UNIVERSIDAD
DE GRANADA

Facultad de Ciencias

Grado en Matemáticas

TRABAJO DE FIN DE GRADO

Introducción a las redes neuronales para el tratamiento de imágenes

Presentado por:
Samuel Santillana Quesada

Tutor:
José Alfredo Cañizo Rincón
Departamento de Matemática Aplicada

Curso académico 2021-2022

Introducción a las redes neuronales para el tratamiento de imágenes

Samuel Santillana Quesada

Samuel Santillana Quesada *Introducción a las redes neuronales para el tratamiento de imágenes*.
Trabajo de fin de Grado. Curso académico 2021-2022.

Responsable de tutorización

José Alfredo Cañizo Rincón
Departamento de Matemática Aplicada

Grado en Matemáticas
Facultad de Ciencias
Universidad de Granada

A Luisa Caba Colmenero, mi profesora de matemáticas de bachillerato, por haberme hecho ver que las matemáticas son mi camino.

Índice general

Agradecimientos xi

Summary xiii

Introducción xv

1 Las Redes Neuronales 1

1.1 Motivación . . . 1
1.2 Conceptos básicos y contexto histórico . . . 3
1.3 Aplicaciones de las redes neuronales . . . 6
1.4 La función coste . . . 7
1.4.1 Las dos hipótesis a asumir sobre la función coste. . . . 8
1.4.2 Problema de minimización de la función coste . . . 9

2 Herramientas para minimizar la función coste: Descenso gradiente y Backpropagation. 11

2.1 El Descenso Gradiente . . . 11
2.1.1 Introducción . . . 11
2.1.2 Convergencia del Método Descenso Gradiente. . . . 15
2.1.3 Resultados de convergencia del Método Descenso Gradiente . . . 16
2.1.4 Introducción del Descenso Gradiente Estocástico . . . 19
2.1.5 Introducción sobre el Optimizador Adam . . . 20
2.2 El backpropagation. . . . 21
2.2.1 Notación. . . . 22
2.2.2 El producto Hadamard . . . 23
2.2.3 Las 4 ecuaciones fundamentales del backpropagation. . . . 23
2.2.4 El algoritmo del backpropagation . . . 27

3 Redes Neuronales Artificiales en el tratamiento de imágenes 29

3.1 Salida de la Red Neuronal Convolucional. . . . 32
3.2 Aplicación de redes convolucionales en la clasificación de imágenes de ropa. . 33
3.2.1 Ejemplo experimental de clasificación de ropa. . . . 36
3.2.2 Predicción de camiseta . . . 38
3.2.3 Predicción de pantalones . . . 39

4 Redes Neuronales en el reconocimiento de caracteres manuscritos 41

Bibliografía 47

Agradecimientos

En primer lugar, querría agradecérselo a mi tutor, José Alfredo Cañizo Rincón. Sin su dedicación, constancia y apoyo no habría sido posible lograr algunos de los objetivos previstos para el TFG.

En segundo lugar, a mi familia, tanto mis padres como mi hermano, por haberme apoyado en la decisión de introducirme al grado de Matemáticas y por haberme animado en los momentos duros del grado.

Y en último lugar, a mi mejor amigo, Jorge Pérez (Ceporrido). Sin duda, tu apoyo ha sido fundamental, sin ti no hubiese logrado todo lo que he conseguido. Gracias por haberme animado en los momentos difíciles y haber compartido conmigo los momentos de alegría, gracias por haberme escuchado y ayudado en todo lo que he necesitado y sobretodo, por seguir siendo mi mejor amigo. Por todo ello, muchas gracias Cepo.

Summary

Nowadays, neural networks have represented a great step forward in the area of computing and artificial intelligence. We have been provided with different ways to, through mathematics in conjunction with some powerful computational tools, recognize patterns in images, classify images or even solve optimization problems. For this reason, this Final Degree Project approaches this field of neural networks through mathematical tools, despite the fact that it uses the necessary computation for its proper function. Also, the area of neural networks is still booming and research continues to maximize their use, although it has been proven that they are quite useful in all scientific and non scientific areas.

In chapter 1 we will explain advances in current neural networks and examples of research articles in which you can appreciate the latest on neural networks (in this case generative adversarial networks that will be introduced later). In addition, we explain the historical process of neural networks and a way to compute the error produced by the neural network using a error function. Like any error function, the ultimate aim is to minimize a certain error function and find what values the variables should have so that the error is the minimum. This problem cannot be solved by basic optimization techniques and, therefore, more advanced optimization methods should be used, such as iterative methods like gradient descent, stochastic gradient descent or Adam method.

In chapter 2, we will go deeper into these optimization methods and we will see certain convergence results along with their respective proofs. These optimization methods have a disadvantage; they need to know the gradient of the function to which it is applied. For this reason, the backpropagation algorithm has also been detailed, which is an iterative method that will provide us with the partial derivatives of the error function with respect to any weight and bias of the neural network. To do this, the 4 fundamental equations of backpropagation will be enunciated and demonstrated. Finally, it will be explained how they are used to find the gradient of the function.

In chapter 3, we will introduce convolutional neural networks and explain them through a practical example. In addition, we will use a code from Tensorflow in Python programming language to be able to classify clothing images using convolutional neural networks.

Finally, in chapter 4, a Python code from the book Neural Networks and Deep Learning will be used to recognize handwritten characters through artificial neural networks and to see, in practice, the theory explained in Chapters 1 and 2. Furthermore, we will be able to check how the neural network is able to train itself automatically and to produce a better output with the least error possible.

Introducción

En la actualidad, las redes neuronales han supuesto un gran avance en el mundo de la computación y de la Inteligencia Artificial. Nos han proporcionado diversas maneras de, a través de las matemáticas junto con alguna herramienta computacional potente, reconocer patrones en imágenes, clasificar imágenes o incluso optimizar problemas. Además, el área de las redes neuronales sigue en pleno auge y se siguen realizando investigaciones para poder maximizar el uso de ellas, aunque se ha demostrado que son de bastante utilidad en todos los campos.

Por ello, este TFG se acerca a este campo de las redes neuronales mediante herramientas matemáticas, a pesar de que emplea la computación necesaria para su funcionamiento.

A continuación voy a exponer los objetivos iniciales planteados junto con su cumplimiento, indicando en qué capítulo se ha logrado.

- Analizar los tipos de redes neuronales y explicar la diferencia entre ellos junto con su aplicación en la actualidad.
- Explicar el concepto de función coste y presentar la dificultad que presenta minimizarla junto con una explicación detallada de herramientas de minimización como el Descenso Gradiente.
- Definir el algoritmo del backpropagation, enunciar y demostrar sus 4 ecuaciones fundamentales.
- Representar y programar una red neuronal artificial capaz de reconocer caracteres manuscritos, capaz de presentar su error y poder entrenarse.
- Programar una red neuronal convolucional que pueda clasificar imágenes de ropa.

Posteriormente, se va a explicar qué se ha estudiado en cada capítulo.

En el capítulo 1 explicaremos avances de las redes neuronales actuales y ejemplos de artículos de investigación en el que se puede apreciar visualmente lo más reciente acerca de redes neuronales (en este caso las redes generativas adversarias que se introducirán más adelante). Además, explicamos el proceso histórico que han llevado las redes neuronales y una manera de computar el error que produce la red neuronal mediante una función coste. Como toda función coste, el objetivo final es minimizarla y encontrar qué valores deberían tener las variables para que el error sea el mínimo. Este problema no se puede resolver mediante técnicas básicas de optimización y por ello hay que usar métodos de optimización más avanzados como son los métodos iterativos como el descenso gradiente, descenso gradiente estocástico o método Adam.

En el capítulo 2 profundizamos acerca de estos métodos de optimización y veremos ciertos resultados de convergencia junto con sus respectivas demostraciones. Estos métodos de optimización tienen una desventaja: necesitan conocer el gradiente de la función a la que se le aplica. Por ello, también se ha detallado el algoritmo de backpropagation, que es un método iterativo el cual nos proporcionará las derivadas parciales de la función coste respecto a cualquier peso y sesgo de la red neuronal. Para ello, se enunciarán y demostrarán las 4

ecuaciones fundamentales del backpropagation y finalmente se explicará como se utilizan para hallar el gradiente de la función.

En el capítulo 3 introduciremos las redes neuronales convolucionales y las explicaremos mediante un ejemplo práctico. Además, utilizaremos un código en el lenguaje de programación *Python* de *Tensorflow* para poder clasificar imágenes de ropa mediante redes neuronales convolucionales. Además, he adaptado el código para poder introducir imágenes de prendas de vestir propias y poder visualizar si el resultado es satisfactorio o no.

Finalmente, en el capítulo 4 se utilizará un código en *Python* del libro *Neural Networks and Deep Learning* para poder reconocer caracteres mediante redes neuronales artificiales y poder ver, aplicado en la práctica, toda la teoría explicada en los Capítulos 1 y 2. Además, podremos comprobar cómo la red neuronal es capaz de entrenarse automáticamente y poder producir una mejor salida.

1 Las Redes Neuronales

En este capítulo explicaremos la importancia de las redes neuronales en la actualidad junto con las definiciones de los elementos que componen las redes neuronales. De esta manera, podremos entender el motivo por el que explicamos métodos de optimización y los procesos que se siguen en el ámbito de la inteligencia artificial para poder mejorar las redes neuronales.

1.1. Motivación

En el ámbito de la Inteligencia Artificial, desde hace bastante tiempo se han creado redes neuronales para identificar objetos. Un ejemplo de ello es cómo funciona *Google Fotos* [3], que reconoce qué objetos y qué personas aparecen en las imágenes registradas. Incluso la aplicación *Cámara* de cualquier dispositivo móvil actual posee redes neuronales, las cuales en la imagen que están enfocando reconocen qué parte es la cara de las personas y, con ello, enfoca con mayor precisión.

Otro ejemplo de ello es el reconocimiento de huellas, en el cual introduces tu huella a un dispositivo y este dispositivo reconoce si es tuya o es de otra persona y así darte acceso a cierto servicio.

Incluso existen investigaciones actuales cuya finalidad es crear una red neuronal e introducirla en una cámara de vigilancia y que esta reconozca si una persona va armada o va desarmada.

Como se ha visto, estas redes neuronales se pueden utilizar en la mayoría de campos científicos, ¡y no sólo científicos! Así que, por ello, he decidido investigar este campo.

Todos estos problemas tienen algo en común: se le introduce un conjunto de datos y estos dan una respuesta al problema. Por ello, numerosos científicos se han cuestionado si es posible crear imágenes que parezcan reales a partir de cierto tema. Por ejemplo, se ha cuestionado si es posible crear una red neuronal que produzca imágenes de gatos que parezcan reales.

Este problema es una de las investigaciones más recientes y se basa en matemáticas bastante avanzadas (junto con la ayuda de herramientas de computación potentes).

Consisten en dos redes neuronales: la red neuronal generadora y la red neuronal discriminadora. La red neuronal generadora consiste en una red neuronal que se encarga de generar imágenes que parecen reales y la red neuronal discriminadora evalúa si la imagen generada parece real o falsa (Véase [14]). Veámoslo con el ejemplo de la red neuronal capaz de crear imágenes de gatos:

Ejemplo 1.1. Imaginemos que queremos crear una red neuronal que genere imágenes de gatos que parezcan reales. Consistirá en dos redes neuronales:

- La red neuronal generadora que se encargará de crear imágenes de gatos que parezcan reales.
- La red neuronal discriminadora que comparará las imágenes creadas por la red neuronal generadora y una base de datos de imágenes de gatos reales y evaluará si las

imágenes generadas parecen reales o no.

Este proceso se repite una cantidad finita de pasos hasta que la red neuronal discriminadora no sea capaz de distinguir la imagen creada de las imágenes reales.

Veamos un ejemplo en el siguiente gráfico

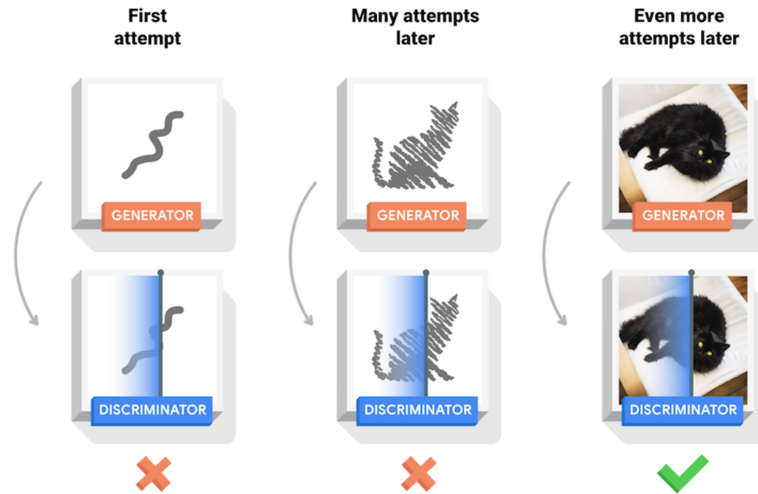


Figura 1.1: Red GAN generando imágenes de gatos. Imagen obtenida en [14]

A esta red neuronal capaz de crear imágenes se le llama *Red Neuronal Adversaria Generativa (GAN)*.

Además de la creación de imágenes a partir de un tema, también se utilizan para transformar imágenes de cierta temática en otra imagen de otra temática diferente. Por ejemplo, se utilizan para transformar imágenes de personas adultas en imágenes de bebés o transformar imágenes de pinturas de Monet en pinturas de Van Gogh, etc.

Un estudio bastante reciente es de los científicos Harshad Rai, Naman Shukla y Ziyu Zhou, los cuales han conseguido generar una red GAN la cual cambie imágenes de caballos en cebras (y viceversa), cambie imágenes de paisajes de verano a invierno (y viceversa), etc. Un ejemplo de este estudio es la siguiente imagen

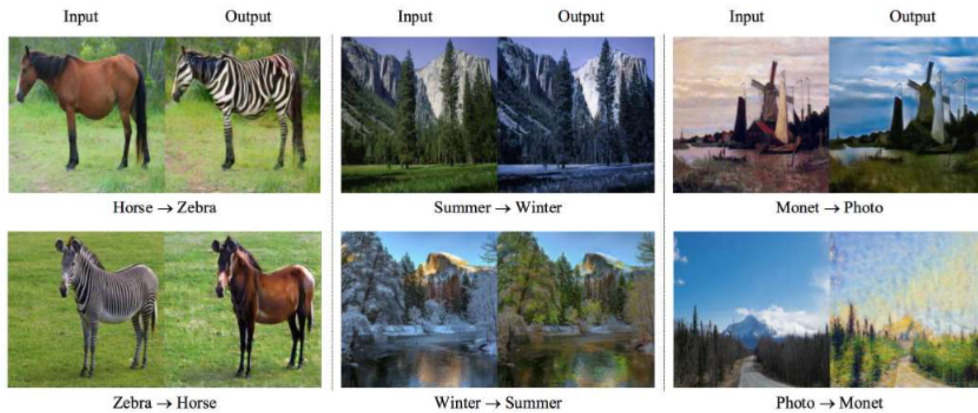


Figura 1.2: Transformación de imágenes mediante Redes GAN. Imagen tomada de [12]

Para poder profundizar sobre las redes GAN y ver cómo funcionan, es necesario entender de una forma matemática que es una red neuronal y cada uno de sus elementos.

1.2. Conceptos básicos y contexto histórico

Para el desarrollo de esta sección, me he basado en el libro [9] junto con contexto histórico de [4].

Durante muchos años se ha buscado intentar reproducir el funcionamiento del cerebro humano en un ordenador. El primer matemático que lo logró fue Alan Turing en 1936. Numerosos matemáticos e informáticos compartieron la forma de pensar de Alan Turing y empezaron a investigar sobre ello y en 1956 se reunieron para hablar de aspectos como la representación de la inteligencia y el aprendizaje en máquinas. A esta reunión se le llamó el Congreso de Dartmouth y tuvo lugar en Hanover, Estados Unidos. Este congreso se considera el origen de una disciplina llamada Inteligencia Articial, la cual se encarga de desarrollar el aprendizaje y la inteligencia mediante ordenadores. Entre 1950 y 1960 se introdujo el concepto de perceptrón, creado por el psicólogo, informático teórico y neurocientífico Frank Roseblatt [4].

Definición 1.1. Un perceptrón es una función que toma *entradas* $x_1, x_2, \ldots, x_n$ binarias, es decir, $x_1, x_2, \ldots, x_n \in \{0, 1\}$ y produce una salida binaria. Para producir esta salida, requiere de *unos pesos* los cuales denotaremos por $w_1, w_2, \ldots$. La salida del perceptrón será 1 si la suma de las entradas por los pesos es mayor que cierta cantidad, la cual llamamos *umbral*. Si dicha suma es menor que el umbral, la salida será 0, esto es, si denotamos al umbral por δ, obtenemos que

$$\text{La salida} = \begin{cases} 0, & si \quad \sum_{j\geq 1} w_j x_j \leq \delta \\ \\ 1, & si \quad \sum_{j\geq 1} w_j x_j > \delta \end{cases}$$

El perceptrón constituye la unidad básica de la Inteligencia Artificial.

Notamos que la condición $\sum_{j\geq 1} w_j x_j \leq \delta$ no es más que un producto escalar del vector de pesos $w = (w_1, w_2, \ldots)$ con el vector de entradas $x = (x_1, x_2, \ldots)$. Esto nos permite simplificar el sumatorio por $w \cdot x$. Además, podemos denotar $b = -\delta$ y así podemos simplificarlo aún más. Al parámetro b se le conoce como *sesgo*. Con estos cambios, obtenemos que

$$\text{La salida} = \begin{cases} 0, & si \quad w \cdot x + b \leq 0 \\ \\ 1, & si \quad w \cdot x + b > 0 \end{cases}$$

Con este concepto, se creó una unión de perceptrones todos conectados entre sí. Esta unión consistía en 3 capas: la primera capa la cual se llama capa de entrada, una capa intermedia la cual se llama capa oculta y una capa final la cual se llama capa de salida. Gracias a esto, se consiguió crear un conjunto de perceptrones en los cuales la primera capa tomaba unas entradas, producían unas salidas las cuales se introducían en las capas ocultas, y estas a su vez proporcionaban una salida que iba a la capa de salida y estas producían un resultado final que corresponde con la salida final 1.1 de la unión de perceptrones.

A este conjunto de perceptrones se le llamó perceptrón multicapas.

Tras unos años, se comprobó que el concepto de perceptrón era incómodo para trabajar, ya

que sólo se podían introducir entradas binarias (y a su vez, sólo producían salidas binarias), esto es, 0 o 1 y, por ello, se crearon las neuronas en 1980 [4].

Definición 1.2. Una neurona es una función que admite entradas reales $x_1, \ldots, x_n \in \mathbb{R}$ y produce una salida entre 0 y 1 utilizando una función de activación[I], esto es, la salida de la neurona es

$$\text{La salida} = \sigma(w \cdot x + b)$$

siendo σ una función de activación $\sigma : \mathbb{R} \to [0, 1]$.

Existen numerosas elecciones de la función activación σ (por ejemplo, la función tangente hiperbólica o la función de Heaviside), pero en Inteligencia Artificial la más utilizada es la función sigmoidea.

Definición 1.3. La función sigmoidea es $\sigma : \mathbb{R} \to [0, 1]$ dada por

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad x \in \mathbb{R}.$$

He representado gráficamente la función sigmoidea mediante un código en *R*.

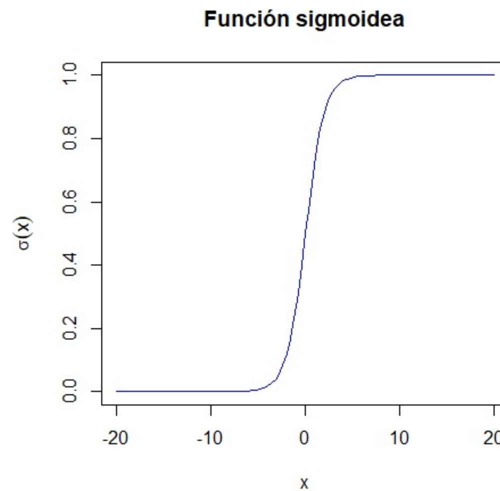


Figura 1.3: Representación gráfica de la función sigmoidea

Podemos apreciar que posee dos asíntotas horizontales (una en $y = 0$ cuando $x \to -\infty$ y otra en $y = 1$ cuando $x \to \infty$) y la función es creciente.

Esta función posee diversas propiedades bastante útiles en la práctica. Una de las propiedades que más vamos a usar a lo largo del TFG va a ser la siguiente proposición.

Proposición 1.1. *La derivada de la función sigmoidea cumple*

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

[I]Una función f es una función de activación si $f : \mathbb{R} \to [0, 1]$

Demostración. Calculemos la derivada de la función sigmoidea $\sigma(x) = \frac{1}{1+e^{-x}}$. Aplicando la regla de la cadena

$$\begin{aligned}\sigma'(x) &= \frac{-(-1)e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{-x}}\frac{1+e^{-x}-1}{1+e^{-x}} \\ &= \frac{1}{1+e^{-x}}\left(\frac{1+e^{-x}}{1+e^{-x}} + \frac{-1}{1+e^{-x}}\right) = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) = \sigma(x)\,(1-\sigma(x))\,.\end{aligned}$$

□

Destacamos que σ está relacionada con la tangente hiperbólica mediante la siguiente relación:

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

Similar que con los perceptrones, se generó una unión de neuronas sigmoideas en la cual todas las neuronas se organizaban en capas y estaban conectadas entre sí. A esta unión se le llamó Red Neuronal Artificial.

Una red neuronal artificial es un conjunto de neuronas sigmoideas conectadas todas entre sí y organizadas en tres capas (capa de entrada, capa oculta y capa de salida), de manera que la salida de una neurona es la entrada de otra salvo en la capa de salida. La red neuronal artificial no es más que una función la cual depende de unos pesos, unos sesgos y una entradas x que devuelve una salida entre 0 o 1 utilizando la función de activación (en nuestro caso usaremos la sigmoidea). Esta definición explícita se explicará con más detalle en el Capítulo 2.

A partir de ahora, a las redes neuronales artificiales las vamos a llamar redes neuronales.

Veamos un ejemplo de red neuronal que tiene 1 neurona en la capa de entrada, 2 neuronas en la capa oculta y 1 neurona en la capa de salida.

Ejemplo 1.2. Supongamos que tenemos una red neuronal que tiene 1 entrada (llamémosla x_1) en la capa de entrada, 2 neuronas en las capas ocultas (llamémoslas h_1 y h_2) y 1 neurona (llamémosla o_1) en la capa de salida y en las que sus pesos correspondientes son $1, 0.5, [0.3, 0.7]$ y sesgos $0, 0.5, 1$ respectivamente. Es decir:

- h_1 tiene peso 1 y sesgo 0.
- h_2 tiene peso 0.5 y sesgo 0.5.
- o_1 tiene pesos $[0.3, 0.7]$ y sesgo 1.

Notamos que la neurona o_1 requiere de un vector como peso mientras que las neuronas h_1, h_2 solo requieren de un número, ya que la neurona o_1 tiene 2 entradas (que son las respectivas salidas de h_1 y h_2) y las neuronas h_1, h_2 tienen sólo 1 entrada. Por ello, he creado un código para calcular la salida de la red neuronal con más facilidad. Este me va a proporcionar la salida de la red neuronal.

```
import numpy as np

def sigmoidea(x):
  return 1/(1+np.exp(-x))
```

```
def salida(x,w,b):
  return sigmoidea(np.dot(x,w)+b)

w1=1; b1=0
w2=0.5; b2=0.5
w3=[0.3,0.7]; b3=1

x=1

h1=salida(x,w1,b1)
h2=salida(x,w2,b2)

o1=salida([h1,h2],w3,b3)

print("La salida de h1 es: %.5f" %h1)
print("La salida de h2 es: %.5f" %h2)
print("La salida de o1 es: %.5f" %o1)
```

La salida del programa es:

```
La salida de h1 es: 0.73106
La salida de h2 es: 0.73106
La salida de o1 es: 0.84955
```

Como se aprecia en el ejemplo, la salida de la red neuronal es un número entre $(0, 1)$. Además, a estos datos se les puede dar una interpretación real, y es lo que se hará en los siguientes ejemplos.

1.3. Aplicaciones de las redes neuronales

Para el desarrollo de esta sección, he utilizado el artículo [8].

Las redes neuronales pueden utilizarse en un gran número y variedad de aplicaciones, tanto en el campo informático como en el campo matemático como en el campo biológico. Las redes neuronales son una de las partes más importantes del Big Data y del análisis de datos. Se pueden desarrollar diversas redes neuronales en un periodo de tiempo razonable, con la capacidad de realizar tareas concretas mejor que otras tecnologías. Algunas de sus aplicaciones son:

- Campo biológico:
  - Estudio del sistema nervioso.
  - Modelización matemática de la retina.
- Campo empresarial:
  - Identificación de candidatos para posiciones específicas.
  - Análisis de bases de datos.
  - Optimización de plazas y horarios en líneas de vuelo, en plantilla de hospitales, etc.

- Optimización del flujo del tráfico.
- Reconocimiento de caracteres escritos.
- Robotización de procesos automatizados (RPA).
- Obtención de datos de imágenes.

- Medio ambiente:
  - Meteorología y predicción del clima.
  - Evaluación de posibles formaciones geológicas.
- Campo económico:
  - Previsión y evaluación del aumento y disminución de los precios.
  - Valoración del riesgo de los créditos.
  - Reconocimiento de falsificaciones en las firmas.
- Campo médico:
  - Analizadores del habla para ayudar a los sordos.
  - Reconocimiento de enfermedades y tratamientos a partir de síntomas y/o de datos obtenidos mediante pruebas médicas (análisis de sangre, TAC, radiografías, ecografías, etc).
  - Predicción de efectos secundarios en los tratamientos médicos.
- Campo militar:
  - Clasificación de señales de radar.
  - Fabricación de armamento inteligente.
  - Optimización ante la falta de recursos.

En este TFG se van a aplicar como ejemplo redes neuronales en la clasificación de imágenes de prendas de vestir y en el reconocimiento de caracteres manuscritos en imágenes.

Existen diversos tipos de Redes Neuronales (redes neuronales convolucionales, redes adversarias generativas, etc) En el Capítulo 3 se profundizará más acerca de las redes neuronales convolucionales (CNN) y se introducirá un ejemplo programado de ellas.

Estudiando profundamente las redes neuronales, se comprobó que estas no producían una salida exacta, es decir, la salida que deberían producir y la que producen diferían un poco. Para estudiar más a fondo esta diferencia, se creó el concepto de función coste con el fin de evaluar esta diferencia y minimizarla lo mayor posible.

1.4. La función coste

Una función coste es una función de variable real que trata de determinar el error entre el valor real y el valor que produce la red neuronal, con el fin de optimizar los parámetros de la red neuronal.

Existen diversos tipos de funciones coste que, dependiendo del campo en el que la utilicemos, se utilizan unas u otras. En este TFG se van a utilizar dos tipos de funciones coste: La función error cuadrático medio (Squared Mean Error, SQE) y la función coste de entropía cruzada (cross-entropy function).

Definición 1.4. Sean $x_1, \dots, x_n \in \mathbb{R}^N$. La función coste del error cuadrático medio (SQE) es una función $SQE : \mathbb{R}^N \times \overset{n}{\dots} \times \mathbb{R}^N \to \mathbb{R}$ definida por

$$SQE(x) = \frac{1}{2n} \sum_{x_i} \|y(x_i) - a^L(x_i)\|^2, \quad \forall i \in \{1, \dots, n\}$$

[II]siendo:

- $n \in \mathbb{N}$ el número total de ejemplos,
- $y(x_i)$ el vector de las salidas reales que corresponden con los datos de entrenamiento x_i,
- $L \in \mathbb{N}$ el número de capas de la red neuronal,
- $a^L(x_i)$ el vector de salidas de la última capa de la red neuronal cuando $x \in \mathbb{R}^N$ es una entrada.
- $(x_1, \dots, x_n)$ el conjunto de vectores que se toman para calcular el error. En la práctica se calcula el error en la base de datos que vamos a utilizar para entrenar la red neuronal.

Definición 1.5. Sean $x_1, \dots, x_n \in \mathbb{R}^N$. La función coste de entropía cruzada es una función $EC : \mathbb{R}^N \times \overset{n}{\dots} \times \mathbb{R}^N \to \mathbb{R}$ definida por

$$EC(x) = -\frac{1}{n} \sum_{x_i} \left[y(x_i) \ln a^L(x_i) + (1 - y(x_i)) \ln(1 - a^L(x_i)) \right], \quad x_i \in \mathbb{R}^N, \ \forall i \in \{1, \dots, n\}$$

[III]donde, de nuevo, $n \in \mathbb{N}$ es el número total de ejemplos, $y(x_i)$ son las salidas deseadas (las salidas reales de la entrada x_i) y $a^L(x_i)$ es la salida proporcionada por la red neuronal cuando $x_i \in \mathbb{R}^N$ es una entrada y $(x_1, \dots, x_n)$ el conjunto de vectores a calcular el error. Además, ln denota el logaritmo neperiano.

El estudio del error con esta función coste EC es similar al realizado por la función de coste original, así que en la red neuronal artificial de reconocimiento de caracteres utilizaré el error cuadrático medio y en la red neuronal convolucional de clasificación de imágenes de ropa utilizaré la función de entropía cruzada porque produce mejores resultados.

1.4.1. Las dos hipótesis a asumir sobre la función coste.

En el Capítulo 2 se presentará una herramienta bastante utilizada en redes neuronales. Esta técnica se llama backpropagation y para aplicarse requiere de unas hipótesis [9] sobre la función coste. Es por ello que he introducido estas hipótesis de la función coste en este apartado para más adelante poder aplicar el backpropagation sin problema alguno.

La primera hipótesis que tenemos que asumir es que podemos reescribir la función coste como media de funciones coste particulares a una entrada x, es decir:

$$C = \frac{1}{n} \sum_{x_i} C_{x_i}, \quad x_i \in \mathbb{R}^N, \ i = 1, \dots, n.$$

[II]Notamos que la función coste no depende únicamente de la entrada x, también depende de unos pesos y unos sesgos. Para no sobrecargar la notación, no escribo explícitamente la dependencia en función de pesos y sesgos.

[III]Similar con la función error cuadrático medio.

Esta hipótesis tenemos que hacerla ya que el algoritmo del backpropagation va a computar las derivadas parciales C_{x_i} respecto de w y b sobre puntos particulares, luego gracias a esta hipótesis podemos recuperar las parciales de la función coste original.

La segunda hipótesis que tenemos que asumir es que la función coste puede ser escrita como función que depende únicamente de la salida de la red neuronal (a pesar de que, a su vez, dependa de otras variables como son los pesos o sesgos), es decir,

$$C = C(a^L).$$

Proposición 1.2. *Las funciones coste 1.4 y 1.5 cumplen las dos hipótesis necesarias para que el algoritmo del backpropagation funcione correctamente.*

Demostración. Veamos el estudio para cada función coste. Para la función coste 1.4:

1. La función coste 1.4 cumple la primera hipótesis ya que se puede poner como media de funciones coste individuales como son $SQE_{x_i} = \frac{1}{2}\|y(x_i) - a^L(x_i)\|^2$. Luego

   $$SQE(x) = \frac{1}{n}\sum_{x_i} SQE_{x_i}$$

   y así cumple la primera hipótesis.

   Utilizando la primera hipótesis, notamos que la función coste SQE depende únicamente de la salida de la red neuronal cuando x_i es una entrada ya que $y(x_i)$ es constante cuando se introduce la entrada x_i [9].

2. La función coste 1.5 cumple la primera hipótesis ya que se puede reescribir la función coste como media de costes individuales tomando

   $$EC_{x_i} = -y(x_i)\ln a^L(x_i) - (1 - y(x_i))\ln(1 - a^L(x_i))$$

   y así obtenemos que

   $$EC(x) = -\frac{1}{n}\sum_{x_i}\left[y(x_i)\ln a^L(x_i) + (1 - y(x_i))\ln(1 - a^L(x_i))\right]$$

   De la misma manera, la función coste 1.5 se puede escribir como media de funciones de coste individuales y estas a su vez dependen únicamente de la salida de la red neuronal ya que $y(x_i)$ es una salida constante cuando se introduce la entrada x_i, luego la función coste 1.5 cumple la hipótesis 2, y así, se puede utilizar el algoritmo del backpropagation con esta función coste.

□

1.4.2. Problema de minimización de la función coste

Notamos que cuanto mayor sea el error de la función coste, menos precisa es la red neuronal y, cuanto menor sea el error de la función coste, más precisa es la red neuronal. Matemáticamente, este problema se traduce en un problema de optimización en el que hay que encontrar los pesos y sesgos para que la función coste sea lo menor posible y, así, mejorar la red neuronal. Dicho problema de optimización se puede resolver de varias maneras. La que a priori

puede parecer más fácil y directa es calculando el gradiente e igualando a 0 (aunque puede dar lugar a un máximo y no a un mínimo). Esta solución se puede computar fácilmente si se trata de una función coste que depende de pocos pesos o sesgos, pero normalmente no es así.

En casos prácticos, la función coste depende de muchísimos pesos y de muchísimos sesgos, luego hacer la derivada respecto a cada uno puede ser bastante complicado, además de que, aunque se pueda calcular, resulta difícil despejar los pesos y sesgos una vez los igualas a 0.

Por ello, existe un método llamado Descenso Gradiente que consiste en calcular el mínimo de una función cualquiera. Dicho método posee un inconveniente, requiere del gradiente de la función para poderse aplicar, además de ciertos parámetros los cuales deben ser los adecuados para asegurar la convergencia de dicho método. Para poder calcular el gradiente de una función, se va a utilizar un método llamado backpropagation, el cual nos va a proporcionar las derivadas parciales de la función coste respecto a cualquier peso y sesgo de la red neuronal y, con ello, obtener el gradiente de la función y así poder aplicar el Método Descenso Gradiente [9].

Existen diversos métodos numéricos para calcular el mínimo de una función (Descenso Gradiente, Descenso Gradiente Estocástico, Optimizador de Adam, etc.) pero este TFG se va a centrar en el Descenso Gradiente (DG) y Descenso Gradiente Estocástico (SGD), aunque utilizaremos el método Adam en las redes neuronales convolucionales.

2 Herramientas para minimizar la función coste: Descenso gradiente y Backpropagation.

En este capítulo se explicarán algunos métodos iterativos para poder minimizar una función cualquiera. En la práctica, se utilizará la función coste y con el método Adam (que se introducirá más adelante) se minimizará. Además, para poder calcular el gradiente (que es uno de los requisitos para poder aplicar los métodos de optimización) se utiliza un algoritmo llamado Backpropagation. Este algoritmo nos proporcionará las derivadas parciales de la función coste respecto a todos los pesos y sesgos de la red neuronal y esto se logra mediante sus 4 ecuaciones fundamentales.

2.1. El Descenso Gradiente

2.1.1. Introducción

El Método del Descenso Gradiente (DG) es un método iterativo de optimización para encontrar un mínimo local de una función. El gradiente de una función se dirige hacia el lugar de mayor crecimiento de la función, por lo tanto, la idea es recorrer la función en el sentido opuesto al gradiente hasta encontrar un mínimo local. Este algoritmo requiere de una condición inicial, la cual llamaremos x_0, que será el punto donde se va a comenzar el descenso. Dependiendo de la elección de este x_0 el algoritmo puede converger a un mínimo u otro, en el caso de que la función tenga uno o más mínimos.

El Método del Descenso Gradiente también requerirá de una sucesión de *tasas de aprendizaje* la cual se denotará por $\{\lambda_n\}_{n\in\mathbb{N}}$. Estos λ_i medirán la *longitud del paso i* que estamos dando y, por lo tanto, es necesario encontrar unos λ_i adecuados para que converja el método.

Definición 2.1. Sea f una función diferenciable. Entonces el Método del Descenso Gradiente [10]

- Empieza con un x_0 inicial, que es el punto donde comenzará el descenso.
- Requiere de una sucesión $\{\lambda_n\}_{n\in\mathbb{N}}$ con $\lambda_i \in \mathbb{R}^+$.
- Define una sucesión $\{x_n\}_{n\in\mathbb{N}}$ dada por

$$x_{n+1} = x_n - \lambda_n \nabla f(x_n), \quad n \in \mathbb{N}$$

  que convergerá al punto crítico x, es decir, $\{x_n\}_{n\in\mathbb{N}} \to x$ si se eligen los correctos x_0 y λ_n.

- Termina cuando $\exists n_0 \in \mathbb{N}$ tal que $\forall m \geq n_0$, $\|\nabla f(x_m)\| \leq \varepsilon$, $\varepsilon > 0$, es decir, cuando la sucesión está suficientemente cerca del punto crítico (en el cual $\nabla f(x) = 0$). Usualmente este ε se escoge lo suficientemente pequeño para considerar que el punto x_n está cerca del punto crítico que queremos buscar, x. Así, cuanto menor sea ε, mayor será la precisión con la que aproxima el Método Descenso Gradiente a nuestro punto crítico.

Observación 2.1. La sucesión $\{\lambda_n\}_{n\in\mathbb{N}}$ no tiene por qué ser una sucesión decreciente, también puede ser una sucesión creciente o constante. Sin embargo, el hecho de que sea creciente o constante puede suponer la pérdida de convergencia del método, es decir, existen funciones suficientemente diferenciables (por ejemplo, C^∞) que si se toma una sucesión constante $\{\lambda_n\}_{n\in\mathbb{N}}$, el algoritmo no converge.

Ejemplo 2.1 (Ejemplo de función en la que el método no converge si se toma la sucesión $\{\lambda_n\}$ constante, es decir, $\{\lambda_n\} = \lambda$)**.** Consideramos la función $f : \mathbb{R} \to \mathbb{R}$ tal que

$$f(x) = cx^2 \text{ con } c > 0$$

Notamos que la función $f \in C^\infty$. Derivamos la función y obtenemos que

$$f'(x) = 2cx$$

Si le aplicamos el método Descenso Gradiente con la sucesión de $\{\lambda_n\}_{n\in\mathbb{N}}$ constante, es decir, $\lambda_i = \lambda \ \forall i \in \mathbb{N}$, obtenemos la sucesión

$$x_{n+1} = x_n - \lambda\nabla f(x_n) = x_n - \lambda 2cx_n = x_n(1-2c\lambda)$$

Notamos que este método va a converger si, y sólo si, $|1-2c\lambda| < 1$.
Además, $\forall\lambda > 0 \ \exists c$ tal que $|1-2c\lambda| \geq 1$, por ejemplo, $\lambda = \frac{1}{c}$.
Luego, a pesar de que $f \in C^\infty$, el método no tiene por qué converger siempre, es necesario escoger los parámetros adecuados.

El siguiente gráfico muestra visualmente por qué es importante la elección de la sucesión de tasas de aprendizaje

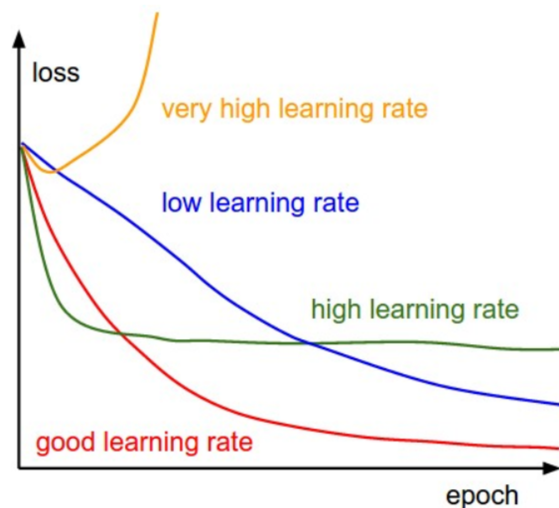


Figura 2.1: Convergencia en función de la tasa de aprendizaje. Imagen obtenida en [2]

Ejemplo 2.2 (Ejemplo de convergencia en función de x_0.)**.** Consideramos la función $f : \mathbb{R} \to \mathbb{R}$ definida por

$$f(x) = \operatorname{sen}(x)$$

En este caso, si elegimos el punto inicial $x_0 = 0.5$, el algoritmo convergerá al punto crítico $x = -\frac{\pi}{2}$ mientras que, si cogemos el punto inicial $x_0 = 6$, el algoritmo convergerá al punto crítico $x = \frac{3\pi}{2}$.

Para comprobar la convergencia numéricamente, he creado un código en Python.

```
import numpy as np

def seno(x):
    return np.sin(x)

def seno_der(x):
    return np.cos(x)

x=6
iteraciones=range(100)
eps=0.00000005
n=0
learn=1
for i in iteraciones:
    if np.abs(seno_der(x))>eps:
        x = x - (learn-i/100)*seno_der(x)
        n=n+1
    else:
        x = x - (learn-i/100)*seno_der(x)
        n=n+1
        break

print("Etapa %d: [%.9f] " %(n,x))
```

En el trozo de código

```
    x=6
```

inicialiamos la semilla o punto inicial.

Si lo inicializamos con $x_0 = 6$ obtenemos que el algoritmo converge a $x = \frac{3\pi}{2} = 4.712388981\ldots$ mediante la salida

```
Etapa 7: [4.712388981]
```

y si inicializamos la semilla en $x_0 = 0.5$ obtenemos que converge al mínimo $x = -1.570796325 = -\frac{\pi}{2}$

```
Etapa 8: [-1.570796325]
```

Ejemplo 2.3. Considero $f : \mathbb{R}_3 \to \mathbb{R}$ tal que $f(x,y,z) = x^2 + y^2 + z^2 - 3$.

Para ver que converge teóricamente al mínimo local (en este caso global) calculamos el gradiente de la función e igualamos a 0 para poder ver posibles mínimos o máximos:

$$\nabla f(x,y,z) = (2x, 2y, 2z) = 0 \Rightarrow x = y = z = 0$$

Para ver que es un mínimo local, calculamos el Hessiano en ese punto y veamos si la matriz es definida positiva:

$$Hess f(x,y,z) = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} > 0$$

luego el punto $x_{\min} = (0,0,0)$ es un mínimo local. De hecho, el punto $x_{\min}$ es un mínimo global, ya que la matriz Hessiana es definida positiva para todo x, y, z y eso implica que la función es convexa $\Rightarrow$ su mínimo es global y, además, es único.

Para aplicar el Descenso Gradiente en este ejemplo y ver que un x_0 cualquiera converge a ese mínimo, he tomado $x_0 = (4,2,1)$, el número de iteraciones máximas $n_{max} = 100$, $\lambda_i = 1 - \frac{i}{n_{max}}$ y $\varepsilon = 0.00000005$.

Para este ejemplo, he modificado el código anterior para introducir la función de tres variables mencionada anteriormente:

```
import numpy as np

def f(list):
    return list[0]**2+list[1]**2+list[2]**2-3

def f_der(list):
    return np.array([2*list[0],2*list[1],2*list[2]])

x=np.array([4,2,1])
iteraciones=range(100)
eps=0.00000005
n=0
learn=1
for i in iteraciones:
    if np.sqrt(x[0]**2+x[1]**2+x[2]**2)>eps:
        x = x - (learn-i/100)*f_der(x)
        n=n+1
        print("Etapa %d: [%.9f,%.9f,%.9f] "
                                %(n,x[0],x[1],x[2]))

    else:
        x = x - (learn-i/100)*f_der(x)
        n=n+1
        print("Etapa %d: [%.9f,%.9f,%.9f] "
                                %(n,x[0],x[1],x[2]))
        break
```

Este programa nos proporciona la salida

```
Etapa 1: [-4.000000000,-2.000000000,-1.000000000]
Etapa 2: [3.920000000,1.960000000,0.980000000]
Etapa 3: [-3.763200000,-1.881600000,-0.940800000]
Etapa 4: [3.537408000,1.768704000,0.884352000]
Etapa 5: [-3.254415360,-1.627207680,-0.813603840]
```

```
...

Etapa 34: [0.000000999,0.000000499,0.000000250]
Etapa 35: [-0.000000320,-0.000000160,-0.000000080]
Etapa 36: [0.000000096,0.000000048,0.000000024]
Etapa 37: [-0.000000027,-0.000000013,-0.000000007]
Etapa 38: [0.000000007,0.000000003,0.000000002]
```

Tras esta salida, notamos que la sucesión proporcionada por el Método Descenso Gradiente $\{x_n\}_{n\in\mathbb{N}}$ converge al mínimo global calculado previamente $x_{\min} = (0,0,0)$ cuando $n \to \infty$.

En este caso, el punto x_0 y la sucesión $\{\lambda_n\}_{n\in\mathbb{N}}$ con $\lambda_i = 1 - \frac{i}{n_{max}}$ para $i = 1,\ldots,n_{max} - 1$ están elegidos correctamente, luego este método es convergente en este caso.

2.1.2. Convergencia del Método Descenso Gradiente.

En esta sección he utilizado el libro [7].

Como bien se ha explicado antes, si elegimos la condición inicial x_0 o las longitudes de paso $\{\lambda_n\}_{n\in\mathbb{N}}$ de una manera errónea, el método puede no converger o converger a otro mínimo totalmente distinto. En este apartado se van a ver algoritmos que aseguran la convergencia local junto con sus respectivas demostraciones respecto a las longitudes de paso.

Definición 2.2. Llamaremos *dirección de descenso fuerte de f en x* a $d = -\nabla f(x)$.

Existen diversas maneras de elegir la sucesión $\{\lambda_n\}$. En este TFG, vamos a elegir la sucesión $\{\lambda_n\} = \{\beta^{m_n}\}$, donde $\beta \in (0,1)$ y $m_n \geq 0$ es el menor número entero no negativo tal que λ_n cumple la siguiente condición (condición 2.1.2)

$$\boxed{f(x_{n+1}) - f(x_n) < -\alpha\lambda_n\|\nabla f(x_n)\|^2, \qquad \alpha \in (0,1).}$$

Esta condición nos asegura que existe *suficiente cercanía* entre un término de la sucesión y su anterior, y con esto nos asegura que no diverge el método.

La razón por la cual se impone esta condición es porque si aproximamos f por un modelo lineal

$$m_n(x) = f(x_n) + \nabla f(x_n)(x - x_n),$$

obtenemos un error que es

$$\begin{aligned}\text{error} = m_n(x_n) - m_n(x_{n+1}) &= f(x_n) + \nabla f(x_n)\underbrace{(x_n - x_n)}_{0} - f(x_n) - \nabla f(x_n)(x_{n+1} - x_n) \\ &= -\nabla f(x_n)(x_{n+1} - x_n) \underbrace{=}_{(1)} \lambda_n \nabla f(x_n)\nabla f(x_n) = \lambda_n\|\nabla f(x_n)\|^2,\end{aligned}$$

donde en (1) se ha utilizado que $x_{n+1} - x_n = x_n - \lambda_n\nabla f(x_n) - x_n = -\lambda_n\nabla f(x_n)$.

Luego, se exige la condicion 2.1.2 para asegurar que el método converge mínimo de manera lineal.

Luego el error que tenemos que cometer en el MDG tiene que ser como máximo el error que cometemos al ajustarlo por un modelo lineal.

A la elección de $\{\lambda_n\} = \{\beta^{m_n}\}$, elección de $\alpha = 10^{-4}$ y a la utilización de este método se le llama **Regla de Armijo** [7].

Definición 2.3. El algoritmo de la Regla de Armijo es un algoritmo que consiste en

- Para $n = 0, 1, \ldots, n_{max}$
  1. Calcular $\nabla f(x_n)$.
  2. Encontrar el mínimo entero $m_n \geq 0$ que cumple la condición 2.1.2 para $\lambda_n = \beta^{m_n}$ con $\beta \in (0, 1)$.
  3. Aplicar el paso $x_{n+1} = x_n - \lambda_n \nabla f(x_n)$.
- Si $n = n_{max}$, el algoritmo termina.

Existe una modificación de este algoritmo reduciendo λ de distinta manera. El siguiente algoritmo consite en

- Para $n = 0, 1, \ldots, n_{max}$
  1. Calcular $\nabla f(x_n)$.
  2. Empezamos por $\lambda_0 = 1$ y comprobamos si se cumple la condición 2.1.2. Si se cumple, seguimos al siguiente paso y, en caso de que no se cumpla, tomamos $\lambda_1 \in [\beta_s \lambda_0, \beta_l \lambda_0]$ con $\beta_s, \beta_l \in (0, 1)$ y así sucesivamente. Es decir, tomamos $\lambda_n \in [\beta_s \lambda_{n-1}, \beta_l \lambda_{n-1}]$ con $\beta_s, \beta_l \in (0, 1)$ y terminamos cuando λ_n cumpla 2.1.2.
  3. Aplicar el paso $x_{n+1} = x_n - \lambda_n \nabla f(x_n)$.
- Si $n = n_{max}$, el algoritmo termina.

2.1.3. Resultados de convergencia del Método Descenso Gradiente

Observación 2.2. Los siguientes resultados de convergencia del Descenso Gradiente se pueden generalizar aún más para una dirección de descenso fuerte d que cumpla que $d = -H^{-1}\nabla f(x)$ siendo H^{-1} la inversa de una matriz Hessiana de la función f. Para poder aplicar los resultados de convergencia a redes neuronales basta con tomar $H = Id$, luego $H^{-1} = Id$ y $d = -\nabla f(x)$.

Observación 2.3. En los siguientes teoremas voy a utilizar el concepto de Lipschitzianidad. Sea $f : \mathbb{R}^n \to \mathbb{R}^n$ una función. Entonces f es una función Lipschitziana si cumple que

$$\exists L > 0 \text{ tal que } \|f(x) - f(y))\| \leq L\|x - y\|, \quad \forall x, y \in \mathbb{R}^n.$$

En ese caso, diremos que f es una función Lipschitziana con constante de Lipschitz L.

El siguiente lema nos va a proporcionar una cota superior de los λ_i a elegir [7].

Lema 2.1. *Sea ∇f el gradiente de la función f. Supongamos que $\nabla f \neq 0$ es una función Lipschitziana y continua con constante de Lipschitz L. Sea $\alpha \in (0, 1)$ y $x \in \mathbb{R}^N$. Entonces la condición de suficiencia 2.1.2 se da para todo λ que cumpla la condición*

$$0 < \lambda \leq \frac{2(1 - \alpha)}{L}.$$

Demostración. Para facilitar los cálculos, llamo $d = -\nabla f(x)$. Aplicando el teorema fundamental del Cálculo

$$\int_0^1 \nabla f(x + t\lambda d)^T \lambda d \, dt = f(x + \lambda d) - f(x).$$

Por consecuente

$$\begin{aligned}
f(x+\lambda d) &= f(x) + \lambda \nabla f(x)^T d - \lambda \nabla f(x)^T d + \lambda \int_0^1 \nabla f(x + t\lambda d)^T d \, dt \\
&= f(x) + \lambda \nabla f(x)^T d - \lambda \int_0^1 (\nabla f(x))^T d \, dt + \lambda \int_0^1 \nabla f(x + t\lambda d)^T d \, dt \\
&= f(x) + \lambda \nabla f(x)^T d + \lambda \int_0^1 (\nabla f(x + t\lambda d) - \nabla f(x))^T d \, dt.
\end{aligned}$$

Así, utilizando la Lipschitzianidad del $\nabla f(x)$ tenemos que

$$\begin{aligned}
f(x+\lambda d) &= f(x) + \lambda \nabla f(x)^T d + \lambda \int_0^1 (\nabla f(x + t\lambda d) - \nabla f(x))^T d \, dt \\
&\leq f(x) + \lambda \nabla f(x)^T d + \lambda d \int_0^1 L\,(x + t\lambda d - x)\, dt \\
&= f(x) + \lambda \nabla f(x)^T d + \lambda^2 \|d\|^2 \int_0^1 Ltdt \\
&= f(x) + \lambda \nabla f(x)^T d + \frac{\lambda^2 L}{2} \|d\|^2.
\end{aligned}$$

Entonces, sustituyendo la definición de $d = -\nabla f(x)$ obtenemos

$$f(x+\lambda d) \leq f(x) + (\lambda - \frac{\lambda^2 L}{2}) \|\nabla f(x)\|^2 = f(x) + (1 - \frac{\lambda L}{2}) \lambda \|\nabla f(x)\|^2$$

que implica que se cumpliría la condición de suficiencia 2.1.2 si

$$\alpha \leq 1 - \frac{\lambda L}{2},$$

que equivale a la condición del lema, ya que, despejando λ obtenemos que

$$\boxed{\lambda \leq \frac{2(1-\alpha)}{L}.}$$

□

Lema 2.2. *Sea ∇f una función Lipschitziana y continua con constante Lipschitz L. Sea $\{x_n\}$ la sucesión dada por el algoritmo. Entonces las longitudes de los pasos $\{\lambda_n\}$ satisfacen*

$$\lambda_n \geq \overline{\lambda} = \frac{2\beta_s(1-\alpha)}{L}$$

y a lo sumo se utilizarán m reducciones [7], siendo m como sigue

$$m = \frac{\log(\frac{2(1-\alpha)}{L})}{\log(\beta_l)}$$

Demostración. La regla de Armijo parará cuando encuentre un λ con el que cumpla todas las condiciones necesarias. Dicho λ se encontrará cuando se encuentre dentro del rango de

nuestros λ, es decir, cuando cumple 2.1

$$0 < \lambda \leq \frac{2(1-\alpha)}{L},$$

si no se para antes. Como máximo lo único que puede hacerlo parar es un factor de β_s. Como β_s es lo suficientemente pequeño, la desigualdad cambia, luego

$$\boxed{\lambda_n \geq \frac{2\beta_s(1-\alpha)}{L} = \overline{\lambda},}$$

que prueba el primer resultado del lema.

La búsqueda en linea requerirá como máximo m reducciones de paso, donde m es el menor entero no negativo cumpliendo

$$\frac{2(1-\alpha)}{L} > \beta_l^m.$$

Para despejar m y obtener la misma expresión del Lema 2.2, basta aplicar logaritmos a ambos lados

$$\log\left(\frac{2(1-\alpha)}{L}\right) > m \cdot \log(\beta_l),$$

que implica que el número de reducciones máximo a tomar es

$$\boxed{m = \frac{\log(\frac{2(1-\alpha)}{L})}{\log(\beta_l)}.}$$

□

Gracias a estos teoremas, podemos demostrar un teorema que nos va a proporcionar la convergencia del algoritmo de búsqueda en línea de Armijo.

Teorema 2.1. *Sea ∇f una función Lipschitziana y continua con constante de Lipschitz L. Entonces se cumple que la sucesión $\{f(x_n)\}$ no está acotada inferiormente o*

$$lim_{n\to\infty}\nabla f(x_n) = 0$$

En particular, si $\{f(x_n)\}$ está acotada inferiormente y $\left\{x_{\sigma(n)}\right\} \to x^*$ es una sucesión parcial convergente de x_n, entonces $\nabla f(x^*) = 0$ [7].

Demostración. Supongamos que $\{f(x_n)\}_{n\in\mathbb{N}}$ está acotada inferiormente y veamos que

$$lim_{n\to\infty}\nabla f(x_n) = 0$$

Como, por construcción, $\{f(x_n)\}$ es decreciente y por hipótesis está acotada inferiormente, entonces es convergente, esto es, $\exists \lim_{n\to\infty} f(x_n) = f^*$. Además,

$$lim_{n\to\infty}(f(x_{n+1}) - f(x_n)) = 0.$$

Aplicando el Lema 2.1 y el Lema 2.2, obtenemos

$$f(x_{n+1}) - f(x_n) < -\alpha\lambda_n\|\nabla f(x_n)\|^2 \leq -\alpha\overline{\lambda}\|\nabla f(x_n)\|^2 \leq 0.$$

Por consecuente,

$$\boxed{\|\nabla f(x_n)\|^2 \leq \frac{(f(x_n) - f(x_{n+1}))}{\alpha\bar{\lambda}} \to 0, \quad n \to \infty.}$$

□

Este último teorema nos afirma que bajo esas condiciones, el gradiente de la función va a converger a 0, luego el Método Descenso Gradiente va a converger a un mínimo. En el caso de que la función no esté acotada inferiormente, la función no tendrá mínimo.

2.1.4. Introducción del Descenso Gradiente Estocástico

En los problemas de optimización, el método Descenso Gradiente es una buena alternativa para poder hallar la solución. Sin embargo, existe un método llamado Descenso Gradiente Estocástico (SGD) que, a diferencia del Descenso Gradiente, consigue hallar el mínimo de una función de una manera más rápida y eficiente.

A diferencia del método Descenso Gradiente, el método Descenso Gradiente Estocástico no computa el gradiente de una función, sino que computa el gradiente respecto a una variable de la función.

Definición 2.4. Sea f una función $f : \mathbb{R}^n \to \mathbb{R}$ y sea ∇f su gradiente. El método Descenso Gradiente Estocástico es un método iterativo el cual parte de un x_0 inicial el cual llamaremos semilla o punto inicial y define la sucesión

$$x_{n+1} = x_n - \lambda_n \nabla_{x_k} f(x_n) \quad \text{para } n = 0, 1, \ldots$$

donde λ_n es el paso o tasa de aprendizaje y siendo x_k una variable tomada aleatoriamente.

Debido a la complejidad del Descenso Gradiente Estocástico, para asegurar la convergencia se necesitan dos hipótesis de la función f [15].

1. El gradiente de f tiene que estar acotado:

$$\exists M > 0 : \sup_x \|\nabla_{x_k} f(x)\|^2 \leq M, \ \forall x \in \mathbb{R}^n.$$

2. f es fuertemente convexa:

$$\exists \mu > 0 : \ f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\mu}{2}\|y - x\|^2, \ \forall x, y \in \mathbb{R}^n.$$

   Notamos que si $\mu = 0$, esta condición implica la convexidad.

2.1.4.1. Descenso Gradiente Estocástico en Redes Neuronales.

La implementación del método Descenso Gradiente Estocástico en Redes Neuronales supone una mayor precisión en la optimización de la función coste que utilizando el método Descenso Gradiente. El Descenso Gradiente Estocástico converge más rápido aunque su convergencia debe probarse de manera experimental debido a que existen pocos resultados que puedan garantizar teóricamente que el método converge por su complejidad.

En la aplicación a Redes Neuronales de este método, cambiamos los pesos por una mínima variación de su peso original [9], esto es,

$$w_i \leftarrow w_i - \lambda \frac{\partial C}{\partial w_i},$$
$$b_i \leftarrow b_i - \lambda \frac{\partial C}{\partial b_i},$$

siendo C la función coste $C = \frac{1}{2n} \sum_{x_i} \|y(x_i) - a^L(x_i)\|^2$.

A λ se le llama de nuevo *tasa de aprendizaje* que controla lo rápido que aprende la red neuronal.

La elección de λ debe ser la adecuada, ya que si es muy grande puede no converger.

2.1.5. Introducción sobre el Optimizador Adam

El Optimizador Adam es un algoritmo de optimización similar al Descenso Gradiente Estocástico. Este algoritmo es bastante útil cuando se trabaja con conjuntos de datos bastante grandes o con una numerosa cantidad de variables. Intuitivamente, el optimizador Adam consiste en una combinación del Método Descenso Gradiente Estocástico junto con un algoritmo llamado *momentum* y junto a otro algoritmo llamado *algoritmo de RMSP*.

2.1.5.1. Algoritmo Momentum

En el desarrollo de esta sección se ha utilizado la fuente [11].

Este algoritmo es utilizado para acelerar el proceso en el que el descenso gradiente converge. Para ello, utiliza una suma ponderada de los gradientes de los anteriores pasos.

Luego, ya que el Optimizador Adam es una técnica del Descenso Gradiente Estocástico, utiliza la sucesión $\{x_n\}_{n\in\mathbb{N}}$

$$x_{n+1} = x_n - \lambda_n m_n \quad n \in \mathbb{N}, \quad \text{con } m_0 = 1,$$

siendo $m_n = \beta m_{n-1} + (1-\beta)\nabla_{x_k} f(x_n)$, x_k una variable aleatoria, $\beta \in \mathbb{R}^+$ y $m_0 = 1$.

A este m_n se le suele llamar por *momentum* [11].

2.1.5.2. Algoritmo RMSP (Root Mean Square Propagation)

Este algoritmo es utilizado también para aumentar la velocidad en la que el descenso gradiente estocástico converge.

Este algoritmo también es una técnica del descenso gradiente estocástico, luego utiliza la sucesión

$$x_{n+1} = x_n - \lambda_n \frac{\nabla_{x_k} f(x_n)}{\sqrt{v_n + \varepsilon}}$$

donde $\varepsilon > 0$ y donde

$$v_n = \beta v_{n-1} + (1-\beta)\|\nabla_{x_k} f(x_n)\|^2, \quad \text{con } v_0 = 1.$$

En este algoritmo, $\beta \in \mathbb{R}^+$ y se suele tomar como $\beta = 0.9$. Además, $\varepsilon > 0$ y se suele tomar como $\varepsilon = 10^{-8}$ [11]. De nuevo, x_k es una variable aleatoria.

2.1.5.3. Optimizador Adam

Una vez introducidos el algoritmo Momentum y el algoritmo RMSP, podemos deducir el optimizador Adam. Como se ha comentado antes, este optimizador es una mezcla de ambos algoritmos, luego combinando ambos métodos obtenemos que

$$x_{n+1} = x_n - \lambda_n \frac{m_n}{\sqrt{v_n + \varepsilon}},$$

siendo $\lambda_n \in \mathbb{R}^+ \ \forall n \in \mathbb{N}$, $\varepsilon > 0$ y siendo m_n y v_n las variables correspondientes del algoritmo Momentum y del algoritmo RMSP [11] respectivamente

$$m_n = \beta_1 m_{n-1} + (1-\beta_1)\nabla_{x_k} f(x_n) \quad \beta_1 \in \mathbb{R}^+, \quad \text{con } m_0 = 1,$$
$$v_n = \beta_2 v_{n-1} + (1-\beta_2)\|\nabla_{x_k} f(x_n)\|^2 \quad \beta_2 \in \mathbb{R}^+, \quad \text{con } v_0 = 1.$$

Para poder comparar este método con el Descenso Gradiente, he introducido un gráfico donde podemos comprobar de manera más visual que el Optimizador Adam es más eficiente que el Descenso Gradiente y converge de manera mucho más rapida.

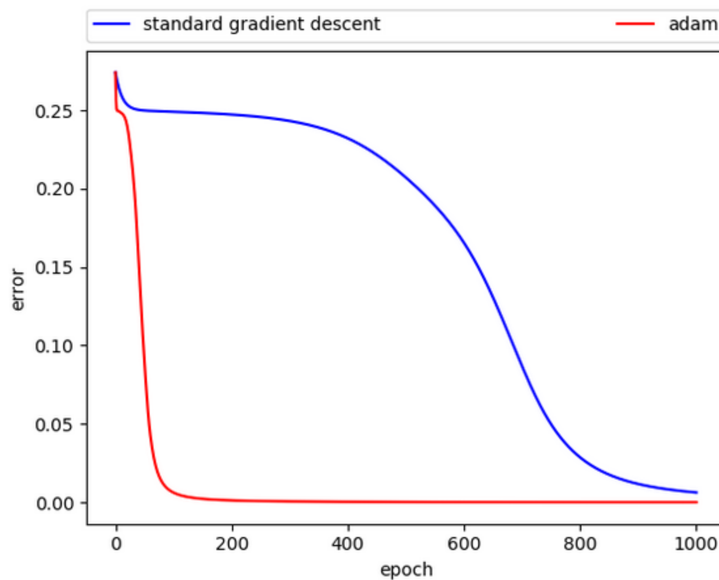


Figura 2.2: Imagen comparativa del Descenso Gradiente frente al Optimizador Adam. Imagen obtenida en [13]

2.2. El backpropagation.

Para esta sección completa, se ha utilizado la fuente [9].

En la anterior sección se ha discutido como aplicar el método Descenso Gradiente a una función y, para ello, era necesario conocer el gradiente de dicha función para aplicar el método.

En algunos casos computar el gradiente de una función puede ser un trabajo muy difícil y costoso, por lo que utilizaremos un algoritmo llamado backpropagation para poder calcular el gradiente de una función con más facilidad.

El backpropagation es un algoritmo que consiste en computar las derivadas parciales de la función coste respecto a cualquier peso o sesgo de la red neuronal y, con esto, obtener el gradiente de la función coste.

2.2.1. Notación.

Antes de profundizar sobre el backpropagation, es necesario asentar una notación fija, precisa y útil a la hora de hacer cuentas.

- Llamaremos w^l_{jk} al peso que conecta la neurona k-ésima en la capa $l-1$ con la neurona j en la capa l.
- Denotaremos por b^l_j al sesgo de la neurona j-ésima en la capa l.
- Usaremos el término a^l_j para la salida de la neurona j-ésima en la capa l.

En el primer capítulo se describió como era la salida de una neurona

$$\text{La salida} = \begin{cases} 0, & si \quad \sum_{j\geq 1} w_j x_j \leq \delta. \\ \\ 1, & si \quad \sum_{j\geq 1} w_j x_j > \delta. \end{cases}$$

Con esta nueva notación, podremos reescribir la ecuación de una manera más simple para hacer cuentas.

Observación 2.4. Con la notación descrita anteriormente, la ecuación 2.2.1 se puede reescribir como

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right).$$

Además, esta ecuación se puede reescribir de forma matricial como sigue

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right)$$

y, denotando $z^l := \sum_k w^l_{jk} a^{l-1}_k + b_j = w^l a^{l-1} + b^l$, obtenemos

$$a^l = \sigma(z^l),$$

siendo w^l la matriz de los pesos para cada capa l cuyas entradas serán los w^l_{jk}, es decir, $w^l = \left\{w^l_{jk}\right\}$ siendo j, k las filas y las columnas respectivamente y el vector b^l como el vector de sesgos de la capa l cuyas coordenadas son los b^l_j.

Notamos que esta ecuación nos relaciona la salida de la capa l con los pesos, el sesgo y la salida de la capa $l-1$. Además, en la Observación 2.4 la función σ está vectorizada, esto es, se aplica la función σ a cada elemento de un vector v, es decir, $\sigma(v)_j = \sigma(v_j)$.

Ejemplo 2.4. Sea la función $f : \mathbb{R} \to \mathbb{R}$ tal que $f(x) = 5x + 3$ y el vector $v = (0, 1, 2) \in \mathbb{R}^3$, entonces

$$f([0, 1, 2]) = [f(0), f(1), f(2)] = [3, 8, 13].$$

Junto a la notación descrita anteriormente, ya se puede introducir las 4 ecuaciones fundamentales del backpropagation que nos ayudaran a computar las derivadas parciales.

Además, en este apartado se van a utilizar las hipótesis de la función coste definidas en el apartado 1.4.1, ya que las vamos a necesitar para poder aplicar todos estos cálculos.

En la siguiente sección se va a utilizar el producto Hadamard. Por ello, voy a introducir brevemente lo que es el producto Hadamard para facilitar su comprensión después.

2.2.2. El producto Hadamard

El producto Hadamard es un producto de matrices donde dos matrices de igual dimensión se multiplican de manera que cada elemento i, j de la matriz resultante es el producto del elemento i, j de la primera matriz por el elemento i, j de la segunda matriz, esto es,

$$(A \odot B)_{i,j} = A_{i,j} \cdot B_{i,j}$$

A esta operación también se le suele llamar producto elemento a elemento. [6]

Presenta diversas propiedades: el producto es asociativo, distributivo y, a diferencia del producto matricial usual, es conmutativo y sólo puede ser utilizado con matrices de la misma dimensión.

Ejemplo 2.5 (Producto Hadamard de dos matrices)**.**

$$\begin{pmatrix} 1 & 0 & 2 \\ 2 & -1 & -2 \\ 3 & 1 & 3 \end{pmatrix} \odot \begin{pmatrix} -1 & 4 & 3 \\ 2 & 2 & -5 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 6 \\ 4 & -2 & 10 \\ 0 & 1 & 0 \end{pmatrix}.$$

Con la teoría explicada del producto Hadamard ya se puede entender lo que sigue.

2.2.3. Las 4 ecuaciones fundamentales del backpropagation.

En este apartado se van a introducir las ecuaciones fundamentales del Backpropagation. [9] Dichas ecuaciones nos proporcionan un método para poder calcular el gradiente de la función coste mediante el cálculo de derivadas que sí conocemos y podemos hacer.

Para poder introducir las 4 ecuaciones fundamentales del backpropagation necesitamos introducir un concepto llamado error de una neurona [9].

Definición 2.5. Definimos el error de la neurona j-ésima en la capa l como sigue:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}, \quad \text{con } l \in \{1, \ldots, L\}.$$

Asimismo, usaremos la notación δ^l para referirnos al vector de errores de las neuronas de la capa l, es decir

$$\delta^l = \left(\delta_1^l, \ldots, \delta_n^l\right), \text{ siendo } n \text{ el número de neuronas en la capa } l \in \{1, \ldots, L\}.$$

El backpropagation nos va a proporcionar una manera de computar el error δ_j^l y de relacionarlo con $\frac{\partial C}{\partial w_{jk}^l}$ y con $\frac{\partial C}{\partial b_j^l}$.

Teorema 2.2 (La ecuación del error en la capa de salida, δ^L)**.** *Sea δ^L el vector de errores de la capa de salida. Entonces las componentes del vector δ^L vienen dadas por*

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{2.2.1}$$

Supongamos que la función coste depende muy poco de una salida de la neurona, es decir, $\frac{\partial C}{\partial a_j^L}$ es muy pequeño, entonces el error δ_j^L será pequeño también y análogamente, si la función coste depende mucho de una salida de la neurona, entonces el error δ_j^L será grande. Por lo tanto, podemos decir que el término $\frac{\partial C}{\partial a_j^L}$ nos mide la velocidad con la que cambia el error como función de la salida de la capa j.

Notamos que esta ecuación está escrita componente a componente. Para aplicar backpropagation, pretendemos utilizar ecuaciones escritas en forma matricial, luego hay que escribir esta ecuación en forma matricial y, para ello, es necesario utilizar el producto Hadamard. El vector formado por $\frac{\partial C}{\partial a_j^L}$ es un vector en las que en cada componente aparece una derivada de la función coste respecto de cada salida neuronal y esto, no es más que el gradiente de la función coste respecto a las salidas neuronales, es decir

$$(\nabla_{a^L} C)_j = \frac{\partial C}{\partial a_j^L},$$

luego la ecuación 2.2.1 se puede reescribir de forma matricial como

$$\delta^L = \nabla_{a^L} C \odot \sigma'(z^L).$$

Demostración. Para demostrar esta fórmula, nos remitimos a la definición de $\delta_j^L = \frac{\partial C}{\partial z_j^L}$. Aplicando regla de la cadena, obtenemos

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L},$$

donde k se mueve entre la primera y la última neurona de la capa L.

Como $a_i^L = \sigma\left(\sum_k w_{ik}^l a_k^{l-1} + b_i^l\right) = \sigma\left(z_i^l\right)$, notamos que a_k^L sólo depende de $z_j^L \Leftrightarrow k = j$, luego $\frac{\partial a_k^L}{\partial z_j^L} = 0$ cuando $k \neq j$. Así, simplificamos la ecuación a

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}.$$

Hemos visto antes que $a_j^L = \sigma(z_j^L)$ y, derivando respecto z_j^L en ambos lados de la igualdad

obtenemos que $\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L)$. Así, obtenemos la primera ecuación del backpropagation:

$$\boxed{\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L).}$$

□

Teorema 2.3 (La ecuación del error δ^l en términos del error de la siguiente capa, δ^{l+1})**.** *Se cumple la siguiente propiedad*

$$\delta^l = ((w^{l+1})^T \cdot \delta^{l+1}) \odot \sigma'(z^l) \tag{2.2.2}$$

Esta ecuación nos proporciona el error en la capa l sabiendo el error en la capa $l+1$.

Combinando la ecuación 2.2.1 y la ecuación 2.2.2, podemos computar el error δ^l de cualquier capa de la neurona. Primero obtenemos el error de la última capa δ^L con la ecuación 2.2.1 y luego utilizamos la ecuación 2.2.2 para obtener, a partir del error de la capa L, el error de la capa $L-1$ y así sucesivamente [9].

Demostración. Para demostrar la fórmula, hay que escribir δ_j^l en términos de δ_k^{l+1} que son los errores de la siguiente capa. Para hacer esto, utilizamos, de nuevo, la regla de la cadena

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1}.$$

Como, por definición, $z_k^{l+1} = \sum_j w_{kj}^{l+1}a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1}\sigma(z_j^l) + b_k^{l+1}$, derivando respecto z_j^l obtenemos que

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'(z_j^l)$$

y, sustituyendo en la ecuación anterior,

$$\boxed{\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma'(z_j^l),}$$

que coincide con la ecuación escrita componente a componente.

□

Corolario 2.1 (La ecuación de la velocidad del cambio del error respecto a cualquier sesgo de la red neuronal)**.** *Se puede computar las derivadas parciales de la función coste respecto a cualquier sesgo como*

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{2.2.3}$$

Esta fórmula nos afirma que la parcial de la función coste respecto a un sesgo de una neurona en la capa l es lo mismo que el error de esa neurona en la misma capa l.

Esto nos proporciona cuánto vale la parcial de la función coste con respecto a cualquier sesgo, que es justo lo que necesitábamos para calcular el gradiente de la función coste, ya que el error δ_j^l sabemos calcularlo utilizando la ecuación 2.2.1 y 2.2.2 del backpropagation.

Escribiendo la ecuación en forma matricial, obtenemos

$$\frac{\partial C}{\partial b} = \delta.$$

Demostración. Para demostrar esta fórmula, basta aplicar la regla de la cadena.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l}.$$

Como, por definición, $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$, al derivar respecto de b_j^l nos queda tal que

$$\frac{\partial z_j^l}{\partial b_j^l} = 1,$$

luego al sustituir esto en la fórmula anterior obtenemos

$$\boxed{\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l.}$$

□

Corolario 2.2 (La ecuación de la velocidad del cambio del error respecto a cualquier peso de la red neuronal)**.** *También se pueden computar las parciales de la función coste respecto a cualquier peso de la red neuronal como sigue*

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{2.2.4}$$

Esta ecuación nos proporciona una forma de calcular la parcial de la función coste respecto a cualquier peso. La parte de la derecha de la igualdad se puede obtener fácilmente aplicando las ecuaciones 2.2.1 y 2.2.2.

Demostración. Para demostrar esta fórmula, de nuevo, basta aplicar la regla de la cadena,

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l}.$$

Como, por definición, $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$, al derivar respecto de w_{jk}^l nos queda tal que

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1},$$

luego al sustituir en la fórmula anterior obtenemos

$$\boxed{\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j.}$$

□

Luego, construyendo una tabla resumen, las 4 ecuaciones del Backpropagation son:

Ecuaciones fundamentales del Backpropagation			
Ecuación 1 (2.2.1)	Ecuación 2 (2.2.2)	Ecuación 3 (2.2.3)	Ecuación 4 (2.2.4)
$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j)$	$\delta^l = ((w^{l+1})^T \cdot \delta^{l+1}) \odot \sigma'(z^l)$	$\frac{\partial C}{\partial b^l_j} = \delta^l_j$	$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$

2.2.4. El algoritmo del backpropagation

Gracias a las ecuaciones descritas anteriormente, ya se puede explicar de qué trata el algoritmo. Consiste en 5 pasos:

1. Introducir la entrada x: Introducimos la entrada $x = a^1$ de la capa inicial.
2. Cálculo de las salidas de la red neuronal: Para cada entrada, computamos $z^l = w^l a^{l-1} + b^l$ y aplicamos la función sigmoide $a^l = \sigma(z^l)$ y con ello obtenemos la construcción de la red neuronal.
3. Cálculo del error de la última capa δ^L: Computamos el error δ^L gracias a la ecuación 2.2.1 del backpropagation.
4. Cálculo de los demás errores a partir del error de la última capa: Para cada capa l desde la capa final L a la capa inicial, computamos su error δ^l_j gracias a la ecuación 2.2.2 del backpropagation.
5. Obtención del gradiente de la función coste: Obtenemos el gradiente de la función coste utilizando las ecuaciones 2.2.3 y 2.2.4: $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ y $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Este algoritmo obtiene el nombre de backpropagation porque a partir del error de la última capa podemos obtener todos los demás. [9]

3 Redes Neuronales Artificiales en el tratamiento de imágenes

En este capítulo se explicará la diferencia entre Redes Neuronales Convolucionales y las Redes Neuronales Artificiales. Por ello, explicamos la definición de redes neuronales convolucionales, el efecto *Pooling* que es bastante útil en el tratamiento de imágenes y la función *Softmax* que nos ayudará a producir una salida. Además, utilizaremos un código de [1] para clasificar ropa y lo aplicaremos a imágenes de ropa propias.

Para esta sección se ha utilizado la fuente [16].

En la actualidad, en el ejemplo de reconocimiento de caracteres o clasificación de imágenes se utilizan redes neuronales convolucionales. Estas Redes Neuronales Convolucionales (CNN) se utilizan más que las Redes Neuronales Artificiales ya que proporcionan varias ventajas frente a estas. Algunas de estas ventajas que proporcionan las redes convolucionales frente a las artificiales son:

- **Reducir el número de pesos a inicializar**.

  Hace unos años la entrada de la red neuronal (que solía ser una imagen en blanco y negro) se representaba como un vector de dimensión 224x224, es decir, un vector que pertenece a $\mathbb{R}^{50\,176}$. En la actualidad, no se usan imágenes en blanco y negro sino que se utilizan imágenes en escala de color RGB (Rojo, Verde y Azul), luego la entrada sería un vector de iguales dimensiones salvo multiplicado por 3 (3 colores posibles), es decir de dimensión 224x224x3 $= 150\,528$ entradas.

  Además, la mayor cantidad de redes neuronales de la actualidad utilizan $2^{10} = 1024$ neuronas en la capa intermedia, luego se tendrían que inicializar aproximadamente $150\,528$ x $1024 = 154\,140\,672$ pesos, es decir, más de 150 millones de pesos. Esto requiere de ordenadores con mucha potencia para poder inicializar estas redes neuronales, por lo tanto, las Redes Neuronales Convolucionales reducen ese número de pesos a inicializar, de forma que un ordenador sin excesiva potencia pueda inicializar y computar estas redes neuronales.

- **Las posiciones pueden cambiar**.

  Supongamos que queremos construir una red neuronal artificial que identifique si es un perro o un gato. Esa red neuronal artificial debería identificar si es un perro o un gato independientemente del fondo, posición y lugar donde esté el perro, pero si colocamos el mismo perro con otro fondo completamente distinto, la red neuronal puede cambiar completamente. Por ello, se utilizan las CNN junto con un efecto llamado *Pooling*, que nos permite simplificar la entrada de la Red Neuronal Convolucional en una entrada más simple que contenga la información importante y necesaria para el cálculo de la salida.

Una red neuronal convolucional es un tipo red neuronal artificial compuesta por tres bloques: la capa convolucional, la capa del efecto *Pooling* y la última capa de salida.

Este descubrimiento fue bastante útil en varias aplicaciones, principalmente en el procesamiento de imágenes. La red convolucional que creó Yann LeCun se basaba en el reconocimiento de letras manuscritas, es decir, crear un programa que pueda reconocer los caracteres escritos a mano mediante una entrada (fotos de caracteres manuscritos).

Las Redes Neuronales Convolucionales sirven tanto para poder transformar una imagen como para predecir ciertas cualidades de la imagen. Para poder transformar imágenes y tratar con ellas, es necesario poder introducirlas en la red neuronal en forma de números reales.

Este problema se arregló utilizando la escala de grises, dividiendo la imagen en píxeles y asignando a cada píxel un número entre 0 y 255, representando por 0 el negro y el 255 el blanco. Para poder transformar las imágenes de entrada en matrices y poder operar con ellas, utilizamos la escala de grises (representando 0 por el negro y 255 por blanco). Por ejemplo:

Ejemplo 3.1. Veamos un ejemplo de una foto de Abraham Lincoln en escala de grises transformada en matriz de números reales

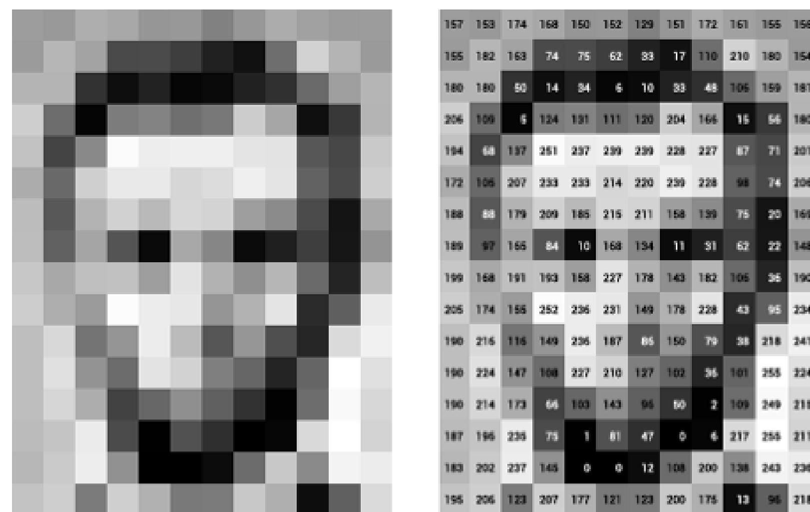


Figura 3.1: Imagen Abraham Lincoln en forma de matriz. Imagen obtenida en [5]

Como se puede apreciar en la Figura 3.1, todos los números de la imagen están entre 0 y 255. Además, los píxeles más oscuros representan números más bajos y los píxeles más claros representan números más altos.

Además del proceso de transformar imágenes en escala de grises a matrices reales, existe también el proceso de transformar matrices de números reales a imágenes en escala de grises.

Una vez tenemos una matriz de números reales entre 0 y 255 (que proviene de una imagen), hay que operar con ella para producir una salida. Para ello, las Redes Neuronales Convolucionales utilizan un objeto llamado *filtro* el cual se va a utilizar para poder transformar la imagen.

Este filtro se opera mediante una operación similar a la convolución o al producto Hadamard. Veamos un ejemplo en el que se ve claramente como se utiliza el filtro:

Ejemplo 3.2. Supongamos que tenemos una imagen 4x4 en escala de grises que al representarla como matriz obtenemos

$$\begin{bmatrix} 0 & 140 & 200 & 0 \\ 0 & 0 & 172 & 0 \\ 0 & 185 & 210 & 180 \\ 0 & 0 & 190 & 0 \end{bmatrix}$$

y supongamos que tenemos el filtro 3x3

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}.$$

La capa en la que la matriz se opera con el filtro para producir otra matriz se le llama *capa convolucional*.

Para poder aplicar el filtro a la imagen, hay que dividir la matriz 3.2 en cajas del mismo tamaño que el filtro, es decir, dividir la matriz 3.2 en las submatrices

$$\begin{bmatrix} 0 & 140 & 200 \\ 0 & 0 & 172 \\ 0 & 185 & 210 \end{bmatrix}, \quad \begin{bmatrix} 140 & 200 & 0 \\ 0 & 172 & 0 \\ 185 & 210 & 180 \end{bmatrix}, \quad \begin{bmatrix} 0 & 0 & 172 \\ 0 & 185 & 210 \\ 0 & 0 & 190 \end{bmatrix}, \quad \begin{bmatrix} 0 & 172 & 0 \\ 185 & 210 & 180 \\ 0 & 190 & 0 \end{bmatrix}.$$

Una vez tenemos unas submatrices de la misma dimensión que el filtro tenemos que hacer el producto Hadamard de esas matrices y, una vez obtenida una matriz del producto Hadamard, sumar todas sus entradas.

Por ejemplo, si tomamos la primera submatriz y aplicamos el filtro 3x3 obtenemos la matriz

$$\begin{bmatrix} 0 & 140 & 200 \\ 0 & 0 & 172 \\ 0 & 185 & 210 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 200 \\ 0 & 0 & 344 \\ 0 & 0 & 210 \end{bmatrix}. \Rightarrow \text{La suma es: } 754.$$

Ahora colocamos este resultado en la posición 1.1 de la matriz salida.

Luego la matriz salida será

$$\begin{bmatrix} 754 & ? \\ ? & ? \end{bmatrix}.$$

Luego repetimos el proceso con las demás submatrices

$$\begin{bmatrix} 140 & 200 & 0 \\ 0 & 172 & 0 \\ 185 & 210 & 180 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -140 & 0 & 0 \\ 0 & 0 & 0 \\ -185 & 0 & 180 \end{bmatrix}. \Rightarrow \text{La suma es: } -145.$$

$$\begin{bmatrix} 0 & 0 & 172 \\ 0 & 185 & 210 \\ 0 & 0 & 190 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 172 \\ 0 & 0 & 420 \\ 0 & 0 & 190 \end{bmatrix}. \Rightarrow \text{La suma es: } = 782.$$

$$\begin{bmatrix} 0 & 172 & 0 \\ 185 & 210 & 180 \\ 0 & 190 & 0 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ -370 & 0 & 360 \\ 0 & 0 & 0 \end{bmatrix}. \Rightarrow \text{La suma es: } -10.$$

Luego el resultado de la convolución de la imagen con el filtro es:

$$\begin{bmatrix} 754 & -145 \\ 782 & -10 \end{bmatrix}.$$

En el tratamiento de imágenes con Redes Convolucionales, usualmente se utiliza el filtro Sobel vertical. Este filtro es el filtro anteriormente utilizado

-1	0	1
-2	0	2
-1	0	1

Figura 3.2: Filtro Sobel Vertical. Imagen obtenida en [16]

Además de este filtro, también existe el filtro Sobel horizontal. Este filtro es el filtro

1	2	1
0	0	0
-1	-2	-1

Figura 3.3: Filtro Sobel Horizontal. Imagen obtenida en [16]

Como se ha podido comprobar en el ejemplo, dependiendo de la dimensión del filtro y de la matriz de la imagen, la dimensión de la matriz resultante puede variar mucho. Como el objetivo de las redes neuronales convolucionales es producir otra imagen de salida, nos gustaría que la matriz resultante tenga la misma dimensión que la matriz de la imagen entrada.

Para ello, la imagen entrada se suelen introducir más entradas de 0 alrededor de la matriz para producir una matriz salida de la misma dimensión. Este proceso se llama *Pooling*.

3.1. Salida de la Red Neuronal Convolucional.

En esta sección se ha utilizado la fuente [17].

En las Redes Neuronales Convolucionales se pueden utilizar numerosas funciones de activación para producir una salida. La función más utilizada es la función Softmax. Esta función nos va a proporcionar la salida verdadera de la CNN.

Definición 3.1. La función Softmax es una función que toma valores reales y los convierte en probabilidad. Denotaremos a la función Softmax como s.

Entonces, la función Softmax es una función $s_i : \mathbb{R}^n \to [0,1]$ tal que

$$s_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \quad x = (x_1, \ldots, x_n), \; i = 1, \ldots, n.$$

Ejemplo 3.3. Supongamos que tenemos la salida de la Red Neuronal Convolucional $x = [-2, 0, 2, 4]$. Entonces la salida de la función Softmax es:

$$s_1(x) = \frac{e^{-2}}{e^{-2} + e^0 + e^2 + e^4} = 0.002.$$
$$s_2(x) = \frac{e^{0}}{e^{-2} + e^0 + e^2 + e^4} = 0.015.$$
$$s_3(x) = \frac{e^{2}}{e^{-2} + e^0 + e^2 + e^4} = 0.117.$$
$$s_4(x) = \frac{e^{4}}{e^{-2} + e^0 + e^2 + e^4} = 0.864.$$

Como se puede apreciar, el denominador es constante, lo único que varía es la componente x_i. Cuanto mayor es el denominador, mayor es la salida de la función Softmax y al ser la exponencial una función creciente implica que cuanto mayor sea x_i mayor será la probabilidad.

3.2. Aplicación de redes convolucionales en la clasificación de imágenes de ropa.

En esta sección he utilizado el código y la información de [1]

Uno de los problemas más típicos en las redes neuronales convolucionales es la creación de redes neuronales convolucionales para reconocer y predecir imágenes de ropa. Este ejemplo es similar al reconocimiento de caracteres manuscritos.

Este ejemplo consiste en crear una red neuronal convolucional en la cual se le pueda importar una base de imágenes (en este caso las vamos a importar del MNIST), estas imágenes se transforman en matrices de números reales entre 0 y 1 y mediante cálculos internos de la red neuronal, que produzca un vector de números reales que pertenezca a $\mathbb{R}^{10}$.

Este vector pertenecerá a $\mathbb{R}^{10}$ ya que se va a intentar clasificar las imágenes en 10 etiquetas distintas, en las cuales ninguna imagen puede pertenecer a dos o más clases a la vez.

Para ello, se van a utilizar 60 000 imágenes para entrenar la red neuronal y más adelante, se introducirán 10 000 imágenes para predecir su tipo.

El código de Python debe empezar por cargar las librerías las cuales se van a utilizar (Tensorflow y Keras para poder descargar el set de imágenes, Numpy para poder realizar álgebra básica y matplotlib.pyplot para poder introducir gráficos que ayuden a la comprensión). Esto se puede realizar mediante el código

```
import tensorflow as tf
from tensorflow import keras

import numpy as np
```

```
import matplotlib.pyplot as plt
```

Una vez tenemos las librerías cargadas, tenemos que descargar las imágenes del MNIST junto con sus respectivos tipos de ropa y, para ello utilizamos

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images,train_labels),(test_images, test_labels)=
      fashion_mnist.load_data()
```

En este código se han almacenado en *train_images* y en *test_images* las respectivas matrices de números procedientes de imágenes de la base de datos, en train las que se utilizarán para entrenar (60 000 imágenes) y en test las que se utilizarán para predecir (10 000 imágenes). Además, en *train_labels* y *test_labels* se han almacenado las respectivas salidas verdaderas que debería producir la red neuronal de las imágenes en train y en test.

Como *train_labels* y *test_labels* son vectores de números entre 0 y 9, para hacerlo más visual se va a crear una variable que almacenará los tipos de ropa en los que se va a clasificar, estos son:

```
class_names = ['Camiseta', 'Pantalon', 'Sudadera',
'Vestido', 'Abrigo', 'Chancla', 'Camisa',
'Deportiva', 'Mochila', 'Bota']
```

Las imágenes descargadas del MNIST son imágenes de 28x28 píxeles, la única diferencia es que no están en escala de grises. Para reducirla a escala de grises, dividimos entre 255 todas las imágenes para así reducirlo a números entre 0 y 1, representando por 0 el color negro y el 1 el color blanco.

```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

Una vez tenemos las imágenes preparadas para el entrenamiento, es hora de crear la red neuronal convolucional. En este caso, se va a crear una red con $28 \cdot 28 = 784$ entradas (1 por cada píxel), 128 neuronas en la capa oculta y 10 salidas, las cuales nos dirá a qué tipo de ropa corresponde cada imagen.

```
model = keras.Sequential([
  keras.layers.Flatten(input_shape=(28, 28)),
  keras.layers.Dense(128, activation='relu'),
  keras.layers.Dense(10, activation='softmax')
])
```

A priori puede parecer un código complicado. En la segunda línea, keras.layers.Flatten crea una capa de neuronas las cuales no necesitan entrenamiento (ya que son entradas).

En la tercera y cuarta línea, utiliza el argumento keras.layers.Dense ya que utiliza parámetros los cuales van a requerir de aprendizaje, sumado al número de neuronas que hay en su respectiva capa y junto con la función activación a realizar. En la última capa, como es conveniente, se utiliza la función *Softmax* para poder transformar las salidas en probabilidades.

Una vez creada la red neuronal, hay que entrenarla. Para ello, utilizaremos un método de optimización distinto al visto y la función coste Cross Entropy:

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

model.fit(train_images, train_labels, epochs=10)
```

La primera parte del fragmento del código se inicializa el optimizador a realizar (en este caso Adam), la función coste a evaluar (en este caso la entropía cruzada) y la variable metrics inicializa la precisión que se va a adquirir.

Estas variables pueden ser cambiadas: en el apartado de optimizer podemos cambiarlo por "SGD"para que utilice el Descenso Gradiente Estocástico y en "loss"podemos cambiarlo por "MeanSquaredError"que es la función coste introducida en el apartado .

En este ejemplo, lo más preciso y lo que da más resultados es utilizar el optimizador Adam y la función entropía cruzada. Debido a su complejidad, esto se ha comprobado experimentalmente y no tiene fundamento teórico que lo demuestre.

Con este código ejecutado obtenemos la salida final

```
Epoch 10/10
1875/1875 [==============================]
- 5s 3ms/step
- loss: 0.2418
- accuracy: 0.9084
```

Esta salida nos informa de que la pérdida alcanzada ha sido un 0.2418 tras haber entrenado la red neuronal, luego la precisión de la red neuronal es de un 90.84 %.

Una vez tenemos la red neuronal entrenada, ya está capacitada para poder predecir el tipo de una prenda de ropa.

Supongamos que tenemos una imagen no entrenada cualquiera de la base de datos del MNIST (*test_images*) como es la siguiente

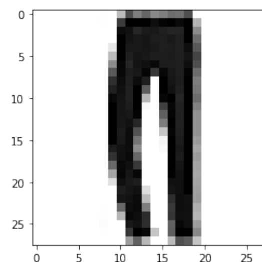


Figura 3.4: Imagen de los pantalones a deducir. Imagen obtenida de una salida del código de la fuente [1]

Claramente, a la vista de un humano, la imagen corresponde con un pantalón.

Utilizando el código

```
predictions = model.predict(test_images)
```

e imprimiento la salida de la imagen correspondiente

```
predictions[2]
```

obtenemos que el pantalón nos produce una vector

```
array([3.5169654e-12, 1.0000000e+00, 6.8531525e-16,
1.0174891e-11, 5.5219322e-13, 3.2149909e-21,
2.7200011e-13, 3.9382509e-27,
7.4258752e-15, 1.9075688e-20],
dtype=float32)
```

Notamos que el número más alto del vector es el número en la posición 2 (posición 1 en Python), luego la imagen corresponde con la etiqueta número 2. Si imprimimos el tipo de ropa correspondiente a la posición 1 del vector *class_names*, vemos que la salida por pantalla nos afirma que es un pantalón:

```
print(class_names[1])
```

```
Pantalon
```

Finalmente, si tratamos de representarlo mediante un gráfico, obtenemos la salida de manera visual

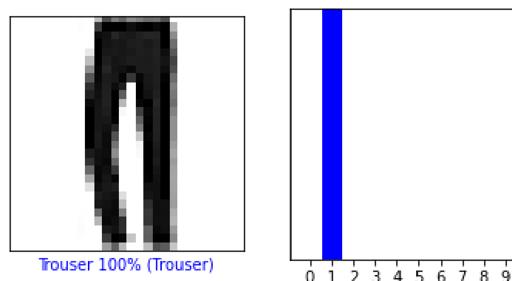


Figura 3.5: Predicción de la red neuronal. Imagen obtenida de la salida del código de [1]

que, como podemos comprobar, la red neuronal apenas ha dudado de que son unos pantalones.

3.2.1. Ejemplo experimental de clasificación de ropa.

En la sección anterior he utilizado imágenes de la base de datos del MNIST para predecir las prendas de ropa. En base a esto, he decidido fotografiar una camiseta y unos pantalones propios y probar el código con estas imágenes.

Como la cámara fotográfica utilizada es la del móvil y esta no produce imágenes de 28*x*28 píxeles, he tenido que reducirla manualmente. Una vez reducido a 28*x*28 píxeles, usando el código de [1] junto con ciertas modificaciones mías, he obtenido un resultado satisfactorio.

Voy a explicar paso por paso la modificación del código junto con las imágenes que he utilizado.

(a) Fotografía de la camiseta propia.

(b) Fotografía de unos pantalones propios.

Figura 3.6: Fotografías realizadas para predecir

Ambas imágenes las he cargado en el directorio en el que estoy trabajando. Para producir una salida correcta, es necesario utilizar el código de la anterior sección (obtenido en [1]) y entrenar la red neuronal con la base de datos del MNIST. Una vez entreado la red neuronal, introducimos la imagen en el código mediante el código

```
import numpy as numpy
from PIL import Image

imagen=Image.open("foto.jpg")
imagen=np.asarray(imagen)
```

Este código utiliza las librerías necesarias para cargar la imagen (llamada "foto.jpg") desde el directorio de trabajo al código y transformarla en una matriz.

Como la imagen introducida no es en escala de grises, he tenido que transformarla en una imagen a escala de grises.

Una vez introducido la imagen en escala de grises, utilizamos la red neuronal para predecirla:

```
imagen=(np.expand_dims(arrayimagen,0))

prediccion=model.predict(imagen)
print(prediccion)
```

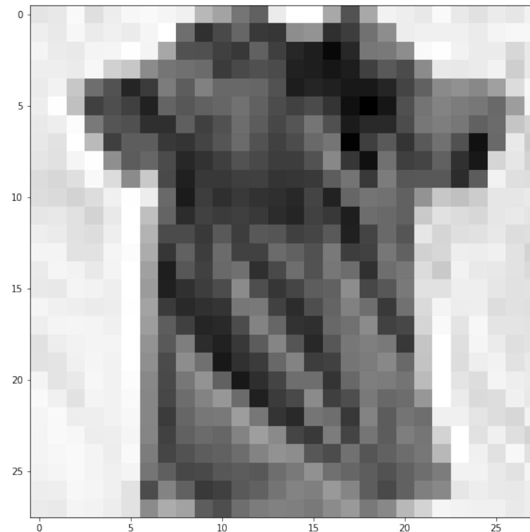
La primera línea del código anterior sirve para convertir la matriz 28x28 en una matriz de matrices, a pesar de que solo haya una matriz ya que el programa trabaja con matrices. Es decir, he transformado la matriz 28x28 en una matriz de una única matriz 28x28. Así, esta nueva matriz tendrá tamaño 1x28x28.

3.2.2. Predicción de camiseta

Mediante el código

```
plt.figure(figsize=(10,10))
plt.imshow(img, cmap=plt.cm.binary)
img=np.array(img)
```

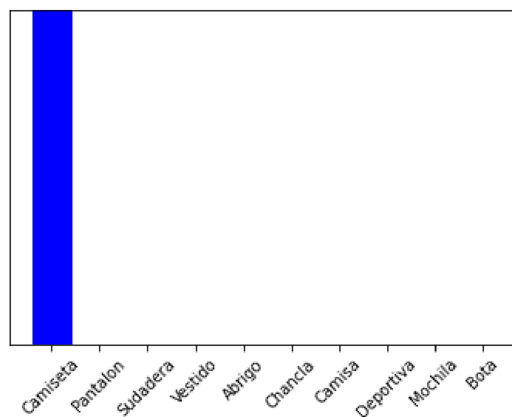
obtengo la impresión de la imagen de la camiseta en blanco y negro con $28x28$ píxeles



y al utilizar el fragmento de código que predice obtenemos la salida

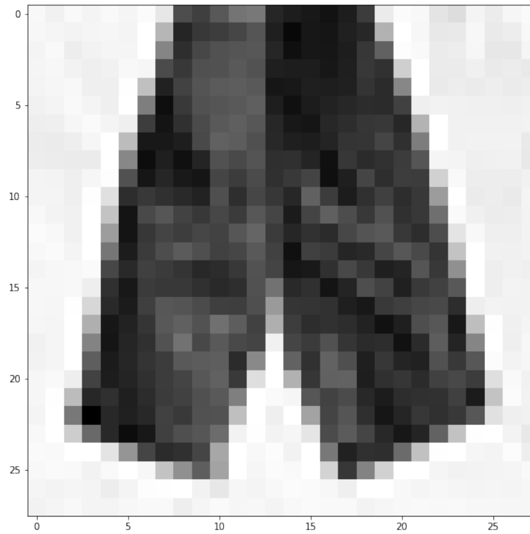
```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Luego, la etiqueta en la posición 1 (0 en Python) es camiseta, como se puede apreciar en el siguiente gráfico



3.2.3. Predicción de pantalones

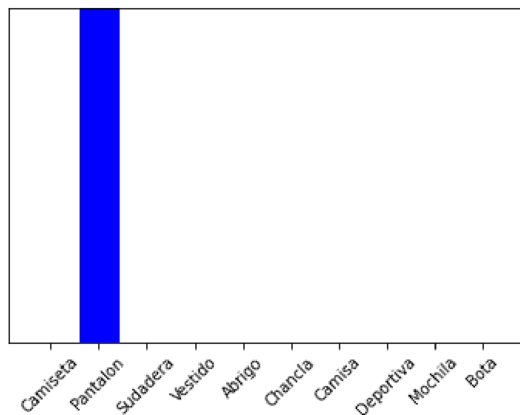
Mediante el código utilizado en el ejemplo de la camiseta para imprimir la imagen en blanco y negro obtengo



y al utilizar el framgento de código que predice obtenemos la salida

```
[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Luego, la etiqueta en la posición 2 (1 en Python) es pantalón, como se puede apreciar en el siguiente gráfico



4 Redes Neuronales en el reconocimiento de caracteres manuscritos

En este capítulo final aplicaremos la teoría explicada en el capítulo 1 y capítulo 2 en un ejemplo de reconocimiento de caracteres manuscritos. Emplearemos un código del libro [9] para ello y en él podremos apreciar cómo emplea el Descenso Gradiente Estocástico y el algoritmo Backpropagation.

En esta sección he utilizado la información y el código de [9].

El ejemplo de reconocimiento de caracteres consiste en introducir una imagen y la red neuronal nos devuelve una salida en forma de vector en el cual la posición del número más proximo a 1 de la salida será el número elegido (recuerdo que *Python* enumera desde el 0, luego si la posición más próxima al 1 es la posición 2, el número será el 3). Ese vector pertenecerá a $\mathbb{R}^{10}$, cada número desde el 0 al 9.

En la práctica, en este ejercicio se utilizan imágenes de números de varias cifras (no tienen por qué solo tener 1 cifra del 0 al 9), luego puede parecer que esta red neuronal no sea de mucha utilidad para este ejemplo, ya que sólo produce salidas de números correspondientes del 0 al 9. Por lo tanto, para que esta red neuronal sea útil en nuestro ejemplo, se emplea un algoritmo el cual separa una imagen de números de más de una cifra en varias imágenes de números de 1 cifra. A este problema se le llama el *problema de segmentación*. Este problema para un humano es bastante sencillo, pero para un ordenador puede ser una tarea bastante ardua.

En este ejemplo, unicamente nos vamos a centrar en el reconocimiento de caracteres (clasificación de dígitos), luego no vamos a abarcar el problema de segmentación. Así que, a partir de ahora, vamos a suponer que la imagen que introducimos es de un número de una única cifra.

Las imágenes que vamos a introducir van a ser imágenes de tamaño 28 por 28 píxeles, luego la red neuronal tendrá en total $28 \cdot 28 = 784$ entradas y estas se agruparán en un vector $x \in \mathbb{R}^{784}$. Este vector será el que se utilizará como entrada de la red neuronal. Un esquema de la red neuronal que se va a utilizar es:

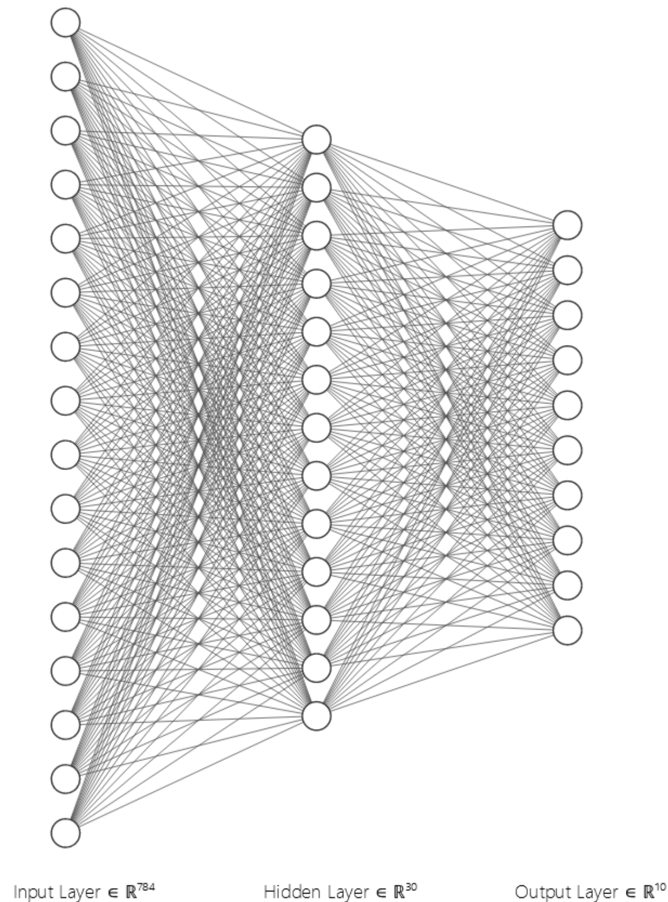


Figura 4.1: Red Neuronal a utilizar, con 784 entradas, 30 neuronas en la capa oculta y 10 salidas.

El vector de 784 entradas tendrá números entre 0 y 1, siendo 0 el blanco total y siendo 1 el negro total, junto con los valores intermedios representando gradualmente la oscuridad en una escala de grises.

Respecto a la cantidad de neuronas en la capa oculta, puede variar bastante. En este caso, se utilizarán 30 neuronas en la capa oculta.

Para poder aplicar este ejemplo, me he ayudado del programa ya creado en *Python* de reconocimiento de caracteres. Este programa se puede descargar en GitHub o se puede encontrar en el libro [9].

Las imágenes que se van a utilizar son imágenes obtenidas del MNIST, de las cuales se han utilizado 50.000 para entrenar y 10.000 para probar el programa y calcular su precisión.

Para reproducir este ejemplo, me he ayudado del código de *Python* del libro [9].

Para poder utilizar algebra lineal y algebra básica (producto de matrices, la función exponencial, etc.), se ha empleado una librería anteriormente utilizada llamada *Numpy* y una función llamada *random* que nos ayudará a inicializar los pesos y sesgos aleatoriamente. Luego, el primer paso es cargarlas en el código y lo haremos mediante los comandos:

```
import random
import numpy as np
```

Creamos una clase llamada **Network** la cual nos creará la red neuronal que queramos, con las entradas, capas intermedias y salidas que queramos. Para ello, hay que usar un bucle *for* e inicializar los pesos y sesgos de manera aleatoria.

```
class Network(object):
def __init__(self, sizes):
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.biases = [np.random.randn(y, 1)
            for y in sizes[1:]]
    self.weights = [np.random.randn(y, x)
            for x, y in zip(sizes[:-1], sizes[1:])]
```

En el código anterior, el vector *sizes* almacenará el número de neuronas de cada capa. Si por ejemplo, utilizamos el código

```
        neurona = Network([4,3,1])
```

crearemos una red neuronal que tendrá 3 capas, en las cuales tendrá 4 entradas, 3 neuronas ocultas y una neurona de salida.

Los pesos y los sesgos de la red neuronal se han inicializado de manera aleatoria usando una función correspondiente a la librería *Numpy* llamada *np.random.randn*, el cual genera números aleatorios mediante una distribución Gaussiana con media 0 y desviación típica 1.

También notamos que los pesos y sesgos están guardadas en matrices. Por ejemplo, si utilizamos el código

```
neurona.weights[0]
```

obtendremos una matriz la cual guarda los pesos que se utilizan para conectar la primera y la segunda capa de neuronas. Esta matriz es la que en el apartado de backpropagation se llamaba w la cual guardaban los pesos w_{jk} que conectaba la neurona k de la capa $l-1$ con la neurona j de la capa l.

Observación 4.1. Como se puede apreciar en el código anterior, se han inicializado únicamente los pesos y sesgos para la capa oculta y las neuronas de la capa de salida, ya que las neuronas de entrada no requieren de pesos y sesgos.

Una vez la red neuronal está inicializada, vamos a definir tanto la función sigmoidea como su derivada como se definió en el Capítulo 1.

```
def sigmoid(z):
    return 1/(1+np.exp(-z))
```

```
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

Ahora vamos a definir la función *feedforward* que es la que calculará la salida de cada neurona y, con ello, la salida de la red neuronal. Para ello, se utilizan dos bucle *for*.

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w,a)+b)
return a
```

Con esto, ya podríamos saber la salida de cualquier red neuronal de la cantidad de entradas, neuronas ocultas y neuronas de salida que queramos. Claramente esta red neuronal no está entrenada, entonces puede contener un error bastante elevado. El siguiente paso será entrenar la red neuronal para que nos proporcione el menor error posible.

Para ello, vamos a inicializar el método Descenso Gradiente Estocástico (SGD) para poder calcular el mínimo de la función coste.

```
def SGD(self,training_data,epochs,
        mini_batch_size,eta,test_data=None):
    if test_data:
        n_test = len(test_data)
        n = len(training_data)
    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches=[training_data[k:k+mini_batch_size]
            for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
    if test_data:
        print ("Etapa {0}: {1} / {2}").format
            (j, self.evaluate(test_data), n_test)
    else:
        print ("Etapa {0} complete").format(j)
```

El objeto *training data* es una lista de tuplas (x, y) en las cuales los x corresponden a las entradas de la red neuronal y las y corresponden a las salidas verdaderas que debería producir la red neuronal sobre los datos x. La variable *epochs* simboliza el número de etapas en las que se quiere entrenar y la variable *mini_batch_size* es el tamaño total de los pasos a dar. La variable *eta* corresponde al paso o tasa de aprendizaje y la variable *test_data* es una variable en la que se almacenan los datos para predecir y si se proporciona el programa nos evaluará la neurona tras cada etapa de entrenamiento y nos dará su progreso. Esta variable es bastante útil para ver cómo de buena es la red neuronal pero a la hora de la práctica puede ser un poco lenta y puede ralentizarnos un poco el programa, por lo que se deja por defecto como *None* y si no se proporciona esa variable, el programa lo interpretará como que hemos puesto que *test_data* es None.

Para cada *mini_batch* hay que aplicar un paso del Descenso Gradiente Estocástico. Para ello, se ha empleado un código que cambia los pesos y sesgos en función de la iteración del Descenso Gradiente Estocástico. El código a utilizar será:

```
nabla_b = [np.zeros(b.shape) for b in self.biases]
nabla_w = [np.zeros(w.shape) for w in self.weights]
for x, y in mini_batch:
    delta_nabla_b, delta_nabla_w = self.backprop(x, y)
    nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
```

```
                                delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
                                delta_nabla_w)]
self.weights = [w-(eta/len(mini_batch))*nw for w,
             nw in zip(self.weights, nabla_w)]
self.biases = [b-(eta/len(mini_batch))*nb for b,
             nb in zip(self.biases, nabla_b)]
```

Como se puede observar, el trozo del código anterior

```
self.weights = [w-(eta/len(mini_batch))*nw for w,
        nw in zip(self.weights, nabla_w)]
self.biases = [b-(eta/len(mini_batch))*nb for b,
        nb in zip(self.biases, nabla_b)]
```

es exactamente lo mismo que se explicó en el apartado 2.1.4.1.
Lo más importante de este código es la línea

```
    delta_nabla_b, delta_nabla_w = self.backprop(x,y)
```

Esto utiliza el algoritmo del *backpropagation* que es una manera rápida de computar el gradiente de la función coste. Este código es:

```
def backprop(self, x, y):
nabla_b = [np.zeros(b.shape) for b in self.biases]
nabla_w = [np.zeros(w.shape) for w in self.weights]

activation = x
activations = [x]
zs = []
for b, w in zip(self.biases, self.weights):
    z = np.dot(w, activation)+b
    zs.append(z)
    activation = sigmoid(z)
    activations.append(activation)

delta = self.cost_derivative(activations[-1], y) * \
    sigmoid_prime(zs[-1])
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
#Parte del backpropagation
for l in range(2, self.num_layers):
    z = zs[-l]
    sp = sigmoid_prime(z)
    delta=np.dot(self.weights[-l+1].transpose(),
        delta)*sp
    nabla_b[-l] = delta
    nabla_w[-l]=np.dot(delta,
        activations[-l-1].transpose())
return (nabla_b, nabla_w)
```

Este código devuelve como salida la tupla $(nabla_b, nabla_w)$ la cual representa el gradiente de la función coste. La variable *nabla_b* representa el gradiente respecto a los sesgos capa por capa y *nabla_w* representa el gradiente respecto a los pesos capa por capa.

Una vez hecho esto, ya podemos crear un código que pueda programar una red neuronal, proporcionarnos su salida y entrenarla mediante el Descenso Gradiente Estocástico y el algoritmo del *backpropagation*.

Vamos a crear la red neuronal que vayamos a utilizar para el reconocimiento de caracteres escritos. Para ello, importamos la librería del MNIST y cargamos las imágenes de esa librería. Además, importamos el código que hemos utilizado anteriormente

```
import mnist_loader
training_data,validation_data,test_data
        = mnist_loader.load_data_wrapper()
training_data=list(training_data)

import network

net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Con la línea

```
net=network.Network([784,30,10])
```

creamos una red neuronal con 784 entradas, 30 neuronas ocultas y 10 salidas (del 0 al 9 como se explicó antes).

La línea que entrena la red neuronal es la línea

```
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Esta línea entrena la red neuronal en cada etapa y nos proporciona la salida

```
Epoch 0 : 9089 / 10000
Epoch 1 : 9285 / 10000
Epoch 2 : 9327 / 10000

...

Epoch 27 : 9503 / 10000
Epoch 28 : 9493 / 10000
Epoch 29 : 9483 / 10000
```

Podemos comprobar que en 29 etapas, la red neuronal tiene una eficiencia del 94.83 %, esto es, casi 95 de cada 100 imágenes de dígitos la salida de la red neuronal corresponde con la salida deseada. [9]

En conclusión, se ha conseguido crear una red neuronal artificial capaz de clasificar con bastante precisión (94.83 %) cualquier imagen de un dígito escrito a mano.

Bibliografía

Las referencias se listan por orden alfabético. Aquellas referencias con más de un autor están ordenadas de acuerdo con el primer autor.

[1] Clasificacion basica: Predecir una imagen de moda. https://www.tensorflow.org/tutorials/keras/classification?hl=es-419. [Citado en págs. 29, 33, 35, 36, and 37]

[2] Convolutional neural networks for visual recognition. https://cs231n.github.io/neural-networks-3/. [Citado en pág. 12]

[3] Qué son y cómo funcionan las gan, esas redes neuronales capaces de crear rostros de personas que no existen. https://www.genbeta.com/a-fondo/que-como-funcionan-gan-esas-redes-neuronales-capaces-crear-rostros-personas-que-no-existen. [Citado en pág. 1]

[4] Breve historia de las redes neuronales artificiales. https://www.aprendemachinelearning.com/breve-historia-de-las-redes-neuronales-artificiales/, 2018. [Citado en págs. 3 and 4]

[5] BRITA INTELIGENCIA ARTIFICIAL. Introducción a la visión por computadora: qué es y cómo funciona. https://brita.mx/introduccion-a-la-vision-por-computadora-que-es-y-como-funciona, 2021. [Citado en pág. 30]

[6] Avnish. https://medium.com/linear-algebra/part-14-dot-and-hadamard-product-b7e0723b9133, 2019. [Citado en pág. 23]

[7] Carl T Kelley. *Iterative methods for optimization*. SIAM, 1999. [Citado en págs. 15, 16, 17, and 18]

[8] Damián Jorge Matich. Redes neuronales: Conceptos básicos y aplicaciones. *Universidad Tecnológica Nacional, México*, 41:12–16, 2001. [Citado en pág. 6]

[9] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015. [Citado en págs. 3, 8, 9, 10, 20, 21, 23, 25, 27, 41, 42, and 46]

[10] Boris Polyak. *Introduction to Optimization*. 07 2020. [Citado en pág. 11]

[11] prakharroy. Intuition of adam optimizer. https://www.geeksforgeeks.org/intuition-of-adam-optimizer/. [Citado en págs. 20 and 21]

[12] Harshad Rai and Naman Shukla. Unpaired image-to-image translation using cycle-consistent adversarial networks. 2018. [Citado en pág. 2]

[13] Sefik Ilkin Serengil. The insider's guide to adam optimization algorithm for deep learning. https://sefiks.com/2018/06/23/the-insiders-guide-to-adam-optimization-algorithm-for-deep-learning/, 2018. [Citado en pág. 21]

[14] Tensorflow. Red antagónica generativa convolucional profunda. https://www.tensorflow.org/tutorials/generative/dcgan. [Citado en págs. 1 and 2]

[15] Gabriel Turinici. The convergence of the Stochastic Gradient Descent (SGD) : a self-contained proof, March 2021. [Citado en pág. 19]

[16] Victor Zhou. https://victorzhou.com/blog/intro-to-cnns-part-1/. [Citado en págs. 29 and 32]

[17] Victor Zhou. https://victorzhou.com/blog/softmax/. [Citado en pág. 32]