dot(inputdata, self.W) + self.bias

    def backward(self, inputdata, grad_output):
        """Calcula df / dx = df / d[capa] d[capa]/dx donde d[capa] / dx = W.T."""
        grad_input = np.dot(grad_output, self.W.T)
        # cálculo del gradiente respecto de W y b
        grad_W = np.dot(inputdata.T, grad_output)
        grad_b = grad_output.mean(axis=0) * inputdata.shape[0]
        assert grad_W.shape == self.W.shape and grad_b.shape == self.bias.shape
        # Actualización de W y b por medio del descenso del gradiente.
        self.W = self.W - self.learning_rate * grad_W
        self.bias = self.bias - self.learning_rate * grad_b
        return grad_input
```

En nuestro ejemplo, la última capa será una instancia de `Dense`, por lo que su salida consiste en los valores de activación que genera esta clase sin pasar por una función de activación. Llamaremos *logits* a estos valores (o vector logit como se lo utiliza habitualmente), que serán utilizados para predecir el valor numérico de la imagen de entrada como una medida de probabilidad. En este contexto utilizaremos una función de activación softmax para la última capa junto con una función costo del tipo entropía cruzada, o *crossentropy*, que se usa frecuentemente para estimar la diferencia entre dos distribuciones de probabilidad y se basa en el concepto de entropía de la información propuesta por Shannon [**shannon1948**].

Llamando l_i a la componente i-ésima del vector logit, y p_i al valor correspondiente luego de pasar por la función softmax, la función costo es:

$$C(\boldsymbol{y}, \boldsymbol{l}) = -\sum_i y_i \left[l_i + \log \left(\sum_j e^{l_j} \right) \right]$$

Esta expresión se simplifica si codificamos las etiquetas (digitos entre cero y nueve) en forma de un vector binario que tiene un 1 en la posición que corresponde a cada dígito (conocido también como *one-hot encoding*, o OHE). Por ejemplo, si el valor de la etiqueta es 4, la codificamos en forma OHE como el vector (`0, 0, 0, 0, 1, 0, 0, 0, 0, 0`), es decir, $y_l = 1$ para $l = 4$ y cero en otro caso. De este modo, combinamos el cálculo de la función softmax sobre el vector logit de la última capa con la función de pérdida de *crossentropy*, y obtenemos para cada nodo j de la última capa:

$$C_j = -l_j + \log \left(\sum_i e^{l_i} \right)$$

La función `softmax_crossentropy_with_logits` implementa este cálculo, devolviendo un array (`xentropy`) que contiene los valores de C_j para cada nodo j de la última capa. El correspondiente cálculo del gradiente de esta función costo se realiza en la función `grad_softmax_crossentropy_with_logits`:

```
99  def softmax_crossentropy_with_logits(logits, reference_labels):
100     """Cálculo de crossentropy.
101
102     Se realiza a partir de logits[batch, n_clases] y las etiquetas de los datos.
103     """
104     logits_for_labels = logits[np.arange(len(logits)), reference_labels]
105     xentropy = - logits_for_labels + np.log(np.sum(np.exp(logits), axis=-1))
106     return xentropy
107
108
109 def grad_softmax_crossentropy_with_logits(logits, reference_labels):
110     """Cálculo del gradiente de crossentropy.
111
112     Se realiza a partir de logits[batch, n_clases] y las etiquetas de los datos.
113     """
114     ones_for_labels = np.zeros_like(logits)
115     ones_for_labels[np.arange(len(logits)), reference_labels] = 1
116     softmax = np.exp(logits) / np.exp(logits).sum(axis=-1, keepdims=True)
117     return (- ones_for_labels + softmax) / logits.shape[0]
```

Con todo lo anterior ya tenemos lo necesario para crear una red. Esto lo hacemos a partir de la clase `Network`, que básicamente consiste en una secuencia (en forma de lista) sobre la que actuarán los correspondientes métodos `_forward`, para propagar los cálculos hacia adelante a partir de los datos de entrada X, `predict` para obtener la predicción de la red ante una entrada determinada, el método `train` para entrenar la red a partir de pares (X, Y), implementando *backpropagation*, y dos métodos más: `dump` para guardar los parámetros de una red entrenada y `load` para recuperar dichos parámetros desde un archivo en disco y crear una red operativa para realizar predicciones.

Declaramos la clase `Network` junto con el método `__init__`, que recibe una lista (`layers`) que contiene objetos instanciados a partir de las clases `ReLU` y `Dense`. A continuación, el método `_forward` calcula los valores de activación de cada capa, secuencialmente a partir de la capa de entrada, y devuelve estos valores en el array `activations`:

```
120 class Network:
121     """Opera sobre una secuencia de capas."""
122
123     def __init__(self, layers):
124         self.layers = layers
125
126     def _forward(self, inputdata):
127         """Calcula las activaciones de todas las capas, secuencialmente.
128
129         La salida de una capa es la entrada de la próxima. Devuelve una lista con
130         las activaciones de cada capa.
131         """
132         activations = []
133         for layer in self.layers:
134             inputdata = layer.forward(inputdata)
135             activations.append(inputdata)
136
137         return activations
```

El método `predict` recibe una entrada X y propaga los cálculos hacia adelante en la red hasta la última capa, donde se calcula el vector logit. La predicción de la red resulta de la posición de este vector que tiene el máximo valor (recordar que usamos OHE para codificar los dígitos, por lo que si el valor máximo del vector logit ocurre en la sexta posición, la predicción es el dígito 5). No hace falta aplicar una función de activación sobre los logits, como softmax, dado que por ser una función estrictamente no negativa, solo realiza un cambio de escala pero no altera la posición del máximo:

```
139     def predict(self, X):
140         """Calcula la predicción de la red.
141
142         Devuelve el índice del mayor valor de probabilidad en logits.
143         """
144         layer_activations = self._forward(X)
145         logits = layer_activations[-1]
146         return logits.argmax(axis=-1)
```

El método `train` realiza secuencialmente los pasos que detallamos del algoritmo *backpropagation*. Recibe como argumentos una entrada X y su correspondiente etiqueta (el valor del dígito al que corresponde la imagen) y. En primer lugar realiza el avance hacia adelante en la red de los cálculos que llevan desde la entrada a las activaciones de la última capa (vector logits). Luego de esto le agregamos al principio el propio array X para tenerlo como entrada de la primera capa. A continuación obtenemos el array `loss`, que corresponde al valor del costo de cada nodo de la capa de salida[9], así como su gradiente, `loss_grad`, de la última capa, y luego mediante un bucle **`for`** recorremos en forma inversa la lista que contiene cada capa de la red, invocando el método `backward` de cada capa que propaga hacia atrás los gradientes y actualiza los pesos y sesgos en las capas del tipo `Dense` (las capas `ReLU` no contienen parámetros para optimizar, ya que simplemente implementan la función no lineal de activación). Este método devuelve el valor medio de las funciones de costo de todos los nodos de la última capa.

```
148     def train(self, X, y):
149         """Entrena a la red con X e y."""
150         # Etapa de avance de activaciones (forward)
151         layer_activations = self._forward(X)
```

[9]Recordemos que a esta función también es habitual llamarla función objetivo o función de pérdida (*loss*).

VERSIÓN PRELIMINAR

```
152
153         # Agregamos al principio la entrada original para tener la entrada de cada capa
154         # y separamos las últimas activaciones como logits
155         *layer_inputs, logits = [X] + layer_activations
156
157         # Cálculo de la función de pérdida (indicativo de lo bien entrenado)
158         # para devolver al final
159         loss = softmax_crossentropy_with_logits(logits, y)
160
161         # Comenzamos la propagación de gradientes para atrás en la red (backpropagation)
162         # por la última capa especial de entropía cruzada
163         loss_grad = grad_softmax_crossentropy_with_logits(logits, y)
164
165         # Seguimos la propagación para atrás por cada capa
166         for layer, originalinput in reversed(list(zip(self.layers, layer_inputs))):
167             loss_grad = layer.backward(originalinput, loss_grad)
168
169         return np.mean(loss)
```

Por último, agregamos los métodos `dump` y `load`. El primero recibe como argumento una cadena que será el nombre del archivo donde se guarda la instancia de la clase `Network` que contiene todos los parámetros luego del entrenamiento, de forma de poder recuperarlos después con el método `load` y poder utilizar la red sin tener la necesidad de pasar nuevamente por el costoso proceso de entrenamiento:

```
171     def dump(self, filepath):
172         """Guarda la red actual en un archivo."""
173         with open(filepath, "wb") as fh:
174             pickle.dump(self, fh)
175
176     @classmethod
177     def load(cls, filepath):
178         """Recupera de disco una instancia de la red."""
179         with open(filepath, "rb") as fh:
180             return pickle.load(fh)
```

Usamos el decorador `@classmethod` en `load` de forma de poder instanciar objetos de la clase `Network` directamente a partir de la estructura de la red y los valores de los parámetros almacenados en `filepath` (ver la sección **??**).

El método definido a continuación, `get_minibatches`, implementa la división del conjunto completo de entrenamiento en pequeñas tandas que permiten acelerar el cálculo, a partir de estimar el gradiente de la función de costo con muestras más pequeñas del conjunto completo (X, Y). Recibe como argumentos las entradas X (`inputs`), las etiquetas Y (`tagets`), el tamaño de la tanda de datos a utilizar en cada etapa de entrenamiento, y una variable booleana `shuffle` que si tiene el valor `True`, realiza una permutación aleatoria de los índices de ambos conjuntos de modo de acceder aleatoriamente a estos datos. Esta función está implementada a modo de generador, que va produciendo estos los índices a medida que se van requiriendo:

```
183 def get_minibatches(inputs, targets, batchsize, shuffle=False):
184     assert len(inputs) == len(targets)
185     if shuffle:
186         indices = np.random.permutation(len(inputs))
187     for start_idx in trange(0, len(inputs) - batchsize + 1, batchsize):
188         if shuffle:
189             excerpt = indices[start_idx:start_idx + batchsize]
190         else:
```


VERSIÓN PRELIMINAR

```
            excerpt = slice(start_idx, start_idx + batchsize)
        yield inputs[excerpt], targets[excerpt]
```

La función `train` es la que realiza el entrenamiento de la red y muestra el grado de precisión que obtiene a medida que progresa. Comienza cargando los datos de las entradas y las etiquetas desde un archivo, y separa estos datos en un conjunto que se utilizará para el entrenamiento (`X_train` y `Y_train`) de otros que se utilizarán como validación (`X_val`, `y_val`), que contienen solo un sexto del total de los datos. Esta validación será útil para cuantificar el desempeño de la red prediciendo a partir de datos que no fueron utilizados durante la etapa de entrenamiento:

```
def train(datasource_path):
    # Cargamos los datos y reservamos los últimos ejemplos de entrenamiento para validación
    X_train, Y_train = load_dataset(datasource_path)
    limit = len(X_train) // 6
    X_train, X_val = X_train[:-limit], X_train[-limit:]
    Y_train, y_val = Y_train[:-limit], Y_train[-limit:]
```

Luego instanciamos un objeto de la clase `Network`, como una secuencia de una capa `Dense` cuya entrada tiene la misma dimensión que los datos de entrada X (en este caso, un array de $28 \times 28 = 784$ elementos) conectados a 100 nodos, la segunda capa es la capa ReLU que recibe la salida de las activaciones de la capa anterior y calcula las correspondientes funciones de activación, luego otra capa `Dense`, ahora con 200 nodos, su correspondiente capa ReLU, y la capa final `Dense` con las 10 unidades que generan el vector logit:

```
    network = Network([
        Dense(X_train.shape[1], 100),
        ReLU(),
        Dense(100, 200),
        ReLU(),
        Dense(200, 10),
    ])
```

A continuación creamos dos listas vacías en las que iremos guardando los resultados de las precisiones obtenidas en cada época de entrenamiento (recordemos que una época consiste en recorrer todos los datos de entrenamiento mediante minibatches), y luego generamos el entrenamiento propiamente dicho en un número fijo de 25 épocas, informando el progreso de la función junto con las precisiones obtenidas en el conjunto de entrenamiento y el de validación, entendiendo estas precisiones como el número medio de aciertos de la red en ambos casos. Al finalizar el recorrido por las épocas, guardamos en un archivo la red entrenada y generamos un gráfico que muestra la evolución de las precisiones obtenidas en los conjuntos de entrenamiento y validación.

```
    train_log = []
    val_log = []

    for epoch in range(25):
        total_loss = 0
        for x_batch, y_batch in get_minibatches(X_train, Y_train, batchsize=32, shuffle=True):
            loss = network.train(x_batch, y_batch)
            total_loss += loss

        train_log.append(np.mean(network.predict(X_train) == Y_train))
        val_log.append(np.mean(network.predict(X_val) == y_val))
```

```
221
222         print(f"Época {epoch}")
223         print(f"Precisión de entrenamiento: {train_log[-1]:8.4f}")
224         print(f"   Precisión de validación: {val_log[-1]:8.4f}")
225         print(f"             Total pérdida: {total_loss:8.4f}")
226
227     # guardar la red con su estado actual entrenado
228     network.dump("trained.pkl")
229
230     plt.plot(train_log, label='Precisión de entrenamiento')
231     plt.plot(val_log, label='Precisión de validación')
232     plt.xlabel("Época")
233     plt.legend(loc='best')
234     plt.grid()
235     plt.show()
```

La función `show_images` tiene la utilidad de generar imágenes a partir de un conjunto de entrada, con el solo propósito de visualizar dichos datos. Es útil para comprender la tarea de clasificación que le encomendaremos a la red neuronal:

```
238 def show_images(datasource_path):
239     """Muestra un ejemplo de las imágenes usadas como datos fuente."""
240     X_train, Y_train = load_dataset(datasource_path)
241
242     # reshape the sequence of bits so each 784 chunk is 28x28 (square image)
243     X_train = X_train.reshape(-1, 28, 28)
244
245     plt.figure(figsize=[6, 6])
246     for i in range(4):
247         plt.subplot(2, 2, i + 1)
248         plt.title(f"Label: {Y_train[i]}")
249         plt.imshow(X_train[i].reshape([28, 28]), cmap='gray')
250     plt.show()
```

Para tener una estimación cuantitativa del grado de precisión de la red, la función `evaluate` recibe como argumento una cadena (`datasource_path`) que apunta a un archivo que contiene pares de entrada y etiquetas: `X_test` y `Y_test`, respectivamente. Luego instancia una red a partir de un archivo donde se ha guardado previamente una red entrenada y predice los valores correspondientes a las entradas. Finalmente muestra un histograma con los porcentajes de aciertos que ha obtenido, a partir de los valores reales de las etiquetas.

```
253 def evaluate(datasource_path):
254     """Evalúa una red entrenada sobre un conjunto de datos de prueba."""
255     print("Cargando red")
256     network = Network.load("trained.pkl")
257     print("Cargando dataset de prueba")
258     X_test, Y_test = load_dataset(datasource_path)
259     print("Prediciendo")
260     predicted = network.predict(X_test)
261     print("Listo")
262
263     cnt = Counter(zip(Y_test, predicted == Y_test))
264
265     digits = []
266     pred_perc = []
267     for digit in range(10):
268         digits.append(digit)
269         quant_ok = cnt[(digit, True)]
```


VERSIÓN PRELIMINAR

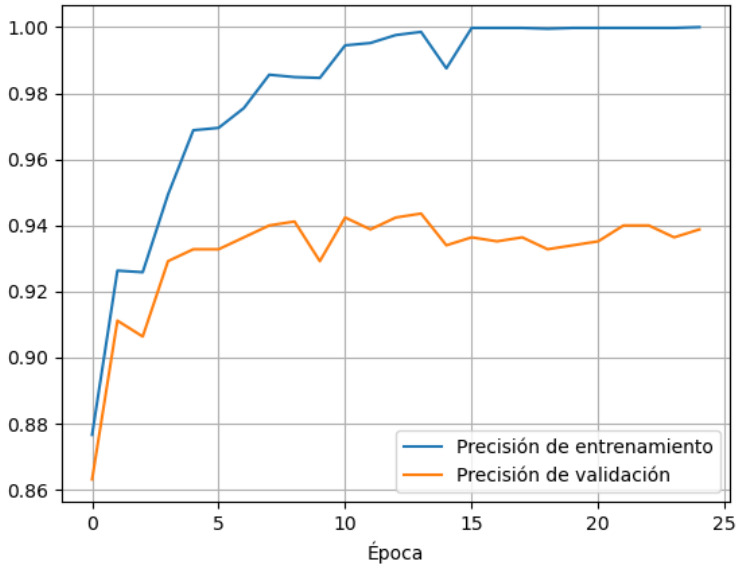


FIGURA 1.6: Evolución de la precisión en el entrenamiento utilizando el conjunto reducido de datos

```
270         quant_bad = cnt[(digit, False)]
271         pred_perc.append(100 * quant_ok / (quant_ok + quant_bad))
272 
273     fig, ax = plt.subplots()
274     ax.bar(digits, pred_perc, 0.7)
275     ax.set_title("Precisión de las predicciones")
276     ax.set_xticks(digits)
277     ax.set_ylim([90, 100])
278     ax.set_yticks(ticks=range(90, 101), labels=[f"{n}%" for n in range(90, 101)])
279     plt.show()
```

Por último, generamos un diccionario (`_actions`) que define diversas acciones que se pueden realizar con este código, en forma de pares función/datos. Las claves de este diccionario son las opciones de argumentos en linea de comando con los que se ejecutará el código. Se pueden realizar las siguientes opciones:

- `devtrain`: realiza un entrenamiento de la red con un conjunto reducido de datos (`mnist_train_dev.csv`), con el propósito de usarlo para pruebas de depuración del código.
- `train`: realiza el entrenamiento sobre el conjunto completo de datos de MNIST (`mnist_train.csv`).
- `show`: invoca a la función `show_images` sobre un conjunto reducido de datos para visualizar las imágenes que contiene.
- `evaluate`: ejecuta la función `evaluate` sobre el conjunto de prueba `mnist_test.csv` para cuantificar la precisión del entrenamiento realizado sobre la red.

Estas opciones se interpretan con el módulo `argparse`, y según el argumento pasado en la linea de órdenes ejecuta cada opción:

```
282 _actions = {
283     "devtrain": [train, "mnist_train_dev.csv"],
```

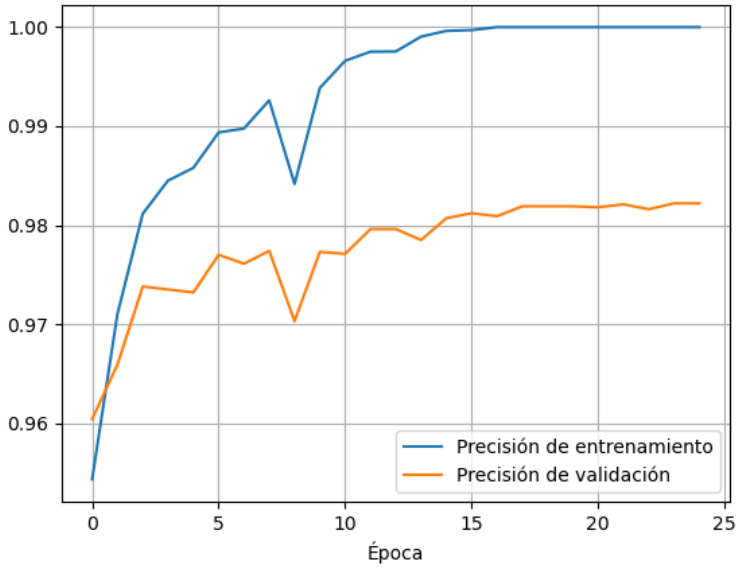


FIGURA 1.7: Evolución de la precisión en el entrenamiento utilizando el conjunto completo de datos MNIST

```
    "train": [train, "mnist_train.csv"],
    "show": [show_images, "mnist_train_dev.csv"],
    "eval": [evaluate, "mnist_test.csv"],
}
parser = argparse.ArgumentParser()
parser.add_argument("action", choices=_actions.keys(), help="Qué acción realizar")
args = parser.parse_args()
func, *params = _actions[args.action]
func(*params)
```

Entonces, para el caso de la opción `devtrain`, que contiene solo 5000 datos, obtenemos una precisión final de 1,0000 para los datos de entrenamiento y 0,9388 para los de validación. La evolución del entrenamiento se puede ver en la Figura 1.6.

Si ahora realizamos el entrenamiento sobre el conjunto completo de datos de MNIST, usando

```
./nn-mnist.py train
```

vemos que la precisión sobre el conjunto de validación mejora hasta el 0,9822, tal como muestra la Figura 1.7.

Por último, evaluaremos la precisión alcanzada por cada dígito del conjunto reducido de datos, con la opción `eval` en la línea de comandos. Tal como muestra la Figura 1.8, vemos que el peor desempeño sucede con la identificación de los dígitos 5 y 4, pero en todos los casos la tasa de aciertos es superior al 97 %.

1.4. Ejemplo usando Keras

En la sección anterior implementamos una red *feedforward* con herramientas básicas de Python (NumPy, Matplotlib) con la idea de desmenuzar las operaciones básicas involucradas en el entrenamiento y uso de la red. Afortunadamente existen bibliotecas que permiten abstraer los detalles de cálculo y facilitan la construcción de estructuras complejas con gran variedad de

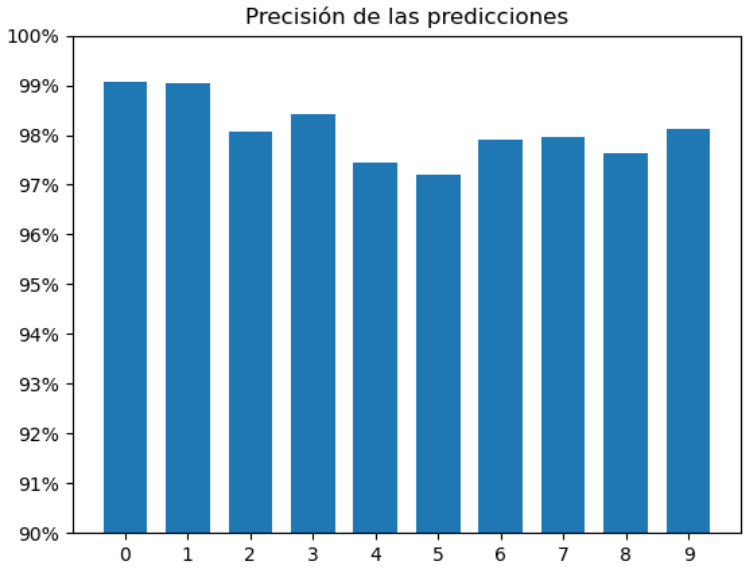


FIGURA 1.8: Porcentaje de aciertos de la red sobre el conjunto reducido de datos.

herramientas (distintos nodos de cálculo, funciones de activación, métodos de optimización, etc.), y también la utilización de *hardware* específico para estas aplicaciones (GPUs, TPUs). Estas bibliotecas mantienen un desarrollo muy activo en los últimos años, acompañanado al explosivo crecimiento de las aplicaciones de las técnicas de las redes neuronales de la actualidad.

A continuación abordaremos la identificación de las imágenes del conjunto MNIST por medio de la biblioteca Keras[10], que a su vez se ejecuta sobre la biblioteca TensorFlow[11], en este caso usando un *notebook* de Jupyter.

Como es usual, comenzamos importando los módulos que nos permiten manipular arrays, realizar gráficos y acceder al conjunto MNIST:

CELL 01

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
np.random.seed(216091)
```

En la segunda celda vemos lo simple que resulta acceder a los datos del conjunto MNIST, y además contamos cuántas valores tienen los conjuntos de entrenamiento y de prueba para cada etiqueta:

[10] https://keras.io/.
[11] https://www.tensorflow.org/.

CELL 02

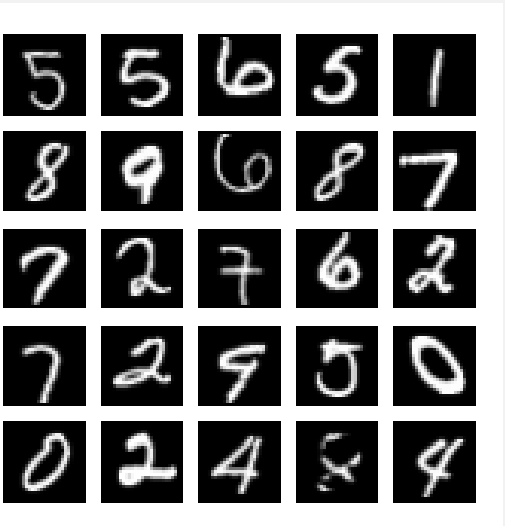
```
(x_train, y_train),(x_test, y_test) = mnist.load_data()
# Contamos el número de etiquetas de entrenamiento
unique, counts = np.unique(y_train, return_counts=True)
print("\nEtiquetas de entrenamiento: ", dict(zip(unique, counts)))
# Contamos el número de etiquetas de prueba
unique, counts = np.unique(y_test, return_counts=True)
print("\nEtiquetas de prueba: ", dict(zip(unique, counts)))
```

```
Etiquetas de entrenamiento:  {0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7:
    ↳ 6265, 8: 5851, 9: 5949}

Etiquetas de prueba:  {0: 980, 1: 1135, 2: 1032, 3: 1010, 4: 982, 5: 892, 6: 958, 7: 1028, 8: 974,
    ↳ 9: 1009}
```

Para ver cómo son las imágenes que la red tratará de identificar, elegimos 25 muestras aleatoriamente del conjunto de entrenamiento y las visualizamos:

CELL 03

```
# Visualizamos 25 muestras del conjunto de entrenamiento
indexes = np.random.randint(0, x_train.shape[0], size=25)
images = x_train[indexes]
labels = y_train[indexes]

plt.figure(figsize=(2,2))
for i in range(len(indexes)):
    plt.subplot(5, 5, i + 1)
    image = images[i]
    plt.imshow(image, cmap='gray')
    plt.axis('off')

plt.show()
plt.close('all')
```



Ahora importamos las herramientas de Keras que nos permite el armado de la red neuronal:

CELL 04

```
from keras.models import Sequential
from keras.layers import Input, Dense, Activation
from keras.utils import to_categorical, plot_model
```

Al igual que hicimos en la sección anterior, convertimos las etiquetas que tenemos en formato de dígito $(0, 1, \cdots, 9)$ en un array codificado en forma *one-hot encoding* (OHE):

CELL 05

```
# Convertimos a un array OHE
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

En las dos celdas siguientes obtenemos la dimensión del array que representa la imagen de entrada, y adaptamos los arrays de entrada de una imagen cuadrada de bits en el rango `[0, 255]` a un array unidimensional cuyos valores están normalizados:

CELL 06

```
# Calculamos la dimensión del array que representa la imagen de entrada
# (asumimos que es cuadrada)
image_size = x_train.shape[1]
input_size = image_size * image_size
input_size
```

```
784
```

CELL 07

```
# Cambiamos la forma de los arrays de entrada y normalizamos
x_train = np.reshape(x_train, [-1, input_size])
x_train = x_train.astype('float32') / 255
x_test = np.reshape(x_test, [-1, input_size])
x_test = x_test.astype('float32') / 255
```

A continuación construimos una red con la misma estructura que usamos en la sección anterior, es decir, una capa de entrada, dos capas densas con activación ReLU con 100 y 200 nodos, respectivamente, y una capa de salida de 10 nodos con activación softmax. Para ello instanciamos un objeto `model` de la clase `Sequential`:

CELL 08

```
# La red (modelo) consiste una capa de entrada, dos capas con activación ReLU
# y la capa de salida que contiene solo los valores de activación
model = Sequential()
model.add(Input(shape=(input_size,)))
model.add(Dense(100))
model.add(Activation('relu'))
model.add(Dense(200))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

Podemos invocar el método `summary()` para ver los detalles de la red, particularmente la cantidad de parámetros que se ajustarán en la fase de entrenamiento:

CELL 09

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	78,500
activation (Activation)	(None, 100)	0
dense_1 (Dense)	(None, 200)	20,200
activation_1 (Activation)	(None, 200)	0
dense_2 (Dense)	(None, 10)	2,010
activation_2 (Activation)	(None, 10)	0

```
 Total params: 100,710 (393.40 KB)

 Trainable params: 100,710 (393.40 KB)

 Non-trainable params: 0 (0.00 B)
```

Esta celda muestra que tenemos un total de 100710 parámetros que entrenar, que ocupan un total de `393.40` KB.

En la celda siguiente invocamos el método `compile` sobre el objeto `model`. Este método verifica que la arquitectura del modelo (red) sea válida y que todas las capas estén correctamente conectadas, y prepara el modelo para la fase de entrenamiento en una CPU, GPU o TPU. `compile` recibe tres argumentos: `loss`, que especifica cuál será la función de costo (o pérdida) a utilizar, en este caso entropía cruzada con valores categóricos; `optimizer`, que corresponde al método de optimización, y que para este ejemplo elegimos `adam` (*Adaptative Moments*) que es una variación del método del descenso estocástico del gradiente; y finalmente una métrica, `accuracy`, que es la fracción o porcentaje de predicciones correctas:

CELL 10

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Ya tenemos entonces todo listo para iniciar el entrenamiento:

CELL 11

```
model.fit(x_train, y_train, epochs=25, batch_size=128, verbose=0)
```

```
<keras.src.callbacks.history.History at 0x7f6d25a63b60>
```

El método `fit` recibe los arrays de entrada y etiquetas (`x_train` y `y_train`), la cantidad de épocas y el tamaño del batch. Adicionalmente pasamos el argumento `verbose=0` para evitar la salida extensa de esta celda, pero si pasamos el valor por defecto de este parámetro (omitiéndolo por ejemplo), el método mostrará el progreso del entrenamiento informando el tiempo que le toma cada época, la precisión obtenida y el valor de la función de costo, tal como se muestra en la Figura 1.9.

Una vez entrenada la red, podemos probar cuán eficaz es para predecir las imágenes del conjunto de prueba. Para ello ejecutamos la celda siguiente:

```
Epoch 22/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 4.9750e-07
Epoch 23/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 3.9767e-07
Epoch 24/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 3.2803e-07
Epoch 25/25
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 1.0000 - loss: 2.6730e-07
```

FIGURA 1.9: Últimas líneas de la salida de ejecución del método fit.

CELL 12

```
loss, acc = model.evaluate(x_test, y_test, batch_size=128)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

```
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 808us/step - accuracy: 0.9721 - loss: 0.1643

Test accuracy: 97.6%
```

Vemos que la tasa de aciertos alcanza el 98.1 %, muy similar a la precisión que alcanzamos con el modelo de la sección anterior. Una descripción más completa que muestra el desempeño de la red la da la matriz de confusión. Esta matriz representa la cantidad de aciertos para cada clase o etiqueta, así como los "desaciertos" al predecir etiquetas erróneas. En la celdas siguientes definimos una función que realiza el gràfico de la matriz de confusion (tomada de la web de scikit-learn, versión 0.18), y obtenemos los valores predichos y verdaderos transformándolos a dígitos desde su representación como arrays OHE:

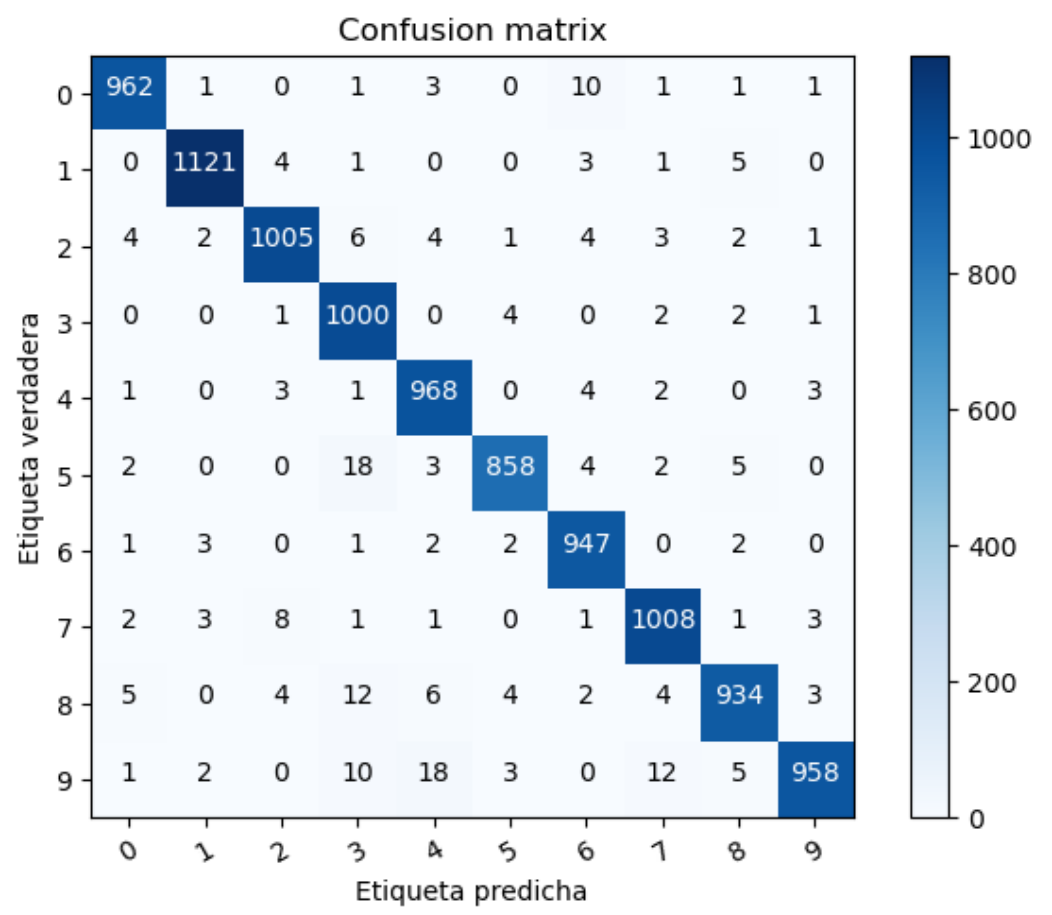
CELL 13

```
# Matriz de confusión
# Nota: este código está tomado directamente de la web de scikit-learn 0.18
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=30)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('Etiqueta verdadera')
    plt.xlabel('Etiqueta predicha')
```

CELL 14

```
from collections import Counter
from sklearn.metrics import confusion_matrix
import itertools

# Predict the values from the validation dataset
Y_pred = model.predict(x_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test, axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(10))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 445us/step
```



Los valores sobre la diagonal son los aciertos de la red, mientras que los que están fuera de la diagonal muestran cómo se distribuyen los fallos de la red sobre el resto de las etiquetas. Tal como ocurrió en la sección anterior, las imágenes que contienen al número 5 son las que más le cuesta identificar a la red. Si pasamos como argumento `normalize=True` a la función que construye la red, se mostrarán las fracciones de acierto en vez de los números absolutos.

En este ejemplo hemos visto solo las capacidades básicas de Keras, pero la biblioteca ofrece una amplia variedad de herramientas tales como capas `Dropout`, que es una técnica de regularización para reducir el sobreajuste de los datos, funciones de penalización, etc. Además, ofrece la posibilidad de construir otros tipos de redes como convolucionales, o recurrentes.

1.5. Lectura recomendadas

Actualmente se publican numerosos libros que abordan los múltiples aspectos de las redes neuronales. Recomendamos la lectura de los siguientes:

- **goodfellow2016**.
- **russel2022**.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [**zen-de-python**].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!