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

Sistema de reconocimiento de caracteres manuscritos usando Redes Neuronales Convolucionales implementado en Python

Autor: Alejandro Trujillo Quevedo
Tutor: Alejandro del Real Torres

Dep. 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

Sistema de reconocimiento de caracteres manuscritos usando Redes Neuronales Convolucionales implementado en Python

Autor:
Alejandro Trujillo Quevedo

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: Sistema de reconocimiento de caracteres manuscritos usando Redes Neuronales Convolucionales implementado en Python

Autor: Alejandro Trujillo Quevedo

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 mi familia

A mis maestros

Agradecimientos

Me gustaría, antes de todo, agradecer a mi tutor, Alejandro del Real Torres, por la oportunidad brindada durante estos meses, así como la ayuda y el apoyo prestados durante la realización de este proyecto.

A mi familia por su inquebrantable confianza y su inestimable apoyo durante todos y cada uno de estos años, en los que me alentaban a no rendirme y seguir adelante.

A mis amigos, los de toda la vida y las grandes amistades formadas durante estos años, por el apoyo incondicional y la ayuda prestada cuando más lo necesitaba.

Resumen

Durante los últimos años, el uso de Redes Neuronales Artificiales ha sufrido un crecimiento exponencial, tanto en funciones como en precisión, y es rara aquella empresa que aún no ha tratado de integrarlas de alguna forma en el servicio prestado.

Los usos de estas redes son muy variados, ya que, por ejemplo, se pueden utilizar para el reconocimiento de sonidos, tal y como hace Google con su aplicación Google Now. Además de esto, se pueden usar Redes Neuronales para la traducción instantánea de texto, tal como hace Skype, o para la predicción de la siguiente palabra a escribir al mandar un mensaje como hace la aplicación de teclado Swiftkey. Sin embargo, el uso más importante y extendido es el de reconocimiento de imágenes.

En este ámbito, grandes empresas como, por ejemplo, Facebook han conseguido desarrollar algoritmos usando estas redes capaces de reconocer rostros humanos con una precisión superior al 97%. Microsoft, del mismo modo, ha implementado en Cortana, su asistente personal, una Red Neuronal capaz de clasificar especies animales.

Dada la importancia del reconocimiento de imágenes hoy en dia, se ha decidido realizar en el presente proyecto una Red Neuronal capaz de identificar imágenes de caracteres manuscritos con una precisión deseable por encima del 95%.

Abstract

During the latest years, the use of Artificial Neural Networks has undergone an exponential growth, both in functions as in accuracy. The number of companies trying to apply these networks somehow onto their services has increased drastically.

The uses of these networks are very varied since they can be used in sound recognition, as Google does with its application Google Now. Furthermore, Neural Networks can be used to translate texts on the fly, as Skype does, or to predict the next word when typing a message, as Swiftkey keyboard application does. However, the most important and widespread use is image recognition.

In this area, large companies such as Facebook have been able to develop algorithms using Neural Networks which can recognize human faces with an accuracy over 97%. Likewise, Microsoft has introduced in Cortana, its personal assistant, a Neural Network capable of classifying animal species.

Given the importance of image recognition nowadays, it has been decided to carry out in the present poject a Neural Network capable of identifying images of handwritten characters with a desirable accuracy over 95%.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

a	Escalar
a	Vector
A	Matriz
$\in$	Perteneciente a
$\Re^n$	Conjuntos de números reales de dimensión n
e	Número e
x_j	Elemento j-ésimo del vector $\boldsymbol{x}$
$\boldsymbol{x}^{(i)}$	Columna i-ésima de la matriz $\boldsymbol{X}$ (por comodidad se denotará esta matriz como $\boldsymbol{x}$)
$\boldsymbol{A}^T$	Matriz $\boldsymbol{A}$ traspuesta
$\sum_i y^{(i)}$	Sumatorio para todo i de los elementos $y^{(i)}$ del vector $\boldsymbol{y}$
∇_θ	Gradiente con respecto a $\boldsymbol{\theta}$
$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}$	Derivada parcial de la función $J(\boldsymbol{\theta})$ con respecto al i-esimo elemento del vector $\boldsymbol{\theta}$
σ(z)	Función sigmoide
$P(y = 1\|\boldsymbol{x})$	Probabilidad de que *y* valga 1 conociendo $\boldsymbol{x}$
$1\{a\ true\ statement\}$	Función indicadora que vale 1 si el "statement" es cierto y 0 en caso contrario
$f'(z)$	Derivada de la función $f(z)$
$\odot$	Función producto de Hadamard
$*$	Función convolución

1 INTRODUCCIÓN

Debido al deseo de mejorar sus condiciones de vida, el ser humano ha descubierto la manera de sustituir el esfuerzo físico o mental que acarreaba la realización de ciertas tareas mediante la creación de aparatos.

De acuerdo con lo anterior, la creación de los ordenadores ha permitido que el avance tecnológico tenga un avance sin precedentes en las últimas décadas. Esto ha desembocado en la resolución de cada vez más problemas de manera algorítmica, mediante bases de datos, programación top-down, programación orientada a objetos y demás métodos. Sin embargo, para resolver problemas tales como reconocer imágenes, analizar patrones o descubrir similitudes en objetos, es necesario realizar un cambio de paradigma y emplear soluciones no convencionales.

Con el objetivo de resolver situaciones como las planteadas anteriormente, los científicos empezaron a estudiar las capacidades humanas cerebrales, haciendo de éstas las bases de la inteligencia artificial. Ésto permitió el desarrollo de redes neuronales, algoritmos genéticos y lógica difusa entre otros.

De esta manera, el objetivo de las redes neuronales no es la resolución de problemas de índole compleja haciendo uso de una secuencia de pasos, sino como una evolución de un sistema computacional inspirado en el cerebro humano.

1.1 Historia y Estado del Arte

La historia de las Redes Neuronales Artificiales proviene de varios campos que buscaron desarrollar la idea de encontrar un modelo matemático que asimilara la forma y el comportamiento de las neuronas humanas.

A lo largo de los siglos XIX y XX se empezó a desarrollar teorías de aprendizaje, visión y acondicionamiento por parte de un equipo de físicos, psicólogos y neurofisiólogos. Sin embargo, aún no se disponía de un modelo específico para la neurona.

En la década de los 40 se probó que las redes neuronales podían realizar cualquier operación aritmética o lógica. Sin embargo, no fue hasta la década de los 50 cuando se inventó el perceptrón, así como las reglas de aprendizaje. Desde ese momento se empezaron a desarrollar más algoritmos y la implementación de éstos creció, sin embargo, solo podían resolver una limitada serie de problemas. Ésto, unido a la falta de computación digital por parte de los ordenadores, resultó en el abandono de este campo durante un tiempo.

El resurgir de la investigación en esta área fue gracias al algoritmo de retropropagación (Backpropagation algorithm). Desde entonces se han escrito muchos artículos y se han encontrado múltiples aplicaciones. Actualmente, la gran capacidad computacional de los ordenadores permite explorar nuevos conceptos, arquitecturas y reglas de aprendizaje.

Con el fin de entender el estado del arte en el campo de las redes neuronales, se van a explicar algunas de las aplicaciones de éstas actualmente:

- Reconocimiento de Caracteres: Esta idea tuvo mucho auge y se hizo muy popular hace algunos años, pues permite reconocer texto manuscrito.
- Compresión de imágenes: Las redes neuronales pueden recibir y procesar grandes cantidades de información a la vez, siendo esto útil en la compresión de imágenes.
- Mercado de Valores: Estas redes pueden examinar una gran cantidad de información de forma rápida y organizar todo de manera que se pueda hacer un adecuado estudio de proyección y predecir el valor de las acciones.

- Medicina: una de las áreas que más ha ganado la atención es en la detección de afecciones cardiopulmonares, es decir compara muchos modelos distintos para identificar similitud en patrones y síntomas de la enfermedad. Estos sistemas ayudan a los médicos con el diagnóstico por el análisis de los síntomas reportados y las resonancias magnéticas y rayos x. También se han usado para dispositivos analizadores de habla para ayudar a personas con sordera profunda, monitorización de cirugías, predicción de reacciones adversas a un medicamento, entendimiento de causas de ataques epilépticos.
- Militar: Las redes neuronales juegan un papel importante en el campo de batalla, especialmente en aviones de combate y tanques que son equipados con cámaras digitales de alta resolución que funciona conectado a un computador que continuamente explora el exterior de posibles amenazas. De igual manera se pueden emplear para clasificar señales de radar o creación de armas inteligentes.
- Diagnóstico de Maquinas: A nivel industrial cuando una de estas máquinas presenta fallos automáticamente las apaga cuando esto ocurre.
- Análisis de Firmas: Las redes neuronales pueden ser empleadas para la comparación de firmas generadas. Esta ha sido una de las primeras aplicaciones implementada a gran escala en USA.
- A nivel Administrativo: Identificaciones de candidatos para posiciones específicas, optimización de plazas y horarios en líneas de vuelo, minería de datos.

Como se puede observar, la multitud de usos que estas redes tienen en casi cualquier campo pone de manifiesto la gran importancia de las mismas, además de hacer ver que las líneas de investigación son cada vez mayores y están llevadas a cabo por las empresas de mayor importancia internacional.

Por último, destacar el logro más importante de este tipo de inteligencia artificial en los últimos años: la derrota del campeón mundial de Go a manos de AlphaGo, la inteligencia artificial de Google.

Go es un juego de tablero que nació en China, y que tiene más de 2.500 años de antigüedad, cuyo objetivo final no es otro más que ser capaz de rodear con unas piedras un área mayor que la del oponente.

En el caso de AlphaGo, los ingenieros de DeepMind enviaron al sistema más de 30 millones de movimientos realizados por los mayores expertos del juego Go. Luego, con una segunda red neuronal conocida como "aprendizaje de refuerzo", hicieron que el sistema jugara contra sí mismo miles de veces. A cada partida, el "software" iba aprendiendo y se hacía cada vez más inteligente en tiempo real. Los creadores del programa hicieron que el sistema además analizara los resultados de las partidas contra sí mismo para predecir futuros resultados y planificar nuevas estrategias. Es decir, inventaron un sistema capaz de analizar el presente e intuir el futuro, un programa que toma decisiones de forma muy parecida a como lo hacemos los humanos.

En la segunda partida, el surcoreano Lee Sedol, campeón mundial del juego Go, se levantó de la silla desesperado ante un movimiento incomprensible, pero que fue un golpe magistral. Desencajado, necesitó 15 minutos para improvisar una respuesta, la cual no sirvió de nada, pues cuatro horas más tarde perdió la partida. La máquina volvería a ganarle en otras dos ocasiones.

1.2 Motivación

Esta investigación tiene como motivación adentrarse en el mundo de la visión por computador, la inteligencia artificial y, más concretamente, las redes neuronales. Dado que este campo es altamente complejo e inabarcable en su totalidad, se ha decidido empezar por los conceptos básicos y, una vez adquiridos, se procederá a crear de una red capaz de identificar y clasificar caracteres manuscritos, en concreto el MNIST (Modified National Institute of Standards and Technology). Esta base de datos es un análogo a "hello world" para la visión por computador. Desde su lanzamiento en 1999, esta base de datos ha servido como referencia para algoritmos de clasificación.

Mi motivación personal es la de introducirme en el campo de la inteligencia artificial, el cual ha captado mi atención debido a la extensa cantidad de logros conseguidos en un periodo relativamente corto de tiempo. Además de todo esto, me motiva la idea de saber que al final del proyecto comprenderé gran parte de los fundamentos que usan la mayoría de servicios del Smartphone, redes sociales o incluso alguna que otra aplicación. Acciones que realizamos todos cada día, tras las cuales están presentes las redes neuronales sin ni siquiera percatarnos de ello.

1.3 Objetivos

El principal objetivo del proyecto es realizar un software capaz de reconocer caracteres manuscritos con una precisión deseable por encima del 95%. Para alcanzar esta meta, se va a proceder a realizar un tutorial proporcionado por la universidad de Stanford, el cual implementa en MATLAB, desde los conocimientos más básicos hasta un apartado de reconocimiento de las imágenes del MNIST. Posteriormente, con el fin de comprobar que los conceptos están totalmente comprendidos y dado que la mayoría de sistemas actuales no usan este lenguaje, se implementará en Python. Además, esto permitirá comprobar las diferencias entre los dos lenguajes y ver cuál es más óptimo.

Por tanto, el orden lógico de los objetivos a alcanzar es:

- Implementar un algoritmo de aprendizaje y optimización simple para entender los conceptos básicos.
- Realizar un algoritmo de aprendizaje capaz de clasificar dos clases.
- Ampliar el concepto anterior para n clases.
- Crear la primera red multicapa.
- Implementar una Red Neuronal Convolucional capaz de clasificar caracteres manuscritos.
- Exportar la idea anterior a Python.
- Optimizar, en la medida de lo posible el rendimiento y el tiempo de entrenamiento de la red.

1.4 Estructura del proyecto

La estructura a seguir en el presente Proyecto es intrínsecamente relacionada con los objetivos descritos en la sección anterior. Este primer apartado es una introducción en el que se ve el porqué de realizar un estudio sobre redes neuronales, el estado del arte y los objetivos a alcanzar.

Posteriormente, se explicarán los algoritmos de aprendizaje y optimización más sencillos, que servirán de base para el objetivo final, y se implementarán, viendo los resultados obtenidos y la posible optimización del código.

Tras esto, se diseñará la primera red multicapa, la cual es de vital importancia para el reconocimiento de imágenes y se explicará uno de los algoritmos de optimización.

A continuación, se implementará una red neuronal convolucional capaz de detectar caracteres manuscritos en MATLAB usando los conceptos anteriores. Tras obtener buenos resultados, se migrará el código a Python para comprobar cuál de los dos lenguajes es más óptimo para este tipo de tareas.

Por último, se presentarán unas conclusiones generales y vías para el desarrollo posterior de este proyecto.

2 HARDWARE Y SOFTWARE

En este apartado del proyecto se va a proceder a especificar cuales han sido los aparatos hardware en los que se ha realizado el proyecto y las herramientas software en las que se ha desarrollado el código encargado de hacerlo funcionar.

2.1 Hardware

El hardware utilizado para este proyecto es simplemente un ordenador de sobremesa y un portátil. Se empezó utilizando el portátil, sin embargo, debido a la antigüedad del mismo, las pruebas tardaban demasiado en realizarse, por lo que se usó un ordenador de sobremesa para acelerar el cálculo y obtener un mayor rendimiento.

El portátil usado es un *Asus N53SV*, el cual cuenta con un procesador Intel i7-2670QM a 2.2 GHz y 6 GB de memoria RAM.

Debido al extenso tiempo en realizar gran parte de los scripts, se procedió a realizar las pruebas en un ordenador de sobremesa, por lo que todos los resultados mostrados son en dicho ordenador. Cabe destacar que el código, por el contrario, funciona perfectamente y los resultados obtenidos en términos de precisión son idénticos.

Por último, comentar que el ordenador de sobremesa utilizado cuenta con un procesador Intel i7-6700 a 3.4 Ghz, 16 GB de memoria RAM y un disco duro tipo SSD de 256 GB.

El equipo utilizado, como puede comprobarse, es uno del que cualquier persona podría disponer en casa y que muchas de estas actualmente tienen.

2.2 Software

A lo largo de proyecto se han utilizado multitud de herramientas software, aunque pueden fácilmente dividirse en dos grupos. Un primer grupo sería en el que se han realizado las pruebas pertinentes para adquirir los conocimientos necesarios y un segundo grupo en el que se ha puesto de manifiesto la correcta comprensión de los conceptos previamente adquiridos.

En la primera parte del proyecto se ha trabajado con Windows 10 como sistema operativo y se ha seguido el tutorial proporcionado por la universidad de Stanford disponible en la siguiente dirección web: http://deeplearning.stanford.edu/tutorial/

Asimismo, en dicha dirección se puede descargar un archivo .rar que contiene todos los archivos necesarios para poder trabajar los apartados del tutorial. En las secciones posteriores se explicará el código contenido en dicho archivo. Cabe destacar que los scripts incluidos en él tienen extensión .m, por lo que se utilizará MATLAB, concretamente en su versión R2016b.

Una vez se hayan aprendido los conceptos correspondientes a la primera parte y se haya conseguido implementar el sistema de reconocimiento de caracteres con Redes Neuronales Convolucionales en MATLAB, se procederá a su implementación en Python.

Para ello, se instalará el sistema operativo Ubuntu 16.04 en una máquina virtual usando el programa VirtualBox en su versión 5.1.10. El programa encargado del reconocimiento de caracteres con Redes Neuronales Convolucionales se implementará en Python 2.7.

Cabe destacar el uso de dos librerías extremadamente importantes en Python: NumPy para tener un soporte para vectores y matrices y SciPy para añadir algunos algoritmos matemáticos necesarios.

3 APRENDIZAJE SUPERVISADO Y OPTIMIZACIÓN

3.1 Linear Regression

3.1.1 Teoría y conceptos

El fin a conseguir mediante la regresión lineal es predecir el valor de y dado un vector con datos de entrada $\boldsymbol{x} \in \Re^n$. Para ello, en un ejercicio propuesto como ejemplo se desea estimar el precio de una casa a partir de las características que la definen, por lo que y representa el precio de la casa y cada uno de los elementos x_j de $\boldsymbol{x}$ representan características que definen dicha casa, tales como superficie o numero de habitaciones. Se representará el conjunto de características de la i-ésima casa como $\boldsymbol{x}^{(i)}$ y el precio como $y^{(i)}$.

El objetivo pues, es encontrar una función que cumpla $y^{(i)} \approx h\left(\boldsymbol{x}^{(i)}\right)$ para cada ejemplo del set de entrenamiento. En el caso de que se encuentre dicha función, a la que se le introducirán suficientes ejemplos de casas con sus precios, cabe esperar que sea capaz de predecir el precio, una vez dadas las características de la casa, aunque éste no sea conocido.

En pos de encontrar la función presentada anteriormente, se debe decidir cómo representarla. Para empezar, se utilizarán funciones lineales: $h_\theta(\boldsymbol{x}) = \sum_j \theta_j x_j = \boldsymbol{\theta}^T \boldsymbol{x}$. Aquí, $h_\theta(\boldsymbol{x})$ representa una extensa familia de funciones parametrizadas en función de $\boldsymbol{\theta}$. El objetivo es, por tanto, encontrar una $\boldsymbol{\theta}$ tal que $h_\theta\left(\boldsymbol{x}^{(i)}\right)$ sea lo más cercano a $y^{(i)}$. Así pes, se busca una $\boldsymbol{\theta}$ que minimice:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_i \left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2}\sum_i \left(\boldsymbol{\theta}^T \boldsymbol{x} - y^{(i)}\right)^2 \tag{3-1}$$

Esta función se denomina coste y mide cuánto error se introduce en la predicción de $y^{(i)}$ para una elección concreta de $\boldsymbol{\theta}$.

Como se ha dicho, se busca una $\boldsymbol{\theta}$ que minimice $J(\boldsymbol{\theta})$. Existe una gran variedad de algoritmos destinados a minimizar funciones, de hecho, en el apartado 4.3 se explicará con detalle uno de ellos. Sin embargo, por ahora se va a dar por supuesto que la mayoría de estos algoritmos necesitan que se les proporcione la función coste $J(\boldsymbol{\theta})$ y el gradiente de dicha función $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ para cualquier valor de $\boldsymbol{\theta}$. Tras esto, el proceso de optimización restante para encontrar la mejor elección de $\boldsymbol{\theta}$ es llevado a cabo por el algoritmo de optimización.

El gradiente queda como:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_n} \end{bmatrix} = \begin{bmatrix} \sum_i x_1^{(i)}\left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \\ \sum_i x_2^{(i)}\left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \\ \vdots \\ \sum_i x_n^{(i)}\left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \end{bmatrix} \tag{3-2}$$

Expresado en forma vectorial:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_i \boldsymbol{x}^{(i)}\left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \tag{3-3}$$

3.1.2 Resultados experimentales

A continuación, se va a implementar una regresión lineal en MATLAB usando como ejemplo el código *ex1a_linreg.m* dentro de la carpeta *ex1/*. El código completo se encuentra disponible en el *Apéndice I – Código 1*, sin embargo, se procederá a explicar dicho código grosso modo y se insertarán aquellas partes del mismo que sean más interesantes, así como de los resultados obtenidos.

En primer lugar, se cargan los datos referentes a las casas y se ponen de manera que la i-ésima columna contenga las características correspondientes a la i-ésima casa. Además, dentro de este paquete de datos, la última fila se corresponde con el precio de la casa en cuestión. Tras cargar los datos, se permutan las columnas para que el resultado de la red no sea causal.

A continuación, se escogen 400 de las 506 casas como set de entrenamiento y el resto como set de prueba. Las características usadas como entradas para el algoritmo de entrenamiento se guardan como *train.X* y las usadas para probar su funcionamiento en *test.X*. Los precios de las casas se almacenarán en *train.y* y *test.y* respectivamente.

Posteriormente, se inicializa la variable *theta* de forma aleatoria y se procede a entrenar la red. Para ello, se utiliza la función *minFunc,* a la cual se le pasa la función *linear_regression* (el código de esta función está en el *Apéndice I – Código 2*), la *theta* inicial, los sets de entrenamiento y unas opciones especificas. La función de entrenamiento, como ya se comentó, se explicará en profundidad en el capitulo 4.3, sin embargo, se va a explicar la función *linear_regression*, que es la encargada de obtener el coste y el gradiente.

El coste y el gradiente se han calculado siguiendo las ecuaciones (3-1) y (3-2), para ello se han utilizado bucles tipo *for*. En concreto el código es el siguiente:

```
sumaJ=0;
for i=1:m
    sumaJ=sumaJ+(theta'*X(:,i)-y(1,i))^2;
end
f=0.5*sumaJ;

sumaG=0;
for j=1:n
    for i=1:m
        sumaG=sumaG+X(j,i)*(theta'*X(:,i)-y(1,i));
    end
    g(j,1)=sumaG;
    sumaG=0;
end
```

Por último, el código calcula el valor RMS tanto del set de entrenamiento como del de prueba (los cuales deberían ser como mucho 5) y muestra una gráfica en la que se comparan los precios reales y los predichos. A continuación, se muestran los valores obtenidos y la gráfica comparativa:

```
Optimization took 0.895590 seconds.
RMS training error: 4.745835
RMS testing error: 4.575823
>>
```

Figura 3-1: Valores RMS obtenidos y tiempo de ejecución

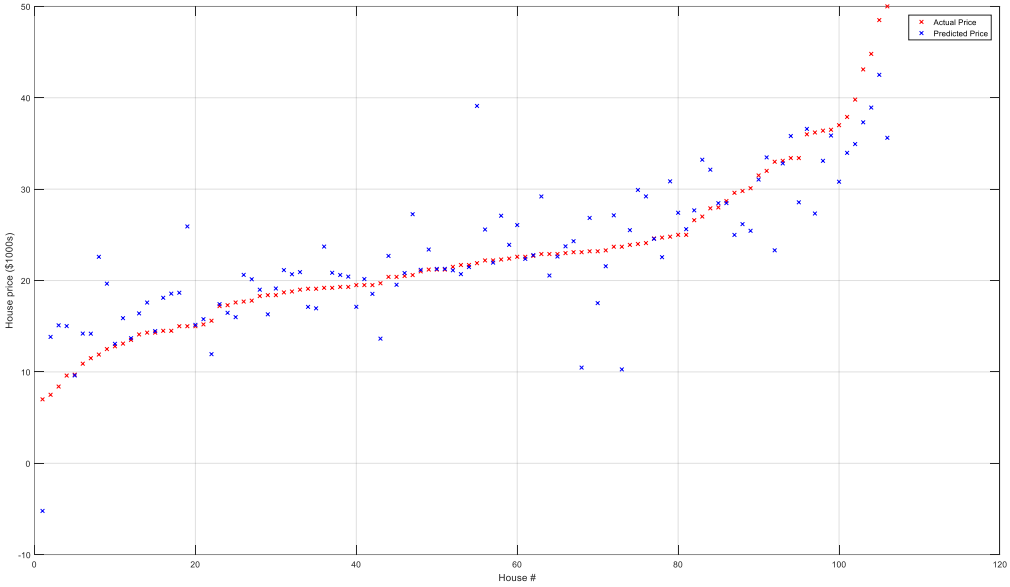

Figura 3-2: Gráfica con precios reales y predichos

3.1.3 Optimización y nuevos resultados

Aunque los resultados anteriores son buenos y el código funciona correctamente, éste es poco óptimo en la forma en que se ha implementado. Para este tipo de predicciones el código no necesita ser extremadamente rápido, sin embargo, en lo sucesivo una programación análoga ralentizaría el código más de lo necesario. La ejecución de bucles es lenta en MATLAB, por lo que es conveniente expresar el código en la medida de lo posible como operaciones matriciales, las cuales se ejecutan significativamente más rápido.

En este ejemplo concreto, la función coste y el gradiente son fácilmente sustituibles por operaciones matriciales teniendo en cuenta lo siguiente:

$$\hat{\boldsymbol{y}} = \boldsymbol{\theta}^T \boldsymbol{x} - \boldsymbol{y} \rightarrow \begin{cases} J(\boldsymbol{\theta}) = \frac{1}{2}\sum_i \hat{y}^{(i)^2} \\ \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{x}\hat{\boldsymbol{y}}^T \end{cases} \tag{3-4}$$

Si se expande la ecuación (3-3), se puede entender la segunda equivalencia de la llave de la ecuación (3-4):

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_i \boldsymbol{x}^{(i)} \left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) = \sum_i \boldsymbol{x}^{(i)} \hat{y}^{(i)} = \boldsymbol{x}^{(1)}\hat{y}^{(1)} + \boldsymbol{x}^{(2)}\hat{y}^{(2)} + \boldsymbol{x}^{(3)}\hat{y}^3 + \cdots =$$
$$= \begin{bmatrix} x_1^{(1)}\hat{y}^{(1)} + x_1^{(2)}\hat{y}^{(2)} + x_1^{(3)}\hat{y}^{(3)} + \cdots \\ x_2^{(1)}\hat{y}^{(1)} + x_2^{(2)}\hat{y}^{(2)} + x_2^{(3)}\hat{y}^{(3)} + \cdots \\ x_3^{(1)}\hat{y}^{(1)} + x_3^{(2)}\hat{y}^{(2)} + x_3^{(3)}\hat{y}^{(3)} + \cdots \\ \vdots \end{bmatrix} = \boldsymbol{x}\hat{\boldsymbol{y}}^T \tag{3-5}$$

Nótese que para calcular el coste solo es necesario sumar todas las componentes de un vector al cuadrado y dividir el resultado entre dos. Para esto, no es necesario el uso de bucles en MATLAB. Además, el gradiente queda expresado como una simple operación matricial.

Escrito en código de MATLAB, queda:

```
y_hat=(theta'*X-y);
f=1/2*(sum(y_hat.^2));

g=X*y_hat';
```

Ejecutando de nuevo el código *ex1a_linreg*, pero pasándole esta vez a la función de optimización la función *linear_regression_vec* (el código completo está en el *Apéndice I – Código 3*), la cual calcula el coste y el gradiente según el código anterior, se obtiene como resultado:

```
Optimization took 0.123545 seconds.
RMS training error: 4.694059
RMS testing error: 4.707466
>>
```

Figura 3-3: Valores RMS y tiempo de ejecución.

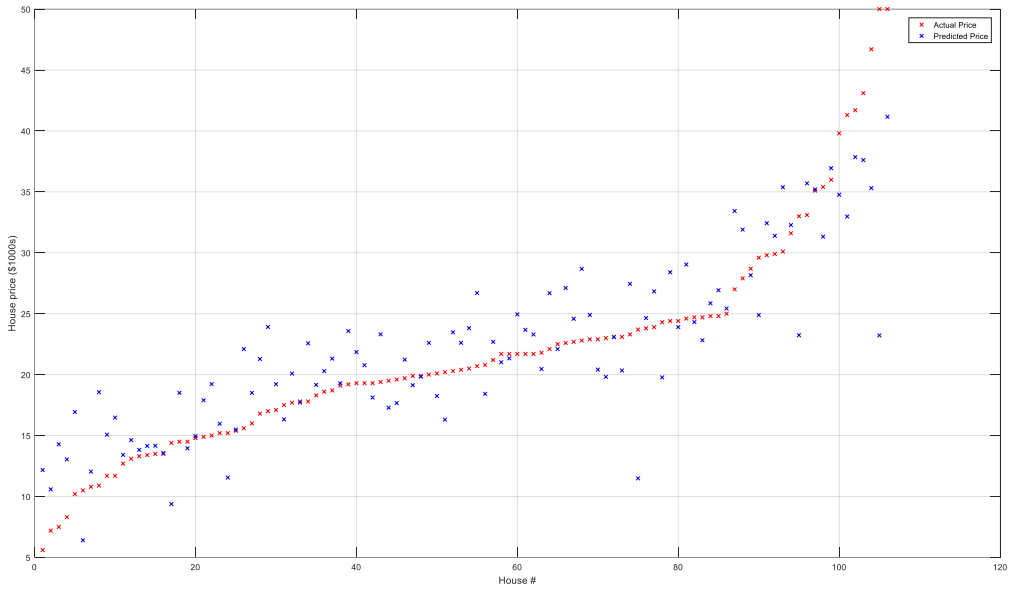

Figura 3-4: Gráfica con precios reales y predichos

Como se puede observar, los resultados obtenidos son prácticamente idénticos, sin embargo, se puede comprobar que el tiempo de ejecución es significativamente menor que en el caso anterior.

3.2 Logistic Regression

3.2.1 Teoría y conceptos

Gracias a la regresión lineal se pueden predecir cantidades continuas como el precio de las casas como una función lineal que depende de los datos de entrada. Sin embargo, en algunas ocasiones resulta de interés predecir variables discretas, es decir, para predecir por ejemplo si un conjunto de píxeles representa un "0" o un "1". Los problemas de este tipo son problemas de clasificación y la regresión logística es un algoritmo de clasificación bastante simple para aprender a tomar dichas decisiones.

En la regresión lineal, se probó a predecir los valores de $y^{(i)}$ usando una función lineal $\boldsymbol{y} = h_\theta(\boldsymbol{x}) = \boldsymbol{\theta}^T\boldsymbol{x}$. Este tipo de funciones no son las adecuadas para predecir valores binarios $\left(y^{(i)} \in \{0,1\}\right)$, por lo que se usa una hipótesis diferente para predecir la probabilidad de pertenecer a una clase frente a la contraria. Concretamente, se usará una función de la forma:

$$\begin{aligned} P(\boldsymbol{y} = 1|\boldsymbol{x}) = h_\theta(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T\boldsymbol{x}}} \equiv \sigma(\boldsymbol{\theta}^T\boldsymbol{x}) \\ P(\boldsymbol{y} = 0|\boldsymbol{x}) = 1 - P(\boldsymbol{y} = 1|\boldsymbol{x}) = 1 - h_\theta(\boldsymbol{x}) \end{aligned} \tag{3-6}$$

La función $\sigma(z) \equiv \frac{1}{1+e^{-z}}$, también llamada función "sigmoide" o "logística", contiene todos los valores de $\boldsymbol{\theta}^T\boldsymbol{x}$ dentro del rango $[0,1]$ para poder ser interpretados como probabilidad. El objetivo es encontrar un valor de $\boldsymbol{\theta}$ tal que la probabilidad $P(\boldsymbol{y} = 1|\boldsymbol{x})$ sea alta cuando $\boldsymbol{x}$ pertenece a la clase "1" y pequeña cuando pertenezca a la clase "0". La función coste que se usará con un set de entrenamiento cuyos targets son binarios es:

$$J(\boldsymbol{\theta}) = -\sum_i \left(y^{(i)} \log\left(h_\theta\left(\boldsymbol{x}^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(\boldsymbol{x}^{(i)}\right)\right)\right) \tag{3-7}$$

En la función anterior, solo uno de los dos términos del sumatorio es distinto de cero para cada ejemplo del set de entrenamiento (dependiendo si la clase de $y^{(i)}$ es "1" o "0"). Una vez que se tiene la función coste, es necesario aprender a clasificar los datos minimizando $J(\boldsymbol{\theta})$ para encontrar una $\boldsymbol{\theta}$ óptima. Una vez hecho esto, se pueden clasificar nuevos datos como "1" o "0" comprobando las probabilidades de pertenecer a cada una de las clases. Si $P(\boldsymbol{y} = 1|\boldsymbol{x}) > P(\boldsymbol{y} = 0|\boldsymbol{x})$, entonces se clasifica como "1", "0" en caso contrario. Esto último es lo mismo que comprobar $h_\theta(\boldsymbol{x}) > 0.5$.

Para minimizar la función coste se pueden utilizar las mismas herramientas que con la regresión lineal. Hay que proveer a la función optimizadora el cálculo del coste y del gradiente para cualquier valor de $\boldsymbol{\theta}$. En este caso, las componentes del gradiente se pueden calcular como:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = \sum_i x_j^{(i)} \left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \tag{3-8}$$

Escrito en forma vectorial, el gradiente se puede expresar como:

$$\nabla_\theta J(\boldsymbol{\theta}) = \sum_i \boldsymbol{x}^{(i)} \left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \tag{3-9}$$

Esta última ecuación es esencialmente la misma que la del gradiente para la regresión lineal, solo que ahora $h_\theta(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^T\boldsymbol{x})$, por lo que, haciendo $\widehat{\boldsymbol{y}} = \sigma(\boldsymbol{\theta}^T\boldsymbol{x}) - \boldsymbol{y}$ la equivalencia de la ecuación (3-5) sigue siendo válida.

3.2.2 Resultados experimentales

A continuación, se va a implementar una regresión logística en MATLAB usando como ejemplo el código *ex1b_logreg.m* dentro de la carpeta *ex1/*. El código completo se encuentra disponible en el *Apéndice II – Código 1*, sin embargo, se procederá a explicar dicho código grosso modo y se insertarán aquellas partes del mismo que sean más interesantes, así como de los resultados obtenidos.

En este ejercicio, se realizará un algoritmo capaz de clasificar imágenes del set de datos MNIST en las clases "0" y "1". Obviamente, para que esto sea posible solo se pueden tomar imágenes del MNIST que correspondan con estos dos valores. Algunas de las imágenes del set de datos son:

Figura 3-5: Imágenes del set de datos MNIST

Lo primero que se realiza en el código es cargar dichas imágenes mediante la función *ex1_load_mnist*, la cual almacena las imágenes en los sets de entrenamiento y de prueba. Las imágenes tienen una intensidad de pixel dentro del rango $[0,1]$ y se han pasado a formato vector. Por tanto, el i-ésimo elemento de la j-ésima columna de la matriz $\boldsymbol{x}$ representa el i-ésimo pixel de la j-ésima imagen, es decir, cada columna de la matriz de entrada es una imagen del set de datos convertida a formato vectorial

Tras obtener los sets de entrenamiento y prueba, se inicializa el vector $\boldsymbol{\theta}$, se define el máximo de iteraciones y se llama a la función encargada de entrenar los parámetros, a la que se le pasa una función que calcula el coste y el gradiente, el vector $\boldsymbol{\theta}$, las opciones y el set de entrenamiento. Una vez más, no se va a explicar en esta sección el algoritmo de optimización, solo la función encargada de calcular el coste y el gradiente. Dicha función es *logistic_regression_vec* y se puede encontrar completa en el *Apéndice II – Código 2*.

En este caso $\hat{y} = \sigma(\boldsymbol{\theta}^T \boldsymbol{x})$ y, por tanto, la función coste y el gradiente se pueden calcular con las siguientes líneas de código:

```
y_hat=logsig(theta'*X);
f=-sum(y.*log(y_hat)+(1-y).*log(1-y_hat));

g=X*(y_hat-y)';
```

Destacar que la función *logsig* es la función sigmoide en MATLAB. Los puntos delante de las operaciones son para que esas operaciones se realicen elemento a elemento evitando así, por tanto, la necesidad de utilizar bucles que ralentizarían dicho código. Los resultados obtenidos en este ejemplo son los siguientes:

```
Optimization took 1.836296 seconds.
Training accuracy: 100.0%
Test accuracy: 100.0%
```

Figura 3-6: Resultados de Regresión Logística

El resultado obtenido es del 100% debido a que es relativamente sencillo clasificar en "0" y "1", pues ambos tienen a ser muy diferentes. En futuros ejercicios será mucho más complicado conseguir resultados tan perfectos como éste.

3.3 Softmax Regression

3.3.1 Teoría y conceptos

La regresión softmax, también llamada regresión logística multinomial, es una generalización de la regresión logística para el caso en el que se manejen más de dos clases. En la regresión logística, se asumió que los objetivos eran binarios $y^{(i)} \in \{0,1\}$ y se utilizó el algoritmo anterior para distinguir entre dos tipos de dígitos manuscritos. La regresión softmax permite que $y^{(i)} \in \{1, \dots, K\}$, donde K es el número de clases. En el caso que nos abarca, el set MNIST tiene 10 clases diferentes.

Dado una entrada $\boldsymbol{x}$, el objetivo sería estimar la probabilidad $P(\boldsymbol{y} = k|\boldsymbol{x})$ para cada valor de $k = 1, \dots, K$, por lo que la salida será un vector de K elementos cuya suma es igual a 1. Por tanto:

$$h_\theta(\boldsymbol{x}) = \begin{bmatrix} P(\boldsymbol{y} = 1|\boldsymbol{x};\boldsymbol{\theta}) \\ P(\boldsymbol{y} = 2|\boldsymbol{x};\boldsymbol{\theta}) \\ \vdots \\ P(\boldsymbol{y} = K|\boldsymbol{x};\boldsymbol{\theta}) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} e^{\boldsymbol{\theta}^{(j)T}\boldsymbol{x}}} \begin{bmatrix} e^{\boldsymbol{\theta}^{(1)T}\boldsymbol{x}} \\ e^{\boldsymbol{\theta}^{(2)T}\boldsymbol{x}} \\ \vdots \\ e^{\boldsymbol{\theta}^{(K)T}\boldsymbol{x}} \end{bmatrix} \tag{3-10}$$

Aquí $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \dots, \boldsymbol{\theta}^{(K)} \in \Re^n$ son los parámetros de nuestro modelo. Asimismo, el termino $\frac{1}{\sum_{j=1}^{K} e^{\boldsymbol{\theta}^{(j)T}\boldsymbol{x}}}$ normaliza la distribución de manera que la suma sea uno.

Por conveniencia, se usará $\boldsymbol{\theta}$ para denotar a todos los parámetros del modelo. Al implementar una regresión softmax, es útil representar $\boldsymbol{\theta}$ como una matriz de $n \times K$ obtenida de la concatenación en columnas de $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \dots, \boldsymbol{\theta}^{(K)}$, es decir:

$$\boldsymbol{\theta} = \begin{bmatrix} | & | & | & | \\ \boldsymbol{\theta}^{(1)} & \boldsymbol{\theta}^{(2)} & \dots & \boldsymbol{\theta}^{(K)} \\ | & | & | & | \end{bmatrix} \tag{3-11}$$

A continuación, se describirá la función coste que se usará para la regresión softmax. En la ecuación (3-11), $1\{\cdot\}$ es una "función indicadora", de manera que $1\{a\ true\ statement\} = 1$ y $1\{a\ false\ statement\} = 0$. Por ejemplo $1\{2 + 2 = 4\} = 1$. Teniendo esto en cuenta, la función coste es:

$$J(\boldsymbol{\theta}) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)} = k\} \log \frac{e^{\boldsymbol{\theta}^{(k)T}\boldsymbol{x}^{(i)}}}{\sum_{j=1}^{K} e^{\boldsymbol{\theta}^{(j)T}\boldsymbol{x}^{(i)}}}\right] \tag{3-12}$$

En la regresión softmax se tiene que:

$$P(y^{(i)} = k|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^{(k)T}\boldsymbol{x}^{(i)}}}{\sum_{j=1}^{K} e^{\boldsymbol{\theta}^{(j)T}\boldsymbol{x}^{(i)}}} \tag{3-13}$$

Debido a que no se puede obtener el mínimo de la función coste de forma analítica, es necesario utilizar una vez más un algoritmo de optimización iterativo. Este algoritmo necesita, a su vez el gradiente, que se puede expresar como:

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \left[\boldsymbol{x}^{(i)} \left(1\{y^{(i)} = k\} - P\left(y^{(i)} = k | \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \right) \right] \tag{3-14}$$

En este caso, $\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\theta})$ es un vector tal que el j-ésimo elemento es $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j^{(k)}}$, es decir, la derivada parcial de $J(\boldsymbol{\theta})$ con respecto al j-ésimo elemento de $\boldsymbol{\theta}^{(k)}$. Con esto, ya se puede calcular el gradiente e introducírselo al algoritmo de optimización para minimizar $J(\boldsymbol{\theta})$.

3.3.2 Resultados experimentales

A continuación, se va a implementar una regresión softmax en MATLAB usando como ejemplo el código *ex1c_softmax.m* dentro de la carpeta *ex1/*. El código completo se encuentra disponible en el *Apéndice III – Código 1*, sin embargo, se procederá a explicar dicho código grosso modo y se insertarán aquellas partes del mismo que sean más interesantes, así como de los resultados obtenidos.

En este ejercicio se procederá a entrenar un clasificador capaz de manejar las 10 clases del set MNIST. El código es muy parecido al utilizado para la regresión logística, solo que esta vez se cargarán los sets de entrenamiento y prueba del MNIST completos. A los objetivos $y^{(i)}$ se les ha añadido un 1, de forma que $y^{(i)} \in \{1, \dots, 10\}$ (este cambio permite usar $y^{(i)}$ como índices dentro de una matriz).

El código ejecuta las mismas operaciones que la regresión logística: carga los sets de entrenamiento y prueba y llama a la función *minFunc*, a la que se le pasa la función *softmax_regression_vec* (se puede ver en el *Apéndice III – Código 2*), que calcula el coste y el gradiente.

Si expandimos la ecuación (3-12), se obtiene los siguientes términos:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\left[\sum_{i=1}^{m} \sum_{k=1}^{K} 1\{y^{(i)} = k\} \log \frac{e^{\boldsymbol{\theta}^{(k)T} \boldsymbol{x}^{(i)}}}{\sum_{j=1}^{K} e^{\boldsymbol{\theta}^{(j)T} \boldsymbol{x}^{(i)}}} \right] = \\ &= -\left[1\{y^{(1)} = 1\} \log \frac{e^{\boldsymbol{\theta}^{(1)T} \boldsymbol{x}^{(1)}}}{e^{\boldsymbol{\theta}^{(1)T} \boldsymbol{x}^{(1)}} + \cdots + e^{\boldsymbol{\theta}^{(K)T} \boldsymbol{x}^{(1)}}} \right. \\ &+ 1\{y^{(1)} = 2\} \log \frac{e^{\boldsymbol{\theta}^{(2)T} \boldsymbol{x}^{(1)}}}{e^{\boldsymbol{\theta}^{(1)T} \boldsymbol{x}^{(1)}} + \cdots + e^{\boldsymbol{\theta}^{(K)T} \boldsymbol{x}^{(1)}}} + \cdots \\ &\left. + 1\{y^{(m)} = K\} \log \frac{e^{\boldsymbol{\theta}^{(K)T} \boldsymbol{x}^{(m)}}}{e^{\boldsymbol{\theta}^{(1)T} \boldsymbol{x}^{(m)}} + \cdots + e^{\boldsymbol{\theta}^{(K)T} \boldsymbol{x}^{(m)}}} \right] \end{aligned} \tag{3-15}$$

Si uno se fija, solo se sumará para cada $y^{(i)}$ aquel término k que cumpla la igualdad y el resto se descartarán. Para implementar dichos términos sin utilizar bucles, se procederá a explicar la analogía entre la expresión anterior, en concreto el cociente dentro del logaritmo, y las siguientes operaciones matemáticas:

$$\boldsymbol{\theta}^T\boldsymbol{x} = \begin{bmatrix} \boldsymbol{\theta}^{(1)T}\boldsymbol{x}^{(1)} & \boldsymbol{\theta}^{(1)T}\boldsymbol{x}^{(2)} & \dots & \boldsymbol{\theta}^{(1)T}\boldsymbol{x}^{(m)} \\ \boldsymbol{\theta}^{(2)T}\boldsymbol{x}^{(1)} & \boldsymbol{\theta}^{(2)T}\boldsymbol{x}^{(2)} & \dots & \boldsymbol{\theta}^{(2)T}\boldsymbol{x}^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{\theta}^{(K)T}\boldsymbol{x}^{(1)} & \boldsymbol{\theta}^{(K)T}\boldsymbol{x}^{(2)} & \dots & \boldsymbol{\theta}^{(K)T}\boldsymbol{x}^{(m)} \end{bmatrix} \tag{3-16}$$

Por tanto, la multiplicación matricial anterior permite, tras elevar e a dicha matriz, tener en una matriz todos los términos que corresponden al dividendo del cociente en la ecuación (3-15). Además, es fácil darse cuenta que el divisor de cada término perteneciente a la i-ésima columna no es más que la suma de todos los elementos de la i-ésima columna de la ecuación (3-16) (tras elevar e a dicho valor, obviamente). Por tanto, el cociente se puede realizar mediante el comando *bsxfun,* el cual permite realizar la división anterior.

Una vez obtenido el logaritmo del cociente de la expresión, cuyo resultado es una matriz, solo resta realizar la suma de aquellos elementos de la matriz que cumplen $1\{y^{(i)} = k\}$. El hecho de haber expresado $y^{(i)} \in \{1, \dots, 10\}$ de dicha forma permite utilizarlo como índices mediante el comando *sub2ind*. Este comando obtiene los índices de una matriz de acuerdo a los argumentos, que actúan indicando la fila y columna.

Para su mejor compresión, se supondrá que se tiene la matriz $\boldsymbol{A} = \begin{bmatrix} 1 & 3 & 4 \\ 10 & 9 & 8 \\ 2 & 7 & 5 \end{bmatrix}$ y que se usa el comando *ind=sub2ind(size(A), [1 3 2], 1:3);*. Con dicho comando las filas son $[1 \quad 3 \quad 2]$ y las columnas $[1 \quad 2 \quad 3]$, por lo que se obtendrán los índices de los elementos A_{11}, A_{32} y A_{23}. Si se ejecuta el comando *A(ind)*, se obtiene un vector con valores $[1 \quad 7 \quad 8]$.

Por tanto, con este comando se pueden obtener aquellos índices que cumplen $1\{y^{(i)} = k\}$, haciendo de $\boldsymbol{y}$ las filas y de *1:size(A)* las columnas.

A continuación, se va a mostrar el código en MATLAB:

```
y_hat=exp(theta'*X);  % It's the numerator
y_hat=[y_hat; ones(1,size(y_hat,2))]; % Last row set to 1 to have 10 classes
y_hat_sum=sum(y_hat,1); % It's the denominator
y_hat_sum=y_hat_sum-1; % As last row doesn't count, substract it from the total sum

coc=bsxfun(@rdivide,y_hat,y_hat_sum);
A=log(coc);
ind=sub2ind(size(A),y,1:size(y_hat,2));
A(end,:)=0; % Last row set to 0 because the prob that y belongs to 10 is null
f=-sum(A(ind));

B=zeros(size(A));
B(ind)=1;
g=-X*(B-coc)';

g=g(:,1:end-1);  % Last row doesn't count
g=g(:); % make gradient a vector for minFunc
```

El hecho de establecer $y^{(i)} \in \{1, \dots, 10\}$ tiene la ventaja de poder usarse como índices, pero la desventaja es que ahora $P\left(y^{(i)} = 10 \middle| \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) = 0$. Por tanto, se trabajará como si fueran 10 clases (de ahí el incluir una fila más en *y_hat*) y posteriormente se eliminará dicha clase del coste y del gradiente, tal y como se observa en el código anterior.

A continuación, se va a proceder a mostrar los resultados obtenidos al ejecutar el script:

```
Optimization took 8.517391 seconds.
Training accuracy: 87.2%
Test accuracy: 87.6%
```

Figura 3-7: Resultados Regresión Softmax

Como se puede observar en la imagen anterior, el resultado es superior al 87%, lo cual no está nada mal teniendo en cuenta que este es el algoritmo más simple que se puede crear con dicho propósito. Además, es notable el hecho de que el tiempo de simulación ya no es tan pequeño como en los casos anteriores. Si se hubieran realizado las operaciones usando bucles, el tiempo se habría incrementado de manera exponencial. Una vez entendidos los algoritmos más básicos, es necesario aumentar el nivel y adentrarse en las redes neuronales.

4 RED NEURONAL MULTICAPA

4.1 La biología como Fuente

Para comprender las redes neuronales artificiales, primero se debe conocer cómo funcionan las naturales y saber las partes de las que se componen. Una neurona está compuesta por las dendritas, ramificaciones que recogen los impulsos eléctricos y los conducen al cuerpo de la célula, también llamado soma; el cuerpo o soma, que procesa la información recibida y el axón, que conduce la señal proveniente del cuerpo a otras neuronas.

La conexión entre neuronas se realiza mediante la sinapsis. Ésta puede ser más o menos fuerte dependiendo del proceso químico encargado. A continuación, se muestra una imagen de una neurona para facilitar su comprensión:

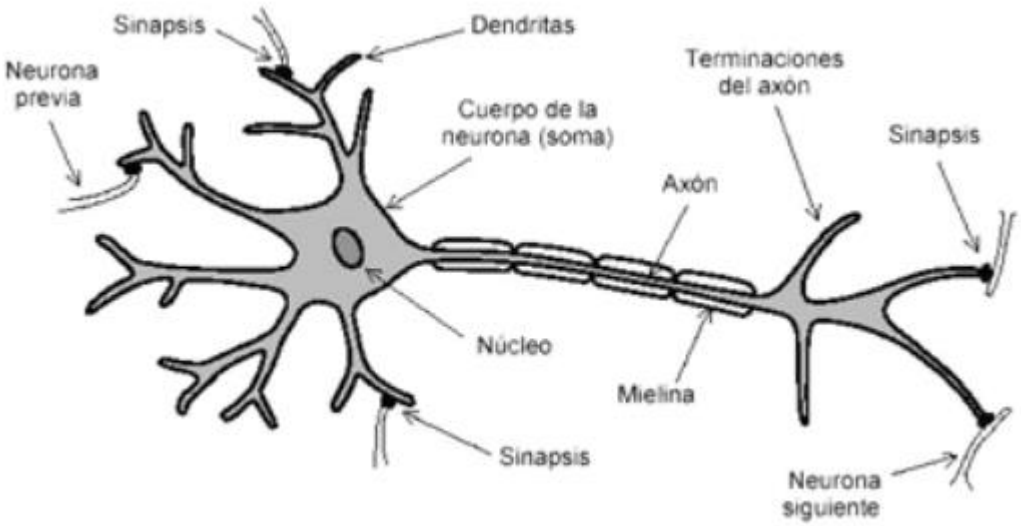

Figura 4-1: Esquema de una neurona

Las Redes Neuronales Artificiales no se aproximan a la complejidad del cerebro, pero tienen los elementos clave que las asimilan a las biológicas para resolver problemas concretos

Aunque los circuitos electrónicos son mucho más rápidos que las neuronas biológicas, la capacidad de estas últimas para trabajar en paralelo hace que sean capaces de resolver cualquier tarea de una manera más rápida que un ordenador. Por ello, las Redes Neuronales Artificiales son ideales para su implementación en VLSI, dispositivos ópticos y procesadores en paralelo.

4.2 La neurona artificial

Si se considera un problema de aprendizaje supervisado en el que se tiene acceso a determinados ejemplos para el entrenamiento $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$, las redes neuronales permiten definir la forma de la función $h_{W,b}(\boldsymbol{x})$, la cual puede ser compleja o incluso no lineal.

Antes de describir redes neuronales, se empezará describiendo la red neuronal más sencilla posible, es decir, aquella que se corresponde con una sola "neurona". El diagrama de una neurona es el siguiente:

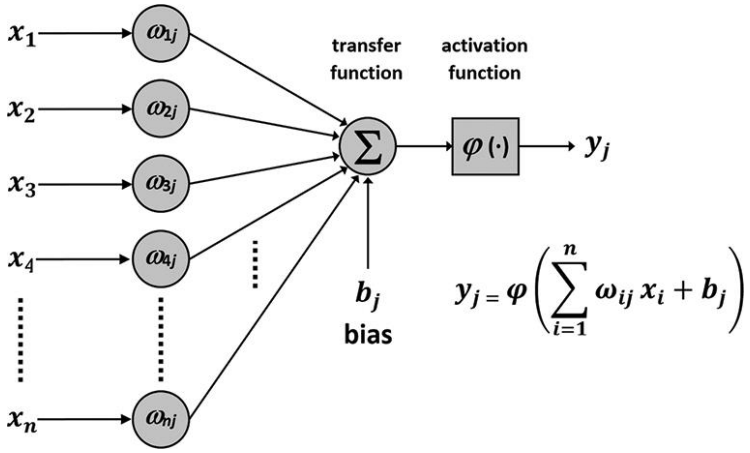

Figura 4-2: Diagrama de una neurona artificial

Esta neurona es una unidad computacional que toma como entradas $x_1, x_2, x_3, x_4, \dots, x_n$ y produce una salida de acuerdo con la ecuación $y = h_{W,b}(\boldsymbol{x}) = f(\boldsymbol{W}^T\boldsymbol{x} + b) = f(\sum_{i=1}^{n} W_i x_i + b)$, donde f (φ en la figura 4-2) es la función de activación y su dominio e imagen son $f: \Re \to \Re$. Se tomará $f(\cdot)$ como la función sigmoide:

$$f(z) = \frac{1}{1 + e^{-z}} \tag{4-1}$$

Por tanto, nuestra neurona simple se corresponde exactamente con el algoritmo de regresión logística visto en el apartado 3.2.

Aunque se va a utilizar la función sigmoide, cabe destacar que otra elección bastante común de f es la tangente hiberbólica o tanh:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{4-2}$$

Además, investigaciones recientes han encontrado una función de activación diferente: la función lineal rectificada (rectified linear function o ReLU en inglés). Esta función trabaja bien en redes neuronales con Deep Learning y es diferente de la sigmoide y la tanh pues no está limitada y no es continuamente diferenciable. Esta función viene dada por:

$$f(z) = \max(0, z) \tag{4-3}$$

A continuación, se muestran en una sola gráfica las tres funciones mencionadas anteriormente para su mejor visualización y comparación:

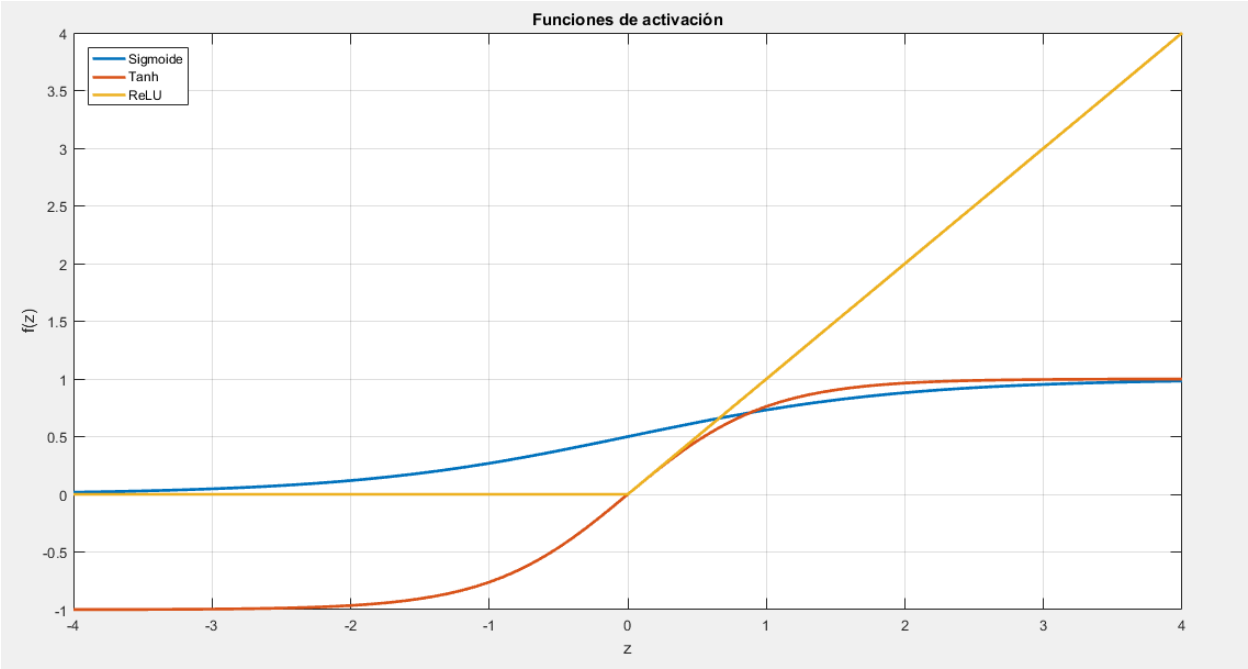

Figura 4-3: Comparación de funciones de activación

La función $\tanh(z)$ es una versión reescalada de la sigmoide, pues el rango de salida es de $[-1,1]$ en vez de ser de $[0,1]$. La función lineal rectificada es una función lineal definida a trozos que satura el valor a 0 cuando la entrada z es menor que dicho valor.

Por último, comentar una igualdad que será de utilidad en el futuro: si $f(z) = 1/(1+e^z)$ es la función sigmoide, entonces su derivada viene dada por $f'(z) = f(z)\big(1-f(z)\big)$. Si, por el contrario, $f(z)$ es la función tanh, entonces su derivada resulta ser $f'(z) = 1-\big(f(z)\big)^2$. La función lineal rectificada tiene un gradiente de 0 cuando $z \leq 0$ y 1 en cualquier otro caso.

4.3 Modelo de Red Neuronal

Una red neuronal se crea interconectando las neuronas simples que se vió en el apartado anterior de manera que la salida de una neurona puede ser la entrada de otra. Por ejemplo, he aquí una pequeña red neuronal:

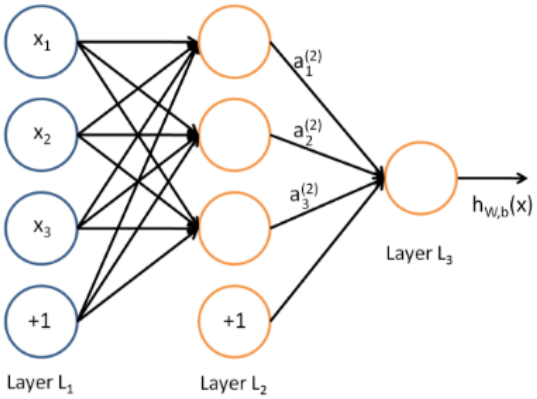

Figura 4-4: Red neuronal con varias neuronas

En la figura anterior, se han usado círculos para denotar también a las entradas de la red. Los círculos con un "+1" en su interior se corresponden con el término de bias. La capa de más a la izquierda se llama capa de entrada y la capa de más a la derecha, capa de salida (que en este caso solo tiene un nodo o neurona). La capa intermedia se llama capa oculta porque sus valores no se observan en el set de entrenamiento. La red de este ejemplo tiene 3 unidades de entrada (sin contar la bias), 3 unidades ocultas y 1 unidad de salida.

A partir de ahora se denotará el número de capas de nuestra red con n_l, por lo que, en el ejemplo anterior $n_l = 3$. A las distintas capas se las llamará L_l, de manera que L_1 se corresponde con la capa de entrada y la capa L_{n_l} es la de salida. Dicha red tiene como parámetros $(\boldsymbol{W}, b) = \left(\boldsymbol{W}^{(1)}, b^{(1)}, \boldsymbol{W}^{(2)}, b^{(2)}\right)$, donde $W_{ij}^{(l)}$ es el parámetro (o peso) asociado a la conexión entre la neurona j de la capa l y la neurona i de la capa $l+1$ y $b_i^{(l)}$ es la bias asociada con la neurona i en la capa $l+1$. Por tanto, se tiene que $\boldsymbol{W}^{(1)} \in \Re^{3\times 3}$ y $\boldsymbol{W}^{(2)} \in \Re^{1\times 3}$.

Se denotará $a_i^{(l)}$ como la activación (el valor de salida) de la i-esima neurona de la capa l. Dado un conjunto de parámetros fijos $\boldsymbol{W}, b$, la función $h_{W,b}(\boldsymbol{x})$ de la red neuronal da como resultado un número real. En concreto, la salida resultante de la red de la figura 4-4 viene dada por:

$$\begin{aligned} a_1^{(2)} &= f\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}\right) \\ a_2^{(2)} &= f\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}\right) \\ a_3^{(2)} &= f\left(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}\right) \\ h_{W,b}(\boldsymbol{x}) = a_1^{(3)} &= f\left(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}\right) \end{aligned} \tag{4-4}$$

Asimismo, se denotará $z_i^{(l)}$ como la suma ponderada de todas las entradas a la i-ésima neurona de la capa l, término de bias incluido. Por ejemplo, $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$, es decir, $a_i^{(l)} = f\left(z_i^{(l)}\right)$. Cabe destacar que esto tiende a una notación aún más compacta si se extiende a forma vectorial, ya que ahora la ecuación (4-4) se puede sobre escribir como:

$$\begin{aligned} \boldsymbol{z}^{(2)} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \\ \boldsymbol{a}^{(2)} &= f\left(\boldsymbol{z}^{(2)}\right) \\ \boldsymbol{z}^{(3)} &= \boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)} \\ h_{W,b}(\boldsymbol{x}) &= \boldsymbol{a}^{(3)} = f\left(\boldsymbol{z}^{(3)}\right) \end{aligned} \tag{4-5}$$

A esto se le llama forward propagation (propagación hacia adelante). Además, se suele usar $\boldsymbol{a}^{(1)} = \boldsymbol{x}$ para denotar a los valores de la capa de entrada. Las activaciones de la capa $l+1$, dadas las activaciones de la capa anterior, se pueden calcular como:

$$\begin{aligned} \boldsymbol{z}^{(l+1)} &= \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)} \\ \boldsymbol{a}^{(l+1)} &= f\left(\boldsymbol{z}^{(l+1)}\right) \end{aligned} \tag{4-6}$$

La ventaja de agrupar los parámetros en matrices y utilizar operaciones vectoriales y matriciales es que los cálculos se ejecutan de manera más rápida gracias al algebra lineal en MATLAB.

Lo expuesto hasta ahora sirve para la red de la figura 4-4, pero también es una generalización que puede ser aplicada a arquitecturas diferentes de redes neuronales, incluyendo aquellas con varias capas ocultas. La opción más común es una red neuronal de n_l capas en la que la capa 1 es la de entrada, la n_l es la de salida y cada una de las l capas está totalmente conectada a la siguiente. Con esta configuración, para calcular la salida de la red, se pueden calcular las activaciones de la capa L_2, luego la de la capa L_3 y así sucesivamente hasta la capa L_{n_l} usando las ecuaciones anteriores que describen la forward propagation. Este es un ejemplo de una red tipo feedforward, ya que no tiene ningún bucle ni ciclo.

Las redes neuronales pueden tener múltiples unidades o neuronas de salida. Por ejemplo, a continuación, se muestra una red con dos capas ocultas L_2 y L_3 y una capa de salida L_4 con dos unidades de salida:

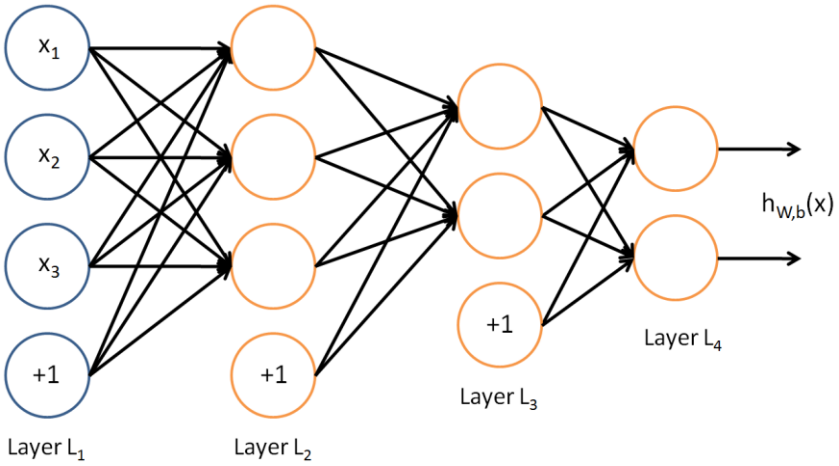

Figura 4-5: Red neuronal multicapa

Para entrenar esta red, se necesitarían ejemplos de entrenamiento $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$ donde $y^{(i)} \in \Re^2$. Este tipo de redes son útiles si existen múltiples salidas que interesen ser predichas. Por ejemplo, en una aplicación de diagnóstico médico, el vector $\boldsymbol{x}$ serviría como las características de entrada de un paciente, y las diferentes salidas y_i podrían indicar la presencia o ausencia de diferentes enfermedades.

4.4 Backpropagation Algorithm

Si ahora se supone un set de entrenamiento $\left\{\left(\boldsymbol{x}^{(1)}, y^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(m)}, y^{(m)}\right)\right\}$ de m ejemplos. Se entrenará a la red neuronal usando el algoritmo de gradient descent (descenso por gradiente). Es más, para un ejemplo concreto $(\boldsymbol{x}, y)$, se define la función coste con respecto a ese ejemplo como:

$$J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \frac{1}{2}\left\|h_{W,b}(\boldsymbol{x}) - y\right\|^2 \tag{4-7}$$

Dado un conjunto de m ejemplos de entrenamiento, se puede definir la función coste total como:

$$\begin{aligned} J(\boldsymbol{W}, \boldsymbol{b}) &= \left[\frac{1}{m}\sum_{i=1}^{m} J\left(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, y^{(i)}\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 = \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 \end{aligned} \tag{4-8}$$

El primer término de la ecuación (4-8) es la media aritmética del error cuadrático medio y el segundo es un término para la regularización (también llamado weight decay o disminución de los pesos) que tiende a disminuir la magnitud de los pesos y ayuda a prevenir el sobreajuste. El parámetro λ del weight decay controla la importancia relativa de los dos términos.

La función coste es usada tanto para problemas de clasificación como de regresión. Con respecto a la clasificación, los valores de $\boldsymbol{y}$ serán 0 ó 1, es decir, alguna de las dos clases. Cabe destacar que esto es así si se supone el uso de la función sigmoide, ya que, si se usara la función tanh, los valores de $\boldsymbol{y}$ deberían ser -1 ó 1, para que coincidiera con la salida de dicha función. En cuanto a regresión, se escalan los valores de $\boldsymbol{y}$ de manera que todos ellos estén comprendidos en el intervalo $[0,1]$ o en el $[-1,1]$ si se usa la función tanh.

El objetivo actual es minimizar la función (4-8) como una función de $\boldsymbol{W}$ y $\boldsymbol{b}$. Para entrenar una red neuronal, primero se inicializará cada uno de los parámetros $W_{ij}^{(l)}$ y $b_i^{(l)}$ con valores pequeños cercanos a cero de manera aleatoria (por ejemplo, siguiendo una distribución normal de media nula y varianza ϵ^2, donde ϵ es del orden de 0.01) y se aplicará un algoritmo de optimización como el gradient descent. Dado que $J(\boldsymbol{W}, \boldsymbol{b})$ es una función no convexa, el método del gradient descent es susceptible de avanzar hacia un mínimo relativo y no absoluto; sin embargo, en la práctica dicho método funciona con mucha fiabilidad. Por último, cabe destacar la importancia de inicializar los parámetros de forma aleatoria en vez de darles un valor nulo, ya que, si todos empiezan con el mismo valor, entonces todas las neuronas de las capas ocultas acabarán aprendiendo la misma función de la entrada, o lo que es lo mismo, $W_{ij}^{(l)}$ será idéntico para todos los valores de i de manera que $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \cdots \ \forall \boldsymbol{x}$. La inicialización aleatoria tiene como propósito romper la simetría.

Cada una de las iteraciones del gradient descent actualiza los parámetros $\boldsymbol{W}, \boldsymbol{b}$ tal que:

$$\begin{aligned} W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) \\ b_i^{(l)} &= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) \end{aligned} \tag{4-9}$$

En la ecuación anterior α es el ratio de aprendizaje. El siguiente paso es calcular las derivadas parciales que aparecen. Para ello, se va a proceder a explicar el algoritmo de retropropagacion (backpropagation algorithm), ya que proporciona una forma eficiente de realizar dicho cálculo.

En primer lugar, se describirá como el algoritmo de retropropagación puede ser usado para calcular las derivadas parciales $\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, b)$ y $\frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$ de la función coste para un solo ejemplo $(\boldsymbol{x}, y)$. Con estas calculadas, las parciales de la función coste total vienen dadas por:

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) &= \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J\left(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, y^{(i)}\right) \right] + \lambda W_{ij}^{(l)} \\ \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J\left(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, y^{(i)}\right) \end{aligned} \tag{4-10}$$

Las dos ecuaciones anteriores difieren ligeramente pues los pesos tienen un weight decay aplicado y, por el contrario, el término de bias no.

Dado un ejemplo del set de entrenamiento $(\boldsymbol{x}, y)$, lo primero a realizar es calcular todas las activaciones a lo largo de la red, incluído el valor de salida de la función $h_{W,b}(\boldsymbol{x})$. Seguidamente, para cada nodo i de la capa l, sería conveniente calcular un "término de error" $\delta_i^{(l)}$ que mida cuan responsable es dicho nodo del error cometido a la salida de la red. Para los nodos en la capa de salida, se puede medir directamente la diferencia entre el resultado y el valor real o target y definir dicha diferencia como $\delta_i^{(n_l)}$, donde n_l es la capa de salida. Para las neuronas de las capas ocultas se calculará $\delta_i^{(l)}$según una media ponderada de los términos de error de los nodos que toman $a_i^{(l)}$ como entrada.

A continuación, se muestra en detalle el algoritmo de retropropagación:

1. Calcular las activaciones de las n_l capas de la red utilizando el método de feedforward visto en el apartado 4.3, concretamente en la ecuación (4-4)
2. Para la i-ésima unidad en la capa de salida, establecer:

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \left(\frac{1}{2} \left\| y - h_{W,b}(\boldsymbol{x}) \right\|^2 \right) = - \left(y_i - a_i^{(n_l)} \right) \cdot f' \left(z_i^{(n_l)} \right) \tag{4-11}$$

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

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) \cdot f' \left(z_i^{(l)} \right) \tag{4-12}$$

4. Calcular las derivadas parciales deseadas, que vienen dadas por:

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) &= \delta_i^{(l+1)} \end{aligned} \tag{4-13}$$

Además, el algoritmo antes descrito se puede reescribir en forma matricial con el fin de acelerar el tiempo en el que se realizan los cálculos. Con este fin, se utilizará el operador producto elemento a elemento (también llamado producto de Hadamard) y se denotará como "⊙". El algoritmo queda ahora como:

1. Calcular las activaciones de las n_l capas de la red utilizando el método de feedforward visto en el apartado 4.3, concretamente en la ecuación (4-6)
2. Para la capa de salida, establecer:

$$\boldsymbol{\delta}^{(n_l)} = -\left(\boldsymbol{y} - \boldsymbol{a}^{(n_l)} \right) \odot f' \left(\boldsymbol{z}^{(n_l)} \right) \tag{4-14}$$

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

$$\boldsymbol{\delta}^{(l)} = \left(\left(\boldsymbol{W}^{(l)}\right)^T \boldsymbol{\delta}^{(l+1)}\right) \odot f'\left(\boldsymbol{z}^{(l)}\right) \tag{4-15}$$

4. Calcular las derivadas parciales deseadas, que vienen dadas por:

$$\begin{aligned} \nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) &= \boldsymbol{\delta}^{(l+1)}\left(\boldsymbol{a}^{(l)}\right)^T \\ \nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) &= \boldsymbol{\delta}^{(l+1)} \end{aligned} \tag{4-16}$$

Por último, el algoritmo completo para calcular el gradient descent expresado en pseudocódigo es el siguiente:

1. Establecer $\Delta \boldsymbol{W}^{(l)} \coloneqq \boldsymbol{0}, \Delta \boldsymbol{b}^{(l)} \coloneqq \boldsymbol{0}$ (matriz/vector de ceros) para todo l
2. Desde $i = 1$ hasta m,
   2.1 Usar la retropropagación para calcular $\nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$ y $\nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$
   2.2 Establecer $\Delta \boldsymbol{W}^{(l)} \coloneqq \Delta \boldsymbol{W}^{(l)} + \nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$
   2.3 Establecer $\Delta \boldsymbol{b}^{(l)} \coloneqq \Delta \boldsymbol{b}^{(l)} + \nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$
3. Actualizar los parámetros:

$$\begin{aligned} \boldsymbol{W}^{(l)} &= \boldsymbol{W}^{(l)} - \alpha\left[\left(\frac{1}{m}\Delta \boldsymbol{W}^{(l)}\right) + \lambda \boldsymbol{W}^{(l)}\right] \\ \boldsymbol{b}^{(l)} &= \boldsymbol{b}^{(l)} - \alpha\left[\frac{1}{m}\Delta \boldsymbol{b}^{(l)}\right] \end{aligned} \tag{4-17}$$

Para entrenar a la red neuronal, se puede realizar el método del gradient descent repetidamente para reducir la función coste $J(\boldsymbol{W}, \boldsymbol{b})$.

4.5 Resultados experimentales

A continuación, se va a implementar red neuronal en MATLAB usando como ejemplo los códigos dentro de la carpeta *multilayer_supervised/*. Los códigos completos se encuentran en apéndices (se especifica en cual al hablar en particular sobre alguno de los códigos en los siguientes párrafos), sin embargo, se procederá a explicar dichos códigos grosso modo y se insertarán aquellas partes del mismo que sean más interesantes, así como de los resultados obtenidos.

En este ejemplo, se va a entrenar una red neuronal capaz de clasificar los 10 dígitos del set de datos MNIST. La salida de la red es idéntica a la regresión softmax vista en el apartado 3.3. Sin embargo, la regresión softmax por sí sola no ajustaba correctamente los parámetros ya que se producía el fenómeno de underfitting (subajuste). Por el contrario, una red neuronal debería ser capaz de ajustarse mejor.

La función coste a la salida es prácticamente idéntica a la de la regresión softmax, solo que en vez de realizar las predicciones en función de los datos de entrada $\boldsymbol{x}$, lo hace según el valor de $h_{W,b}(\boldsymbol{x})$ de acuerdo a la ecuación (4-4). La función coste queda, por tanto:

$$J(\boldsymbol{\theta}) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)} = k\} \log \frac{e^{\boldsymbol{\theta}^{(k)T} h_{W,b}(\boldsymbol{x}^{(i)})}}{\sum_{j=1}^{K} e^{\boldsymbol{\theta}^{(j)T} h_{W,b}(\boldsymbol{x}^{(i)})}}\right] \tag{4-18}$$

Esta diferencia en la función coste afecta directamente al término de error de la capa de salida $\left(\boldsymbol{\delta}^{(n_l)}\right)$, que ahora viene dado por:

$$\boldsymbol{\delta}^{(n_l)} = -\sum_{i=1}^{m} \left(1\{y^{(i)} = k\} - P\left(y^{(i)} = k \middle| \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right)\right) \tag{4-19}$$

A continuación, se va a explicar el funcionamiento de los diferentes códigos necesarios para el correcto funcionamiento de la red, en concreto se comenzará por el *run_train* (completo en el *Apéndice IV – Código 1*) ya que es el código principal. En él, se añade al directorio la carpeta que contiene la función *minFunc* y se cargan los datos del set MNIST mediante la función *load_preprocess_mnist* (contenida en el *Apéndice IV – Código 2*). Dicha función solamente carga los archivos tipo ubyte y para poder usar el vector $\boldsymbol{y}$ como índice en el futuro al igual que en la regresión softmax, se le suma un 1 a dicho vector para evitar la clase 0.

Tras esto se configura la capa de entrada con un tamaño de 784, es decir, los 28×28 píxeles de las imágenes, la capa de salida con un tamaño de 10, para la probabilidad de pertenecer a las 10 clases del MNIST y una capa oculta de 256, la cual da una cantidad de parámetros lo suficientemente amplia como para poder clasificar la imagen de forma correcta. Además, se escoge como función la sigmoide y se establece λ con un valor nulo para el término del weight decay.

Una vez definidos los tamaños de las diferentes capas, es necesario inicializar los pesos y las bias de cada capa. Para ello, se usa la función *initialize_weights* (contenida en el *Apéndice IV – Código 3*), la cual utiliza el método de Xavier para inicializar los parámetros de forma aleatoria y óptima, ya que proporciona unos pesos iniciales que no son ni demasiado pequeños para que se vuelvan inútiles con el tiempo ni demasiado grandes para acabar saturando la red.

Aunque los parámetros están expresados como pesos y bias, el método de optimización necesita que todos estén recogidos en un solo vector. Por ello, se usa la función *stack2params* (completa en el *Apéndice IV – Código 4*) para almacenar en un vector de forma consecutiva todos los pesos y bias de cada una de las capas. Existe también la función recíproca *params2stack* (contenida en el *Apéndice IV – Código 5*) que realiza la operación contraria.

Tras esto, se elige el método de entrenamiento y ejecuta la función *minFunc*. Aunque no se va a explicar ésta, si se explicará la función *supervised_dnn_cost* (contenida en el *Apéndice IV – Código 6*) que es la encargada de proporcionar el coste, el gradiente y las diferentes probabilidades de pertenecer a una clase. Antes de explicarla en profundidad, comentar que el resto del código es simplemente ejecutar dicha función para obtener las probabilidades y ver si se ha clasificado correctamente la imagen. Se imprimirá por pantalla la precisión, calculada como la media de los aciertos, al clasificar los sets de entrenamiento y prueba.

Una vez se ha entrado en esta última función, toma los parámetros de entrada para identificar el número de capas, el número de muestras, crea tantas celdas como capas haya y utiliza la función *params2stack* para devolver el vector de parámetros a la forma de pesos y bias.

Tras esto, realiza el código de propagación hacia adelante o forward propagation, almacenando en cada una de las activaciones en una variable tipo celda. Para esto, se ha seguido la ecuación (4-6) y escrito en código queda como:

```
for l = 1 : numHidden
    if l > 1
        hAct{l} = stack{l}.W * hAct{l - 1} + repmat(stack{l}.b, 1,numSamples);
    else
        hAct{l} = stack{l}.W * data + repmat(stack{l}.b, 1, numSamples);
    end
            hAct{l} = sigmoid(hAct{l});
end
```

A continuación, se calcula la activación de la capa de salida y se calcula la ecuación (4-18) de la misma manera que se hizo en el apartado 3.3.2 teniendo en cuenta que ahora no existe una clase 0 (porque se ha aumentado la etiqueta de la clase en una unidad), por lo que no es necesario realizar las operaciones para obtener una probabilidad nula en la clase 0. Antes de calcular el coste, calcula las probabilidades de pertenecer a cada clase y obtiene el máximo para conocer la clase final predicha. Si el flag *po* está activo, se vuelve a la función principal (esto sirve en el caso de que la red esté entrenada y se quiera conocer la salida producida).

```
l = numHidden+1;
y_hat = stack{l}.W * hAct{l - 1} + repmat(stack{l}.b, 1, numSamples);
y_hat = exp(y_hat);
hAct{l} = bsxfun(@rdivide, y_hat, sum(y_hat, 1));
[pred_prob pred_labels] = max(hAct{l});
pred_prob = hAct{l};

%% return here if only predictions desired.
if po
    cost = -1; ceCost = -1; wCost = -1; numCorrect = -1;
    grad = [];
    return;
end;

%% compute cost
y_hat = log(hAct{numHidden+1});
index = sub2ind(size(y_hat), labels', 1:numSamples);
ceCost = -sum(y_hat(index))/(size(data,2));
```

Una vez calculado el coste, solo resta calcular el gradiente mediante el método de la retropropagación descrito en el apartado anterior. Las ecuaciones (4-14), (4-15) y (4-16) implementadas en MATLAB quedan como

```
for l = numHidden+1 : -1 : 1
    if l > numHidden
        gradFunc = ones(size(gradInput));
    else
        gradFunc = hAct{l} .* (1 - hAct{l});
    end
    gradOutput = gradInput .* gradFunc;
    if l > 1
        gradStack{l}.W = gradOutput * hAct{l-1}'./(size(data,2));
    else
        gradStack{l}.W = gradOutput * data'./(size(data,2));
    end
    gradStack{l}.b = sum(gradOutput, 2)/(size(data,2));
    gradInput = stack{l}.W' * gradOutput;
end
```

El último paso es añadir el término que depende de λ a la ecuación del coste y al gradiente de los pesos en cada una de las capas. Tras esto, se pasa el gradiente a forma vectorial y la función *minFunc* es la que se encarga de actualizar los pesos gracias al coste y el gradiente proporcionados.

```
%% compute weight penalty cost and gradient for non-bias terms
wCost = 0;
for l = 1:numHidden+1
    wCost = wCost + .5 * ei.lambda * sum(stack{l}.W(:) .^ 2);
end

cost = ceCost + wCost;

% Computing the gradient of the weight decay.
for l = numHidden : -1 : 1
    gradStack{l}.W = gradStack{l}.W + ei.lambda * stack{l}.W;
end
```

Una vez explicado el código, es el momento de la ejecución para poder comprobar los resultados obtenidos mediante la implementación de esta red neuronal:

```
test accuracy: 97.080000
train accuracy: 100.000000
Elapsed time is 260.645647 seconds.
```

Figura 4-6: Resultados obtenidos con Red Neuronal Multicapa

Como se puede comprobar el resultado actual está muy por encima del obtenido mediante el uso de la regresión softmax de forma aislada. La precisión obtenida al probar el set de entrenamiento es perfecta y la del de prueba es del 97.08%, la cual es bastante alta del mismo modo. El único punto negativo es que ahora el código tarda bastante más en ejecutarse en comparación, sin embargo, los resultados obtenidos hacen que valga la pena el uso de esta red y el consecuente tiempo de espera.

5 REDES NEURONALES CONVOLUCIONALES

5.1 Extracción de características usando la convolución

Hasta ahora, a lo largo del presente proyecto se ha trabajado con imágenes contenidas en el set de datos MNIST, las cuales tienen una resolución bastante baja. Esto no es lo común a la hora de usar redes neuronales para clasificar imágenes en una clase, ya que la resolución actual de las imágenes asciende como mínimo al HD (1280×720 píxeles). Por tanto, es necesario realizar un cambio de paradigma para desarrollar métodos que permitan el uso de set de datos más realistas, cuyas imágenes sean de resoluciones superiores.

Además de lo anterior, existe un problema añadido: en el diseño anterior se decidió conectar todas y cada una de las unidades de la capa oculta a todas las unidades de entrada. Al trabajar con imágenes cuya resolución es relativamente baja, resulta computacionalmente factible aprender características usando la imagen completa, sin embargo, aprender características que abarquen la imagen en su totalidad usando resoluciones mayores (96×96, por ejemplo) es muy costoso hablando en términos de cálculo computacional. Con la resolución de ejemplo anterior, se tendrían alrededor de 10^4 unidades de entrada y, asumiendo que se quisieran aprender 100 características, el número de parámetros oscila en 10^6. Aparte de esto, la propagación, tanto hacia adelante como hacia atrás, resultaría ser 10^2 veces más lenta comparada con imágenes de 28×28.

Una solución bastante simple al problema descrito es restringir las conexiones entre las unidades de la capa oculta y las de entrada, de manera que cada unidad de entrada queda conectada solamente a un pequeño subconjunto de las unidades de entrada. Concretamente, cada unidad de la capa oculta se conectará a una región continua de pixeles de la entrada. La idea de tener redes locales también se inspira en cómo funciona el sistema visual biológico, ya que las neuronas del córtex visual poseen campos receptivos que responden solo ante estímulos localizados en una región específica.

Las imágenes tienen la propiedad natural de ser "estacionarias", es decir, que los datos estadísticos de una parte de la imagen son los mismos que en cualquier otra parte. Esto sugiere que las características que se aprendan en una parte de la imagen pueden ser aplicadas a otras partes de dicha imagen o incluso en todas las partes de ella.

De forma más específica, tras haber aprendido características sobre pequeñas áreas (de 8×8, por ejemplo) llamadas filtros o kernels, se puede aplicar este detector de características en cualquier lugar de la imagen. Por tanto, se puede tomar este kernel y realizar la convolución con la imagen para obtener diferentes valores de activación en cada región de la imagen.

A continuación, se ofrece una imagen visual para comprender mejor la convolución:

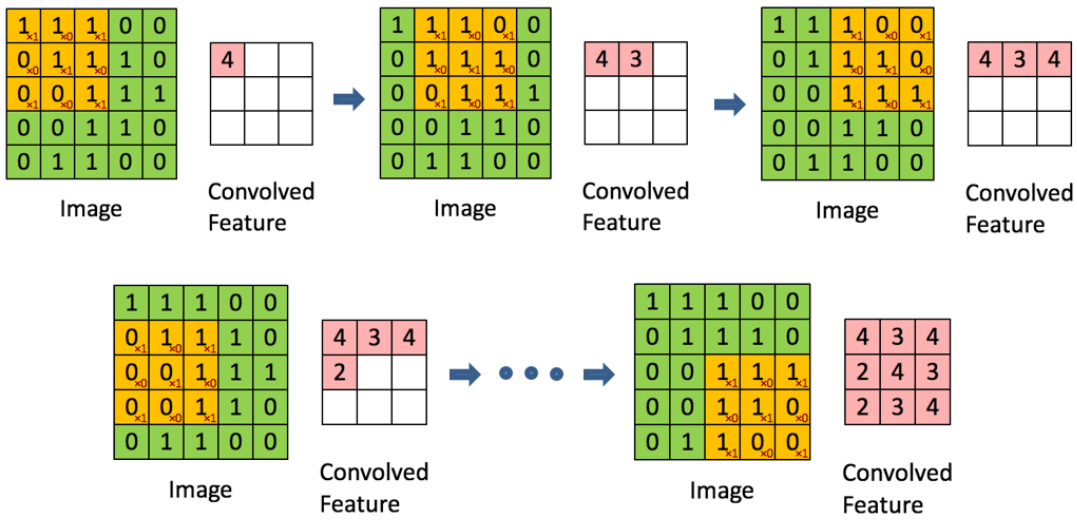

Figura 5-1: Convolución

Matemáticamente, dada una imagen grande $\boldsymbol{x_l}$ de dimensiones $r \times c$ y k filtros $\boldsymbol{x_s}$ de dimensiones $a \times b$, entonces se generarán k mapas de rasgos que en total forman un conjunto de $(r - a + 1) \times (c - b + 1) \times k$. Si se expresa cada uno de lo filtros en forma de $\boldsymbol{W}$ y $\boldsymbol{b}$, para cada subconjunto $\boldsymbol{x_{sample}}$ de $a \times b$ de la imagen $\boldsymbol{x_l}$, se calcula $f_{sample} = \sigma\left(\boldsymbol{W}^{(1)}\boldsymbol{x_{sample}} + \boldsymbol{b}^{(1)}\right)$.

Para implementar la convolución en MATLAB es necesario calcular $\sigma\left(\boldsymbol{W}\boldsymbol{x}_{(\boldsymbol{r},\boldsymbol{c})} + \boldsymbol{b}\right)$ para todos los $(\boldsymbol{r}, \boldsymbol{c})$ válidos. Válido (valid) significa que el kernel está completamente contenido en la imagen; lo cual es lo contrario a completo (full), donde se puede extender el kernel al exterior de la imagen asumiendo estos valores como nulos.

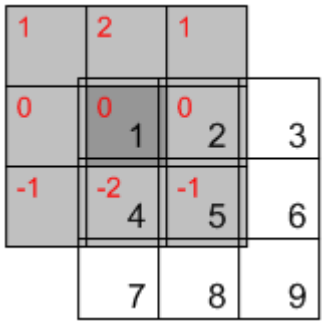

Figura 5-2: Convolución completa

En la ecuación anterior $\boldsymbol{W}$ y $\boldsymbol{b}$ son los pesos y las bias aprendidos que conectan la capa de entrada con la capa oculta y $\boldsymbol{x}_{(\boldsymbol{r},\boldsymbol{c})}$ es el subconjunto de $a \times b$ que empieza en la esquina superior izquierda de la imagen. Para realizar dicha ecuación, un primer método sería realizar un bucle que recorra todos los subconjuntos y calcule $\sigma\left(\boldsymbol{W}\boldsymbol{x}_{(\boldsymbol{r},\boldsymbol{c})} + \boldsymbol{b}\right)$, sin embargo, resulta que en la práctica este método es válido pero muy lento.

Un segundo método es calcular $\boldsymbol{W}\boldsymbol{x}_{(\boldsymbol{r},\boldsymbol{c})}$ para todo $(\boldsymbol{r}, \boldsymbol{c})$, sumar $\boldsymbol{b}$ a los valores calculados y aplicar la función sigmoide al resultado obtenido. Este método no parece significativamente mejor que el anterior pues es necesario un bucle para calcular el primer término, sin embargo, dicho bucle se puede reemplazar por una función propia de MATLAB optimizada para la convolución: *conv2*, acelerando bastante el proceso.

A la hora de usar la función anterior es necesario tener tos aspectos muy en cuenta. El primero de ellos es que la convolución a realizar es en $2D$, pero en el código se manejarán 4 dimensiones: número de imágenes, número de filtros, filas de la imagen y columnas de la imagen. Por ello, es necesario realizar la convolución de forma independiente para cada filtro aplicado a cada una de las imágenes, usando las filas y columnas de la imagen pertinente como las dos dimensiones sobre las que se va a convolucionar. Esto significa que hay que usar dos bucles: uno para el número de imágenes y otro para el número de filtros. Dentro de los dos bucles se realizará la convolución entre la i-ésima imagen y el j-ésimo filtro.

El segundo de los aspectos es que, por la definición matemática de la convolución, el filtro es rotado 180º antes de ejecutar ejecutar la convolución. Esto lo hace MATLAB intrínsecamente en la función, por lo que para usar el filtro deseado es necesario rotarlo antes de pasárselo a la función *conv2*.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \rightarrow \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} \tag{5-1}$$

A continuación, se muestra el código necesario para hacer la convolución de los k filtros para cada una de las imágenes:

```
for imageNum = 1:numImages
  for filterNum = 1:numFilters

    % convolution of image with feature matrix
    convolvedImage = zeros(convDim, convDim);

    % Obtain the feature (filterDim x filterDim) needed during the convolution
    filter=W(:,:,filterNum);

    % Flip the feature matrix because of the definition of convolution, as explained later
    filter = rot90(filter,2);

    % Obtain the image
    im = images(:, :, imageNum);

    % Convolve "filter" with "im", adding the result to convolvedImage
    convolvedImage=conv2(im,filter,'valid');

    % Add the bias unit
    % Then, apply the sigmoid function to get the hidden activation
    convolvedImage=convolvedImage+b(filterNum);
    convolvedImage=logsig(convolvedImage);

    convolvedFeatures(:, :, filterNum, imageNum) = convolvedImage;
  end
end
```

5.2 Pooling

Tras extraer las características usando la convolución, el siguiente paso sería utilizarlas para la clasificación. En teoría, se podrían usar todas las características extraídas con un clasificador tipo softmax, pero esto resultaría todo un reto computacionalmente hablando. Si se considera, por ejemplo, imágenes de 96×96 (no es una alta resolución) y 400 filtros de 8×8, a la salida de la capa convolucional se tendría un vector (tras convertir de matriz a vector) de dimensión $(96-8+1)^2\cdot400=3168400$. Aprender a clasificar con entradas de mas de 3 millones de elementos puede ser inviable e incluso propenso al sobreajuste u over-fitting.

Para manejar esta situación, primero se decidió obtener características mediante convolución pues las imágenes tienen propiedades "estacionarias" que permiten usar características útiles en una parte de la imagen en cualquier otra parte. Por tanto, para describir una imagen grande, resulta natural analizar en que zonas de la imagen éstas características están más presentes. Por ejemplo, se puede calcular la media (o el máximo) valor de una característica a lo largo de una región de la imagen. La salida que se obtiene es mucho menor en cuanto a dimensión se refiere (comparada con los de la salida de la capa de convolución) y pueden mejorar los resultados (menos sobreajuste). La operación que hace esto posible se llama "pooling" e incluso a veces "mean pooling" o "max pooling" (dependiendo de la operación usada).

La siguiente imagen muestra cómo se realiza la operación de pooling a lo largo de una imagen cualquiera:

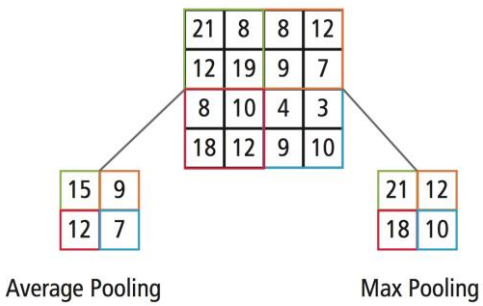

Figura 5-3: Pooling

De una manera más formal, tras obtener las características resultantes de la convolución tal y como se describió en el apartado 5.1, se decide realizar la operación de pooling sobre regiones de dimensiones $m \times n$. Por tanto, se divide la salida de la capa de convolución en regiones disjuntas de $m \times n$ y se toma la media o el máximo, permitiendo así obtener unas nuevas características que ya sí se pueden usar para la clasificación.

Para realizar la operación pooling en MATLAB se puede utilizar de nuevo la función *conv2*, cuyos argumentos ahora serían la salida de la capa convolucional y un filtro que sea una matriz de unos. Una vez realizado este paso, se toman de la matriz los elementos correspondientes y se les hace la media. Cabe destacar que la convolución anterior debe ser *valid*. En código quedaría:

```
W=ones(poolDim,poolDim);

for imageNum=1:numImages
    for filterNum=1:numFilters

        filter=W;
        pooledImage=zeros(convolvedDim / poolDim, convolvedDim / poolDim);

        im=convolvedFeatures(:,:,filterNum,imageNum);
        convolvedImage=conv2(im,filter,'valid');
        pooledImage=convolvedImage(1:poolDim:end, 1:poolDim:end)./(poolDim^2);

        pooledFeatures(:,:,filterNum,imageNum)=pooledImage;
    end
end
```

5.3 Optimización: Stochastic Gradient Descent

Algunos métodos de optimización como el LBFGS (el que se usó en los apartados anteriores) usan el set de entrenamiento completo para actualizar los parámetros en cada iteración y tiende a converger de manera eficiente hacia un mínimo absoluto. Sin embargo, en la práctica el cálculo del coste y del gradiente para todo el set de entrenamiento puede resultar muy lento e incluso imposible para una sola máquina si el set de datos es demasiado grande para poder guardarse en memoria. Además, otro problema con estos métodos de optimización es que no hay una forma fácil de incorporar nuevos datos "en línea". El método del Stochastic Gradient Descent (Descenso por Gradiente Estocástico) o SGD resuelve estos dos problemas siguiendo la dirección negativa del gradiente de la función objetivo tras haber visto solo uno o unos pocos ejemplos de entrenamiento. El alto coste de retropropagar el set completo de entrenamiento hace necesario el uso del SGD.

El algoritmo estándar del gradient descent actualiza los parámetros $\boldsymbol{\theta}$ de forma:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla_{\theta} E[J(\boldsymbol{\theta})] \tag{5-2}$$

En la ecuación anterior la esperanza se aproxima por el gradiente de la función coste usando el set ompleto de entrenamiento. El método del SGD simplemente elimina dicha esperanza y calcula el gradiente usando solo uno o unos pocos ejemplos de entrenamiento. La nueva actualización viene ahora dada por.

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla_{\theta} J\left(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}\right) \tag{5-3}$$

El par $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$ es un par del set de entrenamiento.

Generalmente, cada actualización en el SGD se calcula con vista a unos pocos ejemplos del set de entrenamiento (llamados minibatch) en lugar de solo a un ejemplo. La razón tras esto es doble: en primer lugar, se reduce la varianza en la actualización de los parámetros y esto permite una convergencia más estable, y, en segundo lugar, permite el uso de álgebra lineal para el eficiente cálculo del coste y del gradiente. El tamaño típico de un minibatch es de 256, aunque el tamaño óptimo puede variar según la aplicación y la arquitectura.

En el SGD el ratio de aprendizaje α normalmente es mucho más pequeño que el usado en el método del gradient descent pues la varianza en la actualización es menor. Escoger el valor adecuado para el ratio de aprendizaje y sus actualizaciones a lo largo del entrenamiento puede ser realmente complicado. Un método estándar que funciona bien en la práctica es usar un ratio pequeño y constante que proporcione una convergencia estable en la epoch (uso completo del set de entrenamiento mediante minibatches) inicial y reducir a la mitad su valor a medida que la convergencia se ralentiza. Otro método bastante utilizado es actualizar el ratio de aprendizaje en cada iteración t como $a/(b+t)$, donde a es el ratio inicial y b la iteración en la que comienta dicha actualización. Los métodos más sofisticados incluyen el uso de backtracking (vuelta atrás) a lo largo de una línea para encontrar la actualización óptima.

Por último, pero no menos importante, en el SGD importa el orden en el que los datos son presentados al algoritmo. Si los datos son presentados en un orden cualquiera, ésto puede introducir un offset en el gradiente y conducir a una convergencia más pobre. Un buen método para evitar esto es, por lo general, "barajar" de forma aleatoria los datos antes de cada epoch de entrenamiento.

Si la función coste tiene la forma análoga a un largo desfiladero poco profundo que conduce al mínimo y con paredes empinadas a los lados, el SGD tenderá a oscilar a lo ancho del estrecho desfiladero, pues la dirección negativa al gradiente apuntará a alguna de las paredes en lugar de al mínimo. Esta forma, que puede parecer muy poco probable, es bastante común en arquitecturas más complejas y, por tanto, la convergencia puede llegar a ser muy lenta. El método Momentum es más óptimo en este caso pues apunta al mínimo más rápido a lo largo del desfiladero. La actualización de los parámetros con este método es:

$$\begin{aligned} \boldsymbol{v} &= \gamma \boldsymbol{v} - \alpha \nabla_{\theta} J\left(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}\right) \\ \boldsymbol{\theta} &= \boldsymbol{\theta} - \boldsymbol{v} \end{aligned} \tag{5-4}$$

En la ecuación anterior $\boldsymbol{v}$ es el vector velocidad actual, cuyas dimensiones coinciden con las de $\boldsymbol{\theta}$. El ratio de aprendizaje α se toma como se describió previamente, aunque, al usar este método, α tal vez deba ser más pequeño dado que la magnitud del gradiente será mayor. Por último, $\gamma \in (0,1]$ determina por cuantas iteraciones se incorporarán los gradientes anteriores. Generalmente, γ se establece a 0.5 hasta que el aprendizaje se estabiliza, momento en el que se aumenta a 0.9.

5.4 Red Neuronal Convolucional

Una Red Neuronal Convolucional o Convolutional Neural Network (CNN) es una red compuesta por una o más capas convolucionales (seguidas normalmente de una capa de pooling) y con una o varias capas fully connected (totalmente conectadas, como las del apartado 4) tras éstas. La arquitectura de una CNN está diseñada para aprovechar la entrada como una sola estructura 2D. Esto se consigue con conexiones locales y pesos asociados a ellas, seguidos por alguna operación de pooling cuyas salidas son características invariantes a las traslaciones. Otra ventaja de estas redes es que son más fáciles de entrenar y pueden tener incluso menos parámetros que las redes del apartado 4 con el mismo número de redes ocultas.

La entrada de una capa convolucional es una imagen de dimensión $m \times m \times r$, donde m es el alto y el ancho de la imagen y r es el número de canales, el cual valdrá 3 en una imagen RGB. La capa convolucional contará con k filtros (o kernels) de dimensión $n \times n \times q$, donde n es más pequeña que la dimensión de la imagen y q puede ser del mismo valor que el número de canales r o menor y puede variar según el kernel. A la salida de esta capa se tendrán k mapas de características (feature maps) de dimensión $m - n + 1$ tal y como se vió en el apartado 5.1. Posteriormente, se submuestrea dicha salida mediante alguna operación de pooling a lo largo de regiones continuas de dimensión $p \times p$, donde p vale normalmente 2 para imágenes pequeñas como las del

MNIST y no más de 5 por lo general para imágenes grandes. Además, antes o después de la capa de pooling se añade un término de bias y también una función de activación como la sigmoide a cada uno de los mapas de características. Tras estas capas, puede haber cualquier número de capas fully connected, las cuales son idénticas a las vistas en el apartado 4. A continuación, se muestra la salida tras aplicar a una imagen una capa convolucional y otra de pooling. Las unidades del mismo color tienen los pesos asociados:

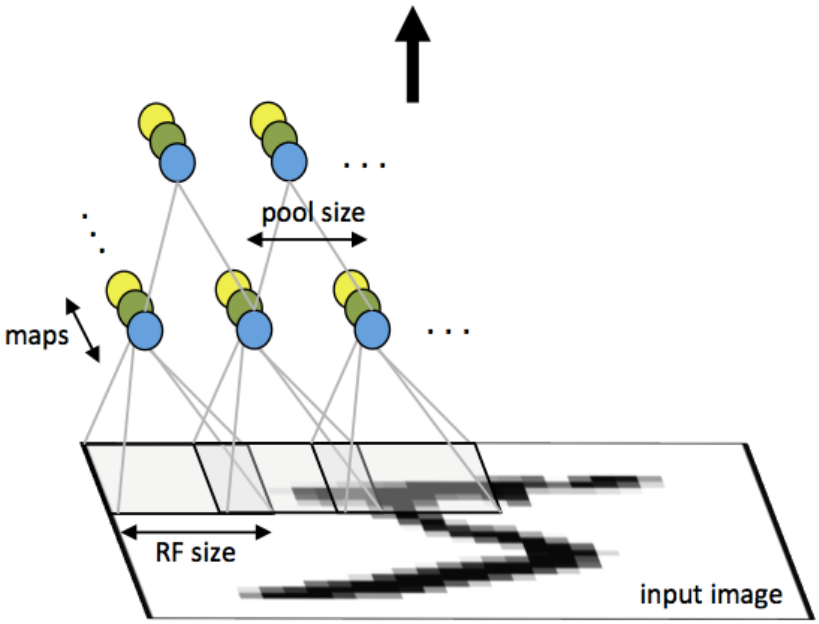

Figura 5-4: Capa convolucional y de pooling aplicadas a una imagen

Se va a ilustrar, también, una de las configuraciones de CNN más comunes:

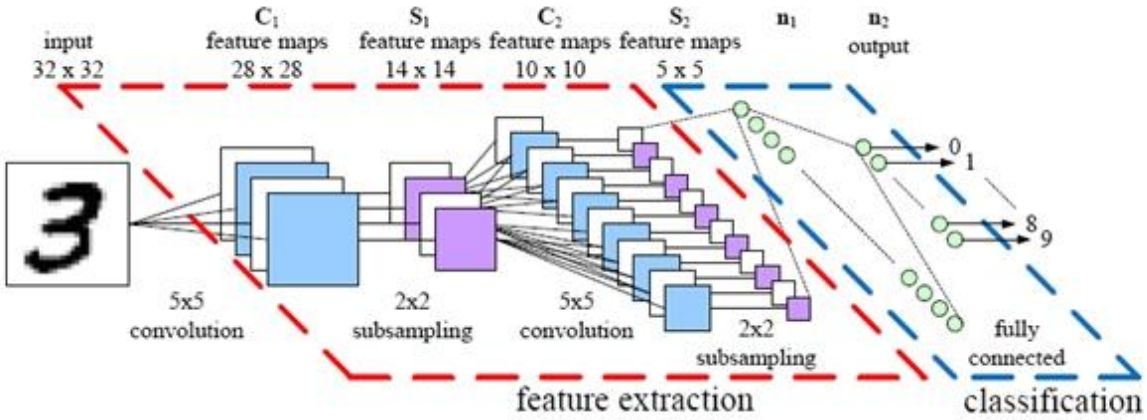

Figura 5-5: Red Neuronal Convolucional

Una vez se ha entendido como funciona una CNN y el método que se va a emplear para entrenarla, solo resta calcular los gradientes de la ecuación (5-4), para lo que se volverá a usar el algoritmo de retropropagación, el cual permite calcular dichos gradientes de una forma rápida y bastante simple.

Si se supone que $\boldsymbol{\delta}^{(l+1)}$ es el término de error de la capa $(l+1)$ de la red con una función coste $J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$, donde $(\boldsymbol{W}, \boldsymbol{b})$ son los parámetros y $(\boldsymbol{x}, y)$ es el par del set de entrenamiento, y si, además, la capa l esta completamente conectada a la $(l+1)$, entonces el error de la capa l se calcula como:

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

Y los gradientes quedan:

$$\begin{gathered} \nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \boldsymbol{\delta}^{(l+1)}\left(\boldsymbol{a}^{(l)}\right)^T \\ \nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \boldsymbol{\delta}^{(l+1)} \end{gathered} \tag{5-6}$$

Sin embargo, si la capa l es convolucional o de pooling, entonces el error se propaga según la siguiente ecuación:

$$\boldsymbol{\delta}_k^{(l)} = upsample\left(\left(\boldsymbol{W}_k^{(l)}\right)^T \boldsymbol{\delta}_k^{(l+1)}\right) \odot \boldsymbol{f}'\left(\boldsymbol{z}_k^{(l)}\right) \tag{5-7}$$

En la ecuación anterior k es un índice que hace referencia al filtro correspondiente y $\boldsymbol{f}'\left(\boldsymbol{z}_k^{(l)}\right)$ 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. A continuación, se muestra un ejemplo práctico para comprender este concepto mejor:

$$\begin{cases} mean\ pooling: \boldsymbol{\delta} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix} \\ max\ pooling: \boldsymbol{\delta} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 \end{pmatrix} \end{cases} \tag{5-8}$$

Por último, para calcular el gradiente con respecto a cada uno de los filtros se realiza una convolución tipo valid y se rota la matriz de error $\boldsymbol{\delta}_k^{(l)}$ de la misma forma que los filtros en la capa convolucional:

$$\begin{aligned}\nabla_{\boldsymbol{W}_k^{(l)}} J(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x},y) &= \sum_{i=1}^{m}\left(\boldsymbol{a}_i^{(l)}\right) * \mathrm{rot90}\left(\boldsymbol{\delta}_k^{(l+1)},2\right)\\ \nabla_{\boldsymbol{b}_k^{(l)}} J(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x},y) &= \sum_{a,b}\left(\boldsymbol{\delta}_k^{(l+1)}\right)_{a,b}\end{aligned} \tag{5-9}$$

Donde $\boldsymbol{a}^{(l)}$ es la entrada a la capa l y $\boldsymbol{a}^{(1)}$ es la imagen de entrada. La operación $\left(\boldsymbol{a}_i^{(l)}\right) * \boldsymbol{\delta}_k^{(l+1)}$ es la convolución válida entre la i-ésima entrada de la l-ésima y el error con respecto al k-ésimo filtro.

5.5 Resultados experimentales

A continuación, se va a implementar red neuronal en MATLAB usando como ejemplo los códigos dentro de la carpeta *cnn/*. Los códigos completos se encuentran en apéndices (se especifica en cual al hablar en particular sobre alguno de los códigos en los siguientes párrafos), sin embargo, se procederá a explicar dichos códigos grosso modo y se insertarán aquellas partes del mismo que sean más interesantes, así como de los resultados obtenidos.

Se va a implementar una CNN para la clasificación de dígitos de los que provee el set de datos MNIST. La arquitectura de la red constará de una capa de convolución y pooling seguidas de una capa fully connected, la cual proporcionará la salida de la red mediante regresión sofmax. El tipo de pooling que se utilizará será el de la media y el algoritmo de retropropagación para poder calcular los gradientes. Por último, se entrenará a la red para conseguir los parámetros óptimos mediante el método Momentum.

El código principal es la función *cnn_train* (contenida en el *Apéndice V – Código 1*), la cual inicializa todos los datos necesarios para poder ejecutar el resto de códigos. Se usarán 20 filtros de dimensión 9×9 para la capa convolucional y la dimensión de pooling será de 2×2. Del mismo modo, se cargan los datos del MNIST y se inicializan los parámetros de la red con la función *cnnInitParams* (completa en el *Apéndice V – Código 2*).

Lo siguiente que se debe realizar para poder continuar el código es la función *cnnCost* (contenida en el *Apéndice V – Código 3*), la cual se explicará posteriormente. Una vez esta función está completa, se ejecuta, si uno lo desea, un código de verificación para comprobar que el gradiente calculado con la función *cnnCost* es correcto. La función que realiza dicha función es *computeNumericalGradient* (*Apéndice V – Código 4*) y se basa en que el gradiente se puede calcular como:

$$\frac{d}{d\theta_i} J(\boldsymbol{\theta}) = \frac{J(\boldsymbol{\theta};\theta_i+\epsilon) - J(\boldsymbol{\theta};\theta_i-\epsilon)}{2\epsilon} \tag{5-10}$$

Es decir, el i-ésimo elemento del gradiente se puede calcular como el valor de la función coste cuando dicho elemento es $\theta_i + \epsilon$ menos el valor del coste cuando es $\theta_i - \epsilon$ partido de dos veces ϵ, cuyo valor se suele establecer en 10^{-4}.

Con los valores obtenido por este método y el gradiente obtenido mediante la función *cnnCost*, se pueden comparar y la diferencia entre ellos debería ser menor que 10^{-9} y si no es así, salta un error y no se continúa con el código.

Una vez se ha comprobado que el gradiente es correcto y, por tanto, la función *cnnCost* devuelve los resultados adecuados, el siguiente paso es entrenar la red para obtener aquellos parámetros que minimizan la función coste. La función encargada de esta tarea es *minFuncSGD* (*Apéndice V – Código 5*). Dicha función realiza todo lo explicado en el apartado 5.3: permuta los datos, toma minibatches hasta completar el set de entrenamiento completo, al cabo de ciertas iteraciones de inicio aumenta γ, calcula el gradiente con la función *cnnCost* y actualiza $\boldsymbol{\theta}$ mediante las ecuaciones (5-4). Esto lo realiza para 3 epochs y al final de cada una reduce el ratio de aprendizaje a la mitad. Escrito en MATLAB resulta:

```
%% SGD loop
it = 0;
for e = 1:epochs

    % randomly permute indices of data for quick minibatch sampling
    rp = randperm(m);

    for s=1:minibatch:(m-minibatch+1)
        it = it + 1;

        % increase momentum after momIncrease iterations
        if it == momIncrease
            mom = options.momentum;
        end;

        % get next randomly selected minibatch
        mb_data = data(:,:,rp(s:s+minibatch-1));
        mb_labels = labels(rp(s:s+minibatch-1));

        % evaluate the objective function on the next minibatch
        [cost grad] = funObj(theta,mb_data,mb_labels);

        % Update theta according to SGD with Momentum
        velocity=mom*velocity+alpha*grad;
        theta=theta-velocity;

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

    % aneal learning rate by factor of two after each epoch
    alpha = alpha/2.0;
end;
```

Tras entrenar a la red solo resta comprobar la precisión de ésta. Para ello, carga el set de prueba y ejecuta la función *cnnCost* con la peculiaridad de que ahora devolverá la clase predicha gracias al último argumento que está a *true*. Gracias a estas predicciones, se puede comprobar cuantas veces ha acertado la red comparando dicho valor con el de los targets, pasa el número de aciertos a porcentaje y muestra el resultado por pantalla.

Por último, solo queda explicar la función *cnnCost* en más profundidad, pues esta es clave para el correcto funcionamiento de toda la red. Esta función devuelve el coste, el gradiente y, si el flag *pred* está a valor *true*, devuelve las predicciones realizadas. Lo primero que realiza es la conversión de los parámetros, que están previamente en forma de vector, mediante la función *cnnParamsToStack* (*Apéndice V – Código 6*) en pesos y bias para poder ser usados por la red.

A continuación, se realiza una propagación feedforward a través de las diferentes capas de la red, de manera que las activaciones de las capas convolucional y pooling se realizan de la manera vista en los apartados 5.1 y 5.2 respectivamente y a través de la capa fully connected de la misma forma vista en el apartado 4.5. Las activaciones de cada capa se guardan en variables independientes.

Del mismo modo, tanto el coste como las predicciones en el caso de que el flag *pred* esté activo se calculan de una forma análoga al código correspondiente provisto en el apartado 4.5. La única diferencia con respecto a dicho código tanto en este cálculo como en el de feedforward es que en este caso no se trabaja con variables tipo celda y estructuras, sino con variables tipo matrices.

Tras calcular el coste y realizar la propagación feedforward, es el momento de realizar el algoritmo de retropropagación. Primero se calcularán los errores en cada capa para, posteriormente, poder calcular los gradientes a lo largo de las diferentes capas. Con respecto a las capas fully connected no hay novedad ya que se calculan según lo visto en los apartados 4.4 y 4.5, sin embargo, para la capa de pooling es necesario realizar la operación *upsample*. Esta operación se puede llevar a cabo en MATLAB mediante el comando *kron*, el cual realiza lo siguiente:

$$\text{kron}\left(\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix},\begin{pmatrix}1 & 1\\ 1 & 1\end{pmatrix}\right)=\begin{pmatrix}1 & 1 & 2 & 2\\ 1 & 1 & 2 & 2\\ 3 & 3 & 4 & 4\\ 3 & 3 & 4 & 4\end{pmatrix} \tag{5-11}$$

Una vez realizada la operación de *upsample*, lo único que falta es propagar el error a través de la capa de pooling dividiendo el resultado entre la dimensión de la región de pooling. Por último, para propagar el error a través de la capa convolucional solo es necesario multiplicar el error de la capa posterior por la derivada de la función de activación. Teniendo todo esto en cuenta, los errores se pueden calcular en MATLAB como:

```
errorsOutputSoftmax=(probs-targets)/numImages;

errorsInputSoftmax=Wd'*errorsOutputSoftmax;
errorsOutputPool=reshape(errorsInputSoftmax,outputDim,outputDim,numFilters,numImages);
errorsInputPool=zeros(convDim,convDim,numFilters,numImages);

for imageNum=1:numImages
    for filterNum=1:numFilters

errorsInputPool(:,:,filterNum,imageNum)=(kron(errorsOutputPool(:,:,filterNum,imageNum),ones(poolD
im)))/(poolDim^2);
    end
end

errorsInputConv=errorsInputPool.*activations.*(1-activations);
```

Posteriormente, gracias a los errores se puede calcular de una manera sencilla los gradientes de cada una de las capas gracias a las ecuaciones (5-6) y (5-9). Dichas ecuaciones expresadas en código de MATLAB quedan como:

```
Wd_grad=errorsOutputSoftmax*(activationsPooled');
bd_grad=sum(errorsOutputSoftmax,2);

for imageNum=1:numImages
    for filterNum=1:numFilters

        e=errorsInputConv(:,:,filterNum,imageNum);
        errorsInputConv(:,:,filterNum,imageNum)=rot90(e,2);

    end
end

for filterNum=1:numFilters
    for imageNum=1:numImages


Wc_grad(:,:,filterNum)=Wc_grad(:,:,filterNum)+conv2(images(:,:,imageNum),errorsInputConv(:,:,filt
erNum,imageNum),'valid');

    end
end
```

```
for filterNum=1:numFilters

    e=errorsInputConv(:,:,filterNum,:);
    bc_grad(filterNum)=sum(e(:));

end
```

El último paso es simplemente convertir los gradientes calculados en un solo vector para poder ser utilizado como argumento al llamar a la función *minFuncSGD*.

Tras haber explicado cada una de las funciones y la labor que desempeñan en cuanto a la red, es hora de conocer los resultados obtenidos:

```
Accuracy is 97.090000
Elapsed time is 612.769199 seconds.
```

Figura 5-6: Resultados obtenidos con una CNN

Como se puede observar, el resultado obtenido es prácticamente idéntico al obtenido con una Red Neuronal Multicapa y el tiempo invertido es de aproximadamente 10 minutos, es decir, un poco más del doble de lo que duró el entrenamiento de la Red Neuronal Multicapa. Por tanto, ¿cuál es la razón para elegir las CNN frente a las redes normales? La respuesta ya se dio previamente en las primeras líneas del apartado 5.1: aunque este ejemplo en concreto también se pueda realizar con una Red Neuronal Multicapa, lo más normal en el reconocimiento de imágenes es que esto no sea posible computacionalmente cuando el tamaño de dichas imágenes empieza a crecer. Por tanto, para aplicaciones más ambiciosas es casi de rigor utilizar este tipo de redes, las cuales solo variarán en el número de capas y la función de activación, ya que el resto es análogo.

6 IMPLEMENTACIÓN EN PYTHON

6.1 Librerías necesarias

En el apartado anterior se ha realizado la implementación en MATLAB de una CNN con una precisión por encima del 97%. El resultado obtenido es realmente positivo y está a la altura de otras redes destinadas a la clasificación de imágenes usando el set de dados MNIST, sin embargo, la realidad es que la mayoría de las redes neuronales están implementadas en Python dado que es un lenguaje de programación libre y, además, es el más extendido a la hora implementar cualquier sistema automático.

Dado que se ha explicado cómo realizar una CNN capaz de clasificar las imágenes provistas por el MNIST en el apartado anterior, no se va a explicar cómo realizar el código, ya que este solamente es la traducción de MATLAB a Python. Sin embargo, debido a que existen algunas diferencias realmente importantes entre ambos lenguajes, se procederá a comparar aquellas partes que si sean realmente diferentes y se mostrará la solución adoptada.

Lo primero que hay que hacer, a diferencia de MATLAB es importar todas aquellas librerías o funciones que vayan a ser necesarias para la ejecución del código. Aparte del nombre de las funciones como *cnnCost* o *cnnInitParams*, que son las funciones propias del código, es necesario el uso de las siguientes librerías:

- *From __future__ import division*: importar el módulo división permite realizar divisiones de manera que el resultado sea un número racional en vez de un entero. Por ejemplo, al dividir 1/2 se obtendría 0.5 en lugar de 0. Si se deseara realizar una división con resultado entero, resultaría de la forma 1//2 cuyo resultado sería 0.
- *Import os*: Esta librería permite manipular directorios o incluso poder leer o escribir diferentes archivos. Se usará para cargar el directorio donde están presentes como archivos binarios del MNIST.
- *Import numpy as np*: Es una librería fundamental de Python, usada casi en cualquier código que necesite realizar operaciones algebraicas. Por ejemplo, para conocer las dimensiones de una matriz, crear matrices nulas, rotar matrices o incluso para guardar datos en un archivo de texto.
- *From pdb import set_trace as pb*: Esta librería es necesaria para poder realizar una depuración del código y establecer breakpoints para comprobar el valor de las variables.
- *Import time*: Sirve solamente para conocer el tiempo en la línea de código correspondiente. Al tener dos puntos en los que medir el tiempo, se puede saber cuánto ha tardado el código en ejecutarse mediante la diferencia entre ambos.
- *Import struct*: Esta librería sirve para poder utilizar los archivos binarios del MNIST y transformalos en imágenes. Cabe destacar que antes se usó la librería *mnist* para cargar los datos, pero esta precisaba de una descarga previa en cada ejecución que ralentizaba el proceso y lo imposibilitaba ante la falta de conexión a internet.
- *Import math*: Es una librería para realizar operaciones matemáticas complejas tales como, por ejemplo, la raíz cuadrada de un número o su exponencial. Esto será muy útil a la hora de calcular la función sigmoide.
- *Import scipy*: Una de las librerías más usadas en Python. De ella, es interesante la función que posee para poder realizar la convolución entre matrices.

Gracias a todas estas librerías podemos implementar sin problemas el código anteriormente realizado en MATLAB y comprobar su correcto funcionamiento en Python con el fin de comprobar en que sistema obtiene mejores resultados (deberían oscilar en el mismo rango) y en cuánto tiempo.

6.2 Diferencias con respecto a MATLAB

A pesar de que ambos códigos van a realizar las mismas funciones, existen grandes diferencias a la hora de implementarlo en Python con respecto a MATLAB. Estas diferencias son imperceptibles a menos que uno realice una depuración completa del código comprobando que los resultados en ambos deben ser idénticos.

La primera diferencia es como Python carga los datos. Más allá de que el código sea bastante más complicado, la diferencia radica en los datos una vez cargados, ya que, por ejemplo, MATLAB carga los archivos binarios como imágenes cuya intensidad de píxeles están comprendida entre $[0,1]$, mientras que en Python se cargan en el rango $[0,255]$. La solución a esto es bastante simple, pues solo hay que dividir la imagen entre 255.

```
46 path="."
47 fname_img = os.path.join(path,'train-images-idx3-ubyte')
48 fname_lbl = os.path.join(path,'train-labels-idx1-ubyte')
49
50 with open(fname_img, 'rb') as fimg:
51         magic, num, rows, cols = struct.unpack(">IIII",fimg.read(16))
52         img = np.fromfile(fimg, dtype=np.uint8).reshape(-1,rows,cols)
53         images = img/255
54
55 with open(fname_lbl, 'rb') as flbl:
56         magic, num = struct.unpack(">II",flbl.read(8))
57         labels = np.fromfile(flbl, dtype=np.int8)
```

Figura 6-1: Carga del set MNIST

Aparte de esto, existe una gran diferencia entre ambos lenguajes de programación, no solo al manejar este set de datos, sino al manejar matrices en general. Si uno se fija en la línea 52 de la figura anterior, se pueden reseñar tres cosas. La primera de ellas es que se pueden concatenar determinados comandos usando un punto, es decir, concatena las órdenes *fromfile* (toma los datos del archivo) y *reshape* (cambia las dimensiones de la matriz). El comando *np.* no cuenta como concatenación pues simplemente indica que se está usando la librería *numpy*. Esto evita el uso de más líneas y permite compactar el código. De no ser por esta concatenación, los comandos quedarían:

img=np.fromfile(fimg, dtype=uint8)

img=np.reshape(-1,rows,cols)

Los dos restantes puntos reseñables residen en el comando *reshape*. En primer lugar, parece obvio, pero es necesario que las dimensiones de destino sean compatibles con las de origen. Para ilustrar esto, si se tiene una matriz de dimensiones $4\times4\times8 = 128$, una posibilidad para redimensionarla sería $16\times8 = 128$, pero no se podría redimensionar a $15\times4 = 60$, ya que el número de elementos no es el mismo en el origen y destino. Esto también ocurre en MATLAB, sin embargo, es necesario explicar el significado del -1 en el comando *reshape*. Tomando como ejemplo las matrices anteriores, el comando *reshape(-1,8)* proporcionará una matriz de 8 columnas y el número de filas necesarias para que el número de elementos entre origen y destino permanezca constante, es decir, 16.

Por último, pero no menos importante, existe una gran diferencia a la hora de manejar matrices de más de dos dimensiones. Mientras que en MATLAB las dimensiones de una imagen se representan mediante *(rows,columns,channels)*, en Python las dimensiones superiores a dos se apilan hacia la izquierda, de modo que queda *(channels,rows,columns)*. Esto hay que tenerlo muy en cuenta, ya que a lo largo del código se usan operaciones con matrices de hasta 4 dimensiones (esta última en Python quedaría a la izquierda de *channels*).

Otra diferencia importante entre MATLAB y Python es que en MATLAB el índice inicial para los vectores y cada dimensión de las matrices era 1, mientras que en Python es 0. Si esto no se tiene en cuenta, se pueden ocasionar errores bastante graves, pues se podría elegir una imagen o filtro diferente a la que debería. Además, aparte de tener esto en cuenta a la hora de indexar matrices, también es importante recalcar que, dado que en MATLAB no se podía indexar con 0, aquellas labels que eran 0 debían establecerse a 10 (para poder usarse como índices en *sub2ind*, explicado en capítulo 3.3.2). En Python no es necesario realizar dicha operación pues el índice inicial es 0.

Con respecto a la función *sub2ind*, ésta no existe en Python y no hay ninguna que se le asemeje para poder extraer las probabilidades correctas de la matriz A, tal como se hizo en los apartados 3.3.2, 4.5 y 5.5. La única solución encontrada es la realización de un bucle, guardando en un vector las probabilidades para después sumarlas, tal como indica la ecuación (4-18):

```
161         p=np.zeros((1,numImages))
162
163         for imageNum in xrange(numImages):
164                 p[0][imageNum]=A[labels[imageNum]][imageNum]
165
166         cost=-np.sum(p)/numImages
```

Figura 6-2: Función sub2ind en Python

Una diferencia mínima, pero que es importante recalcar es la operación "elevar un número a otro". Frente a la mayoría de lenguajes de programación en los que el comando sería *y=x^2*, en Python el comando resulta ser *y=x**2*. Es una diferencia casi sin importancia, pero que cualquier persona nueva en este lenguaje suele ignorar y que puede causar errores en la programación si no se tiene cuidado.

La última diferencia reseñable con respecto al código en MATLAB reside en la función *reshape*. Existen dos formar de redimensionar una matriz: ordenando los índices como C o como Fortran (F). El primero de ellos toma las filas sucesivamente de la matriz de origen y las va apilando en las filas de la matriz destino, mientras que el método F toma las columnas de la matriz origen y las apila como columnas en el destino. Para ilustrar esto:

$$\begin{cases} C: \boldsymbol{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}_{3\times4} \rightarrow \boldsymbol{B} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}_{4\times3} \\ F: \boldsymbol{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}_{3\times4} \rightarrow \boldsymbol{B} = \begin{pmatrix} 1 & 6 & 11 \\ 5 & 10 & 4 \\ 9 & 3 & 8 \\ 2 & 7 & 12 \end{pmatrix}_{4\times3} \end{cases} \quad (6\text{-}1)$$

En MATLAB el método por defecto es Fortran, mientras que en Python es C. En Python, para matrices 2D, es sencillo cambiar el método de indexado a Fortran. Por ejemplo, realizar la operación anterior en C resultaría en el comando *B=np.reshape(A, (4,3))* y en F *B=np.reshape(A, (4,3), order='F')*. El parámetro *order* permite cambiar entre ambos métodos (no es necesario usar *order='C'* en el primer comando porque viene por defecto).

Sin embargo, si la matriz de origen tiene más de dos dimensiones, ambos métodos se complican, ya que en Python, a diferencia de MATLAB, la estructura es *(channels,rows,columns)*. Esto se traduce en que al utilizar la orden *reshape*, independientemente del método de indexación, primero tomará elementos de la dimensión *channels*, y esto no es nada deseable porque se estarían mezclando los valores de las distintas imágenes y filtros, desorganizando todos los valores importantes para la red.

Si uno supone que tiene en Python una matriz $\boldsymbol{A}$ de dimensiones $2\times2\times2$, cuyos *channels* son:

$$\boldsymbol{A}\begin{cases} Channel\ 1: \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \\ Channel\ 2: \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \end{cases} \tag{6-2}$$

En la siguiente tabla se va representar lo que se obtiene en Python con los dos métodos de indexación, así como el resultado deseado al redimensionalizar la matriz como 4×2:

MÉTODO DE INDEXACIÓN	MATRIZ OBTENIDA	MATRIZ DESEADA
INDEXACIÓN POR FILAS (C)	$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \\ 3 & 5 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}$
INDEXACIÓN POR COLUMNAS (F)	$\begin{pmatrix} 1 & 2 \\ 5 & 6 \\ 3 & 4 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 1 & 5 \\ 3 & 7 \\ 2 & 6 \\ 4 & 8 \end{pmatrix}$

Tabla 6-1: Comparación de matrices según el método de indexación

Debido a lo anterior, para redimensionar una matriz multidimensional en Python indexando mediante Fortran, no se puede realizar el comando *reshape*, pues el resultado no es el deseado. En el siguiente apartado, se abordará una solución para este problema. Si se realiza el código utilizando la indexación C, el resultado de la red está bastante lejos de ser bueno, por lo que es necesario implementar el método Fortran.

6.3 Soluciones y Optimización

En el apartado anterior se comentaron las diferencias más significativas que se encontraron a la hora de programar en Python frente a MATLAB. En el presente apartado se va a explicar cómo se llegó a conocer dichas diferencias, las soluciones adoptadas y las optimizaciones realizadas en el código para que tardara menos tiempo.

En primer lugar, la carga de los datos no se realizaba según el código de la Figura 6-1, sino que se hacía mediante la librería *mnist* y se utilizaba el siguiente código:

```
41 mndata=MNIST('.')
42 images, labels = mndata.load_training()
43 images = np.reshape(images,(-1,imageDim,imageDim))
44 images = images/255
```

Figura 6-3: Carga de datos del MNIST con librería mnist

Es cierto que si se realiza de esta manera el código queda más compacto y comprensible, sin embargo, el gran inconveniente que presenta es que en cada ejecución ha de descargar dicho set de datos. Esto ralentiza bastante el código, sobre todo cuando se dispone de los archivos binarios ya descargados, y hace imposible su ejecución si no existe una conexión a internet. Por tanto, debido a estos motivos se decidió cambiar esto y utilizar el código

de la Figura 6-1 para optimizar el proceso y que no dependa de una conexión a internet.

Todas las diferencias explicadas en el apartado anterior se desconocían cuando se empezó a realizar la implementación en Python. Dado que al ejecutar el código los resultados eran completamente diferentes a los esperados, para comprobar los fallos existentes se procedió a realizar una exhaustiva depuración. Para ello, se generó un vector *theta* en MATLAB, se guardó con el comando *save* y se pasó este archivo a Python, donde con un sencillo código (el de la Figura 6-4) se cargaba dicho vector. Teniendo los mismos datos iniciales en MATLAB y en Python, se podía ir línea por línea comprobando los resultados en ambos programas para comprobar los puntos de fallo.

```
5 def test():
6         test=scipy.io.loadmat('paramsInitiated')
7         theta=test['theta']
8
9         return theta
```

Figura 6-4: Carga de un archivo .mat

Gracias a estas sencillas líneas de código, se pudo depurar el programa y conocer aquellos puntos en los que el resultado obtenido no coincidía con el de MATLAB. De esta manera se pudo conocer, por ejemplo, que el método por defecto de indexación de Python es C.

Hasta ahora se ha hablado de cómo se descubrió el problema que presentaba la función *reshape* para matrices multidimensionales, pero no se ha hablado de cómo solucionar dicho problema. La primera idea que se tuvo para resolver esto fue el uso de una función creada a mano en la que, mediante el uso de bucles, permitiera pasar de una dimensión de origen específica a una de destino específica. Por ejemplo, si se quería redimensionar una matriz de 2 dimensiones a una de 4, se usaba la siguiente función:

```
3 def matlabReshape2DTo4D(a,numImages,numFilters,outputDim):
4         b=np.zeros((numImages,numFilters,outputDim,outputDim))
5
6         for i in xrange(numImages):
7                 for j in xrange(numFilters):
8                         for k in xrange(outputDim):
9                                 for l in xrange(outputDim):
10                                        b[i][j][l][k]=a[j*outputDim**2+outputDim*k+l][i]
11
12        return b
```

Figura 6-5: Función para redimensionar matriz 2D a 4D

Se creó una función específica para cada *reshape* necesario de una forma análoga a la mostrada en la Figura anterior. Sin embargo, esto realmente es muy poco óptimo ya que el uso de tantos bucles ralentizaba el código significativamente, aunque el resultado final era el correcto.

Lo deseable sería poder expresar la matriz como en MATLAB, es decir, *(rows,columns,channels)* para poder utilizar *order='F'* en *reshape* y que realizara la operación correctamente. Tras una extensa investigación sobre las diferentes funciones relacionadas con *reshape*, se encontró una que permitía realizar el comportamiento buscado: *swapaxes*.

La función *swapaxes* permite intercambiar las dimensiones de las matrices, de manera que si se tiene una matriz de $4\times4\times2$ y se utiliza la orden *swapaxes(0,2)* se obtiene una matriz de $2\times4\times4$. Para ilustrar mejor el funcionamiento de esta función aplicada al problema que nos atañe, si se supone una matriz $\boldsymbol{A}$ de dimensiones $2\times2\times2\times3$, es decir, *(rows,columns,numFilters,numImages)* y suponiendo que $\boldsymbol{A}$ vale:

	FILTER 1	FILTER 2
IMAGE 1	$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}$	$\begin{pmatrix}5 & 6\\7 & 8\end{pmatrix}$
IMAGE 2	$\begin{pmatrix}9 & 10\\11 & 12\end{pmatrix}$	$\begin{pmatrix}13 & 14\\15 & 16\end{pmatrix}$
IMAGE 3	$\begin{pmatrix}17 & 18\\19 & 20\end{pmatrix}$	$\begin{pmatrix}21 & 22\\23 & 24\end{pmatrix}$

Tabla 6-2: Matriz de 4 dimensiones

La matriz que se desea obtener mediante el método de indexación Fortran es:

$$\boldsymbol{T} = \begin{pmatrix}1 & 9 & 17\\3 & 11 & 19\\2 & 10 & 18\\4 & 12 & 20\\5 & 13 & 21\\7 & 15 & 23\\6 & 14 & 22\\8 & 16 & 24\end{pmatrix} \tag{6-3}$$

La matriz obtenida en Python mediante el comando *B=np.reshape(A,(-1,3),order='F')* resulta ser:

$$\boldsymbol{B} = \begin{pmatrix}1 & 19 & 14\\9 & 7 & 22\\17 & 15 & 4\\5 & 23 & 12\\13 & 2 & 20\\21 & 10 & 8\\3 & 18 & 16\\11 & 6 & 24\end{pmatrix} \tag{6-4}$$

Dicha matriz, como se explicó anteriormente mezcla los elementos de las distintas imágenes y filtros, por lo que no es correcto el resultado. Sin embargo, si se pone en práctica el comando *swapaxes* de manera que se cambien las dimensiones en Python a *(rows,columns,numFilters,numImages)* y luego se redimensionaliza, es decir, se usa el comando *C=A.swapaxes(0,2).swapaxes(1,3).swapaxes(2,3).reshape((-1,3),order='F')*, se obtiene:

$$\boldsymbol{C} = \begin{pmatrix} 1 & 9 & 17 \\ 3 & 11 & 19 \\ 2 & 10 & 18 \\ 4 & 12 & 20 \\ 5 & 13 & 21 \\ 7 & 15 & 23 \\ 6 & 14 & 22 \\ 8 & 16 & 24 \end{pmatrix} \tag{6-5}$$

Como se puede observar, la matriz ahora obtenida se corresponde con la matriz deseada de la ecuación (6-3). Gracias a este comando, se puede realizar sin problemas una indexación tipo Fortran para matrices multidimensionales de una forma bastante compacta concatenando las órdenes. Este método se usará para aquellos casos en los que haya que redimensionalizar matrices, sustituyendo así a las funciones con bucles antes explicadas.

Una vez ha quedado explicado todo lo relativo a la función *reshape* solo resta explicar cómo se ha realizado la operación pooling, pues no es tan sencillo como en MATLAB. El comando equivalente a *pooledImage = convolved(1:poolDim:end, 1:poolDim:end)* en Python sería *pooledImage = convolved[0:None:poolDim] [0:None:poolDim]*, sin embargo, ésto no funciona y da un resultado no deseable.

Para solventar dicho problema, en primera instancia se pensó en dos bucles anidados que recogieran de la matriz *convolved* los valores adecuados y los almacenara en *pooledImage* tal y como se observa en la siguiente imagen:

```
107                     i = 0
108                     j = 0
109                     a = 0
110                     b = 0
111
112                     while (i<(convDim-poolDim+1)):
113                             while (j<(convDim-poolDim+1)):
114                                     pooledImage[a][b] = convolved[i][j]
115                                     j += poolDim
116                                     b += 1
117                             j = 0
118                             b = 0
119                             i += poolDim
120                             a += 1
```

Figura 6-6: Pooling con bucles anidados

El resultado era el que debería ser, pero, debido a que el uso de dichos bucles ralentizaba el código de manera significativa, se buscaron otras formas de realizar la operación de pooling.

Asímismo. se probó también con una función perteneciente a una librería, pero, ya que el tiempo de ejecución era el mismo que con los dos bucles anidados, se descartó dicha función. La solución final adoptada vino de la mano del comando *swapaxes* explicado anteriormente. Se almacenó en un vector los índices necesarios a recoger de la matriz *convolved* y se almacenó en *pooledImage* los valores correspondientes a dichos índices mediante los comandos:

ind=range(convDim)

ind=ind[0:None:poolDim]

pooledImage=convolved[ind].swapaxes(0,1)[ind].swapaxes(0,1)

De esta manera se recogen los resultados de la operación pooling de una manera rápida y compacta gracias a la concatenación.

La última modificación realizada en el código es que guarde en un archivo de texto el vector *theta* si la precisión alcanzada con la red supera el 97% para no tener que entrenarla de nuevo:

```
151 if acc>0.97:
152         print "Saving optimized theta in file savedata.txt"
153         np.savetxt('savedata.txt',opttheta)
```

Figura 6-7: Almacenamiento de parámetros si la precisión supera el 97%

6.4 Resultados obtenidos

Una vez explicadas las diferencias existentes con respecto al lenguaje de programación MATLAB y las soluciones adoptadas para solventar dichos problemas, el resto del código se implementa de forma análoga a MATLAB, pero con la sintaxis de Python.

Si se abre un terminal en Python y uno se dirige al directorio donde está contenido el código, al ejecutar *python run_train.py*, éste se ejecutará. Comenzará entrenando a la red durante unas 700 iteraciones y, tras esto, calculará la precisión de la red, así como el tiempo invertido, con el añadido de que, si la precisión está por encima del 97%, guardará los parámetros en un archivo de texto.

Al ejecutar los pasos anteriores, se obtiene:

```
Testing trained network
Accuracy is 97.25 %
The number of successfully predicted images is 9725
Elapsed time is 17.9043012659 minutes
Saving optimized theta in file savedata.txt
Done
```

Figura 6-8: Resultados obtenido en Python

En la figura anterior se puede observar que la precisión obtenida es del 97.25%, la cual está por encima de lo obtenido en MATLAB y moviéndose en el rango de 97 y 97.4%, por lo que puede decirse que la implementación obtenida en Python ha sido un éxito.

Con respecto al tiempo invertido que es aproximadamente de 18 minutos, es superior al tiempo que tarda MATLAB en entrenar a la red, sin embargo, hay que tener varias cosas en cuenta. La primera de ellas es que este tiempo se ha conseguido después de optimizar el código de la forma descrita en el apartado 6.3, ya que al principio el tiempo era del orden de 30 minutos. Lo segundo es que se está ejecutando sobre una máquina virtual, por lo que, si se instalara Ubuntu en el mismo ordenador y se ejecutara el código, el tiempo de ejecución sería bastante menor ya que aprovecharía más los recursos del ordenador. Por último, Python también ofrece la posibilidad de usar la GPU para acelerar los cálculos (recurso que no se está utilizando) de manera que podría disminuir el tiempo de ejecución y, si se combina ésto con el uso de Ubuntu como Sistema Operativo principal, el tiempo disminuiría aún más.

Por tanto, si se realizaran las mejoras descritas anteriormente, el tiempo de ejecución estaría a la par que el de MATLAB y la diferencia entre ambos tanto en precisión como en duración sería mínima. Gracias a esto, las únicas diferencias entre ambos a la hora de realizar una CNN residirían en la propia sintaxis de cada lenguaje y la aplicación para la que se va a implementar, siendo el resto de elementos análogos.

7 CONCLUSIONES

Tal y como se ha podido observar a lo largo del presente proyecto, las redes neuronales resultan muy interesantes en cuanto a problemas de clasificación se refieren, en concreto las CNN para el reconocimiento de imágenes. La posibilidad de obtener una precisión superior al 97% y de poder añadir clases o imágenes a clasificar sin tener que modificar solo algunos parámetros de la red, hacen que las redes neuronales sean una herramienta muy poderosa y útil.

Además, se han probado diferentes tipos de redes tanto en MATLAB como Python, y en todas ellas, independientemente del lenguaje de programación, han dado unos resultados más que válidos. Si bien la programación en MATLAB resulta más intuitiva que en Python debido al completo álgebra matricial que presenta y la disposición de las dimensiones de las matrices, la implementación en Python conociendo estas diferencias es igual de sencilla.

El único inconveniente de las redes neuronales, sobre todo cuando el número de capas comienza a incrementarse, es la necesidad de utilizar equipos relativamente potentes (al menos a la hora de entrenar a la red), ya que la ejecución del código en un portátil de hace aproximadamente 4 años era demasiado lenta (alrededor de 2 horas). Sin embargo, ésto es algo que todas aquellas personas que usan redes neuronales conocen y aceptan.

Con este proyecto se pretendía brindar una introducción a las redes neuronales, su funcionamiento y las matemáticas detrás de ellas. Este campo está en constante crecimiento e investigación, pero gracias a este proyecto, el lector podrá tener una idea general de cómo se han llevado a cabo las redes más importantes a día de hoy, sobre todo aquellas relacionadas con el reconocimiento de imágenes, y asienta las bases a la hora de crear redes más complejas.

8 LÍNEAS DE TRABAJO FUTURAS

Debido a la gran variedad de redes neuronales existentes a día de hoy, existen muchas líneas de investigación que podrían servir como continuación a este proyecto. A continuación, se presentan las más interesantes:

- Creación de una red neuronal que, en vez de reconocer imágenes, sea capaz de reconocer patrones de sonido para, por ejemplo, reconocer cual es la persona que está hablando por una grabación.
- Creación de una CNN cuyo set de imágenes sea de unas dimensiones más grandes, de manera que esto fuerce el uso de más capas en la red. Esto se podría complementar con, por ejemplo, tomar las imágenes a mano mediante una cámara (de señales de tráfico, por ejemplo) y que un determinado sistema actúe en función de que detecte la red.
- Usar diferentes metodologías para entrenar a la red, ya que en este proyecto siempre se usó el aprendizaje supervisado. Sin embargo, existen otros tipos de aprendizaje como el aprendizaje no supervisado o el aprendizaje por refuerzo. Este último es muy interesante, pues la red va aprendiendo a medida que prueba diferentes opciones (como probar que acción en un videojuego es más beneficiosa en una situación y repetirlo en situaciones análogas).

REFERENCIAS

[1] Hagan, M., Demuth, H., Beale, M. and De Jesús, O. (2014). *Neural network design*. 2nd ed.

[2] Beale, M., Hagan, M., Demuth, H. (2016). *Neural Networks Toolbox*.

[3] Dumoulin, V., Visin, F. (2016). *A guide to convolution arithmetic for deep learning.*

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

[5] Stackoverflow.com. (2017). *Stack Overflow*. [online] Available at: https://stackoverflow.com.

[6] Lawebdelprogramador.com. (2017). *La Web del Programador*. [online] Available at: http://www.lawebdelprogramador.com.

[7] Ujjwalkarn. (2016). *An Intuitive Explanation of Convolutional Neural Networks*. [online] Available at: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

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

[9] Gibiansky A. (2014). *Convolutional Neural Networks - Andrew Gibiansky.* [online] Available at: http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/.

[10] Nielsen, M. (2017). *Neural Networks and Deep Learning*. [online] Available at: http://neuralnetworksanddeeplearning.com/chap6.html.

[11] Zamora E. (2017). *Redes Neuronales Convolucionales*. [online] Available at: https://es.scribd.com/doc/295974900/Redes-Neuronales-Convolucionales.

[12] Torres J. (2015). *Introducción práctica al Deep Learning con TensorFlow de Google - parte 1*. [online] Available at: http://jorditorres.org/introduccion-practica-al-deep-learning-con-tensorflow-de-google-parte-1/

[13] Mazur M. (2015). *A Step by Step Backpropagation Example*. [online] Available at: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[14] Kuwajima H. (2014). *Backpropagation in Convolutional Neural Network*. [online] Available at: https://es.slideshare.net/kuwajima/cnnbp.

[15] Jefkine. (2015). *Backpropagation In Convolutional Neural Networks*. [online] Available at: http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[16] Grzegorzgwardys. (2016). *Convolutional Neural Networks backpropagation: from intuition to derivation*. [online] Available at: https://grzegorzgwardys.wordpress.com/2016/04/22/8/.

[17] Condliffe J. (2015). *You're Using Neural Networks Every Day Online - Here's How They Work*. [online] Available at: http://gizmodo.com/youre-using-neural-networks-every-day-online-heres-h-1711616296.

[18] Borowiec S. (2016). *AlphaGo beats Lee Sedol in third consecutive Go game*. [online] Available at: https://www.theguardian.com/technology/2016/mar/12/alphago-beats-lee-sedol-in-third-consecutive-go-game.

[19] Shiffman D. (2016). *Chapter 10. Neural Networks*. [online] Available at: http://natureofcode.com/book/chapter-10-neural-networks/.

[20] Brownlee J. (2016). *Handwritten Digit Recognition using Convolutional Neural Networks in Python with Keras*. [online] Available at: http://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/.

[21] Jacob S. (2015). *MNIST Hand Written Digits Classification Benchmark*. [online] Available at: http://knowm.org/mnist-hand-written-digits-classification-benchmark/.

APÉNDICE I: CÓDIGOS DE REGRESIÓN LINEAL

Código 1

Script	*ex1a_linreg.m*
Descripción	Carga los datos, ejecuta una regresión lineal y muestra los resultados

```
clear all; clc; close all;
%
%This exercise uses a data from the UCI repository:
% Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository
% http://archive.ics.uci.edu/ml
% Irvine, CA: University of California, School of Information and Computer Science.
%
%Data created by:
% Harrison, D. and Rubinfeld, D.L.
% ''Hedonic prices and the demand for clean air''
% J. Environ. Economics & Management, vol.5, 81-102, 1978.
%
addpath ../common
addpath ../common/minFunc_2012/minFunc
addpath ../common/minFunc_2012/minFunc/compiled

% Load housing data from file.
data = load('housing.data');
data=data'; % put examples in columns

% Include a row of 1s as an additional intercept feature.
data = [ ones(1,size(data,2)); data ];

% Shuffle examples.
data = data(:, randperm(size(data,2)));

% Split into train and test sets
% The last row of 'data' is the median home price.
train.X = data(1:end-1,1:400);
train.y = data(end,1:400);

test.X = data(1:end-1,401:end);
test.y = data(end,401:end);

m=size(train.X,2);
n=size(train.X,1);

% Initialize the coefficient vector theta to random values.
theta = rand(n,1);

% Run the minFunc optimizer with linear_regression.m as the objective.
%
% Implement the linear regression objective and gradient computations
% in linear_regression.m
%

tic;
options = struct('MaxIter', 200);
theta = minFunc(@linear_regression, theta, options, train.X, train.y);
fprintf('Optimization took %f seconds.\n', toc);

% Run minFunc with linear_regression_vec.m as the objective.
%
% Implement linear regression in linear_regression_vec.m
% using MATLAB's vectorization features to speed up your code.
% Compare the running time for your linear_regression.m and
```

```
% linear_regression_vec.m implementations.
%
% Uncomment the lines below to run your vectorized code.
% Re-initialize parameters
theta = rand(n,1);
tic;
theta = minFunc(@linear_regression_vec, theta, options, train.X, train.y);
fprintf('Optimization took %f seconds.\n', toc);

% Plot predicted prices and actual prices from training set.
actual_prices = train.y;
predicted_prices = theta'*train.X;

% Print out root-mean-squared (RMS) training error.
train_rms=sqrt(mean((predicted_prices - actual_prices).^2));
fprintf('RMS training error: %f\n', train_rms);

% Print out test RMS error
actual_prices = test.y;
predicted_prices = theta'*test.X;
test_rms=sqrt(mean((predicted_prices - actual_prices).^2));
fprintf('RMS testing error: %f\n', test_rms);


% Plot predictions on test data.
plot_prices=true;
if (plot_prices)
  [actual_prices,I] = sort(actual_prices);
  predicted_prices=predicted_prices(I);
  plot(actual_prices, 'rx');
  hold on;
  plot(predicted_prices,'bx');
  legend('Actual Price', 'Predicted Price');
  xlabel('House #');
  ylabel('House price ($1000s)');
  grid;
end
```

Código 2

Script	*linear_regression.m*
Descripción	Calcula la función coste y el gradiente de la regresión lineal

```
function [f,g] = linear_regression(theta, X,y)
  %
  % Arguments:
  %   theta - A vector containing the parameter values to optimize.
  %   X - The examples stored in a matrix.
  %       X(i,j) is the i'th coordinate of the j'th example.
  %   y - The target value for each example.  y(j) is the target for example j.
  %

  m=size(X,2);
  n=size(X,1);

  f=0;
  g=zeros(size(theta));

  %
  % Compute the linear regression objective by looping over the examples in X.
  % Store the objective function value in 'f'.
  %

  sumaJ=0;
    for i=1:m
```

```
        sumaJ=sumaJ+(theta'*X(:,i)-y(1,i))^2;
    end
  f=0.5*sumaJ;

  %
  % Compute the gradient of the objective with respect to theta by looping over
  % the examples in X and adding up the gradient for each example.  Store the
  % computed gradient in 'g'.
  %

  sumaG=0;
    for j=1:n
        for i=1:m
            sumaG=sumaG+X(j,i)*(theta'*X(:,i)-y(1,i));
        end
        g(j,1)=sumaG;
        sumaG=0;
    end

end
```

Código 3

Script	*linear_regression_vec.m*
Descripción	Calcula la función coste y el gradiente de la regresión lineal de forma optimizada

```
function [f,g] = linear_regression_vec(theta, X,y)
  %
  % Arguments:
  %   theta - A vector containing the parameter values to optimize.
  %   X - The examples stored in a matrix.
  %       X(i,j) is the i'th coordinate of the j'th example.
  %   y - The target value for each example.  y(j) is the target for example j.
  %
  m=size(X,2);
  n=size(X,1);

  % initialize objective value and gradient.
  f = 0;
  g = zeros(size(theta));

  %
  % Compute the linear regression objective function and gradient
  % using vectorized code.
  % Store the objective function value in 'f', and the gradient in 'g'.
  %

  y_hat=(theta'*X-y);
  f=1/2*(sum(y_hat.^2));

  g=X*y_hat';

end
```

APÉNDICE II: CÓDIGOS DE REGRESIÓN LOGÍSTICA

Código 1

Script	*ex1b_logreg.m*
Descripción	Carga los datos, ejecuta una regresión logística y muestra los resultados

```
clear all; clc; close all;

addpath ../common
addpath ../common/minFunc_2012/minFunc
addpath ../common/minFunc_2012/minFunc/compiled

% Load the MNIST data for this exercise.
% train.X and test.X will contain the training and testing images.
%   Each matrix has size [n,m] where:
%      m is the number of examples.
%      n is the number of pixels in each image.
% train.y and test.y will contain the corresponding labels (0 or 1).
binary_digits = true;
[train,test] = ex1_load_mnist(binary_digits);

% Add row of 1s to the dataset to act as an intercept term.
train.X = [ones(1,size(train.X,2)); train.X];
test.X = [ones(1,size(test.X,2)); test.X];


% Training set dimensions
m=size(train.X,2);
n=size(train.X,1);

% Train logistic regression classifier using minFunc
options = struct('MaxIter', 100);

% First, we initialize theta to some small random values.
theta = rand(n,1)*0.001;

%
% Implement batch logistic regression in logistic_regression_vec.m using
% MATLAB's vectorization features to speed up your code.  Compare the running
%
tic;
theta=minFunc(@logistic_regression_vec, theta, options, train.X, train.y);
fprintf('Optimization took %f seconds.\n', toc);

% Print out training accuracy.
tic;
accuracy = binary_classifier_accuracy(theta,train.X,train.y);
fprintf('Training accuracy: %2.1f%%\n', 100*accuracy);

% Print out accuracy on the test set.
accuracy = binary_classifier_accuracy(theta,test.X,test.y);
fprintf('Test accuracy: %2.1f%%\n', 100*accuracy);
```

Código 2

Script	*logistic_regression_vec.m*
Descripción	Calcula la función coste y el gradiente de la regresión logística

```
function [f,g] = logistic_regression_vec(theta, X,y)
  %
  % Arguments:
  %   theta - A column vector containing the parameter values to optimize.
  %   X - The examples stored in a matrix.
  %       X(i,j) is the i'th coordinate of the j'th example.
  %   y - The label for each example.  y(j) is the j'th example's label.
  %
  m=size(X,2);
  n=size(X,1);

  % initialize objective value and gradient.
  f = 0;
  g = zeros(size(theta));


  %
  % Compute the logistic regression objective function and gradient
  % using vectorized code.  (It will be just a few lines of code!)
  % Store the objective function value in 'f', and the gradient in 'g'.
  %

  y_hat=logsig(theta'*X);
  f=-sum(y.*log(y_hat)+(1-y).*log(1-y_hat));

  g=X*(y_hat-y)';

end
```

APÉNDICE III: CÓDIGOS DE REGRESIÓN SOFTMAX

Código 1

Script	*ex1c_logreg.m*
Descripción	Carga los datos, ejecuta una regresión softmax y muestra los resultados

```
clear all; clc;

addpath ../common
addpath ../common/minFunc_2012/minFunc
addpath ../common/minFunc_2012/minFunc/compiled

% Load the MNIST data for this exercise.
% train.X and test.X will contain the training and testing images.
%   Each matrix has size [n,m] where:
%      m is the number of examples.
%      n is the number of pixels in each image.
% train.y and test.y will contain the corresponding labels (0 to 9).
binary_digits = false;
num_classes = 10;
[train,test] = ex1_load_mnist(binary_digits);

% Add row of 1s to the dataset to act as an intercept term.
train.X = [ones(1,size(train.X,2)); train.X];
test.X = [ones(1,size(test.X,2)); test.X];
train.y = train.y+1; % make labels 1-based.
test.y = test.y+1; % make labels 1-based.

% Training set info
m=size(train.X,2);
n=size(train.X,1);

% Train softmax classifier using minFunc
options = struct('MaxIter', 200);

% Initialize theta.  We use a matrix where each column corresponds to a class,
% and each row is a classifier coefficient for that class.
% Inside minFunc, theta will be stretched out into a long vector (theta(:)).
% We only use num_classes-1 columns, since the last column is always assumed 0.
theta = rand(n,num_classes-1)*0.001;

% Call minFunc with the softmax_regression_vec.m file as objective.
%
% Implement batch softmax regression in the softmax_regression_vec.m
% file using a vectorized implementation.
%
tic;
theta(:)=minFunc(@softmax_regression_vec, theta(:), options, train.X, train.y);
fprintf('Optimization took %f seconds.\n', toc);
theta=[theta, zeros(n,1)]; % expand theta to include the last class.

% Print out training accuracy.
tic;
accuracy = multi_classifier_accuracy(theta,train.X,train.y);
fprintf('Training accuracy: %2.1f%%\n', 100*accuracy);

% Print out test accuracy.
accuracy = multi_classifier_accuracy(theta,test.X,test.y);
fprintf('Test accuracy: %2.1f%%\n', 100*accuracy);
```

Código 2

Script	*softmax_regression_vec.m*
Descripción	Calcula la función coste y el gradiente de la regresión softmax

```
function [f,g] = softmax_regression(theta, X,y)
  %
  % Arguments:
  %   theta - A vector containing the parameter values to optimize.
  %       In minFunc, theta is reshaped to a long vector.  So we need to
  %       resize it to an n-by-(num_classes-1) matrix.
  %       Recall that we assume theta(:,num_classes) = 0.
  %
  %   X - The examples stored in a matrix.
  %       X(i,j) is the i'th coordinate of the j'th example.
  %   y - The label for each example.  y(j) is the j'th example's label.
  %
  m=size(X,2);
  n=size(X,1);

  % theta is a vector;  need to reshape to n x num_classes.
  theta=reshape(theta, n, []);
  num_classes=size(theta,2)+1;

  % initialize objective value and gradient.
  f = 0;
  g = zeros(size(theta));

  %
  % Compute the softmax objective function and gradient using vectorized code.
  % Store the objective function value in 'f', and the gradient in 'g'.
  % Before returning g, make sure you form it back into a vector with g=g(:);
  %

y_hat=exp(theta'*X);  % It's the numerator
y_hat=[y_hat; ones(1,size(y_hat,2))]; % Last row set to 1 to have 10 classes
y_hat_sum=sum(y_hat,1); % It's the denominator
y_hat_sum=y_hat_sum-1; % As last row doesn't count, substract it from the total sum

coc=bsxfun(@rdivide,y_hat,y_hat_sum);
A=log(coc);
ind=sub2ind(size(A),y,1:size(y_hat,2));
A(end,:)=0; % Last row set to 0 because the prob that y belongs to 10 is null
f=-sum(A(ind));

B=zeros(size(A));
B(ind)=1;
g=-X*(B-coc)';

g=g(:,1:end-1);  % Last row doesn't count
g=g(:); % make gradient a vector for minFunc

end
```

APÉNDICE IV: CÓDIGOS DE LA RED NEURONAL MULTICAPA

Código 1

Script	*run_train.m*
Descripción	Carga los datos, crea una red neuronal multicapa, la entrena y muestra los resultados

```
clear all; clc; tic;
% runs training procedure for supervised multilayer network
% softmax output layer with cross entropy loss function

%% setup environment
% experiment information
% a struct containing network layer sizes etc
ei = [];

% add common directory to your path for
% minfunc and mnist data helpers
addpath ../common;
addpath(genpath('../common/minFunc_2012/minFunc'));

%% load mnist data
[data_train, labels_train, data_test, labels_test] = load_preprocess_mnist();

%% populate ei with the network architecture to train
% ei is a structure you can use to store hyperparameters of the network
% the architecture specified below should produce  100% training accuracy
% You should be able to try different network architectures by changing ei
% only (no changes to the objective function code)

% dimension of input features
ei.input_dim = 784;
% number of output classes
ei.output_dim = 10;
% sizes of all hidden layers and the output layer
ei.layer_sizes = [256, ei.output_dim];
% scaling parameter for l2 weight regularization penalty
ei.lambda = 0;
% which type of activation function to use in hidden layers
% feel free to implement support for only the logistic sigmoid function
ei.activation_fun = 'logistic';

%% setup random initial weights
stack = initialize_weights(ei);
params = stack2params(stack);

%% setup minfunc options
options = [];
options.display = 'iter';
options.maxFunEvals = 1e6;
options.Method = 'lbfgs';

%% run training
[opt_params,opt_value,exitflag,output] = minFunc(@supervised_dnn_cost,...
    params,options,ei, data_train, labels_train);

%% compute accuracy on the test and train set
[~, ~, pred] = supervised_dnn_cost( opt_params, ei, data_test, [], true);
```

```
[~,pred] = max(pred);
acc_test = mean(pred'==labels_test);
fprintf('test accuracy: %f\n', acc_test*100);

[~, ~, pred] = supervised_dnn_cost( opt_params, ei, data_train, [], true);
[~,pred] = max(pred);
acc_train = mean(pred'==labels_train);
fprintf('train accuracy: %f\n', acc_train*100);

toc;
```

Código 2

Script	*load_preprocess_mnist.m*
Descripción	Carga los datos del set MNIST y hace las transformaciones pertinentes

```
function [data_train, labels_train, data_test, labels_test] = load_preprocess_mnist()
%% TODO ensure this is consistent with common loaders
% assumes relative paths to the common directory
% assumes common directory on paty for access to load functions
% adds 1 to the labels to make them 1-indexed

data_train = loadMNISTImages('../common/train-images-idx3-ubyte');
labels_train = loadMNISTLabels(['../common/train-labels-idx1-ubyte']);
labels_train  = labels_train + 1;

data_test = loadMNISTImages('../common/t10k-images-idx3-ubyte');
labels_test = loadMNISTLabels(['../common/t10k-labels-idx1-ubyte']);
labels_test = labels_test + 1;
```

Código 3

Script	*initialize_weights.m*
Descripción	Inicializa los pesos y bias, es decir, los parámetros de la red

```
function [ stack ] = initialize_weights( ei )
%INITIALIZE_WEIGHTS Random weight structures for a network architecture
%   eI describes a network via the fields layerSizes, inputDim, and outputDim
%
%   This uses Xavier's weight initialization tricks for better backprop
%   See: X. Glorot, Y. Bengio. Understanding the difficulty of training
%        deep feedforward neural networks. AISTATS 2010.

%% initialize hidden layers
stack = cell(1, numel(ei.layer_sizes));
for l = 1 : numel(ei.layer_sizes)
    if l > 1
        prev_size = ei.layer_sizes(l-1);
    else
        prev_size = ei.input_dim;
    end;
    cur_size = ei.layer_sizes(l);
    % Xaxier's scaling factor
```

```
    s = sqrt(6) / sqrt(prev_size + cur_size);
    stack{l}.W = rand(cur_size, prev_size)*2*s - s;
    stack{l}.b = zeros(cur_size, 1);
end
```

Código 4

Script	*stack2params.m*
Descripción	Convierte los parámetros de la forma pesos y bias a vector

```
function [params] = stack2params(stack)

% Converts a "stack" structure into a flattened parameter vector and also
% stores the network configuration. This is useful when working with
% optimization toolboxes such as minFunc.
%
% [params, netconfig] = stack2params(stack)
%
% stack - the stack structure, where stack{1}.w = weights of first layer
%                                    stack{1}.b = weights of first layer
%                                    stack{2}.w = weights of second layer
%                                    stack{2}.b = weights of second layer
%                                    ... etc.
% This is a non-standard version of the code to support conv nets
% it allows higher layers to have window sizes >= 1 of the previous layer
% If using a gpu pass inParams as your gpu datatype
% Setup the compressed param vector
params = [];


for d = 1:numel(stack)
    % This can be optimized. But since our stacks are relatively short, it
    % is okay
    params = [params ; stack{d}.W(:) ; stack{d}.b(:) ];

    % Check that stack is of the correct form
    assert(size(stack{d}.W, 1) == size(stack{d}.b, 1), ...
        ['The bias should be a *column* vector of ' ...
         int2str(size(stack{d}.W, 1)) 'x1']);
     % no layer size constrain with conv nets
     if d < numel(stack)
        assert(mod(size(stack{d+1}.W, 2), size(stack{d}.W, 1)) == 0, ...
            ['The adjacent layers L' int2str(d) ' and L' int2str(d+1) ...
             ' should have matching sizes.']);
     end
end

end
```

Código 5

Script	*params2stack.m*
Descripción	Convierte los parámetros de la formavectorial a pesos y bias

```
function stack = params2stack(params, ei)

% Converts a flattened parameter vector into a nice "stack" structure
% for us to work with. This is useful when you're building multilayer
% networks.
%
% stack = params2stack(params, netconfig)
%
% params - flattened parameter vector
% ei - auxiliary variable containing
%              the configuration of the network
%


% Map the params (a vector into a stack of weights)
depth = numel(ei.layer_sizes);
stack = cell(depth,1);
% the size of the previous layer
prev_size = ei.input_dim;
% mark current position in parameter vector
cur_pos = 1;

for d = 1:depth
    % Create layer d
    stack{d} = struct;

    hidden = ei.layer_sizes(d);
    % Extract weights
    wlen = double(hidden * prev_size);
    stack{d}.W = reshape(params(cur_pos:cur_pos+wlen-1), hidden, prev_size);
    cur_pos = cur_pos+wlen;

    % Extract bias
    blen = hidden;
    stack{d}.b = reshape(params(cur_pos:cur_pos+blen-1), hidden, 1);
    cur_pos = cur_pos+blen;

    % Set previous layer size
    prev_size = hidden;

end

end
```

Código 6

Script	*supervised_dnn_cost.m*
Descripción	Calcula la probabilidad de pertenecer a cada clase, la función coste y el gradiente

```
function [ cost, grad, pred_prob] = supervised_dnn_cost( theta, ei, data, labels, pred_only)
%SPNETCOSTSLAVE Slave cost function for simple phone net
%   Does all the work of cost / gradient computation
%   Returns cost broken into cross-entropy, weight norm, and prox reg
%        components (ceCost, wCost, pCost)

%% default values
po = false;
if exist('pred_only','var')
    po = pred_only;
end;

%% reshape into network
numHidden = numel(ei.layer_sizes) - 1;
numSamples = size(data, 2);
hAct = cell(numHidden+1, 1);
gradStack = cell(numHidden+1, 1);
stack = params2stack(theta, ei);
%% forward prop
for l = 1 : numHidden
    if l > 1
        hAct{l} = stack{l}.W * hAct{l - 1} + repmat(stack{l}.b, 1,numSamples);
    else
        hAct{l} = stack{l}.W * data + repmat(stack{l}.b, 1, numSamples);
    end
            hAct{l} = sigmoid(hAct{l});
end

l = numHidden+1;
y_hat = stack{l}.W * hAct{l - 1} + repmat(stack{l}.b, 1, numSamples);
y_hat = exp(y_hat);
hAct{l} = bsxfun(@rdivide, y_hat, sum(y_hat, 1));
[pred_prob pred_labels] = max(hAct{l});
pred_prob = hAct{l};

%% return here if only predictions desired.
if po
    cost = -1; ceCost = -1; wCost = -1; numCorrect = -1;
    grad = [];
    return;
end;

%% compute cost
y_hat = log(hAct{numHidden+1});
index = sub2ind(size(y_hat), labels', 1:numSamples);
ceCost = -sum(y_hat(index))./(size(data,2));

%% compute gradients using backpropagation
% Cross entroy gradient
targets = zeros(size(hAct{numHidden+1})); % numLabels * numSamples
targets(index) = 1;
gradInput = hAct{numHidden+1} - targets;

for l = numHidden+1 : -1 : 1
    if l > numHidden
        gradFunc = ones(size(gradInput));
    else
        gradFunc = hAct{l} .* (1 - hAct{l});
    end
    gradOutput = gradInput .* gradFunc;
    if l > 1
        gradStack{l}.W = gradOutput * hAct{l-1}'./(size(data,2));
    else
```

```
        gradStack{l}.W = gradOutput * data'./(size(data,2));
    end
    gradStack{l}.b = sum(gradOutput, 2)./(size(data,2));
    gradInput = stack{l}.W' * gradOutput;
end


%% compute weight penalty cost and gradient for non-bias terms
wCost = 0;
for l = 1:numHidden+1
    wCost = wCost + .5 * ei.lambda * sum(stack{l}.W(:) .^ 2);
end

cost = ceCost + wCost;

% Computing the gradient of the weight decay.
for l = numHidden : -1 : 1
    gradStack{l}.W = gradStack{l}.W + ei.lambda * stack{l}.W;
end


%% reshape gradients into vector
[grad] = stack2params(gradStack);
end
```

APÉNDICE V: CÓDIGOS DE LA RED NEURONAL CONVOLUCIONAL

Código 1

Script	*cnn_train.m*
Descripción	Carga los datos, crea una red neuronal convolucional, la entrena y muestra los resultados

```
clear all; clc; close all;
%% Convolution Neural Network Exercise

%  Instructions
%  ------------
%
%  This file contains code that helps you get started in building a single.
%  layer convolutional nerual network. In this exercise, you will only
%  need to modify cnnCost.m and cnnminFuncSGD.m. You will not need to
%  modify this file.

%%======================================================================
%% STEP 0: Initialize Parameters and Load Data
%  Here we initialize some parameters used for the exercise.
tic;
% Configuration
imageDim = 28;
numClasses = 10;  % Number of classes (MNIST images fall into 10 classes)
filterDim = 9;    % Filter size for conv layer
numFilters = 20;  % Number of filters for conv layer
poolDim = 2;      % Pooling dimension, (should divide imageDim-filterDim+1)

% Load MNIST Train
addpath ../common/;
images = loadMNISTImages('../common/train-images-idx3-ubyte');
images = reshape(images,imageDim,imageDim,[]);
labels = loadMNISTLabels('../common/train-labels-idx1-ubyte');
labels(labels==0) = 10; % Remap 0 to 10

% Initialize Parameters
theta = cnnInitParams(imageDim,filterDim,numFilters,poolDim,numClasses);

%%======================================================================
%% STEP 1: Implement convNet Objective
%  Implement the function cnnCost.m.

%%======================================================================
%% STEP 2: Gradient Check
%  Use the file computeNumericalGradient.m to check the gradient
%  calculation for your cnnCost.m function.  You may need to add the
%  appropriate path or copy the file to this directory.

DEBUG=true;  % set this to true to check gradient
if DEBUG
    % To speed up gradient checking, we will use a reduced network and
    % a debugging data set
    db_numFilters = 2;
    db_filterDim = 9;
    db_poolDim = 5;
    db_images = images(:,:,1:10);
    db_labels = labels(1:10);
    db_theta = cnnInitParams(imageDim,db_filterDim,db_numFilters,...
```

```
                db_poolDim,numClasses);

    [cost grad] = cnnCost(db_theta,db_images,db_labels,numClasses,...
                                db_filterDim,db_numFilters,db_poolDim);


    % Check gradients
    numGrad = computeNumericalGradient( @(x) cnnCost(x,db_images,...
                                db_labels,numClasses,db_filterDim,...
                                db_numFilters,db_poolDim), db_theta);

    diff = norm(numGrad-grad)/norm(numGrad+grad);
    % Should be small. In our implementation, these values are usually
    % less than 1e-9.
    disp(diff);

    assert(diff < 1e-9,...
        'Difference too large. Check your gradient computation again');

end;

%%======================================================================
%% STEP 3: Learn Parameters
%  Implement minFuncSGD.m, then train the model.

options.epochs = 3;
options.minibatch = 256;
options.alpha = 1e-1;
options.momentum = .95;

opttheta = minFuncSGD(@(x,y,z) cnnCost(x,y,z,numClasses,filterDim,...
                      numFilters,poolDim),theta,images,labels,options);

%%======================================================================
%% STEP 4: Test
%  Test the performance of the trained model using the MNIST test set. Your
%  accuracy should be above 97% after 3 epochs of training

testImages = loadMNISTImages('../common/t10k-images-idx3-ubyte');
testImages = reshape(testImages,imageDim,imageDim,[]);
testLabels = loadMNISTLabels('../common/t10k-labels-idx1-ubyte');
testLabels(testLabels==0) = 10; % Remap 0 to 10

[~,cost,preds]=cnnCost(opttheta,testImages,testLabels,numClasses,...
                filterDim,numFilters,poolDim,true);

acc = sum(preds==testLabels)/length(preds);

% Accuracy should be around 97.4% after 3 epochs
fprintf('Accuracy is %f\n',acc*100);
toc;
```

Código 2

Script	*cnnInitParams.m*
Descripción	Inicializa los parámetros de la red

```
function theta = cnnInitParams(imageDim,filterDim,numFilters,...
                               poolDim,numClasses)
% Initialize parameters for a single layer convolutional neural
% network followed by a softmax layer.
%
% Parameters:
%  imageDim   -  height/width of image
%  filterDim  -  dimension of convolutional filter
%  numFilters -  number of convolutional filters
%  poolDim    -  dimension of pooling area
%  numClasses -  number of classes to predict
%
%
% Returns:
%  theta      -  unrolled parameter vector with initialized weights

%% Initialize parameters randomly based on layer sizes.
assert(filterDim < imageDim,'filterDim must be less that imageDim');

Wc = 1e-1*randn(filterDim,filterDim,numFilters);

outDim = imageDim - filterDim + 1; % dimension of convolved image

% assume outDim is multiple of poolDim
assert(mod(outDim,poolDim)==0,...
       'poolDim must divide imageDim - filterDim + 1');

outDim = outDim/poolDim;
hiddenSize = outDim^2*numFilters;

% we'll choose weights uniformly from the interval [-r, r]
r  = sqrt(6) / sqrt(numClasses+hiddenSize+1);
Wd = rand(numClasses, hiddenSize) * 2 * r - r;

bc = zeros(numFilters, 1);
bd = zeros(numClasses, 1);

% Convert weights and bias gradients to the vector form.
% This step will "unroll" (flatten and concatenate together) all
% your parameters into a vector, which can then be used with minFunc.
theta = [Wc(:) ; Wd(:) ; bc(:) ; bd(:)];

end
```

Código 3

Script	*cnnCost.m*
Descripción	Calcula la probabilidad de pertenecer a cada clase, la función coste y el gradiente

```
function [cost, grad, preds] = cnnCost(theta,images,labels,numClasses,...
                                filterDim,numFilters,poolDim,pred)
% Calcualte cost and gradient for a single layer convolutional
% neural network followed by a softmax layer with cross entropy
% objective.
%
% Parameters:
%  theta      -  unrolled parameter vector
%  images     -  stores images in imageDim x imageDim x numImges
%                array
%  numClasses -  number of classes to predict
%  filterDim  -  dimension of convolutional filter
%  numFilters -  number of convolutional filters
%  poolDim    -  dimension of pooling area
%  pred       -  boolean only forward propagate and return
%                predictions
%
%
% Returns:
%  cost       -  cross entropy cost
%  grad       -  gradient with respect to theta (if pred==False)
%  preds      -  list of predictions for each example (if pred==True)


if ~exist('pred','var')
    pred = false;
end;


imageDim = size(images,1); % height/width of image
numImages = size(images,3); % number of images

%% Reshape parameters and setup gradient matrices

% Wc is filterDim x filterDim x numFilters parameter matrix
% bc is the corresponding bias

% Wd is numClasses x hiddenSize parameter matrix where hiddenSize
% is the number of output units from the convolutional layer
% bd is corresponding bias
[Wc, Wd, bc, bd] = cnnParamsToStack(theta,imageDim,filterDim,numFilters,...
                        poolDim,numClasses);

% Same sizes as Wc,Wd,bc,bd. Used to hold gradient w.r.t above params.
Wc_grad = zeros(size(Wc));
Wd_grad = zeros(size(Wd));
bc_grad = zeros(size(bc));
bd_grad = zeros(size(bd));

%%======================================================================
%% STEP 1a: Forward Propagation
%  In this step you will forward propagate the input through the
%  convolutional and subsampling (mean pooling) layers.  You will then use
%  the responses from the convolution and pooling layer as the input to a
%  standard softmax layer.

%% Convolutional Layer
%  For each image and each filter, convolve the image with the filter, add
%  the bias and apply the sigmoid nonlinearity.  Then subsample the
%  convolved activations with mean pooling.  Store the results of the
%  convolution in activations and the results of the pooling in
%  activationsPooled.  You will need to save the convolved activations for
%  backpropagation.
```

```
convDim = imageDim-filterDim+1; % dimension of convolved output
outputDim = (convDim)/poolDim; % dimension of subsampled output

% convDim x convDim x numFilters x numImages tensor for storing activations
activations = zeros(convDim,convDim,numFilters,numImages);

% outputDim x outputDim x numFilters x numImages tensor for storing
% subsampled activations
activationsPooled = zeros(outputDim,outputDim,numFilters,numImages);

W=ones(poolDim,poolDim);

for imageNum=1:numImages
    for filterNum=1:numFilters

        convolvedImage=zeros(convDim,convDim);
        filter=Wc(:,:,filterNum);
        filter=rot90(squeeze(filter),2);
        im=squeeze(images(:,:,imageNum));
        convolvedImage=conv2(im,filter,'valid');
        convolvedImage=logsig(convolvedImage+bc(filterNum));
        activations(:,:,filterNum,imageNum)=convolvedImage;

        pooledImage=zeros(outputDim,outputDim);
        convolved=conv2(convolvedImage,W,'valid');
        pooledImage=convolved(1:poolDim:end, 1:poolDim:end)./(poolDim^2);
        activationsPooled(:,:,filterNum,imageNum)=pooledImage;

    end
end

% Reshape activations into 2-d matrix, hiddenSize x numImages,
% for Softmax layer
activationsPooled = reshape(activationsPooled,[],numImages);

%% Softmax Layer
%  Forward propagate the pooled activations calculated above into a
%  standard softmax layer. For your convenience we have reshaped
%  activationPooled into a hiddenSize x numImages matrix.  Store the
%  results in probs.

% numClasses x numImages for storing probability that each image belongs to
% each class.
probs = zeros(numClasses,numImages);

softmax=Wd*activationsPooled+repmat(bd,1,numImages);
softmax=exp(softmax);
probs=bsxfun(@rdivide,softmax,sum(softmax));


%%=========================================================================
%% STEP 1b: Calculate Cost
%  In this step you will use the labels given as input and the probs
%  calculate above to evaluate the cross entropy objective.  Store your
%  results in cost.

cost = 0; % save objective into cost

A=log(probs);
ind=sub2ind(size(A),labels',1:numImages);
cost=-sum(A(ind))/numImages; % DIVIDED BY numImages??

% Makes predictions given probs and returns without backproagating errors.
if pred
    [~,preds] = max(probs,[],1);
    preds = preds';
    grad = 0;
    return;
end;

%%=========================================================================
%% STEP 1c: Backpropagation
%  Backpropagate errors through the softmax and convolutional/subsampling
%  layers.  Store the errors for the next step to calculate the gradient.
```

```
%  Backpropagating the error w.r.t the softmax layer is as usual.  To
%  backpropagate through the pooling layer, you will need to upsample the
%  error with respect to the pooling layer for each filter and each image.
%  Use the kron function and a matrix of ones to do this upsampling
%  quickly.

targets=zeros(size(probs));
targets(ind)=1;
errorsOutputSoftmax=(probs-targets)/numImages;  % DIVIDED BY numImages???

errorsInputSoftmax=Wd'*errorsOutputSoftmax;
errorsOutputPool=reshape(errorsInputSoftmax,outputDim,outputDim,numFilters,numImages);
errorsInputPool=zeros(convDim,convDim,numFilters,numImages);

for imageNum=1:numImages
    for filterNum=1:numFilters

errorsInputPool(:,:,filterNum,imageNum)=(kron(errorsOutputPool(:,:,filterNum,imageNum),ones(poolD
im)))/(poolDim^2);
    end
end

errorsInputConv=errorsInputPool.*activations.*(1-activations);


%%=====================================================================
%% STEP 1d: Gradient Calculation
%  After backpropagating the errors above, we can use them to calculate the
%  gradient with respect to all the parameters.  The gradient w.r.t the
%  softmax layer is calculated as usual.  To calculate the gradient w.r.t.
%  a filter in the convolutional layer, convolve the backpropagated error
%  for that filter with each image and aggregate over images.

Wd_grad=errorsOutputSoftmax*(activationsPooled');
bd_grad=sum(errorsOutputSoftmax,2);

for imageNum=1:numImages
    for filterNum=1:numFilters

        e=errorsInputConv(:,:,filterNum,imageNum);
        errorsInputConv(:,:,filterNum,imageNum)=rot90(e,2);

    end
end

for filterNum=1:numFilters
    for imageNum=1:numImages


Wc_grad(:,:,filterNum)=Wc_grad(:,:,filterNum)+conv2(images(:,:,imageNum),errorsInputConv(:,:,filt
erNum,imageNum),'valid');

    end
end

for filterNum=1:numFilters

    e=errorsInputConv(:,:,filterNum,:);
    bc_grad(filterNum)=sum(e(:));

end

%% Unroll gradient into grad vector for minFunc
grad = [Wc_grad(:) ; Wd_grad(:) ; bc_grad(:) ; bd_grad(:)];

end
```

Código 4

Script	*computeNumericalGradient.m*
Descripción	Calcula el gradiente de una segunda forma para verificar que el proporcionado por la función anterior es correcto

```
function numgrad = computeNumericalGradient(J, theta)
% numgrad = computeNumericalGradient(J, theta)
% theta: a vector of parameters
% J: a function that outputs a real-number. Calling y = J(theta) will return the
% function value at theta.

% Initialize numgrad with zeros
numgrad = zeros(size(theta));


%% Implement numerical gradient checking, and return the result in numgrad.
% You should write code so that numgrad(i) is (the numerical approximation to) the
% partial derivative of J with respect to the i-th input argument, evaluated at theta.
% I.e., numgrad(i) should be the (approximately) the partial derivative of J with
% respect to theta(i).
%
% You will probably want to compute the elements of numgrad one at a time.

epsilon = 1e-4;

for i =1:length(numgrad)
    oldT = theta(i);
    theta(i)=oldT+epsilon;
    pos = J(theta);
    theta(i)=oldT-epsilon;
    neg = J(theta);
    numgrad(i) = (pos-neg)/(2*epsilon);
    theta(i)=oldT;
    if mod(i,100)==0
       fprintf('Done with %d\n',i);
    end;
end;

end
```

Código 5

Script	*minFuncSGD.m*
Descripción	Entrena la CNN utilizando el algoritmo SGD con Momentum

```
function [opttheta] = minFuncSGD(funObj,theta,data,labels,...
                         options)
% Runs stochastic gradient descent with momentum to optimize the
% parameters for the given objective.
%
% Parameters:
%  funObj     -  function handle which accepts as input theta,
%                data, labels and returns cost and gradient w.r.t
%                to theta.
%  theta      -  unrolled parameter vector
%  data       -  stores data in m x n x numExamples tensor
%  labels     -  corresponding labels in numExamples x 1 vector
%  options    -  struct to store specific options for optimization
%
% Returns:
%  opttheta   -  optimized parameter vector
%
% Options (* required)
%  epochs*     - number of epochs through data
%  alpha*      - initial learning rate
%  minibatch*  - size of minibatch
%  momentum    - momentum constant, defualts to 0.9


%%======================================================================
%% Setup
assert(all(isfield(options,{'epochs','alpha','minibatch'})),...
        'Some options not defined');
if ~isfield(options,'momentum')
    options.momentum = 0.9;
end;
epochs = options.epochs;
alpha = options.alpha;
minibatch = options.minibatch;
m = length(labels); % training set size
% Setup for momentum
mom = 0.5;
momIncrease = 20;
velocity = zeros(size(theta));

%%======================================================================
%% SGD loop
it = 0;
for e = 1:epochs

    % randomly permute indices of data for quick minibatch sampling
    rp = randperm(m);

    for s=1:minibatch:(m-minibatch+1)
        it = it + 1;

        % increase momentum after momIncrease iterations
        if it == momIncrease
            mom = options.momentum;
        end;

        % get next randomly selected minibatch
        mb_data = data(:,:,rp(s:s+minibatch-1));
        mb_labels = labels(rp(s:s+minibatch-1));

        % evaluate the objective function on the next minibatch
        [cost grad] = funObj(theta,mb_data,mb_labels);
```

```
        % Add in the weighted velocity vector to the
        % gradient evaluated above scaled by the learning rate.
        % Then update the current weights theta according to the
        % sgd update rule

        velocity=mom*velocity+alpha*grad;
        theta=theta-velocity;

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

    % aneal learning rate by factor of two after each epoch
    alpha = alpha/2.0;

end;

opttheta = theta;

end
```

Código 6

Script	*cnnParamsToStack.m*
Descripción	Transforma los parámetros de la red en forma vector a pesos y bias

```
function [Wc, Wd, bc, bd] = cnnParamsToStack(theta,imageDim,filterDim,...
                                numFilters,poolDim,numClasses)
% Converts unrolled parameters for a single layer convolutional neural
% network followed by a softmax layer into structured weight
% tensors/matrices and corresponding biases
%
% Parameters:
%  theta      -  unrolled parameter vectore
%  imageDim   -  height/width of image
%  filterDim  -  dimension of convolutional filter
%  numFilters -  number of convolutional filters
%  poolDim    -  dimension of pooling area
%  numClasses -  number of classes to predict
%
%
% Returns:
%  Wc      -  filterDim x filterDim x numFilters parameter matrix
%  Wd      -  numClasses x hiddenSize parameter matrix, hiddenSize is
%             calculated as numFilters*((imageDim-filterDim+1)/poolDim)^2
%  bc      -  bias for convolution layer of size numFilters x 1
%  bd      -  bias for dense layer of size hiddenSize x 1

outDim = (imageDim - filterDim + 1)/poolDim;
hiddenSize = outDim^2*numFilters;

%% Reshape theta
indS = 1;
indE = filterDim^2*numFilters;
Wc = reshape(theta(indS:indE),filterDim,filterDim,numFilters);
indS = indE+1;
indE = indE+hiddenSize*numClasses;
Wd = reshape(theta(indS:indE),numClasses,hiddenSize);
indS = indE+1;
indE = indE+numFilters;
bc = theta(indS:indE);
bd = theta(indE+1:end);


end
```

APÉNDICE VI: CÓDIGOS EN PYTHON TRAS OPTIMIZACIÓN

Código 1

Script	*cnnTrain.py*
Descripción	Carga los datos, crea una red neuronal convolucional, la entrena y muestra los resultados

```
from __future__ import division
import os
import numpy as np
from pdb import set_trace as pb
import cnnInitParams
import cnnCost
import computeNumericalGradient
import minFuncSGD
import time
from numpy import linalg as LA
import struct

t=time.time()

""" Convolutional Neural Network """

# Instructions
# ------------
#
# This file contains code that helps you get started in building a single
# layer convolutional network. In this execirse, you will only
# need to modify cnnCost.py and minFuncSGD.py. You will not need to
# modify this file

""" ====================================================================== """
""" STEP 0: Initialize Parameters and Load Data """
# Here we initialize some parameters used for the exercise

# Configurtion
imageDim = 28  # Dimension of the input images
numClasses = 10  # Number of classes (MNIST images fall into 10 classes)
filterDim = 9  # Filter size for convolutional layer
numFilters = 20  # Numbre of filters for convolutional layer
poolDim = 2  # Pooling dimension, (should divide imageDim-filterDim+1)

# Load MNIST train
path="."
fname_img = os.path.join(path,'train-images-idx3-ubyte')
fname_lbl = os.path.join(path,'train-labels-idx1-ubyte')

with open(fname_img, 'rb') as fimg:
       magic, num, rows, cols = struct.unpack(">IIII",fimg.read(16))
       img = np.fromfile(fimg, dtype=np.uint8).reshape(-1,rows,cols)
       images = img/255

with open(fname_lbl, 'rb') as flbl:
       magic, num = struct.unpack(">II",flbl.read(8))
       labels = np.fromfile(flbl, dtype=np.int8)

# Initialize Parameters
theta=cnnInitParams.cnnInitParams(imageDim,filterDim,numFilters,poolDim,numClasses)

""" ====================================================================== """
""" STEP 1: Implement ConvNet Objective """
```

```
# Implement the funcion cnnCost.py

""" ====================================================================== """
""" STEP 2: Gradient Check """
# Use the file computeNumericalGradient.py to check the gradient
# calculation for you cnnCost.py function. You may need to add the
# appropiate path or copy the file to this directory

DEBUG=True  # Set this to true to check the gradient

if DEBUG:
# To speed up gradient checking, we will use a reduced network and
# a debugging data set

        print "Starting Gradient Check"

        db_numFilters = 2
        db_filterDim = 9
        db_poolDim = 5
        db_images = images[0:10][:][:]
        db_labels = labels[0:10]
        db_theta =
cnnInitParams.cnnInitParams(imageDim,db_filterDim,db_numFilters,db_poolDim,numClasses)

        cost, grad, preds =
cnnCost.cnnCost(db_theta,db_images,db_labels,numClasses,db_filterDim,db_numFilters,db_poolDim,Fal
se)


# Check gradients
        numGrad = computeNumericalGradient.computeNumericalGradient(lambda x: cnnCost.cnnCost(x,
db_images,db_labels,numClasses,db_filterDim,db_numFilters,db_poolDim,False),db_theta)
        diff = LA.norm(numGrad-grad)/LA.norm(numGrad+grad)

# Should be small. In our implementation, these values are usually
# less than 1e-9.
        print "Difference is",diff

        assert(diff < 1e-9),"Difference too large. Check your gradient computation again"

        elapsed=time.time()-t
        print "Elapsed time in checking is",elapsed,"seconds"

""" ====================================================================== """
""" STEP 3: Learn Parameters """
# Implement minFuncSGD.py, then train the model
print "Starting Training"

options = np.zeros((4,1))
options[0] = 3
options[1] = 256
options[2] = 1e-1
options[3] = 0.95

opttheta = minFuncSGD.minFuncSGD(lambda x, y, z:
cnnCost.cnnCost(x,y,z,numClasses,filterDim,numFilters,poolDim,False),theta,images,labels,options)

print "The sum of theta is:",np.sum(opttheta)

""" ====================================================================== """
""" STEP 4: Test """
# Test the performance of the trained model using the MNIST test set. Your
# accuracy should be above 97 after 3 epochs of training
print "Testing trained network"

"""testImages, testLabels = mndata.load_testing()
testImages = np.reshape(testImages,(-1,imageDim,imageDim))
testImages = testImages/255"""

fname_img = os.path.join(path,'t10k-images-idx3-ubyte')
fname_lbl = os.path.join(path,'t10k-labels-idx1-ubyte')

with open(fname_img, 'rb') as fimg:
        magic, num, rows, cols = struct.unpack(">IIII",fimg.read(16))
        img = np.fromfile(fimg, dtype=np.uint8).reshape(-1,rows,cols)
        testImages = img/255

with open(fname_lbl, 'rb') as flbl:
        magic, num = struct.unpack(">II",flbl.read(8))
```

```
        testLabels = np.fromfile(flbl, dtype=np.int8)

_, cost, preds =
cnnCost.cnnCost(opttheta,testImages,testLabels,numClasses,filterDim,numFilters,poolDim,True)
acc = np.sum(preds==testLabels)/(np.shape(testLabels)[0])

# Accuracy should be around 97.4% after 3 epochs
print "Accuracy is",acc*100,"%"
print "The number of successfully predicted images is",np.sum(preds==testLabels)

elapsed=time.time()-t
print "Elapsed time is",elapsed/60,"minutes"

if acc>0.97:
        print "Saving optimized theta in file savedata.txt"
        np.savetxt('savedata.txt',opttheta)

print "Done"
```

Código 2

Script	*cnnInitParams.py*
Descripción	Inicializa los parámetros de la red

```
import math
import numpy as np

def cnnInitParams(imageDim,filterDim,numFilters,poolDim,numClasses):

# Initialize parameters for a single layer convolutional neural
# network followed by a softmax layer
#
# Parameters:
#   imageDim    -  height/width of image
#   filterDim   -  dimension of convolutional filter
#   numFilters  -  number of convolutional filters
#   poolDim     -  dimension of pooling area
#   numClasses  -  number of classes to predict
#
#
# Returns:
#   theta      -  unrolled parametervector with initialized weights

        """ Initialize parameters randomly based on layer sizes """
        assert (filterDim < imageDim),"filterDim must be less than imageDim"

        Wc = 1e-1*np.random.normal(0,1,(numFilters,filterDim,filterDim))

        outDim = imageDim - filterDim + 1  # dimension of convolved image

# Assume outDim is multiple of poolDim
        assert (outDim % poolDim == 0),"poolDim must divide outDim"

        outDim = outDim/poolDim
        hiddenSize = outDim**2*numFilters

# We'll choose weights uniformly from the interval [-r, r]
        r=math.sqrt(6)/math.sqrt(numClasses+hiddenSize+1)
        Wd=np.random.uniform(0,1,(numClasses,hiddenSize))*2*r-r

        bc=np.zeros((numFilters,1))
        bd=np.zeros((numClasses,1))

# Convert weights and bias gradients to the vector form.
# This step will "unroll" (flatten and concatenate together) all
# your parameters into a vector, which can then be used with minFunc
        Wc=np.reshape(Wc,(-1,1))
```

```
        Wd=np.reshape(Wd,(-1,1))
        bc=np.reshape(bc,(-1,1))
        bd=np.reshape(bd,(-1,1))

        theta=np.append(Wc,Wd)
        theta=np.append(theta,bc)
        theta=np.append(theta,bd)
        theta=np.reshape(theta,(-1,1))

        return theta
```

Código 3

Script	*cnnCost.py*
Descripción	Calcula la probabilidad de pertenecer a cada clase, la función coste y el gradiente

```
from __future__ import division
import numpy as np
import scipy
from scipy import signal
import sigmoid
import cnnParamsToStack
from pdb import set_trace as pb
from numpy import linalg as LA
import sys

def cnnCost(theta,images,labels,numClasses,filterDim,numFilters,poolDim,pred):

# Calcualte cost and gradient for a single layer convolutional
# neural network followed by a softmax layer with cross entropy
# objective.
#
# Parameters:
#  theta      -  unrolled parameter vector
#  images     -  stores images in imageDim x imageDim x numImges
#                array
#  numClasses -  number of classes to predict
#  filterDim  -  dimension of convolutional filter
#  numFilters -  number of convolutional filters
#  poolDim    -  dimension of pooling area
#  pred       -  boolean only forward propagate and return
#                predictions
#
#
# Returns:
#  cost       -  cross entropy cost
#  grad       -  gradient with respect to theta (if pred==False)
#  preds      -  list of predictions for each example (if pred==True)


        imageDim = np.shape(images)[1]  # height/width of image
        numImages = np.shape(images)[0] # number of images

        """ Reshape parameters and setup gradient matrices """

# Wc is filterDim x filterDim x numFilters parameter matrix
# bc is the corresponding bias

# Wd is numClasses x hiddenSize parameter matrix where hiddenSize
# is the number of output units from the convolutional layer
# bd is corresponding bias
        Wc, Wd, bc, bd =
cnnParamsToStack.cnnParamsToStack(theta,imageDim,filterDim,numFilters,poolDim,numClasses)

# Same sizes as Wc,Wd,bc,bd. Used to hold gradient w.r.t above params.
        Wc_grad = np.zeros(np.shape(Wc))
```

```
        Wd_grad = np.zeros(np.shape(Wd))
        bc_grad = np.zeros(np.shape(bc))
        bd_grad = np.zeros(np.shape(bd))

        """ ---------- YOUR CODE HERE -------------------------------------- """
        """ STEP 1a: Forward Propagation """
#  In this step you will forward propagate the input through the
#  convolutional and subsampling (mean pooling) layers.  You will then use
#  the responses from the convolution and pooling layer as the input to a
#  standard softmax layer.

        """ Convolutional Layer """
#  For each image and each filter, convolve the image with the filter, add
#  the bias and apply the sigmoid nonlinearity.  Then subsample the
#  convolved activations with mean pooling.  Store the results of the
#  convolution in activations and the results of the pooling in
#  activationsPooled.  You will need to save the convolved activations for
#  backpropagation.
        convDim = imageDim-filterDim+1 # dimension of convolved output
        outputDim = (convDim)//poolDim # dimension of subsampled output

# convDim x convDim x numFilters x numImages tensor for storing activations
        activations = np.zeros((numImages,numFilters,convDim,convDim))

# outputDim x outputDim x numFilters x numImages tensor for storing
# subsampled activations
        activationsPooled = np.zeros((numImages,numFilters,outputDim,outputDim))

        """ YOUR CODE HERE """

        W = np.ones((poolDim,poolDim))
        ind=range(convDim)
        ind=ind[0:None:poolDim]

        for imageNum in xrange(numImages):

                for filterNum in xrange(numFilters):
                        convolvedImage = np.zeros((convDim,convDim))
                        filter = Wc[filterNum][:][:]
                        filter = np.rot90(np.rot90(filter))
                        im = images[:][:][imageNum]
                        convolvedImage = scipy.signal.convolve2d(im,filter,'valid')
                        convolvedImage =
convolvedImage+np.tile(bc[filterNum],np.shape(convolvedImage))
                        convolvedImage = sigmoid.sigmoid(convolvedImage)
                        activations[imageNum][filterNum][:][:] = convolvedImage

                        pooledImage = np.zeros((outputDim,outputDim))
                        filter = W
                        im = convolvedImage
                        convolved = scipy.signal.convolve2d(im,filter,'valid')/(poolDim**2)
                        pooledImage=convolved[ind].swapaxes(0,1)[ind].swapaxes(0,1)
                        activationsPooled[imageNum][filterNum][:][:] = pooledImage

# Reshape activations into 2-d matrix, hiddenSize x numImages,
# for Softmax layer
        activationsPooledAfter=np.zeros((outputDim**2*numFilters,numImages))
        activationsPooledAfter=activationsPooled.swapaxes(0,2).swapaxes(1,3).swapaxes(2,3).reshap
e((-1,numImages),order='F')

        """ Softmax Layer """
#  Forward propagate the pooled activations calculated above into a
#  standard softmax layer. For your convenience we have reshaped
#  activationPooled into a hiddenSize x numImages matrix.  Store the
#  results in probs.

# numClasses x numImages for storing probability that each image belongs to
# each class.
        probs = np.zeros((numClasses,numImages))
        softmax=np.dot(Wd,activationsPooledAfter)+np.tile(bd,(1,numImages))
        softmax=np.exp(softmax)
        probs=softmax/np.sum(softmax,axis=0)


        """ ---------- YOUR CODE HERE -------------------------------------- """
        """ STEP 1b: Calculate Cost """
#  In this step you will use the labels given as input and the probs
#  calculate above to evaluate the cross entropy objective.  Store your
#  results in cost.
```

```
        cost = 0 # save objective into cost

        A=np.log(probs)
        p=np.zeros((1,numImages))

        for imageNum in xrange(numImages):
                p[0][imageNum]=A[labels[imageNum]][imageNum]

        cost=-np.sum(p)/numImages

# Makes predictions given probs and returns without backproagating errors.
        if pred:
                preds=np.argmax(probs,axis=0)
                grad = 0
                return cost, grad, preds


        """ ================================================================== """
        """ STEP 1c: Backpropagation """
#  Backpropagate errors through the softmax and convolutional/subsampling
#  layers.  Store the errors for the next step to calculate the gradient.
#  Backpropagating the error w.r.t the softmax layer is as usual.  To
#  backpropagate through the pooling layer, you will need to upsample the
#  error with respect to the pooling layer for each filter and each image.
#  Use the kron function and a matrix of ones to do this upsampling
#  quickly.

        targets=np.zeros(np.shape(probs))

        for imageNum in xrange(numImages):
                targets[labels[imageNum]][imageNum] = 1

        errorsOutputSoftmax=(probs-targets)/numImages

        errorsInputSoftmax=np.dot(Wd.T,errorsOutputSoftmax)
        errorsOutputPool=errorsInputSoftmax.reshape((outputDim,outputDim,numFilters,numImages),or
der='F').swapaxes(0,2).swapaxes(1,3).swapaxes(0,1)
        errorsInputPool=np.zeros((numImages,numFilters,convDim,convDim))

        for imageNum in xrange(numImages):
                for filterNum in xrange(numFilters):

        errorsInputPool[imageNum][filterNum][:][:]=np.kron(errorsOutputPool[imageNum][filterNum][
:][:],np.ones((poolDim,poolDim)))/poolDim**2


        errorsInputConv=errorsInputPool*activations*(1-activations)


        """ ================================================================== """
        """ STEP 1d: Gradient Calculation """
#  After backpropagating the errors above, we can use them to calculate the
#  gradient with respect to all the parameters.  The gradient w.r.t the
#  softmax layer is calculated as usual.  To calculate the gradient w.r.t.
#  a filter in the convolutional layer, convolve the backpropagated error
#  for that filter with each image and aggregate over images.

        Wd_grad = np.dot(errorsOutputSoftmax,activationsPooledAfter.T)
        bd_grad = np.sum(errorsOutputSoftmax,axis=1)

        for imageNum in xrange(numImages):
                for filterNum in xrange(numFilters):
                        e = errorsInputConv[imageNum][filterNum][:][:]
                        errorsInputConv[imageNum][filterNum][:][:] = np.rot90(np.rot90(e))

        for filterNum in xrange(numFilters):
                for imageNum in xrange(numImages):
                        Wc_grad[filterNum][:][:] =
Wc_grad[filterNum][:][:]+scipy.signal.convolve2d(images[imageNum][:][:],errorsInputConv[imageNum]
[filterNum][:][:],'valid')


        for filterNum in xrange(numFilters):
                for imageNum in xrange(numImages):
                        e = errorsInputConv[imageNum][filterNum][:][:]
                        bc_grad[filterNum] += np.sum(e)

        """ Unroll gradient into grad vector for minFunc """
```

```
        Wc_grad = Wc_grad.swapaxes(0,1).swapaxes(1,2).reshape((-1,1),order='F')
        Wd_grad = np.reshape(Wd_grad,(-1,1),order='F')
        bc_grad = np.reshape(bc_grad,(-1,1),order='F')
        bd_grad = np.reshape(bd_grad,(-1,1),order='F')

        grad = np.append(Wc_grad,Wd_grad)
        grad = np.append(grad,bc_grad)
        grad = np.append(grad,bd_grad)
        grad = np.reshape(grad,(-1,1))

        return cost, grad, probs
```

Código 4

Script	*computeNumericalGradient.py*
Descripción	Calcula el gradiente de una segunda forma para verificar que el proporcionado por la función anterior es correcto

```
from __future__ import division
from pdb import set_trace as pb
import numpy as np
# import cnnCost

#def computeNumericalGradient(db_images,db_labels,theta):
def computeNumericalGradient(J,theta):
# numgrad = computeNumericalGradient(J, theta)
# theta: a vector of parameters
# J: a function that outputs a real-number. Calling y = J(theta) will return the
# function value at theta.

# Initialize numgrad with zeros
        numGrad = np.zeros(np.shape(theta))

        length = np.shape(numGrad)[0]

#
# Implement numerical gradient checking, and return the result in numgrad.
# You should write code so that numgrad(i) is (the numerical approximation to) the
# partial derivative of J with respect to the i-th input argument, evaluated at theta.
# I.e., numgrad(i) should be the (approximately) the partial derivative of J with
# respect to theta(i).
#
# You will probably want to compute the elements of numgrad one at a time.

        epsilon = 1e-4

        for i in xrange(length):
                oldT = theta[i]
                theta[i]=oldT+epsilon
                oldT=theta[i]-epsilon
                pos, _, _ = J(theta)
                theta[i]=oldT-epsilon
                oldT=theta[i]+epsilon
                neg, _, _ = J(theta)
                numGrad[i] = (pos-neg)/(2*epsilon)
                theta[i]=oldT
                if (i%100 == 0):
                        print "Done with",i

        """ ----------------------------------------------------------------- """

        return numGrad
```

Código 5

Script	*minFuncSGD.py*
Descripción	Entrena la CNN utilizando el algoritmo SGD con Momentum

```
from __future__ import division
import numpy as np
from pdb import set_trace as pb

def minFuncSGD(funObj,theta,data,labels,options):
# Runs stochastic gradient descent with momentum to optimize the
# parameters for the given objective
#
# Parameters:
#       funcObj     - function handle which accepts as input theta,
#                     data, labels, and returns cost and gradient w.r.t.
#                     to theta-
#       theta       - unrolled parameter vector
#       data        - stores data in [numExamples x M x N] tensor
#       labels      - corresponding labels in [numExamples x 1] vector
#       options     - array to store speciic options for optimization
#
# Returns:
#       opttheta    - optimized parameter vector
#
# Options (* required)
#       epochs*     - number of epochs through data
#       alpha*      - initial learning rate
#       minibatch*  - size of minibatch
#       momentum    - momentum constant, defaults to 0.9

        """ ================================================================ """
        """ Setup """
        assert (options[0]!=0 and options[1]!=0 and options[2]!=0), "Some options not defined"
        if (options[3]==0):
                options[3] = 0.9

        epochs = np.int_(options[0])
        minibatch = np.int_(options[1])
        alpha = options[2]
        m = np.shape(labels)[0]
        loopopt = np.array(range(m-minibatch))
        # Setup for momentum
        mom = 0.5
        momIncrease = 20
        velocity = np.zeros(np.shape(theta))

        """ ================================================================ """
        """ SGD loop """
        it = 0
        for e in xrange(epochs):

                # randomly permute indices of data for quick minibatch sampling
                rp = np.random.permutation(m)

                for s in loopopt[0:None:minibatch]:
                        it = it+1

                        # Increase momentumafter momIncrease iterations
                        if (it==momIncrease):
                                mom = options[3]

                        # get next randomly selected minibatch
                        ind=rp[s:s+minibatch]
                        mb_data = data[ind][:][:]
                        labels = np.array(labels)
                        mb_labels = labels[ind]

                        # Evaluate the objective function on the next minibatch
                        cost, grad, _ = funObj(theta,mb_data,mb_labels)
```

```
                        # Add in the weighted velocity vector to the
                        # gradient evaluated above scaled by the learning rate.
                        # The update the current weights theta according to the
                        # sgd pdate rule

                        velocity = mom*velocity+alpha*grad
                        theta=theta-velocity

                        print "Epoch", e+1, ": Cost on iteration", it, "is", cost

                # Aneal learning rate by factor of two after each epoch
                alpha = alpha/2

        opttheta=theta
        return opttheta
```

Código 6

Script	*cnnParamsToStack.py*
Descripción	Transforma los parámetros de la red en forma vector a pesos y bias

```
import numpy as np

def cnnParamsToStack(theta,imageDim,filterDim,numFilters,poolDim,numClasses):
# Converts unrolled parameters for a single layer convolutional neural
# network followed by a softmax layer into structured weight
# tensors/matrices and corresponding biases
#
# Parameters:
#  theta      -  unrolled parameter vectore
#  imageDim   -  height/width of image
#  filterDim  -  dimension of convolutional filter
#  numFilters -  number of convolutional filters
#  poolDim    -  dimension of pooling area
#  numClasses -  number of classes to predict
#
#
# Returns:
#  Wc      -  filterDim x filterDim x numFilters parameter matrix
#  Wd      -  numClasses x hiddenSize parameter matrix, hiddenSize is
#             calculated as numFilters*((imageDim-filterDim+1)/poolDim)^2
#  bc      -  bias for convolution layer of size numFilters x 1
#  bd      -  bias for dense layer of size hiddenSize x 1

        outDim = (imageDim - filterDim + 1)/poolDim
        hiddenSize = outDim**2*numFilters

        """ Reshape theta """
        indS = 0
        indE = filterDim**2*numFilters
        Wc=np.zeros((numFilters,filterDim,filterDim))
        Wc=theta[indS:indE].reshape((filterDim,filterDim,numFilters),order='F').swapaxes(0,2).swa
paxes(1,2)

        indS = indE
        indE = indE+hiddenSize*numClasses
        Wd = np.reshape(theta[indS:indE],(numClasses,hiddenSize),order='F')

        indS = indE
        indE = indE+numFilters
        bc = theta[indS:indE]

        bd = theta[indE:None]

        return Wc,Wd,bc,bd
```