Trabajo Fin de Grado

Grado en Ingeniería Robótica, Electrónica y Mecatrónica

Redes Neuronales Convolucionales en R

Reconocimiento de caracteres escritos a mano

Autor: Jaime Durán Suárez

Tutor: Alejandro del Real Torres

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Redes Neuronales Convolucionales en R

Autor:
Jaime Durán Suárez

Tutor:
Alejandro del Real Torres
Profesor asociado

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Redes Neuronales Convolucionales en R

Autor: Jaime Durán Suárez

Tutor: Alejandro del Real Torres

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A Mercedes

A mi familia

Agradecimientos

En primer lugar, me gustaría agradecer a Alejandro del Real Torres toda la ayuda prestada durante estos meses en el desarrollo del proyecto, así como las oportunidades que me ha brindado al abrirme las puertas de su empresa donde he realizado e implementado este trabajo. También a los compañeros de la empresa que han tenido un comportamiento ejemplar y siempre que he necesitado ayuda han estado ahí.

En segundo lugar, a Mercedes Brenes Ferreiro y a mi familia, por su amor y cariño incondicional, además de su total apoyo, tanto en los buenos como en lo malos momentos, sin ellos no hubiese podido acabar esta etapa y siempre les estaré agradecidos por todo lo bueno que hacen por mí.

Resumen

A pesar de que las redes neuronales artificiales son un concepto que se ha investigado desde mediados del siglo XX, no ha sido hasta fechas recientes cuando han experimentado una tasa de crecimiento muy alta. Debido a grandes mejoras en su desempeño, a lo largo de estos últimos años, su uso ha pasado de ser meramente académico y de objeto de estudio e investigación a estar totalmente implementadas y operativas en nuestro día a día, incluso sin que nos percatemos de ello.

El reconocimiento de imágenes que Google usa para su buscador, el algoritmo AlphaGo que fue capaz de ganar al campeón del mundo jugando al Go o el reconocimiento de rostros en imágenes por parte de Facebook son solo algunas muestras que permiten entrever lo presentes que están estos sistemas de inteligencia artificial en el mundo actual y el inmenso número de aplicaciones que éstas pueden llevar a cabo.

Todo lo anterior viene a poner de manifiesto la principal característica de las redes neuronales: son sistemas fáciles de crear, la mayor dificultad es implementar en el lenguaje de programación deseado el algoritmo de aprendizaje, que tan solo consta de una serie de operaciones matemáticas iterativas muy simples (y ni si quiera eso si se usa una de las innumerables librerías ya existentes para la mayoría de lenguajes), y, a su vez, son una herramienta tremendamente potente y versátil.

En el presente proyecto, se hará uso de estas redes neuronales y, en concreto, de una red neuronal convolucional para desarrollar un algoritmo en el lenguaje de programación R que, tras ser entrenado con una extensa base de datos, sea capaz de reconocer caracteres escritos a mano, específicamente se usarán números del 0 al 9. El objetivo que se persigue es el de conseguir, al menos, un porcentaje superior al 95 %, con un conjunto de 10000 imágenes de comprobación.

Abstract

The artificial neural networks are a concept which has been investigated since the mid of the XX century but it hasn`t been until now when they have experienced a very high growth rate. Due to big improvements in their behaviour, during these recently years, their use have passed from be only for academical purposes to be totally implemented and operative in our life.

These neural nets are systems which are used in a lot of different aplications nowadyas. Thus, this bring us the main feature of the neural nets: they are systems easy to build, the biggest problem is to implement the learning algorithm, which is composed by a few very simple iterative maths operations (even less if we use some library), and, at the same time, they are very powerfull systems.

In this project, we will use this neural networks and, specifically, a convolutional neural network to develop an algortihm, in R, which, after being trained with a huge dataset, it will be able to recognize hand-written characters, specifically numbers from 0 to 9. The goal we are looking for is to obtain, at least, a percentage greater than 95 %, with a set of 10000 images to check its behaviour.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

a, b, c	Escalares
a, **b**, **c**	Vectores
A, **B**, **C**	Matrices
e	número e
Re	Parte real
Im	Parte imaginaria
sen	Función seno
tg	Función tangente
cos	Función coseno
arctg	Función arco tangente
arcsen	Función arco seno
arccos	Función arco coseno
$\sin^x y$	Función seno de *y* elevado a *x*
$\cos^x y$	Función coseno de *y* elevado a *x*
∂y ∂x	Derivada parcial de *y* respecto
<	Menor o igual
>	Mayor o igual

1 INTRODUCCIÓN

1.1 Contexto del proyecto y objetivos

Este proyecto surge de la necesidad de realizar el Trabajo de Fin de Grado para la finalización de mis estudios en el Grado de Ingeniería Electrónica, Robótica y Mecatrónica. Con este fin, me puse en contacto con el profesor Alejandro del Real Torres de la asignatura de tercero Control por Computador. Tras un par de reuniones, en las que terminamos de concretarlo todo, nos ofreció a mí y a un par de compañeros la posibilidad de desarrollar nuestro TFG en su empresa, en algún tema relacionado con nuestros estudios y que les fuera de utilidad a ellos. Finalmente, el profesor nos habló de la posibilidad de realizar el TFG sobre las redes neuronales, ya que éstas le serían de utilidad en futuros proyectos de la empresa. Tras investigar qué eran las redes neuronales y para qué servían, decidí que sería un proyecto muy interesante en el que embarcarme y que me aportaría mucho a nivel académico. Así, acepté este proyecto.

La idea que se persigue con la realización de este proyecto es la de profundizar en el campo de las redes neuronales y, más concretamente, en el uso de un tipo en concreto: las redes neuronales convolucionales. Éstas, son un tipo de red neuronal artificial donde las neuronas se corresponden a campos receptivos de una manera muy similar a las neuronas en la corteza visual primaria de un cerebro biológico. Por lo que son muy efectivas en el campo de la visión artificial, cuya aplicación más destacada es el reconocimiento y clasificación de imágenes o de patrones en la imagen.

En concreto, para implementar una red convolucional, entrenarla y probarla se buscará realizar un sistema que sea capaz de reconocer e identificar caracteres (en concreto, dígitos del 0 al 9) escritos a mano. Para esto, la entrada de la red serán imágenes de $28\ x\ 28$ píxeles en escala de grises sacadas de la base de datos MNIST, que contiene 60,000 caracteres para entrenar y 10,000 para comprobar el correcto funcionamiento de ésta. El programa que se desarrolle estará basado en el lenguaje de programación R y se buscará que, al menos, tenga un 95 % de acierto en la fase de testeo de las 10,000 imágenes.

En esta memoria, se explicarán las bases de las redes neuronales y del entrenamiento de las mismas, para tener una idea básica y simple de cómo funcionan y se comportan, y, posteriormente, se hará un análisis profundo de las redes convolucionales y su arquitectura. Finalmente, se verá como se ha implementado todo lo anterior para conseguir el objetivo de reconocer correctamente los caracteres.

1.2 Herramientas usadas

1.2.1 Hardware

A lo largo del desarrollo del proyecto se han utilizado tan solo dos dispositivos hardware diferentes: un ordenador de sobremesa y un ordenador portátil.

En primer lugar, se usó un ordenador de sobremesa proporcionado por la empresa, sin embargo, cuando comenzó la fase de prueba del programa, este empezó a dar muchos problemas por el tamaño de las matrices y vectores que la aplicación manejaba. Debido a la falta de memoria, el entrenamiento de la red neuronal llevaba un largo tiempo o incluso directamente fallaba y esto retrasaba, en gran medida, el poder depurar el programa y solucionar todos los fallos que iban apareciendo. De este dispositivo no se poseen actualmente los datos para poder especificarlos aquí.

En segundo lugar, debido a los problemas de memoria y de procesamiento del ordenador anterior, se hizo uso de un ordenador portátil propio. Finalmente, en este ordenador se pudo terminar de desarrollar todo el trabajo de una manera satisfactoria. Los componentes del ordenador portátil, de la marca MSI, son los siguientes:

- Procesador: Intel Core i5-4210H 2.90 GHz
- Disco duro: 1 terabyte HDD
- Memoria RAM: 8 GB DDR3 798MHz

- Tarjeta gráfica: NVIDIA GeForce GTX 850M

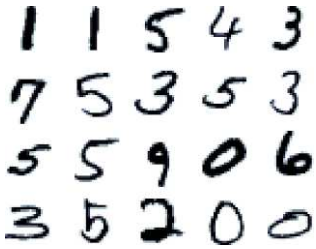
Cabe destacar, como se comentará más adelante, que con este portátil que es de gama media, se ha conseguido llevar a cabo el entrenamiento y la comprobación del buen funcionamiento de la red neuronal en 1 hora aproximadamente. Esto es un claro indicador del buen desempeño del programa creado: con un ordenador de sobremesa potente el entrenamiento de 60,000 imágenes y el posterior testeo de otras 10,000 sería cuestión de pocos minutos.

1.2.2 Software

Para la realización de este proyecto se han usado una serie de herramientas y programas, además de una base de datos con las imágenes para el entrenamiento.

- 'RStudio®' es un entorno de desarrollo integrado (IDE) para R (lenguaje de programación). Incluye una consola, editor de sintaxis que apoya la ejecución de código, así como herramientas para el trazado, la depuración y la gestión del espacio de trabajo. En esta plataforma se ha programado toda la red convolucional.

- 'La base de datos MNIST' (Modified National Institute of Standards and Technology database) es una extensa base de datos compuesta por caracteres escritos a mano que es usada comúnmente para el entrenamiento de sistemas de procesamientos de imágenes. También es muy usada para el entrenamiento y testeo en el campo de la inteligencia artificial. En este proyecto se usará para entrenar y probar la red convolucional. En concreto, se usará en formato csv.



- 'Google Drive' es un servicio de alojamiento o almacenamiento de archivos que fue introducido por Google el 24 de abril de 2012. En este proyecto se ha usado para guardar de manera segura los documentos, los scripts, la base de datos con las imágenes y, en general, todos los elementos que conforman el trabajo completo en la nube. De esta manera, si algún ordenador fallará, existiría en todo momento una copia actualizada de todo el proyecto en Internet y tan solo habría que descargarla.

- 'Pracma' (Practical Numerical Math Function) es una librería para R que provee de un gran número de funciones matemáticas. En concreto, en este proyecto se ha usado para rotar matrices.

2 INTRODUCCIÓN A LAS REDES NEURONALES

2.1 Introducción

Las redes neuronales biológicas son inmensas redes de neuronas interconectadas mediante procesos químicos y eléctricos. Todas esas neuronas conectadas entre sí le permiten al ser humano ser capaz de sentir, memorizar, aprender, etc. Todo esto conlleva preguntarse si es posible construir o crear algún tipo rudimentario de red neuronal y entrenarla para que sea capaz de llevar a cabo determinadas tareas.

La respuesta a la pregunta anterior es sí, y el objetivo principal de este apartado es el de realizar simples abstracciones de las complejas redes neuronales para poder implementarlas y entrenarlas para resolver problemas específicos.

2.2 Inspiración biológica

Como se ha comentado con anterioridad el modelo simplificado y abstracto de las redes neuronales artificiales surge de intentar imitar el comportamiento de las neuronas que se encuentran en el cerebro.

En esencia el cerebro está formado por un número ingente de neuronas (aproximadamente 10^{11}) conectadas entre sí creando la red neuronal. Las neuronas tienen tres partes principales: las dendritas, el cuerpo y el axón. Las dendritas son fibras nerviosas que transportan los impulsos eléctricos al cuerpo de la neurona. El cuerpo de la neurona se encarga de reconocer y trabajar con las señales que le llegan. Por último, el axón es una única fibra nerviosa que comunica el cuerpo de la neurona con las demás. El punto de contacto entre el axón y la dendrita de otra neurona se denomina sinapsis. La disposición de las neuronas y las fuerzas de cada sinapsis individualmente, determinada por reacciones químicas, establecen el funcionamiento del cerebro. Esta estructura de las neuronas se puede observar en la Figura 2-1.

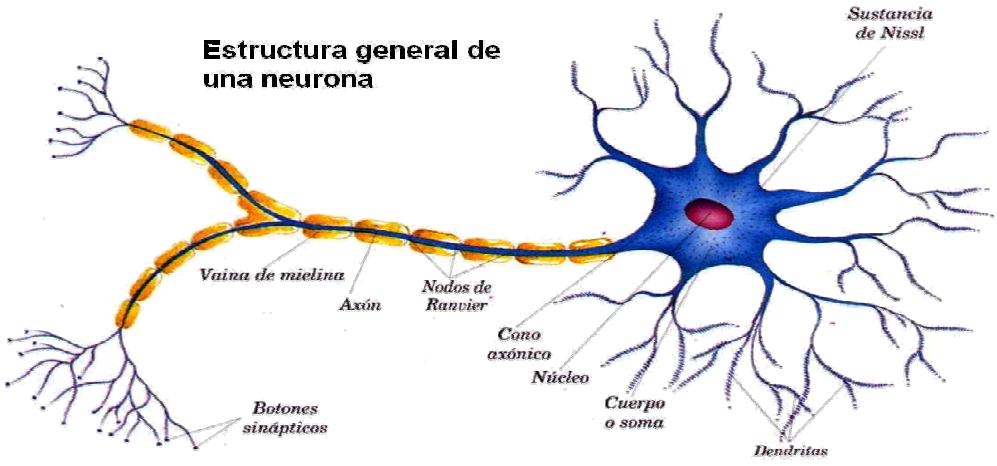


Figura 2-1. Estructura general de una neurona biológica.

Parte de la red neuronal que forma nuestro cerebro es determinada cuando un ser humano nace, sin embargo, muchas otras partes se desarrollan a lo largo del tiempo. A través del aprendizaje esas partes se desarrollan creando nuevas conexiones entre neuronas o modificando las existentes, fortaleciéndose o debilitándose.

Aun teniendo en cuenta las grandes diferencias las redes biológicas y las artificiales comparten ciertas características. En primer lugar, comparten la estructura general en la que las neuronas están poseen muchas conexiones entre ellas. En segundo lugar, la fuerza en las conexiones entre neuronas determina el comportamiento de la red completa.

2.3 Historia de las redes neuronales

Conseguir diseñar y construir máquinas capaces de realizar procesos con cierta inteligencia ha sido uno de los principales objetivos de los científicos a lo largo de la historia. De los intentos realizados en este sentido se han llegado a definir las líneas fundamentales para la obtención de máquinas inteligentes: En un principio los esfuerzos estuvieron dirigidos a la obtención de autómatas, en el sentido de máquinas que realizaran, con más o menos éxito, alguna función típica de los seres humanos. Hoy en día se continúa estudiando en ésta misma línea, con resultados sorprendentes, existen maneras de realizar procesos similares a los inteligentes y que podemos encuadrar dentro de la llamada Inteligencia Artificial (IA).

La otra línea de la investigación ha tratado de aplicar principios físicos que rigen en la naturaleza para obtener máquinas que realicen trabajos pesados en nuestro lugar. De igual manera se puede pensar respecto a la forma y capacidad de razonamiento humano; se puede intentar obtener máquinas con esta capacidad basadas en el mismo principio de funcionamiento.

No se trata de construir máquinas que compitan con los seres humanos, sino que realicen ciertas tareas de rango intelectual con que ayudarle, principio básico de la Inteligencia Artificial.

Las primeras explicaciones teóricas sobre el cerebro y el pensamiento ya fueron dadas ya por Platón (427-347 a.C.) y Aristóteles (348-422 a.C.). Las mismas ideas también las mantuvo Descartes (1569-1650) y los filósofos empiristas del siglo XVIII.

La clase de las llamadas máquinas cibernéticas, a la cual la computación neuronal pertenece, tiene más historia de la que se cree: Herón (100 a.C) construyó un autómata hidráulico.

1936 - Alan Turing. Fue el primero en estudiar el cerebro como una forma de ver el mundo de la computación. Sin embargo, los primeros teóricos que concibieron los fundamentos de la computación neuronal fueron Warren McCulloch, un neurofisiólogo, y Walter Pitts, un matemático, quienes, en 1943, lanzaron una teoría acerca de la forma de trabajar de las neuronas (Un Cálculo Lógico de la Inminente Idea de la Actividad Nerviosa - Boletín de Matemática Biofísica 5: 115-133). Ellos modelaron una red neuronal simple mediante circuitos eléctricos.

1949 - Donald Hebb. Escribió un importante libro: La organización del comportamiento, en el que se establece una conexión entre psicología y fisiología. Fue el primero en explicar los procesos del aprendizaje (que es el elemento básico de la inteligencia humana) desde un punto de vista psicológico, desarrollando una regla de como el aprendizaje ocurría. Aun hoy, este es el fundamento de la mayoría de las funciones de aprendizaje que pueden hallarse en una red neuronal. Su idea fue que el aprendizaje ocurría cuando ciertos cambios en una neurona eran activados. También intentó encontrar semejanzas entre el aprendizaje y la actividad nerviosa. Los trabajos de Hebb formaron las bases de la Teoría de las Redes Neuronales.

1950 - Karl Lashley. En sus series de ensayos, encontró que la información no era almacenada en forma centralizada en el cerebro, sino que era distribuida encima de él.

1956 - Congreso de Dartmouth. Este Congreso frecuentemente se menciona para indicar el nacimiento de la inteligencia artificial.

1957 - Frank Rosenblatt. Comenzó el desarrollo del Perceptrón. Esta es la red neuronal más antigua; utilizándose hoy en día para aplicación como reconocedor de patrones. Este modelo era capaz de generalizar, es decir, después de haber aprendido una serie de patrones podía reconocer otros similares, aunque no se le hubiesen presentado anteriormente. Sin embargo, tenía una serie de limitaciones, por ejemplo, su incapacidad para resolver el problema de la función OR-exclusiva y, en general, era incapaz de clasificar clases no separables linealmente. En 1959, escribió el libro Principios de Neuro Dinámica, en el que confirmó que, bajo ciertas condiciones, el aprendizaje del Perceptrón convergía hacia un estado finito (Teorema de Convergencia del Perceptrón).

1960 - Bernard Widrow/Marcial Hoff. Desarrollaron el modelo Adaline (ADAptative LINear Elements). Esta fue la primera red neuronal aplicada a un problema real (filtros adaptativos para eliminar ecos en las líneas telefónicas) que se ha utilizado comercialmente durante varias décadas.

1961 - Karl Steinbeck: Die Lernmatrix. Red neuronal para simples realizaciones técnicas (memoria asociativa).

1967 - Stephen Grossberg. A partir de sus conocimientos fisiológicos, ha escrito numerosos libros y

desarrollado modelo de redes neuronales. Realizó una red: Avalancha, que consistía en elementos discretos con actividad que varía en el tiempo que satisface ecuaciones diferenciales continuas, para resolver actividades como reconocimiento continuo de habla y aprendizaje de los brazos de un robot.

1969 - Marvin Minsky/Seymour Papert. En este año surgieron críticas que frenaron, hasta 1982, el crecimiento que estaban experimentando las investigaciones sobre redes neuronales. Minsky y Papera, del Instituto Tecnológico de Massachussets (MIT), publicaron un libro Perceptrons. Probaron (matemáticamente) que el Perceptrón no era capaz de resolver problemas relativamente fáciles, tales como el aprendizaje de una función no-lineal. Esto demostró que el Perceptrón era muy débil, dado que las funciones no-lineales son extensamente empleadas en computación y en los problemas del mundo real. A pesar del libro, algunos investigadores continuaron su trabajo. Tal fue el caso de James Anderson, que desarrolló un modelo lineal, llamado Asociador Lineal, que consistía en unos elementos integradores lineales (neuronas) que sumaban sus entradas. Este modelo se basa en el principio de que las conexiones entre neuronas son reforzadas cada vez que son activadas. Anderson diseñó una potente extensión del Asociador Lineal, llamada Brain State in a Box (BSB).

1974 - Paul Werbos. Desarrolló la idea básica del algoritmo de aprendizaje de propagación hacia atrás (backpropagation); cuyo significado quedó definitivamente aclarado en 1985.

1977 - Stephen Grossberg. Teoría de Resonancia Adaptada (TRA). La Teoría de Resonancia Adaptada es una arquitectura de red que se diferencia de todas las demás previamente inventadas. La misma simula otras habilidades del cerebro: memoria a largo y corto plazo.

1977 - Teuvo Kohonen. Ingeniero electrónico de la Universidad de Helsinki, desarrolló un modelo similar al de Anderson, pero independientemente.

1980 - Kunihiko Fukushima. Desarrolló un modelo neuronal para el reconocimiento de patrones visuales.

1985 - John Hopfield. Provocó el renacimiento de las redes neuronales con su libro: "Computación neuronal de decisiones en problemas de optimización."

1986 - David Rumelhart/G. Hinton. Redescubrieron el algoritmo de aprendizaje de propagación hacia atrás (backpropagation).

A partir de 1986, el panorama fue alentador con respecto a las investigaciones y el desarrollo de las redes neuronales. En la actualidad, son numerosos los trabajos que se realizan y publican cada año, las aplicaciones nuevas que surgen (sobre todo en el área de control) y las empresas que lanzan al mercado productos nuevos, tanto hardware como software (sobre todo para simulación).

2.4 Modelos neuronales

2.4.1 Modelo de una única neurona

Para explicar el funcionamiento de las redes neuronales se comenzará describiendo la red neuronal más simple que existe, una red formada por una única neurona y una única entrada. El esquema de esta red es el representado en la Figura 2-2.

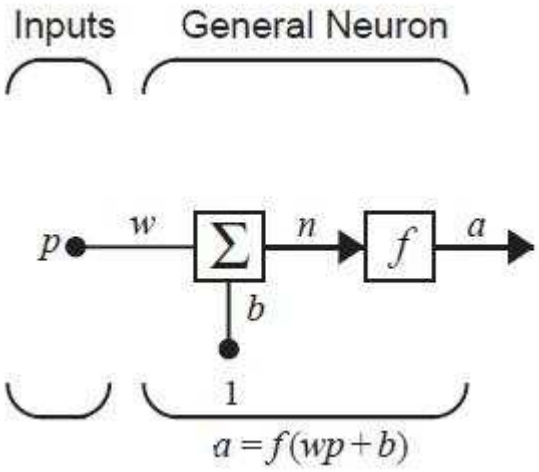


Figura 2-2. Red neuronal de una única neurona y una entrada.

El proceso que se sigue para obtener la salida de la red es simple. A la entrada llega un escalar p que es multiplicado por su peso w, quedando w_p, que es una de las entradas al sumatorio. La otra entrada siempre tiene el valor 1 y es multiplicado por una bias b. Una vez ambos términos se han sumado se obtiene que la entrada n a la función de transferencia es $w_p + b$. Finalmente, a la salida de la función de transferencia se tiene la salida a de la neurona:

$$a = f(wp + b). \tag{2-1}$$

Teniendo esto en cuenta, se puede observar que la salida del sistema depende del tipo de función de transferencia que se tenga. Es importante conocer que la función de transferencia la decide el diseñador y que los parámetros w y b son ajustables y calculados mediante una regla de aprendizaje. Existen múltiples tipos de funciones de transferencia y en una red neuronal se elige intentando satisfacer las especificaciones concretas del problema que se intenta resolver. A continuación, en la Tabla 1.1 se muestran las principales funciones de transferencia.

Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n<0$ $a = 1 \quad n\geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n<0$ $a = +1 \quad n\geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n<0$ $a = n \quad 0\leq n\leq 1$ $a = 1 \quad n>1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n<-1$ $a = n \quad -1\leq n\leq 1$ $a = 1 \quad n>1$		satlins
Log-Sigmoid	$a = \frac{1}{1+e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n<0$ $a = n \quad 0\leq n$		poslin
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons	C	compet

Tabla 2-1. Funciones de transferencia de las redes neuronales.

Normalmente una neurona posee más de una entrada. En la Figura 2-3 se muestra una neurona con R entradas, cada $p_1, p_2, p_3, \dots, p_R$ es multiplicada por su correspondiente peso $w_{11}, w_{12}, \dots, w_{1R}$ y después es sumada con la bias.

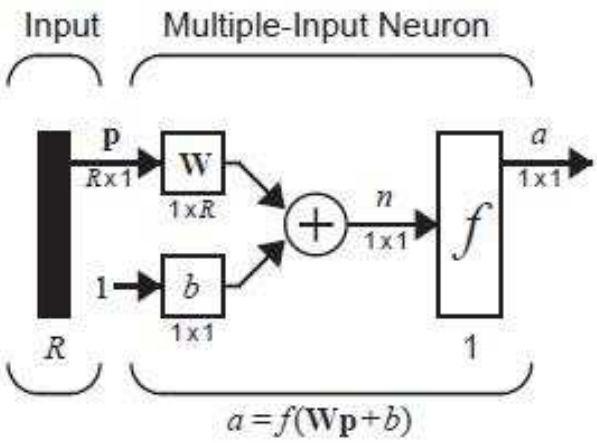


Figura 2-3. Red neuronal de una única neurona y varias entradas

Haciendo un procedimiento análogo al de la neurona con una sola entrada y teniendo en cuenta que por simplicidad de cálculo se trabajará matricialmente, se obtiene que la salida del sistema es:

$$a = f(\boldsymbol{W}\boldsymbol{p} + b). \qquad (2\text{–}2)$$

2.4.2 Modelo de red neuronal

Usualmente con una única neurona no será suficiente para resolver la mayoría de problemas prácticos. Para resolver problemas más complejos se tendrá que hacer un uso conjunto de muchas de estas neuronas simples, dando lugar a una verdadera red neuronal, en la que se tendrán cientos o incluso miles de neuronas. Es ahí donde aparece el concepto de capa, que es la agrupación de todas estas neuronas en varios conjuntos dentro de la red neuronal completa. En la Figura 2-4 se puede observar una red neuronal con múltiples capas.

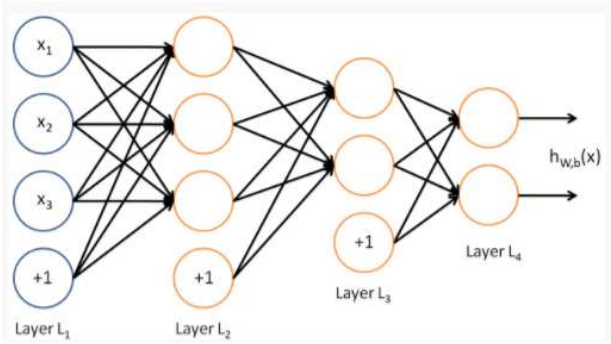


Figura 2-4. Esquema general de una red neuronal de múltiples capas.

Si se considera una red neuronal con varias capas y se tiene en cuenta que cada capa tiene sus propias matrices de peso, sus propios vectores de bias y sus respectivas salidas es necesario introducir una nueva notación para poder trabajar correctamente.

En primer lugar, se tiene que existirán n_l capas en la red neuronal. La *l*-ésima capa se denominará L_l, así la capa de entrada a la red será L_1 y la capa de salida de la red será L_{n_l}. También se tendrá que $w_{ij}^{(l)}$ será el peso asociado a la conexión entre el elemento *j* de la capa *l* y el elemento *i* de la capa *l+1*. Además, $b_i^{(l)}$ es la bias asociada con el elemento *i* de la capa *l+1*. Por último, se usará $a_i^{(l)}$ para denotar la activación (es decir, el valor de salida) del elemento *i* en la capa *l*.

En los elementos propios de cada capa se usa un superíndice que indica en cuál de ellas se está, de esta manera si el superíndice es uno el elemento pertenece a la primera capa, si es dos pertenece a la segunda y así sucesivamente. Es interesante destacar que las entradas a las capas posteriores son las salidas de las capas que les preceden.

A continuación, para explicar las ecuaciones que rigen estos sistemas se usará una red neuronal muy simple que solo consta de 3 capas. Ésta puede verse en la Figura 2-5.

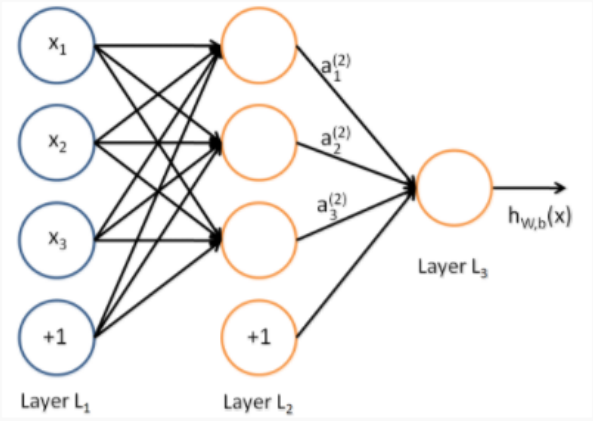


Figura 2-5. Esquema de red neuronal de tres capas.

Como se ha especificado antes, la red consta de 3 capas. La primera de ella es la capa de entrada, situada a la izquierda en la imagen y representada por los círculos azules. La capa de salida de la red es la última, la que está situada más a la derecha. Como se comentó con anterioridad, los círculos que tienen un +1 en su interior son denominados bias. La capa restante es la que se encuentra en medio de las dos anteriores. Todas las capas intermedias (las que se encuentra entre la capa de entrada y la de salida) de las redes neuronales se denominan capas ocultas, en la Figura 2-5 solo hay una. Estas capas se denominan así debido a que sus valores no se observan durante el entrenamiento.

También se puede observar que cada elemento del vector de entrada $\boldsymbol{x}$ está conectado con cada una de las neuronas a través de la matriz de pesos $\boldsymbol{W}$, es decir cada neurona de cada capa está conectada con todos los elementos de la capa que le precede. Este tipo de capas se denomina totalmente conectada o, en inglés, full-connected. Además, cada neurona posee su propia bias. Todo esto supone que la salida de la neurona $n_i$vendrá dada por la suma ponderada entre cada elemento de la entrada multiplicado por su correspondiente elemento de la matriz de pesos y la bias, o lo que es lo mismo, como en las ecuaciones (2-3):

$$\begin{aligned} a_1^{(2)} &= f\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)}\right), \\ a_2^{(2)} &= f\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)}\right), \\ a_3^{(2)} &= f\left(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{S3}^{(1)}x_3 + b_3^{(1)}\right), \\ h_{W,b}(x) = a_1^{(3)} &= f(w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)} + w_{13}^{(2)}a_3^{(2)} + b_1^{(2)}) \end{aligned} \qquad (2\text{–}3)$$

Lo anterior se puede resumir como la suma ponderada de todas las entradas a la i-ésima neurona de la capa *l*, incluyendo el término de bias. Por ejemplo, para el caso anterior, $z_i^{(2)} = \sum_{j=1}^{n} w_{ij}^{(1)}x_j + b_i^{(1)}$, o lo que es lo mismo, $a_i^{(l)} = f\left(z_i^{(l)}\right)$. Se puede reescribir lo anterior de una manera más compacta expresándolo de manera matricial, teniendo en cuenta lo siguiente:

$$\begin{aligned} \boldsymbol{W} &= \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}, \\ \boldsymbol{x} &= \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \\ \boldsymbol{b} &= \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \end{aligned} \qquad (2\text{–}4)$$

Entonces, la expresión (2-3) se reescribe como (2-5):

$$\begin{aligned} \boldsymbol{z}^{(2)} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \\ \boldsymbol{a}^{(2)} &= f\left(\boldsymbol{z}^{(2)}\right) \\ \boldsymbol{z}^{(3)} &= \boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)} \\ h_{W,b}(\boldsymbol{x}) &= \boldsymbol{a}^{(3)} = f\left(\boldsymbol{z}^{(3)}\right) \end{aligned} \qquad (2\text{–}5)$$

Al paso dado en la ecuación (2-5) se le denomina propagación hacia delante, o forward propagation en inglés.

Por otro lado, sabiendo que $a^{(1)} = x$, es decir, que se usa $a^{(1)}$ para referirse a los valores de la capa de entrada, se pueden expresar las activaciones de la capa *l+1*, dadas las activaciones de la capa anterior, como la expresión (2-6):

$$\begin{aligned} \boldsymbol{z}^{(l+1)} &= \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)} \\ \boldsymbol{a}^{(l+1)} &= f\left(\boldsymbol{z}^{(l+1)}\right) \end{aligned} \qquad (2\text{–}6)$$

El objetivo que se persigue normalmente con agrupar los términos y usar notación matricial es aumentar la velocidad de cálculo del sistema ya que, gracias a ello, se pueden usar propiedades propias del álgebra lineal que permiten realizar todas las operaciones necesarias en un tiempo mucho más reducido.

2.5 Ejemplo de funcionamiento de una red neuronal

El objetivo que se persigue en este punto es el de explicar el comportamiento que sigue una red neuronal, para conseguir esto se hará uso de un problema de reconocimiento de patrones muy simple y una red neuronal muy sencilla: el perceptrón.

2.5.1 Enunciado del problema

Un vendedor de fruta posee un almacén en la que almacena gran variedad de frutas y vegetales. Cuando las frutas llegan al almacén estas llegan mezcladas y, por lo tanto, el vendedor quiere una máquina que sea capaz de separarlas según una serie de características propias de cada tipo, que son la forma, la textura y el peso. Los sensores que miden estas propiedades proporcionan como respuesta un 1 o un -1. Así tenemos que si es redonda dará un 1 y si es elíptica tendrá un -1. Si la superficie es lisa del sensor de textura saldrá un 1 y si es rugosa saldrá un -1. Por último, si la fruta pesa más de un kilogramo a la salida pondrá un 1 y si pesa menos un -1.

Así el propósito de la red neural es decidir qué tipo de fruta es la que entra al almacén para poder dividirla correctamente. Por simplicidad se tomarán solo dos tipos de fruta: manzanas y naranjas.

Las características de cada fruta se pueden agrupar en un vector con 3 componentes:

$$\boldsymbol{p} = \begin{bmatrix} Forma \\ Textura \\ Peso \end{bmatrix}. \qquad (2\text{–}7)$$

El prototipo de una naranja y de una manzana, respectivamente, serían:

$$\begin{aligned} \boldsymbol{p}_1 &= \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \\ \boldsymbol{p}_2 &= \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}. \end{aligned} \qquad (2\text{–}8)$$

2.5.2 Perceptron

Para ilustrar paso por paso el comportamiento de una red neuronal, en primer lugar, se introducirá el perceptrón. El esquema general de un perceptrón está ilustrado en la Figura 2-6.

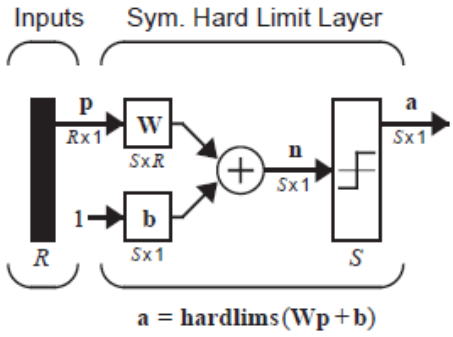


Figura 2-6. Esquema general de un perceptrón.

Caso de dos entradas

Antes de analizar con más profundidad el perceptrón, aplicado al problema de la fruta que requiere la entrada de 3 datos, es recomendable estudiar el comportamiento que ofrece un perceptrón de solo dos entradas (Figura 2-7), que es más fácil de analizar.

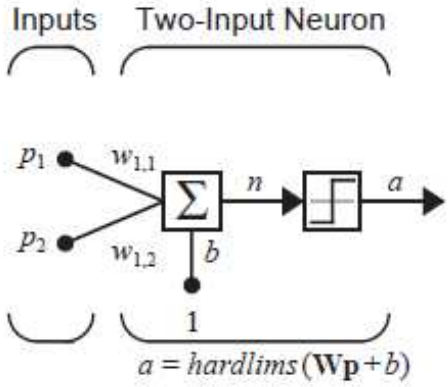


Figura 2-7. Esquema de perceptrón de dos entradas.

Atendiendo a la Figura 2-7, se tiene que a la salida del perceptrón se obtiene la expresión (2-9):

$$a = hardlims(n) = hardlims([w_{11}, w_{22}]\boldsymbol{p} + b). \qquad (2\text{–}9)$$

Para comprender correctamente el resultado anterior, es conveniente definir el concepto de frontera. La frontera es la línea (en este caso concreto es una recta por ser la función hardlims) que divide el espacio de entrada en dos partes. Si los datos de entrada corresponden a un vector definido en uno de los semiplanos, se obtendrá a la salida uno de los resultados posibles (en este caso 1 o -1). Sin embargo, si el vector está definido en el otro semiplano el resultado será el opuesto.

La frontera podrá modificarse variando tanto $\boldsymbol{W}$ como $\boldsymbol{b}$, de esta manera, como se verá más adelante, se puede entrenar a la red neuronal para qué, variando la frontera (a través de $\boldsymbol{w}$ y de b), la red sea capaz de calcular correctamente la salida. Es decir, todos los métodos de entrenamiento que se verán a partir de ahora se basan en este principio básico: si no se obtiene la salida deseada, se modifican tanto los pesos como la bias (el cómo ya depende de cada método en concreto) hasta conseguir que la salida de la red neuronal sea correcta.

En la Figura 2-10, se puede observar un ejemplo específico de frontera para el caso de $w_{11} = -1$ y $w_{22} = 1$.

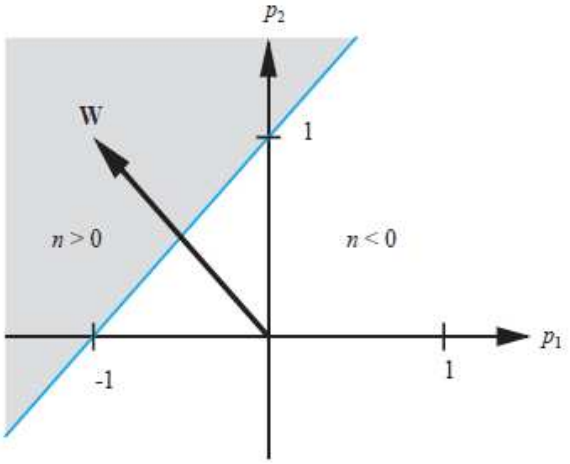


Figura 2-8. Frontera para el caso $w_{11}=-1$ y $w_{22}=1$.

En este caso, la red solo podrá usarse para reconocer patrones que puedan ser separados linealmente, es decir, por una frontera definida por una recta.

Patrón de reconocimiento con el ejemplo

Considerando el ejemplo de la fruta se tiene que el vector de entrada a la red es ahora de 3 componentes. Si ahora queremos definir una frontera que sea capaz de separar las manzanas de las naranjas debe ser un plano. Entonces se tiene que la ecuación de la red es en este caso viene dada por (2-10):

$$a = hardlims\left([w_{11}, w_{22}, w_{33}]\begin{bmatrix}p_1\\p_2\\p_3\end{bmatrix} + b\right). \qquad (2\text{–}10)$$

La frontera resultante será la que divida los dos vectores (los vectores prototipo) simétricamente, es decir, el plano p_1, p_3 este puede ser descrito por:

$$p_2 = 0 \text{ ó } [0\ 1\ 0]\begin{bmatrix}p_1\\p_2\\p_3\end{bmatrix} + 0 = 0. \qquad (2\text{–}11)$$

Así la matriz de peso y la bias quedan:

$$\boldsymbol{W} = [0\ 1\ 0], b = 0 \qquad (2\text{–}12)$$

Cabe destacar que la matriz de peso $\boldsymbol{W}$ es ortogonal a la frontera escogida y que la bias b es nula porque el origen está definido en la frontera.

Ahora se puede corroborar el correcto funcionamiento de la red neuronal probando a introducir los prototipos de cada fruta y ver qué respuesta proporciona.

Naranja:

$$a = hardlims\left([0\ 1\ 0]\begin{bmatrix}1\\-1\\-1\end{bmatrix} + 0\right) = -1 \text{ (correcto).} \qquad (2\text{–}13)$$

Manzana:

$$a = hardlims\left([0\ 1\ 0]\begin{bmatrix}1\\1\\-1\end{bmatrix} + 0\right) = 1 \text{ (correcto).} \qquad \textit{(2-14)}$$

Como se puede ver en las expresiones (2-13) y (2-14) la red ha acertado en ambos casos. Además, se puede comprobar que incluso si los vectores de entrada son parecidos a los patrones y no exactamente iguales la red es capaz de identificarlos correctamente. Como se ha podido ver la principal característica de las redes neuronales creadas a partir de un perceptrón presentan un buen funcionamiento frente a problemas que se pueden resolver mediante fronteras lineales.

3 REGLA DE APRENDIZAJE DEL PERCEPTRÓN

El objetivo que se persigue en este punto es el de explicar de manera clara cómo se lleva a cabo el aprendizaje de las redes neuronales. En concreto, antes de pasar a explicar qué algoritmo se ha usado en el proyecto y cómo funciona, se verá una regla de aprendizaje para una red neuronal muy simple: el perceptrón. Es decir, con este apartado se pretende realizar una introducción al algoritmo de aprendizaje más simple usado en las redes neuronales para posteriormente ser capaces de entender los algoritmos más complejos usados en el desarrollo del trabajo.

Uno de los problemas vistos con anterioridad es cómo se puede determinar la matriz de peso y la "bias" de una red neuronal cuando es imposible determinar la frontera. En este capítulo, se describirá un algoritmo para el entrenamiento de un perceptrón con el objetivo de que aprenda a resolver problemas concretos.

3.1 Tipos de aprendizaje

Una regla de aprendizaje no es más que un procedimiento por el cual se modifican los pesos y las "bias" de la red. El propósito principal de que una red aprenda es que sea capaz de resolver una tarea que antes no podía (sabía) resolver. Existen muchos tipos de métodos de aprendizaje, pero los principales son: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje reforzado.

Aprendizaje supervisado

El aprendizaje supervisado se caracteriza porque el proceso de aprendizaje se realiza mediante un entrenamiento controlado por un agente externo (supervisor, maestro) que determina la respuesta que debería generar la red a partir de una entrada determinada. El supervisor controla la salida de la red y en caso de que ésta no coincida con la deseada, se procederá a modificar los pesos de las conexiones, con el fin de conseguir que la salida obtenida se aproxime a la deseada.

Para lograr lo anterior, la red es provista con una seria de ejemplos que indican como la red debe funcionar, los pares de entrada y objetivo vienen dados por la expresión 3-1:

$$\{p_1, t_1\}, \{p_2, t_2\}, \dots, \{p_Q, t_Q\}. \tag{3-1}$$

Cada p_i se corresponde con un vector de entrada al sistema del cual se conoce su correspondiente salida u objetivo, es decir, el vector t_i. Por lo tanto, si la salida del sistema no es igual que el objetivo que se espera que salga se variará tanto la matriz de peso como la "bias" para intentar acercarse al resultado correcto.

En este proyecto, este será el tipo de aprendizaje usado ya que la red será entrenada con imágenes de números sabiendo a priori qué número se corresponde con cada imagen.

Aprendizaje reforzado

Es parecido al caso anterior, excepto que en vez de tener un objetivo asociado a cada entrada tiene una nota. Una nota (o puntuación) es una medida del rendimiento de la red ante varias secuencias de entrada. Dependiendo de la nota la red actuará de una manera u otra: si la nota es baja, el algoritmo de aprendizaje modificará en gran medida los elementos de la matriz de pesos y la bias, si la nota es media, los modificará más suavemente, y si es buena, los mantendrá o apenas modificará.

Aprendizaje no supervisado

Son aquellos en los que no disponemos de una batería de ejemplos previamente clasificados, si no que la matriz de peso y la *bias* son modificadas en respuesta solo a las entradas de la red, ya que no existen objetivos. De esta manera, los algoritmos trabajan agrupando, es decir, la red aprende a categorizar los patrones de entrada en una serie finita de clases, según las propiedades de los ejemplos buscando la similitud.

3.2 Arquitectura del perceptrón

Previamente a realizar el estudio del método de aprendizaje del perceptrón es conveniente realizar un análisis más exhaustivo del esquema del perceptrón (Figura 3-1) visto en el capítulo 2.

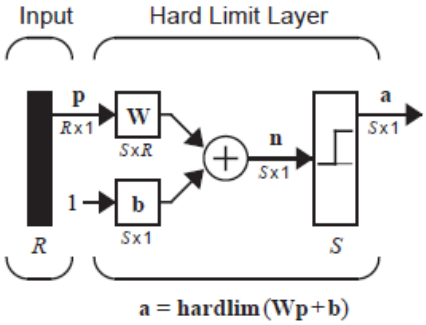


Figura 3-1. Esquema de un perceptrón.

En primer lugar, para trabajar con mayor facilidad se considera la matriz de pesos:

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1R} \\ w_{21} & w_{22} & & w_{2R} \\ \vdots & & \ddots & \vdots \\ w_{S1} & w_{S2} & \cdots & w_{SR} \end{bmatrix}. \tag{3-2}$$

Se define un vector compuesto por los elementos de i-ésima fila de **W**:

$$\boldsymbol{W} = \begin{bmatrix} {}_1w^T \\ {}_2w^T \\ \vdots \\ {}_Sw^T \end{bmatrix}. \tag{3-3}$$

Recordando que la salida de la red que se vio en el capítulo 2 es:

$$\mathbf{a} = hardlim(\mathbf{Wp} + \mathbf{b}). \tag{3-4}$$

Ahora se puede expresar de la siguiente forma:

$$a_i = hardlim(n_i) = \mathrm{h}ardlim({}_iw^T\boldsymbol{p} + b). \tag{3-5}$$

3.2.1 Perceptrón de una sola neurona

Considerando el modelo de perceptrón con una sola neurona y dos entradas (Figura 3-2) tenemos que la salida del sistema es:

$$\begin{aligned} \boldsymbol{a} &= hardlim(n) = hardlim(\boldsymbol{W}\boldsymbol{p} + b) = \\ &= hardlim({}_1\boldsymbol{w}^T\boldsymbol{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b). \end{aligned} \tag{3-6}$$

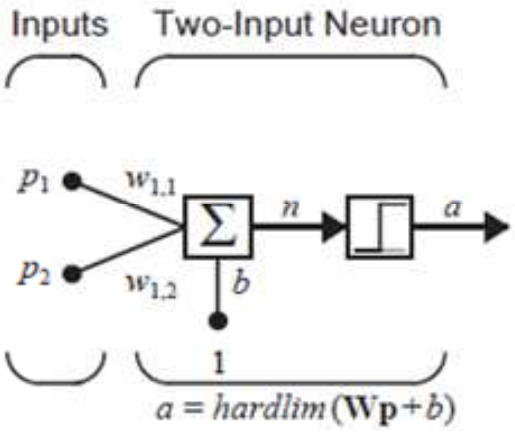


Figura 3-2. Esquema de un perceptrón de dos entradas y una neurona.

Considerando esa salida de la red neuronal, la frontera se determina mediante los vectores de entrada a la red que hacen que el vector *n* sea 0. Es decir,

$$n = {}_1\boldsymbol{w}^T\boldsymbol{p} + b = w_{1,1}\,p_1 + w_{1,2}\,p_2 + b = 0. \tag{3-7}$$

Si tomamos como matriz de peso y como bias, por ejemplo:

$$w_{1,1} = 1,\ w_{1,2} = 1,\ b = -1. \tag{3-8}$$

La frontera resultante será la recta que se puede ver en la Figura 3-3. Para saber en que lado de la frontera 'a' es igual a 1 y en que lado es igual a 0, se coge un punto perteneciente a cualquiera de los dos y se prueba. Por ejemplo, para la entrada $\boldsymbol{p} = [2\ 0]$ se tendrá que a = 1. Por lo que a la derecha de la frontera se da a = 1 y a la izquierda a = 0.

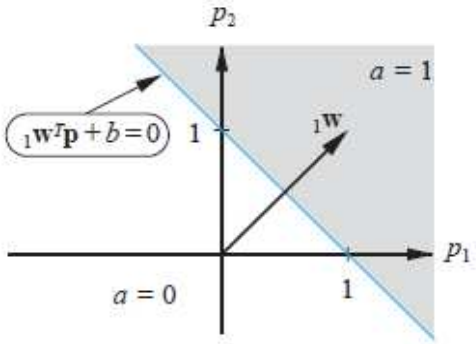


Figura 3-3. Frontera resultante de $w_{1,1} = 1$, $w_{1,2} = 1$, $b = -1$.

3.2.2 Perceptrón de varias neuronas

Los perceptrones de varias neuronas tendrán una frontera para cada una de las neuronas, así se tiene que la frontera para la i-ésima neurona se define por:

$${}_i w^T\boldsymbol{p} + b = 0. \tag{3-9}$$

Un perceptrón simple puede clasificar un vector de entrada en dos categorías (0 o 1), sin embargo, un perceptrón con múltiples neuronas es capaz de diferenciar entre múltiples categorías. Si el número de neuronas es *S* el número de categorías posibles es 2*S*.

3.3 Regla de aprendizaje del perceptrón

Como se ha comentado con anterioridad en este capítulo el modelo de aprendizaje que se usa en este proyecto es el de entrenamiento o aprendizaje supervisado. En este punto, se verá un problema sencillo donde se aplicará aprendizaje supervisado a un perceptrón, con el que se explicará las bases del aprendizaje de las redes neuronales.

3.3.1 Problema de ejemplo

Para este ejemplo se tendrán los pares de entrada y objetivo que vienen expresados en la ecuación (3-10), con estos, la red será entrenada:

$$\left\{p_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\}, \left\{p_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\} \; y \; \left\{p_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}. \tag{3-10}$$

Se puede ver el problema gráficamente en la Figura 3-4. Los círculos representados con un círculo blanco son aquellos cuyo objetivo es 0, los círculos negros representan lo contrario.

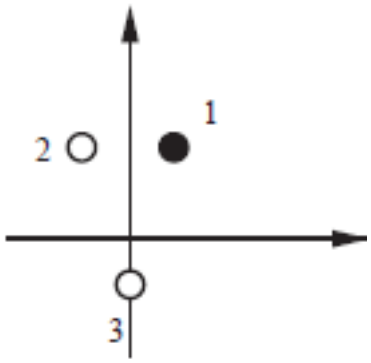


Figura 3-4. Representación del problema

En este caso, la red tendrá solo dos entradas y una salida, además, con objeto de facilitar la comprensión, no se tendrá en cuenta la existencia de la "bias". El perceptrón quedará entonces como en la Figura 3-5.

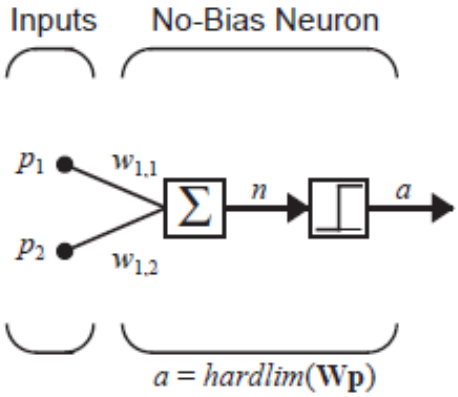


Figura 3-5. Esquema del perceptrón usado para el problema.

Al quitar la *bias*, la frontera debe pasar por el origen, ya que, como se comentó antes, la ecuación que rige este modelo de perceptrón es una recta. En el problema descrito con anterioridad, se puede comprobar que, afortunadamente, esta limitación no influye y se pueden conseguir infinitas fronteras que resuelvan de manera correcta el problema. Esto se ilustra en la Figura 3-6.

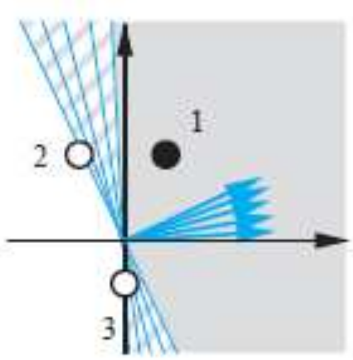


Figura 3-6. Infinitas fronteras posibles que solucionan el problema.

3.3.2 Regla de aprendizaje

El entreno del perceptrón se inicia asignando unos valores iniciales a los parámetros de la red, tanto a la matriz de peso ***w*** como a la *bias b*. En el problema solo se tendrá matriz de peso, los valores serán escogidos aleatoriamente con lo cual resulta que, por ejemplo:

$$_1\boldsymbol{w}^T = [1 \; -0.8]. \tag{3-11}$$

Probando ahora con el primer vector de entrada da como resultado la expresión (3-12):

$$a = hardlim(\,_1w^T p_1) = hardlim\left([1 \;\; -0.8]\begin{bmatrix}1\\2\end{bmatrix}\right) = hardlim(-0.6) = 0. \tag{3-12}$$

Con el resultado obetenido en la ecuación (3-12) se puede comprobar que el resultado no es correcto, ya que debería ser 1. Es decir, la respuesta del sistema es incorrecta y esto es debido a que la frontera que resulta de los parámetros, que se han tomado aleatoriamente, clasifica al vector de entrada en el semiplano incorrecto. Para arreglar esto, y así mejorar el comportamiento de la red neuronal, se realiza el proceso de entrenamiento con el cual se modifica la frontera y se ajusta hasta que proporciona la salida deseada. En la Figura 3-7 se puede apreciar cómo la frontera no cumple con las especificaciones.

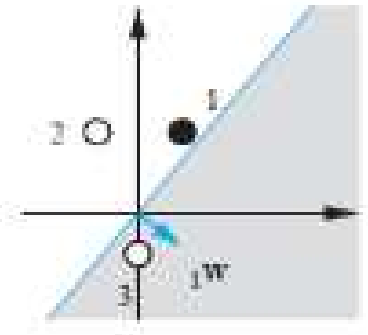


Figura 3-7. Frontera que no cumple especificaciones.

Para solucionar este inconveniente se suma p_1 a $_1w$. Entonces, si a la entrada se sitúa el mismo vector p_1 repetidamente, causará que el vector $_1w$ tienda asintóticamente a p_1. Esta regla se puede expresar como:

$$If\ t = 1\ and\ a = 0,\ then\ \ _1\boldsymbol{w}^{new} = \ _1\boldsymbol{w}^{old} + \boldsymbol{p}. \tag{3-13}$$

Además, si el resultado que se debe obtener es un 0 pero, sin embargo, la salida que se tiene es un 1, se realiza el proceso inverso, se resta:

$$If\ t = 0\ and\ a = 1,\ then\ \ _1\boldsymbol{w}^{new} = \ _1\boldsymbol{w}^{old} - \boldsymbol{p}. \tag{3-14}$$

Por último, si la salida coincide con el objetivo que buscamos:

$$If\ t = 0\ and\ a = 1,\ then\ {}_1\boldsymbol{w}^{new} = {}_1\boldsymbol{w}^{old} - \boldsymbol{p}. \tag{3-15}$$

Si se realiza este proceso para todos los vectores de entrada hasta que todos ellos obtengan la salida correcta, se conseguirá la frontera adecuada, es decir, los parámetros $\boldsymbol{W}$ y $\boldsymbol{b}$ que resuelven el problema específico. Como se puede observar, el entrenamiento consiste, entonces, en un proceso iterativo que no finaliza hasta que la respuesta de la red, ante todos los vectores de entrada, no es la deseada. En el problema, si se realizará el proceso de entrenamiento completo, finalmente quedaría la frontera representada en la Figura 3-8.

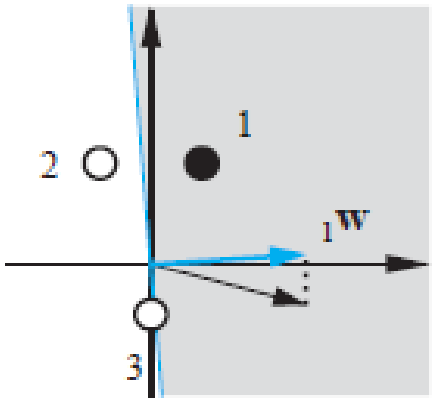


Figura 3-8. Resultado final del problema con frontera correcta.

3.3.3 Regla de aprendizaje unificada

Las ecuaciones 3-13, 3-14 y 3-15 pueden unificarse en una sola expresión. Para comenzar se define el error del perceptrón *e* como:

$$e = t - a. \tag{3-16}$$

Con lo cual es posible reescribir las expresiones del apartado anterior como en (3-17):

$$\begin{aligned} If\ e = 1,\ then\ {}_1\boldsymbol{w}^{new} &= {}_1\boldsymbol{w}^{old} + p, \\ If\ e = -1,\ then\ \mathbf{1}\boldsymbol{w}^{new} &= \mathbf{1}\boldsymbol{w}^{old} - p, \\ If\ e = 0,\ then\ \mathbf{1}\boldsymbol{w}^{new} &= {}_1\boldsymbol{w}^{old}. \end{aligned} \tag{3-17}$$

Esto puede ser reducido a una sola expresión teniendo en cuenta que el signo de *e* es el mismo que el de p,

$${}_1\boldsymbol{w}^{new} = {}_1\boldsymbol{w}^{old} + e\boldsymbol{p} = \mathbf{1}\boldsymbol{w}^{old} + (t - a)\boldsymbol{p}. \tag{3-18}$$

A la expresión anterior hay que añadirle la *bias,* que por simplicidad no se ha tenido en cuenta hasta ahora, con lo que se obtiene (3-19):

$$b^{new} = b^{old} + e. \tag{3-19}$$

Como se ha dicho con anterioridad, un perceptrón de una sola neurona es capaz de dividir el espacio de entrada a la red en dos regiones mediante una frontera lineal determinada por la expresión (3-20):

$${}_1\boldsymbol{w}^{T}\boldsymbol{p} + b = 0. \tag{3-20}$$

Por lo tanto, el perceptrón de una sola entrada solo es capaz de resolver satisfactoriamente problemas que

son linealmente separables. Desgraciadamente, muchos problemas no son linealmente separables, como los mostrados en la Figura 3-9, sin embargo, existen otro tipo de redes neuronales como las redes multicapa que son capaces de resolver problemas con una distribución arbitraria.

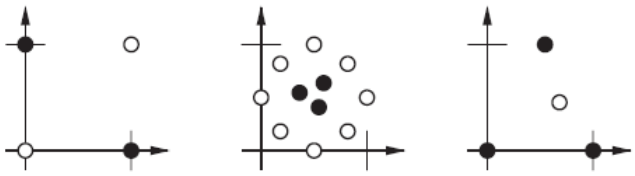


Figura 3-9. Problemas no linealmente separables.

3.3.4 Regla de aprendizaje en perceptron de multiples neuronas

Las reglas anteriores se pueden generalizar para redes neuronales con más de una capa haciendo uso de subíndices. Las fórmulas vistas con anterioridad se expresan entonces como:

$$\begin{cases} {}_i\boldsymbol{w}^{new} = {}_i\boldsymbol{w}^{old} + e\boldsymbol{p} \\ \boldsymbol{b}_i^{new} = \boldsymbol{b}_i^{old} + e \end{cases}. \tag{3-21}$$

Esto puede expresarse con notación matricial:

$$\begin{cases} \boldsymbol{W}^{new} = \boldsymbol{W}^{old} + e\boldsymbol{p} \\ \boldsymbol{b}^{new} = \boldsymbol{b}^{old} + e \end{cases}. \tag{3-22}$$

Al igual que en el perceptrón de una única neurona, a la hora de resolver un problema concreto tanto la matriz de peso como la *bias* se deben inicializar con valores aleatorios y de pequeño valor. Una vez hecho esto, se comienza a iterar hasta encontrar una frontera que clasifica correctamente los vectores de entrada.

3.3.5 Valoración de la regla del perceptrón

La regla de aprendizaje del perceptrón es muy potente, esto es así porque se puede demostrar que ésta siempre converge en unos pesos y valores de bias que satisfacen la clasificación deseada, asumiendo que estos pesos y bias existen.

4 REDES NEURONALES CONVOLUCIONALES

En el apartado 2 se ha desarrollado una introducción básica a las redes neuronales, presentando y explicando las redes neuronales más simples que existen, persiguiendo el objetivo de facilitar la comprensión de cómo la red neuronal convolucional, desarrollada en este proyecto, funciona.

A continuación, en este tema, se seguirá profundizando en las estructuras de las redes neuronales y, en concreto, se explicará el esquema general de una red neuronal convolucional y se hará un análisis en profundidad de cada una de las capas que la componen.

4.1 Esquema general de una red convolucional

Cuando se trabaja con imágenes, como en el caso de las redes neuronales convolucionales, se suele hacer con imágenes con resoluciones muy altas (por ejemplo, el standard actual es Full HD que consta de 1,980 x 1,080 píxeles). Esto conlleva un grave problema en las redes neuronales convencionales, que se han visto anteriormente. Éstas, al ser full-connected, tienen cada una de las neuronas que conforman la primera capa oculta conectada con todos y cada uno de los píxeles de la imagen de entrada a la vez. Si estas imágenes de entrada son de una resolución asumible, por ejemplo 28 x 28, el número de elementos que habrá que entrenar será lo suficientemente comedido como para que el sistema pueda funcionar correctamente, aunque todo el proceso será significativamente más lento. Sin embargo, si la resolución aumenta ligeramente (y no digamos ya a resoluciones actuales) los tiempos de entrenamiento y testeo se vuelven enormes.

A continuación, se expondrá la magnitud del problema anterior de manera cuantitativa. Si, por ejemplo, se usan imágenes de 96 x 96 se tendrían alrededor de 10^4 elementos de entrada y, si se asume que se quieren aprender 100 características, el número de elementos a entrenar sería del orden de 10^6. Esta enorme cantidad de elementos que deben ser entrenados hace que la duración del entrenamiento sea extraordinariamente larga, dificultando, en gran medida, el uso de las redes neuronales convencionales para aplicaciones con imágenes.

Una vez se es consciente de la magnitud del problema, se hace evidente la necesidad de usar algún tipo diferente de sistema, que permita sortear esta cuestión. En los siguientes puntos de este tema, se verá cómo se resuelve este problema.

Una red neuronal convolucional es un tipo de red multicapa que consta de diversas capas convolucionales y de pooling (submuestreo) alternadas, y al final tiene una serie de capas full-connected como una red perceptron multicapa. La entrada de una red capa convolucional suele ser, generalmente, una imagen m x m x r, donde m es tanto la altura como el ancho de la imagen y r es el número de canales (como se verá después, en este proyecto se trabaja con escala de grises por lo que $r = 1$). Las capas convolucionales tienen k filtros (o kernels) cuyas dimensiones son $n\ x\ n\ x\ q$, donde n y q son elegidas por el diseñador (generalmente q suele ser igual a r). Cada filtro genera mediante convolución un mapa de rasgos o características de tamaño $(m - n + 1)\ x\ (m - n + 1)\ x\ p$, siendo p el número de filtros que se desean usar. Después cada mapa es sub-muestreado en la capa de pooling con la operación "mean pooling" o "max pooling" sobre regiones contiguas de tamaño $p\ x\ p$ donde p puede tomar valores desde 2 para imágenes pequeñas hasta, comúnmente, no más de 5 para imágenes grandes. Antes o después del submuestreo, se aplica una función de activación sigmoidal más un sesgo para cada mapa de rasgos.

La composición de bloques anterior se puede ver en la Figura 4-1, en la que está representada una disposición genérica de las capas.

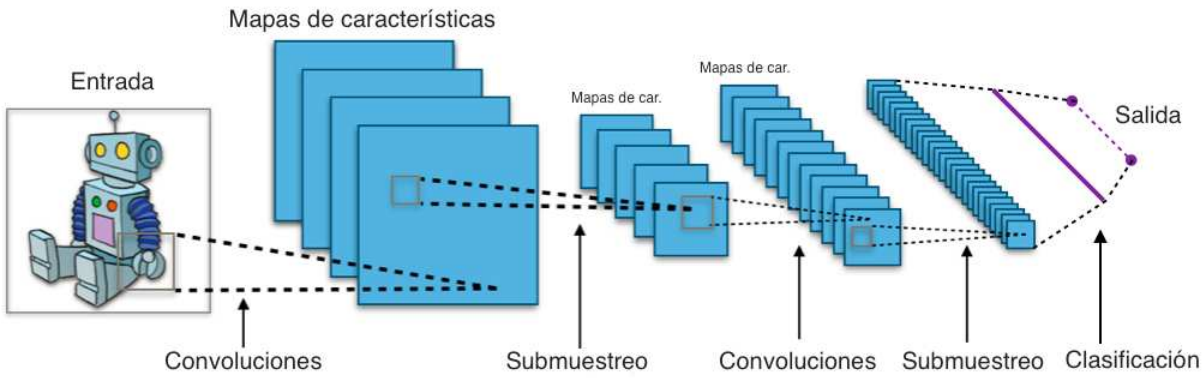


Figura 4-1. Esquema básico de una red neuronal convolucional.

Además, las redes convolucionales están diseñadas suponiendo que la entrada a la red es una imagen, lo cual permite codificar ciertas propiedades en la arquitectura, permitiendo ganar eficiencia y reducir la cantidad de parámetros usados en la red.

4.2 Capa de entrada

La primera capa del esquema de una red convolucional se trata de la entrada a la red. Como se ha visto anteriormente, en las redes convolucionales normalmente se tiene que la entrada a la red es una imagen. Además, como también se explicó al principio del documento, en este proyecto, se usa la base de datos MNIST por lo que estas imágenes de entrada tendrán dimensión de $28\ x\ 28$ y estarán en escala de grises.

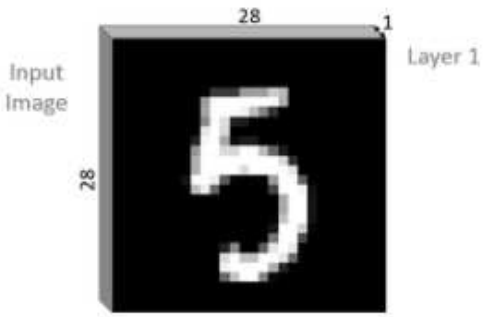


Figura 4-2. Entrada a la red convolucional.

En concreto, se le pasarán a la red 60,000 entradas (imágenes) para llevar a cabo el proceso de entrenamiento y, una vez entrenada, se comprobará su correcto funcionamiento con 10,000 imágenes de test.

4.3 Capa convolucional

Retomando el problema expuesto en el apartado 4.1 de este capítulo, que consistía en reducir la carga computacional del sistema, la capa convolucional toma un importante papel en este aspecto. Para ello, durante esta etapa de la red, se lleva a cabo una solución muy simple: se restringe el número de conexiones posibles entre las neuronas de la capa oculta y los elementos de la imagen de entrada. De esta manera, cada neurona oculta solo estará conectada con un pequeño subconjunto de elementos de la imagen total. Esta idea está inspirada en cómo funciona el sistema visual biológico, debido a que las neuronas que trabajan en el córtex visual poseen una serie de campos receptivos que responden solo ante unos estímulos localizados en una región o área específica.

Además, otra característica reseñable es que las imágenes naturales poseen la propiedad de ser "estacionarias", esto quiere decir que las características o rasgos que existen en alguna parte determinada de la imagen pueden ser los mismos que otros que se encuentren en otra zona totalmente distinta.

Por otro lado, la convolución es una operación de productos y sumas entre la imagen de entrada y un filtro (o kernel) que genera un mapa de características. La ventaja es que el mismo filtro (=neurona) sirve para extraer

el mismo rasgo en cualquier parte de la imagen, atendiendo al carácter estacionario de las imágenes que se ha comentado en el párrafo anterior. Esto permite reducir el número de conexiones y el número de parámetros a entrenar en comparación con una red multicapa full-connected.

En resumen, la capa convolucional permite tanto reducir el número de elementos que conforman la red como detectar una serie de características que serán útiles a la hora de analizar la imagen. Considerando lo anterior, se puede llegar a la conclusión de que la capa convolucional es de gran utilidad en el análisis de imágenes permitiendo reducir la complejidad del sistema y, al mismo tiempo, extraer rasgos útiles de éstas que ayuden con su análisis.

Como se ha explicado en el punto 4.1, la entrada de una capa convolucional es una imagen m x m x r, a esta imagen se le aplica un filtro de dimensiones n x n x q y, finalmente, como resultado se obtiene una matriz de características de tamaño $(\mathrm{m}-\mathrm{n}+1)\,x\,(m-n+1)\,x\,p$. El comportamiento anterior se consigue realizando el procedimiento explicado en los siguientes párrafos:

Cuando le llega la imagen de entrada a la red, se superponen el filtro y ésta y se calcula la convolución de dos dimensiones entre los respectivos elementos de la imagen y el kernel. Una vez se obtiene el resultado de la operación anterior, se almacena en una posición de la matriz de activación, como se pude observar en la Figura 4-3.

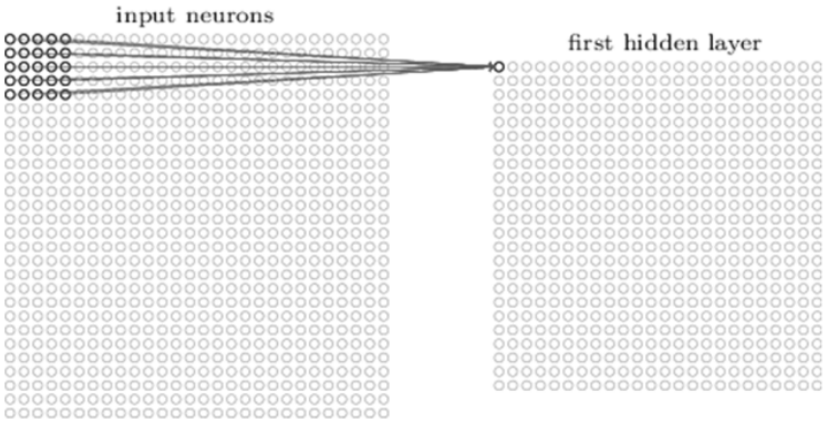


Figura 4-3. Operación básica de convolución.

A continuación, se desplaza o desliza el filtro una posición a la derecha sobre la imagen y se vuelve a calcular la convolución, almacenando el resultado en la siguiente posición de la matriz de activación. De esta manera, este proceso iterativo se repite a lo largo de toda la imagen desplazándose de izquierda a derecha y bajando una unidad al llegar a un borde. Una vez se ha recorrido toda la imagen se obtiene la matriz de activación completa que contiene las características que se buscan en la imagen para cada filtro. Este proceso se puede observar con mayor claridad en la Figura 4-4.

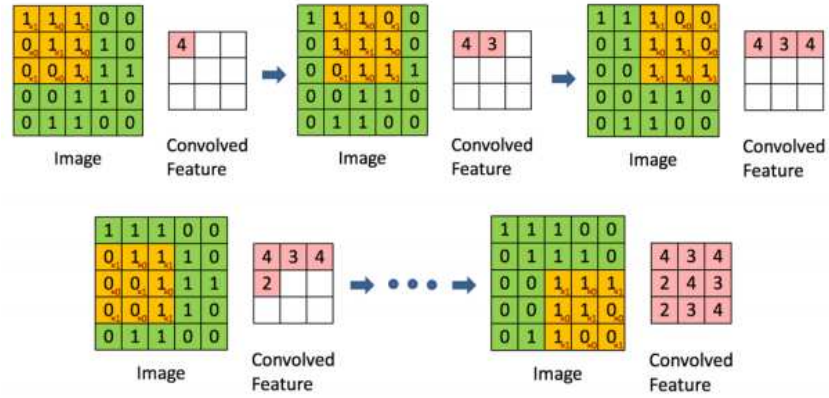


Figura 4-4. Obtención de la matriz de activación.

Si, por ejemplo, tenemos una imagen de $28\,x\,28\,x\,1$ y le aplicamos 6 filtros de $5\,x\,5\,x\,1$, la matriz de activación resultante será de $(28-5+1=)24\,x\,24\,x\,6$, como se muestra en la Figura 4-5.

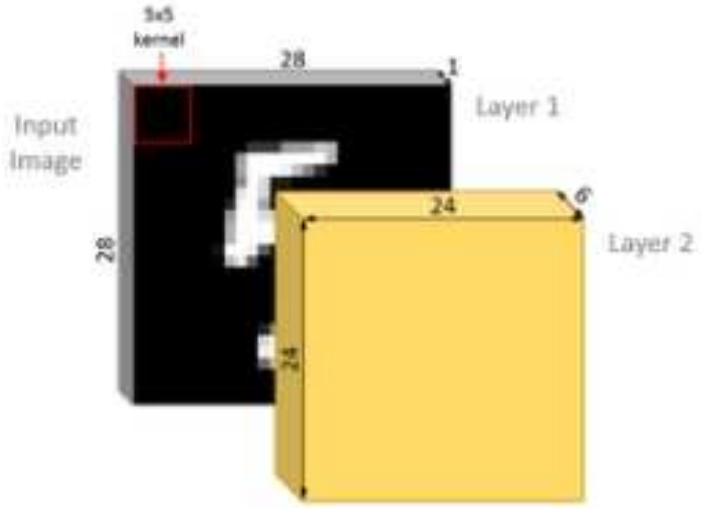


Figura 4-5. Resultado de aplicar la convolución sobre la imagen de entrada.

4.4 Capa de pooling

Siguiendo el esquema de la Figura 4-1, inmediatamente después de la capa de convolución se encuentra la capa de pooling. Después de haber obtenido las características en la capa de convolución, el siguiente paso en la lista es usarlas para la clasificación de las imágenes. En teoría, ya se podría usar cualquier tipo de clasificador para llevar a cabo esta tarea, sin embargo, este proceso aún puede ser muy costoso computacionalmente hablando. Por ejemplo, se considera lo siguiente: si la imagen de entrada es de $96\ x\ 96$ píxeles, a la salida de la capa convolucional se tendrá que el número de elementos será de $(96 - 8 + 1) * (96 - 8 + 1) = 7{,}921$ y, si se tienen, por ejemplo, 400 rasgos, el vector resultante es de $89^2\ x\ 400 = 3{,}168{,}400$ características. Una red con ese ingente número de características tendrá un comportamiento muy lento y, seguramente, se producirán errores de sobre-ajuste durante el aprendizaje.

Como se vio en el apartado 4.3, las imágenes tienen la propiedad de ser estacionarias y en la capa convolucional se ha creado un mapa de características. Así, el siguiente paso natural es ver en qué zona de la imagen son esos rasgos predominantes. Para realizar esta labor, es posible calcular la media o buscar el máximo valor de una característica a lo largo de una región de la imagen. La matriz resultante de la operación anterior resulta de unas dimensiones considerablemente menores que la matriz de características, obtenida en la capa de convolución.

Entonces, el objetivo de esta capa es el disminuir aún más la carga computacional del sistema y, al mismo tiempo, ayudar con la caracterización de la imagen obteniendo y localizando los rasgos predominantes en ella. Además, como se comentó en el párrafo, es importante destacar que existen dos tipos de pooling: mean-pooling o average-pooling y max-pooling. En la Figura 4-6, pueden verse los efectos de aplicar un tipo de pooling u otro.

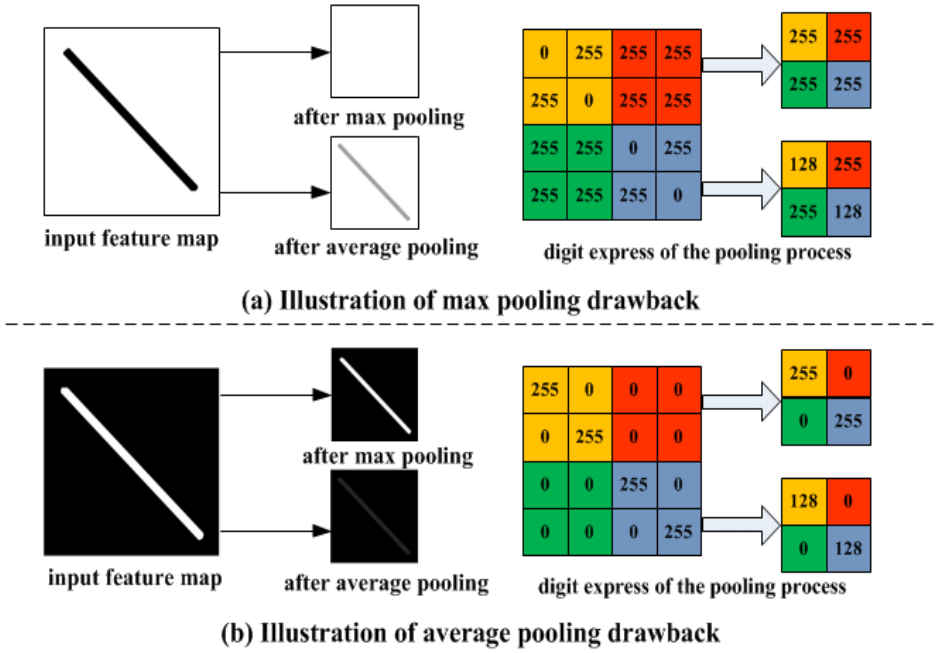


Figura 4-6. Resultado de aplicar diferentes tipos de pooling.

El resultado de aplicar la capa de pooling, en cuanto a disminución de dimensiones como se ha discutido en párrafos anteriores, se puede apreciar en la Figura 4-7.

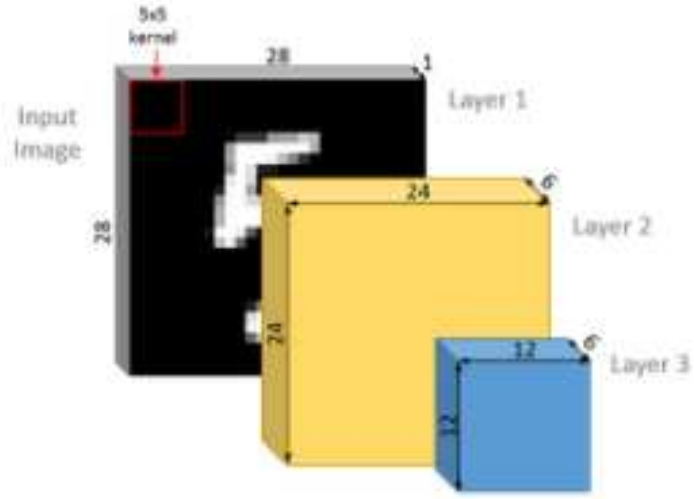


Figura 4-7. Salida de la capa de pooling.

El funcionamiento de la capa de pooling es el siguiente: en primer lugar, desde la esquina superior izquierda de la matriz de activación se selecciona una submatriz de tamaño $p\ x\ p$. A continuación, dependiendo del tipo de pooling se hará lo siguiente: si se trata de mean-pooling o average-pooling se cogen todos los elementos de la submatriz, se calcula su media y el resultado se guarda en la primera posición de la matriz de salida, si es max-pooling se busca el elemento de mayor valor que se encuentre en esa submatriz y se guarda en la primera posición de la matriz de salida. A continuación, se selecciona la siguiente submatriz que está a $p + 1$ posiciones a la derecha y se realiza la misma operación. Una vez llegado al borde derecho de la imagen, se vuelve a la izquierda de la imagen, se da un salto hacia abajo de $p + 1$ elementos y se vuelve a recorrer la imagen de izquierda a derecha. Entonces, si la matriz de activación creada por la capa de convolución es de tamaño $n\ x\ n\ x\ q$, la matriz de salida de la capa de pooling será de $\frac{n}{p}\ x\ \frac{n}{p}\ x\ q$. El procedimiento anterior puede verse en la Figura 4-8.

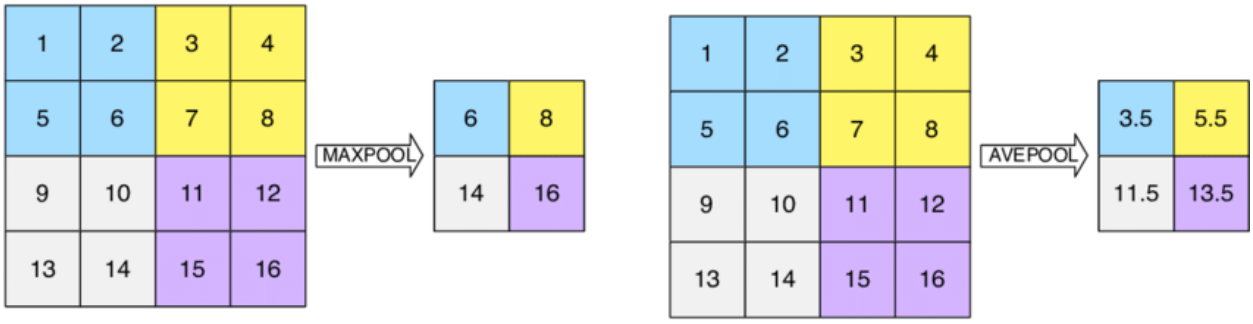


Figura 4-8. Operación de max-pooling y de average-pooling.

4.5 Capa full-connected

Ésta se trata de la última capa del esquema de las redes neuronales convolucionales y se trata de un clasificador que determina a que clase pertenece la imagen de entrada, es decir, su trabajo en este proyecto es indicar que número 'cree' la red que se le ha proporcionado a la entrada.

La capa totalmente conectada se encuentra justo a continuación de la última capa de pooling. Está compuesta por un número de neuronas igual al número de clases, en el caso de este proyecto son 10 neuronas (son 10 clases: los números del 0 al 9). El esquema que sigue esta capa es el explicado en el apartado 2, cuando se habló de las redes neuronales totalmente conectadas. Es decir, cada una de estas neuronas está a su vez conectada con todos y cada uno de los elementos de las matrices de la capa inmediatamente anterior.

A la salida de esta capa se encontrará un vector de 10 componentes. Cada una de estas componentes representa la probabilidad que tiene la imagen de entrada de pertenecer a una determinada clase, lo que es lo mismo, la probabilidad que tiene de ser un número u otro. Por último, la red determinará cuál de estos elementos del vector es mayor y lo indica.

4.6 Red implementada en R

El objetivo de este apartado es hacer un pequeño inciso en cómo se ha implementado la red convolucional en R. Durante todo el tema se ha hablado del proceso que sigue la red cuando a la entrada tiene tan solo una imagen, sin embargo, a la entrada de la red neuronal del proyecto habrá muchas más. En concreto, como se explicará en capítulos posteriores, la red neuronal irá recibiendo de 256 en 256 imágenes. Lo anterior es posible usando matrices: a la entrada se tendrá una matriz de 28 x 28 x 256, con 20 filtros de dimensión 9 x 9, la matriz de activación resulta de 20 x 20 x 20 x 256 y así sucesivamente con el resto de capas.

Por otro lado, la red de este proyecto consta específicamente de las siguientes capas: la capa de entrada, una capa convolucional, una capa de pooling y una capa full-connected con regresión softmax (que es un tipo clasificador que se verá en el tema siguiente). Con estas capas en concreto, como se explicará en el tema 6, se conseguirá un resultado muy bueno, cumpliendo los objetivos marcados. En la Figura 4-9, se puede apreciar el esquema de la red convolucional implementada:

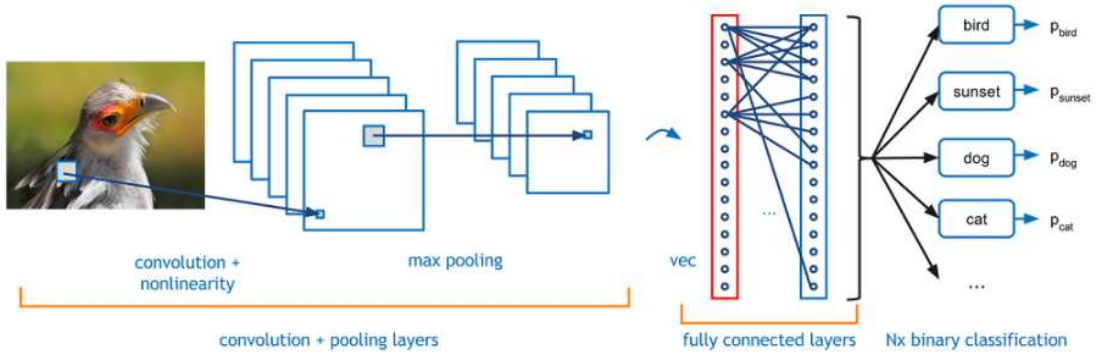


Figura 4-9. Esquema de la red convolucional implementada.

5 REGLAS DE APRENDIZAJE DEL PROYECTO

En el capítulo 3, se hizo una introducción a los algoritmos de aprendizaje de las redes neuronales. En ese tema, se sentaron las bases de cuál es el objetivo que se persigue al entrenar, que efecto tiene el entrenamiento en los elementos de la red y cómo se hace. Para ello, se usó un caso trivial en una red neuronal muy simple, como es el perceptrón. Antes de seguir con el desarrollo de este punto, es conveniente resaltar de nuevo los conceptos más relevantes que se aprendieron.

En primer lugar, el objetivo que se persigue con el entrenamiento de una red neuronal es el de conseguir que la red sea capaz de resolver de manera satisfactoria un problema determinado que antes no sabía resolver. Para logar que la red sea capaz de aprender a solucionar estos nuevos problemas debe ser entrenada modificando los parámetros de la matriz de pesos y de la bias. Estas modificaciones que se producen en los elementos de la red, que no son más que una serie de operaciones matemáticas que se aplican a los pesos y la bias, vienen impuestas por una serie de reglas, que dependen del método de aprendizaje usado.

En este tema, se explicará el algoritmo de aprendizaje que se ha usado para implementar el entrenamiento de la red. En primer lugar, se hará una explicación más general de cómo funcionan los dos algoritmos que se han usado y, finalmente, se detallará como se han implementado ambas reglas en el programa.

Cabe destacar, que el tipo de entrenamiento se trata de aprendizaje supervisado, es decir, se tiene un set de entrenamiento, con parejas de imágenes de entrada y objetivo, con lo que se conoce a priori la salida que debe tener la red ante una imagen de entrada determinada.

5.1 Regresiones

5.1.1 Formulación del problema

Para poder comprender correctamente los algoritmos que se han usado, es conveniente realizar un análisis previo de las regresiones y la optimización, lo cual será el eje central del aprendizaje de la red neuronal de este proyecto. El objetivo que se busca con esta introducción es familiarizarse con una serie de conceptos muy usados en este campo y que será de vital importancia conocer para posteriormente entender los métodos de aprendizaje de este proyecto.

El objetivo que se persigue en cualquier tipo de regresión es ser capaz de predecir la salida, $\boldsymbol{y}$, del sistema de manera correcta simplemente conociendo el vector, $\boldsymbol{x}$, de entrada. Por poner un ejemplo, hacer una predicción sobre el precio de una casa tan solo conociendo sus características: número de habitaciones, número de baños, etc. En este caso, la *y* que se pretende adivinar es el precio de la casa y la $\boldsymbol{x}$ es un vector cuyos elementos representan las características de la casa en cuestión.

Ahora, suponiendo que se tiene un gran número de ejemplos de casas de las cuales se conocen sus respectivas características y precios, donde $\boldsymbol{x}^{(i)}$ representa el vector de características de la i-ésima casa e $y^{(i)}$ representa el valor de la i-ésima casa, el objetivo es encontrar una función $y = h(\boldsymbol{x})$ que cumpla que $y^{(i)} \cong h\left(\boldsymbol{x}^{(i)}\right)$ para cada ejemplo de entrenamiento. Si se encuentra la función que cumpla esto y se usan suficientes ejemplos de casas para entrenar, se puede esperar que la función $h(\boldsymbol{x})$ resultante sea un buen predictor del precio de una casa, incluso cuando sus características no se correspondan con un ejemplo visto con anterioridad.

Para encontrar una función $h(\boldsymbol{x})$ donde se cumpla que $y^{(i)} \cong h\left(\boldsymbol{x}^{(i)}\right)$ primero hay que decidir cómo representar la función $h(\boldsymbol{x})$. Como ejemplo para ilustrar ese procedimiento se usará la regresión lineal, aunque podría usarse cualquier otra, la regresión logística, por ejemplo, ya que lo que cambia son las expresiones matemáticas pero la idea subyacente es la misma. Entonces, para la regresión lineal la función es la de la expresión (5-1):

$$h_\theta(\boldsymbol{x}) = \sum_j \theta_j \boldsymbol{x}_j = \theta^T \boldsymbol{x}. \qquad (5\text{-}1)$$

Donde $h_\theta(\boldsymbol{x})$ representa una gran familia de funciones parametrizadas por la elección de θ. Entonces, con esta representación de $h(\boldsymbol{x})$, la tarea se centra en encontrar un θ que haga que $h(\boldsymbol{x}^{(i)})$ sea lo más parecido posible a $y^{(i)}$. En particular, se buscará la elección de θ que minimice la siguiente ecuación:

$$J(\theta) = \frac{1}{2}\sum_i \left(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2}\sum_i \left(\theta^T \boldsymbol{x}^{(i)} - y^{(i)}\right)^2. \tag{5-2}$$

La ecuación (5-2) es llamada función de coste, es diferente para cada tipo de regresión e indica cuanto error se está cometiendo a la hora de predecir el valor de $y^{(i)}$ para un valor determinado de θ.

5.1.2 Minimización de la función

Existen muchos tipos de algoritmos que minimizan funciones como ésta, como por ejemplo el del descenso del gradiente que se verá más adelante. Por ahora, es importante reseñar el hecho de que la mayoría de los algoritmos más usados para minimizar las funciones de coste requieren para funcionar dos elementos de $J(\theta)$: por un lado, se necesitará la función de coste en sí, $J(\theta)$, y, por otro lado, su respectivo gradiente, $\nabla_\theta J(\theta)$. Una vez se le pasen estos datos al algoritmo de optimización, éste será el encargado de modificar el valor de θ.

El gradiente se calcula como en la ecuación (5-3):

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}, \qquad \frac{\partial J(\theta)}{\partial \theta_j} = \sum_i x_j^{(i)} \left(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)}\right). \tag{5-3}$$

5.1.3 Regresión Softmax

En el punto anterior se ha explicado los objetivos que se persiguen al aplicar la regresión y para ello se ha puesto de ejemplo la regresión lineal. Sin embargo, existen otros tipos de regresiones. En este apartado se explicará la regresión Softmax, la cual es útil en la implementación de la red neuronal convolucional de este proyecto.

En primer lugar, se tiene la regresión logística que sirve para predecir valores en problemas que se necesita un resultado binario, es decir, 0 o 1. Sin embargo, en el problema de clasificación de los números de la MNIST este no es un resultado válido y es necesario un clasificador que pueda manejar más clases. Este clasificador buscado se trata del Softmax.

En segundo lugar, la regresión Softmax es una generalización de la regresión logística que permite clasificar un problema de *K* clases distintas. El resultado que proporciona el clasificador Softmax es la probabilidad que tiene la entrada actual al sistema de pertenecer a cada una de las clases, en el caso de la red implementada se tratan de 10 clases.

Dado una entrada $\boldsymbol{x}$, se busca que $h_\theta(\boldsymbol{x})$ estime la probabilidad de que $P(y = k|x)$ para cada valor de $k = 1, \dots, K$. A la salida habrá un vector de *K* elementos donde cada componente se corresponde con la probabilidad antes mencionada, o lo que es lo mismo:

$$h_\theta(\boldsymbol{x}) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp\left(\theta^{(j)T}\boldsymbol{x}\right)} \begin{bmatrix} \exp\left(\theta^{(1)T}\boldsymbol{x}\right) \\ \exp\left(\theta^{(2)T}\boldsymbol{x}\right) \\ \vdots \\ \exp\left(\theta^{(K)T}\boldsymbol{x}\right) \end{bmatrix} \quad (5\text{-}4)$$

El término $\frac{1}{\sum_{j=1}^{K} \exp\left(\theta^{(j)T}\boldsymbol{x}\right)}$ de la ecuación (5-4) sirve para normalizar el resultado, por lo que la suma de todos los elementos del vector debe ser igual a 1.

Por último, se presentarán la función de coste y el gradiente que se usan para la regresión softmax y que son necesarias calcular para su posterior uso en un algoritmo de optimización. Éstas vienen recogidas en las ecuaciones (5-5):

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(l)} = k\} \log\left(\frac{e^{(\theta)T\,\boldsymbol{x}^{(i)}}}{\sum_{j=1}^{K} e^{(\theta)T\,\boldsymbol{x}^{(i)}}}\right)\right], \quad (5\text{-}5)$$

$$\nabla_{\theta^{(k)}} J(\theta) = -\sum_{i=1}^{m}\left[\boldsymbol{x}^{(i)}\left(1\{y^{(i)} = k\} - P(y^{(i)} = k|\boldsymbol{x}^{(i)};\theta)\right)\right].$$

Donde, $1\{\cdot\}$ es una "función indicadora", de manera que $1\{a\ true\ statement\} = 1$ y $1\{a\ false\ statement\} = 0$. Por ejemplo $1\{2 + 2 = 4\} = 1$.

5.2 Algoritmo de retropropagación

Para explicar este apartado, se supondrá que la red neuronal se trata de una red totalmente conectada y que ésta, será entrenada con el método del aprendizaje batch gradient descent (descenso del gradiente). Se supone, además, que para entrenar esta red genérica se usa un conjunto de entrenamiento de la forma $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, con un número *m* de ejemplos. Sin entrar más en detalle, dado un determinado ejemplo de entrenamiento, se define la función de coste con respecto a ese ejemplo como la expresión dada por (5-6):

$$J(\boldsymbol{W}, \boldsymbol{b}; x, y) = \frac{1}{2}\left\|h_{\boldsymbol{W},\boldsymbol{b}}(x) - y\right\|^2. \quad (5\text{-}6)$$

La ecuación (5-6) es una función de coste de error cuadrático. Si, por contrario, se tiene un conjunto de *m* ejemplos, se define la función de coste total como:

$$\begin{aligned} J(\boldsymbol{W}, \boldsymbol{b}) &= \left[\frac{1}{m}\sum_{i=1}^{m} J\left(\boldsymbol{W}, \boldsymbol{b}; x^{(i)}, y^{(i)}\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(w_{ji}^{(l)}\right)^2 = \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left\|h_{\boldsymbol{W},\boldsymbol{b}}\left(x^{(i)}\right) - y^{(i)}\right\|^2\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(w_{ji}^{(l)}\right)^2. \end{aligned} \quad (5\text{-}7)$$

El primer término en la definición del coste es la media aritmética de los términos de error cuadrático medio. El segundo elemento es un término de regularización, también llamado disminución de pesos, que tiende a decrementar la magnitud de los pesos de la red, lo cual ayuda a prevenir el sobreajuste en el entrenamiento. Esencialmente, la disminución de pesos, o en inglés weigth decay, es una modificación de la regularización Bayesiana. El término de disminución de pesos λ controla la importancia relativa entre los dos términos de la ecuación (5-7).

Esta función de coste total es usada, a menudo, en problemas de clasificación y en problemas de regresión. Para los problemas de clasificación, los valores que podrá tomar *y* serán 0 u 1, si se usa la función de

activación sigmoidea (-1 y 1 si se usa la función de activación tanh). Para los problemas de regresión, primero habrá que escalar la salida para asegurarse que esté comprendida dentro del rango [0,1] ([-1,1] si se usa tanh).

El objetivo que se busca es el de minimizar $J(\boldsymbol{W}, \boldsymbol{b})$ en función de $\boldsymbol{W}$ y de $\boldsymbol{b}$. Para entrenar la red neuronal, se inicializará cada parámetro $\boldsymbol{W}_{ij}^{(l)}$ y cada $\boldsymbol{b}_{ij}^{(l)}$ con un valor muy pequeño (cercano a cero) y aleatorio. Una posibilidad es la de inicializar estos parámetros siguiendo una distribución normal de media nula y varianza ϵ^2, donde ϵ es un valor del orden de 0.01, y, entonces, aplicar un algoritmo de optimización, que es el que trata de minimizar el coste. Por ejemplo, un posible algoritmo que se puede aplicar es el del descenso del gradiente. Como la función de coste $J(\boldsymbol{W}, \boldsymbol{b})$ es una función no convexa, el método del gradient descent es susceptible de alcanzar un mínimo de carácter local, en vez de global, por lo que el resultado obtenido no sería óptimo. Sin embargo, en la práctica este método de aprendizaje suele tener un desempeño bastante satisfactorio, encontrando la solución óptima.

Es de vital importancia destacar que la inicialización de los elementos de la red no es un asunto baladí: es primordial realizar una inicialización aleatoria de estos, y no igualarlos simplemente a cero. La importancia de este aspecto reside en que, si todos los valores iniciales de los pesos son exactamente el mismo entonces, todas las neuronas de las capas ocultas acabarán aprendiendo la misma función de la entrada. Es decir, $\boldsymbol{W}_{ij}$ será el mismo para todos los valores de *i*, lo que implica que $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \cdots$, para cualquier entrada de *x*. La inicialización aleatoria sirve al propósito de romper la simetría de la red.

En cada iteración del descenso del gradiente se actualizan los parámetros $\boldsymbol{W}, \boldsymbol{b}$ como sigue:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}),$$
$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}). \qquad (5\text{-}8)$$

Donde α es el ratio de aprendizaje. El paso clave es calcular de manera correcta las derivadas parciales de las ecuaciones de (5-3). A continuación, se describirá el algoritmo de retropropagación, que proporciona una manera muy eficiente de calcular estas derivadas parciales.

En primer lugar, se verá como el método de retropropagación ayuda en el cálculo de $\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y)$ y de $\frac{\partial}{\partial \boldsymbol{b}_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y)$, es decir, las derivadas parciales de la función de coste en el caso de que solo se tenga un único ejemplo. Una vez calculadas las derivadas parciales anteriores, se verá que las derivadas parciales de la función de coste total se calculan como en las ecuaciones de (5-9):

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \boldsymbol{W}_{ij}^{(l)}} J\left(\boldsymbol{W}, \boldsymbol{b}; x^{(i)}, y^{(i)}\right) \right] + \lambda W_{ij}^{(l)},$$
$$\frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J\left(\boldsymbol{W}, \boldsymbol{b}; x^{(i)}, y^{(i)}\right). \qquad (5\text{-}9)$$

Las dos líneas anteriores difieren debido a que el descenso del peso es aplicado a $\boldsymbol{W}$ y no a $\boldsymbol{b}$.

La idea que subyace bajo el concepto del algoritmo de retropropagación es la siguiente: dado un único ejemplo de entrenamiento *(x, y)*, en primer lugar, se llevará a cabo un "feedfordward pass" o paso hacia delante, donde se calcularán todas las activaciones a lo largo de la red neuronal, incluyendo el valor de salida $h_{\boldsymbol{W},\boldsymbol{b}}(x)$. Entonces, para cada nodo *i* de la capa *l*-ésima, habrá que calcular un término de error, representado por $\delta_i^{(l)}$, que regula cómo de responsable es ese nodo del posible error que haya a la salida. Para un nodo de salida de la red se puede calcular directamente este término como la diferencia entre el valor de activación actual de la red y el valor objetivo correcto, con lo que $\delta_i^{(n_l)}$ queda definido. Para las neuronas que se encuentran en las capas ocultas, se calculará $\delta_i^{(l)}$ como una media ponderada de los términos de error de los nodos que tengan como entrada $a_i^{(l)}$. Con más detalle, el algoritmo de retropropagación es el siguiente:

1. Realizar el paso hacia delante o feedforward pass, visto en el capítulo 2 (concretamente la ecuación (2-3)), calculando las activaciones de las capas $L_2, L_3, \dots, L_{n_l}$.
2. Para cada i-ésima neurona de la capa de salida, calcular lo siguiente:

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{\boldsymbol{W},\boldsymbol{b}}(x) \right\|^2 = -\left(y_i - a_i^{(n_l)} \right) \cdot f'\left(z_i^{(n_l)} \right). \tag{5-10}$$

3. Para $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$, calcular para cada nodo *i* de la capa *l:*

$$\delta_i^{(l)} = \left(\sum_{j=1}^{S_{l+1}} w_{ji}^{(l)} \delta_j^{(l+1)} \right) f'\left(z_i^{(n_l)} \right). \tag{5-11}$$

4. Por último, se calculan las derivadas parciales deseadas:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = a_j^{(l)} \delta_i^{(l+1)},$$
$$\frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \delta_i^{(l+1)}. \tag{5-12}$$

Finalmente, puede reescribirse el algoritmo completo usando notación vectorial y matricial. Así, se incrementará notablemente el desempeño de la red en términos de tiempo de cálculo, disminuyendo éste de forma considerable. Persiguiendo este objetivo, se usará el operador ".*", denominado producto de Hadamard, que consiste en realizar las operaciones elemento a elemento. El algoritmo puede ahora ser reescrito como:

1. Realizar el paso hacia delante o feedforward pass, visto en el capítulo 2 (en este caso, la ecuación (2-5)), calculando las activaciones de las capas $L_2, L_3, \dots, L_{n_l}$.
2. Para la capa de salida, calcular:

$$\delta^{(n_l)} = -\left(y - a^{(n_l)} \right) .* f'\left(z^{(n_l)} \right). \tag{5-13}$$

3. Para $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$, calcular*:*

$$\delta^{(l)} = \left(\left(\boldsymbol{W}^{(l)} \right)^T \delta^{(l+1)} \right) .* \ f'\left(z^{(l)} \right). \tag{5-14}$$

4. Por último, se calculan las derivadas parciales deseadas:

$$\nabla \boldsymbol{W}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \delta^{(l+1)} \left(a^{(l)} \right)^T,$$
$$\nabla \mathbf{b}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \delta^{(l+1)}. \tag{5-15}$$

En el siguiente pseudocódigo, $\Delta \boldsymbol{W}^{(l)}$ se trata de una matriz de dimensiones iguales que $\boldsymbol{W}^{(l)}$, análogamente, $\Delta \boldsymbol{b}^{(l)}$ es un vector cuyo tamaño coincide con $\boldsymbol{b}^{(l)}$. Por último, el algoritmo completo para calcular el gradient descent expresado en pseudocódigo es el siguiente:

1. Se establece $\Delta \boldsymbol{W}^{(l)} := 0$ y $\Delta \boldsymbol{b}^{(l)} := 0$, para todo *l*.
2. Desde $i = 1$ hasta *m*,
   2.1. Usar la retropropagación para calcular $\nabla \boldsymbol{W}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; x, y)$ y $\nabla \mathbf{b}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; x, y)$.
   2.2. Establecer $\Delta \boldsymbol{W}^{(l)} := \Delta \boldsymbol{W}^{(l)} + \nabla \boldsymbol{W}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; x, y)$.
   2.3. Establecer $\Delta \boldsymbol{b}^{(l)} := \Delta \boldsymbol{b}^{(l)} + \nabla \mathbf{b}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; x, y)$.

3. Se actualizan los parámetros:

$$\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \alpha\left[\left(\frac{1}{m}\Delta\boldsymbol{W}^{(l)}\right) + \lambda\boldsymbol{W}^{(l)}\right],$$
$$\boldsymbol{b}^{(l)} = \boldsymbol{b}^{(l)} - \alpha\left[\frac{1}{m}\Delta\boldsymbol{b}^{(l)}\right]. \qquad (5\text{-}16)$$

Para el caso particular de la red neuronal convolucional de este proyecto, visto en el capítulo 4, los errores y gradientes se calculan como:

Sea $\boldsymbol{\delta}^{(l+1)}$ el error de la capa *(l+1)* con una función de coste $J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$. Si entonces la capa l-ésima se encuentra totalmente conectada con la capa *(l+1)*, entonces el error de la capa l-ésima es calculado mediante la expresión (5-17):

$$\delta^{(l)} = \left(\left(\boldsymbol{W}^{(l)}\right)^T \delta^{(l+1)}\right).* \ f'\left(z^{(l)}\right). \qquad (5\text{-}17)$$

Y los gradientes son:

$$\nabla\boldsymbol{W}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \delta^{(l+1)}\left(a^{(l)}\right)^T,$$
$$\nabla\mathbf{b}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \delta^{(l+1)}. \qquad (5\text{-}18)$$

Sin embargo, si la l-ésima capa es una capa convolucional o de pooling entonces el error es propagado a través de ellas como:

$$\delta_k^{(l)} = upsample\left(\left(\boldsymbol{W}_k^{(l)}\right)^T \delta_k^{l+1}\right).* f'\left(z_k^{(l)}\right). \qquad (5\text{-}19)$$

En la ecuación anterior, *k* es un índice que hace referencia al filtro correspondiente y $f'\left(z_k^{(l)}\right)$ es la derivada de la función de activación. La operación upsample debe propagar el error a través de la capa de pooling calculando el error para cada unidad de dicha capa, es decir, si se usa la operación mean pooling, entonces la operación upsample simplemente distribuye el error de forma uniforme entre aquellas unidades que sirvieron como entrada. Si, por el contrario, se usó max pooling, entonces aquella unidad de entrada que tenía el valor máximo es la que recibe todo el error.

Finalmente, para calcular el gradiente de cada uno de los filtros se realiza de nuevo una convolución y se vuelve a rotar la matriz de error, quedando al final:

$$\nabla\boldsymbol{W}_k^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \sum_{i=1}^{m}\left(a_i^{(l)}\right) * rot90\left(\delta_k^{l+1}, 2\right),$$
$$\nabla\mathbf{b}_{\mathrm{k}}^{(l)} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \sum_{a,b}\left(\delta_k^{(l+1)}\right)_{a,b}. \qquad (5\text{-}20)$$

5.3 Optimización: Stochastic Gradient Descent

Como se introdujo en el apartado 5.1.2 los algoritmos de optimización se centran en minimizar la función de coste. En el punto 5.2, para explicar el algoritmo de retropropagación, se utilizó el algoritmo del batch gradient descent. Este método de minimización usa el set completo de entrenamiento en cada iteración para actualizar los parámetros y tiende a converger de manera eficiente en un mínimo absoluto. Sin embargo, este tipo de

algoritmos son buenos para sets de entrenamientos no muy grandes debido a que, en cuanto éste aumenta de tamaño, calcular el coste y el gradiente del set completo se vuelve una tarea muy lenta e, incluso, imposible para un único sistema si el set de datos es demasiado grande para caber en la memoria. Otro de los problemas principales es que este tipo de algoritmos no permiten incorporar nuevos datos de entrenamiento online, es decir mientras el algoritmo está en funcionamiento.

Para solucionar ambos problemas surge el Stochastic Gradient Descent o SGD que es capaz de seguir la dirección negativa del gradiente tras haber visto tan solo un o unos pocos ejemplos. El uso del SGD en las redes neuronales viene motivado por el alto coste computacional de la retropropagación a través del set de entrenamiento completo. Stochastic Gradient Descent puede sobrellevar este problema y aun así obtener una rápida convergencia.

El algoritmo por descenso del gradiente estándar actualiza los parámetros θ de la red de la función de coste $J(\theta)$ como en la ecuación (5-21):

$$\theta = \theta - \alpha \nabla_\theta E[J(\theta)]. \qquad (5\text{-}21)$$

Donde la esperanza de la ecuación anterior es determinada mediante el cálculo del coste y del gradiente a lo largo del set de datos completo. Sin embargo, el Stochastic Descent Gradient, en vez de usar la esperanza que como se acaba de comentar usa todo el set de entrenamiento completo, es capaz de calcular el gradiente de los parámetros usando tan solo un o unos pocos ejemplos de entrenamiento en cada iteración. La nueva regla de actualización es:

$$\theta = \theta - \alpha \nabla_\theta J\left(\theta; \boldsymbol{x}^{(i)}, y^{(i)}\right), \qquad (5\text{-}22)$$

para el caso de un par de entrenamiento $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$.

Generalmente las actualizaciones de los parámetros con el SGD son calculadas con unos pocos ejemplos (o minibatch) y no con uno solo. La razón de lo anterior es doble: en primer lugar, reduce la varianza en la actualización de los parámetros y. gracias a esto, es capaz de alcanzar una convergencia más estable., en segundo lugar, el usar un minibatch permite aprovecharse de una serie de ventajas que implican la notación matricial y el álgebra lineal, además de la posibilidad de usar una serie de funciones en R muy optimizadas para el cálculo con matrices, en el cómputo del coste y del gradiente. Un tamaño típico del minibatch es de 256 imágenes, aunque el valor óptimo del minibatch puede variar dependiendo del problema en concreto que se esté tratando.

En el Stochastic Gradient Descent el valor de α es típicamente mucho más pequeño que el correspondiente valor del ratio de aprendizaje en el Batch Gradient Descent porque hay mucha más varianza en la actualización. Elegir el correcto ratio de aprendizaje y su programación (el valor del ratio puede ir variando a lo largo del entrenamiento) puede ser muy dificultoso. Una forma de elegir el ratio de aprendizaje correcto que funciona bien en la práctica es usar un valor constante y lo suficientemente pequeño para conseguir una convergencia estable en la primera o dos primeras épocas y entonces disminuir su valor a la mita. Una época es el paso completo del set de entrenamiento por la red, en este proyecto se harán 3 épocas para entrenarla.

Un último aspecto importante en el SGD es el orden en el que son presentados los datos al algoritmo. Si los datos son presentados en un orden significativo para la red, puede provocar que se obtenga una mala convergencia. Generalmente, un buen método para evitar que pase lo anterior es reordenar los datos de manera aleatoria antes de cada época de entrenamiento.

5.3.1 Momentum

Si el objetivo tiene la forma de un barranco largo y estrecho que conduce al mínimo con paredes empinadas a los lados, el algoritmo estándar del Stochastic Gradient Descent oscilará a lo largo de los lados del barranco. Esto es debido a que el gradiente negativo apuntará hacia abajo respecto a cada una de las paredes del barranco, entonces en cada iteración irá de una pared a otra, en lugar de ir a lo largo del barranco hasta llegar al punto óptimo. Los objetivos de las arquitecturas de redes neuronales profundas suelen tener esta forma cera del punto óptimo y el SGD estándar tenderá a una convergencia muy lenta, particularmente tras los primeros pasos del entrenamiento.

El momentum es un método que empuja a llegar al punto objetivo de una manera más rápida a través de este

baranco. La regla de actualización del momentum viene dada por la ecuación (5-23),

$$\boldsymbol{v} = \gamma \boldsymbol{v} + \alpha \nabla_{\theta} J\left(\theta; \boldsymbol{x}^{(i)}, y^{(i)}\right),$$
$$\theta = \theta - \boldsymbol{v}. \qquad \textit{(5-23)}$$

En la ecuación (5-19), $\boldsymbol{v}$ es el vector de velocidad actual que posee las mismas dimensiones que el vector de parámetros θ. El ratio de aprendizaje α es el mismo que en el apartado anterior, aunque al usar el momentum el valor de α deba ser algo menor ya que la magnitud del gradiente será mayor. Por último, $\gamma \in (0, 1]$ determina en cuantas iteraciones los gradientes anteriores será incorporados. Usualmente, λ es establecido a 0.5 hasta que el aprendizaje se estabiliza, entonces su valor es incrementado hasta un mínimo de 0.9.

5.4 Aprendizaje implementado en R

Durante el transcurso de este tema se han explicado todos los aspectos implicados en el aprendizaje de la red neuronal del proyecto y en este punto se hace una pequeña aclaración de cómo todo lo anterior ha sido implementado en R.

Se comenzó explicando los tipos de regresiones y el objetivo de usar éstas, el cual no es más que ejercer de clasificador de las clases de la red, es decir, deducir de que número se trata la imagen de la entrada. Como se explicó en el punto 5.1, la regresión Softmax es una generalización de la regresión logística y está especializada para el caso de tener varias clases, como se da en este proyecto. En el programa, será la encargada de calcular la probabilidad que tiene la imagen de entrada de pertenecer a cada clase y calculará la función de coste, necesaria para el cálculo posterior del gradiente.

A continuación, se trató el algoritmo de retropropagación que es el encargado de calcular el gradiente de la iteración actual y de propagar hacia atrás los errores a lo largo de la red convolucional.

Por último, el algoritmo de optimización Stochastic Gradient Descent, aplicándole el momentum, se encarga de actualizar los pesos y las bias de la red convolucional, modificando en cada iteración el vector θ del cual dependen todos los elementos de la red.

La red será entrenada con 60,000 imágenes de la base de datos MNIST, mediante el método anteriormente descrito. Durante el transcurso del entrenamiento los algoritmos irán variando el vector θ, que contiene todos los pesos y las bias de la red, hasta obtener el valor final con el vector optimizado. Este proceso tarda alrededor de 1 hora en completarse.

6 ANÁLISIS DE RESULTADOS Y CONCLUSIONES

En este punto se presentarán los resultados obtenidos en el proyecto y se hará un análisis de las conclusiones del trabajo. En primer lugar, se analizará los resultados tras aplicarle a la red un test de muchas imágenes y, en segundo lugar, se verán algunos ejemplos concretos de la respuesta de la red ante la entrada de diferentes imágenes.

6.1 Test del Sistema

Tras entrenar la red con los métodos descritos en el punto anterior y obtener el vector θ optimizado, el siguiente paso a comprobar es ver si el entrenamiento ha sido fructífero y corroborar que la red convolucional sea capaz de reconocer los números correctamente. Para realizar la prueba de funcionamiento del sistema se usa la parte correspondiente al test de la base de datos de números escritos a mano de la MNIST.

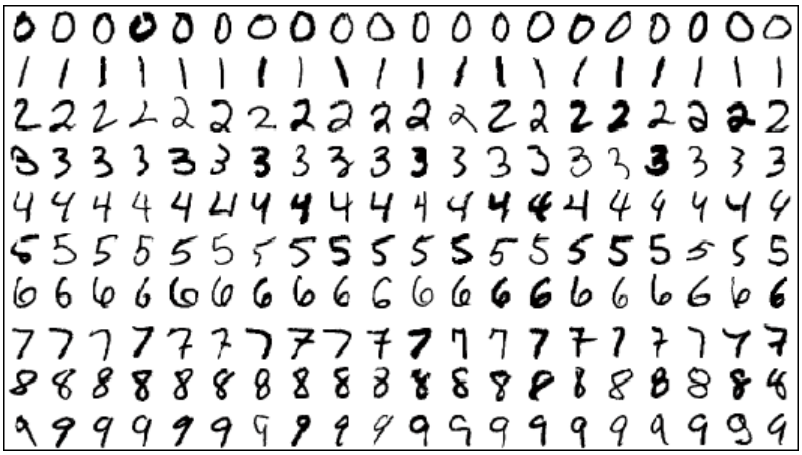


Figura 6-1. Base de datos MNIST.

Así, a la red se le pasarán 10,000 imágenes de los números escritos a mano, las primeras 5,000 se tratan de números escritos de manera normal y las restantes son números girados o deformados con la intención de "engañar" a la red. Estos últimos pueden observarse en la Figura 6-2.

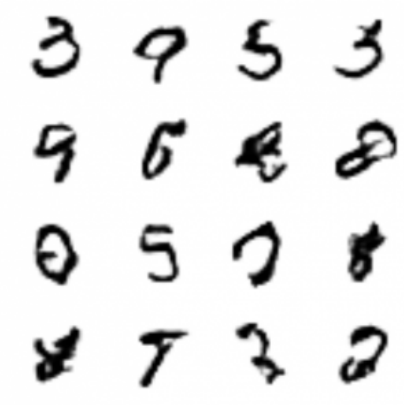


Figura 6-2. Números deformados.

Tras pasarle todas estas imágenes, la red convolucional realiza la predicción de a qué número se corresponde cada imagen de entrada y comprueba que la salida sea correcta o no. Finalmente, sabiendo el número de aciertos y de fallos en las predicciones se calcula el porcentaje de acierto. Este proceso de comprobación tarda aproximadamente entre 3 y 5 minutos. Los resultados obtenidos son los siguientes:

```
Epoch 3: Cost on iteration 694 is 0.139129
Epoch 3: Cost on iteration 695 is 0.134319
Epoch 3: Cost on iteration 696 is 0.148567
Epoch 3: Cost on iteration 697 is 0.199481
Epoch 3: Cost on iteration 698 is 0.173908
Epoch 3: Cost on iteration 699 is 0.181627
Epoch 3: Cost on iteration 700 is 0.176238
Epoch 3: Cost on iteration 701 is 0.161384
Epoch 3: Cost on iteration 702 is 0.137586
Accuracy is 95.629563
>
```

Figura 6-3. Resultado del test.

Como se puede comprobar, tras 702 iteraciones de entrenamiento, la red ha obtenido un porcentaje de acierto en el test del 95.63 %. Este resultado tan bueno, incluso supera la meta impuesta al principio de esta memoria de conseguir al menos un 95% de acierto en la red y corrobora que la implementación en R ha sido todo un éxito.

Además, es interesante remarcar que el resultado obtenido es excepcional teniendo en cuenta que ha sido capaz de acertar en aproximadamente 9,563 casos de un total de 10,000, o lo que es lo mismo, ha fallado en tan solo 437 casos. Además, para darle más relevancia al resultado obtenido, hay que tener en consideración que 5,000 de estas imágenes están de alguna manera modificadas haciendo más difícil que la red acierte el número.

6.2 Conclusiones

Durante el desarrollo del presente proyecto, se ha podido ver la gran potencia que tienen las redes neuronales y, concretamente, las redes neuronales convolucionales. Ha quedado de manifiesto cómo estos sistemas, que son relativamente fáciles de implementar, son capaces de conseguir unos resultados muy precisos en cuanto a problemas de clasificación e identificación de imágenes se refiere. Como ha quedado demostrado, con 1 hora aproximada de entrenamiento en un ordenador de gama media, la red ha obtenido un 96 % de acierto con un set de entrenamiento de 10,000 imágenes, de las cuales 5,000 de ellas estaban distorsionadas.

En lo que respecta a R, éste ha demostrado ser un lenguaje de programación perfecto para este proyecto. Por un lado, tiene muchas similitudes con Matlab, tanto a nivel del entorno de programación (tiene consola de comandos, un workspace, representa gráficas e imágenes, etcétera) como a nivel de codificación (tiene muchas funciones implementadas que son iguales o muy parecidas a las de Matlab y permite trabajar de manera óptima matricialmente). Gracias a lo anterior, aunque no todos los aspectos entre R y Matlab son directamente equivalentes, programar todos los scripts ha resultado ser, en su mayor parte, sencillo.

El objetivo que se perseguía lograr con el vigente trabajo es el de resultar de introducción al campo de las redes neuronales y de las redes neuronales convolucionales. En específico, se han visto las redes convolucionales aplicadas al reconocimiento de imágenes, entrenando a la red para que sea capaz de distinguir números escritos a mano en imágenes en escala de grises de 28 x 28. Además, también trata de mostrar la gran versatilidad y desempeño que estas redes convolucionales tienen, siendo capaces de "aprender" a solucionar problemas de multitud de campos distintos. Durante la lectura de estas páginas, se ha podido aprender cómo funcionan y cómo se entrenan las redes neuronales más básicas, cómo es el esquema de una red convolucional totalmente completa y práctica y cómo es el procedimiento de entrenamiento de éstas.

Por último, otro punto a destacar es la rapidez con la que la red es entrenada y testeada, tardando en un sistema portátil de gama media alrededor de 1 hora en realizar el proceso completo. Esto evidencia el buen comportamiento tanto del programa creado como del lenguaje de programación escogido, R.

7 REFERENCIAS

[1] Deeplearning.stanford.edu. (2017). Unsupervised Feature Learning and Deep Learning Tutorial. [online] Available at: http://deeplearning.stanford.edu/tutorial/.

[2] T. Hagan, Martin. B. Demuth, Howard. Hudson Beale, Mark. De Jesús, Orlando. (1996). *Neural Network Design.* 2ª edición.

[3] Deeplearning.net. (2017). Convolutional Neural Networks (LeNet) — DeepLearning 0.1 documentation. [online] Available at: http://deeplearning.net/tutorial/lenet.html.

[4] Jiménez Motte, Fernando. Ayuque Arenas, Kevin José Marino. (2016). *Diseño de un sistema de clasificación*
de señales de tránsito vehicular utilizando redes neuronales convolucionales.

[5] Isasi Viñuela, Pedro. Galván León, Inés M. (2004). *Redes de neuronas artificiales: un enfoque práctico.* 1ª edición.

[6] Núñez Sánchez-Agustino, Francisco José. (2016). *Diseño de un sistema de reconocimiento automático de matrículas de vehículos mediante una red neuronal convolucional.* Trabajo Fin de Máster. Barcelona: Universitat Oberta de Catalunya.

[7] Zamora, Erik. (2015). *Redes neuronales convolucionales.* [online]
Availabe at: https://es.scribd.com/doc/295974900/Redes-Neuronales-Convolucionales

[8] Matich, Damián Jorge (2001). *Redes neuronales: Conceptos Básicos y Aplicaciones.* Cátedra. Rosario: Universidad Tecnológica Nacional – Facultad Regional Rosario

[9] Lopez Briega, Raul E. (2016). *Redes neuronales convolucionales con TensorFlow.* [online]
Availabe at: https://relopezbriega.github.io/blog/2016/08/02/redes-neuronales-convolucionales-con-tensorflow/

[10] Google. (2017). *A Guide to TF Layers: Building a Convolutional Neural Network.* [online]
Availabe at: https://www.tensorflow.org/tutorials/layers

[11] McCaffrey, James. (2014). *Deep Neural Networks: A Getting Started Tutorial.* [online]
Availabe at: https://visualstudiomagazine.com/Articles/2014/06/01/Deep-Neural-Networks.aspx?Page=1

[12] Deshpande, Adit. (2016). A Beginner's Guide To Understanding Convolutional Neural Networks. [online]

Availabe at: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

[13] McCormick, Chris. (2015). Understanding the DeepLearnToolbox CNN Example. [online]

Availabe at: http://mccormickml.com/2015/01/10/understanding-the-deeplearntoolbox-cnn-example/

8 IMPLEMENTACIÓN EN R

En este último capítulo, se pondrán todos los códigos de las funciones realiadas uno a uno.

8.1 Códigos en R

cnnTrain.R: Inicializa todo el programa, con todas las funciones y variables

```
# cnnTrain

# Inicializa todo el programa, con todas las funciones y variables

rm(list = ls()) # Limpia el espacio de trabajo

##=========================================================================
=========================================================================
============
## INICIALIZAMOS LOS PARÁMETROS Y LOS DATOS DE ENTRADA

require('pracma')

# Inicialización de todas las funciones del programa
source('cnnFunctions.R')

# Parámetros de la red
filterDim = 9 # Dimensión de la matriz del filtro
numFilters = 20 # Número de filtros usados
poolDim = 2 # Dimensión de la matriz de pooling
numClasses = 10 # El número de clases existentes, números de 0 a 9
imageDim = 28 # Dimensión de las imágenes de entrada

# Obtenemos los datos para entrenar, usaremos la base de datos de caracterers
'mnist'
train = read.csv("C:\\Users\\Jaime\\Google
Drive\\Universidad\\Cuarto\\Trabajo Fin de Grado\\R\\Mnist\\mnist_train.csv")

# Adaptamos los datos para su uso
images = train[,-1] # Cogemos toda la matriz excepto la primera columna
images = t(images/255) # Se reescala de escala de grises a 0/1
numImages = ncol(images) # Obtenemos el número de imágenes para reescalar
images = array(images, dim = c(imageDim, imageDim, numImages)) # Reordenamos
la matriz a imagenes de 28x28

labels = train[,1] # Cogemos solo la primera columna de la matriz
labels = array(labels) # La convertimos a vector
labels[labels == 0] = 10 # Cambiamos los objetivos de 0 a 10 para poder
aplicar logaritmos luego

# Limpiamos datos innecesarios
rm(train)
rm(numImages)

# Inicializamos theta
theta = cnnInitParams(imageDim, filterDim, numFilters, poolDim, numClasses) #
Llamamos a la función que inicializa theta
theta = array(theta, c(dim(theta),1)) # Ponemos en forma de vector lo que
devuelve la función

##=========================================================================
```

```
================================================================================
============
## ENTRENAMIENTO DE LA RED

# Establecemos las opciones de entrenamiento
options.epochs = 3 # Número de veces que las imágenes serán pasadas
options.minibatch = 256 # Número de imágenes de cada paso
options.alpha = 0.15 # Valor del coeficiento alpha
options.momentum = 0.95 # Valor del momentum

opttheta = minFuncSGD(theta, images, labels, options) # Llama a la función
que realiza el entrenamiento

##================================================================================
================================================================================
============
## COMPROBACIÓN DE FUNCIONAMIENTO

# Testea el comportamiento de la red con el conjunto de prueba
test = read.csv("C:\\Users\\Jaime\\Google
Drive\\Universidad\\Cuarto\\TFG\\R\\Mnist\\mnist_test.csv")

# Adaptamos los datos para su uso
#testImages = test$x
testImages= test[,-1] # Coge toda la matriz excepto la primera columna
testImages = t(testImages/255) # Reescala los valores de [0, 255] a [0, 1]

#testLabels = test$y
testLabels = test[,1] # Coge solo la primera columna de la matriz

# Remap 0 values to 10
testLabels[testLabels == 0] = 10 # Cambia los valores que son 0 por 10

#numTests = test$n
numTest = ncol(testImages) # Obtiene el número de imágenes
testImages = array(testImages, dim = c(imageDim, imageDim, 9999)) # Reordena
la matriz a 28 x 28

# Borra datos innecesarios
rm(test)

# Comienza la comprobación del funcionamiento
pred = 1 # Establece la variable que indica que se esta testeando a 1
res = cnnCost(opttheta, testImages, testLabels, numClasses, filterDim,
numFilters, poolDim, pred) # Llama a la función que estima los caracteres
cost = res$cost # Guarda el resultado del coste
preds = res$preds # Guarda el vector con las predicciones
acc = sum((preds==testLabels)/length(preds))*100 # Calcula el porcentaje de
acierto de las predicciones

fprintf('Accuracy is %f\n',acc) # Muestra por pantalla el resultado del
porcentaje
```

cnnCost.R: Función que calcula el coste, el gradiente y los errores.

```r
# cnnCost

# Función que calcula el coste, el gradiente y los errores del algoritmo de
retropropagación de una red neuronal con softmax

cnnCost = function(theta, images, labels, numClasses, filterDim, numFilters,
poolDim, pred){
  # Obtiene los datos necesarios
  imageDim = dim(images)[1] # Obtiene la dimensión de la imagen
  numImages = dim(images)[3] # Obtiene el número de imágenes

  # Configura las matrices de peso y bias para el gradiende
  a = cnnParamsToStack(theta, imageDim, filterDim, numFilters, poolDim,
numClasses) # Llama a la función que devuelve los parámetros

  Wc = a$Wc # Asigna las correspondiented matrices
  Wd = a$Wd
  bc = a$bc
  bd = a$bd

  Wc_grad = array(0,dim(Wc))
  Wd_grad = array(0,dim(Wd))
  bc_grad = array(0,dim(bc))
  bd_grad = array(0,dim(bd))

  Wc_gradFilter = array(0, c(nrow(Wc_grad),ncol(Wc_grad)))


##=====================================================================
=====================================================================
============
  # PROPAGACIÓN HACIA DELANTE

  # Calcula las dimensiones de la matriz de activación
  convDim = imageDim - filterDim + 1

  # Calcula las dimensiones de la matriz tras el submuestreo
  outDim = convDim/poolDim

  # Inicializa la matriz de características de la capa de convolución
  activations = array(0,c(convDim,convDim,numFilters,numImages))

  # Inicializa la matriz de pooling para la capa de pooling
  activationsPooled = array(0,c(outDim^2*numFilters,numImages))
  activationsPool = array(0,c(outDim,outDim,numFilters,numImages))

  # Realiza la convolución de las imágenes con el filtro
  activations = cnnConvolve(filterDim, numFilters, images, Wc, bc) # Llama a
la función que calcula la convolución
  print(dim(activations))

  # Realiza el pooling a la matriz de activación
  activationsPool = cnnPool(poolDim, activations) # Llama a la función que
realiza el pooling

  # Reorganiza la matriz a una de 2 dimensiones para realizar el softmax
  activationsPooled = array(activationsPool, dim = c(outDim^2*numFilters,
numImages))

  # Capa de softmax
  probs = array(0, c(numClasses, numImages)) # Inicializa matriz donde se
guardan la posibilidades de que pertenezca a cada clase
  aux2 = array(0, c(numClasses, numImages)) # Variable auxiliar
```

```
  aux1 = Wd %*% activationsPooled + repmat(bd, 1, numImages)
  for(i in 1:ncol(aux1)){
    aux2[,i] = aux1[,i] - max(aux1[,i]) # Substracts the maximm value of the
matrix aux1
  }
  aux3 = exp(aux2)
  probs = sweep(aux3,2,colSums(aux3),`/`) # bsxfun(@rdivide, aux3, sum(aux3))

  rm(aux2)
  rm(aux3)


##=============================================================================
=============================================================================
============
  # CÁLCULO DEL COSTE

  cost = 0
  groundTruth = matrix(0, numClasses, numImages)
  groundTruth[cbind(labels, 1:numImages)] = 1 # cbind = sparse

  aux4 = groundTruth * probs
  aux5 = log(aux4[aux4 != 0])

  cost = -mean(aux5)

  rm(aux4)
  rm(aux5)

  # Si está en la fase de test calcula las predicciones sin hacer
retropopagación
  if(pred == 1){
    preds = array(0, numImages)
    for(j in 1:numImages){
      preds[j] = which.max(probs[,j])
    }

    grad = 0
    return(list(preds=preds, cost=cost))
  }


##=============================================================================
=============================================================================
============
  # RETROPROPAGACIÓN

  # Propaga el error hacia atrás a través de las diferentes capas
  # Guarda el error para calcular el gradiente de las siguientes iteraciones
  deriv_1 = (-1/numImages) * (groundTruth - probs)
  rm(groundTruth)

  Wd_grad = deriv_1 %*% t(activationsPooled)
  rm(activationsPooled)

  bd_grad = deriv_1 %*% array(1,c(numImages,1))
  deriv_2_pooled_sh = t(Wd) %*% deriv_1
  rm(deriv_1)

  deriv_2_pooled =
array(deriv_2_pooled_sh,c(outDim,outDim,numFilters,numImages))
  deriv_2_upsampled = array(0,c(convDim,convDim,numFilters,numImages))
```

```
  for(imageNum in 1:numImages){
    im = drop(images[,,imageNum])
    for (filterNum in 1:numFilters){
      aux3 = (1/(poolDim^2)) *
kron(drop(deriv_2_pooled[,,filterNum,imageNum]),matrix(1, poolDim, poolDim))
      deriv_2_upsampled[,,filterNum,imageNum] = aux3 *
activations[,,filterNum,imageNum] * (1 - activations[,,filterNum,imageNum])

      f_now = drop(deriv_2_upsampled[,,filterNum,imageNum])
      noww = conv2(im,rot90(drop(f_now),2))[c(20:28),c(20:28)]

      Wc_grad[,,filterNum] = drop(Wc_grad[,,filterNum]) + noww
      bc_grad[filterNum] = bc_grad[filterNum] + sum(f_now)
    }
  }

  grad =
array(c(as.vector(Wc_grad),as.vector(Wd_grad),as.vector(bc_grad),as.vector(bd
_grad)), c(dim(theta)))
  return(list(grad=grad, cost=cost))
}
```

minFuncSGD.R: Función que implementa el método de optimización de los parámetros SGD.

```
# minFuncSGD.R

# Calcula el stochastic gradient descent con momentum para optimizar los
```

```
parámetros

minFuncSGD = function(theta, data, labels, options){
  # Obtiene la configuración de la estructur options
  epochs = options.epochs
  alpha = options.alpha
  minibatch = options.minibatch

  # Obtiene el tamaño del set de entrenamiento
  numSamples = dim(labels)

  # Configura el momentum y la velocidad
  mom = 0.6
  momIncrease = 20
  velocity = array(0, c(dim(theta)))

  # Comienzan las iteraciones del SGD
  it = 0
  for(e in 1:epochs){
    # Escoge las imágenes de manera aleatoria
    rp = array(sample(numSamples))

    for(s in seq(1, numSamples - minibatch + 1, minibatch)){
      it = it + 1

      if(it == momIncrease){
        mom = options.momentum
      }

      # Obtiene el siguiente minibatch de manera aleatoria
      mb_data = data[,,rp[s:(s+minibatch-1)]]
      mb_labels = labels[rp[s:(s+minibatch-1)]]

      # Calcula el coste y el gradiente del minibatch
      pred = 0
      a = cnnCost(theta, mb_data, mb_labels, numClasses, filterDim,
numFilters, poolDim, pred)
      cost = a$cost
      grad = a$grad

      # Actualiza los parámetros con el stochastic gradient descent
      velocity = (mom * velocity) + (alpha * grad)
      theta = theta - velocity

      fprintf('Epoch %d: Cost on iteration %d is %f\n',e,it,cost)
    }

    # Después de cada epoch disminuye el ratio a la mitad
    alpha = alpha/2.0
  }

  # Guarda el resultado del entrenamiento para el test posterior
  opttheta = theta
  return(opttheta)
}
```

cnnConvolve.R: Función que implementa la convolución.

```
# cnnConvolve
```

```
# Función que calcula la convolución

cnnConvolve = function(filterDim, numFilters, images, W, b){
  # Obtenemos los datos necesarios
  numImages = dim(images)[3] # Obtenemos el número de imágenes
  imageDim = dim(images)[1] # Obtenemos la dimensión de la imagen
  convDim = imageDim - filterDim + 1 # Calculamos la dimensión de la matriz
de características
  convolvedFeatures = array(0,c(convDim,convDim,numFilters,numImages)) #
Inicializamos la matriz de características
  # Iniciamos las convolución
  for(imageNum in 1:numImages){
    # Obtenemos la imagen
    im = drop(images[,,imageNum]) # drop() es el equivalente de squeeze() en
matlab()
    for(filterNum in 1:numFilters){
      # Inicializamos la matriz donde se guardará el valor de la convolución
provisionalmente
      convolvedImage = array(0, c(convDim, convDim))

      # Asignamos a filter el filtro o kernel actual
      filter = drop(W[,,filterNum])

      # Rotamos 90º el kernel
      filter = rot90(drop(filter), 2)

      # Calculamos la convolución entre el filtro y la imagen, sumando el
resultado a convolvedImage
      # La convolución debe ser del tipo válida, por eso solo cogemos una
región del resultado
      convolvedImage = convolvedImage + conv2(im, filter)[c(5:24),c(5:24)]

      # Se le suma la bias, a continuación, se le aplica la función sigmoid
      convolvedImage = sigmoid(convolvedImage + b[filterNum])
      # Se va guardando el resultado en la matriz convolvedFeatures
      convolvedFeatures[,,filterNum,imageNum] = convolvedImage
    }
  }
  return(convolvedFeatures) # La función devuelve la matriz
convolvedFeatures
}
```

cnnPool.R: Función que implementa el pooling.

```
# cnnPool

# Realiza el pooling a la matriz dada por la capa convolucional
```

```
cnnPool = function(poolDim, activations){
  # Obtiene los datos necesarios
  numImages = dim(activations)[4] # Obtiene el número de imágenes
  numFilters = dim(activations)[3] # Obtiene el número de filtros
  convDim = dim(activations)[1] # Obtiene las dimensiones de la matriz de
características
  pooledFeatures = array(0,
c(convDim/poolDim,convDim/poolDim,numFilters,numImages)) # Inicializa la
matriz de pooling
  index = seq(1,convDim,poolDim) # Calcula el índice de los elementos de la
matriz pertenecientes a la esquina superior izquierda de cada iteración del
pooling
  for(imageNum in 1:numImages){
    for(filterNum in 1:numFilters){
      # Calcula el pooling
      topool = drop(activations[,,filterNum,imageNum])
      pooled = conv2(topool, matrix(1, poolDim, poolDim))[c(2:20),c(2:20)]
      pooledFeatures[,,filterNum,imageNum] =
pooled[index,index]/(poolDim*poolDim) # Guarda el resultado normalizado
    }
  }
  return(pooledFeatures) # Devuelve la matriz de pooling
}
```

cnnInitParams.R: Función que inicializa los parámetros para la red convolucional.

```
# cnnInitParams

# Inicizaliza los parámetros para la red convolucional
```

```
cnnInitParams = function(imageDim, filterDim, numFilters, poolDim,
numClasses){
  # Inicializamos los parámetros basados en el tamaño de las capas
  Wc =
0.1*array(rnorm(filterDim^2*numFilters),c(filterDim,filterDim,numFilters))

  # Dimensión resultante después de la convolución
  outDim = imageDim - filterDim + 1

  # Dimensión de la capa posterior al pooling
  outDim = outDim/poolDim
  hiddenSize = outDim^2*numFilters

  # Los pesos son escogidos entre -r y r
  r = sqrt(6)/sqrt(numClasses + hiddenSize + 1)
  Wd = array(runif(numClasses*hiddenSize), c(numClasses, hiddenSize)) * 2 *
r - r

  bc = array(0, c(numFilters,1))
  bd = array(0, c(numClasses,1))

  # Agrupa todos los pesos y bias en formato vector en theta
  theta = array(c(as.vector(Wc),as.vector(Wd),as.vector(bc),as.vector(bd)))
  return(theta) # Devuelve theta
}
```

cnnParamsToStack.R: Función que convierte el vector de parámetros en las matrices y vectores de pesos y bias para la red neuronal.

```
# cnnParamsToStack
```

```
# Convierte el vector de parámetros en las matrices y vectores de pesos y
bias para la red neuronal

cnnParamsToStack = function(theta, imageDim, filterDim, numFilters, poolDim,
numClasses){
  outDim = (imageDim - filterDim + 1)/poolDim
  hiddenSize = outDim^2*numFilters
  # Reordenamos theta
  indS = 1
  indE = filterDim^2*numFilters
  Wc = array(theta[indS:indE],c(filterDim,filterDim,numFilters))
  indS = indE+1
  indE = indE+hiddenSize*numClasses
  Wd = array(theta[indS:indE], c(numClasses, hiddenSize))
  indS = indE+1
  indE = indE+numFilters
  bc = array(theta[indS:indE])
  bd = array(theta[indE+1:nrow(theta)],c(nrow(theta)-indE))
  return(list(Wc=Wc, Wd=Wd, bc=bc, bd=bd))
}
```

cnnFunctions.R: Carga todas las funciones en el workspace.

```
# cnnFunctions
```

```
# Carga en el workspace todas las funciones que se usarán

source('cnnMnist.R')
source('cnnInitParams.R')
source('conv2.R')
source('cnnParamsToStack.R')
source('cnnConvolve.R')
source('cnnPool.R')
source('cnnCost.R')
source('minFuncSGD.R')
```