USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

FACULTADE DE MATEMÁTICAS

Traballo Fin de Grao

Aprendizaje profundo: una introducción para matemáticos

Fernández Martínez, Luis

2020/2021

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

Traballo Fin de Grao

Aprendizaje profundo: una introducción para matemáticos

Fernández Martínez, Luis

Julio, 2021

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Trabajo propuesto

Área de Coñecemento: Matemática Aplicada
Título: Aprendizaje profundo: una introducción para matemáticos
Breve descrición do contido
El Aprendizaje Profundo (Deep Learning en inglés -DL-) es el presente y futuro del procesamiento y uso de información. Se basa en el empleo de redes neuronales con múltiples capas. Sus variadas aplicaciones se sustentan en disciplinas como la Matemática Aplicada y Computacional, Álgebra Lineal, Optimización y Teoría de la Aproximación. En este trabajo se busca que el alumno entienda qué es una red neuronal, cómo se entrena, qué es el algoritmo de la propagación inversa y en qué consiste el método del gradiente estocástico. Se ilustrarán estos conceptos con códigos Matlab sencillos que permitan resolver algún problema de interés aplicativo. Algunos ejemplos son el reconocimiento de la escritura, el reconocimiento de patrones en imágenes o la creación de modelos matemáticos complejos. Si el alumno lo desea podrá crear códigos para más de un problema, así como profundizar en los conceptos en los que se sustenta el DL, siempre y cuando el tiempo y la dificultad lo permitan.
Recomendacións
El alumno deberá poseer conocimientos básicos de métodos numéricos para ecuaciones diferenciales ordinarias y optimización, además de un nivel de inglés suficiente para entender la bibliografía.

Índice general

Resumen

En los últimos años el aprendizaje profundo ha supuesto un cambio notable en el reconocimiento de patrones en una (sonido), dos (imágenes) y tres (vídeo) dimensiones. Además de las aplicaciones derivadas del uso de redes neuronales artificiales, resulta de interés académico y práctico conocer las ideas y formalismo matemático que se esconden detrás de ellas. Cálculo Numérico, Teoría de la Aproximación, Optimización y Álgebra Lineal son necesarios para desarrollar adecuadamente esta teoría. En este trabajo vamos a presentar dos de las redes actuales, el Perceptrón Multicapa, que llamaremos de forma genérica Red Neuronal Artificial y la Red Neuronal Convolucional. En ambos casos se pretende mostrar qué son formalmente y cómo se entrenan. Para ello haremos uso de ejemplos que faciliten la comprensión del desarrollo teórico. Presentaremos los métodos del gradiente estocástico y mini-batch y el algoritmo de la propagación inversa. Para ilustrar las ideas expuestas elaboraremos varios programas con los que generar ambos tipos de redes para clasificar imágenes de menor a mayor complejidad. Con ello concluiremos que si bien las redes neuronales son una buena herramienta para la tarea de clasificación, todavía quedan múltiples cuestiones sin resolver en lo que se refiere a su estructura, aprendizaje y aplicabilidad en otros campos.

Abstract

During the last years, deep learning has brought about a notable change in pattern recognition in one (sound), two (images) and three (video) dimensions. In addition to the applications derived from the use of artificial neural networks, it is of academic and practical interest to know the ideas and mathematical formalism that lie behind them. Numerical Calculus, Approximation Theory, Optimization and Linear Algebra are necessary to adequately develop this theory. In this work we are going to present two of the current networks, the Multilayer Perceptron, which we will generically call Artificial Neu-

ral Network and the Convolutional Neural Network. In both cases it is intended to show what they are formally and how they are trained. With this purpose in mind, we will use examples that help the understanding of the theoretical development. We will present the stochastic and mini-batch gradient methods and the backpropagation algorithm. To illustrate the ideas presented, we will develop several programs with which we will generate both types of networks to classify images from less to greater complexity. We will conclude that although neural networks are a good tool for the classification task, there are still many unresolved questions regarding their structure, learning and applicability in other fields.

Introducción

La Inteligencia Artificial (IA) es la rama de la Informática consagrada a la creación de máquinas capaces de realizar actividades propias de los seres humanos, como son identificar objetos en imágenes o extraer el mensaje contenido en una onda sonora. Dentro de ella se encuentra el Aprendizaje Automático o Machine Learning (ML), que busca que esas actividades sean aprendidas sin una programación explícita de las mismas [1]. Esto se consigue, por ejemplo, a través del uso de las llamadas *Redes Neuronales Artificiales*. Poco a poco su presencia se ha hecho notable en un amplio abanico de campos: coches con aparcamiento automático, asistentes de voz o detectores de spam son algunos ejemplos. Pese a las aplicaciones tan variadas que ofrecen, todas ellas tienen una historia en común.

El origen de las redes neuronales se remonta al primer intento del ser humano de entender cómo razona su propio cerebro. Con ese objetivo, W. McCulloch y W. Pitts publican en 1943 un artículo [2] que reflexiona acerca de cómo traducir el funcionamiento de las neuronas a un lenguaje basado en lógica proposicional. Otro paso importante se produce en 1958 gracias a la definición del *perceptrón* por parte de F. Rosenblatt [3], que cambia el lenguaje lógico por el numérico. Su funcionamiento es más intuitivo y las operaciones que contiene son muy similares a las empleadas en los *Perceptrones Multicapa*, que llamaremos Redes Neuronales Artificiales (RNA) y explicaremos en el primer capítulo de este trabajo: una o varias unidades, llamadas unidades lógicas umbral, TLU, (del inglés Threshold Logic Unit), reciben información en forma numérica que combinan linealmente. Si el número obtenido supera un cierto umbral, devuelve un valor. Si no lo supera, devuelve otro.

Presentar la estructura y funcionamiento básicos de las neuronas *biológicas* permitirá entender el porqué del nombre de las RNA [4]. Cada una de ellas consta de cuatro partes (figura 1): dendritas, cuerpo, axón y terminaciones sinápticas. Las dendritas son extensiones delgadas que comunican la neurona con otras anteriores. El cuerpo es el lugar donde se analiza la información recibida y se genera la respuesta. Esta viaja a través del axón en forma de impulso eléctrico hasta las terminaciones sinápticas, que comunican la célula con otras a través de sus respectivas dendritas. Si el impulso tiene un potencial suficien-

temente alto, se transmite la información. Por otro lado, la unión de muchas de ellas en cadena constituye una *red neuronal*. Si comparamos esta descripción con la del párrafo anterior, veremos que esencialmente se ha repetido en otros términos el funcionamiento del perceptrón, precursor de las RNA.

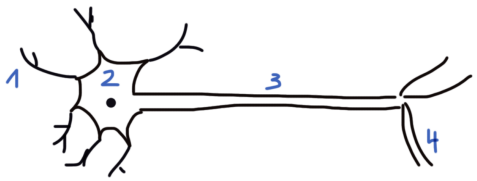

Figura 1: Esquema neurona: (1) dendritas, (2) cuerpo, (3) axón, (4) terminaciones sinápticas.

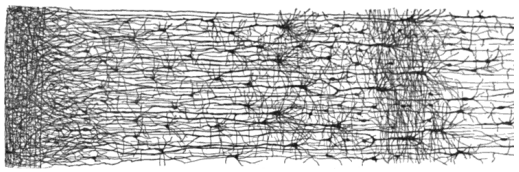

Figura 2: Disposición de las neuronas en una red neuronal biológica [5].

Con el objetivo de fijar las ideas tras las que se encuentra las estructura de las redes actuales, presentamos un ejemplo sencillo [6]. Supongamos el siguiente problema: un individuo desea saber si la asignatura que está a punto de coger le va a ser útil en su futuro. Para averiguarlo, decide establecer un sistema de puntuación basado en cuatro criterios:

- Si la asignatura tiene 6 créditos, obtiene 3 puntos. En caso contrario, 0.
- Si la asignatura se evalúa a través de trabajos, obtiene 1 punto. En caso contrario, 0.
- Si la asignatura da una importancia mayor o igual al 60 % a la parte de programación, obtiene 5 puntos. En caso contrario, 0.
- Si la asignatura requiere de conocimientos previos, obtiene 3 puntos. En caso contrario, 0.

El sistema de puntuación dado se basa en sus prioridades. Una vez definidas, basta saber cuántos puntos son necesarios para aceptar la asignatura (se supondrán 8) y si cumple o no cada criterio. Imagínese que la asignatura cumple todos los requisitos salvo el tercero. En tal caso suma 7 puntos y el individuo descarta coger la asignatura.

Matemáticamente este proceso de suma de puntuaciones y comparación con un valor mínimo puede representarse por la función

$$\begin{array}{rccl} f\colon & \mathbb{R}^4 & \longrightarrow & \mathbb{R} \\ & y & \longmapsto & f(y) = \begin{cases} 1 & \text{si } Wy - b \geq 0 \\ 0 & \text{si } Wy - b < 0 \end{cases} \end{array} \tag{1}$$

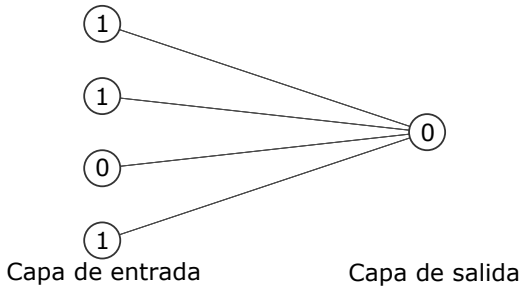

Figura 3: Modelización del ejemplo de la asignatura a través de un perceptrón (gráfico generado con las herramientas en [7]).

considerando $y^t = [1, 1, 0, 1]$ los datos de entrada (1 si se cumple el criterio, 0 si no se cumple), $W = [3, 1, 5, 3]$ la matriz con los puntos asociados a cada criterio (que llamaremos *matriz de pesos*) y $b = 8$ la puntuación mínima para aceptar la asignatura (que llamaremos *sesgo*).

La función resultante (1) es la composición de una función lineal y de la función de Heaviside, conocida en este contexto como *función de activación*. Observemos cómo se traduce en la figura 3: cada circunferencia (que llamaremos *neurona artificial*) contiene un número que indica si se cumple o no el criterio asociado. Esa información se transmite ponderada por unos pesos y un sesgo a la neurona posterior, que tras un cálculo interno devuelve un valor (una *señal*). Si consideramos que la neurona artificial es el cuerpo de una neurona real y las conexiones entre neuronas las dendritas, cuerpo y axón, recuperamos la estructura y funcionamiento de estas células. Las neuronas que se encuentran en la misma etapa de cálculo definen una *capa*. En el perceptrón observamos que las neuronas de la izquierda están relacionadas con argumentos de entrada y por ello diremos que conforman la *capa de entrada*. La neurona de la derecha contiene el valor devuelto tras aplicar (1), que es la salida buscada. Por ello, diremos que se encuentra en la *capa de salida*. Cada segmento representa el peso que conecta la neurona i-ésima de la capa de entrada con la de la capa de salida. El sesgo suele no representarse en este tipo de esquemas.

En el ejemplo considerado debe tenerse en cuenta que tanto la puntuación de los criterios (los llamados pesos) como el valor que genera la elección de la asignatura (el sesgo) han sido seleccionadas por el individuo en cuestión. Sin embargo no está claro (por ejemplo) que el que la asignatura tenga 6 créditos sea más importante que el que evalúe a través de trabajos. Por ello se podrían considerar tanto los pesos como el sesgo (que llamaremos *parámetros*) variables de optimización a ajustar. El cálculo de dichos coeficientes óptimos se podría efectuar tratando de minimizar la discrepancia entre el resultado devuelto por el perceptrón y la opinión de antiguos alumnos sobre la utilidad de la materia una vez cursa-

da. Para ello se haría uso de algún método de descenso. Una vez calculados los coeficientes, el perceptrón resultante podría ser empleado para decidir sobre la utilidad de una materia, aún cuando ningún alumno hubiera dado su opinión sobre la misma.

Por otro lado, la estructura del perceptrón (formado por una capa de entrada y otra de salida) es demasiado simple como para emplearla en la resolución de problemas de interés como la clasificación de imágenes. De ahí surge la idea de concatenar varios de ellos formando una RNA (figura 4). Si somos precisos, como todas las neuronas de una capa están conectadas con todas las de las capas anterior y posterior, se habla de Perceptrón Multicapa. Pese a ello es habitual llamarla RNA. Exploremos esta idea a través de otro ejemplo. Supongamos que tenemos una imagen con cierta resolución (digamos 28×28) en la que se representa un número del 0 al 9 (ver figura 6, izquierda) y deseamos construir una función capaz de determinar de qué número se trata. A cada píxel de la imagen, se le asocia una neurona de la capa de entrada que contiene un número en el intervalo $[0, 1]$ que designa su intensidad de blanco. Como son 10 los posibles números que contiene, en la capa final se definen 10 neuronas. En lugar de definir solamente estas capas, consideremos la introducción de varias intermedias a las que llamaremos *capas internas u ocultas* en las que cada neurona de una capa está conectada a todas las de la capa anterior. Al aprendizaje de redes con varias capas ocultas es a lo que se le conoce como aprendizaje profundo.

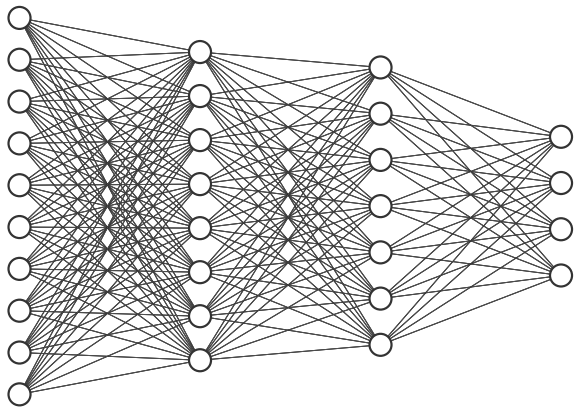

Figura 4: Ejemplo de RNA de cuatro capas (dos internas) con 10, 8, 7 y 4 neuronas de izquierda a derecha [7].

El siguiente paso consistiría en definir los pesos y sesgos asociados a la RNA. Deben ser tales que la red resultante clasifique correctamente cualquier número introducido. En lugar de determinar cuánto deben valer a mano (es decir, de forma explícita), vamos a hacer uso del ML. O lo que es lo mismo, vamos a hacer que sea la propia red la que determine cuáles son los valores que le permiten clasificar correctamente la imagen de un número arbitrario

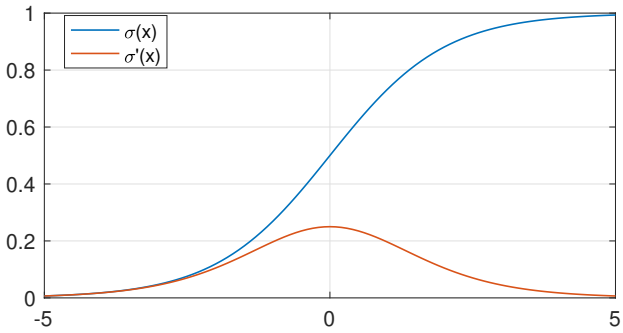

Figura 5: Función logística o sigmoide y su derivada.

del 0 al 9. Para ello haremos como en el ejemplo anterior, es decir, consideraremos los pesos y sesgos variables de optimización que minimizan la diferencia entre la clase devuelta por la red y la clase real sobre una extensa base de datos previamente clasificados. Como técnica de resolución del problema de minimización usaremos el método del gradiente. Dado que en este se trabaja con la derivada de la función a minimizar y en ella está explícita la función de activación, conviene sustituir la función de Heaviside por otra diferenciable. Tradicionalmente, dicha función es la función sigmoide o logística (figura 5, curva azul), cuya forma recuerda a la función escalón:

$$\begin{array}{rccl} \sigma\colon & \mathbb{R} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & \sigma(x) = \frac{1}{1+\exp(-x)} \end{array} \tag{2}$$

Nótese que la función de Heaviside es no diferenciable en un punto y diferenciable con derivada nula en el resto de puntos, de manera que el método del gradiente no actualizaría los valores de los parámetros a optimizar.

Una vez se obtiene una red capaz de clasificar los números, la interpretación de qué realiza cada capa es más complicada que en el ejemplo anterior. A priori vamos a suponer que lo que hace cada capa interna es una incógnita y que cualquier interpretación está supeditada a una comprobación posterior. Podemos pensar que la red razona como el ser humano. De esta manera, en la primera capa interna cada neurona se encargaría de establecer si existe o no cierto trazado presente al dibujar un número: una neurona detectaría segmentos horizontales, otra verticales, otra inclinados, otra curvos, etc. Según cada una detecte más o menos, devolverá un valor más cercano a 1 ó 0. Continuando con este razonamiento, las neuronas de la siguiente capa se encargarían de indicar cómo están conectados los trazados detectados: si se forman trazados cerrados (pensemos en el número 8), si un segmento vertical está unido a uno horizontal (pensemos en el 5), etc. Las capas posteriores analizarían geometrías más complejas de la imagen hasta finalmente llegar a la

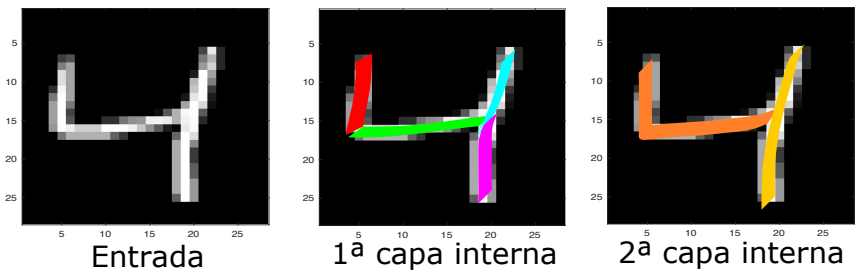

Figura 6: Esquema del posible funcionamiento de una RNA para clasificar números. En la primera capa interna (centro) se detectarían segmentos que forman parte del número. En la segunda capa interna (derecha) se unirían los segmentos detectados para reconstruir el número. Recuérdese que el verdadero funcionamiento es a priori una incógnita [8].

capa final. Pero como decimos, esta no es más que una suposición sin fundamento alguno.

Tras esta motivación inicial, en el resto del trabajo se formalizará el proceso de definición y obtención de una red capaz de clasificar un conjunto de imágenes. En el capítulo 1 nos centraremos en las RNA (que como ya dijimos son realmente Perceptrones Multicapa). Además de su estructura, se describirán las herramientas necesarias para minimizar el error cometido por la red en la clasificación de imágenes (se habla del *entrenamiento de la red*. Como las imágenes que se usan para entrenarla están clasificadas de antemano, se habla de *aprendizaje supervisado de la red*). Para ello se introducirán los métodos del gradiente estocástico y mini-batch y el algoritmo de la propagación inversa. Finalmente se presentarán los resultados devueltos por varias redes con el fin de analizar el papel que juegan los distintos elementos de las mismas. En el capítulo 2 se describirá otro tipo de redes, llamadas Redes Neuronales Convolucionales (RNC), muy utilizadas en la clasificación de imágenes. Se presentarán y se indicarán los cambios a realizar respecto de las RNA para adaptar el aprendizaje a la nueva estructura. Nuevamente el capítulo concluye con el análisis de varias redes obtenidas mediante el uso de códigos elaborados a partir de la teoría.

Capítulo 1

Redes Neuronales Artificiales

En este capítulo vamos a diseñar una RNA arbitraria aplicable al ejemplo de clasificación de imágenes expuesto en la introducción [9]. Primero formalizaremos su estructura. A continuación explicaremos cómo entrenar la red con los métodos del gradiente estocástico y mini-batch para después deducir cómo calcular de forma eficiente el gradiente de la función a minimizar haciendo uso del algoritmo de la propagación inversa. Finalmente se mostrará el comportamiento de varias redes para visualizar el papel que juega cada elemento de la propia red en su aprendizaje.

1.1. Presentación

Recordemos cómo introdujimos los parámetros de una red. Cuando se habló de la matriz de pesos en un perceptrón, W, se mostró una matriz fila que se empleaba para obtener un valor en la neurona de la capa de salida de la figura 3. Posteriormente se incrementó el número de neuronas por capa, n_l, de manera que la matriz de pesos entre dos capas consecutivas, $l-1$ y l, pasa a ser una matriz $W^{[l]}$ de dimensiones $n_l \times n_{(l-1)}$. Análogamente, los sesgos se transforman en vectores $b^{[l]}$ de dimensión n_l. Por comodidad, las neuronas de cada capa se definen en forma de vector. De esta manera, suponiendo que queremos calcular el valor de la neurona i en la capa l partiendo de los valores de la capa previa, basta hacer la operación

$$y_i^{[l]} = \sigma\left(z_i^{[l]}\right) = \sigma\left(\sum_{j=1}^{n_{l-1}} W_{ij}^{[l]} y_j^{[l-1]} + b_i^{[l]}\right) \tag{1.1}$$

Vectorialmente, la operación es

$$y^{[l]} = \sigma\left(z^{[l]}\right) = \sigma\left(W^{[l]} y^{[l-1]} + b^{[l]}\right) \tag{1.2}$$

donde σ actúa sobre cada elemento del vector, ie, $\sigma\left(z_i^{[l]}\right) = \sigma\left(z^{[l]}\right)_i$.

Procedemos a formalizar el problema en cuestión. Supongamos que tenemos una RNA compuesta por $L+1$ capas, $L \geq 1$: la inicial, $l = 0$, la final, $l = L$ y las $L-1$ intermedias. Supongamos que cada una de las capas internas tiene n_l neuronas, $l \in \{1, 2, ..., L-1\}$. Recordemos que el número de neuronas de la primera capa, n_0, es igual al número de píxeles de la imagen introducida (en nuestro caso 28×28) y el número de la última, n_L, es igual al número de clases (en nuestro caso 10, una por cada número del 0 al 9). Sean esas clases $\{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_{n_L}\} = C$. Dada una imagen $y^{[0]}$, se desea construir una función $\widetilde{\mathcal{F}}$ capaz de determinar a cuál de esas clases pertenece. Es decir, se busca hallar $\widetilde{\mathcal{F}}$ tal que

$$\begin{array}{rccl} \widetilde{\mathcal{F}}\colon & \mathbb{R}^{n_0} & \longrightarrow & C \\ & y^{[0]} & \longmapsto & \widetilde{\mathcal{F}}\left(y^{[0]}\right) = \mathcal{C}_i, \text{ si } y^{[0]} \in \mathcal{C}_i \end{array} \tag{1.3}$$

Para dar un valor numérico a las clases, basta con hacer la correspondencia

$$\mathcal{C}_i \equiv (0, 0, ..., 0, 1_i, 0, ..., 0) = \boldsymbol{e}_i \in \mathbb{R}^{n_L} \tag{1.4}$$

de manera que la función deseada, que denotaremos de la misma manera, es

$$\begin{array}{rccl} \widetilde{\mathcal{F}}\colon & \mathbb{R}^{n_0} & \longrightarrow & \mathbb{R}^{n_L} \\ & y^{[0]} & \longmapsto & \widetilde{\mathcal{F}}(y^{[0]}) = \boldsymbol{e}_i, \text{ si } y^{[0]} \in \mathcal{C}_i \end{array} \tag{1.5}$$

La función así construida pretende imitar el conjunto de operaciones internas que realiza un individuo a la hora de ver un número dibujado y deducir de cuál se trata. Por tanto la función no solo es ideal, sino que la solución que devuelve puede ser distinta a la correcta, de la misma manera que una persona puede cometer un error al clasificar la imagen. El objetivo es encontrar una aproximación $\mathcal{F} \approx \widetilde{\mathcal{F}}$ que pueda ser implementada en un ordenador. Dicha función $\mathcal{F}$ se define como la composición de tantas funciones como transiciones entre capas presente la red (L). A su vez, la función asociada a cada una de las transiciones es la composición de una función afín con otra no lineal, que se denotará por σ por coherencia con lo explicado en la introducción. Es decir:

$$\mathcal{F} = \mathcal{F}_{n_L} \circ \left(\mathcal{F}_{n_{L-1}} \circ (... \circ \mathcal{F}_1)\right) \tag{1.6}$$

con

$$\begin{array}{rccl} \mathcal{F}_i\colon & \mathbb{R}^{n_{i-1}} & \longrightarrow & \mathbb{R}^{n_i} \\ & y^{[i-1]} & \longmapsto & \mathcal{F}_i\left(y^{[i-1]}\right) = y^{[i]} = \sigma\left(z^{[i]}\right) = \sigma\left(W^{[i]} y^{[i-1]} + b^{[i]}\right),\ i \in \{1, 2, ..., L\} \end{array} \tag{1.7}$$

donde

$$\begin{aligned} &z^{[i]} \in \mathbb{R}^{n_i},\ i = 1, 2, ..., L \\ &y^{[i]} \in \mathbb{R}^{n_i},\ i = 0, 1, ..., L \\ &W^{[j]} \in \mathcal{M}_{n_j \times n_{j-1}}(\mathbb{R}),\ j = 1, 2, , ..., L \\ &b^{[j]} \in \mathbb{R}^{n_j},\ j = 1, 2, ..., L \end{aligned} \tag{1.8}$$

representan los valores a los que se les aplica la función no lineal en cada capa, el valor de la neurona en cada capa y los valores de las matrices de pesos y vectores de sesgos de cada función afín, respectivamente.

Será conveniente agrupar los pesos y sesgos de la red en una única variable, que llamaremos *variable de control*,

$$\boldsymbol{u} \equiv \left(W^{[1]}, b^{[1]}, ..., W^{[L]}, b^{[L]}\right) \tag{1.9}$$

que pertenece al conjunto

$$\mathcal{U} = \left\{ \boldsymbol{u} \in \prod_{j=1}^{L} \left(\mathcal{M}_{n_j \times n_{j-1}}(\mathbb{R}) \times \mathbb{R}^{n_j}\right) \right\} \tag{1.10}$$

Asimismo, agruparemos el resto de elementos de (1.8) en otra variable, que llamaremos *variable de estado*,

$$\boldsymbol{y}^t \equiv \left(y^{[0]t}, z^{[1]t}, y^{[1]t}, ..., z^{[L]t}, y^{[L]t}\right) \tag{1.11}$$

que pertenece al conjunto

$$\mathcal{E} = \left\{ \boldsymbol{y} \in \mathbb{R}^{n_0} \times \prod_{j=1}^{L} \left(\mathbb{R}^{n_j} \times \mathbb{R}^{n_j}\right) \right\} \tag{1.12}$$

El objetivo es hallar $\boldsymbol{u}_{opt}$ que permitan a la red clasificar adecuadamente cualquier número del 0 al 9 introducido. Los valores iniciales de los elementos de $\boldsymbol{u}$, $\boldsymbol{u}_{inicial}$, se tomarán de acuerdo con alguna inicialización adecuada a la función de activación empleada. En este caso haremos uso de la *inicialización Xabier* [10]:

$$\begin{aligned} W^{[l]} &\in \mathrm{Unif}(0, \tfrac{2}{n^{[l]} + n^{[l-1]}}) \\ b^{[l]} &= 0 \\ l &= 1, 2, ..., L \end{aligned} \tag{1.13}$$

La evolución de $\boldsymbol{u}_{inicial}$ a $\boldsymbol{u}_{opt}$ constituye el entrenamiento de la red. Para poder realizar este proceso es necesario poseer un conjunto de datos $\{{}^i y^{[0]}\}_{i=1}^{N}$ (N imágenes de números en nuestro caso) con $\widetilde{\mathcal{F}}\left({}^i y^{[0]}\right)$ conocidos de antemano (como ya dijimos, estamos ante un

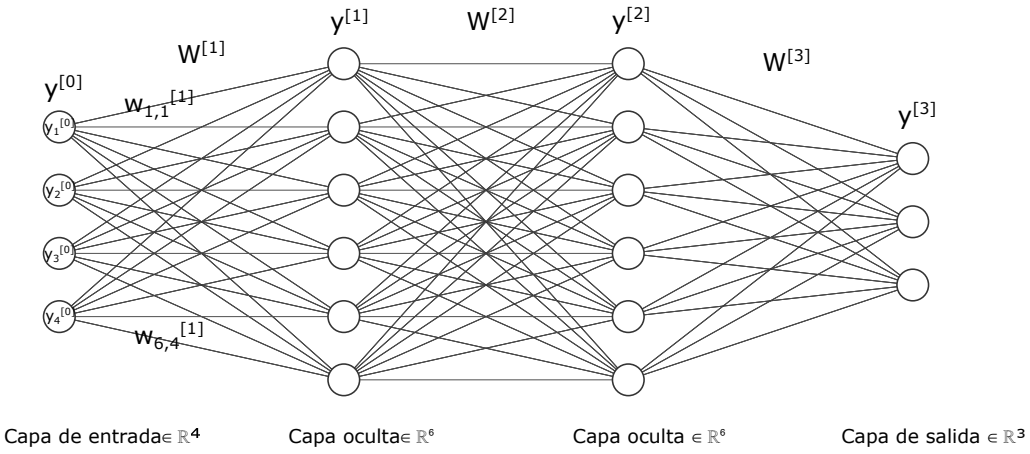

Figura 1.1: RNA de cuatro capas ($L = 3$) con 4, 6, 6 y 3 neuronas, respectivamente. El conjunto de conexiones entre cada par de capas representa la matriz $W^{[i]}$, siendo cada conexión uno de los pesos de la matriz, $i \in \{1, 2, 3\}$. Los valores de las neuronas de cada capa conforman el vector $y^{[j]}$, $j \in \{0, 1, 2, 3\}$ [7].

aprendizaje supervisado). Como definimos (1.11) suponiendo que había un solo dato inicial, ahora debemos definir tantas variables de estado ${}^{i}\boldsymbol{y} \in \mathcal{E}$ como datos ${}^{i}y^{[0]}$ haya. Dado que la aproximación $\mathcal{F} \approx \widetilde{\mathcal{F}}$ se construye sobre esos datos, cuantos más sean, mejor, dado que la red clasificará de forma más precisa números externos a ese banco de datos. Gracias a la traducción de las clases en vectores (ecuación (1.4)), es sencillo visualizar el entrenamiento de la red como un problema de mínimos cuadrados. Para ello basta plantear el aprendizaje de la red como un problema donde debemos encontrar $\boldsymbol{u}_{opt} \in \mathcal{U}$ tal que

$$\widetilde{\mathcal{J}}(\boldsymbol{u}_{opt}) = \min_{\boldsymbol{u}\in\mathcal{U}} \widetilde{\mathcal{J}}(\boldsymbol{u}) \tag{1.14}$$

donde

$$\widetilde{\mathcal{J}}(\boldsymbol{u}) = \frac{1}{N}\sum_{i=1}^{N} {}^{i}\widetilde{\mathcal{J}}(\boldsymbol{u}) \tag{1.15}$$

con

$${}^{i}\widetilde{\mathcal{J}}(\boldsymbol{u}) = \frac{1}{2}\left|\left|\mathcal{F}\left({}^{i}y^{[0]}\right) - \widetilde{\mathcal{F}}\left({}^{i}y^{[0]}\right)\right|\right|^2 \tag{1.16}$$

Notemos que la dependencia de ${}^{i}\widetilde{\mathcal{J}}$ en $\boldsymbol{u}$ está implícita en la propia construcción de $\mathcal{F}$ de la forma indicada en (1.6) y (1.7). La expresión de $\widetilde{\mathcal{J}}$ se simplifica notablemente introduciendo el funcional

$${}^{i}\mathcal{J}({}^{i}\boldsymbol{y}, \boldsymbol{u}) = \frac{1}{2}\left|\left|\mathcal{F}\left({}^{i}y^{[0]}\right) - {}^{i}y^{[L]}\right|\right|^2 \tag{1.17}$$

que depende de la variable de control y de la variable de estado. En efecto,

$${}^{i}\widetilde{\mathcal{J}}(\boldsymbol{u}) = {}^{i}\,\mathcal{J}({}^{i}\boldsymbol{y}(\boldsymbol{u}), \boldsymbol{u}) \tag{1.18}$$

donde ${}^i\boldsymbol{y}(\boldsymbol{u})$ se define mediante las ecuaciones

$$\left.\begin{array}{rcl} {}^iz^{[k]} - \left(W^{[k]}{}^iy^{[k-1]} + b^{[k]}\right) & = & 0 \\ {}^iy^{[k]} - \sigma\left({}^iz^{[k]}\right) & = & 0 \end{array}\right\} \tag{1.19}$$

con $k = 1, 2, ..., L$ e $i = 1, 2, ..., N$.

De acuerdo con lo mostrado, resulta conveniente reinterpretar el problema de optimización original como otro donde el objetivo sea encontrar $\boldsymbol{u}_{opt} \in \mathcal{U}$, ${}^i\boldsymbol{y}_{opt} \in \mathcal{E}$, $i = 1, 2, ..., N$, tales que

$$\mathcal{J}({}^1\boldsymbol{y}_{opt}, ..., {}^N\boldsymbol{y}_{opt}, \boldsymbol{u}_{opt}) = \min_{\substack{\boldsymbol{u}\in\mathcal{U} \\ {}^1\boldsymbol{y},...,{}^N\boldsymbol{y}\in\mathcal{E}}} \mathcal{J}({}^1\boldsymbol{y}, ..., {}^N\boldsymbol{y}, \boldsymbol{u}) \tag{1.20}$$

donde

$$\mathcal{J}({}^1\boldsymbol{y}, ..., {}^N\boldsymbol{y}, \boldsymbol{u}) = \frac{1}{N}\sum_{i=1}^{N} {}^i\mathcal{J}({}^i\boldsymbol{y}, \boldsymbol{u}) \tag{1.21}$$

con ${}^i\mathcal{J}({}^i\boldsymbol{y}, \boldsymbol{u})$ dada por (1.17) e ${}^i\boldsymbol{y}$ y $\boldsymbol{u}$ relacionadas por (1.19) para cada $i = 1, 2, ..., N$.

1.2. Entrenamiento de la red

1.2.1. Método del gradiente estocástico y mini-batch

Como se anunció en la introducción, para conseguir minimizar la función $\widetilde{\mathcal{J}}$ se hará uso de un método de descenso. En concreto se empleará una variante del método de gradiente clásico [11]. De forma resumida, dada la función $\widetilde{\mathcal{J}} : \mathbb{R}^s \to \mathbb{R}$ y un valor inicial $\boldsymbol{u}_0$, el método permite desplazarse hacia valores que reducen el valor de $\widetilde{\mathcal{J}}$. Realizando una perturbación suficientemente pequeña $\Delta\boldsymbol{u}$ en $\boldsymbol{u}$ y haciendo una expansión de Taylor de primer orden,

$$\widetilde{\mathcal{J}}(\boldsymbol{u} + \Delta\boldsymbol{u}) \approx \widetilde{\mathcal{J}}(\boldsymbol{u}) + \sum_{r=1}^{s} \frac{\partial\widetilde{\mathcal{J}}(\boldsymbol{u})}{\boldsymbol{u}_r}\Delta\boldsymbol{u}_r = \widetilde{\mathcal{J}}(\boldsymbol{u}) + \nabla\widetilde{\mathcal{J}}(\boldsymbol{u})^T\Delta\boldsymbol{u} \tag{1.22}$$

se deduce que $\widetilde{\mathcal{J}}(\boldsymbol{u} + \Delta\boldsymbol{u}) \leq \widetilde{\mathcal{J}}(\boldsymbol{u})$ cuanto más negativo sea el sumando $\nabla\widetilde{\mathcal{J}}(\boldsymbol{u})^T\Delta\boldsymbol{u}$. De acuerdo con la desigualdad de Cauchy-Schwarz, la dirección que lo garantiza es la definida por $-\nabla\widetilde{\mathcal{J}}(\boldsymbol{u})$. El método del gradiente no es más que la aplicación de esta idea,

$$\boldsymbol{u}_k = \boldsymbol{u}_{k-1} - \mu\nabla\widetilde{\mathcal{J}}(\boldsymbol{u}) \tag{1.23}$$

en donde μ es el *paso* y k hace referencia a la iteración (se parte de un valor inicial $\boldsymbol{u}_0$). En el contexto de las RNA se le conoce como *razón de aprendizaje*.

En el caso objeto de estudio la actualización iterativa puede expresarse de la forma

$$\begin{array}{l} W^{[l]} \longleftarrow W^{[l]} - \frac{\mu}{N}\sum_{i=1}^{N}\left(\frac{\partial}{\partial W^{[l]}}({}^i\widetilde{\mathcal{J}})\right) \\ b^{[l]} \longleftarrow b^{[l]} - \frac{\mu}{N}\sum_{i=1}^{N}\left(\frac{\partial}{\partial b^{[l]}}({}^i\widetilde{\mathcal{J}})\right) \end{array} \tag{1.24}$$

donde las derivadas parciales deben entenderse como las derivadas respecto a cada una de las componentes de la matriz de pesos y vector de sesgos. Nótese que el sumando es un promedio de los gradientes de cada ${}^i\widetilde{\mathcal{J}}$.

Es comúnmente conocida la efectividad de este método para hallar mínimos en funciones con ciertas buenas propiedades como la convexidad. Sin embargo, ese no es el caso de la función asociada al entrenamiento de una red. Pese a ser regular, la función que tiene tras de sí presenta múltiples mínimos relativos y es posible tender a uno de ellos y no poder salir de él. A esto debemos añadir el elevado coste computacional que representa realizar cada paso del método. Tengamos en cuenta que en general cuando se desea entrenar una red, $N \gg 1000$ (en nuestro ejemplo, $N = 60000$). Todo ello hace necesario otro planteamiento a la hora de hacer uso de este método.

Nos proponemos como objetivo describir una variante en la que se usen todos los datos que se posean por igual y que sea capaz de evitar los problemas del método clásico. La idea fundamental de la variante reside en reemplazar el gradiente del funcional coste (interpretado como la media de los N gradientes de ${}^i\widetilde{\mathcal{J}}$) por una aproximación del mismo que use m datos [12]. Tradicionalmente se usan tres nombres para distinguir los métodos de gradiente aquí expuestos: si $m = 1$ hablaremos del *gradiente estocástico* , si $1 < m < N$ hablaremos del *gradiente mini-batch* y si $m = N$ hablaremos del *gradiente batch*. Dejando a un lado el tercer gradiente (que es el caso (1.24)), donde se usan todas las imágenes disponibles (todo el *batch* o lote), en el resto de casos se toman conjuntos más pequeños de manera aleatoria y sin reemplazamiento hasta haber empleado todas. Se dice que se completa un *epoch* cuando se ha usado una vez todo el batch. Pueden hacerse tantos epochs como uno desee. En el caso del segundo método, conviene decir que se denomina mini-batch a cada uno de los subconjuntos de m elementos en los que se dividen las N imágenes, siendo en general $m \ll N$. Podemos expresar los tres métodos conjuntamente mediante la expresión

$$\begin{aligned} W^{[l]} &\longleftarrow W^{[l]} - \tfrac{\mu}{m} \textstyle\sum_{i=1}^{m} \left(\frac{\partial}{\partial W^{[l]}} ({}^i\widetilde{\mathcal{J}}) \right) \\ b^{[l]} &\longleftarrow b^{[l]} - \tfrac{\mu}{m} \textstyle\sum_{i=1}^{m} \left(\frac{\partial}{\partial b^{[l]}} ({}^i\widetilde{\mathcal{J}}) \right) \end{aligned} \tag{1.25}$$

donde m puede ser igual a 1 (estocástico), $1 < m < N$ (mini-batch) o $m = N$ (batch). En la práctica, a la hora de trabajar con mini-batches de m elementos se procura que m divida a N, de manera que se permuten todos los datos y se formen $k = \frac{N}{m}$ conjuntos. En cada una de las k iteraciones en las que se aplique (1.25) se usa uno de los mini-batches. Una vez usados todos se vuelven a permutar las N imágenes, formándose k nuevos conjuntos.

Los casos $m < N$ superan el problema del coste por iteración y permiten a la función escapar de los mínimos relativos y puntos de silla [13] [14]. Además se siguen empleando

todos los datos. Podemos preguntarnos si no usar todas las imágenes en cada iteración impide que el método converja de la misma forma que lo haría el caso clásico. Como el método usa todas las fotos, pero no a la vez, su comportamiento es similar en promedio al del caso $m = N$, pero la trayectoria que sigue es menos regular [15]. Esta irregularidad es la que facilita el escape de mínimos relativos. Por el mismo motivo, en caso de estar cerca de un mínimo óptimo, no le será posible permanecer en él. A modo de aclaración del comportamiento de los tres métodos, se muestra en la figura 1.2 la evolución del valor de los dos parámetros Θ_0 y Θ_1 presentes en la expresión del error cuadrático medio de un modelo de regresión lineal [5]. En el capítulo de resultados se hará una comparativa del impacto del valor de m en el aprendizaje de la red.

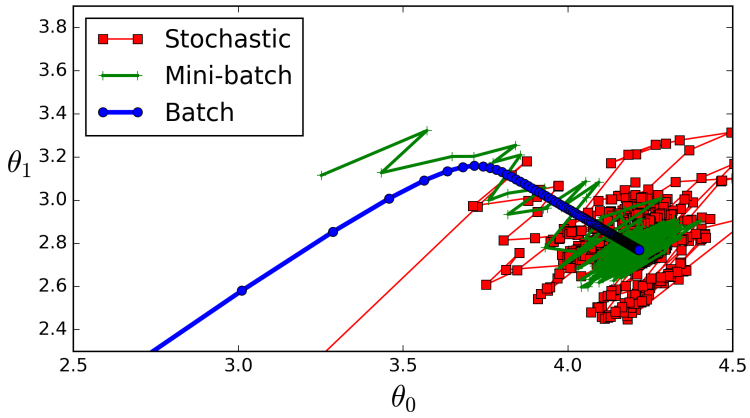

Figura 1.2: Ejemplo del comportamiento de los métodos de gradiente clásico (azul, $m = N$), estocástico (rojo, $m = 1$) y mini-batch (verde, $1 < m < N$). Obsérvese que cuanto menor es m, más errática es la trayectoria [5].

Dada la complejidad de la función asociada a una red neuronal, minimizar la función (1.15) conduce en general a la obtención de una red con una baja precisión en la clasificación de imágenes externas a la base de datos empleada en su entrenamiento. Al problema se le conoce como *sobreestimación* (u overfitting en inglés). Además de emplear el mayor número de datos posible, es recomendable hacer uso de alguna otra técnica que actúe sobre la red para prevenir el problema [16]. Dichas técnicas se recogen bajo el nombre de *regularizadores*.

En nuestro caso añadiremos a la expresión (1.15) el sumando

$$\frac{\eta}{2m} \sum_{i,j,l} \left(W_{ij}^{[l]} \right)^2 \tag{1.26}$$

inspirado en regularizaciones clásicas como la de Tikhonov [17]. En dicho sumando el término η es la *razón del regularizador* y m es el tamaño del mini-batch. El motivo por el

que se añade m en el denominador es para que su contribución en la expresión (1.25) sea similar a la del valor promedio de los m gradientes de ${}^{i}\widetilde{\mathcal{J}}$. Si tenemos en cuenta (1.26) a la hora de deducir una expresión análoga a (1.25), se obtiene

$$W^{[l]} \longleftarrow W^{[l]}\left(1-\frac{\mu\eta}{m}\right)-\frac{\mu}{m}\sum_{i=1}^{m}\left(\frac{\partial}{\partial W^{[l]}}({}^{i}\widetilde{\mathcal{J}})\right) \tag{1.27}$$

Nótese cómo esta expresión requiere de tres parámetros a definir antes del proceso de aprendizaje de la red: m, μ y η. Se comprobará cómo modificar los valores de η y m produce efectos significativos tanto en la evolución como en la red final obtenida.

1.2.2. Algoritmo de la propagación inversa

Dado que nos hemos propuesto aplicar el método del gradiente (ya sea estocástico o mini-batch), el siguiente paso consiste en determinar el gradiente de $\widetilde{\mathcal{J}}$. A la hora de calcularlo se hace uso del método del adjunto [18][19]. Dado que en los cálculos que se van a mostrar puede perderse la esencia de este método, se hace una breve presentación del mismo.

Observación 1.1. Para simplificar la notación vamos a almacenar las expresiones de (1.19) para cada i en

$${}^{i}g({}^{i}\boldsymbol{y},\boldsymbol{u})=\left({}^{i}g_1,{}^{i}g_2,...,{}^{i}g_{2L}\right)^{t} \tag{1.28}$$

con

$$\begin{aligned} {}^{i}g_{2k-1} &= {}^{i}z^{[k]}-\left(W^{[k]}{}^{i}y^{[k-1]}+b^{[k]}\right) \\ {}^{i}g_{2k} &= {}^{i}y^{[k]}-\sigma\left({}^{i}z^{[k]}\right) \end{aligned} \tag{1.29}$$

$k=1,2,...,L$. Además trabajaremos solo con uno de los sumandos ${}^{i}\widetilde{\mathcal{J}}$ y omitiremos el superíndice i (es decir, ${}^{i}\boldsymbol{y}$ y ${}^{i}g({}^{i}\boldsymbol{y},\boldsymbol{u})$ se denotarán como $\boldsymbol{y}$ y $g(\boldsymbol{y},\boldsymbol{u})$, por ejemplo).

Sean las funciones diferenciables $\mathcal{J}(\boldsymbol{y},\boldsymbol{u}):\mathbb{R}^{n_{\boldsymbol{y}}}\times\mathbb{R}^{n_{\boldsymbol{u}}}\to\mathbb{R}$ y $g(\boldsymbol{y},\boldsymbol{u}):\mathbb{R}^{n_{\boldsymbol{y}}}\times\mathbb{R}^{n_{\boldsymbol{u}}}\to\mathbb{R}^{n_{\boldsymbol{y}}}$. Supongamos que la ecuación $g(\boldsymbol{y},\boldsymbol{u})=0$ define a $\boldsymbol{y}$ como función de $\boldsymbol{u}$ (es decir, podremos hablar de $\boldsymbol{y}(\boldsymbol{u})$) y que la jacobiana $\partial_{\boldsymbol{y}}g$ (que es una matriz $n_{\boldsymbol{y}}\times n_{\boldsymbol{y}}$) es no singular. Nótese que en nuestro caso estas hipótesis se verifican por la estructura de las restricciones en (1.19). Consideremos $\widetilde{\mathcal{J}}(\boldsymbol{u})=\mathcal{J}(\boldsymbol{y}(\boldsymbol{u}),\boldsymbol{u})$ y tratemos de calcular $\nabla_{\boldsymbol{u}}\widetilde{\mathcal{J}}$:

$$\nabla_{\boldsymbol{u}}\widetilde{\mathcal{J}}=\partial_{\boldsymbol{y}}\mathcal{J}\partial_{\boldsymbol{u}}\boldsymbol{y}+\partial_{\boldsymbol{u}}\mathcal{J} \tag{1.30}$$

Para obtener $\partial_{\boldsymbol{u}}\boldsymbol{y}$ derivamos la ecuación $g(\boldsymbol{y},\boldsymbol{u})=0$ con respecto a $\boldsymbol{u}$:

$$\partial_{\boldsymbol{y}}g\partial_{\boldsymbol{u}}\boldsymbol{y}+\partial_{\boldsymbol{u}}g=0 \tag{1.31}$$

Despejando $\partial_{\boldsymbol{u}}\boldsymbol{y}$ en (1.31) y sustituyéndolo en (1.30), que es posible por la hipótesis realizada sobre g, se obtiene

$$\nabla_{\boldsymbol{u}}\widetilde{\mathcal{J}} = -\partial_{\boldsymbol{y}}\mathcal{J}(\partial_{\boldsymbol{y}}g)^{-1}\partial_{\boldsymbol{u}}g + \partial_{\boldsymbol{u}}\mathcal{J} \tag{1.32}$$

Podemos preguntarnos por qué no calcular de forma directa $\partial_{\boldsymbol{u}}\boldsymbol{y}$ en (1.30) o haciendo uso de (1.31). En ambos casos la respuesta es la misma: el elevado coste computacional que acompaña a esos cálculos dada la elevada dimensión de $\boldsymbol{y}$ y $\boldsymbol{u}$ invita a buscar un método más eficiente. Por ese motivo sustituiremos el vector $-\partial_{\boldsymbol{y}}\mathcal{J}\ (\partial_{\boldsymbol{y}}g)^{-1}$ por la solución del sistema lineal

$$(\partial_{\boldsymbol{y}}g)^t\boldsymbol{\lambda} = (\partial_{\boldsymbol{y}}\mathcal{J})^t \tag{1.33}$$

conocido como *sistema adjunto*. Sustituyéndolo en (1.32), se obtiene

$$\nabla_{\boldsymbol{u}}\widetilde{\mathcal{J}} = -\boldsymbol{\lambda}^t\partial_{\boldsymbol{u}}g + \partial_{\boldsymbol{u}}\mathcal{J} \tag{1.34}$$

Observación 1.2. Con lo aquí expuesto es suficiente para calcular el gradiente deseado. Sin embargo resulta útil definir la función Lagrangiana

$$\mathcal{L}(\boldsymbol{y},\boldsymbol{u},\boldsymbol{\lambda}) = \mathcal{J}(\boldsymbol{y},\boldsymbol{u}) - \boldsymbol{\lambda}^t g(\boldsymbol{y},\boldsymbol{u}) \tag{1.35}$$

para automatizar los cálculos (obtención del sistema adjunto (1.33) y del gradiente (1.34)). De esta manera, si calculamos las derivadas parciales respecto a los elementos de $\boldsymbol{\lambda}$ y las igualamos a 0 se obtiene

$$\nabla_{\boldsymbol{\lambda}}\mathcal{L} = -g(\boldsymbol{y},\boldsymbol{u}) = 0 \tag{1.36}$$

Repitiendo el razonamiento con las parciales respecto a $\boldsymbol{y}$, se obtiene

$$\nabla_{\boldsymbol{y}}\mathcal{L} = \partial_{\boldsymbol{y}}\mathcal{J} - \boldsymbol{\lambda}^t\partial_{\boldsymbol{y}}g = 0 \tag{1.37}$$

que reconocemos como el sistema adjunto definido en (1.33). Finalmente calculando la parcial respecto de $\boldsymbol{u}$, obtenemos

$$\nabla_{\boldsymbol{u}}\mathcal{L} = \partial_{\boldsymbol{u}}\mathcal{J} - \boldsymbol{\lambda}^t\partial_{\boldsymbol{u}}g \tag{1.38}$$

Evaluando las tres expresiones anteriores en $(\boldsymbol{y},\boldsymbol{u},\boldsymbol{\lambda}) = (\boldsymbol{y}(\boldsymbol{u}),\boldsymbol{u},\boldsymbol{\lambda}(\boldsymbol{u}))$, conseguimos que (1.38) coincida con (1.34).

En resumen, el método adjunto es una técnica empleada en el cálculo de gradientes de dimensión elevada por el menor número de operaciones que necesita comparado con otras técnicas. En *teoría del control óptimo* se usa, por ejemplo, en el estudio de sistemas dinámicos [20].

Como vamos a ver a continuación, resolveremos el sistema adjunto de nuestro problema y obtendremos la expresión del gradiente en función de los elementos de $\boldsymbol{\lambda}$. Recordemos que todo este apartado se está desarrollando suponiendo que estamos usando una sola imagen, de manera que lo que obtengamos se aplicará a cada ${}^{i}\widetilde{\mathcal{J}}(\boldsymbol{u})$. Para facilitar los cálculos, distinguiremos los elementos de $\boldsymbol{\lambda}$ que acompañan a cada expresión en (1.29) con las letras λ y μ. De esta manera, construimos el Lagrangiano del problema,

$$\begin{aligned}&\mathcal{L}(\boldsymbol{u},\boldsymbol{u},\boldsymbol{\lambda}) = \\ &= \tfrac{1}{2}\sum_{i=1}^{n_L}\left(y_i^{[L]} - \left(\widetilde{\mathcal{F}}\left(y^{[0]}\right)\right)_i\right)^2 - \sum_{m=1}^{L}\sum_{j=1}^{n_m}\lambda_j^{[m]}\left(z_j^{[m]} - \sum_{k=1}^{n_{m-1}}W_{jk}^{[m]}y_k^{[m-1]} - b_j^{[m]}\right) - \\ &- \sum_{m=1}^{L}\sum_{k=1}^{n_m}\mu_j^{[m]}\left(y_j^{[m]} - \sigma\left(z_j^{[m]}\right)\right)\end{aligned} \tag{1.39}$$

Observación 1.3. Haremos uso a mayores del convenio de Einstein [1]. Como ejemplo se presenta uno de los elementos de una de las ecuaciones de (1.19):

$$z_j^{[i]} = \sum_{k=1}^{n_{i-1}} W_{jk}^{[i]}y_k^{[i-1]} + b_j^{[i]} = W_{jk}^{[i]}y_k^{[i-1]} + b_j^{[i]} \tag{1.40}$$

También emplearemos la delta de Kronecker,

$$\delta_{ij} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases} \tag{1.41}$$

Con esta notación se procede a derivar los sumandos de $\mathcal{L}$ respecto de $\boldsymbol{y}$ para aplicar (1.19):

$$\frac{\partial}{\partial z_s^{[l]}}(\mathcal{J}) = \frac{\partial}{\partial z_s^{[l]}}\left[\frac{1}{2}\left(y_i^{[L]} - \left(\widetilde{\mathcal{F}}(y^{[0]})\right)_i\right)^2\right] = 0 \tag{1.42}$$

$$\frac{\partial}{\partial y_s^{[l]}}(\mathcal{J}) = \left(y_i^{[L]} - \left(\widetilde{\mathcal{F}}(y^{[0]})\right)_i\right)\delta_{si}\delta_{lL} = \left(y_s^{[L]} - \left(\widetilde{\mathcal{F}}(y^{[0]})\right)_s\right)\delta_{lL} \tag{1.43}$$

$$\frac{\partial}{\partial z_s^{[l]}}\left(z_j^{[i]} - W_{jk}^{[i]}y_k^{[i-1]} - b_j^{[i]}\right) = \delta_{sj}\delta_{il} \tag{1.44}$$

$$\frac{\partial}{\partial z_s^{[l]}}\left(y_j^{[i]} - \sigma\left(z_j^{[i]}\right)\right) = -\sigma'\left(z_j^{[i]}\right)\delta_{sj}\delta_{il} = -\sigma'\left(z_s^{[l]}\right) \tag{1.45}$$

$$\frac{\partial}{\partial y_s^{[l]}}\left(z_j^{[i]} - W_{jk}^{[i]}y_k^{[i-1]} - b_j^{[i]}\right) = -W_{jk}^{[i]}\delta_{sk}\delta_{(i-1)l} = -W_{js}^{[i]}\delta_{(i-1)l} \tag{1.46}$$

$$\frac{\partial}{\partial y_s^{[l]}}\left(y_j^{[i]} - \sigma\left(z_j^{[i]}\right)\right) = \delta_{sj}\delta_{il} \tag{1.47}$$

[1] dicho convenio es el siguiente: "índices repetidos en un producto indica suma en dicho índice"

Aplicando la expresión (1.37),

$$\left.\begin{array}{rcl} \frac{\partial}{\partial z_s^{[l]}}(\mathcal{L}) & = & 0 \\ \frac{\partial}{\partial y_s^{[l]}}(\mathcal{L}) & = & 0 \end{array}\right\} \text{ con } l = 1, 2, ..., L; s = 1, 2, ..., n_l \tag{1.48}$$

y haciendo uso por un lado de (1.42), (1.44) y (1.45) obtenemos

$$\begin{array}{c} \frac{\partial}{\partial z_s^{[l]}}(\mathcal{L}) = \left(-\lambda_j^{[m]} + \mu_j^{[m]}\sigma'(z_j^{[m]})\right)\delta_{sj}\delta_{ml} = -\lambda_s^{[l]} + \mu_s^{[l]}\sigma'(z_s^{[l]}) \\ \Rightarrow \lambda_s^{[l]} = \mu_s^{[l]}\sigma'(z_s^{[l]}) \end{array} \tag{1.49}$$

y usando por otro lado (1.43), (1.46) y (1.47) obtenemos

$$\begin{array}{c} \frac{\partial}{\partial y_s^{[l]}}(\mathcal{L}) = \left(y_i^{[L]} - (\mathcal{F}(y^{[0]}))_i\right)\delta_{si}\delta_{Ll} + \lambda_j^{[m]}W_{jk}^{[m]}\delta_{sk}\delta_{(m-1)l} - \mu_j^{[m]}\delta_{sj}\delta_{ml} = \\ = \left(y_s^{[L]} - (\mathcal{F}(y^{[0]}))_s\right)\delta_{Ll} + \lambda_j^{[m]}W_{js}^{[m]}\delta_{(m-1)l} - \mu_s^{[l]} \\ \Rightarrow \mu_s^{[l]} = \begin{cases} y_s^{[L]} - (\widetilde{\mathcal{F}}(y^{[0]}))_s & \text{si } l = L \\ ((W^{[l+1]})^t\lambda^{[l+1]})_s & \text{si } l \neq L \end{cases} \end{array} \tag{1.50}$$

Ahora despejamos $\lambda^{[l]}$ en función de los valores de $\mu^{[l]}$ y expresamos ambos en forma vectorial[2],

$$\mu^{[l]} = \begin{cases} y^{[L]} - (\widetilde{\mathcal{F}}(y^{[0]})) & \text{si } l = L \\ (W^{[l+1]})^t\lambda^{[l+1]} & \text{si } l \neq L \end{cases} \tag{1.51}$$

$$\lambda^{[l]} = \begin{cases} \left(y^{[L]} - (\widetilde{\mathcal{F}}(y^{[0]}))\right) \odot \sigma'(z^{[l]}) & \text{si } l = L \\ \left((W^{[l+1]})^t\lambda^{[l+1]}\right) \odot \sigma'(z^{[l]}) & \text{si } l \neq L \end{cases} \tag{1.52}$$

El último paso consiste en obtener (1.38). Como para el dato inicial fijado $y^{[0]}$, tenemos $\boldsymbol{y}(\boldsymbol{u})$ y $\boldsymbol{\lambda}(\boldsymbol{u})$, $\nabla_{\boldsymbol{u}}\mathcal{L}$ coincide con $\nabla_{\boldsymbol{u}}\widetilde{\mathcal{J}}$:

$$\frac{\partial}{\partial W_{sp}^{[l]}}(\mathcal{L}) = \lambda_j^{[m]}y_k^{[m-1]}\delta_{pk}\delta_{js}\delta_{ml} = \lambda_s^{[l]}y_p^{[l-1]} \tag{1.53}$$

$$\frac{\partial}{\partial b_s^{[l]}}(\mathcal{L}) = \lambda_j^{[m]}\delta_{js}\delta_{ml} = \lambda_s^{[l]} \tag{1.54}$$

Las expresiones (1.53) y (1.54) pueden reescribirse en forma vectorial como ya se hizo

[2] $\odot$ hace referencia al producto de Hadamard o producto componente a componente de dos vectores

en (1.51) y (1.52), obteniéndose

$$\boxed{\begin{array}{l} \frac{\partial}{\partial W^{[l]}}(\widetilde{\mathcal{J}}) = \left(\lambda^{[l]}\right)\left(y^{[l-1]}\right)^t \\ \frac{\partial}{\partial b^{[l]}}(\widetilde{\mathcal{J}}) = \lambda^{[l]} \\ \text{con } \lambda^{[l]} = \left\{ \begin{array}{ll} \left(y^{[L]} - \widetilde{\mathcal{F}}(y^{[0]})\right) \odot \sigma'(z^{[L]}) & \text{si } l = L \\ \left((W^{[l+1]})^t \lambda^{[l+1]}\right) \odot \sigma'(z^{[l]}) & \text{si } l \neq L \end{array} \right. \end{array}} \tag{1.55}$$

Si se analiza el resultado, se observa que para obtener las derivadas parciales de la matriz de pesos y vector de sesgos l-ésimos de la red se emplea información de la capa (l+1)-ésima. Es decir, la información de cómo varía $\widetilde{\mathcal{J}}$ se *propaga* del final al inicio de la red. Por ese motivo al algoritmo que de estas expresiones se deduce se le conoce como algoritmo de la propagación inversa.

Una vez calculado, debe aplicarse a cada una de las m imágenes del mini-batch en (1.25) para poder realizar una iteración del método del gradiente mini-batch (o estocástico si $m = 1$).

Hacemos constar que las componentes $\lambda^{[l]}$ de la variable adjunta $\boldsymbol{\lambda}$ verifican

$$\lambda^{[l]} = \frac{\partial}{\partial b^{[l]}}(\widetilde{\mathcal{J}}) = \frac{\partial}{\partial z^{[l]}}(\widetilde{\mathcal{J}}) \tag{1.56}$$

porque será de gran utilidad para la deducción simplificada del algoritmo en Redes Neuronales Convolucionales del capítulo 2.

Históricamente el algoritmo se desarrolla en los años 60 de forma paralela a la teoría de las redes neuronales y se implementa por primera vez en 1970 [21] (publicación en inglés en 1976). Pese a ello, el interés inicial de la comunidad científica se desvanece paulatinamente. En este contexto, se publica un artículo en 1982 en el que algoritmo y redes se combinan [22]. Finalmente en 1985 la publicación del artículo [23] populariza de nuevo este campo, aumentando cada año el número de contribuciones hasta el día de hoy.

Pese a la sencillez de su deducción, actualmente sigue siendo usado (con alguna modificación según el tipo de red neuronal, como ya veremos). Lo que debe quedar claro es que la propagación inversa *no es solo aplicar la regla de Leibniz de forma iterativa, sino que es la forma eficiente de aplicar la regla de la cadena a grandes redes con nodos diferenciables* [24] (entiéndase por *nodos diferenciables* la composición de función afín y función no lineal que se aplica componente a componente sobre cada neurona de una misma capa). A modo de demostración de la eficiencia del algoritmo, vamos a hacer un breve cálculo. Supongamos que tenemos una red neuronal con 4 capas de 3 neuronas cada una. En tal caso hay un total de 36 elementos en $\boldsymbol{u}$ teniendo en cuenta las 3 matrices de pesos 3×3 y los tres vectores de sesgos 3×1. El número de operaciones para obtener el valor de una neurona en

función de las de la capa anterior es 7, dado que se realizan 3 multiplicaciones y dos sumas en el producto matriz-vector. Posteriormente se le suma la componente correspondiente del sesgo $\left(W_{ij}^{[l]} y_j^{[l-1]} + b_i^{[l]}\right)$. Finalmente se aplica la función sigmoide. Por tanto en toda la red se realizan $7 \cdot 9 = 63$ operaciones de este estilo (en las tres neuronas de la capa de entrada no se realizan cálculos). En la última capa se resta el vector que representa la clase a la que pertenece el dato inicial (3 operaciones), se calcula la norma L_2 (6 operaciones) y se divide entre $\frac{1}{2}$. En total se realizan 73 operaciones. Suponiendo que la derivada de la función no lineal tiene el mismo coste de evaluación que la propia función y que hallar la matriz traspuesta no tiene coste, el número total de operaciones al aplicar el algoritmo es el siguiente: 9 operaciones para calcular $\lambda^{[3]}$ (el caso $l = L$ del algoritmo) y $7 \cdot 3 = 21$ para calcular $\lambda^{[2]}$ y $\lambda^{[1]}$. A esto se le añaden los cálculos para determinar la derivada parcial respecto a cada peso, con un total de 9 operaciones por matriz. Así se obtienen 78 operaciones. En total, evaluar la red *hacia delante* y *hacia atrás* supone hacer 151 operaciones, que es aproximadamente igual a evaluar la red dos veces.

Hagamos lo mismo con el método de aproximación de la derivada de $\widetilde{\mathcal{J}}$ respecto de uno de los elementos de $\boldsymbol{u}$. Se explicita el caso de uno de los pesos, siendo para los sesgos análogo:

$$\frac{\partial \widetilde{\mathcal{J}}}{\partial W_{ij}^{[k]}} \approx \frac{\widetilde{\mathcal{J}}\left(W_{11}^{[1]}, ..., b_{n_1}^{[1]}, ..., W_{ij}^{[k]} + \varepsilon, ..., b_{n_L}^{[L]}\right) - \widetilde{\mathcal{J}}(\boldsymbol{u})}{\varepsilon} \tag{1.57}$$

(obsérvese que, por comodidad, abusamos de la notación). Esencialmente lo que se hace es evaluar la red dos veces (además de hacer una diferencia y un cociente). Teniendo en cuenta que $\boldsymbol{u}$ tiene 36 elementos, el número de operaciones aumenta a $36 \cdot 73 + 1 = 2701$ solo teniendo en cuenta las asociadas a la evaluación de la red.

Supongamos por último que en lugar de aplicar el algoritmo de la propagación inversa utilizamos la regla de la cadena en $\mathcal{J}(\boldsymbol{y}(\boldsymbol{u}), \boldsymbol{u})$. En este caso se va a dar una idea del orden de magnitud del número de cálculos necesarios. Supongamos que se desea calcular $\frac{\partial \mathcal{J}}{\partial w_{11}^{[2]}}$ en la red del ejemplo. En tal caso debe tenerse en cuenta sobre qué elementos de la red actúa. Como las neuronas de cada capa están conectadas a todas las neuronas de la capa posterior, dicho peso afecta a múltiples neuronas.

Por ese motivo, el cálculo de esa parcial dará lugar a

$$\frac{\partial \mathcal{J}}{\partial W_{11}^{[2]}} = \frac{\partial \mathcal{J}}{\partial y_i^{[3]}} \frac{\partial y_i^{[3]}}{\partial z_i^{[3]}} \frac{\partial z_i^{[3]}}{\partial y_1^{[2]}} \frac{\partial y_1^{[2]}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial W_{11}^{[2]}} \tag{1.58}$$

con $i \in \{1, 2, 3\}$. Si consideramos que cada derivada consiste en hacer una operación, son necesarios un total de $9 \cdot 3 = 27$ operaciones más dos sumas, dando un total de 29

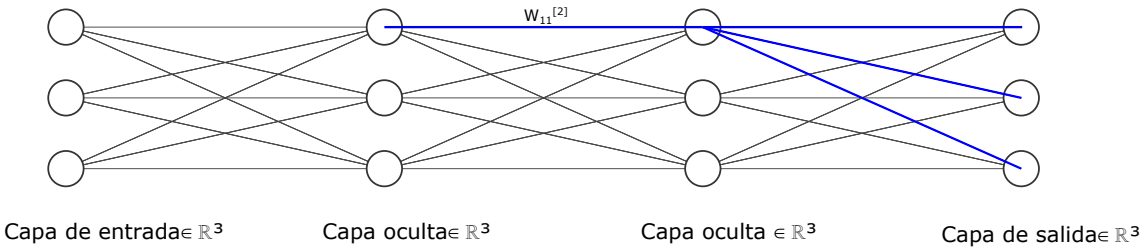

Figura 1.3: Influencia del peso $W_{11}^{[2]}$ en la red [7].

operaciones. Lo mismo sucede para cada sesgo de esa capa. Como $W^{[2]}$ y $b^{[2]}$ contienen 12 elementos, el número de operaciones totales en esa capa es igual a $12 \cdot 29 = 384$.

Si hacemos los cálculos para la capa anterior, aumenta el orden de magnitud de los cálculos. En efecto,

$$\frac{\partial \mathcal{J}}{\partial W_{11}^{[1]}} = \frac{\partial \mathcal{J}}{\partial y_i^{[3]}} \frac{\partial y_i^{[3]}}{\partial z_i^{[3]}} \frac{\partial z_i^{[3]}}{\partial y_j^{[2]}} \frac{\partial y_j^{[2]}}{\partial z_j^{[2]}} \frac{\partial z_j^{[2]}}{\partial y_1^{[1]}} \frac{\partial y_1^{[1]}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial W_{11}^{[1]}} \tag{1.59}$$

con $i, j \in \{1, 2, 3\}$. Para cada i, j fijados, hay 13 operaciones a realizar. Como hay 9 combinaciones de esos subíndices diferentes se tienen 9 sumandos diferentes. Por tanto, para $\frac{\partial \mathcal{J}}{\partial W_{11}^{[1]}}$ hay $13 \cdot 9 + 8 = 125$ cálculos a realizar. Finalmente como estas operaciones se deben hacer para un total de 12 parámetros, el número total de operaciones es de $12 \cdot 125 = 1500$. En realidad el resultado es ligeramente inferior porque algunas de las parciales para pesos y sesgos se repiten. Sin embargo el número real sigue siendo muy elevado en comparación con el obtenido con el algoritmo de la propagación inversa. Tengamos en cuenta además que la red empleada es bastante sencilla y por tanto la diferencia en el número de operaciones es relativamente pequeña, incrementándose para las redes presentadas en apartados posteriores.

1.2.3. Aspectos técnicos de la implementación

Antes de presentar los resultados conviene comentar algunos aspectos de la implementación de la estructura y aprendizaje de RNA en el lenguaje de Matlab.

Nuestro objetivo es obtener y almacenar el juego de pesos y sesgos óptimo, $\boldsymbol{u}_{opt}$, asociado a la red definida por el usuario antes de su aprendizaje. Para ello se usa el método del gradiente estocástico o mini-batch, además del algoritmo de la propagación inversa. De acuerdo con (1.55), el cálculo del gradiente requiere del conocimiento de los valores de $y^{[l]}$ y $z^{[l]}$ $(l = 1, 2, ..., L)$. Además cada elemento de $\boldsymbol{u}$ puede ver modificado su valor en cada iteración del método de gradiente empleado. Conocer estos hechos de antemano permite estructurar el programa de aprendizaje de una forma clara y ordenada.

Como descripción general del programa se puede decir que presenta tres conjuntos de códigos: el programa principal, la función de datos de entrada y la función de activación. El primero se encarga de crear las estructuras para almacenar la red, de llamar a la base de datos (imágenes) y de aplicar el algoritmo y métodos mencionados. También realiza comprobaciones periódicas sobre el grado de aprendizaje de la red informando al usuario por pantalla, además de almacenar datos de interés para un posterior análisis (si el lector desea ver el código, vaya al apéndice).

El segundo código contiene los datos esenciales para la construcción de la red y su aprendizaje. En la literatura suelen ser llamados *hiperparámetros*, distinguiéndose de los parámetros de la red (pesos y sesgos) por el hecho de ser manipulables por el usuario. Dichos datos son: número de neuronas por capa y número de capas internas, razón de aprendizaje (μ), factor del regularizador (η), valor del mini-batch, número de epochs y reducción de la razón de aprendizaje. Esto último hace referencia a la reducción de μ en base a algún criterio (número de epochs, precisión alcanzada, etc.) con el objetivo de reducir el efecto de "escape" del gradiente estocástico o mini-batch.

El tercer código contiene la expresión de la función sigmoide.

Continuemos ahora con la base de datos (en este caso, imágenes). La red necesita información de la que aprender, a ser posible extensa y fácil de manejar. Para el caso objeto de estudio, haremos uso de la base de datos MNIST [8]. Se trata de un conjunto de 70000 imágenes de números de tamaño 28×28. 60000 de ellas son usadas para el entrenamiento de la RNA, mientras que las 10000 restantes se usan como test de evaluación de la capacidad de generalización de la red. Es importante no mezclar estos dos subconjuntos de imágenes para así obtener un porcentaje de acierto fiable. Como ya se dijo en la introducción, los números están en blanco y negro y los píxeles (los 28×28 números que conforman cada imagen) tienen un valor entre 0 (negro) y 1 (blanco). Se introducen en la red en forma de vector 784×1, de manera que la primera capa de la RNA tiene 784 neuronas.

Además de la introducción de esta información, deben definirse unos valores iniciales para los pesos y sesgos de la red de la manera que indicamos en (1.13). Este paso resulta crucial a la hora de conseguir que una red aprenda. Definamos por ejemplo todos los pesos y sesgos iguales a cero y pensemos en cómo se traduce en el algoritmo de la propagación inversa (1.55): en la primera iteración, todos los valores de $\lambda^{[l]}$ (salvo el caso $l = L$) son nulos y por tanto también lo serán las componentes de $\frac{\partial}{\partial W^{[l]}}(\widetilde{\mathcal{J}})$ y $\frac{\partial}{\partial b^{[l]}}(\widetilde{\mathcal{J}})$. En iteraciones posteriores, los cambios en los valores de pesos y sesgos son pequeños, de manera que se compromete el aprendizaje de la RNA, o al menos se ralentiza.

Finalmente cabe mencionar cómo se programa el cálculo del gradiente. Dado que el gradiente de cada ${}^{i}\widetilde{\mathcal{J}}$ es independiente del resto de los que conforman el mini-batch, po-

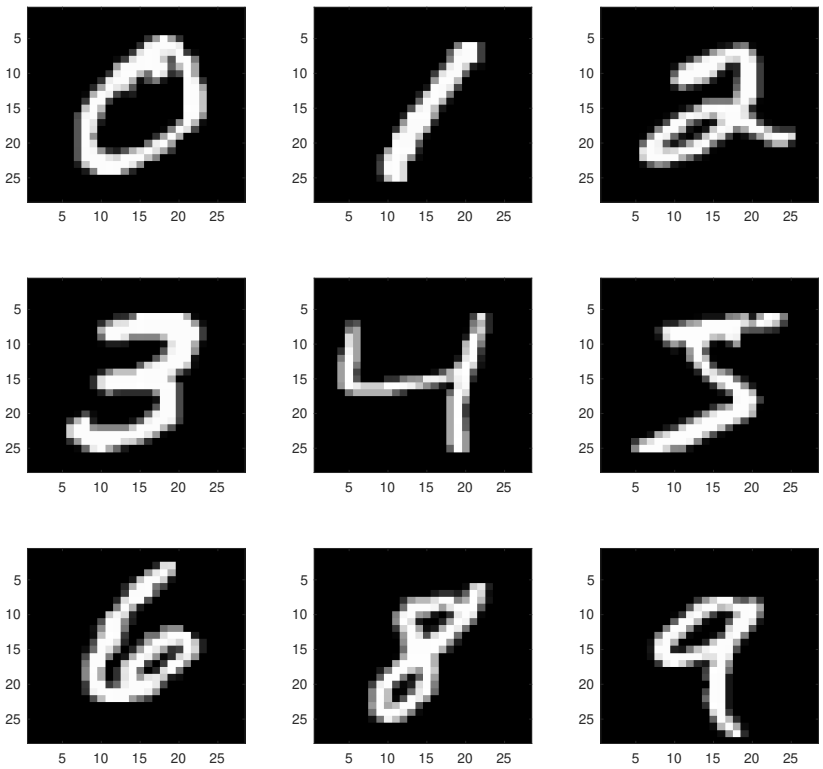

Figura 1.4: Ejemplo de imágenes del banco MNIST [8].

demos paralelizar su cálculo. Esto resulta sencillo por la capacidad de vectorización de operaciones de Matlab.

1.3. Resultados numéricos

En este apartado se va a hacer un estudio detallado del funcionamiento de RNA capaces de clasificar números del 0 al 9 analizando todos los parámetros y problemas presentados: tamaño del mini-batch, presencia o no del regularizador, correcta inicialización, etc. Para aligerar la notación a la hora de describir las características de la red a la que corresponde cada gráfico, vamos a emplear la siguiente notación:

$$[n_0, n_2, ..., n_L] \text{ para las dimensiones de las capas interiores de la RNA} \tag{1.60}$$

$$\{\#\text{ epoch, mini-batch}, \mu, \eta, \text{reducción}\} \text{ para el aprendizaje de la RNA} \tag{1.61}$$

donde *reducción* hace referencia al valor que se sustrae al paso inicial de acuerdo a un criterio definido de antemano, como superar una determinada precisión de clasificación o

por iteración. En ese segundo caso debe prestarse atención para no tener valores $\mu < 0$ tras un cierto número de iteraciones.

1.3.1. Análisis de resultados para una RNA concreta

Primero vamos a estudiar la red con la que más porcentaje de acierto se ha conseguido, tratando de comprobar las hipótesis iniciales lanzadas acerca del funcionamiento de una RNA genérica. Las características de dicha red son $[784, 100, 100, 10]$ y $\{40, 10, 1, 0.75, 0.5\}$. En este caso, la reducción del ratio de aprendizaje se aplica una sola vez en el epoch en el que se alcanza el 98 % de precisión. Recordemos que una red no es más que una función cuyos parámetros se desean optimizar. Concretamente son 89610 parámetros y el tiempo empleado para su optimización es de $97s$.

Para visualizar el comportamiento de la red sobre las 10000 imágenes de test se usa una matriz de confusión (cuadro 1.1). Este tipo de matrices debe analizarse de la siguiente forma: las columnas hacen referencia a la clase real del número analizado por la red. Las filas hacen referencia al valor que devuelve la red. Si la red acierta todos los números que analiza, entonces la matriz es diagonal. En caso contrario, habrá elementos fuera de la diagonal no nulos. Es importante resaltar que no tiene por qué ser simétrica. Además, cada elemento de la matriz presenta dos valores, el absoluto y el porcentaje respecto al total por columna.

La red tiene una precisión del 98.27 % (promedio de las precisiones por clase). Analizando cada clase por separado (cada número), dicha precisión varía. Los números que más acierta son el 0 y el 1 (en ambos supera el 99 %) y el que menos el 7 (la precisión desciende al 97.2 %). Lo más interesante de la matriz es no centrarse en, por ejemplo, el porcentaje asociado al 9, sino analizar con qué números suele confundirlo la red. En este caso, domina la confusión con el 4, no sucediendo lo mismo cuando clasifica el número 4 y lo confunde con el 9 (de ahí que la matriz no sea simétrica). Lo mismo sucede al clasificar el número 7 y confundirlo con el 1.

Pese a que solamente es una máquina, se puede apreciar que los errores dominantes para cada clase son "humanamente comprensibles". Es decir, si se piensa un poco, es razonable que cualquier persona pueda llegar a cometer errores como los de la figura 1.5. Por tanto puede decirse que el tratar de imitar el proceso de asociación de significante (imagen del número) con significado (nombre del número) por parte de un ordenador se ha conseguido.

Una vez hemos visto qué devuelve la red, vamos a ver si la hipótesis realizada respecto a cómo razona en cada capa neuronal es correcta. Empleando solamente la matriz de

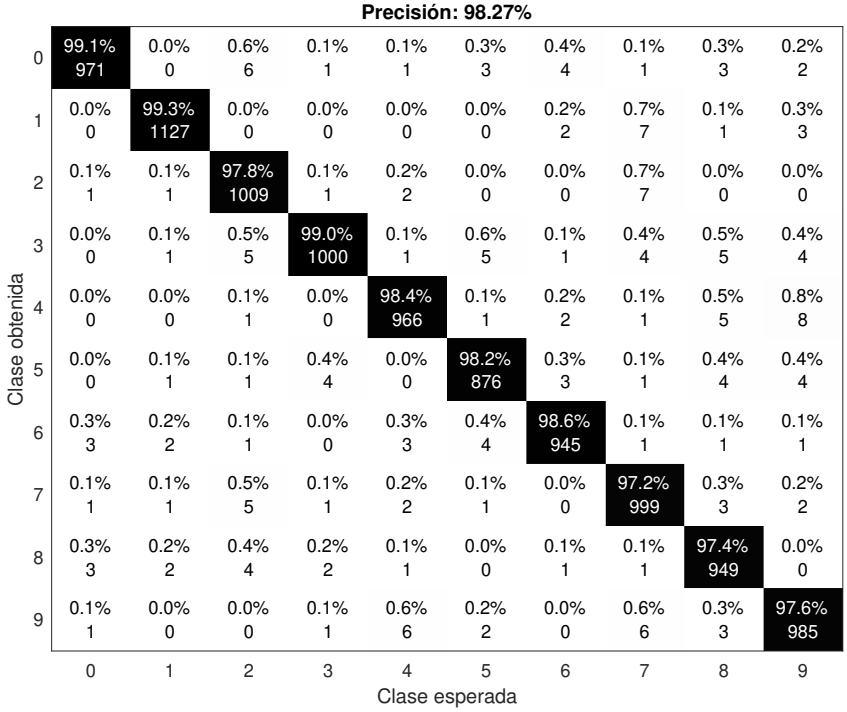

Tabla 1.1: Matriz de confusión asociada a la red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ (matriz conseguida con el código de la referencia [25]).

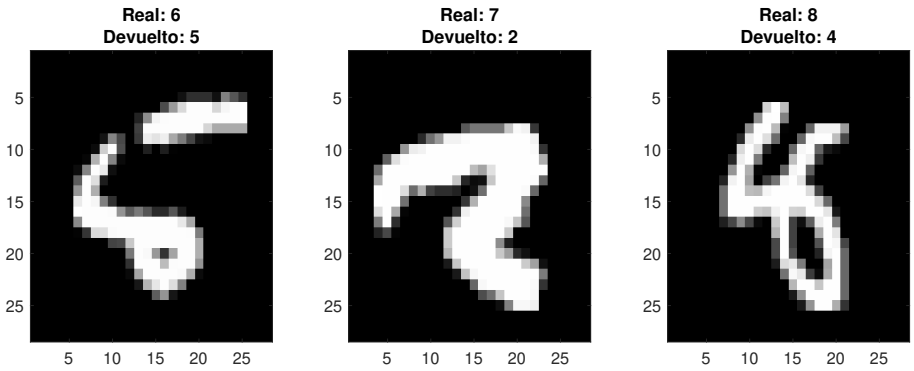

Figura 1.5: Ejemplos de números que la red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ no clasificó adecuadamente. Imágenes de [8].

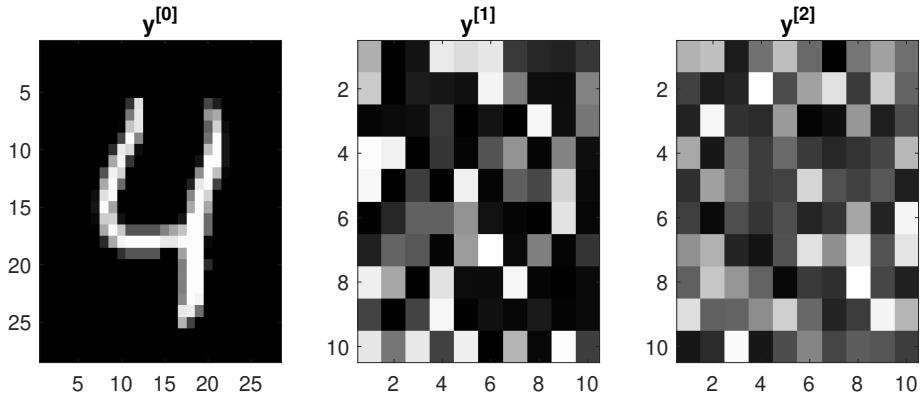

Figura 1.6: Valores de las neuronas en cada capa con la red red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$.

confusión, no puede descartarse que la red comience buscando líneas y curvas y luego las vaya juntando para terminar construyendo los números. Por ejemplo, la construcción del 8 y el 0 es similar (se dibujan con dos y un círculo, respectivamente), de manera que es factible que la red pueda confundirlos. Para ver qué sucede en las capas internas, se hacen dos análisis. Por un lado se toma uno de los números que sí clasifica correctamente y se representan los valores de las neuronas de capas internas como una imagen. Por otro lado se interpreta qué es lo que puede estar detectando la matriz de pesos de la primera capa [26].

Como cabía de esperar, las neuronas de una misma capa no dibujan ningún patrón reconocible. Su función se limita a devolver un número más cercano a 1 o a 0 según si el criterio que cada una tiene asociado se cumple más o menos (lo veremos más claro después del otro estudio).

El segundo estudio cualitativo se centra en la propia red y no en el valor de las neuronas debido a la imagen introducida. Dada una neurona i de la segunda capa, esta está conectada a todas las de la capa de entrada y los pesos que las relacionan son los de la fila i-ésima de la matriz $W^{[1]}$. Veamos entonces que el valor de dicha neurona es la respuesta a la interacción de la fila i-ésima de $W^{[1]}$ con la imagen. En la figura 1.7 se puede observar el resultado de reordenar 3 de esas filas. Efectivamente es posible interpretarlas como filtros que detectan distintos patrones en imágenes. Por ejemplo, la figura izquierda parece reconstruir el segmento más pequeño de los dos que suelen conformar la figura "1" y la del medio una parte del "4". Por supuesto existen otras filas de $W^{[1]}$ para las que no es posible identificar un patrón concreto. Tomando los valores de las neuronas de la primera capa asociadas a las filas de $W^{[1]}$ dibujadas en la figura 1.7, comprobamos que estas funcionan

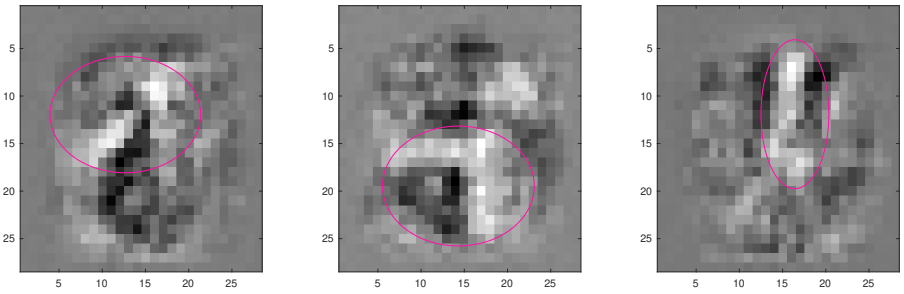

Figura 1.7: Interpretación de las filas 40, 47 y 53 de $W^{[1]}$ como filtros de las imágenes. Las elipses indican los patrones que parecen detectar en las fotos analizadas por la red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$.

como habíamos supuesto: $y^{[1]}_{40} = 0.40$, $y^{[1]}_{47} = 0.99$ e $y^{[1]}_{53} = 0.86$. La neurona que más se "activa", por ejemplo, es la asociada al filtro con el perfil más similar a un "4" (la 47). Omitimos un análisis de la siguiente capa por no ser tan visual como el de la capa mostrada (pensemos que en esa siguiente capa $W^{[2]}$ ya no pondera la información de la imagen de partida, sino el valor de las neuronas de la primera capa).

Dejando a un lado el análisis de la red desde la perspectiva de su utilidad, podemos ver cómo ha cambiado su estructura interna entre su inicialización y final del aprendizaje. Para ello analizaremos la evolución en los valores de los pesos y sesgos. La inicialización se hace de acuerdo con (1.13), de manera que en la figura 1.8 (parte superior) se empieza con una distribución uniforme centrada en el 0 en cada una de las matrices ($W^{[1]}$, $W^{[2]}$ y $W^{[3]}$). La distribución final de los valores deja de ser uniforme (figuras 1.8 -parte inferior- y 1.9).

Globalmente se concluye que los valores máximos y mínimos se alejan del 0, si bien la separación es mayor cuanto más próxima sea la matriz a la capa final de la red. Además se aprecia un dominio de los colores asociados a valores próximos al 0. El comportamiento de los sesgos es similar (figura 1.10).

Finalmente se presenta la evolución del aprendizaje de la red a través del valor de la función coste por epoch. Se hace para las imágenes usadas para el entrenamiento y para las usadas en el testeo. En la figura 1.11 se muestra la evolución tanto de la función coste como del porcentaje de acierto. Debemos apreciar tres características del proceso de aprendizaje: primero, el haber inicializado los pesos y sesgos de forma adecuada permite obtener un valor de la precisión y de la función coste realmente buenos para un solo epoch. Segundo, la precisión de la red es mucho mayor al emplearla con las imágenes del entrenamiento (99.5 %). Lo mismo pasa con la función coste (que vale más del doble para

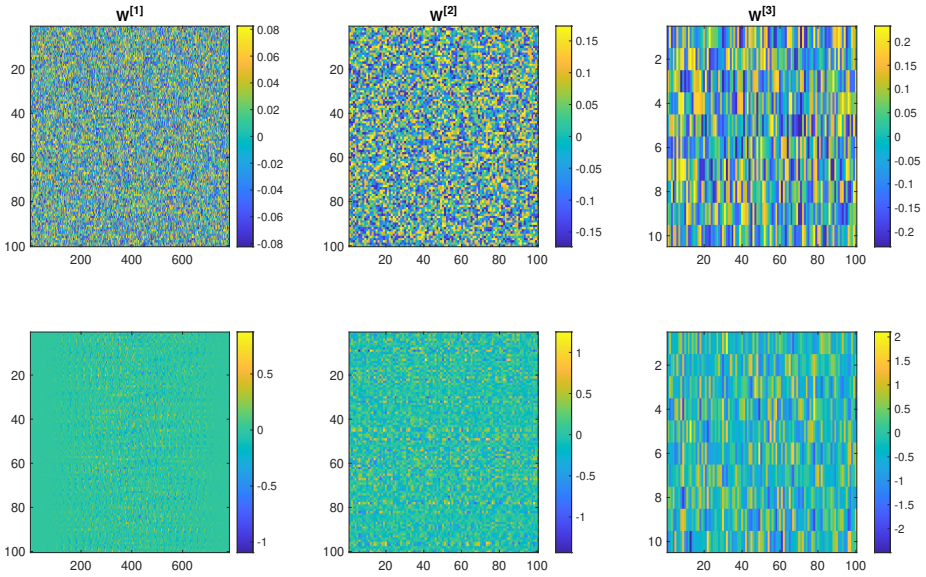

Figura 1.8: Valores de los pesos de la red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ al inicio y final del aprendizaje.

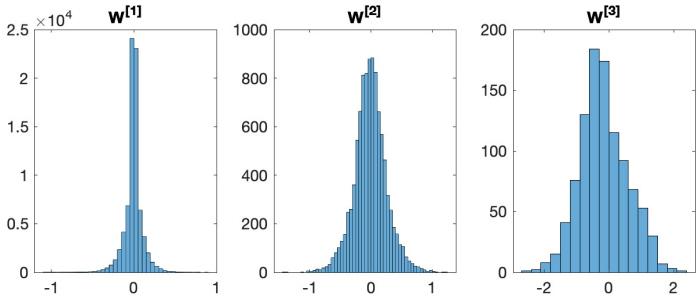

Figura 1.9: Histogramas de los pesos de la red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ tras el aprendizaje. Nótese que se partió de una distribución uniforme centrada en el 0 en todos los casos.

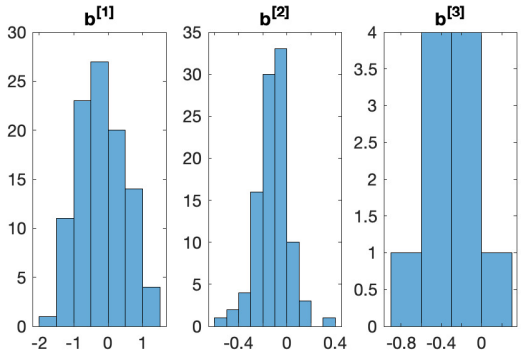

Figura 1.10: Histograma de los sesgos de la red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ tras el aprendizaje. Nótese que todos los valores comenzaron siendo iguales a 0.

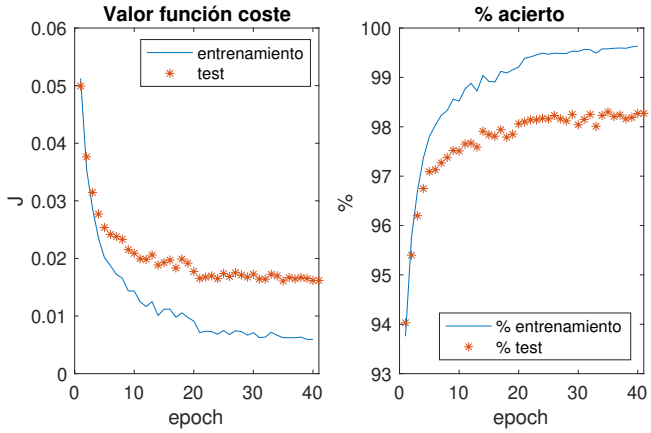

Figura 1.11: Evolución del valor de $\widetilde{\mathcal{J}}$ y de la precisión durante el aprendizaje de la red de características $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$.

el caso de las imágenes del test). Tercero, la evolución no es monótona, existen epochs en los que la función coste aumenta su valor, coincidiendo con disminuciones en el porcentaje de acierto de la red.

Una vez estudiada a fondo esta red, procedemos a realizar modificaciones en los parámetros que definen su estructura o aprendizaje para mostrar la influencia de cada uno de ellos sobre la RNA final. En la medida de lo posible, se tratará de comparar los cambios con esta red.

Estructura	$[100, 100]$	$[50, 50]$	$[150, 150]$	$[200]$	$[50, 50, 50, 50]$
Núm. parám.	89610	42310	141910	158010	47410
Precisión (%)	98,27	97,49	98,15	98,25	96,57
Tiempo (s)	97	67	117	131	97

Tabla 1.2: Estudio del número de capas y neuronas sobre la precisión y tiempo de computación. Se omite la información de las capas externas (784 y 10 neuronas, respectivamente). En todos ellos se usó $\{80, 10, 1, 1.25 \times 10^{-4}, 0.5\}$.

1.3.2. Dependencia en los hiperparámetros de una RNA

1.3.2.1. Número de capas y neuronas

Respecto al número de capas y neuronas empleado no existe un método definido [6]. Según el problema se busca una estructura para la que exista un equilibrio entre precisión y coste computacional. En [8] pueden encontrarse enlaces a distintos tipos de redes para los que se alcanzan valores inferiores y superiores al porcentaje de acierto de nuestra mejor red.

En nuestro caso se entrenan otras cuatro redes para estudiar la influencia de estos hiperparámetros. En las dos primeras se modifica el número de neuronas, dejando fijo el número de capas totales y en las otras dos se fija el número de neuronas y se modifica el número de capas internas.

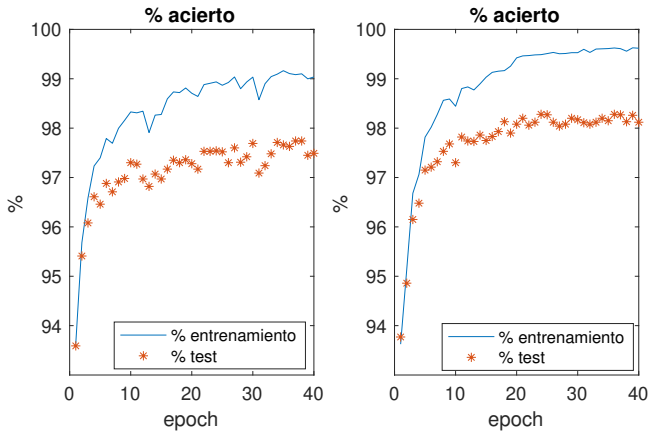

Figura 1.12: Precisión de dos RNA de características $[784, 50, 50, 10]$ (izquierda) y $[784, 150, 150, 10]$ (derecha). Se distinguen en el número de neuronas por capa interna. Ambas se obtienen con $\{40, 10, 1, 1.25 \times 10^{-4}, 0.5\}$.

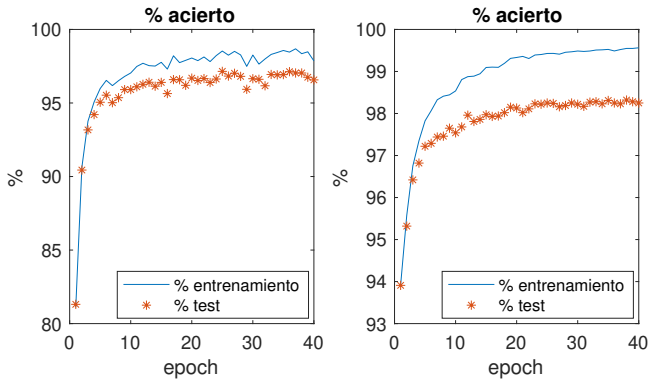

Figura 1.13: Evolución de la precisión de dos RNA de características $[784, 50, 50, 50, 50, 10]$ (izquierda) y $[784, 200, 10]$ (derecha). Se distinguen en el número de capas internas. Ambas se obtienen con $\{40, 10, 1, 0.75, 0.5\}$.

En el caso de emplear distinto número de neuronas (figura 1.12) se observa que el inicio del aprendizaje es similar y que la precisión sobre las imágenes del entrenamiento y del test es la misma. No obstante, a partir de los 10 epochs ambas precisiones se separan, además de tomar distintos valores para cada red. De acuerdo con el cuadro 1.2 el aumento del número de neuronas por capa representa un aumento del tiempo de ejecución, pero no necesariamente un incremento en la precisión de la red (al menos con el resto de hiperparámetros fijados).

Es más sencillo variar el número de capas (figura 1.13) para observar diferencias entre las redes resultantes. Frente al caso anterior, en este sí que hay un inicio en la precisión distinto (hay una diferencia de más del 10 %). Si observamos nuevamente el cuadro 1.2, se concluye que, para un número fijo de neuronas, precisión y tiempo de aprendizaje dependen de cómo se distribuyan estas en las capas internas.

Como conclusión de este apartado debe quedarnos claro que si bien no hay un criterio que determine cuál es la estructura óptima de la red, es posible establecer cotas en el número de capas y neuronas para una base de imágenes fijada. En este caso podemos decir que no se espera una mejora significativa para más de dos capas internas de la red ni más de 200 neuronas en ellas.

1.3.2.2. Número de epochs

Otro aspecto a analizar es la ya mencionada diferencia entre la precisión de las redes sobre las imágenes del aprendizaje y del test. Idealmente ambas precisiones deberían coincidir en cada epoch. No obstante se puede ver que en todas las gráficas mostradas no sucede eso. La forma más sencilla de reducir esta separación es reducir el número de epochs. Esto solo

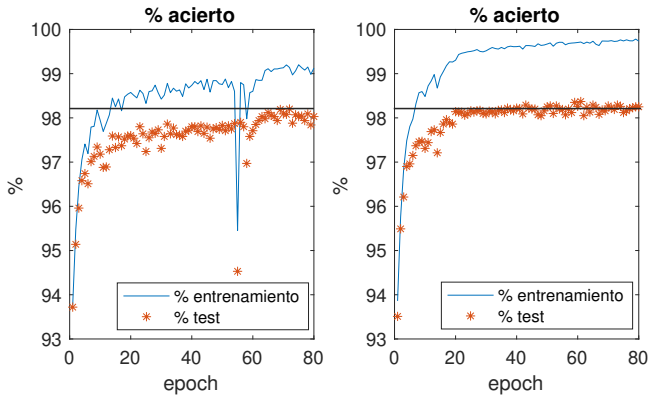

Figura 1.14: Evolución de la precisión de dos RNA de características $\{80, 10, 1; 3.33 \times 10^{-4}, 0.5\}$ (izquierda) y $\{80, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ (derecha). Ambas se obtienen con $[784, 100, 100, 10]$.

debe hacerse si claramente la precisión en las imágenes del test se estabiliza o disminuye mientras que en las imágenes del aprendizaje sigue aumentando. Por ejemplo, en el estudio de la red inicial se definieron 40 epochs porque para valores mayores el comportamiento de la precisión deja de aumentar. En la figura 1.14 se puede ver cómo en la gráfica derecha la precisión del test se estabiliza mucho antes de terminar el aprendizaje (de manera que a partir del epoch 20 podría haberse interrumpido el entrenamiento de la red).

1.3.2.3. Razón del regularizador

Otra forma de controlar este problema es a través del hiperparámetro η. Cuanto más grande sea su valor, más cercanas serán sendas precisiones entre sí por epoch. En la figura 1.14 se puede ver cómo el cambio de su valor de 3.33×10^{-4} a 1.25×10^{-4} aumenta la diferencia entre ambas precisiones del 1 al 1.5 %, pero a su vez mejora la precisión de la red para ambos conjuntos de imágenes. El objetivo es encontrar un valor que permita una evolución en las precisiones más o menos simétrica sin que ello implique una reducción de sus valores. En la figura citada, la evolución es más similar en la gráfica con mayor valor de η, si bien tal vez sea más conveniente usar el otro valor y parar el entrenamiento en el epoch 20.

1.3.2.4. Tamaño del mini-batch

Analicemos la influencia del valor de m. Hasta ahora se usó $m = 10$, que se corresponde con el 0.017 % del tamaño de la base de datos. Usamos ese valor y no otro porque fue el

que permitió obtener la red con la mejor precisión de todas y nos propusimos analizar cada hiperparámetro dejando fijos el resto de ellos.

m (% de 60000)	0.0017	0.017	0.17	1.7	17	100
Precisión test (%)	92.11	98.27	96.24	90.14	90.84	89.50
Tiempo (s)	675	97	38	34	401	833

Tabla 1.3: Estudio del valor de m sobre la precisión y tiempo de computación. Todas las redes se calcularon con $[784, 100, 100, 10]$ y $\{40, 10, 1, 1.25\times10^{-4}, 0.5\}$ salvo en la penúltima, $\{500, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ y última, $\{1000, 10, 1, 1.25 \times 10^{-4}, 0.5\}$.

Como podemos observar en la tabla 1.3, tanto el coste computacional como la precisión son especialmente sensibles al valor de m. El gradiente estocástico ($m = 1$) proporciona una precisión superior al 92 % en 40 epochs frente al gradiente mini-batch con $m = 10000$ (17 % de 60000 imágenes), que requiere de 500 epochs para superar el 90 % de aciertos. Sin embargo el coste computacional es mucho mayor en el primer caso debido a que no se puede aprovechar la paralelización del programa (porque en cada iteración solo se usa una imagen). Por otro lado, el uso del método del gradiente batch provoca una ralentización importante del aprendizaje de la red. El menor coste por iteración no compensa al coste derivado de la necesidad de realizar más epochs (hasta 1000) para alcanzar una precisión sobre las imágenes del test cercana al 90 %.

Se puede concluir que el gradiente mini-batch es el método óptimo que garantiza un equilibrio entre precisión final de la red y tiempo necesario para el entrenamiento de la misma.

1.3.2.5. Inicialización de pesos y sesgos

Como último estudio de estas redes se vuelve sobre el ejemplo de la inicialización de todos los pesos y sesgos a 0.

La figura 1.15 pone de manifiesto la importancia de una inicialización adecuada. Vemos que los porcentajes de acierto de imágenes del aprendizaje y del test no alcanza el 70 % e incluso descienden ligeramente entre los epochs 20 y 80. A nivel de precisión, la diferencia ya es manifiesta desde el primer epoch. Frente a un valor superior al 90 % en redes previas, en este caso se parte del 20 %, solo un 10 % más que la precisión asociada al etiquetado aleatorio de las imágenes.

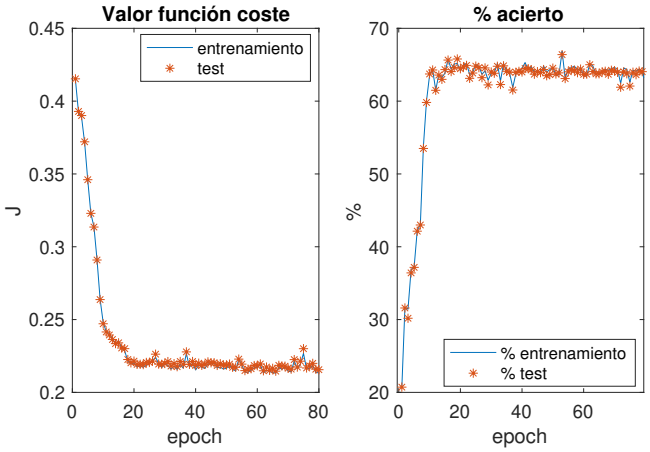

Figura 1.15: Evolución de la precisión y función coste de una RNA con características $\{80; 10; 1; 0.75; 0.5\}$ y $[784, 100, 100, 10]$ con los pesos inicializados a cero.

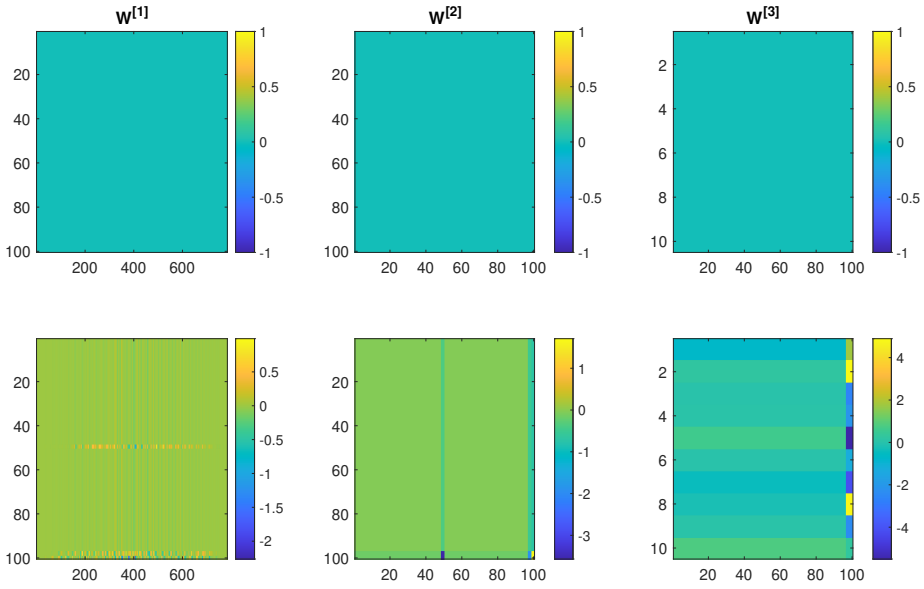

Figura 1.16: Evolución de los pesos inicializados a cero de una RNA con características $\{80, 10, 1, 1.25 \times 10^{-4}, 0.5\}$ y $[784, 100, 100, 10]$.

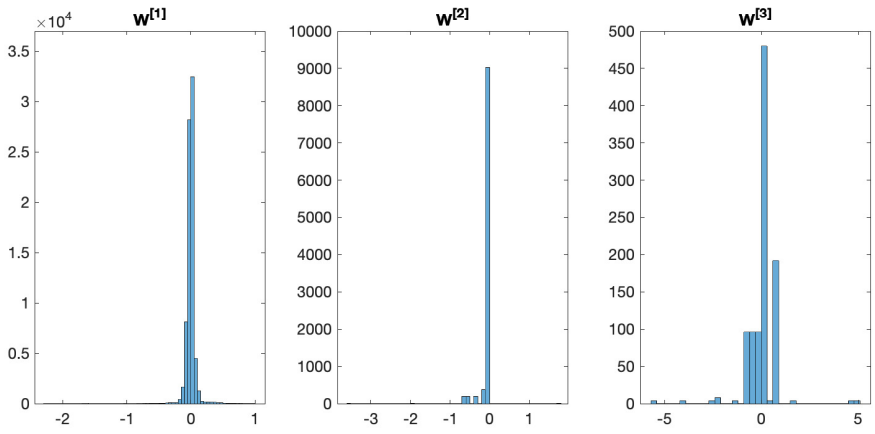

Figura 1.17: Histograma de los pesos finales de una RNA con características $\{80, 10, 1; 1.25 \times 10^{-4}, 0.5\}$ y $[784, 100, 100, 10]$ inicializados a cero.

Si observamos las figuras 1.16 y 1.17, vemos una clara diferencia en el resultado final de los pesos en comparación con los obtenidos en las figuras 1.9 y 1.8. En este segundo caso todos los pesos evolucionan de forma similar (los cuadros que representan cada matriz de pesos son prácticamente monocromáticos) siendo los valores máximos y mínimos mayores en valor absoluto que en el caso de la figura (1.8) para $W^{[3]}$.

Se descarta un estudio del valor de la razón de aprendizaje por no aportar información nueva. Sí que es cierto que una vez estabilizada la precisión de la red, las fluctuaciones son mayores cuanto mayor sea el valor de μ, de ahí que en la red del principio del capítulo se redujera su valor una vez alcanzado el 98 % de precisión sobre las imágenes del test.

En el caso de desear realizar su propio estudio, el lector debe tener en cuenta que lo más probable es que obtenga resultados ligeramente distintos dada la inicialización aleatoria de los pesos. Por otro lado, si llegara a usar la red entrenada con imágenes MNIST para clasificar imágenes propias, observará un más que probable empeoramiento de su precisión. Esto se debe a que las imágenes del banco de datos han sido preprocesadas, de manera que todas están normalizadas (las imágenes originales no tienen los valores de los píxeles entre 0 y 1, sino entre 0 y 255) y centradas en un recuadro de 28×28 píxeles.

El primer problema puede solucionarse de forma más o menos satisfactoria a partir de comandos propios del procesamiento de imágenes de Matlab. Para resolver el segundo problema será necesario redefinir la estructura de la red neuronal. Antes de eso, reflexionemos acerca de por qué surge este problema: en general no todas las personas dibujan un número cualquiera de la misma forma. Dado un papel, no todos colocarán el número en el mismo sitio. Por ejemplo, pensemos en el número 1. ¿Qué significa *centrar el 1* en un

folio? Para algunos, significará colocar su segmento vertical en el centro. Para otros, será colocarlo ligeramente a la derecha del centro. Y según vimos en la figura 1.7, los filtros que contiene la matriz $W^{[1]}$ no responderán de la misma forma a sendas colocaciones del número, pues tienen en cuenta la propia posición de este en la imagen. Lo mismo sucede con el "tipo" de número que dibuje la persona (el número 1, por ejemplo, puede dibujarse con y sin "base" horizontal). Es cierto que usar un banco de imágenes grande reduce este problema, pero no lo resuelve.

Por otro lado, intentemos aplicar esta red para clasificar imágenes ya no de signos monocromáticos que podemos escribir en un papel, sino de objetos a color reales (un perro, un coche). Si entendemos las imágenes como unidades de almacenamiento de píxeles con un ancho, un alto y una profundidad (igual a 1 si son en blanco y negro e igual a 3 si son a color), podemos ver cómo la dimensión de la matriz $W^{[1]}$ se dispara (recordemos que su número de columnas es igual al número de neuronas de la capa de entrada, habiendo tantas de estas como píxeles tenga la imagen). El aumento de parámetros de la red, unido a la mayor complejidad de las imágenes de objetos, hace de la RNA una estructura poco útil, viéndose sustituida por otra, la Red Neuronal Convolucional, diseñada específicamente para superar estos problemas.

Capítulo 2

Redes Neuronales Convolucionales

En este segundo capítulo se desarrolla la teoría fundamental de las Redes Neuronales Convolucionales (RNC). Partiendo de una motivación análoga a la de la introducción, se presentarán los cambios estructurales a realizar en las RNA. También se mostrarán las modificaciones necesarias para adaptar su aprendizaje. Finalmente se presentarán los resultados relativos a la aplicación de estas redes a las clasificación de las imágenes MNIST y a la identificación de diez clases de objetos en imágenes en color.

2.1. Motivación

Para comprender por qué surgen estas redes centrémonos en su objetivo: dada una imagen, deben identificar si en ella hay un objeto perteneciente a una clase de entre varias definidas durante el entrenamiento de la red y en caso de haberlo, clasificarlo adecuadamente. ¿Y qué es una imagen? Para nosotros es un objeto bi o tridimensional que contiene información en forma de números. Será bidimensional si la imagen es en blanco y negro y tridimensional si es a color. En ambos casos, lo que se hace es dividir la imagen en pequeños cuadrados, los píxeles, a cada uno de los cuales se les asigna o bien un número que indica la intensidad de blanco o bien tres números, que indican la intensidad de cada uno de los tres colores primarios. En este segundo caso, la imagen puede reinterpretarse como la superposición de tres imágenes bidimensionales, cada una con la información de uno de esos tres colores (figura 2.1).

Según el número de píxeles de la imagen, esta tendrá más o menos resolución. Las imágenes de MNIST eran de 28×28 píxeles cada una. Las imágenes que realizan las cámaras de foto suelen poseer más de 10^6 píxeles. Es decir, si quisiéramos usar una RNA para clasificar imágenes del orden de la resolución típica de imágenes sin comprimir (sin disminuir previamente su resolución), sería necesario usar una matriz $W^{[1]}$ mucho más

grande. Por ejemplo, en el caso de querer imitar la estructura de la red cuya matriz de confusión se muestra en 1.1, sería necesario pasar de menos de 10^5 parámetros a más de 10^8, tres órdenes de magnitud por encima.

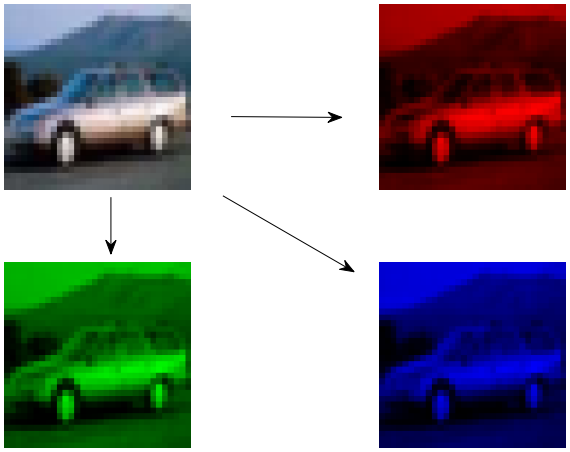

Figura 2.1: Descomposición de una imagen en color 32×32 en cada una de las capas que la conforman. La imagen se almacena en un tensor $32 \times 32 \times 3$ [27].

Analicemos ahora el problema de la posición del objeto. Como dijimos previamente, a la hora de clasificar un número, para la red el número es la imagen. Sin embargo nosotros queremos que detecte el número en la imagen, independientemente de dónde esté. Es decir, queremos que la red analice las formas y figuras de la imagen de manera local, con independencia de dónde se sitúen. Pensemos en cómo razonaría una persona para identificar un coche. De manera similar a como razonamos con números, tendría en mente la forma que para ella posee e iría buscándolo zona por zona en la imagen. Sin embargo debe tener en cuenta que el coche puede no salir entero, ser de juguete o no estar en el plano que esperaba. Podríamos incrementar la dificultad subyacente para identificar un objeto en una imagen añadiendo otros similares al buscado, como se hace en los libros de M. Handford, *"¿Dónde está Wally?"*.

En cualquier caso, lo que debe quedarnos claro es que nuestro modo de proceder es visualizar en la mente la forma del objeto que buscamos y *filtrar* la imagen localmente para determinar si en alguna zona hay una identificación positiva. Desde el punto de vista de una red neuronal, lo que queremos entonces es obtener una estructura que no analice la foto como un todo, sino que la estudie localmente mediante filtros similares a los que contienen las filas de las matrices de pesos, pero de menor tamaño y que proporcionen cierta

invariancia traslacional. Es decir, que reconozcan el mismo patrón independientemente de dónde esté situado en la imagen [28]. Además, dado que la función que construimos a partir de la red neuronal tiene su imagen en $\mathbb{R}^{n_L}$, con n_L el número de clases asignables al objeto de la imagen, conviene que dichos filtros reduzcan la resolución de las fotos.

2.2. Presentación

La operación que cumple estas propiedades es la *convolución*. En su forma matricial, diremos que la operación se realiza entre una matriz de entrada $A \in \mathcal{M}_{m\times m}(\mathbb{R})$ y una matriz $K \in \mathcal{M}_{p\times p}(\mathbb{R})$ con $p \leq m$, llamada filtro o núcleo (ambas son cuadradas por simplicidad). El resultado es otra matriz, $A * K \in \mathcal{M}_{(m-p+1)\times(m-p+1)}(\mathbb{R})$. Cada elemento de $A * K$ se obtiene como

$$(A * K)_{ij} = \sum_{u=1}^{p}\sum_{v=1}^{p} A_{i+u-1,j+v-1}\,(K)_{p-u+1,p-v+1} \tag{2.1}$$

aunque con frecuencia suele sustituirse por la expresión

$$(A * K)_{ij} = \sum_{u=1}^{p}\sum_{v=1}^{p} A_{i+u-1,j+v-1}\,(K^{\pi})_{uv} \tag{2.2}$$

donde $i, j \in \{1, 2, ..., m-p+1\}$ y

$$(K^{\pi})_{uv} = K_{p-u+1,p-v+1} \tag{2.3}$$

es la matriz obtenida a partir de K mediante una rotación de 180^o[1][29].

Nótese que la convolución así realizada es una operación bidimensional. Para que tenga sentido en una imagen a color (que es el resultado de la superposición de 3, como ya vimos), es necesario que el filtro posea tres capas, una para cada una de las imágenes que conforman la original. De esta manera el filtro resulta ser de dimensiones $p\times p\times 3$. Se hacen tres convoluciones, una con cada par bidimensional filtro-imagen y posteriormente se suman las 3 matrices resultantes. De esta manera el resultado final es una matriz bidimensional.

Cada filtro realiza la función equivalente a la de una fila de una matriz de pesos en las RNA (que si recordamos era así como se almacenaban los filtros en esas redes). De la misma manera, podemos aplicar a una misma imagen varios filtros, supongamos Q. De esta forma, sobre una imagen $m \times m \times R$ se pueden aplicar Q filtros $p \times p \times R$ y obtenerse Q nuevas imágenes $(m-p+1)\times(m-p+1)$. Para almacenar tantos filtros e imágenes se hace uso de dos tensores de dimensiones $p \times p \times R \times Q$ y $(m-p+1)\times(m-p+1)\times Q$, respectivamente.

Debemos entender que al aplicar la convolución bidimensional a dos tensores $A \in \mathcal{T}_{m\times m\times R}(\mathbb{R})$ y $K \in \mathcal{T}_{m\times m\times R\times S}(\mathbb{R})$, la expresión final de la convolución será

$$(A * K)_{ijs} = \sum_{w=1}^{R}\sum_{u=1}^{p}\sum_{v=1}^{q} A_{i+u-1,j+v-1,w}\,(K^{\pi})_{uvws} \tag{2.4}$$

donde ahora $(A * K) \in \mathcal{T}_{(m-p+1)\times(m-p+1)\times S}(\mathbb{R})$ y

$$(K^{\pi})_{uvws} = K_{p-u+1,p-v+1,w,s} \tag{2.5}$$

A modo de ejemplo supongamos una "imagen" A a color de dimensiones $5\times5\times3$ a la que se le aplica un filtro K de dimensiones $2\times2\times3$ (o si se prefiere, $2\times2\times3\times1$). Como la tercera dimensión coincide en ambos tensores, podemos calcular la convolución para cada capa de las tres presentes y a continuación sumar las matrices resultantes:

$$\begin{bmatrix} 2 & 2 & 2 & 1 & 1\\ 1 & 5 & 4 & 1 & 3\\ 2 & 3 & 4 & 3 & 3\\ 4 & 3 & 2 & 4 & 1\\ 3 & 5 & 3 & 5 & 2 \end{bmatrix}_{A_1} * \begin{bmatrix} 0 & 1\\ 1 & 0 \end{bmatrix}_{K_1} = \begin{bmatrix} 3 & 7 & 5 & 2\\ 7 & 7 & 5 & 6\\ 7 & 7 & 5 & 7\\ 6 & 7 & 7 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 1 & 1 & 1 & 4\\ 5 & 2 & 5 & 3 & 1\\ 1 & 3 & 5 & 5 & 5\\ 5 & 5 & 3 & 4 & 5\\ 4 & 5 & 5 & 5 & 4 \end{bmatrix}_{A_2} * \begin{bmatrix} 1 & 0\\ 0 & 1 \end{bmatrix}_{K_2} = \begin{bmatrix} 7 & 6 & 4 & 2\\ 8 & 7 & 10 & 8\\ 6 & 6 & 9 & 10\\ 10 & 10 & 8 & 8 \end{bmatrix} \tag{2.6}$$

$$\begin{bmatrix} 4 & 4 & 5 & 3 & 3\\ 4 & 1 & 4 & 2 & 3\\ 2 & 2 & 2 & 4 & 4\\ 4 & 1 & 5 & 4 & 4\\ 1 & 1 & 1 & 1 & 4 \end{bmatrix}_{A_3} * \begin{bmatrix} 0 & 0\\ 0 & 1 \end{bmatrix}_{K_3} = \begin{bmatrix} 4 & 4 & 5 & 3\\ 4 & 1 & 4 & 2\\ 2 & 2 & 2 & 4\\ 4 & 1 & 5 & 4 \end{bmatrix}$$

$$A * K = \begin{bmatrix} 14 & 17 & 14 & 7\\ 19 & 15 & 19 & 16\\ 15 & 15 & 16 & 21\\ 20 & 18 & 20 & 18 \end{bmatrix}$$

La forma de los sesgos también varía en las RNC. En este trabajo se entenderá que cada sesgo es un número asociado a *cada* filtro. Siguiendo con la notación del párrafo anterior, si hay Q filtros, el sesgo será un vector de Q componentes. Cada componente se corresponde con uno de los filtros de R capas. Una vez realizada la operación del ejemplo (2.6), se sumaría dicha componente a todos los elementos de la matriz.

Para terminar se presenta la función no lineal empleada. En este caso se hace uso de la función PReLU en todas las capas salvo en la última,

$$x \longmapsto \text{PReLU}(x) = \text{máx}(x, 0) + 0.01 \cdot \text{mín}(x, 0) \tag{2.7}$$

por garantizar un mejor aprendizaje de la red [30] [31]. Es más conocida la función ReLU, pero experimentalmente hemos comprobado una mejora en la precisión de las redes haciendo uso de esta variante.

A mayores, se aplica otra función que reduce la dimensión de los tensores en cada capa de manera eficaz. Se conoce como *maxpool*. La idea detrás de su aplicación es que la información almacenada en neuronas cercanas es similar, de manera que basta tomar un representante entre ellas. La función divide cada capa del tensor en submatrices de la misma dimensión (en general 2×2) y selecciona de cada una de ellas el valor más alto. Mostramos un ejemplo que facilitará la comprensión de su funcionamiento:

$$\text{maxpool}\left(\left[\begin{array}{cc:cc} 14 & 17 & 14 & 7 \\ 19 & 15 & 19 & 16 \\ \hdashline 15 & 15 & 16 & 21 \\ 20 & 18 & 20 & 18 \end{array}\right]\right) = \begin{bmatrix} 19 & 19 \\ 20 & 21 \end{bmatrix} \tag{2.8}$$

Notemos la importancia de controlar las dimensiones de cada capa del tensor para que la función maxpool funcione adecuadamente (si las caras son de dimensión impar, no se podrá dividir en submatrices 2×2). En ocasiones también se usa la función *average pooling*, que toma el valor medio de cada submatriz.

El uso de estas capas de convolución con (o sin) maxpool está destinado a extraer la información relevante de las imágenes de cara a la clasificación del objeto que contienen. Podemos entender que dicha información se presenta en forma de alas, ruedas, bordes,... y juntas permiten decidir de qué objeto se trata. Esta última operación, la clasificación final del objeto, se realiza con una o varias capas como las descritas en el capítulo anterior. En conjunto, la red resultante tiene el aspecto de la figura 2.2.

El siguiente paso consiste en formalizar una RNC arbitraria. Para su entrenamiento se usa también un método de descenso (gradiente estocástico, por ejemplo) que nos lleva al cálculo de gradientes. Por ello deduciremos las expresiones del algoritmo de la propagación inversa para este tipo de redes. Como se pudo apreciar en la expresión (2.4), la notación puede resultar complicada. Por ese motivo, solamente se deducirán aquellas expresiones del algoritmo distintas a las del caso de RNA.

Sea pues una red RNC compuesta por $L - 1$ capas de convolución y una última capa completamente conectada (como las de la RNA). Supóngase nuevamente un conjunto de

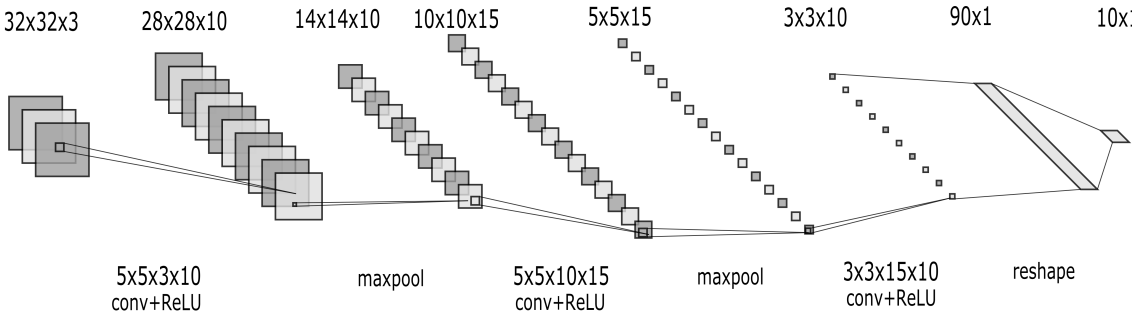

Figura 2.2: Ejemplo de RNC. Inicialmente hay una imagen a color $32 \times 32 \times 3$. En dos capas sucesivas se aplica convolución, activación y maxpooling (2×2). Las dimensiones de los tensores de filtros se indican en la parte inferior de la imagen. En la tercera capa solo se aplica convolución y ReLU (en nuestro caso, PReLU), para luego reordenar el tensor resultante en un vector. La última capa es propia de una RNA, donde la función de activación empleada es la sigmoide. El término *reshape* hace referencia a la transformación del tensor $3 \times 3 \times 10$ en un vector para así situarse en el contexto de RNA.

imágenes que es posible dividir en un determinado número de clases $C = \{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_{n_L}\}$. Dada la imagen $y^{[0]}$, la aplicación (1.5) pasa a ser

$$\begin{array}{rccl} \widetilde{\mathcal{F}}\colon & \mathbb{T}_{n_0 \times n_0 \times d_0}(\mathbb{R}) & \longrightarrow & \mathbb{R}^{n_L} \\ & y^{[0]} & \longmapsto & \widetilde{\mathcal{F}}(y^{[0]}) = \boldsymbol{e}_i,\ si\, y^{[0]} \in \mathcal{C}_i \end{array} \tag{2.9}$$

La función que se puede construir y pretende imitar a $\widetilde{\mathcal{F}}$, $\mathcal{F}$, es igual a la composición de funciones $\mathcal{F}_i$ definidas de forma distinta a (1.7) salvo el caso $\mathcal{F}_L$ (que omitimos por ser igual que en la expresión citada):

$$\begin{array}{rccl} \mathcal{F}_i\colon & \mathbb{T}_{n_{i-1} \times n_{i-1} \times d_{i-1}}(\mathbb{R}) & \longrightarrow & \mathbb{T}_{n_i \times n_i \times d_i}(\mathbb{R}) \\ & y^{[i-1]} & \longmapsto & \mathcal{F}_i(y^{[i-1]}) = y^{[i]} = (\mathcal{H}_i \circ \sigma \circ \mathcal{G}_i)(y^{[i-1]}),\ i \in \{1, 2, ..., L-1\} \end{array} \tag{2.10}$$

donde $\mathcal{G}_i$ viene dada por

$$\begin{array}{rccl} \mathcal{G}_i\colon & \mathbb{T}_{n_{i-1} \times n_{i-1} \times d_{i-1}}(\mathbb{R}) & \longrightarrow & \mathbb{T}_{n'_i \times n'_i \times d_i}(\mathbb{R}) \\ & y^{[i-1]} & \longmapsto & \mathcal{G}_i(y^{[i-1]}) = y^{[i-1]} * (K^{\pi})^{[i]} + \mathbf{1}^{[i]} * b^{[i]} = z^{[i]} \end{array} \tag{2.11}$$

con

$$\begin{aligned}
&(K^{\pi})^{[i]} \in \mathbb{T}_{m_i \times m_i \times d_{i-1} \times d_i}(\mathbb{R}),\ i = 1, 2, ..., L-1 \\
&b^{[i]} \in \mathbb{R}^{d_i},\ i = 1, 2, , ..., L-1 \\
&\mathbf{1}^{[i]} \in \mathbb{T}_{n_i' \times n_i' \times d_i}(\mathbb{R}),\ i = 1, 2, ..., L-1
\end{aligned} \tag{2.12}$$

donde $\mathbf{1}^{[i]}$ tiene todos sus elementos iguales a 1 $\forall i$.

La función σ es la función de activación, que en este caso es la función PReLU. Como ya sucedía con la función sigmoide, debe entenderse como una función que actúa sobre cada neurona (sobre cada elemento del tensor resultante tras aplicar $\mathcal{G}_i$).

Finalmente, la función $\mathcal{H}_i$ viene dada por

$$\begin{array}{rccl}
\mathcal{H}_i\colon & \mathbb{T}_{n_i' \times n_i' \times d_i}(\mathbb{R}) & \longrightarrow & \mathbb{T}_{n_i \times n_i \times d_i}(\mathbb{R}) \\
& \sigma\left(z^{[i]}\right) & \longmapsto & \mathcal{H}_i\left(\sigma\left(z^{[i]}\right)\right) = \text{maxpool}\left(\sigma\left(z^{[i]}\right)\right)
\end{array} \tag{2.13}$$

con $n_i = \frac{n_i'}{2}$ en general. Cabe recordar que esta función actúa capa a capa, de manera que reduce el ancho y largo del tensor tridimensional, no así su profundidad. También debe recordarse que en la práctica es posible no aplicar $\sigma \circ \mathcal{G}_i$ ó $\mathcal{H}_i$ en alguna de las capas.

La formalización de la última capa (la L) es igual que en el caso de las RNA. Como paso previo, en caso de no ser $y^{[L-1]}$ de la forma $1 \times 1 \times d_{L-1}$, se reordenan sus componentes (es decir, debe pasarse de un tensor de dimensiones $n_{L-1} \times n_{L-1} \times d_{L-1}$ a otro $1 \times 1 \times (n_{L-1}^2 d_{L-1})$). De esta manera los elementos con los que se trabaja son por parte de la RNC

$$\begin{aligned}
&z^{[i]} \in \mathbb{T}_{n_i' \times n_i' \times d_i}(\mathbb{R}),\ i = 1, 2, ..., L-1 \\
&y^{[i]} \in \mathbb{T}_{n_i \times n_i \times d_i}(\mathbb{R}),\ i = 0, 1, ..., L-1 \\
&K^{[i]} \in \mathbb{T}_{m_i \times m_i \times d_{i-1} \times d_i}(\mathbb{R}),\ i = 1, 2, , ..., L-1 \\
&b^{[i]} \in \mathbb{R}^{d_i},\ i = 1, 2, ..., L-1
\end{aligned} \tag{2.14}$$

y por parte de la RNA, tomando $p = n_{L-1}^2 d_{L-1}$,

$$\begin{aligned}
&z^{[L]} \in \mathbb{R}^{n_L} \\
&y^{[L]} \in \mathbb{R}^{n_L} \\
&W^{[L]} \in \mathbb{R}^{n_L} \times \mathbb{R}^{p} \\
&b^{[L]} \in \mathbb{R}^{n_L}
\end{aligned} \tag{2.15}$$

Sería posible, como ya dijimos, añadir varias capas propias de una RNA. El programa mostrado en el anexo permite tal cambio, si bien en este desarrollo nos hemos limitado al caso de una sola capa, como también se hace en [9].

Como ya hicimos en el capítulo 1, agrupamos los elementos de tensores, matriz de pesos y sesgos en una variable de control

$$\boldsymbol{u} \equiv \left(K^{[1]}, b^{[1]}, ..., W^{[L]}, b^{[L]}\right) \tag{2.16}$$

perteneciente al conjunto

$$\mathcal{U} = \left\{ \boldsymbol{u} \in \prod_{j=1}^{L-1} \left(\mathbb{T}_{m_j \times m_j \times d_{j-1} \times d_j}(\mathbb{R}) \times \mathbb{R}^{d_j} \right) \times \mathcal{M}_{n_L \times p}(\mathbb{R}) \times \mathbb{R}^{n_L} \right\} \tag{2.17}$$

Análogamente, definimos una variable de estado

$$\boldsymbol{y} \equiv \left(y^{[0]}, z^{[1]}, ..., z^{[L]}, y^{[L]}\right) \tag{2.18}$$

perteneciente al conjunto

$$\mathcal{E} = \left\{ \boldsymbol{y} \in \mathbb{T}_{n_0 \times n_0 \times d_0}(\mathbb{R}) \times \prod_{j=1}^{L-1} \left(\mathbb{T}_{n'_j \times n'_j \times d_j}(\mathbb{R}) \times \mathbb{T}_{n_j \times n_j \times d_j}(\mathbb{R}) \right) \times \mathbb{R}^{n_L} \times \mathbb{R}^{n_L} \right\} \tag{2.19}$$

De esta manera volvemos a situarnos ante un problema de minimización análogo al presentado en la ecuación (1.14).

2.3. Entrenamiento de la red

2.3.1. Algoritmo de la propagación inversa

Para el entrenamiento de la red se usarán nuevamente los métodos del gradiente estocástico y mini-batch. Para el cálculo del gradiente se usará el algoritmo de la propagación inversa adaptado a la estructura de estas redes. Deduciremos aquellas que difieran de las obtenidas en (1.55) haciendo uso de la expresión (1.56). Sea pues $l < L-1$, entonces

$$\begin{aligned} \lambda_{ijk}^{[l]} = \frac{\partial}{\partial z_{ijk}^{[l]}}(\widetilde{\mathcal{J}}) = \frac{\partial \widetilde{\mathcal{J}}}{\partial z_{pqr}^{[l+1]}} \cdot \frac{\partial z_{pqr}^{[l+1]}}{\partial z_{ijk}^{[l]}} = \\ = \frac{\partial \widetilde{\mathcal{J}}}{\partial z_{pqr}^{[l+1]}} \cdot \frac{\partial (\mathcal{G}_{l+1}(y^{[l]}))_{pqr}}{\partial z_{ijk}^{[l]}} = \lambda_{pqr}^{[l+1]} \frac{\partial}{\partial z_{ijk}^{[l]}} \left(y_{p+m-1,q+n-1,c}^{[l]} \left(K^{\pi}\right)_{mncr}^{[l+1]} + b_r^{[l+1]} \right) = \\ = \lambda_{pqr}^{[l+1]} \frac{\partial}{\partial z_{ijk}^{[l]}} \left(\mathcal{H}_l(\sigma(z_{p+m-1,q+n-1,c}^{[l]})) \left(K^{\pi}\right)_{mncr}^{[l+1]} + b_r^{[l+1]} \right) = \\ = \lambda_{i-m+1,j-n+1,r}^{[l+1]} \left(K^{\pi}\right)_{mnkr}^{[l+1]} (\mathcal{H}_l \circ \sigma)'(z_{ijk}^{[l]}) \end{aligned} \tag{2.20}$$

La última expresión requiere de una breve explicación. Por un lado, $(\mathcal{H}_l \circ \sigma)'(z_{ijk}^{[l]}))$ es un tensor de dimensiones $n'_l \times n'_l \times d_l$. Para obtenerlo basta hacer el producto elemento a elemento de los tensores que contienen las derivadas $\mathcal{H}'_l\left(\sigma\left(z^{[l]}\right)\right)$ y $\sigma'\left(z^{[l]}\right)$. La primera de las derivadas depende de si el elemento tomado es el mayor de los cuatro de la submatriz

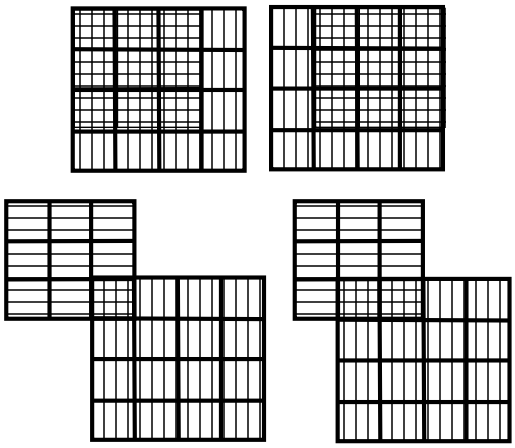

Figura 2.3: Diferencia entre la convolución por capa de la expresión (2.2) y la convolución en la última igualdad de (2.20). En la parte superior se muestra cómo se obtienen el primer y segundo elemento de la matriz convolucionada con "$*$". Su dimensión es 2×2. En la parte inferior se muestra cómo se obtiene la primera y segunda componente de la matriz convolucionada con "$*_c$". Su dimensión es 6×6.

definida durante el pooling a la que pertenece. Si lo es, valdrá 1, si no lo es, valdrá 0. Es decir,

$$\mathcal{H}'_l((M_{2\times 2})_{ij}) = \begin{cases} 1 & \text{si } m_{ij} = \text{máx}\{m_{ab}, a = 1,2; b = 1,2\} \\ 0 & \text{si } m_{ij} \neq \text{máx}\{m_{ab}, a = 1,2; b = 1,2\} \end{cases} \tag{2.21}$$

Para almacenar la información de esa derivada se crea un tensor de la dimensión de $z^{[l]}$. La segunda de las derivadas se define elemento a elemento. Como la función PReLU no es diferenciable en $x = 0$, se fuerza a que en ese punto valga 0.01.

Finalmente debemos analizar el término $\lambda^{[l+1]}_{i-m+1,j-n+1,r}\,(K^{\pi})^{[l+1]}_{mnkr}$. A priori sabemos que $\lambda^{[l+1]}$ y $K^{[l+1]}$ son tensores de dimensiones $n'_{l+1} \times n'_{l+1} \times d_{l+1}$ y $m_{l+1} \times m_{l+1} \times d_l \times d_{l+1}$, respectivamente. La operación entre esos dos tensores es una convolución que denotaremos por $\lambda^{[l+1]} *_c (K^{\pi})^{[l+1]}$, donde $*_c$ indica que la convolución aumenta la dimensión. La mejor forma de entender cómo funciona es a través de la figura 2.3. El resultado es un tensor de dimensiones $(m_{l+1}+n'_{l+1}-1)\times(m_{l+1}+n'_{l+1}-1)\times d_l$ (como el tensor $y^{[l]}$). Si se revisa cómo evolucionan las dimensiones de los tensores, se deduce que $n'_l = 2 * (m_{l+1} + n'_{l+1} - 1)$. Es decir, las dimensiones del tensor $(\mathcal{H}_l \circ \sigma)'(z^{[l]}_{ijk})$ pueden reescribirse como $(2*(m_{l+1}+n'_{l+1}-1)) \times (2*(m_{l+1}+n'_{l+1}-1)) \times d_l$. Esto no significa que la expresión obtenida en (2.20) esté mal (las dimensiones de los dos tensores no coinciden). Simplemente debe entenderse que dado un elemento α del tensor $\lambda^{[l+1]}_{i-m+1,j-n+1,r}\,(K^{\pi})^{[l+1]}_{mnkr}$, tomado de una de las d_l capas, se debe multiplicar por la submatriz cuadrada 2×2 de la misma capa del tensor $(\mathcal{H}_l \circ \sigma)'(z^{[l]}_{ijk})$

de la que se obtuvo la componente de $y^{[l]}$ que ocupa la misma posición que α (porque como dijimos las dimensiones de $y^{[l]}$ y del tensor del que tomamos α, son las mismas). Vamos a representar esa operación con el símbolo "$\circledcirc$". En el caso de no aplicarse en una capa la función maxpool, la expresión se reduce al producto elemento a elemento de dos tensores.

Así, la expresión de $\lambda^{[l]}$ es

$$\lambda^{[l]} = (\lambda^{[l+1]} *_c (K^\pi)^{[l+1]}) \circledcirc (\mathcal{H}_l \circ \sigma)'(z^{[l]}),\ l = 1, 2, ..., L-2 \tag{2.22}$$

Para el caso $l = L-1$, $\lambda^{[l]}$ se obtiene como en el caso de las RNA, si bien puede suceder que haya que reordenar los elementos del producto $(W^{[L]})^t \lambda^{[L]}$ de la expresión (1.55).

La obtención de las derivadas parciales de la función coste respecto de los filtros y sesgos es más sencilla. Fijado $l \in \{1, 2, ..., L-1\}$,

$$\frac{\partial}{\partial b_i^{[l]}}(\widetilde{\mathcal{J}}) = \frac{\partial \widetilde{\mathcal{J}}}{\partial z_{pqr}^{[l]}} \cdot \frac{\partial z_{pqr}^{[l]}}{\partial b_i^{[l]}} = \lambda_{pqr}^{[l]} \frac{\partial}{\partial b_i^{[l]}} \left(y_{p+m-1,q+n-1,c}^{[l-1]} (K^\pi)_{mncr}^{[l]} + b_r^{[l]} \right) = \lambda_{pqi}^{[l]} \tag{2.23}$$

$$\begin{aligned} \frac{\partial}{\partial (K^\pi)_{ijkd}^{[l]}}(\widetilde{\mathcal{J}}) = \frac{\partial \widetilde{\mathcal{J}}}{\partial z_{pqr}^{[l]}} \cdot \frac{\partial z_{pqr}^{[l]}}{\partial (K^\pi)_{ijkd}^{[l]}} = \lambda_{pqr}^{[l]} \frac{\partial}{\partial (K^\pi)_{ijkd}^{[l]}} \left(y_{p+m-1,q+n-1,c}^{[l-1]} (K^\pi)_{mncr}^{[l]} + b_r^{[l]} \right) = \\ = \lambda_{pqd}^{[l]} y_{p+i-1,q+j-1,k}^{[l-1]} \end{aligned} \tag{2.24}$$

Por tanto

$$\frac{\partial}{\partial (K^\pi)^{[l]}}(\widetilde{\mathcal{J}}) = y^{[l-1]} * (\lambda^\pi)^{[l]} \Leftrightarrow \frac{\partial}{\partial K^{[l]}}(\widetilde{\mathcal{J}}) = (y^{[l-1]} * (\lambda^\pi)^{[l]})^\pi \Leftrightarrow \frac{\partial}{\partial K^{[l]}}(\widetilde{\mathcal{J}}) = (y^\pi)^{[l-1]} * \lambda^{[l]} \tag{2.25}$$

En resumen, el algoritmo de la propagación inversa en RNC es como sigue (recuérdese que en todas las capas se usa la función PReLU salvo en la última, donde se sigue usando la sigmoide):

$$\begin{array}{l} \frac{\partial}{\partial K^{[l]}}(\widetilde{\mathcal{J}}) = (y^\pi)^{[l-1]} * \lambda^{[l]} \\ \frac{\partial}{\partial b_s^{[l]}}(\widetilde{\mathcal{J}}) = \sum_{pq} \lambda_{pqs}^{[l]} \\ \frac{\partial \widetilde{\mathcal{J}}}{W^{[L]}}(\widetilde{\mathcal{J}}) = \left(\lambda^{[L]}\right) \left(y^{[L-1]}\right)^t \\ \frac{\partial}{\partial b^{[l]}}(\widetilde{\mathcal{J}}) = \lambda^{[l]} \\ \text{con } \lambda^{[l]} = \begin{cases} \left(\lambda^{[l+1]} *_c (K^\pi)^{[l+1]}\right) \circledcirc (\mathcal{H}_l \circ \sigma)'(z^{[l]}) & \text{si } l = 1, 2, ..., L-2 \\ \left((W^{[l+1]})^t \lambda^{[l+1]}\right) \circledcirc (\mathcal{H}_l \circ \sigma)'(z^{[l]}) & \text{si } l = L-1 \\ \left(y^{[L]} - \widetilde{\mathcal{F}}(y^{[0]})\right) \odot \sigma'(z^{[L]}) & \text{si } l = L \end{cases} \end{array} \tag{2.26}$$

2.3.2. Aspectos técnicos de la implementación

La estructura del programa es similar a la del caso de RNA. Cabe decir que los parámetros de la red se inicializan con la inicialización Xabier, si bien algunos autores recomiendan para el caso de usar la función PReLU la inicialización He [31]. La elección se hace porque experimentalmente se comprueba que no proporciona mejores resultados que la inicialización ya usada en RNA.

Por otro lado, como ya sucedía con el código del capítulo anterior, se vectorizan las operaciones para obtener un programa más eficiente. Nuevamente el lector puede ir al apéndice para ver el código.

2.4. Resultados numéricos

Para el entrenamiento de todas las RNC se hizo uso del clúster Mariscal del grupo de investigación mat+i de la USC.

Con el fin de mostrar la capacidad y limitaciones de estas redes vamos a hacer dos estudios. Por un lado, vamos a analizar la mejor red obtenida de este tipo para clasificar la base de datos MNIST y así poder compararla con la mejor RNA obtenida en el capítulo 1. Por otro lado, presentaremos otras dos redes con las que se buscó clasificar una base de datos de objetos en lugar de números. Con esa parte se busca mostrar que efectivamente estas redes son capaces de clasificar imágenes complicadas y por otro lado que con las técnicas expuestas no es suficiente para conseguir la misma precisión que con las imágenes de números.

2.4.1. RNC para clasificar números

Como ya se hizo en la presentación de resultados de las RNA, en este caso también vamos a presentar los hiperparámetros empleados para entrenar la red. De esta manera, la red que empleamos se definió de la siguiente forma:

$$\begin{gathered} 1^{\underline{\text{a}}} \text{ capa de conv.: } 5 \times 5 \times 1 \times 12 \text{ y maxpool} \\ 2^{\underline{\text{a}}} \text{ capa de conv.: } 5 \times 5 \times 12 \times 12 \text{ y maxpool} \\ 3^{\underline{\text{a}}} \text{ capa (RNA): } 10 \times 192 \end{gathered} \tag{2.27}$$

y $\{22, 20, 0.5, 0.0001, 0.25\}$ (recuérdese la notación $\{\text{epoch, mini-batch}, \mu, \eta, \text{reducción}\}$). Esta red tiene un total de 5854 parámetros, 15 veces menos que la RNA con matriz de confusión mostrada en la tabla 1.1. En contrapartida, el tiempo de entrenamiento es notablemente superior (2.5 horas). Para conseguir alcanzar tal precisión se impuso que el valor del ratio de aprendizaje se redujera de 0.5 a 0.25 cuando la precisión superase el 98.6 %.

Evidentemente, tanto la estructura de la red como la reducción de μ se definieron tras varias pruebas previas.

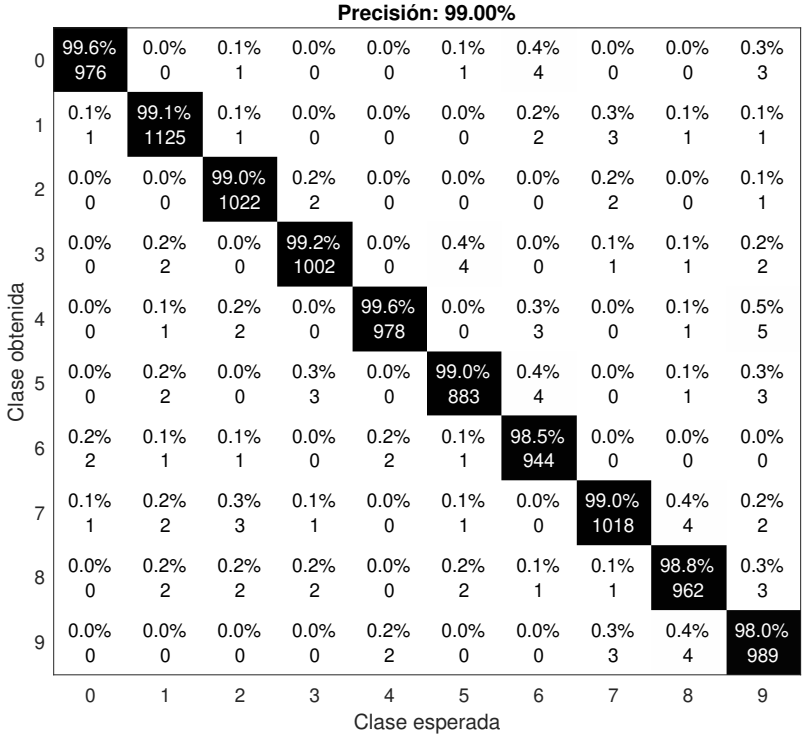

Tabla 2.1: Matriz de confusión asociada a la RNC de características (2.27) y $\{22, 20, 0.5; 0.0001, 0.25\}$ [25].

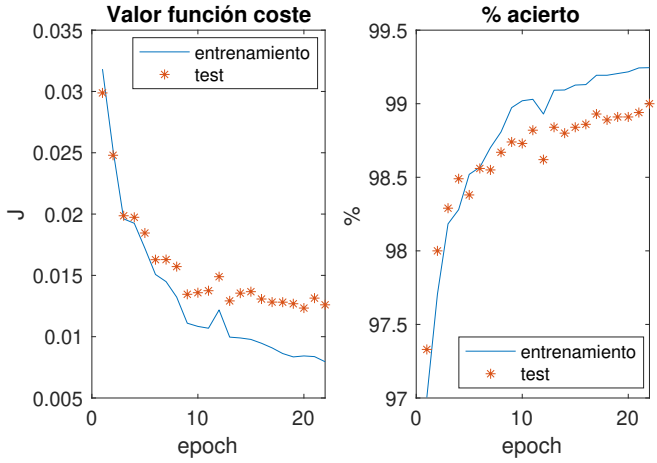

Figura 2.4: Evolución de la función coste y precisión de la RNC de características (2.27) y $\{22, 20, 0.5, 0.0001, 0.25\}$.

La matriz de confusión asociada (tabla 2.1) demuestra una mejora global en la capacidad de clasificación de la red, si bien hay un descenso de la precisión asociada al 1 y al 6 del 0.2 y 0.1 % respectivamente en comparación con la red descrita en la tabla 1.1. Obsérvese por otro lado que a diferencia de la precisión de la RNA para el primer epoch, 94 % (figura 1.1), en este caso es del 97 % (figura 2.4). En esta segunda figura es importante resaltar que la red no sigue aprendiendo a partir del epoch 40, se estabiliza y baja (por eso se detuvo manualmente el aprendizaje). Es posible que reentrenando la red se consigan alcanzar valores ligeramente superiores o inferiores, siendo responsable de ello la inicialización aleatoria de los parámetros de red.

Empleando ambas redes para clasificar números dibujados por el usuario se extraen las siguientes conclusiones:

- Si la RNA acierta, la RNC también. No siempre sucede lo mismo en el otro caso.
- Si ambas fallan, la RNC sitúa el valor que realmente debería dar en una posición igual o superior al de la RNA.

De la misma manera que se hizo con la RNA, se va a presentar cómo "ve" la RNC una imagen. En lugar de hacerlo para todos los filtros, se hará para tres de cada capa de convolución. A diferencia de lo obtenido con la RNA al introducir la imagen con un "4", en la figura 1.6 sí se aprecia en las imágenes obtenidas tras la primera capa de convolución la misma figura del número inicial, solo que de menor dimensión y desplazado ligeramente en algún caso. Para la segunda capa de convolución las imágenes dejan de ser tan evidentes. Se muestran en la parte inferior de la figura tres imágenes representantes: la mayoría son como la de la derecha, no asociables a ninguna zona del número inicial. Otras como la de la izquierda, pueden entenderse como bordes del número. Finalmente la figura del centro sí puede entenderse como un zoom de los píxeles próximos a las coordenadas (4,6) de la imagen situada encima de ella.

En el caso de analizar los filtros en sí (de manera análoga a lo mostrado en la figura 1.7), no se puede concluir nada acerca de qué detectan, al menos de forma evidente. Los filtros mostrados en la figura 2.6 son muestra de ello. El de la izquierda puede recordarnos a un segmento del "1" y el de la derecha al borde superior de un "9" ó un "2". Sin embargo la mayoría de filtros son como los de la imagen del medio, difíciles de describir.

Asimismo, podemos analizar cómo son los valores de los elementos de los filtros al final de la simulación. En este caso es suficiente con mostrar el histograma asociado (figura 2.7).

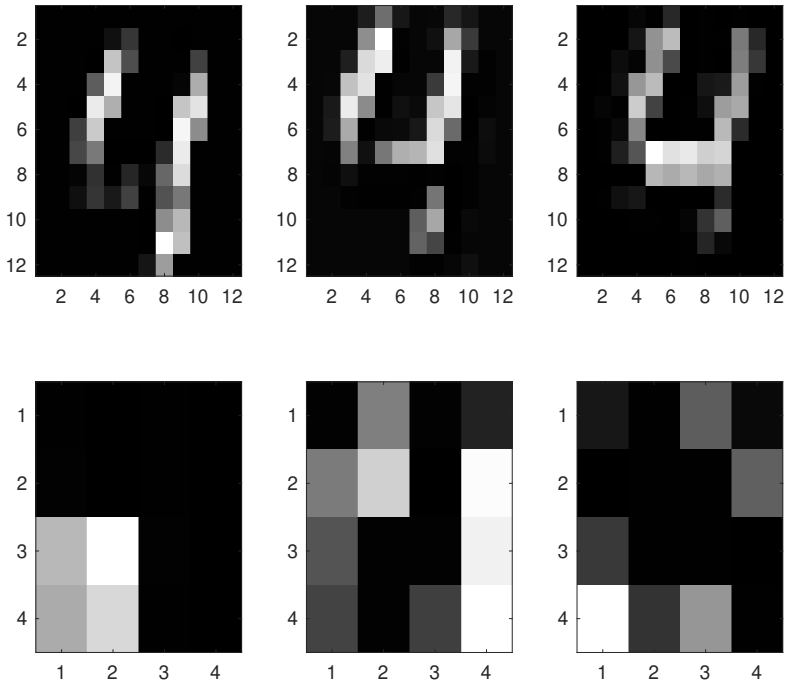

Figura 2.5: Algunas de las imágenes obtenidas por la RNC de características (2.27) y $\{22, 20, 0.5, 0.0001, 0.25\}$, tras aplicar la primera (parte superior) y segunda (parte inferior) capa de convolución.

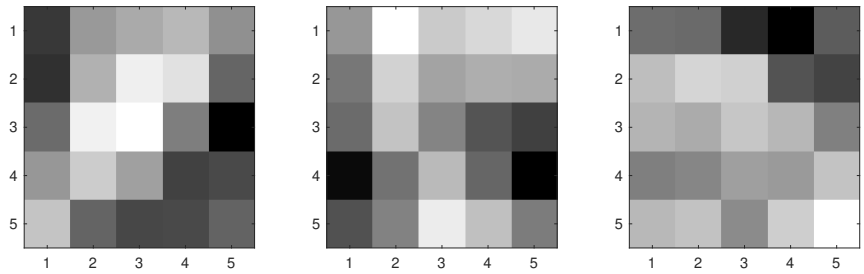

Figura 2.6: Filtros de la primera capa de la RNC de características (2.27) y $\{22, 20, 0.5, 0.0001, 0.25\}$.

Nuevamente, la disposición final de los valores es acampanada, pero los valores no están tan concentrados en el 0 como sí sucede en el caso de la RNA.

Todo lo mostrado permite concluir que la RNC es ligeramente superior que la RNA a la hora de clasificar imágenes de números pese a tener un número notablemente inferior de parámetros (del orden de 15 veces menor). En ambos casos la distribución final de los

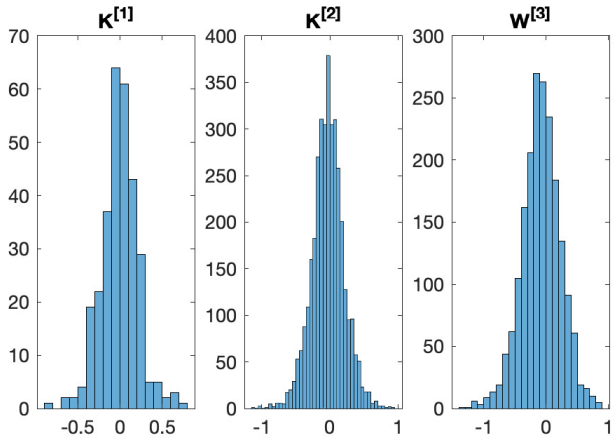

Figura 2.7: Histograma de los valores de la matriz de pesos y filtros de la RNC de características (2.27) y $\{22, 20, 0.5, 0.0001, 0.25\}$.

parámetros es similar, no así el tiempo empleado para llegar a ellos.

Indirectamente hemos comprobado que la explicación dada de las expresiones presentadas en (2.26) es correcta, dado que el código construido a partir de ellas ha permitido entrenar una RNC.

2.4.2. RNC para clasificar objetos y animales

Para no tener que crear nosotros las imágenes de objetos, haremos uso de la base CIFAR-10 [27]. De manera similar a la base MNIST, presenta 60000 imágenes a color (algunos ejemplos se muestran en la figura 2.8) de 10 clases distintas (avión, coche, ave, gato, ciervo, perro, rana, caballo, barco y camión), divididas en un conjunto de aprendizaje de 50000 y otro de testeo. Todas ellas tienen un tamaño de $32 \times 32 \times 3$.

Después de numerosas pruebas, la red que alcanzó la mayor precisión fue una red con propiedades

$$\begin{gathered} 1^{\underline{a}} \text{ capa (conv. + maxpool): } 3 \times 3 \times 3 \times 32 \\ 2^{\underline{a}} \text{ capa (conv. + maxpool): } 4 \times 4 \times 32 \times 64 \\ 3^{\underline{a}} \text{ capa (conv.): } 3 \times 3 \times 64 \times 64 \\ 4^{\underline{a}} \text{ capa (conv.) : } 3 \times 3 \times 64 \times 64 \\ 5^{\underline{a}} \text{ capa (RNA): } 10 \times 256 \end{gathered} \tag{2.28}$$

y $\{35, 20, 0.1, 0.0001, 0.05\}$, donde el cambio de la razón de aprendizaje se hizo una vez alcanzado el 46 % de precisión sobre las imágenes del test. Recordemos que tal precisión se obtiene promediando las precisiones por cada clase. En la matriz de confusión asociada (figura 2.9) se puede apreciar que el bajo valor de la precisión global se debe a la mala

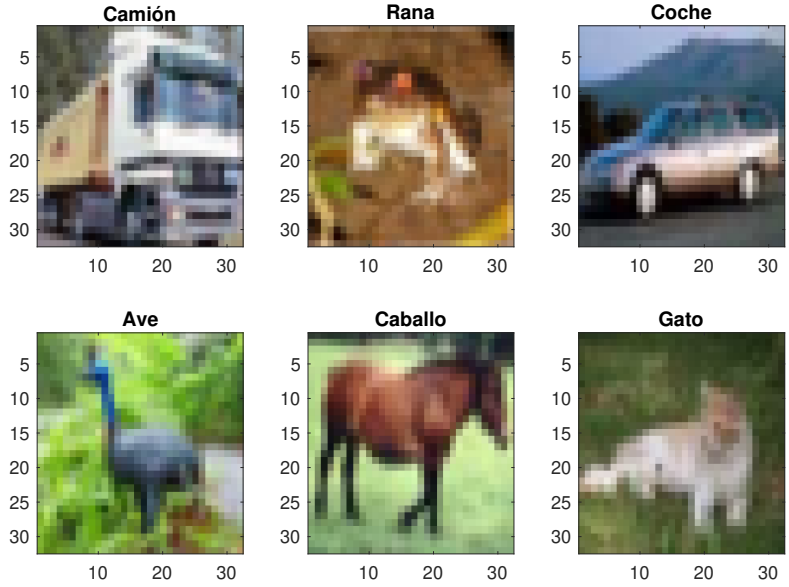

Figura 2.8: Ejemplos de la base de datos CIFAR-10 [27].

clasificación de clases como la de ciervos y gatos. En el caso concreto de los gatos es solo un 6 % superior a la precisión asociada a la clasificación aleatoria de una imagen en 10 clases.

Pese a la baja precisión, vuelven a identificarse confusiones de la red propias del ser humano. Los ejemplos más claros son la clasificación errónea de gatos como perros y de camiones como coches. Como en redes anteriores, el error no es tan alto al considerar el caso recíproco. Por otro lado, con la excepción de las ranas, la red tiende a clasificar mejor los objetos inanimados que los animados. Esto tiene sentido, dado que hay mayor variedad en la forma, color y tamaño en animales que en vehículos.

El tiempo de ejecución para obtener estas redes es tan elevado (del orden de 1.5 días) que se opta por no calcular el porcentaje de acierto ni valor de la función coste para el caso de las imágenes empleadas en el entrenamiento de la red.

Llegados a este punto nos preguntamos si la baja precisión obtenida en comparación con la de redes anteriores se debe realmente a la complejidad de las imágenes o a las técnicas empleadas en el aprendizaje de las mismas. Para comprobar que realmente sí es posible clasificar este tipo de imágenes se opta por emplear otra RNC para clasificar dos de las 10 clases anteriores: perros y gatos. Con la red anterior se consigue una precisión un 27 % y un 6.4 % superior al caso de clasificar aleatoriamente las imágenes, respectivamente. El objetivo que nos planteamos fue obtener una red que mejorara globalmente esos

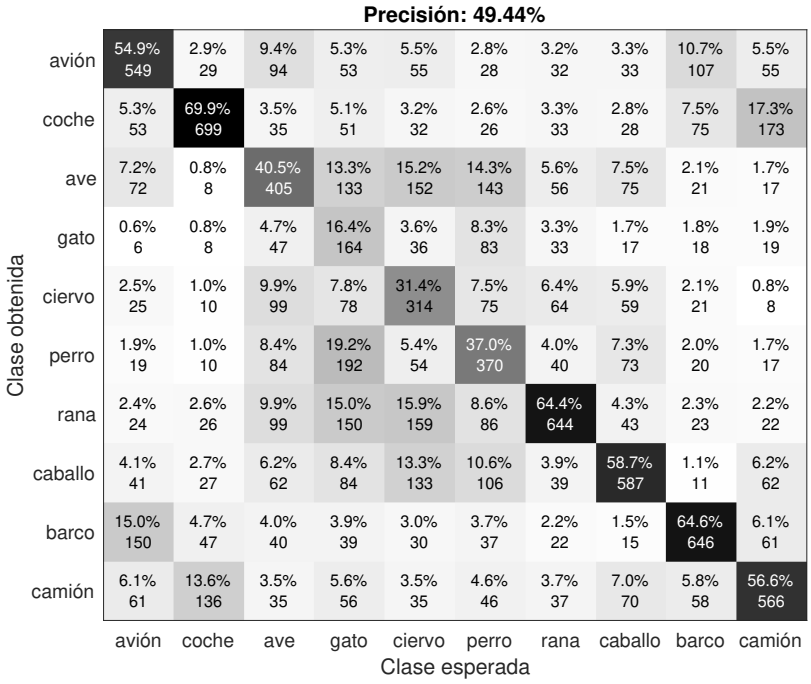

Figura 2.9: Matriz de confusión asociada a la RNC de características (2.28) y $\{35, 20, 0.1, 0.0001, 0.05\}$ para clasificar las imágenes CIFAR-10 del conjunto de testeo [25].

porcentajes.

Para ello hicimos uso de una red con propiedades

$$\begin{gathered} 1^{\underline{a}} \text{ capa (conv. + maxpool): } 3 \times 3 \times 3 \times 32 \\ 2^{\underline{a}} \text{ capa (conv. + maxpool): } 4 \times 4 \times 32 \times 64 \\ 3^{\underline{a}} \text{ capa (conv.): } 3 \times 3 \times 64 \times 64 \\ 4^{\underline{a}} \text{ capa (RNA): } 2 \times 1024 \end{gathered} \tag{2.29}$$

y $\{50, 20, 0.5, 0.0001, (0.4, 0.05, 0.04)\}$,

Se comenzó con un paso para el método del gradiente igual a 0.5 para posteriormente reducirlo a 0.1, 0.05 y finalmente 0.01. Cada cambio se realizó al alcanzarse un determinado valor de precisión de la red (65 %, 67 % y 69 %, respectivamente) sobre las imágenes de test.

De acuerdo con la figura 2.2, globalmente se mejora la precisión, no porque ahora la red supere el 70 %, sino porque ahora la precisión de las imágenes de perros está un 17.5 % y las de gatos un 24.4 % por encima de la precisión asociada a la clasificación aleatoria de las mismas. Aunque la asociada a la primera clase haya disminuido, más ha aumentado la de la segunda clase, haciendo que globalmente sea mejor esta red para clasificar imágenes

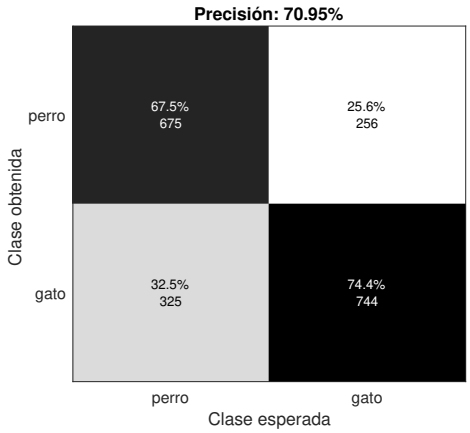

Tabla 2.2: Matriz de confusión asociada a la RNC de características (2.29) y $\{50, 20, 0.5, 0.0001, (0.4, 0.05, 0.04)\}$ para clasificar las imágenes de perros y gatos de la base CIFAR-10 [25].

de estas dos clases.

Concluimos por tanto que aunque las técnicas expuestas permitan entrenar redes para clasificar imágenes, es posible realizar mejoras que incrementen la precisión alcanzada en conjuntos más complejos que la base de datos MNIST. En este sentido, las mejoras más importantes se realizan en el método de descenso utilizado. En este trabajo se hizo uso del más popular, si bien es cierto que poco a poco los investigadores introducen cambios con los que acelerar e incrementar el aprendizaje de las redes [32]. Destacamos el algoritmo ADAM [33], que es una versión adaptativa del método de gradiente con momento [34].

Lo que debe quedar claro es que existen múltiples modificaciones a lo que podemos definir como *teoría clásica* de estas redes que mejoran el aprendizaje de las mismas, no así un criterio unificado respecto a cuál es mejor para cada problema.

Capítulo 3

Conclusiones

En el presente trabajo se ha conseguido desarrollar la teoría fundamental que rodea a las redes neuronales. En el caso de las RNA se partió de su antecesor, el perceptrón, poniéndonos como objetivo obtener una red capaz de clasificar números del 0 al 9. Para ello formalizamos su estructura y presentamos los dos elementos necesarios para su aprendizaje: un método de descenso y un algoritmo para obtener las expresiones empleadas en el mismo. Elegimos los método del gradiente estocástico y mini-batch por ser los clásicamente más empleados. Por otro lado, nos detuvimos para desarrollar con detalle el algoritmo de la propagación inversa mediante el método del adjunto. Finalmente comprobamos que esos ingredientes eran suficientes para completar el objetivo inicial. Además de probar que una RNA puede clasificar números, se analizó la influencia de cada hiperparámetro sobre ella.

Las limitaciones detectadas en este tipo de redes dieron pie a la presentación de las RNC, para las que se formalizó su estructura y se indicaron los cambios a realizar en el algoritmo de la propagación inversa. Para comparar estas redes con las anteriores, se usaron para clasificar la misma base de imágenes que la usada con RNA. Por otro lado, para ver su potencialidad y limitaciones, se usaron para clasificar imágenes más complejas. De sus resultados se concluyó que efectivamente estas redes son más prometedoras que las RNA, pero resulta necesario emplear técnicas más recientes para conseguir una precisión más cercana a la alcanzada en el caso de la clasificación de números.

A pesar de este último resultado poco satisfactorio, el objetivo principal, que no era otro que entender qué se esconde a nivel matemático detrás de una RNA y una RNC, ha sido completado y así lo demuestran los programas con los que se han obtenido todas las redes presentadas. Dicho esto, se es consciente de que los códigos desarrollados no pueden competir con la librería de Matlab, Deep Learning Toolbox, o con programas diseñados específicamente para generar y entrenar redes como Keras o TensorFlow.

Esto no quita valor al trabajo realizado: en el contexto actual del Machine Learning,

son muchas las investigaciones que buscan emplear el aprendizaje profundo en problemas como la resolución de ecuaciones diferenciales [35]. Conocer el formalismo y programación de las redes existentes cataliza el proceso de diseño e implementación de nuevas estructuras neuronales capaces de lograr tales objetivos.

Códigos para RNA

.1. Programa principal

```
clear all
%cargo imagenes para que la red aprenda. tambien cargo las
    etiqueas
%asociadas (indican lo que representa la imagen)
load('storage.mat');
datos=storage{1,1};
l=size(datos,2);
etiquetas=storage{1,2};
%analogo para im genes test
test=storage{2,1};
test_labels=storage{2,2};
m=size(test,2);
%llamada a los hiperparametros
[v,epoch,batch,paso,reduccion,reg]=hiperparam;
n=length(v);
%creacion del cell array
net=cell(n-1,2);
pesos=net;forward=cell(n-1,1);

%para criterios de cambio de paso
auxiliar_red=0;
%

%inicializaci n red
for i=1:n-1
```

```
25      net{i,1}=unifrnd(-sqrt(6)/sqrt(v(i+1)+v(i)),sqrt(6)/sqrt(v(i
            +1)+v(i)),v(i+1),v(i));
26      net{i,2}=zeros(v(i+1),1);
27  end
28
29  %
        ------------------------------------------------------------------------


30  %mini indica cuantos subgrupos de batch elementos hay por epoch
31  mini=l/batch;
32  error=zeros(epoch,1);error_test=error;aciertos_datos=error;
        aciertos_test=error;
33  contador=1;
34  clase_real=zeros(1,m);clase_output=zeros(1,m);
35  for i=1:epoch
36      %barajo los datos al principio de cada epoch
37      random=randperm(l);
38      datos_r=datos(:,random);
39      etiquetas_r=etiquetas(:,random);
40
41      for j=1:mini
42          %definicion red para almacenar el gradiente de J
43          for k=1:n-1
44              pesos{k,1}=zeros(v(k+1),v(k));
45              pesos{k,2}=zeros(v(k+1),batch);
46          end
47          %nota: aunque dimensionalmente es incorrecto, matlab "
                sabe" como
48          %sumar ax+b
49          datos_mini=datos_r(:,(1+(j-1)*batch):batch*j);
50          forward{1,1}=activate(datos_mini,net{1,1},net{1,2});
51          for k=2:n-1
52              forward{k,1}=activate(forward{k-1,1},net{k,1},net{k
                    ,2});
53          end
54          coste=forward{n-1,1}-etiquetas_r(:,(1+(j-1)*batch):batch*
```

```
            j);
55          %valor funcion error, la ultima iteracion de todas no la
               graba
56
57          %bpa
58          delta=forward{n-1,1}.*(1-forward{n-1,1}).*coste;
59          pesos{n-1,1}=delta*forward{n-2,1}'+reg*net{n-1,1};
60          pesos{n-1,2}=delta;
61          for k=n-2:-1:2
62              delta=forward{k,1}.*(1-forward{k,1}).*(net{k+1,1}'*
                   delta);
63              pesos{k,1}=delta*forward{k-1,1}'+reg*net{k,1};
64              pesos{k,2}=delta;
65          end
66          delta=forward{1,1}.*(1-forward{1,1}).*(net{2,1}'*delta);
67          pesos{1,1}=delta*datos_mini'+reg*net{1,1};
68          pesos{1,2}=delta;
69
70          %sgd y actualizacion
71          for k=1:n-1
72              net{k,1}=net{k,1}-paso/batch*pesos{k,1};
73              net{k,2}=net{k,2}-paso/batch*sum(pesos{k,2},2);
74          end
75      end
76      error(contador,1)=error(contador,1)*1/mini;
77
78      %testeo de la calidad de la red al final de cada epoch
79      forward{1,1}=activate(test,net{1,1},net{1,2});
80      for j=2:n-1
81          forward{j,1}=activate(forward{j-1,1},net{j,1},net{j,2});
82      end
83      aciertos=0;
84      coste=forward{n-1,1}-test_labels;
85      error_test(contador,1)=1/m*sum(vecnorm(coste,2).^2)/2;
86      for k=1:m
87          [val, idx]=max(forward{n-1,1}(:,k));
```

```
88          [val2, idx2]=max(test_labels(:,k));
89          if i==epoch
90              clase_real(k)=idx2-1;
91              clase_output(k)=idx-1;
92          end
93          if idx==idx2
94              aciertos=aciertos+1;
95          end
96      end
97      aciertos_test(contador,1)=aciertos*100/m;
98      fprintf('epoch %d: acierta %d/%d \n',i,aciertos,m);
99          if aciertos>=9800 && auxiliar_red==0
100          paso=paso-reduccion;
101          auxiliar_red=1;
102         end
103
104     %
```

```
105     forward{1,1}=activate(datos,net{1,1},net{1,2});
106     for j=2:n-1
107         forward{j,1}=activate(forward{j-1,1},net{j,1},net{j,2});
108     end
109     aciertos=0;
110     coste=forward{n-1,1}-etiquetas;
111     error(contador,1)=1/l*sum(vecnorm(coste,2).^2)/2;
112     for k=1:l
113         [val, idx]=max(forward{n-1,1}(:,k));
114         [val2, idx2]=max(etiquetas(:,k));
115         if idx==idx2
116             aciertos=aciertos+1;
117         end
118     end
119     aciertos_datos(contador)=aciertos*100/l;
120     contador=contador+1;
121     end
```

```
122 %analisis resultados
123 figure()
124 subplot(1,2,1);
125 plot(error,'-'); hold on
126 plot(error_test,'*'); hold off
127 title('Valor funci n coste');xlabel('epoch');ylabel('J');
128 legend('entrenamiento','test');
129 subplot(1,2,2);
130 plot(aciertos_datos,'-'); hold on
131 plot(aciertos_test,'*'); hold off
132 title('Tasa de acierto');xlabel('epoch');ylabel('%');
133 legend('% entrenamiento','% test');
134 figure()
135 plotConfMat(confusionmat(clase_output,clase_real),{'0','1','2','3
    ','4','5','6','7','8','9'});
```

.2. Hiperparámetros

```
1  function [v,epoch,batch,paso,reduccion,reg]=hiperparam
2  %function [v,epoch,batch,paso,reduccion,reg]=hiperparam
3  % definici n de los hiperparametros de la red
4
5  % v-> neuronas por capa, incluir capas inicial y final
6  % epoch-> numero de veces que se pasa por todos los datos
7  % batch-> numero de imagenes usadas por iteracion del met.grad.
       est/minb
8  % paso -> paso del m tod gradiente
9  % reduccion paso-> cantidad a restar al paso cuando se cumple
       algun
10 % criterio -> (con un if, por iteracion... debe cambiarse en el
       codigo de
11 %           red_neuronal)
12 % reg-> valor de la razon del regularizador
13 v=[28*28,100,100,10];
14
15 epoch=500;
16
```

```
batch=10000;

paso=0.5;

reduccion=0.5;

reg=1.2500e-04;
end
```

.3. Función de activación

```
function y = activate(x,W,b)
%function y = activate(x,W,b)
% funcion sigmoide
y = 1./(1+exp(-(W*x+b)));
end
```

Códigos para RNC

.4. Programa principal

```
%cargo fotos
load('fotos.mat');
datos=fotos{1,1}(:,:,:,:);
l=size(datos,4);
etiquetas=fotos{1,2}(:,:);
test=fotos{2,1}(:,:,:,:);
test_labels=fotos{2,2}(:,:);
m=size(test,4);

%-------------------------------------------------------------------------
[capas_cnn,capas_dnn,w_cnn,w_dnn,z_cnn,z_dnn,y_cnn,epoch,batch,
    paso,crit,reg,red]=hiperparam;
contador=1;
mini=l/batch;
%creo red: OJO-> net{1,1}, cambia con la resolucion de la foto
net=cell(capas_cnn+capas_dnn,2);
net{1,1}=unifrnd(-sqrt(6)/sqrt(3072),sqrt(6)/sqrt(3072),w_cnn
    (1,1),w_cnn(1,2),w_cnn(1,3),w_cnn(1,4));
net{1,2}=zeros(w_cnn(1,4),1);

for i=2:capas_cnn
      net{i,1}=unifrnd(-sqrt(6)/sqrt(y_cnn(i-1,1)*y_cnn(i-1,2)),
          sqrt(6)/sqrt(y_cnn(i-1,1)*y_cnn(i-1,2)),w_cnn(i,1),w_cnn(
          i,2),w_cnn(i,3),w_cnn(i,4));
      net{i,2}=zeros(w_cnn(i,4),1);
```

```
22 end
23 for i=1:capas_dnn
24     net{capas_cnn+i,1}=unifrnd(-sqrt(6)/(w_dnn(i,2)+w_dnn(i,1)),
           sqrt(6)/(w_dnn(i,2)+w_dnn(i,1)),w_dnn(i,1),w_dnn(i,2));
25     net{capas_cnn+i,2}=zeros(w_dnn(i,1),1);
26 end
27 pesos=net;
28 %forward
29 forward=cell(capas_cnn+capas_dnn,1);
30 %almaceno memoria para las derivadas
31 derivadas=cell(capas_cnn+capas_dnn,1);
32 %para analisis posterior
33 error=zeros(epoch,1);error_test=error;aciertos_datos=error;
       aciertos_test=error;
34 clase_real=zeros(1,m);clase_output=zeros(1,m);
35 paraporc=0;
36 %-----------------
37 for i=1:epoch
38     random=randperm(l);
39     datos_r=datos(:,:,:,random);
40     etiquetas_r=etiquetas(:,random);
41     for j=1:mini
42         datos_mini=datos_r(:,:,:,(1+(j-1)*batch):batch*j);
43         [forward{1,1},derivadas{1,1}]=adelante(datos_mini,net
               {1,1},net{1,2},crit(1));
44         for k=2:capas_cnn
45             [forward{k,1},derivadas{k,1}]=adelante(forward{k
                   -1,1},net{k,1},net{k,2},crit(k));
46         end
47         auxiliar=reshape(forward{capas_cnn,1},w_dnn(1,2),batch);
48         if capas_dnn>1
49             [forward{capas_cnn+1,1},derivadas{capas_cnn+1,1}]=
                   adelante_dnn(auxiliar,net{capas_cnn+1,1},net{
                   capas_cnn+1,2});
50             for k=2:capas_dnn-1
51                 [forward{capas_cnn+k,1},derivadas{capas_cnn+k
```

```
                    ,1}]=adelante_dnn(forward{capas_cnn+k-1,1},net
                    {capas_cnn+k,1},net{capas_cnn+k,2});
52              end
53              forward{capas_cnn+capas_dnn,1}=sigmoid(forward{
                    capas_cnn+capas_dnn-1},net{capas_cnn+capas_dnn,1},
                    net{capas_cnn+capas_dnn,2});
54          else
55              forward{capas_cnn+1,1}=sigmoid(auxiliar,net{capas_cnn
                    +1,1},net{capas_cnn+1,2});
56          end
57          coste=forward{capas_cnn+capas_dnn,1}-etiquetas_r(:,(1+(j
                -1)*batch):batch*j);
58              for k=1:capas_cnn
59                  pesos{k,1}=zeros(w_cnn(k,1),w_cnn(k,2),w_cnn(k,3)
                        ,w_cnn(k,4));
60                  pesos{k,2}=zeros(z_cnn(k,1),z_cnn(k,2),z_cnn(k,3)
                        );
61              end
62              for k=1:capas_dnn
63                  pesos{k+capas_cnn,1}=zeros(w_dnn(k,1),w_dnn(k,2))
                        ;
64                  pesos{k+capas_cnn,2}=zeros(w_dnn(k,1),1);
65              end
66          %bpa del dnn
67          delta=forward{capas_cnn+capas_dnn,1}.*(1-forward{
                capas_cnn+capas_dnn,1}).*coste;
68          if capas_dnn>1
69              for k=capas_dnn:-1:2
70                  pesos{k,1}=delta*forward{capas_cnn+k-1,1}'+reg*
                        net{capas_cnn+k,1};
71                  pesos{k,2}=sum(delta,2);
72                  delta=derivadas{capas_cnn+k-1,1}.*(net{capas_cnn+
                        k,1}'*delta);
73              end
74          end
75          pesos{capas_cnn+1,1}=delta*auxiliar'+reg*net{capas_cnn
```

```
    +1,1};
76          pesos{capas_cnn+1,2}=sum(delta,2);
77
78          %bpa del cnn
79          auxiliar=reshape(net{capas_cnn+1,1}'*delta,y_cnn(
                capas_cnn,1),y_cnn(capas_cnn,2),y_cnn(capas_cnn,3),
                y_cnn(capas_cnn,4));
80          delta=zeros(z_cnn(capas_cnn,1),z_cnn(capas_cnn,2),z_cnn(
                capas_cnn,3),z_cnn(capas_cnn,4));
81          for k=1:y_cnn(capas_cnn,1)
82              for h=1:y_cnn(capas_cnn,2)
83                  if crit(end)==1
84                      delta(2*k-1:2*k,2*h-1:2*h,:,:)=auxiliar(k,h
                            ,:,:).*derivadas{capas_cnn,1}(2*k-1:2*k,2*
                            h-1:2*h,:,:);
85                  else
86                      delta=auxiliar.*derivadas{capas_cnn,1};
87                  end
88              end
89          end
90          if capas_cnn>1
91              pesos{capas_cnn,2}=sum(reshape(sum(sum(delta)),w_cnn(
                    capas_cnn,4),batch),2);
92              pesos{capas_cnn,1}=calculo_w(delta,forward{capas_cnn
                    -1,1},w_cnn(capas_cnn,:),batch,reg,net{capas_cnn
                    ,1});
93              for k=capas_cnn-1:-1:2
94                  delta=creo_delta(delta,net{k+1,1},derivadas{k,1},
                        crit(k));
95                  pesos{k,2}=sum(reshape(sum(sum(delta)),w_cnn(k,4)
                        ,batch),2);
96                  pesos{k,1}=calculo_w(delta,forward{k-1,1},w_cnn(k
                        ,:),batch,reg,net{k,1});
97              end
98              %ultima capa hacia atras
99              delta=creo_delta(delta,net{2,1},derivadas{1,1},crit
```

```
                (1));
100         end
101             pesos{1,2}=sum(reshape(sum(sum(delta)),w_cnn(1,4),
                    batch),2);
102             pesos{1,1}=calculo_w(delta,datos_mini,w_cnn(1,:),
                    batch,reg,net{1,1});
103             %actualizacion pesos
104         for k=1:capas_cnn+capas_dnn
105             net{k,1}=net{k,1}-paso/batch*pesos{k,1};
106             net{k,2}=net{k,2}-paso/batch*pesos{k,2};
107         end
108     end
109     %
```

```
110     %test por epoch
111     [forward{1,1},~]=adelante(test,net{1,1},net{1,2},crit(1));
112     for j=2:capas_cnn
113         [forward{j,1},~]=adelante(forward{j-1,1},net{j,1},net{j
                ,2},crit(j));
114     end
115     auxiliar=reshape(forward{capas_cnn,1},w_dnn(1,2),m);
116     forward{capas_cnn+1,1}=sigmoid(auxiliar,net{capas_cnn+1,1},
            net{capas_cnn+1,2});
117     for k=2:capas_dnn
118         forward{capas_cnn+k}=sigmoid(forward{capas_cnn+k-1},net{
                capas_cnn+k,1},net{capas_cnn+k,2});
119     end
120     coste=forward{capas_cnn+capas_dnn,1}-test_labels;
121     aciertos=0;
122     error_test(contador,1)=1/m*sum(vecnorm(coste,2).^2)/2;
123     for k=1:m
124         [val, idx]=max(forward{capas_cnn+capas_dnn,1}(:,k));
125         [val2, idx2]=max(test_labels(:,k));
126         if i==epoch
127             clase_real(k)=idx2-1;
```

```
128             clase_output(k)=idx-1;
129         end
130         if idx==idx2
131             aciertos=aciertos+1;
132         end
133     end
134     aciertos_test(contador,1)=aciertos*100/m;
135     fprintf('epoch %d: acierta %d/%d \n',i,aciertos,m);
136     %--------
137     %--------
138
139     [forward{1,1},~]=adelante(datos,net{1,1},net{1,2},crit(1));
140     for j=2:capas_cnn
141         [forward{j,1},~]=adelante(forward{j-1,1},net{j,1},net{j
                ,2},crit(j));
142     end
143     auxiliar=reshape(forward{capas_cnn,1},w_dnn(1,2),l);
144     forward{capas_cnn+1,1}=sigmoid(auxiliar,net{capas_cnn+1,1},
            net{capas_cnn+1,2});
145     for k=2:capas_dnn
146         forward{capas_cnn+k}=sigmoid(forward{capas_cnn+k-1},net{
                capas_cnn+k,1},net{capas_cnn+k,2});
147     end
148     aciertos=0;
149     coste=forward{capas_cnn+capas_dnn,1}-etiquetas;
150     error(contador,1)=1/l*sum(vecnorm(coste,2).^2)/2;
151     for k=1:l
152         [val, idx]=max(forward{capas_cnn+capas_dnn,1}(:,k));
153         [val2, idx2]=max(etiquetas(:,k));
154         if idx==idx2
155             aciertos=aciertos+1;
156         end
157     end
158     aciertos_datos(contador)=aciertos*100/l;
159     if aciertos_test(contador)>=46 && paraporc==0
160         save ('red_cnn_tfg_supero46.mat','net');
```

```
161             grafica=[error,error_test,aciertos_datos,aciertos_test];
162             save ('grafica46.mat','grafica');
163             paraporc=1;
164             paso=paso/2;
165         end
166         if aciertos_test(contador)>=49 && paraporc==1
167             save ('red_cnn_tfg_supero49.mat','net');
168             grafica=[error,error_test,aciertos_datos,aciertos_test];
169             save ('grafica49.mat','grafica');
170             paraporc=2;
171             paso=paso/2;
172 
173         end
174         contador=contador+1;
175 end
176 %analisis resultados
177 %figure(1)
178 %subplot(1,2,1);
179 %plot(error,'-'); hold on
180 %plot(error_test,'*'); hold off
181 %title('Valor funci n coste');xlabel('epoch');ylabel('J');
182 %legend('entrenamiento','test');
183 %subplot(1,2,2);
184 %plot(aciertos_datos,'-'); hold on
185 %plot(aciertos_test,'*'); hold off
186 %title('Tasa de acierto');xlabel('epoch');ylabel('%');
187 %legend('% entrenamiento','% test');
188 
189 
190 %para el cluster
191 %guarda la red
192 save ('red_cnn_tfg_junio200.mat','net');
193 %guarda los datos de ploteo
194 grafica=[error,error_test,aciertos_datos,aciertos_test];
195 save ('grafica_junio200.mat','grafica');
```

.5. Hiperparámetros

```
function [capas_cnn,capas_dnn,w_cnn,w_dnn,z_cnn,z_dnn,y_cnn,epoch
    ,batch,paso,crit,reg,red]=hiperparam
%function [capas_cnn,capas_dnn,w_cnn,w_dnn,z_cnn,z_dnn,y_cnn,
    epoch,batch,paso,crit,reg,red]=hiperparam
% funcion que indica toda la informaci n necesaria para
    construir la RNC
% capas_cnn, capas_dnn-> cu ntas capas de cada tipo hay
% batch-> tama o mini-batch
% w_cnn,z_cnn,y_cnn,w_dnn,z_dnn-> dimensiones de los tensores
    involucrados
% en la RNC
% epoch-> n mero de veces que se usa cada foto
% paso-> paso del metodo de gradiente elegido
% crit-> indica si se aplica (1) o no (0) PReLU en cada capa_cnn
% reg -> razon del regularizador
% red-> reduccion de paso

    capas_cnn=4;capas_dnn=1;

    batch=20;

    w_cnn=[3,3,3,32;4,4,32,64;3,3,64,64;3,3,64,64];w_dnn
        =[10,256];
    z_cnn=[30,30,32,batch;12,12,64,batch;4,4,64,batch;2,2,64,
        batch];z_dnn=[10];
    y_cnn=[15,15,32,batch;6,6,64,batch;4,4,64,batch;2,2,64,batch
        ];

    epoch=35;

    paso=0.1;

    crit=[1,1,0,0];

```

```
28      reg=0.0001;
29
30      red=0.05;
31  end
```

.6. Evaluación de la red, funciones auxiliares

```
1   function [y,derivada]=adelante(input,w,b,criterio)
2   %function [y,derivada]=adelante(input,w,b,criterio)
3   %funcion que evalua la red en las capas de convolucion
4       dim_w=size(w);dim_input=size(input);mini=dim_input(4);
5       can_filtros=dim_w(4);grosor=dim_w(3);
6       z=zeros(1+dim_input(1)-dim_w(1),1+dim_input(1)-dim_w(1),
            can_filtros,mini);
7       for i=1:can_filtros
8           c=0;
9           for k=1:grosor
10              c=convn(input(:,:,k,:),w(:,:,k,i),'valid')+c;
11          end
12          z(:,:,i,:)=c+b(i,1);
13      end
14      %z=z+b;
15  %     realmente seria este codigo, pero es mas eficiente el otro,
        basta
16  %     aprovechar c mo funciona convn. He comprobado que ambos
        resultados
17  %     son iguales, pero el primero tarda siete veces menos
18  %     for i=1:can_filtros
19  %         for j=1:mini
20  %             c=0;
21  %             for k=1:grosor
22  %                 c=c+conv2(input(:,:,k,j),w(:,:,k,i),'valid');
23  %             end
24  %             x(:,:,i,j)=c+b(i,1);
25  %         end
26  %     end
27
```

```
%ReLU
    y=max(z,0)+0.01*min(z,0);
    derivada_ReLU=1*(y>0)+0.01*(y<=0);%el 1* es para restaurar el
        caracter double
    derivada_pooling=1;
%posible maxpooling
    if criterio==1
        [derivada_pooling,y]=pooling(y);
    end
    derivada=derivada_pooling.*derivada_ReLU;
end
```

```
function [derivada,y]=pooling(input)
%function [derivada,y]=pooling(input)
% funcion para realizar el maxpool

    dim_input=size(input);dimension=dim_input(1)/2;filtros=
        dim_input(3);cantidad=dim_input(4);
    y=zeros(dimension,dimension,filtros,cantidad);derivada=zeros(
        dimension*2,dimension*2,filtros,cantidad);
    for k=1:2:(2*dimension-1)
        for l=1:2:(2*dimension-1)
            d=input(k:k+1,l:l+1,:,:);
            c=max(max(d));
            derivada(k:k+1,l:l+1,:,:)=1*(d==c);
            y((k+1)/2,(l+1)/2,:,:)=c;
        end
    end
end
```

```
function [y,derivada]=adelante_dnn(input,w,b)
%function [y,derivada]=adelante_dnn(input,w,b)
% funcion para aplicar PReLU en las capas de RNA que no sean la
    final
    z=w*input+b;
    y=max(z,0)+0.01*min(z,0);
%derivada
```

```
7       derivada=1*(y>0)+0.01*(y<=0); %el 1* es para restaurar el
            caracter double
8   end
```

```
1   function y = sigmoid(x,W,b)
2   %function y = sigmoid(x,W,b)
3   %funcion logistica para la ultima capa de RNA
4
5   y = 1./(1+exp(-(W*x+b)));
```

.7. Algoritmo propagación inversa, funciones auxiliares

```
1   function delta=creo_delta(delta_pre,w,derivada,crit)
2   %function delta=creo_delta(delta_pre,w,derivada,crit)
3   %funcion para crear el multiplicador de Lagrange del bpa
4
5       w=rot90(rot90(w));dim_w=size(w);dim_deltapre=size(delta_pre);
6       verd_dim=dim_w(1)+dim_deltapre(1)-1;
7       auxiliar=zeros(verd_dim,verd_dim,dim_w(3),dim_deltapre(4));
8       delta=zeros(verd_dim*2,verd_dim*2,dim_w(3),dim_deltapre(4));
9
10
11      for i=1:dim_w(4)
12          for j=1:dim_deltapre(4)
13              auxiliar(:,:,:,j)=convn(w(:,:,:,i),delta_pre(:,:,i,j)
                    );
14          end
15      end
16
17  %forma no eficiente
18  %      for i=1:dim_deltapre(4)
19  %          for j=1:dim_w(3)
20  %              c=0;
21  %              for k=1:dim_deltapre(3)
22  %                  c=c+conv2(w(:,:,j,k),delta_pre(:,:,k,i),'full')
        ;
23  %              end
```

```
%                 auxiliar(:,:,j,i)=c;
%             end
%         end


if crit==1
    for i=1:verd_dim
        for j=1:verd_dim
            delta(2*i-1:2*i,2*j-1:2*j,:,:)=auxiliar(i,j,:,:).*
                derivada(2*i-1:2*i,2*j-1:2*j,:,:);
        end
    end
else
    delta=auxiliar.*derivada;
end
end
```

```
function peso=calculo_w(delta,sigmaz,dimensiones,mini,reg,net)
%function peso=calculo_w(delta,sigmaz,dimensiones,mini,reg,net)
%funcion para realizar calculos del algoritmo de la propagacion
    inversa

    sigmaz=rot90(rot90(sigmaz));
    alto=dimensiones(1);profundidad=dimensiones(3);cantidad=
        dimensiones(4);
    peso=zeros(alto,alto,profundidad,cantidad);
        for j=1:cantidad
            c=0;
            for k=1:mini
                c=c+convn(sigmaz(:,:,:,k),delta(:,:,j,k),'valid')
                    ;
            end
            peso(:,:,:,j)=c;
        end
        peso=peso+reg*net;
        %forma no eficiente
%     for i=1:profundidad
```

```
18 %           for j=1:cantidad
19 %               c=0;
20 %               for k=1:mini
21 %                   c=c+conv2(sigmaz(:,:,i,k),delta(:,:,j,k),'valid
       ');
22 %               end
23 %               weqa(:,:,i,j)=c;
24 %           end
25 %       end
26 end
```

Bibliografía

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[2] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[3] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[4] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[5] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[6] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, 2015.

[7] Alexander LeNail. Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33):747, 2019.

[8] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database, 2010. `http://yann.lecun.com/exdb/mnist/`. Último acceso: 2021-05-29.

[9] Catherine F Higham and Desmond J Higham. Deep learning: An introduction for applied mathematicians. *SIAM Review*, 61(4):860–891, 2019.

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[11] Wolfgang Hackbusch. *Gradient Method*, pages 211–228. Springer International Publishing, Cham, 2016.

[12] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[13] Ahmed Khaled and Peter Richtárik. Better theory for sgd in the nonconvex world. *arXiv preprint arXiv:2002.03329*, 2020.

[14] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle points—online stochastic gradient for tensor decomposition. In *Conference on learning theory*, pages 797–842. PMLR, 2015.

[15] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[16] Xue Ying. An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, volume 1168, page 022022. IOP Publishing, 2019.

[17] Jen-Tzung Chien and Tsai-Wei Lu. Tikhonov regularization for deep neural network acoustic modeling. In *2014 IEEE Spoken Language Technology Workshop (SLT)*, pages 147–152. IEEE, 2014.

[18] Andrew M Bradley. Pde-constrained optimization and the adjoint method. Technical report, Technical Report. Stanford University., 2013.

[19] Steven G Johnson. Notes on adjoint methods for 18.335. *Introduction to Numerical Methods*, 2012. `https://math.mit.edu/~stevenj/18.336/adjoint.pdf`. Último acceso: 2021-04-15.

[20] Dimitri P Bertsekas et al. *Dynamic programming and optimal control: Vol. 1*. Athena scientific Belmont, 2000.

[21] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[22] Paul J Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[24] Jürgen Schmidhuber. Who invented backpropagation? *More[DL2]*, 2014.

[25] Vahe Tshitoyan. *Plot Confusion Matrix*, 2020. `https://github.com/vtshitoyan/plotConfMat`. Último acceso: 2021-03-12.

[26] 3Blue1Brown. Neural networks, 2017. `https://cutt.ly/smhiHoB`. Último acceso: 2021-06-20.

[27] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[28] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[29] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[30] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[32] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[33] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[34] Jason Brownlee. *Better Deep Learning: Train Faster, Reduce Overfitting, and Make Better Predictions*. Machine Learning Mastery, 2018.

[35] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Índice alfabético