UNIVERSIDAD REY JUAN CARLOS

ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA INFORMÁTICA


CURSO ACADÉMICO 2019/2020

TRABAJO FIN DE GRADO

# DEEP LEARNING PARA EL RECONOCIMIENTO DE IMÁGENES: ENTRENAMIENTO DE REDES CONVOLUCIONALES MEDIANTE EL ALGORITMO ADAM

GRADO EN INGENIERÍA
INFORMÁTICA

ÁLVARO RUBIO SEGOVIA
MADRID, 2020

E.T.S. DE INGENIERÍA INFORMÁTICA, UNIVERSIDAD REY JUAN CARLOS

# DEEP LEARNING PARA EL RECONOCIMIENTO DE IMÁGENES: ENTRENAMIENTO DE REDES CONVOLUCIONALES MEDIANTE EL ALGORITMO ADAM

**Autor:** Álvaro Rubio Segovia

**Tutor:** César Beltrán Royo

**Departamento:** Escuela Técnica Superior de Ingeniería Informática

**Titulación:** Grado en Ingeniería Informática

**Palabras clave:** deep learning, optimización, reconocimiento de imágenes, redes neuronales, redes neuronales convolucionales, gradiente descente estocástico, ADAM, función de coste, algortimos de optimización.

## Resumen

El siguiente trabajo de fin de grado pretende explicar de manera sencilla, y sin necesidad de conocimientos previos, la teoría necesaria para entender cómo se realiza el reconocimiento de imágenes digitales y los algoritmos de optimización utilizados en su aplicación.

Con el objetivo de asentar los conocimientos básicos de visión artificial e introducir los cimientos de este campo, se explican los términos de inteligencia artificial y aprendizaje automático y se explican las redes neuronales y su funcionamiento básico.

Las redes neuronales son el principal modelo de computación en el deep learning, un subcampo del aprendizaje automático centrado en redes neuronales profundas, es decir, con una gran cantidad de capas y neuronas. Se relatan los beneficios que tiene la implementación de un modelo de IA basado en deep learning y cómo se construye una red neuronal sencilla.

Con estos conocimientos explicados se introducen las redes neuronales convolucionales, redes específicas para el reconocimiento de imágenes. Este tipo de redes permiten guardar o captar las relaciones entre los píxeles de una imagen para extraer información útil y aumentar la eficacia del entrenamiento.

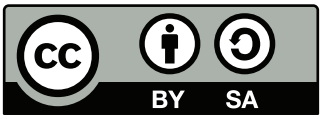
Se explican los diferentes algoritmos de optimización que pueden ser utilizados en el reconocimiento de imágenes y se realiza un experimento para medir los resultados de diferentes modelos de reconocimiento de imágenes con los conjuntos de datos MNIST y CIFAR-10.

Este experimento se ha realizado utilizando el framework Pytorch. Pytorch es una framework de código abierto escrito en Python que facilita la investigación de prototipos basados en redes neuronales. Cuenta con una gran cantidad de manuales y una interfaz sencilla para la creación de modelos de deep learning.

Con los resultados del experimento se ha llevado a cabo una comparativa del rendimiento y desempeño de los diferentes algoritmos en los modelos llegando varias conclusiones o ideas. Estos algoritmos son el gradiente descendente estocástico, con y sin momento, y el algoritmo ADAM.

El algoritmo ADAM es utilizado actualmente por numerosas aplicaciones dado su grado de eficiencia y rapidez a la hora de conseguir modelos óptimos basados en deep learning. Este algoritmo se basa en el uso del momento y tasas de aprendizaje adaptativos para cada parámetros ajustable de la red neuronal.

Se concluye con un resumen de los objetivos conseguidos, observaciones obtenidas del experimento y líneas futuras de investigación.



Agradecer enormemente...
A mi familia... por el apoyo recibido durante la realización del TFG.
Gracias.

*Álvaro Rubio Segovia*

# Acrónimos

| | |
|---|---|
| **ETSII** | Escuela Técnica Superior de Ingeniería Informática |
| **PFC** | Proyecto Fin de Carrera |
| **TFG** | Trabajo Fin de Grado |
| **URJC** | Universidad Rey Juan Carlos |
| **MIT** | Instituto Tecnológico de Massachusetts |
| **IA** | Inteligencia Articifial |
| **ML** | Machine Learning |
| **RAE** | Real Academia de la Lengua Española |
| **ANN** | Artificial Neural Network |
| **ECM** | Error Cuadrático Medio |
| **GD** | Gradiente Descendente |
| **VGD** | Vainilla Gradient Descent |
| **SGD** | Stochastic Gradient Descent |
| **SGDM** | Stochastic Gradient Descent with Momentum |
| **ADAM** | Adaptative Moment Estimation |
| **CPU** | Unidad Central de Procesamiento |
| **GPU** | Unidad de Procesamiento de Gráficos |

# Índice

**Resumen** II

**Acrónimos** V

**Notación** XII

## I Introducción 1

### 1 Introducción y visión general 2

1.1 Motivación . . . . . 2

1.2 Objetivo . . . . . 3

1.3 Estructura del documento . . . . . 4

## II Aprendizaje automático basado en redes neuronales: Deep Learning 6

### 2 Introducción al deep learning y redes neuronales 7

2.1 Inteligencia artificial y aprendizaje automático . . . . . 8

2.1.1 Inteligencia Artificial . . . . . 8

2.1.2 Aprendizaje automático . . . . . 9

2.2 Redes neuronales artificiales . . . . . 11

2.2.1 Inspiración biológica . . . . . 12

2.2.2 Funcionamiento de una neurona . . . . . 14

2.2.3 Jerarquización del conocimiento . . . . . 16
2.3 Introducción al deep learning . . . . . 17
2.3.1 ¿Por qué utilizar deep learning? . . . . . 18

**3 Creación y entrenamiento de redes neuronales 21**
3.1 Pasos en la creación de un modelo basado en redes neuronales de deep learning . . . . . 22
3.1.1 Estructura de la red neuronal . . . . . 23
3.1.2 Función base y de activación . . . . . 24
3.1.3 Función de coste . . . . . 25
3.1.4 Gradiente descendente . . . . . 26
3.1.5 Backpropagation . . . . . 27
3.1.6 Hiperparámetros . . . . . 28
3.2 Ejemplo . . . . . 29
3.2.1 Herramientas . . . . . 29
3.2.2 Resultados . . . . . 29
3.3 Validación y evaluación de redes neuronales . . . . . 32

**III Experimento con redes neuronales: Reconocimiento de imágenes con Pytorch 34**

**4 Redes neuronales para el reconocimiento de imágenes 35**
4.1 Redes neuronales para el reconocimiento de imágenes . . . . . 36
4.1.1 Redes neuronales convolucionales . . . . . 37
4.2 Algoritmos de optimización avanzados . . . . . 37
4.2.1 Gradiente descendente estocástico con momento . . . . . 39
4.2.2 ADAM . . . . . 40
4.3 Tecnologías y librerías utilizadas en el experimentO . . . . . 42
4.3.1 Pytorch. El framework para redes neuronales . . . . . 43
4.3.2 Google Colaboratory. Cuadernos de Python con ejecución en la nube . . . . . 45

4.3.3 TensorBoard. Visualización sencilla de datos . . . . . 45
4.4 Comparativa de SGD, SGD con momento y ADAM con el conjunto de datos MINST . . . . . 46
4.4.1 Datos de trabajo . . . . . 46
4.4.2 Red neuronal convolucional . . . . . 47
4.4.3 Función de coste . . . . . 51
4.4.4 Obtención del modelo óptimo. Aprendizaje y evaluación . . 52
4.5 Comparativa de SGD, SGD con momento y ADAM con el conjunto de datos CIFAR-10 . . . . . 57
4.5.1 Datos de trabajo . . . . . 57
4.5.2 Red neuronal convolucional . . . . . 59
4.5.3 Función de coste . . . . . 62
4.5.4 Obtención del modelo óptimo. Aprendizaje y evaluación . . 62

**IV Conclusión. 69**

**Conclusiones y líneas futuras 70**
4.6 Objetivos conseguidos . . . . . 70
4.7 Observaciones sobre los resultados obtenidos . . . . . 70
4.7.1 Líneas futuras . . . . . 72

**Bibliografía 76**

**V Apéndice 77**

**A Ejemplo de red neuronal sencilla 78**

**B Clasificador de imágenes MNIST 97**

**C Clasificador de imágenes CIFAR-10 122**

**Glosario 154**

# Índice de figuras


2.1 Diagrama de flujo de modelo con aprendizaje supervisado . . . . 11
2.2 Diagrama de flujo de modelo con aprendizaje no supervisado . . . 12
2.3 Imagen de neurona biológica y artificial . . . . . 13
2.4 Imagen de neuronas biológicas y red neuronal artificial . . . . . 13
2.5 Clasificador unineuronal . . . . . 15
2.6 Clasificador con dos neuronas . . . . . 15
2.7 Funciones de activación escalonada y sigmoide . . . . . 17
2.8 Funciones de activación tangente hiperbólica y ReLU . . . . . 18
2.9 Red neuronal profunda . . . . . 20
2.10 Red neuronal sin jerarquización de conocimiento . . . . . 20

3.1 Conjunto de datos de entrenamiento/ puntos del huerto . . . . . 22
3.2 Red neuronal artificial ejemplo . . . . . 23
3.3 Función de coste del modelo SGD (Inicialización A) . . . . . 30
3.4 Función de coste del modelo GD (Inicialización A) . . . . . 30
3.5 Función de coste del modelo GD (Inicialización B). . . . . 31
3.6 Función de coste del modelo SGD (Inicialización B). . . . . 31
3.7 Limites de decisión de los modelos de SGD y GD. Inicialización A 33
3.8 Limites de decisión de los modelos de SGD y GD. Inicialización B 33

4.1 Función de convolución en imágenes. Fuente: Diego Calvo . . . . 38
4.2 Función de max pooling 2x2 en imágenes. . . . . 39

4.3 Puntos importantes en la optimización de funciones. Fuente: Página web 'Off the convex path' . . . . . 39
4.4 Evolución del valor del coste de los modelos de SGD con y sin momento. Fuente: Genevieve B. Orr . . . . . 40
4.5 Ejemplo lote de entrenamiento del conjunto de datos MNIST . . . 47
4.6 Tamaño de la matriz de pesos de la segunda capa convolucional. MNIST . . . . . 48
4.7 Red neuronal convolucional para el conjunto de datos MNIST. Primera capa convolucional . . . . . 49
4.8 Red neuronal convolucional para el conjunto de datos MNIST. Segunda capa convolucional . . . . . 50
4.9 Red neuronal convolucional para el conjunto de datos MNIST. Capas feed forward . . . . . 50
4.10 MNIST. Evolución del coste del lote de entrenamiento por iteración (Smooth = 0,6). Datos de entrenamiento . . . . . 53
4.11 MNIST. Early stopping. . . . . 55
4.12 MNIST. Evolución del coste de entrenamiento y validación por épocas. SGD. Early stopping . . . . . 56
4.13 MNIST. Evolución del coste de entrenamiento y validación por épocas. SGDM. Early stopping . . . . . 56
4.14 MNIST. Evolución del coste de entrenamiento y validación por épocas. ADAM. Early stopping . . . . . 57
4.15 MNIST. Evolución de la precisión por época. Conjunto de datos de validación . . . . . 58
4.16 Ejemplo lote de entrenamiento del conjunto de datos CIFAR-10 . . 59
4.17 Tamaño de la matriz de pesos de la segunda capa convolucional. CIFAR-10 . . . . . 60
4.18 Red neuronal convolucional para el conjunto de datos CIFAR-10. Primera capa convolucional . . . . . 61
4.19 Red neuronal convolucional para el conjunto de datos CIFAR-10. Segunda capa convolucional . . . . . 61
4.20 Red neuronal convolucional para el conjunto de datos CIFAR-10. Tercera capa convolucional . . . . . 62

4.21 CIFAR-10. Evolución del coste por iteración (Smooth = 0,6). Datos de entrenamiento . . . . . 63
4.22 CIFAR-10. Early stopping. . . . . 65
4.23 CIFAR-10. Evolución del coste de entrenamiento y validación por épocas. SGD. Early stopping . . . . . 66
4.24 CIFAR-10. Evolución del coste de entrenamiento y validación por épocas. SGDM. Early stopping . . . . . 67
4.25 CIFAR-10. Evolución del coste de entrenamiento y validación por épocas. ADAM. Early stopping . . . . . 68
4.26 CIFAR-10. Evolución de la precisión por iteración. Conjunto de datos de validación . . . . . 68

# Notación

Esta sección provee una concisa referencia que define la notación usada en el trabajo. Se ha seguido la notación seguida en el libro "Deep Learning"[1].

### Números y arreglos

| | |
|---|---|
| $a$ | Un escalar (entero o real) |
| $\boldsymbol{a}$ | Un vector |
| $\boldsymbol{A}$ | Una matriz |
| $\mathbf{A}$ | Un tensor |
| $\mathrm{a}$ | Una variable aleatoria escalar |
| $\mathbf{a}$ | Un vector aleatorio |
| $\mathbf{A}$ | Una matriz aleatoria |

### Conjuntos y grafos

| | |
|---|---|
| $\mathbb{A}$ | Conjunto A |
| $\mathbb{R}$ | El conjunto de números reales |
| $\mathbb{N}$ | El conjunto de número naturales |
| $\{0, 1\}$ | Conjunto que contiene 0 y 1 |
| $[a, b]$ | Intervalo real incluyendo a $a$ y $b$ |

### Indexación

| | |
|---|---|
| $x^{\{i\}}$ | Elemento $i$ de conjunto de datos de entrenamiento |
| $a_i$ | Elemento $i$ de vector $\boldsymbol{a}$, con comienzo de indexación en 1 |
| $a_{-i}$ | Todos los elementos del vector $\boldsymbol{a}$ excepto el elemento $i$ |
| $A_{i,j}$ | Elemento $i, j$ de la matriz $\boldsymbol{A}$ |
| $\boldsymbol{A}_{i,:}$ | Fila $i$ de la matriz $\boldsymbol{A}$ |
| $\boldsymbol{A}_{:,i}$ | Columna $i$ de la matriz $\boldsymbol{A}$ |
| $\mathrm{a}_i$ | Elemento $i$ de un vector aleatorio $\mathbf{a}$ |

### Operaciones de Álgebra Lineal

| | |
|---|---|
| $\boldsymbol{A}^\top$ | Traspuesta de la matriz $\boldsymbol{A}$ |
| $\boldsymbol{A} \odot \boldsymbol{B}$ | Producto por componentes (Hadamard) de $\boldsymbol{A}$ y $\boldsymbol{B}$ |

### Cálculo

| | |
|---|---|
| $\frac{dy}{dx}$ | Derivada de $y$ con respecto a $x$ |
| $\frac{\partial y}{\partial x}$ | Derivada parcial de $y$ con respecto a $x$ |
| $\nabla_{\boldsymbol{x}} y$ | Gradiente de $y$ con respecto a $\boldsymbol{x}$ |
| $\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E}[f(x)]$ | Esperanza de $f(x)$ con respecto a $P(\mathrm{x})$ |

### Funciones

| | |
|---|---|
| $f : \mathbb{A} \to \mathbb{B}$ | Función $f$ con dominio $\mathbb{A}$ y rango $\mathbb{B}$ |
| $f \circ g$ | Composición de las funciones $f$ y $g$ |
| $\log x$ | Logaritmo natural de $x$ |
| $\sigma(x)$ | Función sigmoide, $\dfrac{1}{1+\exp(-x)}$ |
| $\tanh(x)$ | Función tangente hiperbólica, $\frac{(e^x - e^{-x})}{(e^x + e^{-x})}$ |
| $x^+$ | Parte positiva de $x$, i.e., $\max(0, x)$, Función ReLU |
| $\zeta(x)$ | Softplus, $\log(1 + \exp(x))$ |
| $h(x)$ | Función base |
| $g(x)$ | Función de activación |

### Conjuntos de datos y distribuciones

| | |
|---|---|
| $pnet(x_i)$ | Distribución de probabilidad generada por el dato $i$ (output de red neuronal) |
| $\hat{p}(x_i)$ | Distribución de probabilidad esperada por el dato $i$ |
| $\mathbb{X}$ | Conjunto de datos de entrenamiento |
| $\boldsymbol{x}^{(i)}$ | El ejemplo $i$-ésimo (input) de un conjunto de datos |
| $y^{(i)}$ or $\boldsymbol{y}^{(i)}$ | La etiqueta asociada al elemento $\boldsymbol{x}^{(i)}$ para aprendizaje supervisado |
| $\boldsymbol{X}_i$ | La matriz de orden $m \times n$ del ejemplo input $\boldsymbol{x}^{(i)}$ |

### Notación de redes neuronales

| | |
|---|---|
| $F(x)$ | Función compuesta de la red neuronal |
| $l$ | Capa número $l$, $l \in \mathbb{N}$ |
| $z^{[l]}$ | Vector de salida de la capa $l$ antes de aplicar la función de activación. |
| $a^{[l]}$ | Vector de salida de la capa $l$ tras aplicar la función de activación. Notesé que $a[L] = F(x)$. |
| $\delta^{[l]}$ | Vector que mide la sensibilidad de la función de coste a variaciones en $z^{[l]}$. |
| $\boldsymbol{W}$ | Matriz de pesos de una capa. |
| $\boldsymbol{W}^l$ | Matriz de pesos de la capa $l$ |
| $b$ | Sesgo de una función lineal / neurona |
| $\boldsymbol{B}^l$ | Vector de sesgos de la capa $l$ |
| $\eta$ | Tasa de aprendizaje |
| $p$ | Parámetro entrenable de la red neuronal (Peso o sesgo) |
| $v_t)$ | Momento en iteración $t$ |
| $\boldsymbol{g}$ | Vector gradientre |

# Parte I

# Introducción

# Capítulo 1

# Introducción y visión general

## Contenido


**1.1 Motivación . . . . . 2**

**1.2 Objetivo . . . . . 3**

**1.3 Estructura del documento . . . . . 4**


## Sinopsis

Éste es el capítulo de introducción, donde se explica el objetivo del proyecto, las razones que hacen de este tema un tema relevante, interesante y motivador sobre el que trabajar y la estructura del documento para comprender de manera integral el trabajo.

La reconocimiento de imágenes es una campo que crece año a año y aporta un gran beneficio a la sociedad, este TFG pretende iniciar al lector en esta área patiendo de conocimientos básicos. El documento cuenta con una parte teórica y una pare experimental aplicando la teoría.

## 1.1. Motivación

El reconocimiento de imágenes puede ser aplicado para una gran variedad de tareas. Reconocer una enfermedad en una radiografía o almacenar de forma ordenada las fotos que realizas desde tu teléfono en función de las personas que aparecen en ella son algunas de ellas.

Gracias a los avances en el rendimiento de sistemas de computación las posibilidades de procesar datos por parte de los ordenadores es mucho mayor. Aplicaciones de la inteligencia artificial a decisiones médicas, como la detección de enfermedades [1] a través de imágenes, son un paso al frente hacia una sociedad con una mayor calidad de vida y son mucho mas reales y plausibles gracias a estos avances en la tecnología. De igual modo, la inteligencia artificial sirve para ofrecer ayuda a aquellas personas con menos recursos facilitando las labores de ayuda humanitaria para que sea más eficaz y eficiente [2].

Dentro del mundo de la inteligencia artificial, el Deep Learning está ganando más y más relevancia gracias a las grandes oportunidades y aspectos que puede aportar. La traducción automática, aplicación del deep learning, brinda la posibilidad de eliminar muchas barreras de comunicación cuando las personas queremos movernos por la Tierra.

Aprender cómo se elaboran estos modelos y los diferentes algoritmos de optimización utilizados en el desarrollo de estos es esencial para emprender el camino en este ámbito. Una puerta a la mejora del desempeño y la minimización del tiempo empleado en el entrenamiento de redes neuronales.

En resumen, el reconocimiento de imágenes tiene una gran cantidad de aportaciones potenciales a la calidad de vida de las personas desde diversos ámbitos como el médico, el comercial o el día a día. Un primer paso es saber cómo se construyen y las opciones para mejorar el rendimiento de estás aplicaciones, objetivo de este TFG.

## 1.2. Objetivo

En esta sección, se describe el objetivo del proyecto, es decir, qué se pretende, a qué se aspira, y cuál es la meta. Es importante comprender esta sección, porque de otro modo, no se entiende el resto de la documentación.

Partiendo del artículo *Deep Learning: An Introducction for Applied Mathematicians* [2] se pretende comprender desde un nivel básico el diseño y funcionamiento de las redes neuronales, para qué se utilizan dentro del reconocimiento

[1] Artículo sobre detección de enfermedades a partir de imágenes de rayos X: http://informatica.blogs.uoc.edu/2018/01/25/diagnostico-automatico-de-enfermedades-a-partir-de-imagenes-de-rayos-x-de-torax/

[2] Artículo sobre la aplicación de IA en labores humanitarias: https://www.businessinsider.es/algoritmos-humanitarios-ia-ayuda-emergencias-humanitarias-461053

de imágenes y cómo se optimizan las funciones de coste utilizadas en modelos de aprendizaje supervisado basado en deep learning.

Para cualquier estudiante de Ingeniería informática que desee emprender una carrera en el ámbito de la inteligencia artificial y obtener unos conocimientos base sólidos y esenciales. Este TFG podría ser una buen punto de partida para iniciarse en el mundo de la visión artificial.

A modo de objetivo general, el presente Trabajo de Fin de Grado pretende explicar y servir de inicialización a las metodologías de diseño de redes neuronales para reconocimiento de imágenes y clasificación de objetos, así como realizar una valoración de algunos algoritmos de optimización utilizados en la clasificación multiclase. Esta valoración se realiza con una comparativa de los resultados obtenidos de forma empírica a través de un experimento.

## 1.3. Estructura del documento

El proyecto presenta un nivel básico de conocimientos en matemáticas para facilitar la comprensión de los conceptos explicados de manera completa. Se estructura en cuatro grandes bloques:

- **Conocimientos básicos de inteligencia artificial y aprendizaje automático** $\rightarrow$ Esta primera parte expone y explica los conocimientos previos necesarios para entender la creación de modelos de Deep Learning desde un nivel básico.
- **Introducción a las redes neuronales** $\rightarrow$ Cómo aprenden los modelos de aprendizaje automático basados en redes neuronales y diferentes técnicas de optimizar este aprendizaje para obtener un rendimiento mayor en la tarea. Se expone un ejemplo sencillo para facilitar la comprensión.
- **Diseño y entrenamiento de redes neuronales para el reconocimiento de imágenes** $\rightarrow$ Con tecnologías y herramientas que facilitan el desarrollo del proyecto y partiendo de dos conocidas bases de datos de imágenes como fuente[3], se diseñan y entrenan distintos modelos de redes neuronales variando distintos aspectos de la optimización de las mismas. Conjuntamente se exponen y analizan los resultados obtenidos. Concluye con una reflexión sobre los resultados obtenidos y posibles lineas futuras de investigación.

[3]MNIST y CIFAR-10

- **Scripts utilizados** $\rightarrow$ Código utilizado para la creación, entrenamiento y representación de datos significativos durante el desarrollo y realización del trabajo. Además, se incluye las salidas del programa gracias al formato Jupyer Notebook utilizado en el experimento.

# Parte II

# Aprendizaje automático basado en redes neuronales: Deep Learning

# Capítulo 2

# Introducción al deep learning y redes neuronales

## Contenido


**2.1 Inteligencia artificial y aprendizaje automático . . . . . 8**
2.1.1 Inteligencia Artificial . . . . . 8
2.1.2 Aprendizaje automático . . . . . 9
**2.2 Redes neuronales artificiales . . . . . 11**
2.2.1 Inspiración biológica . . . . . 12
2.2.2 Funcionamiento de una neurona . . . . . 14
2.2.3 Jerarquización del conocimiento . . . . . 16
**2.3 Introducción al deep learning . . . . . 17**
2.3.1 ¿Por qué utilizar deep learning? . . . . . 18


## Sinopsis


En este capítulo se explica de forma ordenada y sencilla los conocimientos básicos necesarios para una comprensión completa del proyecto desarrollado en los siguientes capítulos. Se definen términos y conceptos esenciales como la inteligencia artificial o aprendizaje automático.

Las redes neuronales son uno de los modelos computacionales más utilizados en el aprendizaje automático. Su origen es de inspiración biológica y permiten jerarquizar y extraer conocimiento de los datos para crear inteligencia.

Se hace una pequeña introducción del término deep learning o aprendizaje profundo y los beneficios de utilizarlo.

## 2.1. Inteligencia artificial y aprendizaje automático

### 2.1.1. Inteligencia Artificial

Dentro de las ciencias de la computación existen diferentes áreas que tratan distintos aspectos para la resolución de problemas. Una de ellas es la inteligencia artificial, posiblemente una de las áreas o disciplinas técnico-científicas con más potencial de los últimos años y que más inquietud ha creado por aquellos que temen que el beneficio producido por la inteligencia artificial se vuelva en nuestra contra, como Stephen Hawking[1] y Elon Musk[2].

#### Definición

La inteligencia artificial es un término difícil de definir y es a día de hoy sujeto de muchas discusiones. No se ha llegado a una definición común y se suele ver una definición diferente dependiendo del autor. A continuación se muestra algunas de las definiciones que distintos autores y organismos que gozan de gran reconocimiento en el campo de la tecnología han propuesto.

Según la revista oficial del Instituto Tecnológico de Massachusetts, una de las universidades más prestigiosas en el campo de la tecnología, *"la inteligencia artificial es la búsqueda de construir máquinas que puedan razonar, aprender y actuar de manera inteligente.* [3]

Por otro lado, según la conocida empresa Iberdrola[3] *La inteligencia artificial es un conjunto de algoritmos planteados con el objetivo de crear sistemas que presenten las mismas capacidades que el ser humano* [4]. Como podemos observar, Iberdrola da una definición más concreta a la palabra restringiéndola a algoritmos.

Como definición más extensa, Alan Turing, considerado el padre de la computación, fue el primero en intentar definir la inteligencia artificial. En 1950 publicó un artículo académico llamado *Computing machinery and intelligence* [5] en el

[1] Reconocido físico teórico, astrofísico, cosmólogo y divulgador científico británico.

[2] Físico, inversionista y magnate conocido por ser cofundador de Paypal y Tesla Motors. También es cofundador de OpenAI, una compañía de investigación de inteligencia artificial sin ánimo de lucro.

[3] Empresa española dedicada a la producción, distribución y comercialización de energía.

que argumentaba que *una máquina es inteligente si puede actuar como un humano*, y definió una serie de rasgos o capacidades que un sistema ha de tener para ser inteligente, que tienen gran relevancia hoy en día (Test de turing).

Como idea común a todas las definiciones, podemos extraer que la inteligencia artificial, es el estudio y la técnica de todo aquello que dote de inteligencia a una máquina. Entendiendo inteligencia la capacidad de aprender, razonar, entender, tomar decisiones y formarse una idea determinada de la realidad.

### 2.1.2. Aprendizaje automático

Lo primero que debemos preguntarnos es, ¿Qué es aprender? y ¿Qué es automático?. Aprender, según la Real Academia de la Lengua Española, es *"Adquirir el conocimiento de algo por medio del estudio o de la experiencia"*. Por otro lado, una de las definiciones que la RAE expone de automático es *"Dicho de un mecanismo o de un aparato: Que funciona en todo o en parte por sí solo."*.

Según Tom Mitchell (1997): "Un programa informático aprende de la experiencia $E$ con respecto a una clase de tarea $T$ y un desempeño $D$, si el desempeño de en la tarea $T$ medido por $D$ mejora con la experiencia $E$". [6]

#### Definición

De estas definiciones podemos sacar una idea esencial. El aprendizaje automático se refiere a la creación de conocimiento en una máquina por sí sola a partir de la experiencia. En este caso la experiencia serán datos que facilita el usuario para que el sistema de aprendizaje automático cree conocimiento por sí mismo.

Al proceso de proveer al modelo[4] con datos para crear conocimiento se le llama entrenamiento. Este entrenamiento se lleva a cabo a través de un algoritmo de aprendizaje que hace que el modelo se vaya ajustando. Como resultado de este entrenamiento se creará un modelo inteligente capaz de realizar diversas funciones dependiendo de para qué se haya programado.

Este modelo puede ser utilizado para hacer predicciones o clasificar una serie de objetos dependiendo de las características del mismo.

[4]Se entiende modelo como la concreción de un sistema de aprendizaje automático

### Tipos de reglas de aprendizaje

Atendiendo a la experiencia que se pueda aportar durante el entrenamiento puede haber tres tipos básicos de reglas de aprendizaje. Los algoritmos de aprendizaje se pueden clasificar en tres tipos [7].

- **Aprendizaje supervisado**: Se sabe a priori las posibles respuestas o etiquetas (output) por parte del sistema inteligente. El conjunto de datos de entrada contiene las características de los datos. Cada uno tiene asociado una etiqueta u objetivo.

  Un supervisor o agente externo facilita los datos y su salida deseada para ajustar el mismo y que produzca la etiqueta correcta para nuevos datos no etiquetados. De esta manera, el supervisor puede comparar la respuesta real con la respuesta esperada y, si existen diferencias, ajustar iterativamente el sistema para que produzca el output deseado (Entrenamiento).

  El diagrama de flujo de la figura 2.1 muestra el funcionamiento de un modelo con este tipo de aprendizaje.

- **Aprendizaje no supervisado**: En este tipo de modelos no se conocen o no se esperan respuestas deseadas, es decir, el sistema no sabe si las respuestas que genera son correctas o no. No se necesita de un supervisor o agente externo para su entrenamiento.

  Este tipo de aprendizaje consiste en que el sistema descubra características, regularidades, categorías, propiedades o correlaciones en los datos de entrada por sí mismo y que los produzca de forma codificada en la salida. Puede por ejemplo mostrar el grado de similitud entre los datos o realizar un agrupamiento[5] y determinar a que cluster o grupo pertenecen nuevos datos.

  El diagrama de flujo de la figura 2.2 muestra el funcionamiento de un modelo con este tipo de aprendizaje.

- **Aprendizaje por refuerzo**: Este tipo de aprendizaje no encaja en ninguno de los dos expuestos. Se distinguen dos componentes principales de estos sistemas: El agente (modelo inteligente) y el entorno. En este modelo, el aprendizaje se realiza valorando el grado de idoneidad de la respuesta dada medido por una “recompensa“ que varía en función del estado en el que se encuentre el estado.

[5] También denominado *clustering*

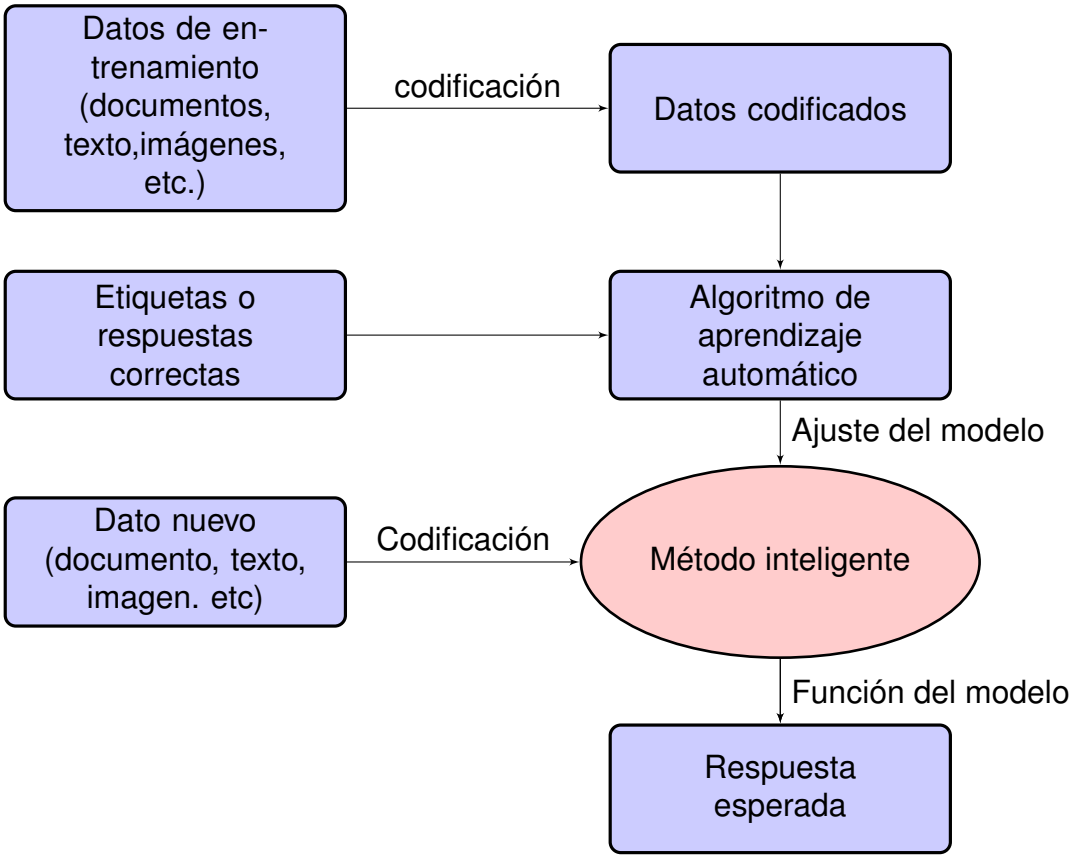



**Figura 2.1:** Diagrama de flujo de modelo con aprendizaje supervisado

El objetivo de estos modelos es maximizar la recompensa, es decir, el modelo tiene que ser capaz de dar la acción con mayor recompensa para el estado actual del entorno.

Dentro de estos tipos de aprendizaje, el proyecto está centrado en el aprendizaje supervisado. La tarea de clasificación de imágenes hace uso del aprendizaje supervisado.

## 2.2. Redes neuronales artificiales

Las redes neuronales artificiales son una de las principales herramientas utilizadas en el aprendizaje automático. Su nombre y su creación fueron inspirados por el funcionamiento de redes neuronales biológicas de los humanos [8].

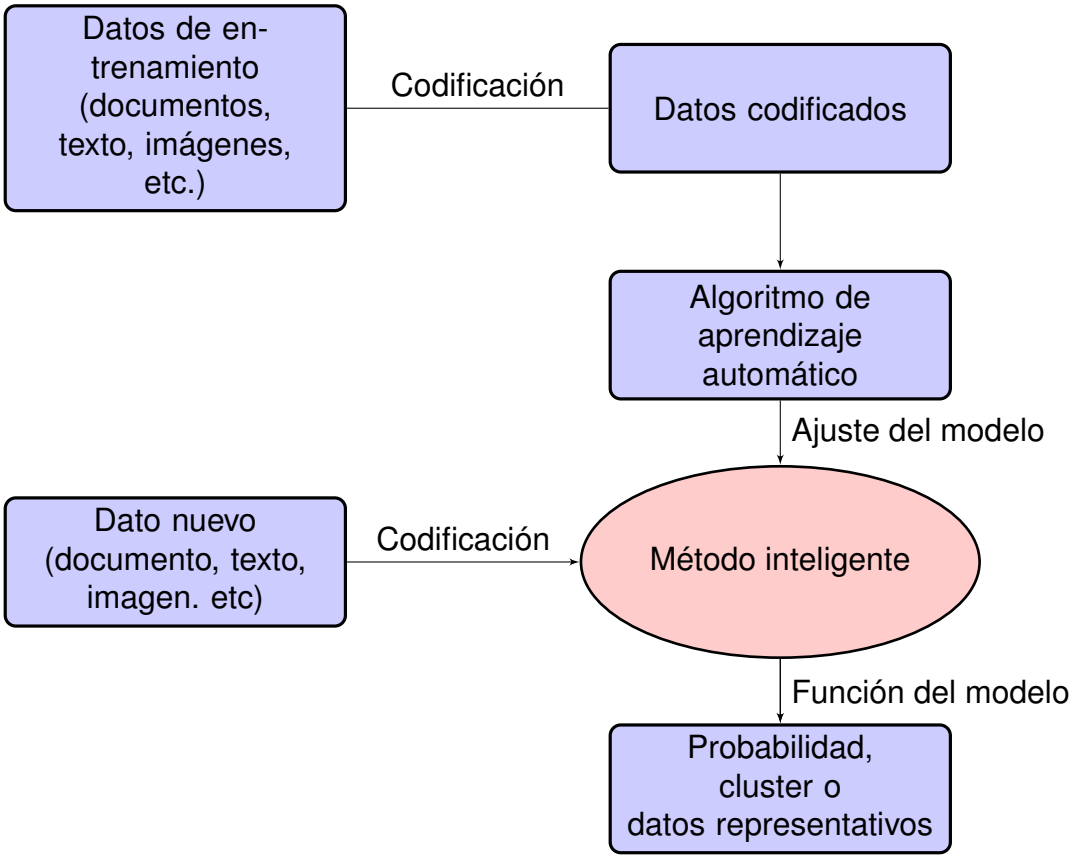


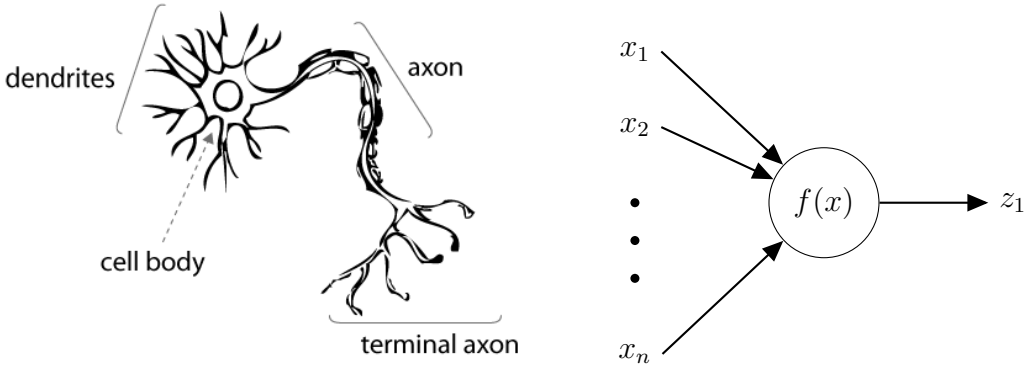

**Figura 2.2:** Diagrama de flujo de modelo con aprendizaje no supervisado

### 2.2.1. Inspiración biológica

En la figura 2.3a se puede observar una neurona biológica compuesta por dendritas, el cuerpo de la célula, un axón y la terminación del axón. En la figura 2.3b se representa una neurona artificial. Tal y como se muestra en la figura las neuronas son representadas por una función $f(x)$ que toma valores de entrada $x_i$ y genera un output $z_i$.

Las neuronas biológicas se comunican mediantes señales eléctricas y químicas. Las dendritas son las encargadas de recibir las señales de otras neuronas, y la terminación del axón se encarga de transmitir la señal de la neurona a otras neuronas. A este tipo de transmisión se le denomina sinapsis y es producido por el potencial de acción.

El potencial de acción es la descarga eléctrica que transmite un impulso desde una neurona a la siguiente que sigue la ley "todo o nada"[9], es decir, las descarga solo se propagara si el estímulo que llega a la neurona (suma de potencial

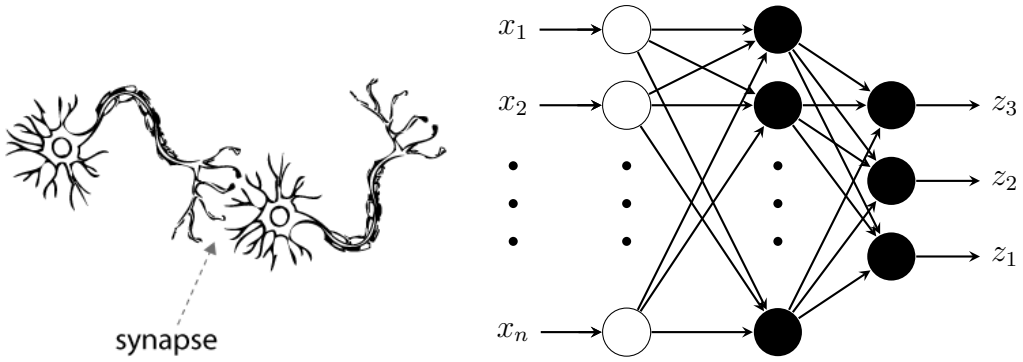



**(a)** Neurona biológica **(b)** Neurona artificial

**Figura 2.3:** Imagen de neurona biológica y artificial

eléctrico proveniente de las conexiones con otras neuronas) por las dendritas sobrepasa un límite.

Haciendo uso de una la analogía en nuestra neurona artificial. La función $f(x)$ podemos definirla como una función compuesta por la función base $g(x)$ y la función de activación $h(x)$. De esta manera $f(x) = (h \circ g)(x)$. Cada dato de entrada funciona como las ramificaciones de las dendritas, ponderando el valor de la conexión existente. La función base entonces será la suma de estos valores. La función de activación es una función que funciona como límite a partir del cual la neurona de "activarse" y propagar su valor a las neuronas adyacentes [10].




**(a)** Neuronas biológicas interconectadas **(b)** Red neuronal artificial

**Figura 2.4:** Imagen de neuronas biológicas y red neuronal artificial

La figura 2.4a muestra la conexión entre dos neuronas y en la figura 2.4b se representa un tipo de red neuronal denominado Feedforward Neural Network, en el que las neuronas de cada capa (hilera de neuronas alineadas verticalmente) son "alimentadas" con el output de las neuronas de la capa anterior.

A la primera capa de neuronas se le denomina capa de entrada, a las capas intermedias se les llama capas escondidas, y, a la ultima capa se le denomina capa de salida. El número de capas escondidas y de neuronas en cada capa es variable. En la sección 2.3.1 se explica la importancia del número de capas y su función, base del deep learning.

Teniendo en cuenta estas ideas, una red neuronal funciona como una gran función matemática $F(x)$, que toma unos valores de entrada y devuelve unos valores de salida.

Se puede observar que los datos de entrada de las neuronas situadas en la misma capa son los mismos. Lo que diferenciará la salida de cada neurona serán los pesos que que asignen a cada dato de entrada. Pero, ¿Cómo se genera la salida, qué importancia tienen la función base y las neuronas?. ¿Cómo una red neuronal puede utilizarse para, por ejemplo, clasificar una serie de datos complejos?

### 2.2.2. Funcionamiento de una neurona

#### Funcionamiento de una neurona

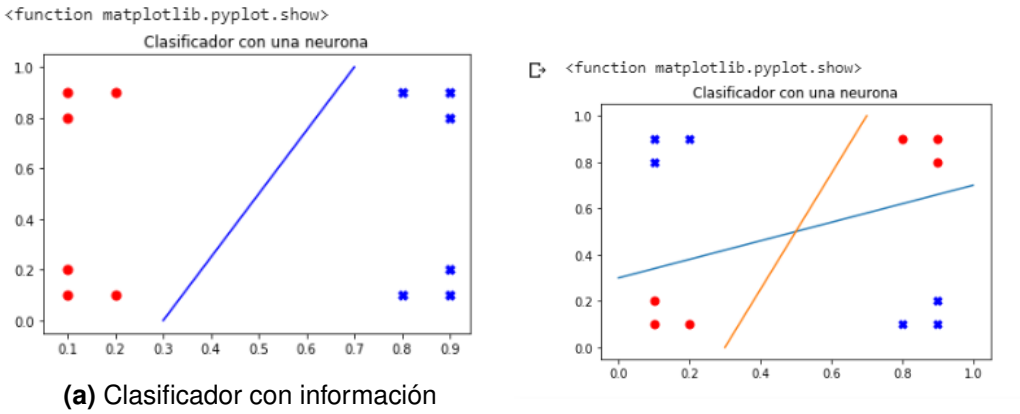
En el caso más sencillo una neurona funcionaría como un modelo de regresión lineal [6] de la siguiente manera $h(x) = (w_i x_i + b)$, pudiendo definir una función de clasificación sencilla que se podría ver como una función de activación de la siguiente manera.

$$g(x) = \begin{cases} 1 \quad si \quad f(x) > 0 \\ \\ 0 \quad si \quad f(x) \leq 0 \end{cases} \tag{2.1}$$

Con este modelo se puede realizar una simple tarea de clasificación binaria en el que la función matemática de la neurona representa el separador entre los dos grupos (Ver figura 2.5a). En este caso la función de activación será la que realice la tarea de clasificación. Si el output de la red es 1, pertenece a una clase y, si el output es 0, pertenece a la otra.

Sin embargo, este modelo unineuronal tiene muchas limitaciones. Con dos valores de entrada y representándolos en un plano euclidiano rápidamente se puede observar que si las nubes de puntos de los grupos se distribuyen como se muestra en la figura 2.5b, es imposible hacer un clasificador eficaz al no poder separar linealmente los dos grupos. El límite de decisión no es eficaz (Rectas azul o naranja) Solucionar este problema es sencillo añadiendo otra neurona en

[6] Este modelo se utiliza como herramienta de predicción




**(a)** Clasificador con información sencilla

**(b)** Clasificador con información compleja

**Figura 2.5:** Clasificador unineuronal

la misma capa.De esta manera se está formando una red neuronal para crear un modelo más complejo que permita crear un clasificador eficaz. Con este modelo sí se puede crear dos separadores para clasificar estas nubes de puntos como se puede comprobar en la figura 2.6, es decir, permite modelar información más compleja.

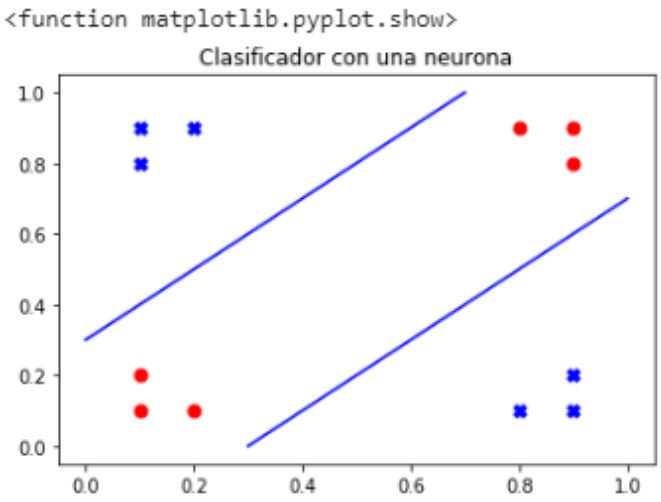



**Figura 2.6:** Clasificador con dos neuronas

### 2.2.3. Jerarquización del conocimiento

Las neuronas pueden estar ordenadas de forma secuencial u conectadas horizontalmente ordenándose en capas. La principal ventaja que nos aporta esta distribución en capas, y en la que se basa el funcionamiento de las redes neuronales profundas, es que permite crear conocimiento jerarquizado [11].

De esta manera la red neuronal puede aprender conocimientos sencillos en las primeras capas y conocimiento más complejo y abstracto según se van añadiendo capas. De esta profundidad de capas surge el término deep learning. Sin embargo, existe un problema cuando estructuramos la red neuronal en capas.

#### Función de activación

Se puede demostrar matemáticamente que sin una función de activación que aporte no linealidad al modelo, la organización de unidades de proceso lineales en capas se podría simplificar en una única función. Esto hace imposible la jerarquización efectiva del conocimiento. Por ese motivo se utilizan funciones de activación que aportan la no linealidad necesaria.

La función de activación anterior (Función 2.1) se denomina función escalonada (Ver figura 2.7a). Esta tiene una gran desventaja, en una red neuronal con varias capas, el conocimiento que se pasa a la siguiente capa se simplifica de manera total. Un pequeño cambio en la salida de la función base de la neurona puede tener un efecto drástico en la salida de la neurona.

Este cambio de valor instantáneo no favorece el aprendizaje porque las capas se pueden simplificar en una sola función, por lo tanto, esta función no interesa.

Por este motivo surgieron funciones de activación que modelaban los cambios de una forma más suave. Una de las primeras funciones que se empezó a utilizar es la función sigmoide (Función 2.2)(ver figura 2.7b):

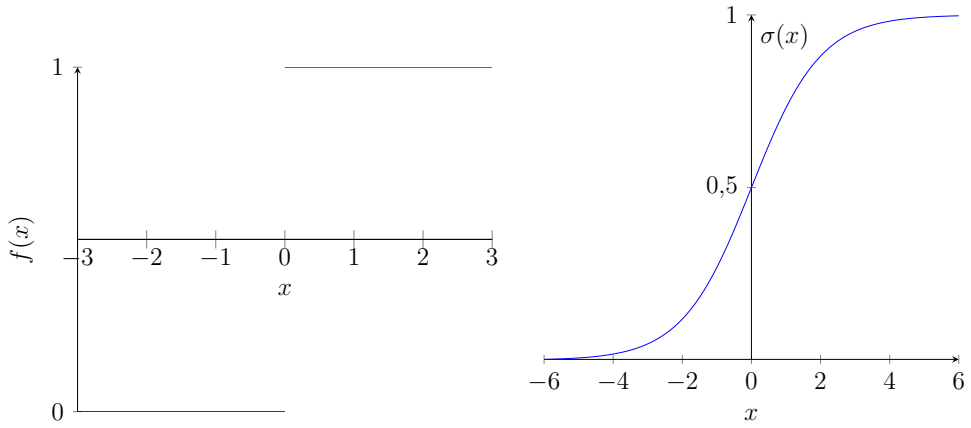
$$\sigma(x) = \sigma(\frac{1}{1+e^{-x}}) \tag{2.2}$$

Otra función de activación que se utiliza es la función tangente hiperbólica (Función 2.3)(ver figura 2.8a).

$$\tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})} \tag{2.3}$$

La función de activación más utilizada actualmente es la función rampa o función de unidad lineal rectificada definida como (Función 2.4)(Ver figura 2.8b).

$$ReLU = max(0, x) \tag{2.4}$$




**(a)** Función escalonada **(b)** Función sigmoide

**Figura 2.7:** Funciones de activación escalonada y sigmoide

Gracias a estas funciones, pequeñas variaciones en el valor de salida de una neurona desencadenan una pequeña variación en el resto de neuronas de las capas siguientes, permitiendo encadenar varias neuronas en capas de forma eficaz. La unión de una gran cantidad de capas de neuronas posibilitan la solución de problemas muy complejos.

Para ver una visualización geométrica de lo explicado anteriormente se recomienda "jugar" un poco con la siguiente herramienta de clasificación:

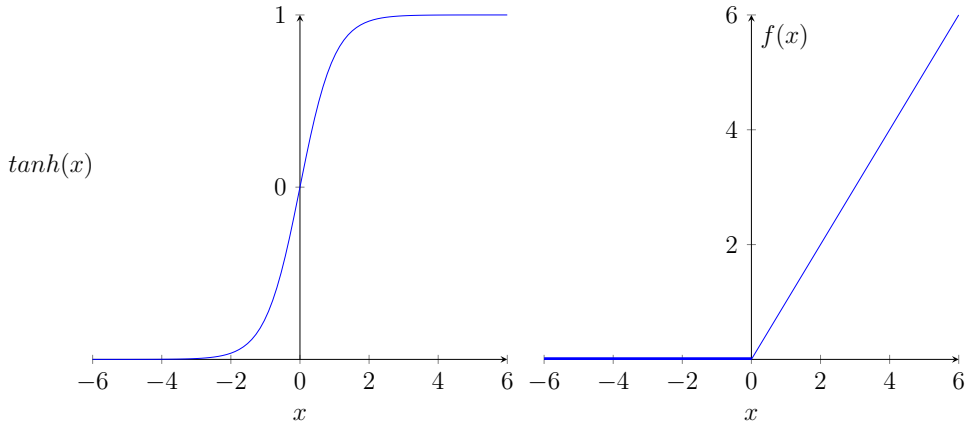
**A neural network playground:** `https://playground.tensorflow.org`

## 2.3. Introducción al deep learning

El Deep Learning es un subcampo del aprendizaje automático. Según el autor del libro *"Hands-on Deep Learning algorithms with Python"* [12], el Deep Learning es el aprendizaje automático que utiliza redes neuronales artificiales con muchas capas como modelo computacional para aprender.

El deep learning está basado en algoritmos de aprendizaje que aprenden múltiples niveles de representación con el objetivo de modelar relaciones complejas entre los datos. Conceptos y características complejas están definidas a partir de características más sencillas aprendidas en las primeras capas. A esta arquitectura jerárquica es a lo que se llama arquitectura profunda. [13]

Hasta ahora las restricciones de rendimiento computacional imposibilitaban el




**(a)** Función tangente hiperbólica **(b)** Función ReLU

**Figura 2.8:** Funciones de activación tangente hiperbólica y ReLU

uso de este tipo de métodos para aprender, pero esto ha cambiado en los últimos años. Una red neuronal con muchas capas la denominamos red profunda.

### 2.3.1. ¿Por qué utilizar deep learning?

El éxito del aprendizaje automático reside, en mayor o menor medidad, en la elección correcta de las características más representativas de los conjuntos de datos (Datos de entrada del modelo). La ingeniería de características juega un papel crucial en el aprendizaje automático. Si elegimos los rasgos de los datos con los que nutrir el modelo correctamente haremos que el mismo tenga un buen rendimiento, pero esta tarea no es tan sencilla.

Con el deep learning los rasgos a aportar al modelo pueden ser más sencillos. Gracias al empleo de varias capas, el modelo aprende las características complejas intrínsecas a los datos de manera automática. Para una comprensión más clara de este concepto se va a profundizar con un ejemplo relacionado con el proyecto desarrollado en este TFG.

Supongamos que se quiere realizar una tarea de clasificación de imágenes. El modelo va a aprender a reconocer si una imagen contiene un perro o no.

Con aprendizaje automático, tenemos que seleccionar que características de la imagen ayudan al modelo a entender si hay un perro o no ,realizar una etiquetación previa, y nutrir al modelo con estos datos para que aprenda a través del algoritmo de aprendizaje (Haga un mapeo entre las características y la etiqueta de perro).

Sin embargo, extraer esas características de una imagen es algo tedioso y trabajoso. Con deep learning, las capas de la red neuronal harán el trabajo de extractor de características por nosotros. La primera capa puede extraer las características de la imagen que caracterizan el cuerpo del perro, la siguiente capa la forma o el color, de forma que se vaya jerarquizando el conocimiento.

A pesar de este aspecto tan beneficioso, no se recomienda utilizar el deep learning para conjunto de datos que sean pequeños o que sean muy simples, ya que se incurre en el riesgo de de causar overfitting del conjunto de datos. Debido a esto se recomiendo utilizar este método cuando se tenga una cantidad de datos significativa.

En resumen, la redes neuronales de deep learning aprenden o seleccionan de manera automática las características más representativas para la tarea a realizar.

### Ejemplo

Para un visión clara de todo lo explicado en este capitulo se expone el siguiente ejemplo de clasificación[7].

En la figura 2.9, la red neuronal profunda identifica automáticamente, con unos datos de entrada más simple, las características de los mismos. Sin embargo, se obtiene el mismo resultado si se hubiera realizado una tarea de ingeniería de datos y extraído características relevantes para la clasificación (Figura 2.10).

No se pretende restar importancia al tratamiento de datos antes de alimentar el modelo con ellos, si no dar una idea de la función que desempeña la estructuración en capas. El poder de la redes neuronales profundas es la habilidad de extraer las características apropiadas y una discriminación de las mismas para un mejor resultado.

[7] Ejemplo realizado con la herramienta de Playground de redes neuronales

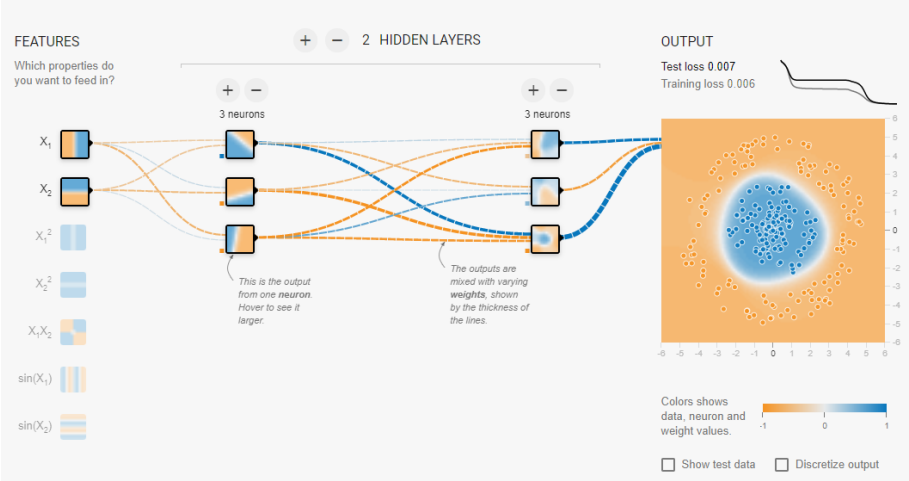



**Figura 2.9:** Red neuronal profunda

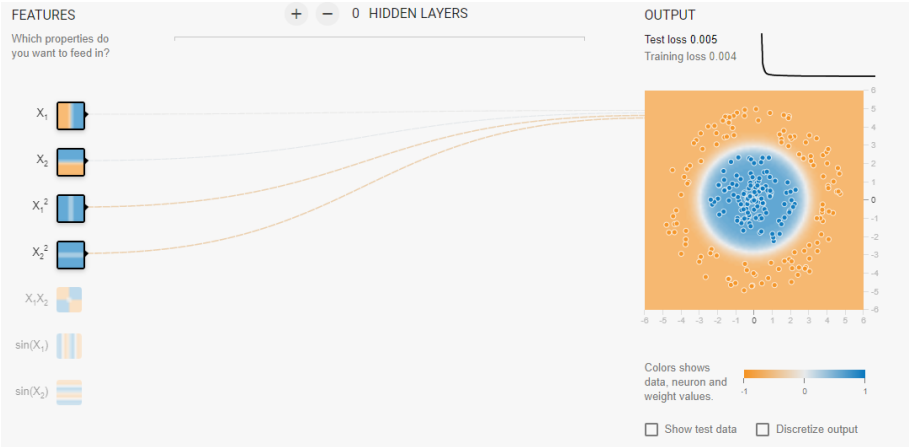



**Figura 2.10:** Red neuronal sin jerarquización de conocimiento

# Capítulo 3

# Creación y entrenamiento de redes neuronales

## Contenido


**3.1 Pasos en la creación de un modelo basado en redes neuronales de deep learning . . . . . 22**
3.1.1 Estructura de la red neuronal . . . . . 23
3.1.2 Función base y de activación . . . . . 24
3.1.3 Función de coste . . . . . 25
3.1.4 Gradiente descendente . . . . . 26
3.1.5 Backpropagation . . . . . 27
3.1.6 Hiperparámetros . . . . . 28
**3.2 Ejemplo . . . . . 29**
3.2.1 Herramientas . . . . . 29
3.2.2 Resultados . . . . . 29
**3.3 Validación y evaluación de redes neuronales . . . . . 32**


## Sinopsis


En este capítulo se explica los conceptos clave del aprendizaje supervisado basado en redes neuronales. Se detallan los componentes y el diseño de un modelo inteligente de deep learning.

La estructura básica de una red neuronal, las operaciones matemáticas y las funciones necesarias para el aprendizaje son algunos de los componentes esenciales. Se explica la forma en la que aprenden las redes neuronales a través de la optimización de la función de coste y el método básico utilizado, el gradiente descendente.

Junto con las explicaciones se desarrolla un problema y una solución utilizando una red neuronal profunda. Concluye con el entrenamiento de la solución propuesta y la evaluación de los resultados para verificar la teoría expuesta.

## 3.1. Pasos en la creación de un modelo basado en redes neuronales de deep learning

El ejemplo seguido trata la creación de un clasificador binario. Dado un conjunto de datos etiquetados $\mathbb{X}$ (datos de entrenamiento) representados en la Figura 3.1, $x^{(i)} \in \mathbb{R}^2$ . Las etiquetas pueden ser $\mathbb{A}$ o $\mathbb{B}$.

Por poner un ejemplo realista, se puede imaginar que los datos representan las coordenadas de un huerto. Los puntos de $\mathbb{A}$ son puntos en los que se ha medido la calidad de la tierra y ha salido calidad alta, mientras que los puntos de $\mathbb{B}$ son puntos en los que ha salido calidad baja. El modelo deberá ser capaz de predecir si la calidad del terreno es alta o baja dado un punto para poder plantar con mayor confianza.

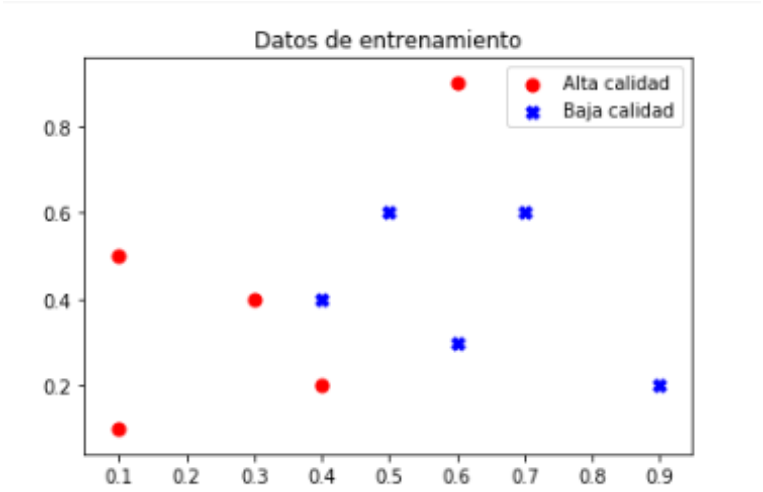



**Figura 3.1:** Conjunto de datos de entrenamiento/ puntos del huerto

### 3.1.1. Estructura de la red neuronal

La estructura puede variar tanto en el número de capas y neuronas, como la relación que existe entre las neuronas. Estos factores dan lugar a redes neuronales feed forward, redes neuronales convolucionales o redes neuronales recurrentes entre otros más.

El modelo ejemplo es una red neuronal feed forward de cuatro capas (Una capa input - dos capas escondidas - una capa de salida). Se le denomina red neuronal feed forward porque la información fluye de capa en capa hacia delante. Toda la información que sale de la primera capa es la entrada de la segunda, esto quiere decir que el output de cada neurona en una capa $l$ es tomado como entrada por cada neurona de la capa $l + 1$.

La capa de entrada consistirá en dos neuronas, una por cada componente de las coordenadas del plano. La segunda capa y la tercera capa tendrán dos y tres neuronas respectivamente. Por último, la capa de salida tendrá dos neuronas. La estructura se puede ver en la figura 3.2.

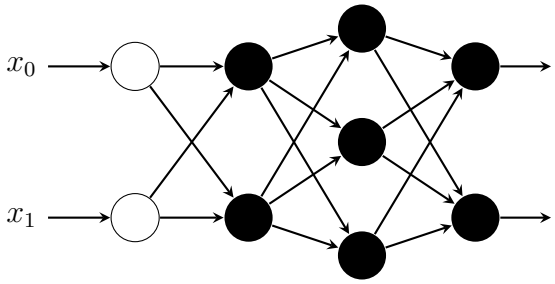



**Figura 3.2:** Red neuronal artificial ejemplo

El número de capas escondidas, el número de neuronas en estas capas y sus relaciones no es una ciencia cierta y dependerá de la complejidad del problema a resolver. No hay criterios teóricos para fijarlos, se utiliza el método de prueba y error para encontrar el modelo óptimo intuyendo cuál podría ser de forma general. Sin embargo, hay estructuras que se ha demostrado que generan un mejor resultado, como las redes neuronales convolucionales con la clasificación de imágenes.

En cuanto al número de neuronas de la capa de salida, en los problemas de clasificación se suele utilizar el mismo número de neuronas que de clases posibles en la clasificación. Por este motivo se ha elegido dos neuronas en la capa de salida para el ejemplo (Clases $\mathbb{A}$ y $\mathbb{B}$). Una buena clasificación resultaría si en los puntos en los que hay buena calidad, el resultado se aproximara a $\begin{bmatrix}1\\0\end{bmatrix}$.

Y, en los puntos de mala calidad de la tierra a $\begin{bmatrix}0\\1\end{bmatrix}$.

De esta manera se podría definir la función de clasificación para que determinara la clase dependiendo de qué componente es mayor y la función de salida deseable de la siguiente manera:

$$y^{(i)} = \begin{cases} \begin{bmatrix}1\\0\end{bmatrix} & si \quad x^{\{i\}} \in \mathbb{A} \\ \\ \begin{bmatrix}0\\1\end{bmatrix} & si \quad x^{\{i\}} \in \mathbb{B} \end{cases}$$

### 3.1.2. Función base y de activación

El output del modelo es la salida de una función compuesta $F(x) = a^{[L]}$, $a^{[L]} \in \mathbb{R}^2$, $a_i^{[L]} \in [0,1]$ que toma los valores de entrada y realiza una serie de operaciones sobre los inputs para obtener los valores de salida. Como ya se ha explicado, cada neurona puede ser representada como una composición de una función lineal $\boldsymbol{W}x + b$ y una función de activación.

Los parámetros $\boldsymbol{W}$ son los pesos que otorga cada neurona a las conexiones con neuronas anteriores y $b$ el sesgo que otorgará mayor o menor facilidad de activación a la neurona. Un sesgo alto activará la neurona para una mayor rango de valores de entrada.

Para una representación más sencilla se pueden agrupar los parámetros de cada capa de tal forma que los parámetros $\boldsymbol{W}$ y $\boldsymbol{B}$ sean representados por una matriz y un vector respectivamente.La matriz de pesos deberá tener de dimensiones, nº de neuronas en la capa $l$ x nº de neuronas en la capa $l-1$. El vector de sesgos tendrá el mismo número de componentes que de neuronas en la capa $l$.

La función de activación utilizada para el ejemplo es la función sigmoide (Función 2.2, figura 2.7b).

Los parámetros son los que realmente realizan la parte de aprendizaje. Para ello, en algoritmos de aprendizaje supervisado, se define la función de coste. Esta función pretende determinar el error entre el valor estimado y el valor real o de salidad de la red neuronal.

Esta función es una medida de las propiedades deseables del modelo, cuanto menor sea el valor del coste, mas cerca se estará de dar con el modelo ideal. Más adelante se detallaran los límites de esta función para no incurrir en un

sobreajuste del modelo a los datos de entrenamiento, ya que queremos que el sistema tenga un buen desempeño en datos que no hayan sido utilizados para el entrenamiento.

### 3.1.3. Función de coste

La función de coste depende de los parámetros del modelo elegido. Se define atendiendo a la naturaleza del problema que se desea resolver con el modelo y mide el desempeño de nuestro modelo. La elección de como se mide el desempeño es a menudo difícil debido a que debemos fijar una medida que corresponda correctamente al comportamiento que queremos que tenga el sistema.

La mejor o peor obtención de un modelo efectivo y confiable para el ejemplo se convierte en un problema de optimización para minimizar la función de coste.

En el ejemplo propuesto, una función de coste adecuada es el error cuadrático medio (Función 3.1. ECM).

$$Cost(p) = ECM = \frac{1}{n}\sum_{i=1}^{n} ||y^{(i)} - F(x^{\{i\}})||_2^2 \tag{3.1}$$

Se puede observar que es la media de errores cuadráticos de los puntos usados para la medición del coste. De esta manera el coste de cada punto viene definido por la función 3.2.

$$Cx^{\{i\}} = ||y^{(i)} - F(x^{\{i\}})||_2^2 \tag{3.2}$$

Cuanto menor sea el error cuadrático medio de los datos de validación $\mathbf{X}$, mayor fiabilidad tiene el modelo para la predicción de nuevos puntos. Otra función de medida del desempeño en problemas de clasificación es la precisión del modelo. Dado un conjunto de datos no visto por el modelo, la precisión mide el porcentaje de aciertos.

El caso propuesto cuenta con 23 parámetros. Se necesita un algoritmo para ajustar estos parámetros en la etapa de entrenamiento para un desempeño mejor. Una primera idea podría ser solucionarlo con la técnica de análisis numérico de los mínimos cuadrados. Sin embargo, este método no podría ser escalable para los modelos de deep learning que cuentan con miles de parámetros como se verá en el siguiente capítulo.

El objetivo es encontrar el punto de la función de coste (se debe tener en cuenta que depende de todos los parámetros) en el que la derivada se haga cero[1]. Para llevar a cabo esa tarea se utilizan diferentes algoritmos que paso a

[1] Se aplica el criterio de la primera derivada

paso van reduciendo la función de coste hasta encontrar un mínimo de la misma. Estos son los denominados algoritmos de optimización.

Puede ser un mínimo local o el mínimo absoluto. El estancamiento en un mínimo local dependerá de diversos factores como la inicialización de los parámetros y los hiperparámetros definidos.

Un algoritmo de optimización que ayuda a encontrar los mínimos en el ejemplo es el gradiente descendente. Procede iterativamente, calculando una serie de vectores que representan los cambios a efectuar en los parámetros para encontrar un mínimo. El objetivo es converger en un vector de parámetros que minimice la función de coste.

### 3.1.4. Gradiente descendente

El método del gradiente descendente, ampliamente usado en problemas de optimización y en aprendizaje automático, indica que se deben realizar pequeñas perturbaciones en los parámetros para encontrar los mínimos de la función a optimizar.

Se ha demostrado matemáticamente [2] que esta perturbación ha de ser igual al valor negativo de la derivada de la función con respecto al parámetro. El vector de perturbaciones queda definido en la ecuación 3.3.

$$\Delta p = -\nabla Cost(p) \tag{3.3}$$

Teniendo en cuenta que las derivadas solo tienen relevancia para cambios pequeños de la función. Se elige el hiperparámetro llamado tasa de aprendizaje o learning rate en inglés. La fórmula de actualización (Función 3.4) queda de la siguiente forma.

$$p_t = p_{t-1} - \eta \nabla Cost(p_{t-1}) \tag{3.4}$$

En otras palabras, se sigue la dirección de la pendiente de la superficie creada por la función de coste hacia abajo hasta llegar a un valle. El learning rate posibilita que no se estanque en los bordes de un valle local (mínimo local).

El ejemplo elegido tiene 10 datos de entrenamiento, utilizando el método del gradiente descendente, se calcula el vector gradiente $\boldsymbol{g}$. Este vector representa la derivada de los parámetros con respecto a la función de coste.

Los componentes son las derivadas parciales de cada parámetro $w_{ij}^l$ y $b_j^l$ con respecto al coste. Dado que el coste es la media de los errores cuadráticos de

[2] Utilizando la teoría de la desigualdad de Cauchy-Schwarz

cada punto, el vector gradiente es la media de las derivadas parciales en cada dato de entrenamiento.

Para el ejemplo dado es viable calcularlo por este método, pero es fácil darse que no es muy escalable para conjuntos de datos extensos.

#### Variantes del gradiente descendente

Existen diferentes variaciones de este método. Estos son:

- **Gradiente descendente por lotes (a.k.a Vanilla gradient descent)**. $\rightarrow$ Computa el gradiente de la función de coste para todo el conjunto de datos de entrenamiento. Para realizar una actualización se necesita calcular los gradientes para todo el conjunto de datos por lo que puede ser lento y demasiado costoso para conjuntos de datos grandes.
- **Gradiente descendente estocástico**. $\rightarrow$ Este método elige aleatoriamente un punto de entrenamiento para calcular el vector gradiente. Este punto representa a todo el conjunto de datos de entrenamiento. Esta variante es más rápida que la anterior pero la función de coste fluctúa sustancialmente a lo largo de las iteraciones.
- **Gradiente descendente por mini-lotes**. $\rightarrow$ Este método agrupa los datos de entrenamiento en lotes de tamaño $n$ y los elige aleatoriamente en cada iteración para calcular el vector gradiente. Esta variante permite una convergencia más estable que el SGD y más rápida que VGD.

Para el ejemplo se utiliza el gradiente descendente estocástico y el gradiente descendente por lotes. Aún así, los cálculos de las derivadas pueden ser complejos y costoso. Por ello se utiliza el método de backpropagation para el cálculo del vector gradiente.

### 3.1.5. Backpropagation

Es un algoritmo ampliamente utilizado en el entrenamiento de redes neuronales para el aprendizaje supervisado. Este método se basa en el principio fundamental de que en el cálculo de derivadas parciales influyen los parámetros de capas superiores pero los de capas inferiores no. De esta manera, se calculan primero las derivadas parciales de los parámetros de la última capa, después de la penúltima, y así, sucesivamente.

Para entender las funciones que utiliza el método se define la siguiente notación.

- $L \rightarrow$ Capa de salida
- $z^{[l]} \rightarrow$ Vector de salida de la capa $l$ antes de aplicar la función de activación.
- $a^{[l]} \rightarrow$ Vector de salida de la capa $l$ tras aplicar la función de activación. Notesé que $a[L] = F(x)$.
- $\delta^{[l]} =\rightarrow$ Vector que mide la sensibilidad de la función de coste a variaciones en $z^{[l]}$. $\delta^{[l]} = \frac{\partial C}{\partial z^{[l]}}$

El método de backpropagation define cuatro funciones (Funciones 3.5) para el cálculo del vector gradiente (Lema).

$$\begin{aligned}
\delta^{[L]} &= \sigma'(z^{[L]}) \odot (a^L - y) \\
\delta^{[l]} &= \sigma'(z^{[L]}) \odot (\boldsymbol{W}^{[l+1]})^T \delta^{[l+1]}) && \text{si } 2 \leq l \leq L-1 \\
\frac{\partial C}{\partial b_j^{[l]}} &= \delta_j^{[l]} && \text{si } 2 \leq l \leq L \\
\frac{\partial C}{\partial w_{j,k}^{[l]}} &= \delta_j^{[l]} a_k^{[l-1]} && \text{si } 2 \leq l \leq L
\end{aligned} \tag{3.5}$$

En la práctica, las librerías y frameworks utilizados en deep learning implementan diferentes métodos para el cálculo de los gradientes de forma automática sin tener que preocuparse por la obtención de gradientes.

*Pytorch*[3] utiliza el paquete llamado *Autograd*. Existe un atributo en las variables Tensor de los parámetros llamado *grad* que registra cada operación en la que influya. De esta manera se guarda el gradiente acumulado en dicho atributo.

Realizando la llamada a la función correspondiente (backwards()) sobre la variable objetivo se obtiene la derivada de los parámetros con respecto a la variable objetivo y se almacena en *grad*. En este caso se llamaría a la función backwards() desde la variable que almacene el coste.

### 3.1.6. Hiperparámetros

Los hiperparámetros son variables que utiliza el algoritmo de optimización durante el proceso y que no son parte del modelo, es decir, no aprenden a través

[3]Librería escrita en python utilizada en este TFG

de las sucesivas iteraciones de entrenamiento, pero como veremos más tarde, se pueden adaptar a los parámetros que si aprenden.

La tasa de aprendizaje es uno de ellos. Indica en que medida ha de actualizarse los parámetros en cada iteración. Un valor alto hará que los cambios sean mayores. El momento, el número de muestras por lote o el número de capas escondidas son ejemplos de hiperparámetros que se deben fijar para entrenar un modelo y que condicionan el propio proceso de entrenamiento.

## 3.2. Ejemplo

### 3.2.1. Herramientas

Para el entrenamiento del modelo se utiliza la herramienta Colaboratory de Google. Google Colaboratory permite ejecutar código Python desde el navegador. El esfuerzo computacional es realizado por las máquinas de Google en la nube. Para el manejo de matrices y vectores se utiliza el marco de trabajo Pytorch. Este framework de código abierto esta diseñado para facilitar la investigación de redes neuronales y acelerar su posterior despliegue de producción. Los gráficos utilizan la librería matplotlib.

Las dos primeras herramientas mencionadas se explican con más detalle en la sección 4.3.

### 3.2.2. Resultados

Lo explicado hasta ahora se va a aplicar al ejemplo planteado en la sección 3.1, utilizando el método del gradiente descendente y el gradiente descendente estocástico para ver los resultados. Se utiliza la misma semilla para que los dos métodos inicialicen los parámetros con el mismo valor. Los hiperparámetros también son los mismos ($lr = 0,05$). El código se puede ver en el apéndice A.

En las figuras 3.3 y 3.4 se puede comprobar la diferencia de aplicar el gradiente descendente y el gradiente descendente estocástico. En el gradiente descendente, la función de coste (figura 3.4) tiene una tendencia descendente continua mientras que en el gradiente descendente estocástico (figura 3.3) existen fluctuaciones o picos debido a la representación del conjunto de datos por un punto.

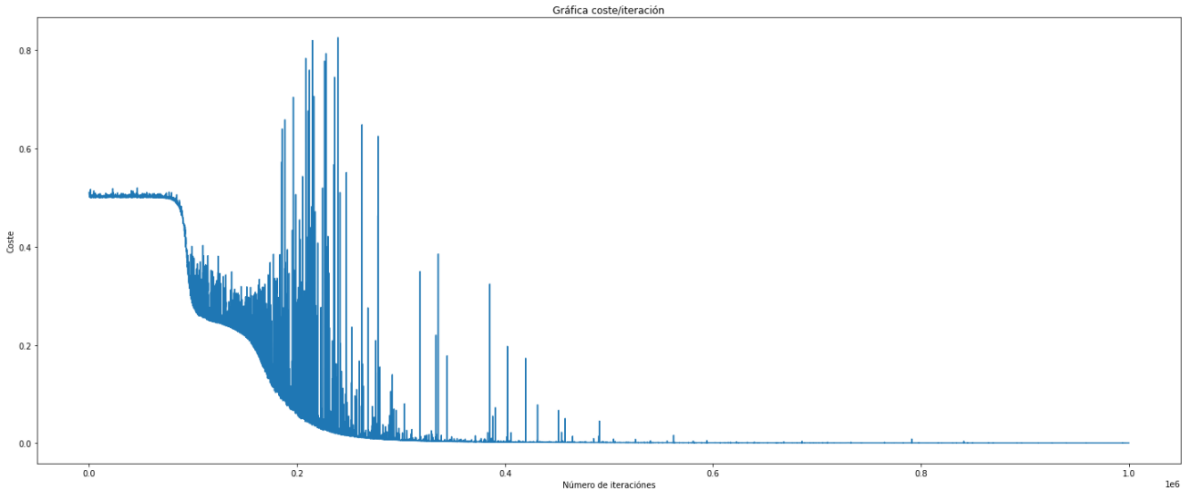



**Figura 3.3:** Función de coste del modelo SGD (Inicialización A)

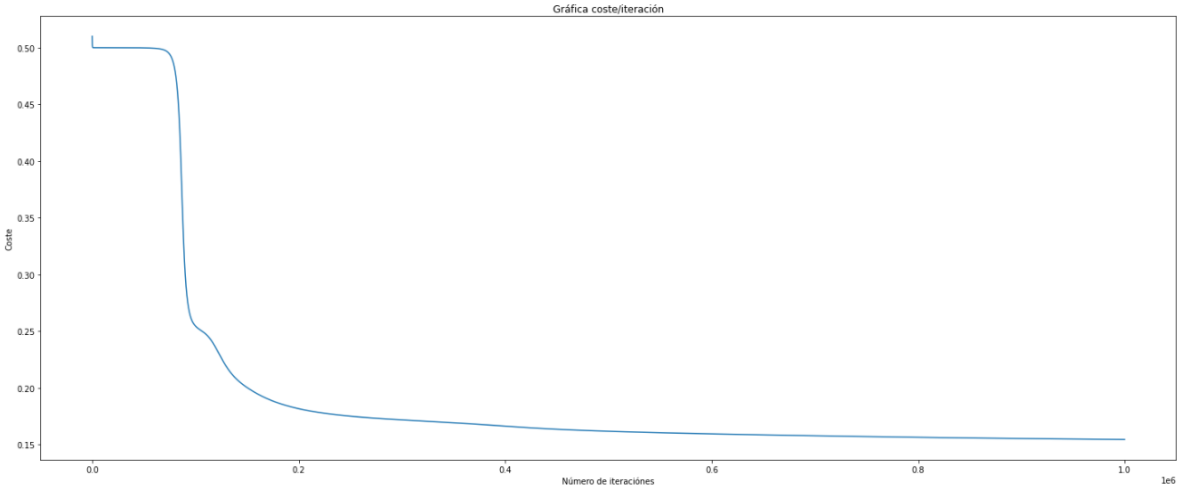



**Figura 3.4:** Función de coste del modelo GD (Inicialización A)

El modelo de GD con la inicialización A se ha quedado en un mínimo local, siendo incapaz de sobrepasarlo mientras que el SGD si ha sido capaz de llegar a un menor coste y por lo tanto, mejor resultado. El valor de la función de coste del GD es de 0.1544 mientras que el valor de la función de coste del modelo que utiliza SGD tras el entrenamiento es de 0,0004. No podemos afirmar que este sea el mínimo global.

Sin embargo, si se cambia los valores iniciales[4] de los parámetros es posible

[4]Semilla para la reproducibilidad de resultados. torch.manual_seed(89)

saltarse ese mínimo local y alcanzar un coste de 0.0004. El método del gradiente descendente es bastante dependiente de la inicialización de los parámetros. Con esta misma inicialización de parámetros el modelo SGD es capaz de alcanzar un valor de coste igual a 0.0004 en menos tiempos, es decir, disminuye el tiempo de convergencia.

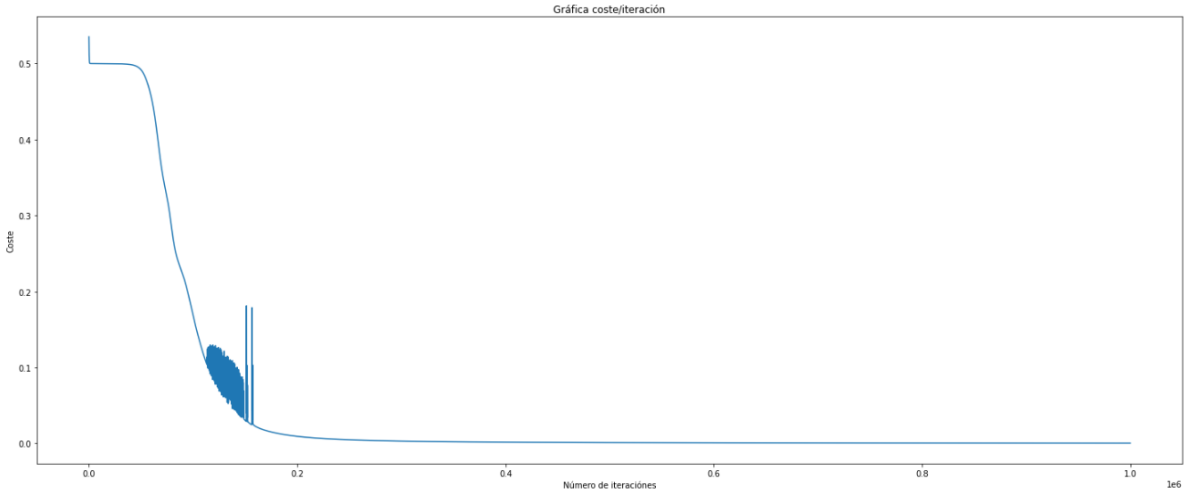



**Figura 3.5:** Función de coste del modelo GD (Inicialización B).

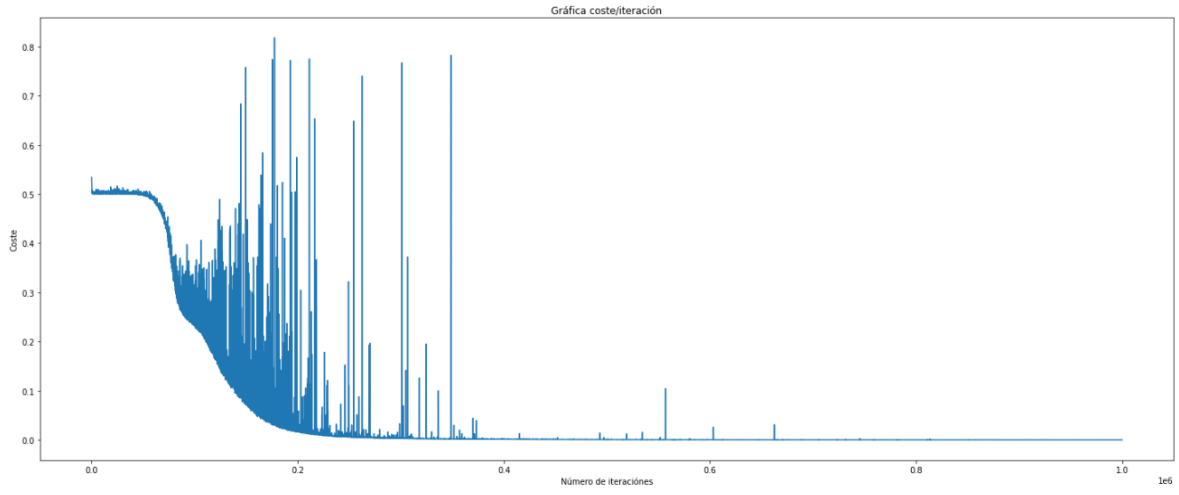



**Figura 3.6:** Función de coste del modelo SGD (Inicialización B).

Otra diferencia es el tiempo de entrenamiento, siendo en el gradiente descendente un 1200 % mayor. El tiempo tardado utilizando el gradiente descendente

estocástico es de 4 minutos y 43 segundos mientras que en el gradiente descendente es de 52 minutos y 36 segundos. Esta diferencia se incrementa conforme más grande es el conjunto de datos de entrenamiento.

Los resultados obtenidos por un método u otro son similares aunque el SGD ha sido capaz de obtener un resultado mejor en uno de los casos (Inicialización A) y en menos tiempo. Además, el SGD es capaz de sobrepasar mínimos locales con más facilidad por lo que se podría afirmar que el método del gradiente descendente estocástico es más eficiente y eficaz.

## 3.3. Validación y evaluación de redes neuronales

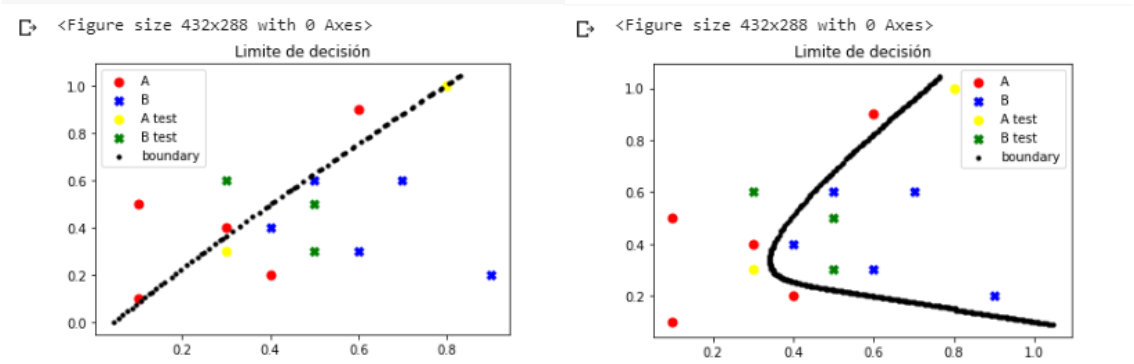
Uno de los aspectos más importantes en la construcción y desarrollo de redes neuronales para clasificación es la capacidad de generalizar, evitando la memorización de patrones de aprendizaje basados en los datos de entrenamiento. De esta manera, el objetivo es dar una respuesta correcta a muestras no presentadas durante la fase de entrenamiento [14].

En modelos con la tarea de clasificación, para la validación y evaluación de los modelos se realiza una prueba con el conjunto de datos de test. Estos datos no han sido "vistos" por el modelo, por lo que se mide la capacidad de generalización de nuestro modelo.

Se calcula el porcentaje de aciertos o predicciones correctas que el modelo es capaz de conseguir. A esto se le llama precisión o exactitud (accuracy). En otras palabras, es la fracción de predicciones correctas en un modelo de clasificación. Cuanto más alto sea este porcentaje, mayor eficacia y confianza tendrá el modelo.

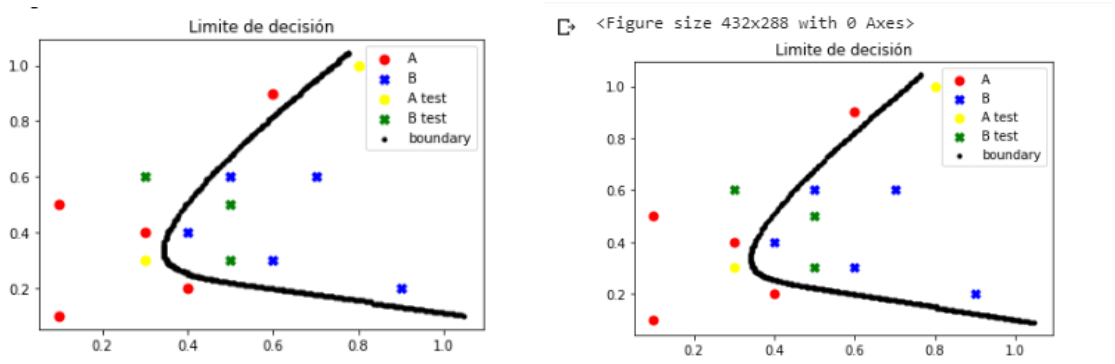
El modelo de SGD ha obtenido una precisión del 60 % en las dos inicializaciones, es decir, ha acertado $\frac{3}{5}$ predicciones mientras que el modelo de GD ha obtenido un 40 % en la inicialización A y un 60 % en la inicialización B.

En este ejemplo podemos representar los datos en un plano, por lo que la frontera de decisión es fácil de representar. La frontera de decisión marca los límites para que el output del modelo indique que un punto pertenece a una clase o a otra.




**(a)** Límite de decisión del modelo GD. **(b)** Límite de decisión del modelo SGD

**Figura 3.7:** Limites de decisión de los modelos de SGD y GD. Inicialización A




**(a)** Límite de decisión del modelo GD. **(b)** Límite de decisión del modelo SGD

**Figura 3.8:** Limites de decisión de los modelos de SGD y GD. Inicialización B

# Parte III

# Experimento con redes neuronales: Reconocimiento de imágenes con Pytorch

# Capítulo 4

# Redes neuronales para el reconocimiento de imágenes

## Contenido


**4.1 Redes neuronales para el reconocimiento de imágenes . . . 36**

4.1.1 Redes neuronales convolucionales . . . . . 37

**4.2 Algoritmos de optimización avanzados . . . . . 37**

4.2.1 Gradiente descendente estocástico con momento . . . . 39

4.2.2 ADAM . . . . . 40

**4.3 Tecnologías y librerías utilizadas en el experimentO . . . . . 42**

4.3.1 Pytorch. El framework para redes neuronales . . . . . 43

4.3.2 Google Colaboratory. Cuadernos de Python con ejecución en la nube . . . . . 45

4.3.3 TensorBoard. Visualización sencilla de datos . . . . . 45

**4.4 Comparativa de SGD, SGD con momento y ADAM con el conjunto de datos MINST . . . . . 46**

4.4.1 Datos de trabajo . . . . . 46

4.4.2 Red neuronal convolucional . . . . . 47

4.4.3 Función de coste . . . . . 51

4.4.4 Obtención del modelo óptimo. Aprendizaje y evaluación . 52

**4.5 Comparativa de SGD, SGD con momento y ADAM con el conjunto de datos CIFAR-10 . . . . . 57**

4.5.1 Datos de trabajo . . . . . 57

4.5.2 Red neuronal convolucional . . . . . 59
4.5.3 Función de coste . . . . . 62
4.5.4 Obtención del modelo óptimo. Aprendizaje y evaluación . 62

# Sinopsis

Este es el cuarto capítulo. Se introduce el reconocimiento de imágenes con redes neuronales convolucionales y se realiza el entrenamiento de dos modelos de redes neuronales convolucionales de reconocimiento de imágenes utilizando tres métodos de optimización diferentes para su posterior comparativa y análisis de los resultados obtenidos.

Los métodos de optimización utilizados son el gradiente descendente estocástico, gradiente descendente estocástico con momento y tasa de aprendizaje adaptativo (ADAM).

## 4.1. Redes neuronales para el reconocimiento de imágenes

En 2012 se presentó un estudio de la Universidad de Toronto en la Conferencia sobre sistemas de procesamiento de información neural (NIPS) llamado *ImageNet Classification with Deep Convolutional Networks* [15]. Este estudio asentó las bases de las técnicas utilizadas hoy en día en el campo de la visión computacional.

Para el experimento se utilizaron las denominadas redes neuronales convolucionales consiguiendo un 50 % de reducción de la tasa de error en ImageNet [1].

Utilizaron la función ReLU (Función 2.4) como función de activación, demostrando que tenía un mejor rendimiento en tiempo que la función $tanh$ (Función 2.3). A partir de entonces se empezó a utilizar más esta función para redes neuronales profundas. Además, se usaron técnicas de transformación de imágenes

[1] El proyecto ImageNet es una gran base de datos visual diseñada para su uso en la investigación de software de reconocimiento de objetos visuales.

para aumentar el conjunto de datos de entrenamiento y capas dropout[2] para evitar sobreajuste.

Estas método se utiliza hoy en día por una gran cantidad de tareas de visión computacional.

### 4.1.1. Redes neuronales convolucionales

El input es una imagen en una-tres dimensiones normalmente. Esta puede ser en blanco y negro (un canal; ancho x alto) o con color (tres canales; RGB x ancho x alto) en términos de la intensidad de los colores primarios de la luz (RGB).

El propósito principal de las capas convolucionales es extraer las características de las imágenes de entrada utilizando un pequeña matriz (filtro) que preserve las relaciones espaciales entre los píxeles de la misma para aprender de la imagen [16]. El filtro es utilizado como detector de características. Las capas de neuronas con esta estructura se denominan capas convolucionales, normalmente seguidas de una capa de agrupación o *pooling*. La capa o máscara convolucional se aplica a la imagen creando mapas de características (figura 4.1). Después se aplica una capa de pooling para reducir el número de datos eliminando información redundante (Figura 4.2). Los parámetros son los valores de los filtros utilizados en la convolución.

En la figura 4.7 se puede ver una red neuronal convolucional estándar con dos capas convolucionales, dos de polling y dos capas lineales. Esta red ha sido utilizada para el modelado del clasificador MNIST[3].

A lo largo de los años se han ido introduciendo mejoras en los algoritmos de optimización de redes neuronales que agilizan y optimizan el proceso de entrenamiento.

## 4.2. Algoritmos de optimización avanzados

El entrenamiento de redes neuronales trata la obtención de los pesos para un eficaz funcionamiento del modelo, es decir, como aprende la red neuronal. La función objetivo de la optimización es la función de coste.

[2]Técnica de regularización que consiste en la desactivación aleatoriamente un porcentaje de las neuronas en cada capa oculta, acorde a una probabilidad de descarte previamente definida.

[3]Base de Datos de imágenes de dígitos escritos a mano

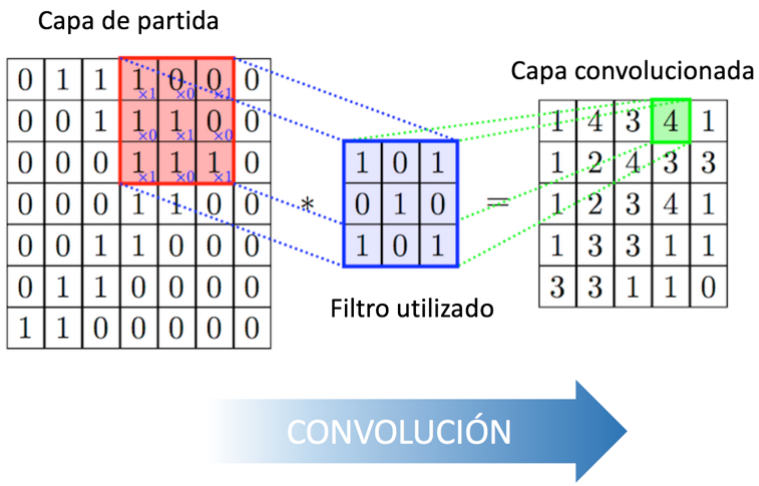



**Figura 4.1:** Función de convolución en imágenes. Fuente: Diego Calvo

Anteriormente se ha explicado el algoritmo de optimización de gradiente descendente y su variantes. Este algoritmo no garantiza una buena convergencia como se puede comprobar en el ejemplo del capítulo anterior (GD; inicialización A), y deja abiertos ciertos puntos para mejorar. Algunos de estos puntos son [17]:

- **Elegir una tasa de aprendizaje eficiente.** Una tasa de aprendizaje muy pequeña puede generar una convergencia extremadamente lenta, mientras que una tasa de aprendizaje alta puede dificultar la convergencia y provocar grandes fluctuaciones en la función de coste.
- **Evitar quedarse atrapado en mínimos locales no óptimos.** Yann N. Dauphin, un científico investigador en IA, junto con otros investigadores achacan el problema de estancarse en una solución no óptima a los denominados saddle points (Ver figura 4.3) (Puntos de silla en castellano) en vez de mínimos locales [18]. Los puntos de silla se dan cuando existe una pendiente ascendente en una dimensión y una pendiente descendente en otra dimensión para un punto dado.
- **Uso de una tasa adaptativa para cada parámetro.** En caso en los que los datos estén dispersos y con frecuencias muy diferentes sería eficiente no actualizar todos los parámetros al mismo nivel Y efectuar una actualización mayor para características que raramente aparezcan.

Estos puntos se han abordado anteriormente reduciendo la tasa de aprendizaje conforme van sucediendo las iteraciones o con unos tiempos predefinidos. Es

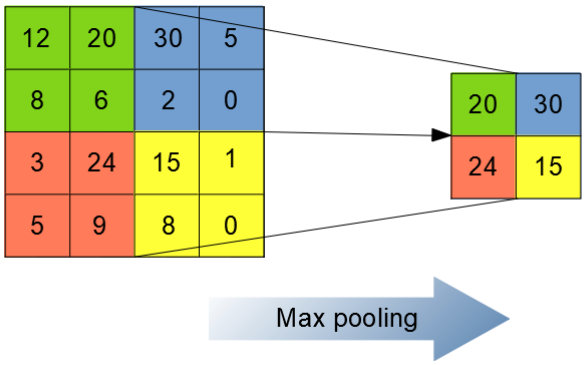



**Figura 4.2:** Función de max pooling 2x2 en imágenes.

el denominado *simulated annealing* o enfriamiento simulado. Este método tiene un inconveniente, no es capaz de adaptarse a las características del conjunto de datos de entrenamiento.

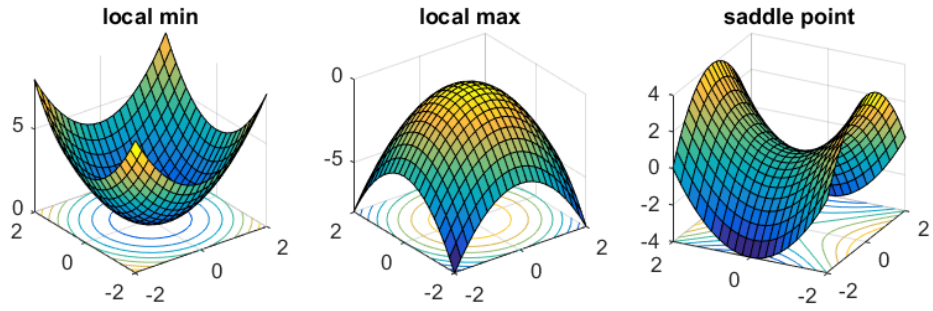



**Figura 4.3:** Puntos importantes en la optimización de funciones.
Fuente: Página web 'Off the convex path'

A lo largo de los años se han ido abordando estos puntos, mejorando y optimizando este método para obtener resultados mejores y más rápidos. Algunos de ellos son:

### 4.2.1. Gradiente descendente estocástico con momento

Del ejemplo expuesto en la sección 3.2 se puede observar fácilmente que si la función de coste es compleja (tiene bastantes parámetros), la varianza de la función de coste en el tiempo es demasiado alta, y puede fallar a la hora de converger en una solución óptima. La variante en mini-lotes del GD reduce la

varianza, en cierta medida, aprovechando la media de gradientes de los datos del lote pero puede encontrar dificultad a la hora de hacer progresos en regiones planas o converger de forma rápida. Para una convergencia más rápida y suave se utiliza el momento. [17]

La forma de actualizar los parámetros tiene en cuenta los gradientes previamente calculados. Se introduce el término gamma $\gamma$, $0 \leq \gamma \leq 1$ y el término momento $v_t$ en la fórmula de actualización (back step). $t$ indica la iteración de entrenamiento. La fórmula para actualizar los parámetros es la siguiente:

$$v_t = \gamma v_{t-1} + \eta_t \nabla_{Cost}(p)_t$$

$$p_t = p_{t-1} - v_t$$

Se puede observar que el momento suaviza la actualización de parámetros y que con $\gamma = 0$ se obtendría el mismo resultado que con el SGD. Un valor común para $\gamma$ es 0.9.

$$v_t = \eta_t \nabla_{Cost}(p)_t + \gamma v_{t-1} = \eta \nabla_{Cost}(p)_t + \gamma \eta_{t-1} \nabla_{Cost}(p)_{t-1} + \gamma^2 v_{t-2}$$

**(a)** Función de coste de SGD con momento **(b)** Función de coste de SGD sin momento

**Figura 4.4:** Evolución del valor del coste de los modelos de SGD con y sin momento. Fuente: Genevieve B. Orr

El problema de este algoritmo se produce cuando los gradientes tienen una gran diferencia en valor. Una tasa de aprendizaje alta lleva a una gran divergencia en parámetros con un gradiente alto mientras que una tasa de aprendizaje baja dificulta el aprendizaje de los parámetros con gradientes bajos.[19]

### 4.2.2. ADAM

ADAM es el resultado de la mejora de algoritmos de optimización que buscan tasa de aprendizaje diferentes y adecuadas para cada parámetro. La evolución es la siguiente:

### Adagrad

Este algoritmo hace exactamente lo dicho anteriormente, su nombre viene de *adaptative gradiente*. Realiza actualizaciones mayores para parámetros poco frecuentes y menores actualizaciones para parámetros frecuentes. Por este motivo es un buen algoritmo para tratar con datos dispersos.

*Adagrad* modifica la tasa de aprendizaje para cada parámetro en cada iteración atendiendo a los gradientes anteriormente calculados. La fórmula de actualización para cada parámetro queda de la siguiente manera:

$$p_{t,i} = p_{t-1,i} - \frac{\eta}{\sqrt{G_{t,i} + \varepsilon}} \cdot \nabla_{Cost}(p)_t$$

$G_t$ es la suma de los cuadrados de los gradientes pasados mientras que $\varepsilon$ es una constante pequeña para evitar la división por cero[4].

Se ha demostrado que este método sin la raíz cuadrada tiene un rendimiento mucho peor [17]. Se puede observar que la tasa de aprendizaje se va reduciendo con cada iteración. En este sentido, este algoritmo realiza la misma función que la técnica *simulated annealing* mencionada anteriormente.

El problema se alcanza cuando se acumulan demasiados gradientes para un parámetro dificultando la actualización del mismo. La tasa de actualización adaptada termina siendo demasiado pequeña. El siguiente algoritmo aborda este problema

### RMSProp

*RMSProp* busca reducir la agresividad de *Adagrad* y el decremento monótono de la tasa de aprendizaje adaptativa. En vez de calcular la suma acumulada de los cuadrados de todos los gradientes pasados, calcula los gradientes dentro de una ventana de tamaño fijo $n$.

A diferencia de *Adagrad*, *RMSProp* utiliza la media de los $n$ cuadrados de gradientes previos, es decir, calcula la media móvil de ventana $n$ y añade un término $\gamma$ para transformarlo en una media móvil exponencial[5]. Las fórmulas para la actualización son las siguientes:

$$E[\nabla_{Cost}(p)^2]_t = \gamma E[\nabla_{Cost}(p)^2]_{t-1} + (1-\gamma)\nabla_{Cost}(p)_t^2$$

$$p_t = p_{t-1} - \frac{\eta}{\sqrt{E[\nabla_{Cost}(p)_{t-1}^2] + \varepsilon}} \cdot \nabla_{Cost}(p)_t$$

[4]Constante que suele tener un valor de $10^{-8}$

[5]Al igual que $\gamma$ en SGD con momento su valor suele ser 0.9

### ADAM

El algoritmo *Adaptative Moment Estimation* (ADAM) busca incluir el momento y las tasas de aprendizaje adaptativas anterior, es decir, una combinación de RMSProp y SGD con momento. De esta manera se calcula la media móvil exponencial de los cuadrados de los gradientes anteriores y la media móvil exponencial de los gradientes (momento) utilizando dos betas $\beta$. La fórmula para su cálculo son las siguientes:

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1)\nabla_{Cost}(p)_t$$

$$m_t = \beta_2 m_{t-1} + (1 - \beta_2)\nabla_{Cost}(p)_t^2$$

Los autores de este algoritmo se dieron cuenta que durante las primeras iteraciones estaban demasiado influenciados por la inicialización de estas variables a 0. Para paliar este problema se realiza un paso más:

$$\hat{v_t} = \frac{v_t}{1 - \beta_1}$$

$$\hat{m_t} = \frac{m_t}{1 - \beta_2}$$

Quedando la fórmula de actualización:

$$p_t = p_{t-1} - \frac{\eta}{\sqrt{\hat{m_t}} + \varepsilon} \cdot \hat{v_t}$$

## 4.3. Tecnologías y librerías utilizadas en el experimentO

Se han utilizado las siguientes tecnologías y/o herramientas:

- Python. Lenguaje de programación.
- Google Colaboratory. Aplicación web de Cuadernos de Python.
- Pytorch. Librería para el manejo de redes neuronales con Python.
- Tensorboard. Librería para la visualización de datos y resultados.

### 4.3.1. Pytorch. El framework para redes neuronales

Pytorch es un framework escrito en Python diseñado para realizar cálculos numéricos haciendo uso de la programación de tensores[6]. Además permite la realización de estos cálculos tanto en CPU como en GPU para acelerar estos [20].

Existen otras alternativas a Pytorch. *Tensorflow*, desarrollado por Google Brain Team, es la más conocida y la que tienen una mayor comunidad de desarrolladores, investigadores y empresas que hacen uso del mismo. A parte, también se pueden utilizar *Caffe*, *Microsoft CNTK* o *Theano* para el desarrollo de proyectos de inteligencia artificial.

Se ha elegido Pytorch por las siguientes razones.

- Es una plataforma de investigación para deep learning que proporciona una gran flexibilidad y rapidez a la hora de construir modelos.
- Proporciona una interfaz sencilla para la creación de redes neuronales sin la necesidad de una librería de nivel superior como sucede con Keras para Tensoflow.
- La operaciones que se realizan sobre los objetos Tensor pueden ser acelerados utilizando la GPU. Para esta funcionalidad se utiliza una API que conecta la CPU con la GPU llamada CUDA.

EL paquete principal del framework es *torch*. Este paquete contiene las estructuras de datos utilizas para las operaciones necesarias. La principal estructura de datos es el tensor. La clase *Tensor* define esta estructura de datos que almacena los parámetros y los gradientes utilizados en el entrenamiento de redes neuronales.

El cálculo de gradientes es esencial para el entrenamiento de redes neuronales. Pytorch tiene un paquete específico (*torch.autograd*) que proporciona diferenciación automática para todas las operaciones ejecutadas sobre los tensores.

#### El paquete autograd. Diferenciación automática

Para el uso de este paquete se debe establecer a *True* el atributo *requires_grad* de los objetos *Tensor*. Las operaciones realizadas sobre los mismos se registran. Cuando se finalizan los cálculos se llama a la función *.backwards()*

[6] Utiliza la clase Tensor para el manejo de los mismos

sobre el objeto tensor que almacena el valor de la función objetivo. En el caso de modelos basado en deep learning para clasificación, es la función de coste. Por lo que se llama a la función *.backwards()* sobre el objeto tensor que almacena el valor del coste para calcular los gradientes.

Los parámetros son almacenados en objetos *Tensor* y el gradiente de los mismos es almacenado en el atributo *.grad* tras la llamada a la función *.backwards()*.

Los gradientes calculados son utilizados en los algoritmos de optimización para la actualización de los pesos y los sesgos de la red neuronal. Pytorch tiene un paquete que implementa varios algoritmos de optimización (entre ellos los utilizados para el experimento). Este es el paquete *torch.optim*

#### El paquete optim. Algoritmos de optimización

En el paquete optim se define la clase *Optimizer*, interfaz para implementar los algoritmos. Las clases hijas de *Optimizer* deben contener una lista de parámetros a optimizar. Después se establecen las opciones específicas para cada algoritmo en la implementación de los mismos como la tasa de aprendizaje, el momento, etc.

Los parámetros a optimizar no se inicializan manualmente, si no que son inicializados automáticamente al crear las capas de la red neuronal. Pytorch también proporciona una abstracción para las capas en el paquete *torch.nn*

#### El paquete nn. Redes neuronales

Este paquete para el manejo de redes neuronales incluye la definición de la clase *nn.Parameter*, clase hija de *Tensor*, que añade algunas funcionalidades específicas de los parámetros de redes neuronales. Utilizado con la clase *nn.Module*, estos parámetros son añadidos a una lista para un manejo más sencillo.

La clase *nn.Module* es la interfaz para la implementación de redes neuronales. La clase hijas deben implementar la función *forward()*. Esta función actúa como $F(x)$, es decir, contiene las operaciones o funciones que se ejecutan sobre los datos de entrada de la red neuronal para producir el output.

Las operaciones se definen en el módulo *nn.functional*. Dentro de este módulo se incluyen clases como *functional.conv2d*, una abstracción de las capas convolucionales con filtros de 2 dimensiones. Para la creación de objetos de estas clases se especifican los hyperparámetros necesarios, como el número de

neuronas en la capa[7] o la dimensión de los filtros utilizados en capas convolucionales.

### 4.3.2. Google Colaboratory. Cuadernos de Python con ejecución en la nube

Colab es un entorno gratuito de Jupyter Notebook que permite ejecutar y programa en Python desde un navegador de forma sencilla y completamente en la nube. Este entorno permite desarrollar código de forma dinámica facilitando la investigación y la prueba de fragmentos de código.

Los recursos utilizados por la plataforma de Google Colaboratory para la ejecución del código en la nube son los siguientes (Última comprobación a 1 de septiembre de 2020):

- Intel(R) Xeon(R) CPU @ 2.20GHz
- 13 gb de RAM

### 4.3.3. TensorBoard. Visualización sencilla de datos

TensorBoard es un kit de herramienta para la visualización de datos utilizados en proyectos de aprendizaje automático. Aunque originalmente ha sido desarrollado para ser utilizado con TensorFlow, Pytorch provee el paquete *torch.utils.tensorboard* para el uso de estas herramientas.

La clase *SummaryWriter* permite añadir valores a las gráficas y/o añadir imágenes al almacén de datos para su posterior visualización.

Las herramientas de TensorBoard han sido utilizadas para:

- Seguir y visualizar métricas, como el coste o el porcentaje de predicciones correctas.
- Visualización de los datos de entrada utilizados para el entrenamiento de las redes neuronales.
- Creación de gráficos que facilitan la comparación de los algoritmos de optimización.

[7] El parámetro a pasar es el output que se desea generar con la capa

Estas herramientas han sido utilizadas para le elaboración de dos clasificadores de imágenes, es decir, dos modelos de red neuronal. A continuación, se comparan los resultados obtenidos del entrenamiento del mismo modelo de red neuronal (Uno por cada conjunto de imágenes) utilizando los optimizadores de gradiente descendente estocástico, gradiente descendente estocástico con momento y ADAM[8].

Para la realización de esta comparativa se han escogido dos bases de datos.

1. **MINST**. Base de datos de imágenes de números escritos a mano. Ver figura 4.5
2. **CIFAR-10**. Base de datos de imágenes en las que aparece uno de los siguientes vehículos o animales: avión, coche, pájaro, gato, ciervo, perro, rana, caballo, barco o camión. Ver figura 4.16

## 4.4. Comparativa de SGD, SGD con momento y ADAM con el conjunto de datos MINST

El código utilizado se puede ver en el apéndice B.

### 4.4.1. Datos de trabajo

El total de muestras del conjunto de datos es de 70.000 muestras, que se han dividido entre los conjuntos de datos de entrenamiento, validación y test. Las muestras son imágenes que están codificadas como una matriz de 28 x 28 con valores del 0 al 254 siguiendo una escala de grises. Los valores de las matrices se pueden interpretar como píxeles.

La entrada de datos de entrenamiento al modelo se hace mediante lotes. Para cada modelo con diferente optimizador se repiten exactamente los mismos lotes, ya que se ha utilizado la misma semilla para el algoritmo de selección aleatorio de lotes.De esta manera, el input para el entrenamiento del modelo será un lote imágenes de $size\_lote \times 1 \times 28 \times 28$ (Figura 4.5).

El conjunto de datos de entrenamiento con un total de 60.000 muestras se ha divido en datos de entrenamiento efectivo y datos de validación para evitar el sobreajuste. El tamaño del conjunto de datos de validación es de 10.000 muestras mientras que el tamaño del conjunto de datos utilizado para el entrenamiento

[8]Adaptative Moment Estimation

es de 50.000 muestras. El conjunto de datos de prueba o test contiene 10.000 muestras.

| | Datos de entrenamiento | Datos de validación | Datos de test |
|---|---|---|---|
| **Nº de Muestras** | 50.000 | 10.000 | 10.000 |

**Tabla 4.1:** Conjuntos de datos de trabajo

**Figura 4.5:** Ejemplo lote de entrenamiento del conjunto de datos MNIST

### 4.4.2. Red neuronal convolucional

Para el conjunto de datos MNIST se ha usado una red neuronal convolucional con dos capas convolucionales con un filtro de '$channel\_depth \times 5 \times 5$', zancada (1,1) y sin relleno, dos capas de max pooling [9] de $2 \times 2$ y dos capas de neuronas feed forward.

El output final es un vector de 10 componentes, resultado de aplicar la función softmax [10] o función exponencial normalizada al output de la última capa de neuronas feed forward. Estos 10 componentes indican la probabilidad de que el dato de entrada pertenezca a la clase indicada por la posición del componente, es decir, una distribución de probabilidad.

[9] El output es el valor más alto

[10] Esta función convierte la salida de la red neuronal en probabilidades, es decir, normaliza la salida para que la suma de los componentes sea igual a uno.

Esta función es utilizada en el aprendizaje supervisado para realizar tareas de clasificación multi-clase. La función softmax es:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{4.1}$$

Los datos de entrada pasan por las siguientes capas generando los siguientes outputs.

- Capa convolucional 1 $(5 \times 5)$ (Ver figura 4.7). Genera 10 mapas de características de dimensiones $24 \times 24$ por cada imagen. El número de parámetros en esta capa que varían en cada entrenamiento son: $1 \cdot 10 \cdot 5 \cdot 5 = 250$ pesos más $10 \cdot 1$ sesgos, es decir, $260$ parámetros.

  Estos mapas son transformados a su vez por una función de max pooling de $2 \times 2$ para reducir información redundante, generando 10 mapas de características de $12 \times 12$. El output de esta capa es el resultado de aplicar la función ReLU a los 10 mapas generados.

- Capa convolucional 2 $(5 \times 5)$ (Ver figura 4.8) . Esta vez genera 20 mapas de características por lo que se aplican 20 filtros de $10 \times 5 \times 5$ por cada mapa. El número de parámetros de esta capa son $20 \cdot 10 \cdot 5 \cdot 5 = 5000$ pesos (Ver figura 4.6) más $2 \cdot 10 = 20$ sesgos. Un total de $5200$ parámetros entrenables.

  Estos mapas son transformados al igual que los anteriores por una función de pooling de $2 \times 2$. Reduciendo los mapas a una dimensión de $4 \times 4$.

```
SGD_modelo.conv2.weight.shape

torch.Size([20, 10, 5, 5])
```

**Figura 4.6:** Tamaño de la matriz de pesos de la segunda capa convolucional. MNIST

- Capas feed forward (Ver figura 4.9). El output de la capa convolucional es aplanado a un vector de 320 componentes y pasado como entrada a una capa de feed forward con 50 neuronas, una por cada valor en el output.

  Recordar que el output son 20 mapas de características de $4 \times 4$ ( $4 \cdot 4 \cdot 20 = 320$ ). Esta capa es conectada a otra de 10 neuronas, una por cada clase a la que pertenecen las imágenes ( del 0 al 9). El número de parámetros entrenables en estas capas asciende a: $320 \cdot 50 + 50 \cdot 10$ pesos y $50 + 10$ sesgos. En total $16050 + 510 = 16560$ parámetros entrenables en las dos capas feed forward.

La tabla 4.2 muestra los parámetros entrenables del modelo.

| Capa | Nº de parámetros entrenables |
|---|---|
| Capa convolucional 1 | 260 |
| Capa convolucional 1 | 5.020 |
| Capa feed forward 1 | 16.050 |
| Capa feed forward 2 | 510 |
| Total | 21.840 |

**Tabla 4.2:** Parámetros entrenables del clasificador de imágenes del conjunto de datos MNIST

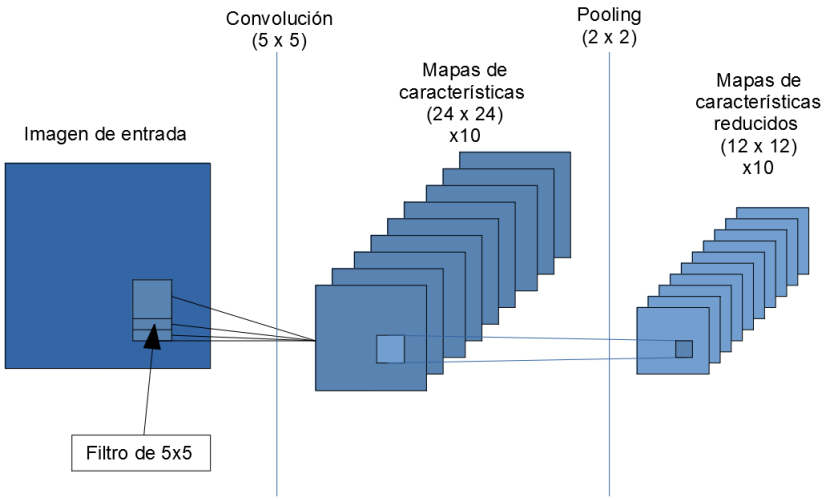



**Figura 4.7:** Red neuronal convolucional para el conjunto de datos MNIST. Primera capa convolucional

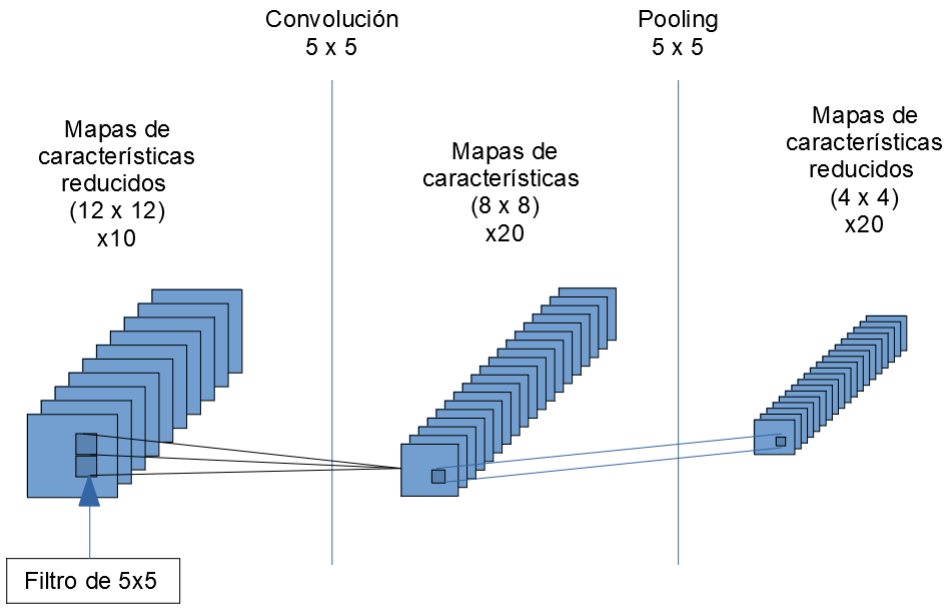



**Figura 4.8:** Red neuronal convolucional para el conjunto de datos MNIST. Segunda capa convolucional

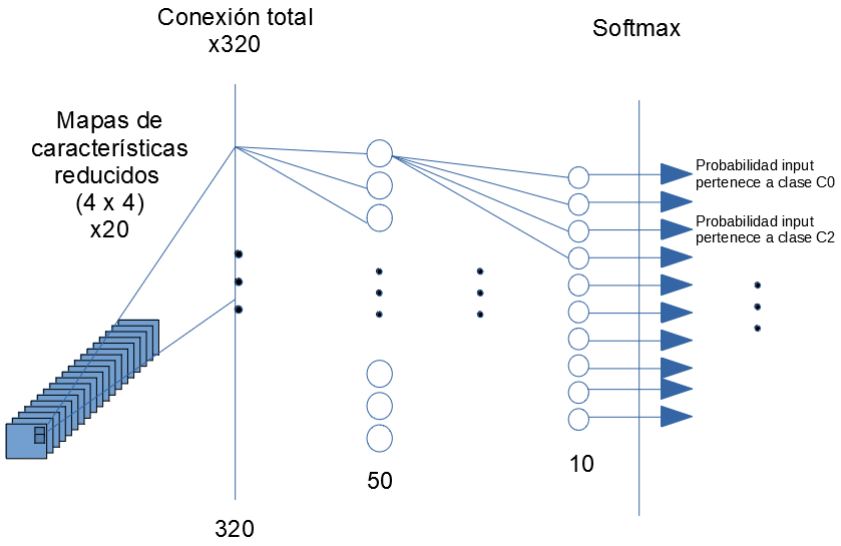



**Figura 4.9:** Red neuronal convolucional para el conjunto de datos MNIST. Capas feed forward

### 4.4.3. Función de coste

La función de coste utilizada es la función de entropía cruzada[11].Se utiliza para reflejar la precisión de las predicciones probabilísticas que la red neuronal tiene por output.

Su valor se incrementa cuando la probabilidad predicha diverge de la etiqueta correcta, es decir, una probabilidad de 0,15 en el componente de la etiqueta correcta generará un valor de la función de coste más alto que una probabilidad de 0.50 en la misma situación.

La función de entropía cruzada con 10 clases es la siguiente:

$$Cross\_Entropy(\hat{p}, pnet) = -\sum_{c=1}^{10} \hat{p}_c log(pnet_c) \tag{4.2}$$

Donde $\hat{p}$ es la distribución de probabilidad esperada, todos los componentes igual a 0 menos el componente de la clase correcta igual a 1[12] e $pnet$ es la salida de la red neuronal. Al ser el resto de componentes 0 podemos simplificar la función a la siguiente:

$$Loss\_function = -log(pnet_c)$$

Donde c es el componente del output de la red neuronal correspondiente a la etiqueta correcta.

A esta función también se le denomina *negative log likelihood loss*. Se escoge la función negativa porque se pretende minimizar la función de coste.

En los problemas de clasificación multiclase aplicar la función de *negative log likelihood loss* tiene el mismo efecto que aplicar la entropía cruzada entre la distribución de probabilidad de salida de la red neuronal (Capa con la función Softmax) y la esperada.

La función de coste aplicada al conjunto de validación y de test es la misma que la aplicada en el conjunto de datos de entrenamiento, la función de entropía cruzada.

[11] Función utilizada para cuantificar la diferencia entre dos distribuciones de probabilidad. Una la real y otra la esperada

[12] Este tipo de codificación se denomina one-hot encoding

### 4.4.4. Obtención del modelo óptimo. Aprendizaje y evaluación

#### Entrenamiento

El modelo aprende en cada iteración con los datos de entrenamiento. Estos datos están divididos en lotes de 64 imágenes. Las 64 muestras se utilizan en cada iteración para la actualización de los parámetros y el cálculo de los gradientes que utilizan los algoritmos de optimización.

Se ha entrenado el modelo por 15 épocas, es decir, que ha visto el conjunto entero de datos de entrenamiento, para alcanzar un modelo óptimo. Esto hace un total de $50{,}000 \cdot 15/64 = 11718$ iteraciones aproximadamente[13] (Actualizaciones de los parámetros $p$ del modelo). En cada actualización se tenían en cuenta 64 muestras para la actualización (Cálculo de gradientes).

La tasa de aprendizaje usada por los algoritmos de optimización es $lr = 0,01$. El momento utilizado por el algoritmo del gradiente descendente estocástico con momento es $\gamma = 0,9$. Las betas utilizadas por el algoritmo ADAM son $\beta_1 = 0,9$ y $\beta_2 = 0,999$.

El tiempo de entrenamiento transcurrido ha sido muy similar. Con los recursos disponibles (recursos de Google Colaboratory), el entrenamiento de una época es de 20 segundos. Este tiempo cambia en función de la estructura de la red neuronal y la potencia del equipo en el que se entrene la red neuronal.

A parte, se ha registrado el coste obtenido por el lote de datos de entrada por cada iteración para poder observar la progresión aproximada del coste durante el entrenamiento.

En la figura 4.10 se puede observar la evolución del valor de la función de coste conforme se va entrenando los modelos del conjunto de datos MNIST diferenciados por el algoritmo de optimización utilizado.

Los tres modelos alcanzan un error final similar tras 5.000 iteración. LLegan al mínimo que puede alcanzar el modelo propuesto. El método de SGD sin momento es el que tarda más en llegar a un valor mínimo estable, seguido del SGD con momento y por último, el modelo que utiliza el algoritmo ADAM. ADAM reduce más rápidamente el coste gracias a las mejoras que este algoritmo incluye.

Desde una vista más cercana, se puede observar que el modelo de SGD comienza a estabilizarse o tener una pendiente menos inclinada tras 5000 iteraciones, mientras que el modelo con SGD con momento y ADAM tardan 3000 y 500 iteraciones respectivamente.

[13] La iteraciones exactas han sido 11.730.


**Figura 4.10:** MNIST. Evolución del coste del lote de entrenamiento por iteración (Smooth = 0,6). Datos de entrenamiento

El modelo con ADAM converge un 900 % más rápido que el modelo con SGD atendiendo al número de iteraciones de entrenamiento. Sin embargo, este es un problema sencillo y no se puede apreciar bien la cuantía de la mejora.

En la tabla 4.3 se puede ver los costes del conjunto de datos de entrenamiento tras el entrenamiento.

| | ADAM | SGDM | SGD |
|---|---|---|---|
| **Coste con los datos de entrenamiento** | 0,04201 | 0,03225 | 0.054 |

**Tabla 4.3:** Coste del conjunto de entrenamiento tras la última iteración.

### Test

El conjunto de datos de prueba esta compuesto por 10.000 imágenes no utilizadas para el entrenamiento. Estos datos son utilizados para medir el desempeño o capacidad de generalización del modelo obtenido, ya que, se prueba con datos no vistos anteriormente por el mismo.

Se ha medido la precisión y el coste del modelo óptimo para medir dicha cualidad de los modelos con los distintos algoritmos de optimización. El modelo óptimo es aquel que tiene una capacidad de generalización mayor. Se ha utilizado el total de imágenes del conjunto para la medición de las métricas.

En la tabla 4.4 se puede observar las precisiones obtenidas con el modelo óptimo y los datos de test.

| | ADAM | SGDM | SGD |
|---|---|---|---|
| **Precisión con los datos de test ( %)** | 98,2 | 98,63 | 98,18 |
| **Coste con los datos de test** | 0.05748 | 0,0429 | 0.054 |

**Tabla 4.4:** Métricas del clasificador Óptimo con los datos de test.

Se observa que la precisión del modelo óptimo que utiliza SGDM (98,63 %) es mayor que la del modelo que utiliza ADAM (98,2 %), y a su vez, mayor que el modelo que utiliza SGD (98,18 %).

El coste de los datos de prueba de los modelos óptimos concuerda con los resultados obtenidos de precisión, el modelo con SGDM es el que menor coste consigue con los parámetros óptimos ($p*$).

En condiciones óptimas el modelo con ADAM tendría mejores resultados. Pero la elección de una tasa de aprendizaje demasiado alta hace que el coste de entrenamiento sea difícil de minimizar y se generen fluctuaciones como se puede comprobar en la figura 4.14.

Repitiendo el mismo procedimiento y cambiando la tasa de aprendizaje a $lr = 0,001$ se obtienen los resultados de la tabla 4.5. Resalta el hecho de la importancia de la elección e influencia de hiperparámetros en los distintos algoritmos de optimización.

| | ADAM | SGDM | SGD |
|---|---|---|---|
| **Precisión con los datos de test ( %)** | 98,8 | 98,07 | 90,93 |
| **Coste con los datos de test** | 0,01524 | 0,05792 | 0,3271 |

**Tabla 4.5:** Métricas del clasificador Óptimo con los datos de test. lr = 0,001

### Validación (early stopping)

Durante el entrenamiento de redes neuronal se puede dar un sobreajuste del modelo a los datos de entrenamiento. El sobreajuste implica que seguir entrenando el modelo con esos datos hace que se incremente el error de generalización, es decir, la precisión del modelo ante nuevos datos no vistos anteriormente sea más baja. El objetivo del modelo es que tenga una buena capacidad para generalizar.

Para solucionar este problema se ha implementado el método denominado parada temprana o early stopping basado en validación para obtener el modelo

óptimo [21]. A las técnicas que tienen por objetivo evitar el sobreajuste se las conoce como técnicas de regularización.

Este método consiste en una serie de comprobaciones para guardar los valores del modelo óptimo cuando la precisión, el valor de la función de coste o un indicador sobre el desempeño del modelo sobre un conjunto de datos no utilizados para el entrenamiento no mejora durante un número $patience = x$. Este conjunto de datos utilizados se suele extraer del conjunto de datos de entrnamiento y se denomina conjunto de datos de validación. Se ha fijado en 10.000 muestras.

Para medir el error de generalización se ha utilizado el coste de los datos de validación. Para la parada temprana se define un accionador o trigger que active y guarde el modelo óptimo tras una serie de comprobaciones. Las comprobaciones se realizan cada época, es decir, cada 50.000 muestras vistas por el modelo. Cada época que se reduzca el coste de los datos de validación, el valor de los parámetros $p$ del modelo se guardan.

En el caso de que el valor de la función de coste sobre los datos de validación no mejore (se reduzca) durante 3 épocas seguidas, se activa el accionador, fijando el valor mínimo encontrado antes de la parada temprana y declarando y almacenando los parámetros de la red neuronal utilizados en la obtención de ese valor mínimo como parámetros óptimos ($p*$).

En la figura 4.11 se puede observar la salida del sistema de early stopping implementado tras el entrenamiento de los modelos.

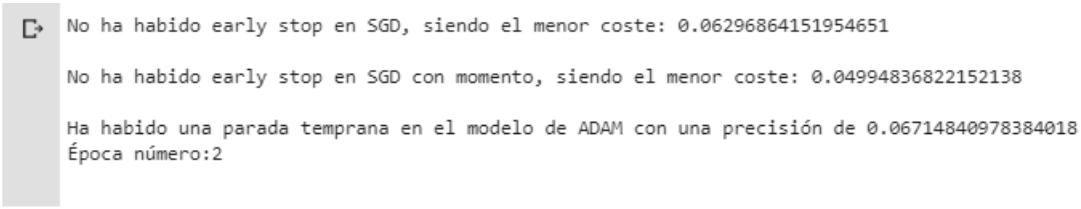



**Figura 4.11:** MNIST. Early stopping.

La figura 4.12 muestra los valores de la función de coste sobre el conjunto de datos de entrenamiento y validación del modelo que utiliza SGD como optimizador. No se observa que haya un sobreajuste del modelo a los datos de entrenamiento.

La figura 4.13 muestra los mismos datos pero para el modelo que utiliza SGDM. No se observa un sobreajuste del modelo a los datos de entrenamiento. El coste de validación y entrenamiento tienen una tendencia monótona decreciente.

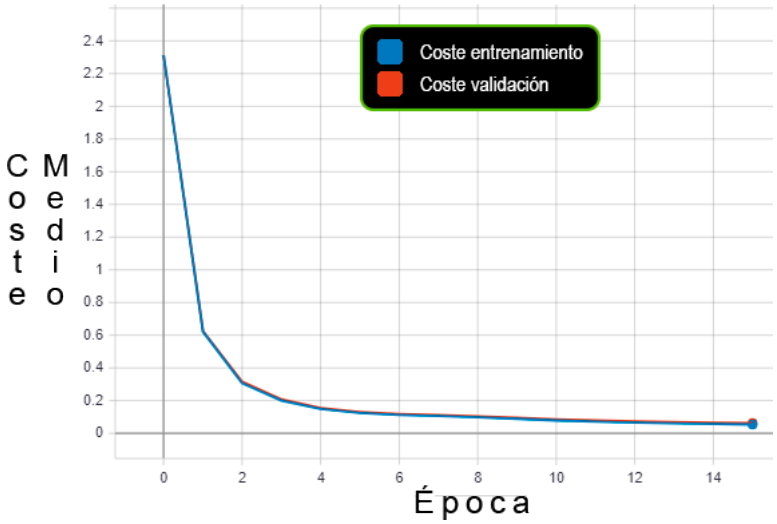



**Figura 4.12:** MNIST. Evolución del coste de entrenamiento y validación por épocas. SGD. Early stopping

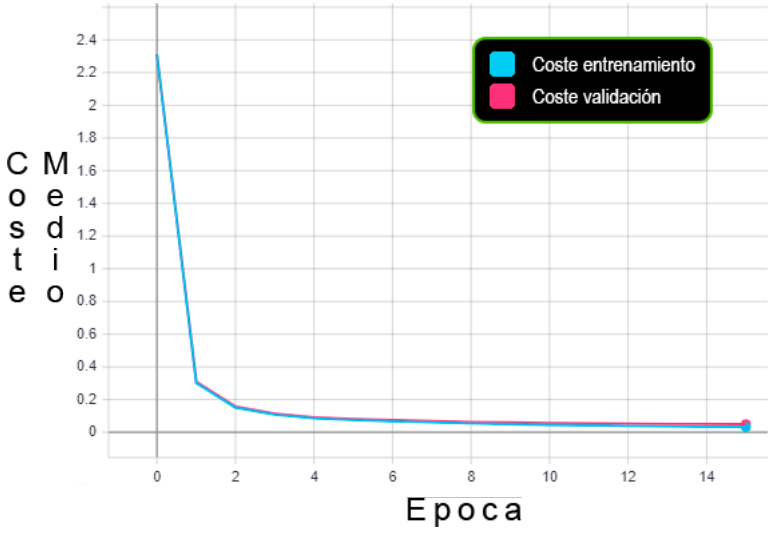



**Figura 4.13:** MNIST. Evolución del coste de entrenamiento y validación por épocas. SGDM. Early stopping

La figura 4.14 muestra los costes del modelo que utiliza ADAM. Se alcanza un mínimo en la segunda época, pero existen otro cambios de tendencia a diferencia de los modelos anteriores. Alcanza un modelo óptimo casi de manera inmediata, resaltando la hipótesis de una tasa de aprendizaje demasiado alta. en las épocas posteriores se produce una fluctuación del coste de validación con tendencia creciente que indica que se ha producido un sobreajuste. La elección de una

tasa de aprendizaje alta dificulta la obtención de un modelo óptimo.


**Figura 4.14:** MNIST. Evolución del coste de entrenamiento y validación por épocas. ADAM. Early stopping

Sin embargo, el mejor desempeño con los hiperparámetros utilizados lo obtiene el modelo con SGDM. En la figura 4.15 se puede ver la progresión de la precisión de los modelos con los distintos optimizadores, siendo el modelo que utiliza ADAM como algoritmo de optimización, el más rápido en alcanzar una precisión adecuada u óptima. Sin embargo, la precisión no tiene una tendencia monótona creciente, sino que fluctúa con una tendencia decreciente indicando un ratio de aprendizaje demasiado alto y un sobreajuste.

## 4.5. Comparativa de SGD, SGD con momento y ADAM con el conjunto de datos CIFAR-10

El código utilizado se puede ver en el apéndice C.

### 4.5.1. Datos de trabajo

El conjunto de datos CIFAR-10 tiene un total de 60.000 muestras divididos en conjunto de datos de entrenamiento, validación y test.

Las imágenes tienen color por lo que están codificadas como una matriz de

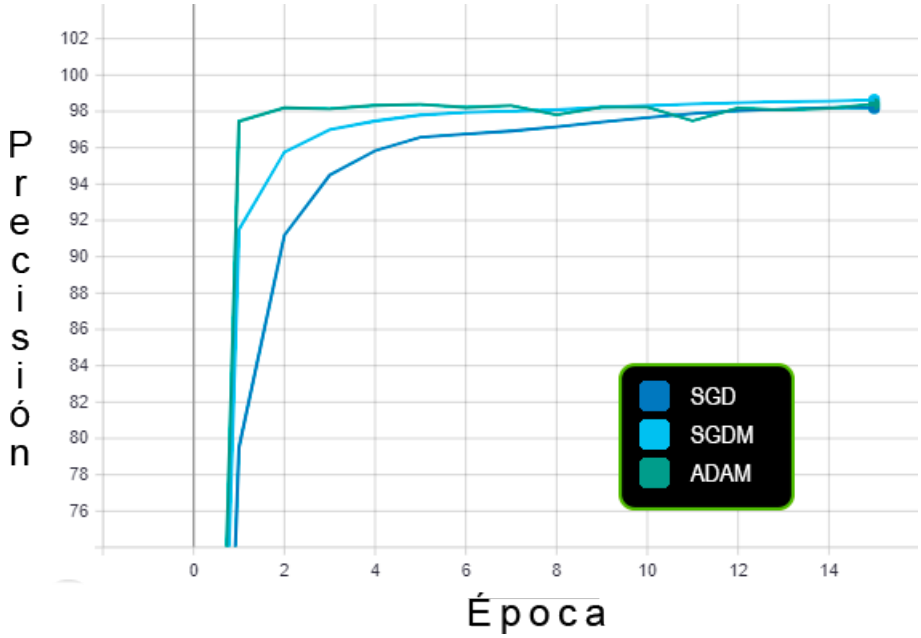



**Figura 4.15:** MNIST. Evolución de la precisión por época. Conjunto de datos de validación

3 dimensiones. La primera dimensiones compuesta por tres componentes corresponden a los tres colores primarios rojo, verde y azul. A estos componentes se les denomina canales. Cada canal esta compuesto por una matriz de orden $32 \times 32$ que indican la intensidad del color correspondiente.

La entrada de datos de entrenamiento al modelo se hace mediante lotes. Para cada modelo con diferente optimizador se repiten exactamente los mismos lotes, ya que se ha utilizado la misma semilla para el algoritmo de selección aleatorio de lotes.De esta manera, el input para el entrenamiento del modelo será una matriz de $size\_lote \times 3 \times 32 \times 32$ (Ver figura 4.16)

El conjunto de datos de entrenamiento con un total de 50.000 muestras se ha divido en datos de entrenamiento y datos de validación al igual que en el modelo para el conjunto de datos MNIST. El conjunto de datos de validación tiene 10.000 muestras mientras que el conjunto de datos utilizado para el entrenamiento es de 40.000 muestras. El tamaño del conjunto de datos de test es de 10.000 imágenes (Ver tabla 4.6)

| | Datos de entrenamiento | Datos de validación | Datos de test |
|---|---|---|---|
| **Nº de Muestras** | 40.000 | 10.000 | 10.000 |

**Tabla 4.6:** Conjuntos de datos de trabajo. CIFAR-10

**Figura 4.16:** Ejemplo lote de entrenamiento del conjunto de datos CIFAR-10

### 4.5.2. Red neuronal convolucional

Para el conjunto de datos CIFAR-10 se ha usado una red neuronal convolucional con dos capas convolucionales con un filtro de $channel\_depth \times 5 \times 5$, zancada (1,1) y sin relleno, dos capas de max pooling de $2 \times 2$ y tres capas de neuronas feed forward.

Al igual que en el clasificador de imágenes para el conjunto de datos MNIST, el output final es un vector de 10 componentes resultado de aplicar la función softmax (Función 4.1).

Los datos de entrada pasan por las siguientes capas generando los siguientes outputs.

- Capa convolucional 1 $(5 \times 5)$ (Ver figura 4.18). Genera 6 mapas de características de dimensiones $28 \times 28$ por cada imagen, es decir, un tamaño de [4, 6, 28, 28]. El número de parámetros en esta capa que varían en cada entrenamiento son: $3 \cdot 6 \cdot 5 \cdot 5 = 450$ pesos y $6$ sesgos. Un total de $456$ parámetros entrenables en la primera capa convolucional.

Estos mapas son transformados a su vez por una función de max pooling de $2 \times 2$ para reducir información redundante, generando 6 mapas de características de $14 \times 14$. El output de esta capa es el resultado de aplicar la función ReLU a los 6 mapas generados.

- Capa convolucional 2 ($5 \times 5$) (Ver figura 4.19). Esta vez genera 16 mapas de características, se aplican filtros de $6 \times 5 \times 5$ por cada mapa. El número de parámetros de esta capa son $16 \cdot 6 \cdot 5 \cdot 5 = 2400$ pesos (Ver figura 4.17 más $16$ sesgos. Sumando un total de $2416$ parámetros entrenables en la segunda capa convolucional.

  Estos mapas son transformados al igual que los anteriores por una función de pooling de $2 \times 2$. Reduciendo los mapas a una dimensión de $5 \times 5$.

```
SGD_modelo.conv2.weight.shape
torch.Size([16, 6, 5, 5])
```

**Figura 4.17:** Tamaño de la matriz de pesos de la segunda capa convolucional. CIFAR-10

- Capas feed forward (Ver figura 4.20). El output de la capa convolucional es aplanado a un vector de $16 \cdot 5 \cdot 5 = 400$ componentes y pasado como entrada a una capa de feed forward con 120 neuronas. Esta capa es conectada a otra de 84 neuronas y esta a su vez a una de 10 neuronas, una por cada clase a la que pertenecen las imágenes ( del 0 al 9). El número de parámetros entrenables en estas capas asciende a: $400 \cdot 120 + 120 \cdot 84 + 84 \cdot 10$ pesos y $120 + 84 + 10$ sesgos. En total $48120 + 10164 + 850 = 59134$ parámetros entrenables en las capas feed forward.

La tabla muestra los parámetros entrenables del modelo.

| Capa | Nº de parámetros entrenables |
|---|---|
| Capa convolucional 1 | 456 |
| Capa convolucional 1 | 2416 |
| Capa feed forward 1 | 48.120 |
| Capa feed forward 2 | 10.164 |
| Capa feed forward 3 | 850 |
| Total | 62.106 |

**Tabla 4.7:** Parámetros entrenables del clasificador de imágenes del conjunto de datos Cifar-10

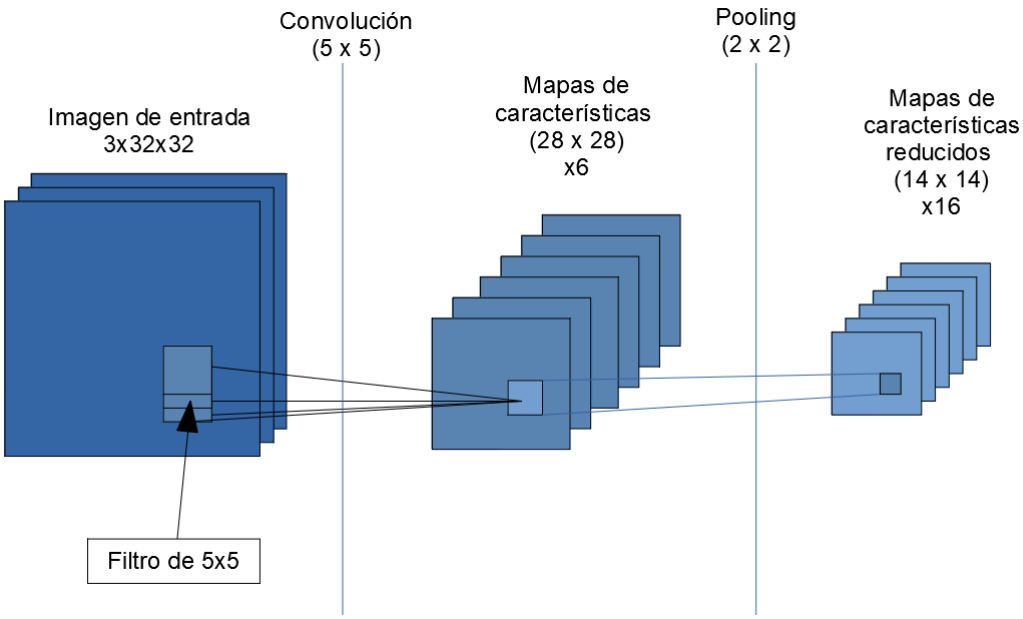



**Figura 4.18:** Red neuronal convolucional para el conjunto de datos CIFAR-10. Primera capa convolucional

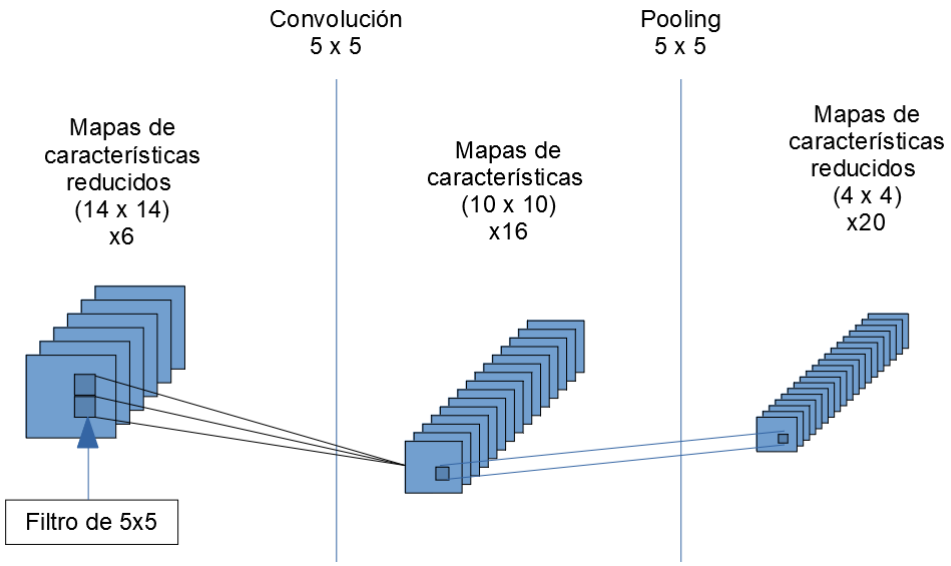



**Figura 4.19:** Red neuronal convolucional para el conjunto de datos CIFAR-10. Segunda capa convolucional

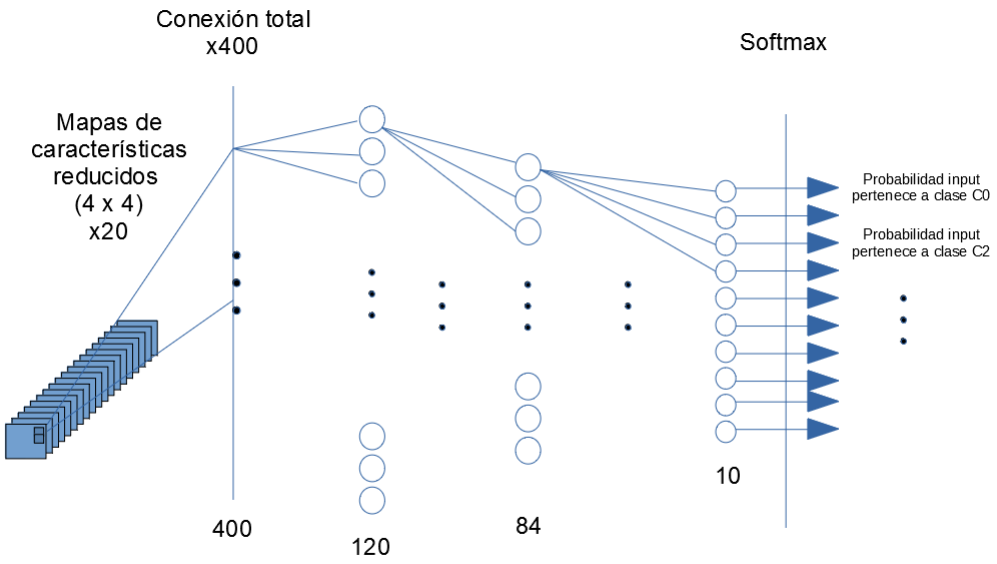



**Figura 4.20:** Red neuronal convolucional para el conjunto de datos CIFAR-10. Tercera capa convolucional

### 4.5.3. Función de coste

La función de coste utilizada es la función de entropía cruzada (Función 4.2). Es utilizada para medir el coste de los conjuntos de datos de entrenamiento, test y validación.

### 4.5.4. Obtención del modelo óptimo. Aprendizaje y evaluación

#### Entrenamiento

Los datos de entrenamiento están divididos en lotes de 25 muestras. Estas 25 imágenes son utilizadas en cada iteración para el cálculo de los gradientes que utilizan los algoritmos de optimización en la actualización de parámetros.

Se ha entrenado el modelo por 40 épocas, ya que la complejidad del problema es mayor que con MNIST. En total se han realizado $40{,}000 \cdot 40/25 = 64{,}000$ iteraciones. En cada una de ellas se ha utilizado 25 muestras para actualizar los parámetros del modelo.

La tasa de aprendizaje utilizadas por los algoritmos de SGDM y ADAM ha

sido de $lr = 0{,}0005$ y de $lr = 0{,}001$ para el algoritmo de SGD. Se ha tomado esta decisión para que el modelo con SGD disminuyera el coste más rápidamente, ya que con el $lr$ utilizado por los otros algoritmos a penas se producía mejora. El momento utilizado por el SGDM es de $\gamma = 0{,}9$. Las betas utilizadas por el algoritmos ADAM son $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$.

El tiempo de entrenamiento transcurrido en una época ha sido muy similar. El entrenamiento de una época para los modelos con SGD, SGDM y ADAM han sido 17, 18 y 19 segundos respectivamente. El más lento es ADAM al tener que realizar más operaciones para las actualizaciones.

Sin embargo el tiempo de entrenamiento transcurrido hasta encontrar el modelo óptimo ha tenido una gran diferencia. Tardando 7 minutos y 20 segundos el modelo con ADAM, 15 minutos y 51 segundos el modelo con SGDM. EL modelo con SGD todavía puede mejorar con más entrenamiento.

En la figura 4.21 se puede observar la evolución del valor de la función de coste sobre los datos de entrenamiento utilizados conforme se va entrenando los modelos.

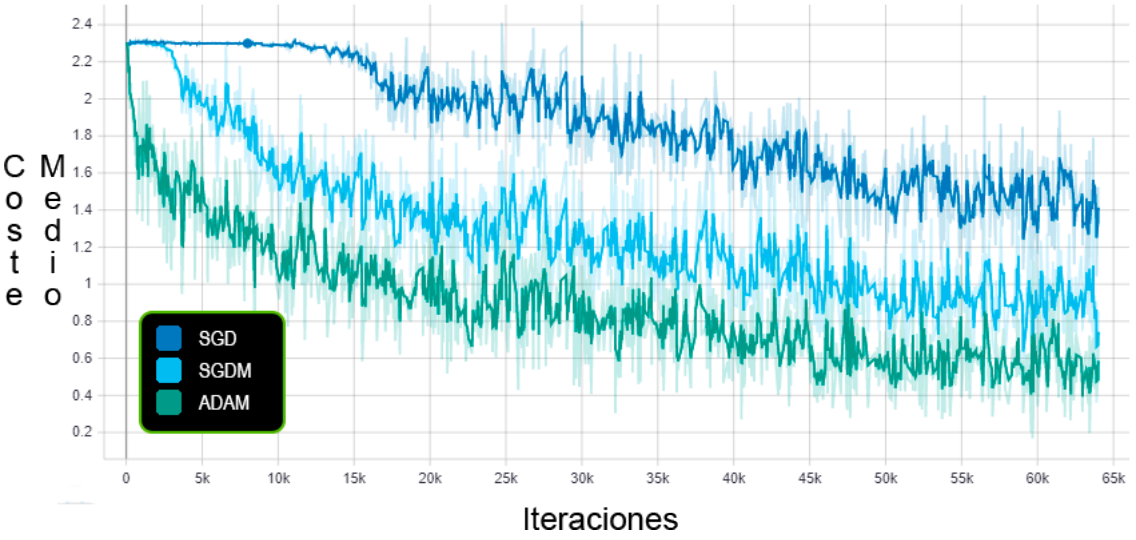



**Figura 4.21:** CIFAR-10. Evolución del coste por iteración (Smooth = 0,6). Datos de entrenamiento

El clasificador que utiliza SGD sin momento como optimizador mejora muy lentamente desde un principio y no tiene una mejora notable hasta haber realizado 15.000 iteraciones. El modelo con SGDM tarda cerca de 2.500 iteraciones (1,56 épocas) en comenzar a disminuir mientras que el modelo con ADAM comienza a disminuir casi de manera inmediata.

Los valores de los costes reflejan las mejoras incluidas en los distintos optimizadores para la rapidez de convergencia. La tabla 4.8 muestra los costes del conjunto de datos de entrenamiento tras la finalización de la última iteración.

| | ADAM | SGDM | SGD |
|---|---|---|---|
| **Coste con los datos de entrenamiento** | 0,7738 | 0,8908 | 1,436 |

**Tabla 4.8:** Coste del conjunto de entrenamiento tras la última iteración.

### Test

El conjunto de prueba esta compuesto por 10.000 imágenes no utilizadas para el entrenamiento. Se ha utilizado la precisión y el coste sobre los datos de test para medir la capacidad de generalización del modelo obtenido. Utilizando el total de imágenes del conjunto para la medición de las métricas.

Los resultados demuestran que el modelo que utiliza ADAM tiene una capacidad de generalización ligeramente mayor que el modelo óptimo con SGDM. SGD se queda atrás con una precisión de solamente 47,52 %. La tabla 4.9 indica las precisiones y los costes obtenidos con el modelo óptimo y los datos de test.

| | ADAM | SGDM | SGD |
|---|---|---|---|
| **Precisión con los datos de test ( %)** | 61,50 | 58,71 | 47,52 |
| **Coste con los datos de test** | 1,109 | 1,187 | 1.445 |

**Tabla 4.9:** Métricas del clasificador Óptimo con los datos de test.

#### Estado del arte del modelo óptimo para CIFAR-10

La precisión máxima obtenida en el modelo de ADAM se ha obtenido para unos hiperparámetros definidos, una topología sencilla y un conjunto de datos fijo sin normalizar. Es decir, se ha fijado el límite de mejora en los distintos algoritmos de optimización utilizados.

Existe una gran cantidad de variantes u opciones, eligiendo combinaciones diferentes de lo anteriormente citado. Actualmente la precisión máxima alcanzada en el problema de CIFAR-10 es de 99,37 % [22]. Esto se debe a la inclusión de técnicas avanzadas que no se han cubierto en este proyecto por ser una introducción al reconocimiento de imágenes.

La mejoras añadidas pueden ser tomadas como posibles líneas futuras de

continuación del trabajo. El aspecto principal que incluye el proyecto del modelo con un 99,37 % de precisión anteriormente citado es:

- Paradigma "Big Transfer". Se basa en la transferencia de conocimiento ya aprendido. Para ello se utilizan modelos pre-entrenados sobre conjuntos de datos genéricos grandes. Después, se adaptan los hiperparámetros a la tarea específica (*fine-tuning hyperparameters*).

  Se empieza a entrenar el modelo con una conocimiento sobre imágenes previo y no aleatorio. Este sistema llega a alcanzar un 97 % de precisión incluso con 10 muestra por clase en CIFAR-10.

Esta mejora fue motivada por la necesidad de contar con grandes cantidades de datos específicos para la tarea de clasificación que se quiera hacer. Recolectar esta gran cantidad de datos puede ser demasiado costoso dependiendo de la tarea a realizar.

Además de esta técnica, se pueden utilizar otras como el pre-procesamiento de datos de entrada (giros, rotación, escalados, etc.), también conocido como *data augmentation* en inglés.

### Validación (early stopping)

El conjunto de datos de validación contiene 10.000 muestras. Para medir el error de generalización se ha utilizado la valor de la función de coste sobre los datos de validación medidos tras cada época (40.000 muestras). En el caso de que el coste no haya mejorado durante 3 comprobaciones, los parámetros del modelo se guardan ($p*$).

En la figura 4.22 se puede observar la salida del sistema de early stopping implementado tras el entrenamiento de los modelos.

```
No ha habido early stop en SGD, siendo el menor coste: 1.4608945846557617

Ha habido una parada temprana en el modelo de SGD con momento con coste de 1.1873843669891357
Época número:33

Ha habido una parada temprana en el modelo de ADAM con coste de 1.117644190788269
Época número:14
```

**Figura 4.22:** CIFAR-10. Early stopping.

La figura 4.23 muestra los valores de la función de coste sobre el conjunto de datos de entrenamiento y validación del modelo que utiliza SGD como opti-

mizador. Las dos gráficas disminuyen de forma constante indicando que no se ha producido un sobreajuste ni se ha alcanzado un modelo óptimo, es decir, se puede seguir entrenando para ganar capacidad de generalización.

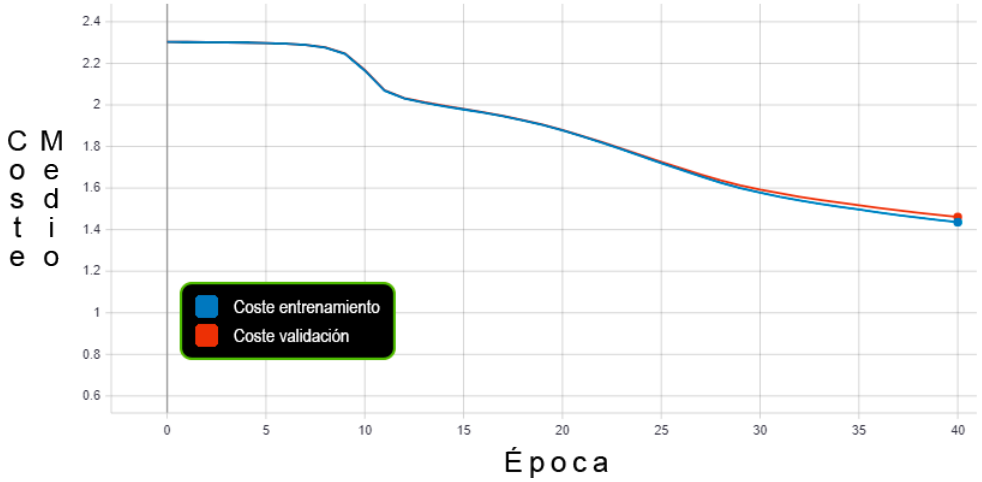



**Figura 4.23:** CIFAR-10. Evolución del coste de entrenamiento y validación por épocas. SGD. Early stopping

La figura 4.24 muestra los mismos datos pero para el modelo que utiliza SGDM. Tras la época número 33 la gráfica cambia de tendencia ligeramente a creciente. Esto indica que se ha producido un sobreajuste y seguir entrenando el modelo provocaría una disminución de aciertos en la predición de imágenes nuevas.

La figura 4.25 muestra los costes del modelo que utiliza ADAM. Se alcanza un mínimo en la época número 14. El coste final del conjunto de datos de validación tras la última época es mayor que en los modelo con SGDM y SGD. Esto resalta la importancia de parar el entrenamiento o guardar los parámetros $p*$ del modelo óptimo.

El valor del coste del modelo con ADAM es de 1,638 (El mínimo en la época 14 es de 1,118), mientras que en el modelo con SGDM es de 1,207 y en el modelo con SGD es de 1,461. Este hecho indica que una vez producido sobreajuste, continuar entrenando reduce la capacidad de generalización del modelo de forma regular o constante.

En la figura 4.26 se puede observar como va aumentando la precisión de los modelos sobre el conjunto de datos de validación. Llegados a la época fijada para la obtención del modelo óptimo, la precisión empieza a disminuir. No se observa que haya sobreajuste en el modelo con SGDM. La elección de la métrica de

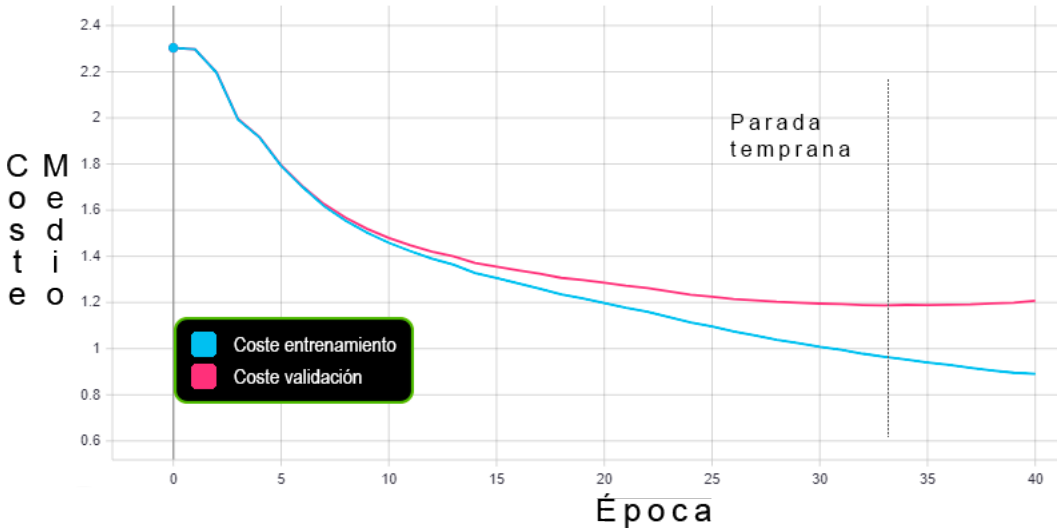



**Figura 4.24:** CIFAR-10. Evolución del coste de entrenamiento y validación por épocas. SGDM. Early stopping

precisión en vez de la de coste para implementar la técnica de early stopping significa que el modelo óptimo se va a obtener en una época posterior.

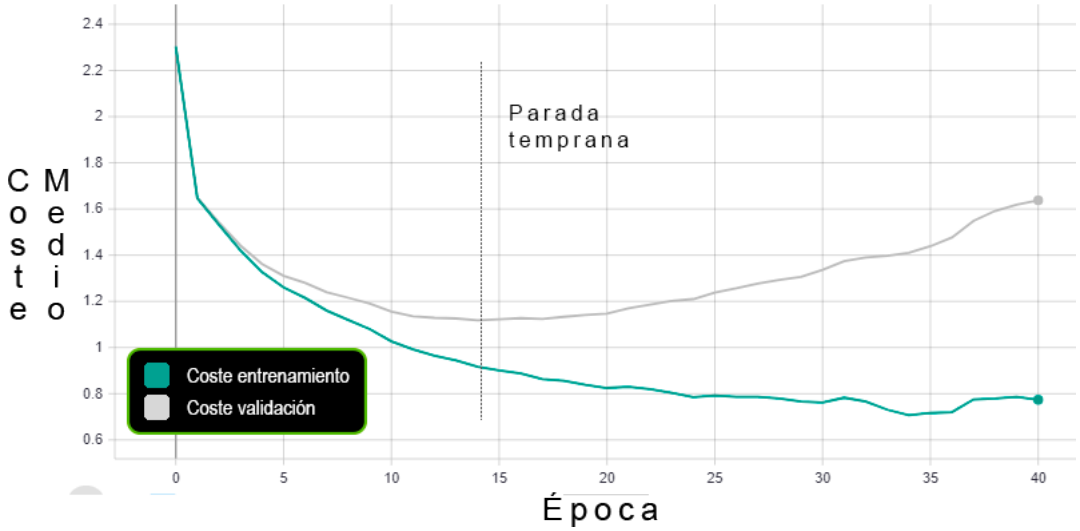



**Figura 4.25:** CIFAR-10. Evolución del coste de entrenamiento y validación por épocas. ADAM. Early stopping

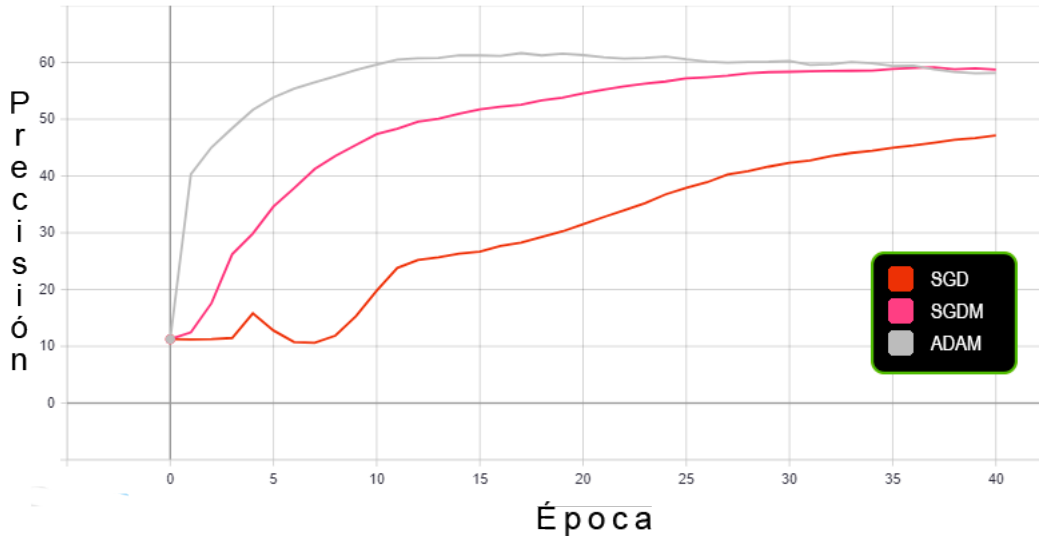



**Figura 4.26:** CIFAR-10. Evolución de la precisión por iteración. Conjunto de datos de validación

# Parte IV

# Conclusión.

# Conclusiones y líneas futuras

## 4.6. Objetivos conseguidos

En el presente TFG se ha explicado con términos sencillos y/o explicados durante el desarrollo del mismo la construcción de redes neuronales para el reconocimiento de imágenes. Se ha desarrollado y diseñado diferentes modelos de aprendizaje automático basado en deep learning. Estos modelos son dos clasificadores de imágenes para los conjuntos de datos de MNIST y CIFAR-10.

Para concluir, se ha entrenado y realizado una comparativa de algoritmos de optimización para minimizar la función de coste en los dos problemas de clasificación multiclase. Los algoritmos comparados son el gradiente descendente estocástico sin momento, gradiente descendente estocástico con momento y ADAM. Siendo este último un algoritmo centrado en redes neuronales profundas, base de la visión artificial.

Los resultados muestran que se ha conseguido construir dos clasificador 100 % funcionales llegando a alcanzar una precisión del 98 % en el clasificador de imágenes del conjunto de datos MNIST y del 61,50 % en el clasificador de imágenes del conjunto de datos CIFAR-10.

## 4.7. Observaciones sobre los resultados obtenidos

De los resultados del experimento se han obtenido las siguientes observaciones.

Las mejoras añadidas al optimizador hacen que el tiempo de convergencia sea menor y que los resultados obtenidos sean mejores. El deep learning utilizado para la clasificación de imágenes, al ser un aprendizaje supervisado, trata la optimización de una función de coste. El algoritmo que se utilice para optimizar juega un papel importante.

En este sentido el algoritmo ADAM consigue disminuir la función de coste más rápidamente a un valor de coste más bajo que los otros dos algoritmos de la comparativa (SGD y SGDM). Sin embargo, esta mejora se vuelve casi inexistente en modelos que abordan un problema demasiado sencillo como el de MNIST. A medida que el problemas se vuelve más complejo, la mejora obtenida por el algoritmo ADAM, se hace más notable.

Las redes neuronales convolucionales también mejoran el rendimiento de la red neuronal para el reconocimiento de imágenes, debido a que el número de parámetros entrenables es considerablemente menor (Ver tabla 4.10), es capaz de obtener las características de las imágenes de una manera más sencilla y rápida.

| Modelo | Parámetros capas convolucionales | Parámetros capas feed forward |
|---|---|---|
| MNIST | 5.460 | 16.560 |
| CIFAR-10 | 2.872 | 59.134 |

**Tabla 4.10:** Parámetros por tipos de capas

Además, las capas convolucionales son capaces de captar las relaciones entre valores contiguos, característica muy importante para la extracción de conocimiento de las imágenes.

La medición de si se ha producido sobreajuste o no, depende de la métrica utilizada (Coste, precisión). La precisión del conjunto de datos de validación en el modelo con ADAM para el conjunto de datos de CIFAR-10 indica que se produce un sobreajuste tras la época 17 mientras que el coste del mismo conjunto de datos indica que se produce tras la época 14.

Los hiperparámetros utilizados pueden alterar la obtención de un modelo eficaz y dependen del algoritmo de optimización a utilizar aunque tengan un mismo significado.Por ejemplo, en el caso de utilizar el SGD, la elección de una tasa de aprendizaje demasiado baja puede llevar a un entrenamiento demasiado largo y poco productivo. Sin embargo, el mismo valor de tasa de aprendizaje con ADAM, da resultados exitosos.

Estas observaciones dejan ver que el reconocimiento de imágenes es complejo y depende de una gran cantidad de parámetros y decisiones a la hora de crear un modelo óptimo para la función que le queremos asignar (Clasificador). En muchas ocasiones, encontrar el modelo óptimo, require realizar pruebas y modificaciones de manera experimental.

### 4.7.1. Líneas futuras

Con el objetivo cumplido de servir de iniciación al reconocimiento de imágenes basado en deep learning y la introducción a uno de los algoritmos más utilizados y eficientes actualmente en este campo (ADAM), este trabajo se puede extender siguiendo varios caminos relacionados en busca de un modelo resultante mejor:

- Influencia de los hiperparámetros y la elección de datos de entrada al rendimiento del modelo. De los experimentos se ha podido observar que pequeños cambios en los hiperparámetros, el uso de redes convolucionales o el cambio de los datos de entrada aumentando el número de muestra por lote tiene un impacto en el entrenamiento del modelo y la posterior obtención de una clasificador eficaz.

  Experimentar con las consecuencias de variar estos parámetros sobre conjuntos de datos complejos y la obtención de estos datos para que puedan ser usados de forma efectiva es interesante.

- Regularización. Trata las técnicas para impedir el sobreajuste de los modelos de inteligencia artificial a los datos de entrenamiento. En este trabajo se ha introducido e implementado una de las técnicas de regularización llamada early stopping por validación.

  Otra técnica similar es, por ejemplo, la validación cruzada. Esta técnica utiliza múltiples particiones del conjunto de datos en vez de utilizar conjuntos fijos para el entrenamiento y la validación.

- Arquitectura de la red neuronal. Uso de otras topologías de redes neuronales o cómo la distribución de neuronas afecta a la obtención del modelo final. Por ejemplo, la investigación de redes neuronales siamesas[14] es interesante y puede derivar en mejoras de vanguardia.

- Nuevos algoritmos para la obtención de parámetros óptimos $p*$. Los algoritmos genéticos, por ejemplo, no trata la minimización de una función de coste. A través de una función de actitud se evalúan diferentes modelos, para elegir los que mejor desempeño tengan. Con estos modelos, se realizan mutaciones que crean modelos hijos. Con estos hijos, se realiza el mismo procedimiento.

[14] Dos redes neuronales simétricas, con los mismos pesos y arquitectura, que al final combinan y usan la función de energía

Realizar este trabajo ha requerido tomar decisiones que caracterizan el desarrollo de proyectos de reconocimiento de imágenes. Debido a la gran cantidad de combinaciones posibles para construir una red neuronal se ha intentado centrar o asilar dichas características de manera que su entendimiento sea sencillo y la comparación de algoritmos de optimización sea fiel.

El reconocimiento de imágenes es una campo que crecerá en los próximos años y se extenderá su aplicación gracias a la digitalización de los procesos y la disponibilidad de la gran cantidad de datos con la que se trabaja hoy en día. Aprender los conocimientos básicos para poder entender los nuevos avances se hace crítico para un ingeniero informático.

# Bibliografía


[1] Aaron C. Ian G. Yoshua B. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[2] Desmond J. H. Catherine F. H. "Deep Learning: An Introduction for Applied Mathematicians". En: *SIAM review* 64.4 (2019), págs. 860-891.

[3] Instituto Tecnológico de Massachusetts. *Artificial Intelligence*. Massachusetts, 2020. URL: https://www.technologyreview.com/artificial-intelligence/.

[4] Iberdrola S.A. *¿Qué es la inteligencia artificial?* Bilbao, 2020. URL: https://www.iberdrola.com/innovacion/que-es-inteligencia-artificial.

[5] Alan M. T. "Computing machine and intelligence". En: *Mind* 49 (oct. de 1950), págs. 433-460.

[6] Tom M. M. *Machine Learning*. McGraw-Hill, mar. de 1997, págs. 2-3.

[7] Redacción Asociación para el Progreso de la Dirección. "¿Cuáles son los tipos de algoritmos del machine learning?" En: (abr. de 2019). URL: https://www.apd.es/algoritmos-del-machine-learning/.

[8] Josh. "Everything You Need to Know About Artificial Neural Networks". En: *Medium* (dic. de 2015). URL: https://medium.com/technology-invention-and-more/everything-you-need-to-know-about-artificial-neural-networks-57fac18245a1.

[9] unProfesor. *Qué es el potencial de acción*. Youtube, 2015. URL: https://www.youtube.com/watch?v=YzOncQ9KXZk.

[10] Juan Miguel M. D. "Introducción a las redes neuronales aplicadas". En: ((s.f)). URL: http://halweb.uc3m.es/esp/Personal/personas/jmmarin/esp/DM/tema3dm.pdf.

[11] DotCSV. *¿Qué es una Red Neuronal? Parte 1 : La Neurona | DotCSV*. Youtube. Mar. de 2015. URL: https://www.youtube.com/watch?v=MRIv2IwFTPg.

[12] Sudharsan R. *Hands-On Deep Learning Algorithms with Python: Master deep learning algorithms with extensive math by implementing them using TensorFlow*. Packt Publishing Ltd, jul. de 2019, págs. 7-10.

[13] Dong Y. Li D. *Deep Learning: Methods and Applications*. Vol. 7. 3–4. Foundations y Trends in Signal Processing, 2014, pág. 200.

[14] José Miguel F. F. Raquel F. L.. *Las Redes Neuronales Artificiales*. Metodología y Análisis de Datos en Ciencias Sociales. Netbiblo, 2008, págs. 38-42.

[15] Geoffrey E. H. Alex K. Ilya S. *ImageNet Classification with Deep Convolutional Neural Networks*. Ene. de 2012. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[16] Quan Z. "Convolutional Neural Networks". En: *3rd International Conference on Electromechanical Control Technology and Transportation - Volume 1: ICECTT,* INSTICC. SciTePress, 2018, págs. 434-439.

[17] Sebastian R. "An overview of gradient descent optimization algorithms". En: *Computing Research Repository* abs/1609.04747 (2016). URL: http://arxiv.org/abs/1609.04747.

[18] Yann N. D. et al. “Identifying and attacking the saddle point problem in high-dimensional nonconvex optimization”. En: *Computing Research Repository* abs/1406.2572 (2014). URL: http://arxiv.org/abs/1406.2572.

[19] Alex Smola. *L26/1 Momentum, Adagrad, RMSProp, Adam*. Youtube, mayo de 2019. URL: https://www.youtube.com/watch?v=gmwxUy7NYpA.

[20] PyTorch. *Tutoriales de Pytorch*. URL: https://pytorch.org/tutorials/.

[21] Simon H. et al. *Neural networks and learning machines/Simon Haykin.* New York: Prentice Hall, 2009.

[22] Alexander K. et al. “Big transfer (BiT): General visual representation learning”. En: *arXiv preprint arXiv:1912.11370* (2019). URL: https://arxiv.org/pdf/1912.11370v3.pdf.

[23] Till T. *The TikZ and PGF Packages*. Institut für Theoretische Informatik, Universität zu Lübeck, 2007. URL: https://www.bu.edu/math/files/2013/08/tikzpgfmanual.pdf.

# Parte V

# Apéndice

# Apéndice A

# Ejemplo de red neuronal sencilla

# Red neuronal sencilla

## 1 Starting article with Pytorch low level

[0]:
```
# Visualización de datos
import matplotlib.pyplot as plt
import numpy as np

# Selección estocástica de dato de entrenamiento
from random import randrange

# Función exponencial
import math

import torch
import datetime
import os

#Logger
from torch.utils.tensorboard import SummaryWriter
```

[0]:
```
class cronometro:
  def __init__(self):
    self.time_passed = None
    self.total_time = None
    self.lapses = []
    pass

  def start(self):
    if self.time_passed is None:
      self.start_time = datetime.datetime.now()

  def stop(self):
    try:
      time_passed = datetime.datetime.now() - self.start_time
      if self.total_time is None:
        self.total_time = time_passed
      else:
        self.total_time += time_passed
```

```
      self.lapses.append(time_passed)
    except:
      print("You did not start the crhonometer")

  def get_time(self, delta_time, verbose=True):
    minutos = delta_time.total_seconds() // 60
    segundos = delta_time.total_seconds() - 60 * minutos
    if verbose :
      print(f"Tiempo transcurrido: {minutos} minutos y {segundos:2.4} segundos.")
    return (minutos, segundos)
```

## 1.1 Datos de entrenamiento y test

### 1.1.1 Datos de entrenamiento

[0]:
```
# x = np.random.rand(100,2)
datos_ent = torch.tensor([[0.1, 0.1], [0.3, 0.4], [0.1, 0.5], [0.6, 0.9], [0.4,␣
 ↪0.2], [0.6, 0.3] , [0.5, 0.6], [0.9, 0.2], [0.4, 0.4], [0.7, 0.6]],␣
 ↪dtype=torch.float, requires_grad=False) # 10 datos
labels = ("A", "B")

# Crear trainset
data = []
for i in range(5):
    data.append((datos_ent[i] ,labels[0]))
for i in range(5,datos_ent.size()[0]):
    data.append((datos_ent[i], labels[1]))

cost_label = lambda label : torch.tensor([1,0],  dtype=torch.float,␣
 ↪requires_grad=False) if label == "A" else torch.tensor([0,1],  dtype=torch.
 ↪float, requires_grad=False)

cost_labels = torch.tensor([[1,0], [1,0], [1,0], [1,0], [1,0], [0,1], [0,1],␣
 ↪[0,1], [0,1], [0,1]],  dtype=torch.float)

log_interval = 100
```

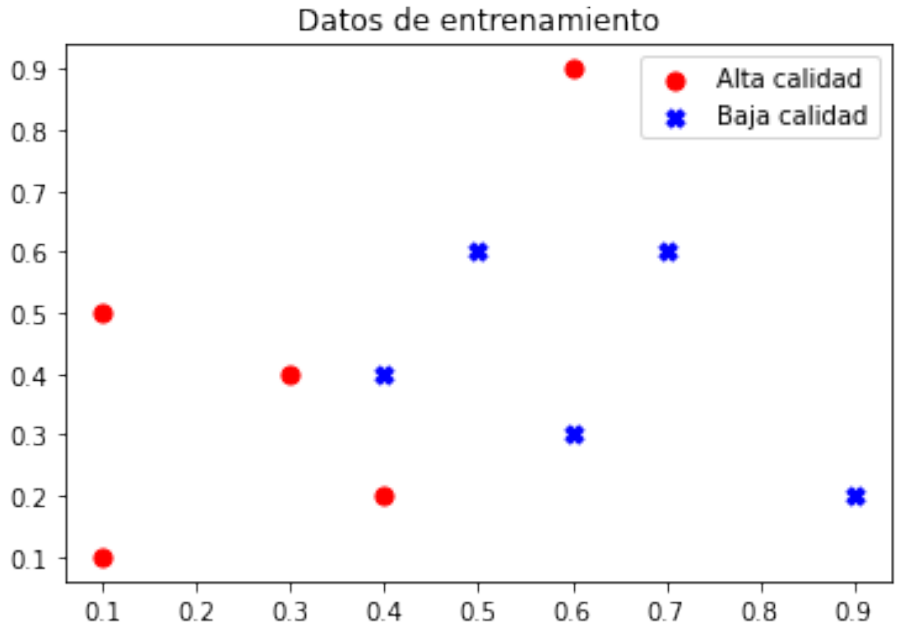
### 1.1.2 Conjunto de datos de prueba

[0]:
```
# Testset

testSetLabels = ["A", "A", "B", "B", "B"]

testSet = torch.tensor([[0.3,0.3], [0.8,1],[0.5,0.3], [0.3,0.6], [0.5,0.5]],␣
 ↪dtype=torch.float, requires_grad=False)
```

### 1.1.3 Visualización de los datos de entrenamiento

Crear la figura para representar los datos (un scatter plot)

[23]:
```
# Train Set
plt.scatter(datos_ent[:5,0],datos_ent[:5,1], c="red", label='Alta calidad',␣
 ↪marker="o", s=50)
plt.scatter(datos_ent[5:datos_ent.size()[0],0],datos_ent[5:datos_ent.
 ↪size()[0],1], c="blue", label='Baja calidad', marker="X", s=50)

#plt.scatter(testSet[:2,0],testSet[:2,1], c="yellow", label=labels[0] + " test",␣
 ↪marker="o", s=50)
#plt.scatter(testSet[2:5,0],testSet[2:5,1], c="green", label=labels[1] + "␣
 ↪test", marker="X", s=50)

plt.title("Datos de entrenamiento")
plt.legend()
plt.show()
```




## 1.2 Funciones base, activación y coste

[0]:
```
def activation (entrada):
  return 1 / (1 + torch.exp(-entrada))
```

```
def linear_fc(weight, bias, entrada):
  return torch.sum(entrada * weight, dim=1) + bias

def output_net(entrada):
  return activation(linear_fc(w4, b4, activation(linear_fc(w3, b3,␣
 ↪activation(linear_fc(w2,b2,entrada))))))

# Función de coste
def coste_dato(output, label):
  cost = (label[0] - output[0]) ** 2  + (label[1] - output[1]) ** 2
  return cost

def coste_total():
  costes = [coste_dato(output_net(x),y) for (x,y) in zip(datos_ent, cost_labels)]
  return (torch.sum(torch.stack(costes)) / len(costes)).item()

# Hyperparámetros

#Tasa de aprendizaje
eta = 0.05

#Número de iteraciones
niter = 10 ** 6
```

**##Entrenamiento gradiente descendente estocástico**

###Inicialización de parámetros e hyperparámetros

[0]:
```
#Semilla para la reproducibilidad de resultados
torch.manual_seed(0)

# Inicialización de variables
w2 = torch.rand(2, 2, dtype=torch.float, requires_grad=False) * 0.5
w3 = torch.rand(3, 2, dtype=torch.float, requires_grad=False) * 0.5
w4 = torch.rand(2, 3, dtype=torch.float, requires_grad=False) * 0.5
b2 = torch.rand(2, dtype=torch.float, requires_grad=False) * 0.5
b3 = torch.rand(3, dtype=torch.float, requires_grad=False) * 0.5
b4 = torch.rand(2, dtype=torch.float, requires_grad=False) * 0.5

save_cost_sgd = []
```

[26]:
```
# Mostrar valores
print("Valor de w2:")
print(w2)
print()

print("Valor de w3:")
print(w3)
```

```
print()

print("Valor de w4:")
print(w4)
print()

print("Valor de b2:")
print(b2)
print()

print("Valor de b3:")
print(b3)
print()

print("Valor de b4:")
print(b4)
print()
```

```
Valor de w2:
tensor([[0.2481, 0.3841],
        [0.0442, 0.0660]])

Valor de w3:
tensor([[0.1537, 0.3170],
        [0.2450, 0.4482],
        [0.2278, 0.3162]])

Valor de w4:
tensor([[0.1744, 0.2009, 0.0112],
        [0.0844, 0.1469, 0.2593]])

Valor de b2:
tensor([0.3488, 0.4000])

Valor de b3:
tensor([0.0805, 0.1411, 0.3408])

Valor de b4:
tensor([0.4576, 0.1985])
```

###Rutina de entrenamiento

[27]:
```
cron_gde = cronometro()
cron_gde.start()

for iteracion in range(niter):
  # extraer los inputs; data es una lista de [input, label]
```

```
  inputData, label = data[randrange(datos_ent.shape[0])]
  #inputData, label = data[i]

  # Forward pass
  a2 = activation(linear_fc(w2, b2,inputData))
  a3 = activation(linear_fc(w3, b3, a2))
  a4 = activation(linear_fc(w4, b4, a3))

  # Backward pass
  delta4 = a4 * (1-a4) * (a4 - cost_label(label))
  delta3 = a3 * (1- a3) * (w4.t() @ delta4)
  delta2 = a2 * (1- a2) * (w3.t() @ delta3)

  # Gradient step

  w2 = w2 - eta * (delta2.repeat(2,1).t() * inputData)
  w3 = w3 - eta * (delta3.repeat(2,1).t() * a2)
  w4 = w4 - eta * (delta4.repeat(3,1).t() * a3)

  b2 = torch.clone(delta2)
  b3 = torch.clone(delta3)
  b4 = torch.clone(delta4)

  if iteracion % log_interval == 0:
    coste_tot = coste_total()
    save_cost_sgd.append(coste_tot)
    if iteracion % (log_interval * 100) == 0:
      print("Iteración: [{}/{}] {:1.2f} % --> Coste: {:1.4f}".format(iteracion +␣
↪1, niter, iteracion/niter * 100, coste_tot))

cron_gde.stop()
cron_gde.get_time(cron_gde.total_time)
```

```
Iteración: [1/1000000] 0.00 % --> Coste: 0.5105
Iteración: [10001/1000000] 1.00 % --> Coste: 0.5017
Iteración: [20001/1000000] 2.00 % --> Coste: 0.5002
Iteración: [30001/1000000] 3.00 % --> Coste: 0.4999
Iteración: [40001/1000000] 4.00 % --> Coste: 0.5029
Iteración: [50001/1000000] 5.00 % --> Coste: 0.5047
Iteración: [60001/1000000] 6.00 % --> Coste: 0.5024
Iteración: [70001/1000000] 7.00 % --> Coste: 0.5000
Iteración: [80001/1000000] 8.00 % --> Coste: 0.4953
Iteración: [90001/1000000] 9.00 % --> Coste: 0.4602
Iteración: [100001/1000000] 10.00 % --> Coste: 0.2950
Iteración: [110001/1000000] 11.00 % --> Coste: 0.3815
Iteración: [120001/1000000] 12.00 % --> Coste: 0.3018
```

```
Iteración: [130001/1000000] 13.00 % --> Coste: 0.2453
Iteración: [140001/1000000] 14.00 % --> Coste: 0.2325
Iteración: [150001/1000000] 15.00 % --> Coste: 0.2589
Iteración: [160001/1000000] 16.00 % --> Coste: 0.1891
Iteración: [170001/1000000] 17.00 % --> Coste: 0.1934
Iteración: [180001/1000000] 18.00 % --> Coste: 0.3304
Iteración: [190001/1000000] 19.00 % --> Coste: 0.0865
Iteración: [200001/1000000] 20.00 % --> Coste: 0.0639
Iteración: [210001/1000000] 21.00 % --> Coste: 0.0470
Iteración: [220001/1000000] 22.00 % --> Coste: 0.4075
Iteración: [230001/1000000] 23.00 % --> Coste: 0.0248
Iteración: [240001/1000000] 24.00 % --> Coste: 0.0258
Iteración: [250001/1000000] 25.00 % --> Coste: 0.0224
Iteración: [260001/1000000] 26.00 % --> Coste: 0.0122
Iteración: [270001/1000000] 27.00 % --> Coste: 0.0098
Iteración: [280001/1000000] 28.00 % --> Coste: 0.0083
Iteración: [290001/1000000] 29.00 % --> Coste: 0.0071
Iteración: [300001/1000000] 30.00 % --> Coste: 0.0063
Iteración: [310001/1000000] 31.00 % --> Coste: 0.0051
Iteración: [320001/1000000] 32.00 % --> Coste: 0.0046
Iteración: [330001/1000000] 33.00 % --> Coste: 0.0042
Iteración: [340001/1000000] 34.00 % --> Coste: 0.0037
Iteración: [350001/1000000] 35.00 % --> Coste: 0.0032
Iteración: [360001/1000000] 36.00 % --> Coste: 0.0030
Iteración: [370001/1000000] 37.00 % --> Coste: 0.0027
Iteración: [380001/1000000] 38.00 % --> Coste: 0.0081
Iteración: [390001/1000000] 39.00 % --> Coste: 0.0023
Iteración: [400001/1000000] 40.00 % --> Coste: 0.0023
Iteración: [410001/1000000] 41.00 % --> Coste: 0.0021
Iteración: [420001/1000000] 42.00 % --> Coste: 0.0020
Iteración: [430001/1000000] 43.00 % --> Coste: 0.0018
Iteración: [440001/1000000] 44.00 % --> Coste: 0.0020
Iteración: [450001/1000000] 45.00 % --> Coste: 0.0017
Iteración: [460001/1000000] 46.00 % --> Coste: 0.0015
Iteración: [470001/1000000] 47.00 % --> Coste: 0.0015
Iteración: [480001/1000000] 48.00 % --> Coste: 0.0014
Iteración: [490001/1000000] 49.00 % --> Coste: 0.0014
Iteración: [500001/1000000] 50.00 % --> Coste: 0.0013
Iteración: [510001/1000000] 51.00 % --> Coste: 0.0014
Iteración: [520001/1000000] 52.00 % --> Coste: 0.0011
Iteración: [530001/1000000] 53.00 % --> Coste: 0.0011
Iteración: [540001/1000000] 54.00 % --> Coste: 0.0011
Iteración: [550001/1000000] 55.00 % --> Coste: 0.0011
Iteración: [560001/1000000] 56.00 % --> Coste: 0.0010
Iteración: [570001/1000000] 57.00 % --> Coste: 0.0010
Iteración: [580001/1000000] 58.00 % --> Coste: 0.0009
Iteración: [590001/1000000] 59.00 % --> Coste: 0.0009
Iteración: [600001/1000000] 60.00 % --> Coste: 0.0009
```

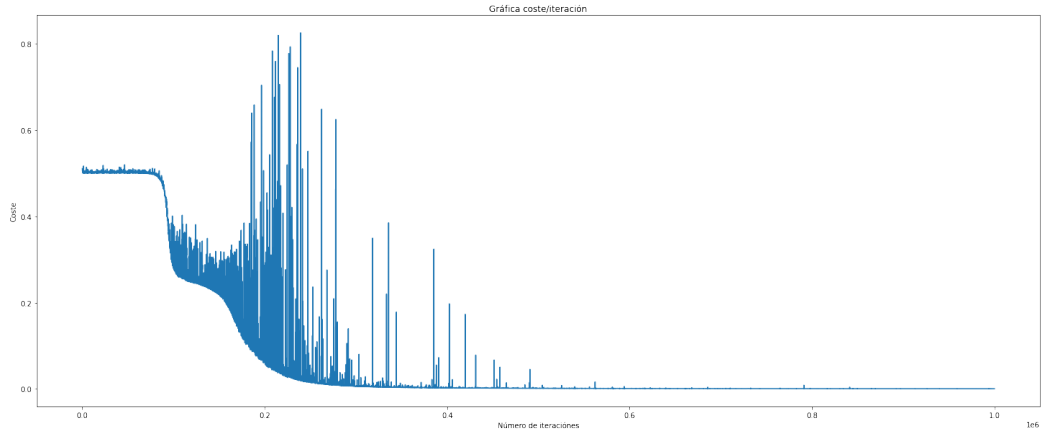
```
Iteración: [610001/1000000] 61.00 % --> Coste: 0.0009
Iteración: [620001/1000000] 62.00 % --> Coste: 0.0009
Iteración: [630001/1000000] 63.00 % --> Coste: 0.0008
Iteración: [640001/1000000] 64.00 % --> Coste: 0.0008
Iteración: [650001/1000000] 65.00 % --> Coste: 0.0008
Iteración: [660001/1000000] 66.00 % --> Coste: 0.0007
Iteración: [670001/1000000] 67.00 % --> Coste: 0.0007
Iteración: [680001/1000000] 68.00 % --> Coste: 0.0007
Iteración: [690001/1000000] 69.00 % --> Coste: 0.0007
Iteración: [700001/1000000] 70.00 % --> Coste: 0.0007
Iteración: [710001/1000000] 71.00 % --> Coste: 0.0006
Iteración: [720001/1000000] 72.00 % --> Coste: 0.0006
Iteración: [730001/1000000] 73.00 % --> Coste: 0.0006
Iteración: [740001/1000000] 74.00 % --> Coste: 0.0006
Iteración: [750001/1000000] 75.00 % --> Coste: 0.0006
Iteración: [760001/1000000] 76.00 % --> Coste: 0.0006
Iteración: [770001/1000000] 77.00 % --> Coste: 0.0006
Iteración: [780001/1000000] 78.00 % --> Coste: 0.0006
Iteración: [790001/1000000] 79.00 % --> Coste: 0.0005
Iteración: [800001/1000000] 80.00 % --> Coste: 0.0005
Iteración: [810001/1000000] 81.00 % --> Coste: 0.0005
Iteración: [820001/1000000] 82.00 % --> Coste: 0.0008
Iteración: [830001/1000000] 83.00 % --> Coste: 0.0005
Iteración: [840001/1000000] 84.00 % --> Coste: 0.0005
Iteración: [850001/1000000] 85.00 % --> Coste: 0.0005
Iteración: [860001/1000000] 86.00 % --> Coste: 0.0005
Iteración: [870001/1000000] 87.00 % --> Coste: 0.0006
Iteración: [880001/1000000] 88.00 % --> Coste: 0.0005
Iteración: [890001/1000000] 89.00 % --> Coste: 0.0005
Iteración: [900001/1000000] 90.00 % --> Coste: 0.0004
Iteración: [910001/1000000] 91.00 % --> Coste: 0.0004
Iteración: [920001/1000000] 92.00 % --> Coste: 0.0005
Iteración: [930001/1000000] 93.00 % --> Coste: 0.0004
Iteración: [940001/1000000] 94.00 % --> Coste: 0.0005
Iteración: [950001/1000000] 95.00 % --> Coste: 0.0004
Iteración: [960001/1000000] 96.00 % --> Coste: 0.0004
Iteración: [970001/1000000] 97.00 % --> Coste: 0.0004
Iteración: [980001/1000000] 98.00 % --> Coste: 0.0004
Iteración: [990001/1000000] 99.00 % --> Coste: 0.0004
Tiempo transcurrido: 4.0 minutos y 54.0 segundos.
```

[27]:
```
(4.0, 53.997322)
```

###Gráfico del valor de la función de coste por iteración

[28]:
```
fig = plt.figure(figsize=(25, 10))
plt.plot([x * log_interval for x in range(len(save_cost_sgd))], save_cost_sgd)
plt.title("Gráfica coste/iteración")
```

```
plt.ylabel("Coste")
plt.xlabel("Número de iteraciónes")

fig.show()
```



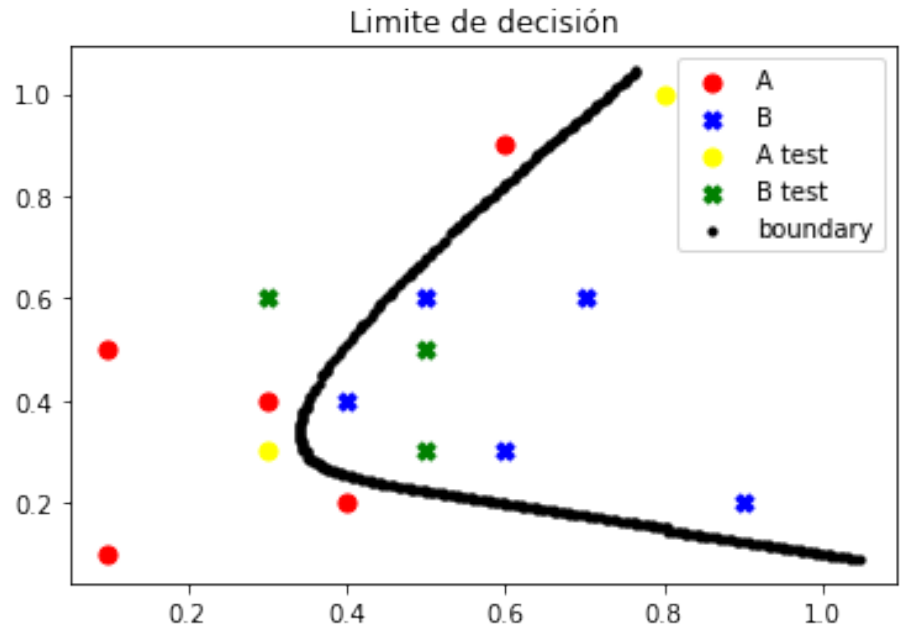

###Evaluación del modelo

####Límite de decisión

[29]:
```
boundary = []
with torch.no_grad():
  for a in np.arange(0.0,01.05,0.005):
    for b in np.arange(0.0,01.05,0.005):
      out = output_net(torch.tensor([a,b], dtype=torch.float))
      if(abs(out[0]-out[1]) <= 0.1):
        boundary.append([a, b])

limit = torch.tensor(boundary)

plt.clf();
fig2 = plt.figure()
plt.scatter(datos_ent[:5,0],datos_ent[:5,1], c="red", label=labels[0],␣
 ↪marker="o", s=50)
plt.scatter(datos_ent[5:datos_ent.size()[0],0],datos_ent[5:datos_ent.
 ↪size()[0],1], c="blue", label=labels[1], marker="X", s=50)
plt.scatter(testSet[:2,0],testSet[:2,1], c="yellow", label=labels[0] + " test",␣
 ↪marker="o", s=50)
plt.scatter(testSet[2:5,0],testSet[2:5,1], c="green", label=labels[1] + " test",␣
 ↪marker="X", s=50)

plt.scatter(limit[:,0],limit[:,1], c="black", label="boundary", marker="o", s=10)
```

```
plt.title("Limite de decisión")
plt.legend()
fig2.show()
```

```
<Figure size 432x288 with 0 Axes>
```




####Precisión del modelo

[30]:
```
predFc = lambda output : "A" if output[0] > output[1] else "B"
numCorrect = 0
with torch.no_grad():
  for i in range(5):
    out = predFc(output_net(testSet[i]))
    if (out == testSetLabels[i]):
      numCorrect += 1
    else:
      print(f"Fallo al clasificar el punto: {i} -->")
      print(testSet[i])

print(f"La precisión de la red neuronal es: {(numCorrect / 5 * 100):.2f}%")
```

```
Fallo al clasificar el punto: 1 -->
tensor([0.8000, 1.0000])
Fallo al clasificar el punto: 3 -->
tensor([0.3000, 0.6000])
```

```
La precisión de la red neuronal es: 60.00%
```

**##Entrenamiento gradiente descendente**

###Inicialización de parámetros e hyperparámetros

[0]:
```
#Semilla para la reproducibilidad de resultados
torch.manual_seed(0)

# Inicialización de variables

w2 = torch.rand(2, 2, dtype=torch.float, requires_grad=False) * 0.5
w3 = torch.rand(3, 2, dtype=torch.float, requires_grad=False) * 0.5
w4 = torch.rand(2, 3, dtype=torch.float, requires_grad=False) * 0.5
b2 = torch.rand(2, dtype=torch.float, requires_grad=False) * 0.5
b3 = torch.rand(3, dtype=torch.float, requires_grad=False) * 0.5
b4 = torch.rand(2, dtype=torch.float, requires_grad=False) * 0.5

save_cost_gd = []
```

[35]:
```
# Mostrar valores
print("Valor de w2:")
print(w2)
print()

print("Valor de w3:")
print(w3)
print()

print("Valor de w4:")
print(w4)
print()

print("Valor de b2:")
print(b2)
print()

print("Valor de b3:")
print(b3)
print()

print("Valor de b4:")
print(b4)
print()
```

```
Valor de w2:
tensor([[0.2481, 0.3841],
        [0.0442, 0.0660]])
```

```
Valor de w3:
tensor([[0.1537, 0.3170],
        [0.2450, 0.4482],
        [0.2278, 0.3162]])

Valor de w4:
tensor([[0.1744, 0.2009, 0.0112],
        [0.0844, 0.1469, 0.2593]])

Valor de b2:
tensor([0.3488, 0.4000])

Valor de b3:
tensor([0.0805, 0.1411, 0.3408])

Valor de b4:
tensor([0.4576, 0.1985])
```

###Rutina de entrenamiento

[36]:
```
cron_gd = cronometro()
cron_gd.start()

for iteracion in range(niter):
  # extraer los inputs; data es una lista de [input, label]

  ws2 = torch.zeros((10,2,2))
  ws3 = torch.zeros((10,3,2))
  ws4 = torch.zeros((10,2,3))
  bs2 = torch.zeros((10,2))
  bs3 = torch.zeros((10,3))
  bs4 = torch.zeros((10,2))

  for i in range(10):
    #inputData, label = data[randrange(x.shape[0])]
    inputData, label = data[i]

    # Forward pass
    a2 = activation(linear_fc(w2, b2,inputData))
    a3 = activation(linear_fc(w3, b3, a2))
    a4 = activation(linear_fc(w4, b4, a3))

    # Backward pass
    #Derivadas locales
    delta4 = a4 * (1-a4) * (a4 - cost_label(label))
    delta3 = a3 * (1- a3) * (w4.t() @ delta4)
    delta2 = a2 * (1- a2) * (w3.t() @ delta3)
```

```
    # Derivadas parciales de los parámetros
    ws2[i] = delta2.repeat(2,1).t() * inputData
    ws3[i] = delta3.repeat(2,1).t() * a2
    ws4[i] = delta4.repeat(3,1).t() * a3
    bs2[i] = torch.clone(delta2)
    bs3[i] = torch.clone(delta3)
    bs4[i] = torch.clone(delta4)

  # Gradient step
  w2 = w2 - eta * torch.tensor([[torch.mean(ws2[:,0,0]) , torch.mean(ws2[:
↪,0,1])],
                                [torch.mean(ws2[:,1,0]) ,torch.mean(ws2[:
↪,1,1])]])

  w3 = w3 - eta * torch.tensor([[torch.mean(ws3[:,0,0]) , torch.mean(ws3[:
↪,0,1])],
                                [torch.mean(ws3[:,1,0]) , torch.mean(ws3[:
↪,1,1])],
                                [torch.mean(ws3[:,2,0]) , torch.mean(ws3[:
↪,2,1])]])

  w4 = w4 - eta * torch.tensor([[torch.mean(ws4[:,0,0]) , torch.mean(ws4[:
↪,0,1]),torch.mean(ws4[:,0,2])],
                                [torch.mean(ws4[:,1,0]) , torch.mean(ws4[:
↪,1,1]),torch.mean(ws4[:,1,2])]])

  b2 = torch.tensor([torch.mean(bs2[:,0]), torch.mean(bs2[:,1])])
  b3 = torch.tensor([torch.mean(bs3[:,0]), torch.mean(bs3[:,1]), torch.mean(bs3[:
↪,2])])
  b4 = torch.tensor([torch.mean(bs4[:,0]), torch.mean(bs4[:,1])])


  if iteracion % log_interval == 0:
    coste_tot = coste_total()
    save_cost_gd.append(coste_tot)
    if iteracion % (log_interval * 100) == 0:
      print("Iteración: [{}/{}] {:1.2f} % --> Coste: {:1.4f}".format(iteracion +␣
↪1, niter, iteracion/niter * 100, coste_tot))

cron_gd.stop()
cron_gd.get_time(cron_gd.total_time)
```

```
Iteración: [1/1000000] 0.00 % --> Coste: 0.5101
Iteración: [10001/1000000] 1.00 % --> Coste: 0.5000
Iteración: [20001/1000000] 2.00 % --> Coste: 0.5000
Iteración: [30001/1000000] 3.00 % --> Coste: 0.5000
```

```
Iteración: [40001/1000000] 4.00 % --> Coste: 0.4999
Iteración: [50001/1000000] 5.00 % --> Coste: 0.4998
Iteración: [60001/1000000] 6.00 % --> Coste: 0.4995
Iteración: [70001/1000000] 7.00 % --> Coste: 0.4979
Iteración: [80001/1000000] 8.00 % --> Coste: 0.4811
Iteración: [90001/1000000] 9.00 % --> Coste: 0.3018
Iteración: [100001/1000000] 10.00 % --> Coste: 0.2544
Iteración: [110001/1000000] 11.00 % --> Coste: 0.2474
Iteración: [120001/1000000] 12.00 % --> Coste: 0.2348
Iteración: [130001/1000000] 13.00 % --> Coste: 0.2186
Iteración: [140001/1000000] 14.00 % --> Coste: 0.2074
Iteración: [150001/1000000] 15.00 % --> Coste: 0.2002
Iteración: [160001/1000000] 16.00 % --> Coste: 0.1946
Iteración: [170001/1000000] 17.00 % --> Coste: 0.1901
Iteración: [180001/1000000] 18.00 % --> Coste: 0.1866
Iteración: [190001/1000000] 19.00 % --> Coste: 0.1837
Iteración: [200001/1000000] 20.00 % --> Coste: 0.1815
Iteración: [210001/1000000] 21.00 % --> Coste: 0.1796
Iteración: [220001/1000000] 22.00 % --> Coste: 0.1781
Iteración: [230001/1000000] 23.00 % --> Coste: 0.1768
Iteración: [240001/1000000] 24.00 % --> Coste: 0.1758
Iteración: [250001/1000000] 25.00 % --> Coste: 0.1749
Iteración: [260001/1000000] 26.00 % --> Coste: 0.1741
Iteración: [270001/1000000] 27.00 % --> Coste: 0.1734
Iteración: [280001/1000000] 28.00 % --> Coste: 0.1727
Iteración: [290001/1000000] 29.00 % --> Coste: 0.1722
Iteración: [300001/1000000] 30.00 % --> Coste: 0.1716
Iteración: [310001/1000000] 31.00 % --> Coste: 0.1711
Iteración: [320001/1000000] 32.00 % --> Coste: 0.1706
Iteración: [330001/1000000] 33.00 % --> Coste: 0.1701
Iteración: [340001/1000000] 34.00 % --> Coste: 0.1696
Iteración: [350001/1000000] 35.00 % --> Coste: 0.1691
Iteración: [360001/1000000] 36.00 % --> Coste: 0.1685
Iteración: [370001/1000000] 37.00 % --> Coste: 0.1679
Iteración: [380001/1000000] 38.00 % --> Coste: 0.1673
Iteración: [390001/1000000] 39.00 % --> Coste: 0.1667
Iteración: [400001/1000000] 40.00 % --> Coste: 0.1661
Iteración: [410001/1000000] 41.00 % --> Coste: 0.1655
Iteración: [420001/1000000] 42.00 % --> Coste: 0.1649
Iteración: [430001/1000000] 43.00 % --> Coste: 0.1643
Iteración: [440001/1000000] 44.00 % --> Coste: 0.1638
Iteración: [450001/1000000] 45.00 % --> Coste: 0.1633
Iteración: [460001/1000000] 46.00 % --> Coste: 0.1629
Iteración: [470001/1000000] 47.00 % --> Coste: 0.1625
Iteración: [480001/1000000] 48.00 % --> Coste: 0.1622
Iteración: [490001/1000000] 49.00 % --> Coste: 0.1619
Iteración: [500001/1000000] 50.00 % --> Coste: 0.1616
Iteración: [510001/1000000] 51.00 % --> Coste: 0.1613
```

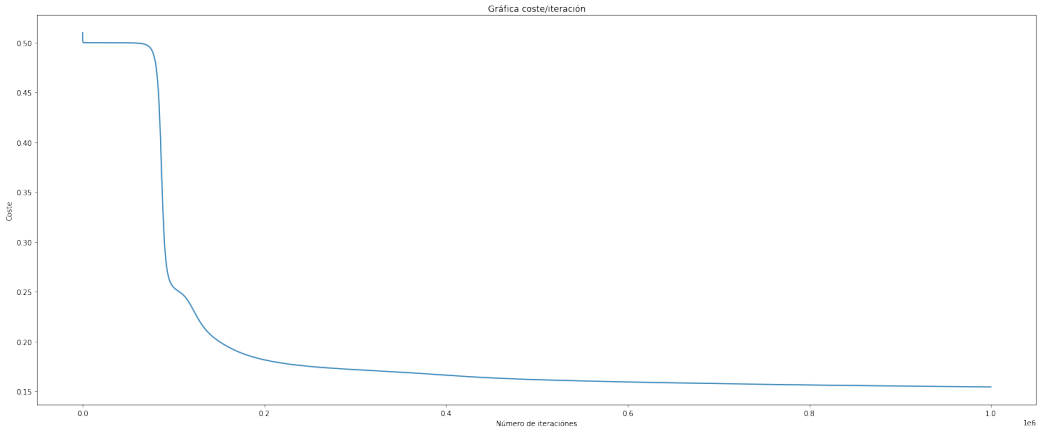
```
Iteración: [520001/1000000] 52.00 % --> Coste: 0.1610
Iteración: [530001/1000000] 53.00 % --> Coste: 0.1607
Iteración: [540001/1000000] 54.00 % --> Coste: 0.1605
Iteración: [550001/1000000] 55.00 % --> Coste: 0.1603
Iteración: [560001/1000000] 56.00 % --> Coste: 0.1600
Iteración: [570001/1000000] 57.00 % --> Coste: 0.1598
Iteración: [580001/1000000] 58.00 % --> Coste: 0.1596
Iteración: [590001/1000000] 59.00 % --> Coste: 0.1594
Iteración: [600001/1000000] 60.00 % --> Coste: 0.1592
Iteración: [610001/1000000] 61.00 % --> Coste: 0.1591
Iteración: [620001/1000000] 62.00 % --> Coste: 0.1589
Iteración: [630001/1000000] 63.00 % --> Coste: 0.1587
Iteración: [640001/1000000] 64.00 % --> Coste: 0.1585
Iteración: [650001/1000000] 65.00 % --> Coste: 0.1584
Iteración: [660001/1000000] 66.00 % --> Coste: 0.1582
Iteración: [670001/1000000] 67.00 % --> Coste: 0.1581
Iteración: [680001/1000000] 68.00 % --> Coste: 0.1579
Iteración: [690001/1000000] 69.00 % --> Coste: 0.1578
Iteración: [700001/1000000] 70.00 % --> Coste: 0.1576
Iteración: [710001/1000000] 71.00 % --> Coste: 0.1575
Iteración: [720001/1000000] 72.00 % --> Coste: 0.1573
Iteración: [730001/1000000] 73.00 % --> Coste: 0.1572
Iteración: [740001/1000000] 74.00 % --> Coste: 0.1570
Iteración: [750001/1000000] 75.00 % --> Coste: 0.1569
Iteración: [760001/1000000] 76.00 % --> Coste: 0.1567
Iteración: [770001/1000000] 77.00 % --> Coste: 0.1566
Iteración: [780001/1000000] 78.00 % --> Coste: 0.1565
Iteración: [790001/1000000] 79.00 % --> Coste: 0.1563
Iteración: [800001/1000000] 80.00 % --> Coste: 0.1562
Iteración: [810001/1000000] 81.00 % --> Coste: 0.1561
Iteración: [820001/1000000] 82.00 % --> Coste: 0.1560
Iteración: [830001/1000000] 83.00 % --> Coste: 0.1559
Iteración: [840001/1000000] 84.00 % --> Coste: 0.1557
Iteración: [850001/1000000] 85.00 % --> Coste: 0.1556
Iteración: [860001/1000000] 86.00 % --> Coste: 0.1555
Iteración: [870001/1000000] 87.00 % --> Coste: 0.1554
Iteración: [880001/1000000] 88.00 % --> Coste: 0.1553
Iteración: [890001/1000000] 89.00 % --> Coste: 0.1552
Iteración: [900001/1000000] 90.00 % --> Coste: 0.1551
Iteración: [910001/1000000] 91.00 % --> Coste: 0.1550
Iteración: [920001/1000000] 92.00 % --> Coste: 0.1549
Iteración: [930001/1000000] 93.00 % --> Coste: 0.1549
Iteración: [940001/1000000] 94.00 % --> Coste: 0.1548
Iteración: [950001/1000000] 95.00 % --> Coste: 0.1547
Iteración: [960001/1000000] 96.00 % --> Coste: 0.1546
Iteración: [970001/1000000] 97.00 % --> Coste: 0.1545
Iteración: [980001/1000000] 98.00 % --> Coste: 0.1544
Iteración: [990001/1000000] 99.00 % --> Coste: 0.1544
```

```
Tiempo transcurrido: 52.0 minutos y 36.49 segundos.
```

[36]:
```
(52.0, 36.487944999999854)
```

###Gráfico del valor de la función de coste por iteración

[37]:
```
fig = plt.figure(figsize=(25, 10))
plt.plot([x * log_interval for x in range(len(save_cost_gd))], save_cost_gd)
plt.title("Gráfica coste/iteración")
plt.ylabel("Coste")
plt.xlabel("Número de iteraciónes")
fig.show()
```



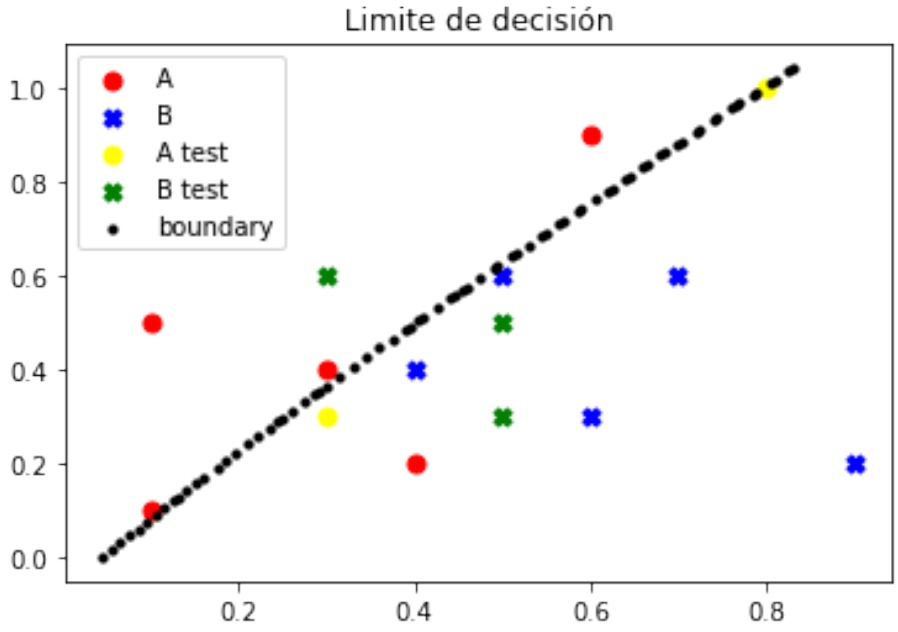

###Evaluación del modelo

**Límite de decisión**

[38]:
```
boundary = []
with torch.no_grad():
  for a in np.arange(0.0,01.05,0.005):
    for b in np.arange(0.0,01.05,0.005):
      out = output_net(torch.tensor([a,b], dtype=torch.float))
      if(abs(out[0]-out[1]) <= 0.1):
        boundary.append([a, b])

limit = torch.tensor(boundary)

plt.clf();
fig2 = plt.figure()
plt.scatter(datos_ent[:5,0],datos_ent[:5,1], c="red", label=labels[0],␣
↪marker="o", s=50)
```

```
plt.scatter(datos_ent[5:datos_ent.size()[0],0],datos_ent[5:datos_ent.
 ↪size()[0],1], c="blue", label=labels[1], marker="X", s=50)
plt.scatter(testSet[:2,0],testSet[:2,1], c="yellow", label=labels[0] + " test",␣
 ↪marker="o", s=50)
plt.scatter(testSet[2:5,0],testSet[2:5,1], c="green", label=labels[1] + " test",␣
 ↪marker="X", s=50)

plt.scatter(limit[:,0],limit[:,1], c="black", label="boundary", marker="o", s=10)

plt.title("Limite de decisión")
plt.legend()
fig2.show()
```

```
<Figure size 432x288 with 0 Axes>
```




####Precisión del modelo

[39]:
```
predFc = lambda output : "A" if output[0] > output[1] else "B"
numCorrect = 0
with torch.no_grad():
  for i in range(5):
    out = predFc(output_net(testSet[i]))
    if (out == testSetLabels[i]):
      numCorrect += 1
```

```
    else:
      print(f"Fallo al clasificar el punto: {i} -->")
      print(testSet[i])

print(f"La precisión de la red neuronal es: {(numCorrect / 5 * 100):.2f}%")
```

```
Fallo al clasificar el punto: 0 -->
tensor([0.3000, 0.3000])
Fallo al clasificar el punto: 1 -->
tensor([0.8000, 1.0000])
Fallo al clasificar el punto: 3 -->
tensor([0.3000, 0.6000])
La precisión de la red neuronal es: 40.00%
```

# Apéndice B

# Clasificador de imágenes MNIST

# Clasificador del conjunto de datos MNIST

## 1 Classificador de imágenes (MINST Database)

### 1.1 Librerías y clases necesarías

[1]:
```
import torch
import torchvision

import matplotlib.pyplot as plt
import os, datetime
from itertools import cycle

#Tensorboard
from torch.utils.tensorboard import SummaryWriter
import numpy as np
```

#### 1.1.1 Clases

[2]:
```
class EarlyStopping:
    """Parada temprara del entrenamiento si el coste de validación no mejora␣
 ↪tras {patience} iteraciones"""
    def __init__(self, ident, path, patience=7, verbose=True, delta=0):

        self.ident = ident
        self.path = path
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta
        self.stop_iter = 0

    def __call__(self, val_loss, model, i):

        score = -val_loss
```

```
        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta:
            self.counter += 1
            #print(f'EarlyStopping counter: {self.counter} de {self.patience}')
            if self.counter >= self.patience:
              if not self.early_stop:
                self.early_stop = True
                self.stop_iter = i - self.patience
                if self.verbose:
                  print(f'[{i}] Mejora estancada por {self.patience} iteraciones␣
↪({self.val_loss_min:.6f} --> {val_loss:.6f}).  Activando early stop ...')

        else:
          self.save_checkpoint(val_loss, model)
          self.best_score = score
          self.counter = 0

    def save_checkpoint(self, val_loss, model):
        '''Guarda el valor de los parámetros en checkpoint'''
        torch.save(model.state_dict(), os.path.join(self.path,␣
↪f'checkpoint-{self.ident}.pt'))
        self.val_loss_min = val_loss
```

### 1.1.2 Conectar google colab a sistema de archivos en la nube (Google Drive)

[3]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/gdrive
```

[4]:
```
data_path = os.path.join(os.getcwd(),'gdrive', 'My Drive','Colab Notebooks',␣
 ↪'data', 'mnist')
log_path = os.path.join(os.getcwd(),'gdrive', 'My Drive','Colab Notebooks',␣
 ↪'logs', 'minst')
os.makedirs(data_path, exist_ok=True, )
```

```
os.makedirs(log_path, exist_ok=True)
tensorboard_path = f"'{log_path}'"
```

[5]:
```
os.listdir(os.getcwd())
```

[5]:
```
['.config', 'gdrive', 'sample_data']
```

### 1.2 Cargar TensorBoard

[6]:
```
!pkill tensorboard
!pkill ngrok
```

[7]:
```
%load_ext tensorboard
```

[8]:
```
%tensorboard --logdir $tensorboard_path --host localhost --port 6006
```

```
<IPython.core.display.Javascript object>
```

#### 1.2.1 Utilizar ngrok para ver tensorboard en un navegador

[9]:
```
!wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
!unzip ngrok-stable-linux-amd64.zip
```

```
--2020-07-23 14:03:17--  https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-
linux-amd64.zip
Resolving bin.equinox.io (bin.equinox.io)... 35.175.20.97, 52.5.95.18,
52.54.124.219, ...
Connecting to bin.equinox.io (bin.equinox.io)|35.175.20.97|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13773305 (13M) [application/octet-stream]
Saving to: ‘ngrok-stable-linux-amd64.zip’

ngrok-stable-linux- 100%[===================>]  13.13M  50.1MB/s    in 0.3s

2020-07-23 14:03:17 (50.1 MB/s) - ‘ngrok-stable-linux-amd64.zip’ saved
[13773305/13773305]

Archive:  ngrok-stable-linux-amd64.zip
  inflating: ngrok
```

[10]:
```
get_ipython().system_raw('./ngrok http 6006 &')
```

[11]:
```
! curl -s http://localhost:4040/api/tunnels | python3 -c \
    "import sys, json; print(json.load(sys.stdin)['tunnels'][0]['public_url'])"
```

```
https://a9ad4746d53b.ngrok.io
```

## 1.3 Preparando el conjunto de datos

[12]:
```
n_epochs = 15

# Fijando la semilla siempre nos dara los mismo resultados cada vez que lo␣
 ↪corramos
random_seed = 1

batch_size_train = 64
validation_set_size = 10000

learning_rate = 0.01

momentum = 0.9
log_interval = 10
patience_value = 3
```

### 1.3.1 Datos de entrenamiento y testeo

**Manejo de los DataLoaders**

Enlace a la documentación

[13]:
```
data_set =   torchvision.datasets.MNIST(data_path, train=True, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor()
                               # , torchvision.transforms.Normalize((0.1307,),␣
 ↪(0.3081,))
                             ]))

test_set = torchvision.datasets.MNIST(data_path, train=False, download = True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor()
                              # , torchvision.transforms.Normalize((0.1307,), (0.
 ↪3081,))
                             ]))

train_set, val_set = torch.utils.data.random_split(data_set, [len(data_set) -␣
 ↪validation_set_size , validation_set_size])
validation_loader = torch.utils.data.DataLoader(val_set, batch_size =␣
 ↪validation_set_size, shuffle=False)
no_batch_train_loader = torch.utils.data.DataLoader(train_set,␣
 ↪batch_size=len(data_set)-validation_set_size, shuffle=False)
train_loader = torch.utils.data.DataLoader(train_set,␣
 ↪batch_size=batch_size_train, shuffle=False)
test_loader = torch.utils.data.DataLoader(test_set,batch_size = len(data_set),␣
 ↪shuffle=False)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
/content/gdrive/My Drive/Colab Notebooks/data/mnist/MNIST/raw/train-images-
idx3-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Extracting /content/gdrive/My Drive/Colab Notebooks/data/mnist/MNIST/raw/train-
images-idx3-ubyte.gz to /content/gdrive/My Drive/Colab
Notebooks/data/mnist/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
/content/gdrive/My Drive/Colab Notebooks/data/mnist/MNIST/raw/train-labels-
idx1-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Extracting /content/gdrive/My Drive/Colab Notebooks/data/mnist/MNIST/raw/train-
labels-idx1-ubyte.gz to /content/gdrive/My Drive/Colab
Notebooks/data/mnist/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
/content/gdrive/My Drive/Colab Notebooks/data/mnist/MNIST/raw/t10k-images-
idx3-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Extracting /content/gdrive/My Drive/Colab
Notebooks/data/mnist/MNIST/raw/t10k-images-idx3-ubyte.gz to /content/gdrive/My
Drive/Colab Notebooks/data/mnist/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
/content/gdrive/My Drive/Colab Notebooks/data/mnist/MNIST/raw/t10k-labels-
idx1-ubyte.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```
Extracting /content/gdrive/My Drive/Colab
Notebooks/data/mnist/MNIST/raw/t10k-labels-idx1-ubyte.gz to /content/gdrive/My
Drive/Colab Notebooks/data/mnist/MNIST/raw
Processing...
```

```
/pytorch/torch/csrc/utils/tensor_numpy.cpp:141: UserWarning: The given NumPy
array is not writeable, and PyTorch does not support non-writeable tensors. This
means you can write to the underlying (supposedly non-writeable) NumPy array
using the tensor. You may want to copy the array to protect its data or make it
writeable before converting it to a tensor. This type of warning will be
suppressed for the rest of this program.
```

```
Done!
```

[14]:
```
len(test_set)
```

[14]: 10000

### 1.3.2 Cargar imágenes en tensorboard

[15]:
```
SGD_wr = SummaryWriter(os.path.join(log_path, 'images'))
```

[16]:
```
def load_images(wr, loader, set_name):
  print(f"Saving {set_name}...")
  for idx, (images, labels) in enumerate(loader):
    batch_grid = torchvision.utils.make_grid(images)
    wr.add_image(f"{set_name}/Lote-{idx}", batch_grid)
  print(f"{set_name} saved.")
```

[17]:
```
load_images(SGD_wr, train_loader, 'Training data')
load_images(SGD_wr, test_loader, 'Test data')
load_images(SGD_wr, validation_loader, 'Validation data')
```

```
Saving Training data...
Training data saved.
Saving Test data...
Test data saved.
Saving Validation data...
Validation data saved.
```

### 1.3.3 Muestra de los datos

[18]:
```
test_examples = enumerate(test_loader)
test_batch_idx, (example_test_data, example_test_targets) = next(test_examples)

train_examples = enumerate(train_loader)
train_batch_idx, (example_train_data, example_train_targets) =␣
 ↪next(train_examples)
```

[19]:
```
print(f"Dimensión de un batch del conjunto de datos de test es␣
 ↪{example_test_data.shape}")
```

```
Dimensión de un batch del conjunto de datos de test es torch.Size([10000, 1, 28,
28])
```

[20]:
```
print(f"Dimensión de un batch del conjunto de datos de test es␣
 ↪{example_train_data.shape}")
```

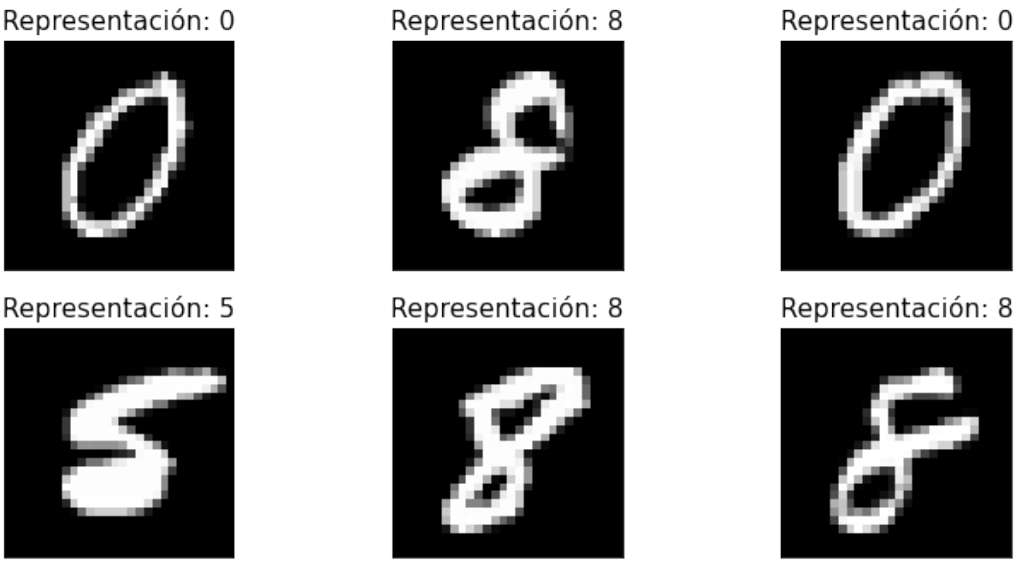
```
Dimensión de un batch del conjunto de datos de test es torch.Size([64, 1, 28,
28])
```

[21]:
```
print(f"Dimensión de los conjuntos:\nEntrenamiento: {len(data_set) -␣
↪validation_set_size }\nValidación: {validation_set_size}\nTest:␣
↪{len(test_set)}")
```

```
Dimensión de los conjuntos:
Entrenamiento: 50000
Validación: 10000
Test: 10000
```

[22]:
```
fig = plt.figure(figsize=(10,5))
for i in range(6):
  plt.subplot(2,3,i+1)
  plt.tight_layout()
  plt.imshow(example_train_data[i][0], cmap='gray', interpolation='none')
  plt.title("Representación: {}".format(example_train_targets[i]), {'fontsize':
↪15})
  plt.xticks([])
  plt.yticks([])

fig.show
```

[22]:
```
<bound method Figure.show of <Figure size 720x360 with 6 Axes>>
```



## 1.4 Construyendo la red neuronal

[23]:
```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

[24]:
```
class Net(nn.Module):
    def __init__(self, name):
        super(Net, self).__init__()
        self.name = name
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))

        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        # x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

### 1.4.1 Creación de modelos a comparar fijación de criterios de optimización

Optimización –> - Gradiente descendente estocástico - Gradiente descendente estocastico con momento - ADAM

[25]:
```
torch.manual_seed(random_seed)
SGD_modelo = Net("Modelo con gradiente descendente estocástico")
SGD_optimizer = optim.SGD(SGD_modelo.parameters(), lr=learning_rate)
SGD_early_stopper = EarlyStopping('SGD', data_path, patience=patience_value)
SGD_wr = SummaryWriter(os.path.join(log_path,'SGD'))
SGD_val_wr = SummaryWriter(os.path.join(log_path,'SGD_val'))

torch.manual_seed(random_seed)
SGDM_modelo = Net("Modelo con gradiente descendente estocástico con momento")
SGDM_optimizer = optim.SGD(SGDM_modelo.parameters(), lr=learning_rate,␣
 ↪momentum=momentum)
SGDM_early_stopper = EarlyStopping('SGD', data_path, patience=patience_value)
SGDM_wr = SummaryWriter(os.path.join(log_path,'SGDM'))
SGDM_val_wr = SummaryWriter(os.path.join(log_path,'SGDM_val'))


torch.manual_seed(random_seed)
```

```
ADAM_modelo = Net("Modelo con ADAM")
ADAM_optimizer = optim.Adam(ADAM_modelo.parameters(), lr=learning_rate)
ADAM_early_stopper = EarlyStopping('ADAM', data_path, patience=patience_value)
ADAM_wr = SummaryWriter(os.path.join(log_path,'ADAM'))
ADAM_val_wr = SummaryWriter(os.path.join(log_path,'ADAM_val'))
```

[26]:
```
p = sum(p.numel() for p in SGD_modelo.parameters() if p.requires_grad)
print(f'El número de parámetros entrenables es: {p}')
```

```
El número de parámetros entrenables es: 21840
```

## 1.5 Transformación del input en el output

[27]:
```
examples_train_data = enumerate(train_loader)
batch_idx, (example_data, example_targets) = next(examples_train_data)
```

### 1.5.1 Salida de bloque uno (Convolucional)

**Convolución**

- Padding = 0
- Filters size = 5 x 5
- Number of features map per image = 10
- Size of the features map = 24 x 24

En este caso el batch es igual a 64 por lo que existen 64 x 10 mapas de características.

[28]:
```
SGD_modelo.conv1(example_data).shape
```

[28]:
```
torch.Size([64, 10, 24, 24])
```

**Pooling**

- Kernel size = 2 x 2
- Features map size = 12 x 12

[29]:
```
F.max_pool2d(SGD_modelo.conv1(example_data), 2).shape
```

[29]:
```
torch.Size([64, 10, 12, 12])
```

**Función de activación –> ReLu** Solo lo valores mayores que cero permanecen

[30]:
```
block_one_output = F.relu(F.max_pool2d(SGD_modelo.conv1(example_data), 2))
print("La dimensión de cada mapa de características es {} \n ".
 ↪format(block_one_output[0][0].shape))
block_one_output.shape
```

```
La dimensión de cada mapa de características es torch.Size([12, 12])
```

[30]:
```
torch.Size([64, 10, 12, 12])
```

### 1.5.2 Salida de bloque dos (Convolucional)

**Convolución**

- Padding = 0
- Filters size = 5 x 5
- Number of features map per image = 20
- Size of the features map = 8 x 8

En este caso el batch es igual a 64 por lo que existen 64 x 20 mapas de características.

[31]:
```
SGD_modelo.conv2(block_one_output).shape
```

[31]:
```
torch.Size([64, 20, 8, 8])
```

**Pooling**

- Kernel size = 2 x 2
- Features map size = 4 x 4

[32]:
```
F.max_pool2d(SGD_modelo.conv2(block_one_output), 2).shape
```

[32]:
```
torch.Size([64, 20, 4, 4])
```

**Función de activación –> ReLu** Solo lo valores mayores que cero permanecen

[33]:
```
block_two_output = F.relu(F.max_pool2d(SGD_modelo.conv2(block_one_output), 2))
print("La dimensión de cada mapa de características es {} \n ".
 ↪format(block_two_output[0][0].shape))
block_two_output.shape
```

```
La dimensión de cada mapa de características es torch.Size([4, 4])
```

[33]:
```
torch.Size([64, 20, 4, 4])
```

### 1.5.3 Salida de bloque tres

Transforma los mapas de características en un vector de características

El argumento -1 indica que el número de imágenes en el batch se infiere

[34]:
```
block_three_output = block_two_output.view(-1, 20 * 4 * 4)
block_three_output.shape
```

[34]:
```
torch.Size([64, 320])
```

### 1.5.4 Salida del bloque cuatro

Se opera como si fuera una Feed Forward Neural Network > Función de activación ReLu > Reduce el vector de características de 320 a 50

[35]:
```
block_four_output = F.relu(SGD_modelo.fc1(block_three_output))
block_four_output.shape
```

[35]:
```
torch.Size([64, 50])
```

### 1.5.5 Salida del bloque cinco (Función Softmax)

Se reduce de la salida de la última capa a un vector de 10 valores > Número de clases = 10 (rango 0-9)

[36]:
```
  block_five_output =  F.log_softmax(SGD_modelo.fc2(block_four_output), dim=1)
  block_five_output.shape
```

[36]:
```
torch.Size([64, 10])
```

## 1.6 Entrenando la red neuronal

Función de coste > $Cross\_entropy = nll\_loss(log\_softmax(x))$

Negative log likelihood loss (nll)

- Función de activación Softmax –> comúnmente utilizada para problemas de aprendizaje multiclase donde un conjunto de características puede asociaser a 1 de K clases.

[37]:
```
def train(epoch, modelo, optimizer, wr:SummaryWriter):

# train epoch
  for batch_idx, (data, target) in enumerate(train_loader):

    modelo.train()
    optimizer.zero_grad()
    output = modelo(data)
    loss = F.nll_loss(output, target, reduction='mean')
    loss.backward()
    optimizer.step()
    wr.add_scalar('Loss/train', loss.item(), batch_idx + (epoch - 1) *␣
 ↪len(train_loader))

# end epoch
```

[38]:
```
def train_error(epoch, modelo, wr):
  modelo.eval()
  correct = 0
  train_loss = 0
  with torch.no_grad():
    for data, target in no_batch_train_loader:
      output = modelo(data)
      train_loss += F.nll_loss(output, target, reduction='mean').item()
    wr.add_scalar('Loss/early', train_loss, epoch)
  print(f'\nTrain set: Loss: {train_loss:.4f}')
```

[39]:
```
def validation(epoch, modelo, wr, es):
  modelo.eval()
  correct = 0
  validation_loss = 0
  with torch.no_grad():
    for data, target in validation_loader:
      output = modelo(data)
      validation_loss += F.nll_loss(output, target, reduction='mean').item()
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
    es(validation_loss, modelo, epoch)
    wr.add_scalar('Loss/early', validation_loss, epoch)
    wr.add_scalar('Accuracy/Validation', 100. * correct / len(validation_loader.
 ↪dataset), epoch)
  print(f'\nValidation set: Avg. loss: {validation_loss:.4f}, Accuracy:␣
 ↪{correct}/{len(validation_loader.dataset)} ({100. * correct /␣
 ↪len(validation_loader.dataset):.0f}%)\n')
```

[40]:
```
def test(epoch, modelo, wr):
  modelo.eval()
  correct = 0
  test_loss = 0
  with torch.no_grad():
    for data, target in test_loader:
      output = modelo(data)
      test_loss += F.nll_loss(output, target, reduction='mean').item()
      # pred --> Devuelve el index del valor maximo (predicción)
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
    wr.add_scalar('Loss/test', test_loss, epoch)
    wr.add_scalar('Accuracy/test', 100. * correct / len(test_loader.dataset),␣
 ↪epoch)
  print(f'\nTest set: Avg. loss: {test_loss:.4f}, Accuracy: {correct}/
 ↪{len(test_loader.dataset)} ({100. * correct / len(test_loader.dataset):.
 ↪0f}%)\n\n')
```

### 1.6.1 Rutina de aprendizaje

[41]:
```
print("Training with SGD\n ***************** Sin entrenamiento␣
 ↪**********************")
train_error(0, SGD_modelo, SGD_wr)
validation(0, SGD_modelo, SGD_val_wr, SGD_early_stopper)
test(0, SGD_modelo, SGD_wr)

for epoch in range(1, n_epochs + 1):
  print(f'***************** Época: {epoch} *********************')
  train(epoch, SGD_modelo, SGD_optimizer, SGD_wr)

  train_error(epoch, SGD_modelo, SGD_wr)
  validation(epoch, SGD_modelo, SGD_val_wr, SGD_early_stopper)
  test(epoch, SGD_modelo, SGD_wr)
```

```
Training with SGD
 ***************** Sin entrenamiento *********************

Train set: Loss: 2.3121

Validation set: Avg. loss: 2.3105, Accuracy: 1166/10000 (12%)


Test set: Avg. loss: 2.3121, Accuracy: 1135/10000 (11%)


***************** Época: 1 *********************

Train set: Loss: 0.6215

Validation set: Avg. loss: 0.6255, Accuracy: 7874/10000 (79%)


Test set: Avg. loss: 0.6000, Accuracy: 7950/10000 (80%)


***************** Época: 2 *********************

Train set: Loss: 0.3073

Validation set: Avg. loss: 0.3169, Accuracy: 9033/10000 (90%)


Test set: Avg. loss: 0.2835, Accuracy: 9118/10000 (91%)
```

```
********** Época: 3 **********
Train set: Loss: 0.1998
Validation set: Avg. loss: 0.2077, Accuracy: 9365/10000 (94%)

Test set: Avg. loss: 0.1809, Accuracy: 9450/10000 (94%)

********** Época: 4 **********
Train set: Loss: 0.1494
Validation set: Avg. loss: 0.1550, Accuracy: 9521/10000 (95%)

Test set: Avg. loss: 0.1345, Accuracy: 9584/10000 (96%)

********** Época: 5 **********
Train set: Loss: 0.1251
Validation set: Avg. loss: 0.1297, Accuracy: 9602/10000 (96%)

Test set: Avg. loss: 0.1131, Accuracy: 9658/10000 (97%)

********** Época: 6 **********
Train set: Loss: 0.1130
Validation set: Avg. loss: 0.1180, Accuracy: 9643/10000 (96%)

Test set: Avg. loss: 0.1037, Accuracy: 9676/10000 (97%)

********** Época: 7 **********
Train set: Loss: 0.1063
Validation set: Avg. loss: 0.1120, Accuracy: 9664/10000 (97%)

Test set: Avg. loss: 0.0985, Accuracy: 9692/10000 (97%)
```

```
********** Época: 8 **********
Train set: Loss: 0.0975
Validation set: Avg. loss: 0.1036, Accuracy: 9697/10000 (97%)

Test set: Avg. loss: 0.0909, Accuracy: 9716/10000 (97%)

********** Época: 9 **********
Train set: Loss: 0.0889
Validation set: Avg. loss: 0.0957, Accuracy: 9720/10000 (97%)

Test set: Avg. loss: 0.0830, Accuracy: 9741/10000 (97%)

********** Época: 10 **********
Train set: Loss: 0.0786
Validation set: Avg. loss: 0.0858, Accuracy: 9751/10000 (98%)

Test set: Avg. loss: 0.0738, Accuracy: 9766/10000 (98%)

********** Época: 11 **********
Train set: Loss: 0.0715
Validation set: Avg. loss: 0.0790, Accuracy: 9761/10000 (98%)

Test set: Avg. loss: 0.0676, Accuracy: 9788/10000 (98%)

********** Época: 12 **********
Train set: Loss: 0.0652
Validation set: Avg. loss: 0.0730, Accuracy: 9780/10000 (98%)
```

```
Test set: Avg. loss: 0.0622, Accuracy: 9804/10000 (98%)


********** Época: 13 **********

Train set: Loss: 0.0605

Validation set: Avg. loss: 0.0687, Accuracy: 9793/10000 (98%)


Test set: Avg. loss: 0.0586, Accuracy: 9813/10000 (98%)


********** Época: 14 **********

Train set: Loss: 0.0569

Validation set: Avg. loss: 0.0655, Accuracy: 9801/10000 (98%)


Test set: Avg. loss: 0.0560, Accuracy: 9821/10000 (98%)


********** Época: 15 **********

Train set: Loss: 0.0540

Validation set: Avg. loss: 0.0630, Accuracy: 9805/10000 (98%)


Test set: Avg. loss: 0.0540, Accuracy: 9818/10000 (98%)
```

[42]:
```python
print("Training with SGDM\n********** Sin entrenamiento␣
 ↪**********")
train_error(0, SGDM_modelo, SGDM_wr)
validation(0, SGDM_modelo, SGDM_val_wr, SGDM_early_stopper)
test(0, SGDM_modelo, SGDM_wr)

for epoch in range(1, n_epochs + 1):
  print(f'********** Época: {epoch} **********')
  train(epoch, SGDM_modelo, SGDM_optimizer, SGDM_wr)
  train_error(epoch, SGDM_modelo, SGDM_wr)
  validation(epoch, SGDM_modelo, SGDM_val_wr, SGDM_early_stopper)
  test(epoch, SGDM_modelo, SGDM_wr)
```

```
Training with SGDM
****************** Sin entrenamiento **********************

Train set: Loss: 2.3121

Validation set: Avg. loss: 2.3105, Accuracy: 1166/10000 (12%)


Test set: Avg. loss: 2.3121, Accuracy: 1135/10000 (11%)


****************** Época: 1 **********************

Train set: Loss: 0.3014

Validation set: Avg. loss: 0.3100, Accuracy: 9033/10000 (90%)


Test set: Avg. loss: 0.2767, Accuracy: 9150/10000 (92%)


****************** Época: 2 **********************

Train set: Loss: 0.1516

Validation set: Avg. loss: 0.1574, Accuracy: 9519/10000 (95%)


Test set: Avg. loss: 0.1378, Accuracy: 9575/10000 (96%)


****************** Época: 3 **********************

Train set: Loss: 0.1079

Validation set: Avg. loss: 0.1135, Accuracy: 9661/10000 (97%)


Test set: Avg. loss: 0.0985, Accuracy: 9700/10000 (97%)


****************** Época: 4 **********************

Train set: Loss: 0.0865

Validation set: Avg. loss: 0.0923, Accuracy: 9727/10000 (97%)
```

```
Test set: Avg. loss: 0.0795, Accuracy: 9747/10000 (97%)

********** Época: 5 **********
Train set: Loss: 0.0757
Validation set: Avg. loss: 0.0818, Accuracy: 9747/10000 (97%)

Test set: Avg. loss: 0.0708, Accuracy: 9780/10000 (98%)

********** Época: 6 **********
Train set: Loss: 0.0677
Validation set: Avg. loss: 0.0747, Accuracy: 9763/10000 (98%)

Test set: Avg. loss: 0.0649, Accuracy: 9794/10000 (98%)

********** Época: 7 **********
Train set: Loss: 0.0611
Validation set: Avg. loss: 0.0689, Accuracy: 9783/10000 (98%)

Test set: Avg. loss: 0.0607, Accuracy: 9802/10000 (98%)

********** Época: 8 **********
Train set: Loss: 0.0548
Validation set: Avg. loss: 0.0636, Accuracy: 9795/10000 (98%)

Test set: Avg. loss: 0.0565, Accuracy: 9809/10000 (98%)

********** Época: 9 **********
Train set: Loss: 0.0502
Validation set: Avg. loss: 0.0602, Accuracy: 9811/10000 (98%)
```

```
Test set: Avg. loss: 0.0535, Accuracy: 9823/10000 (98%)

********** Época: 10 **********
Train set: Loss: 0.0460
Validation set: Avg. loss: 0.0573, Accuracy: 9826/10000 (98%)

Test set: Avg. loss: 0.0506, Accuracy: 9831/10000 (98%)

********** Época: 11 **********
Train set: Loss: 0.0422
Validation set: Avg. loss: 0.0547, Accuracy: 9835/10000 (98%)

Test set: Avg. loss: 0.0483, Accuracy: 9841/10000 (98%)

********** Época: 12 **********
Train set: Loss: 0.0390
Validation set: Avg. loss: 0.0528, Accuracy: 9837/10000 (98%)

Test set: Avg. loss: 0.0464, Accuracy: 9848/10000 (98%)

********** Época: 13 **********
Train set: Loss: 0.0362
Validation set: Avg. loss: 0.0512, Accuracy: 9846/10000 (98%)

Test set: Avg. loss: 0.0449, Accuracy: 9854/10000 (99%)

********** Época: 14 **********
Train set: Loss: 0.0341
```

```
Validation set: Avg. loss: 0.0504, Accuracy: 9844/10000 (98%)

Test set: Avg. loss: 0.0438, Accuracy: 9856/10000 (99%)

********** Época: 15 **********

Train set: Loss: 0.0323

Validation set: Avg. loss: 0.0499, Accuracy: 9846/10000 (98%)

Test set: Avg. loss: 0.0429, Accuracy: 9863/10000 (99%)
```

[43]:
```
print("Training with ADAM\n********** Sin entrenamiento␣
 ↪**********")
train_error(0, ADAM_modelo, ADAM_wr)
validation(0, ADAM_modelo, ADAM_val_wr, ADAM_early_stopper)
test(0, ADAM_modelo, ADAM_wr)

for epoch in range(1, n_epochs + 1):
  print(f'********** Época: {epoch} **********')
  train(epoch, ADAM_modelo, ADAM_optimizer, ADAM_wr)

  train_error(epoch, ADAM_modelo, ADAM_wr)
  validation(epoch, ADAM_modelo, ADAM_val_wr, ADAM_early_stopper)
  test(epoch, ADAM_modelo, ADAM_wr)
```

```
Training with ADAM
********** Sin entrenamiento **********

Train set: Loss: 2.3121

Validation set: Avg. loss: 2.3105, Accuracy: 1166/10000 (12%)

Test set: Avg. loss: 2.3121, Accuracy: 1135/10000 (11%)

********** Época: 1 **********

Train set: Loss: 0.0852

Validation set: Avg. loss: 0.0953, Accuracy: 9687/10000 (97%)
```

```
Test set: Avg. loss: 0.0768, Accuracy: 9745/10000 (97%)

********** Época: 2 **********

Train set: Loss: 0.0534

Validation set: Avg. loss: 0.0671, Accuracy: 9799/10000 (98%)

Test set: Avg. loss: 0.0575, Accuracy: 9820/10000 (98%)

********** Época: 3 **********

Train set: Loss: 0.0514

Validation set: Avg. loss: 0.0759, Accuracy: 9796/10000 (98%)

Test set: Avg. loss: 0.0589, Accuracy: 9814/10000 (98%)

********** Época: 4 **********

Train set: Loss: 0.0454

Validation set: Avg. loss: 0.0715, Accuracy: 9800/10000 (98%)

Test set: Avg. loss: 0.0556, Accuracy: 9833/10000 (98%)

********** Época: 5 **********

Train set: Loss: 0.0531
[5] Mejora estancada por 3 iteraciones (0.067148 --> 0.079713).  Activando early
stop ...

Validation set: Avg. loss: 0.0797, Accuracy: 9787/10000 (98%)

Test set: Avg. loss: 0.0625, Accuracy: 9839/10000 (98%)

********** Época: 6 **********
```

```
Train set: Loss: 0.0460

Validation set: Avg. loss: 0.0778, Accuracy: 9802/10000 (98%)


Test set: Avg. loss: 0.0623, Accuracy: 9822/10000 (98%)


********** Época: 7 **********

Train set: Loss: 0.0457

Validation set: Avg. loss: 0.0683, Accuracy: 9812/10000 (98%)


Test set: Avg. loss: 0.0646, Accuracy: 9832/10000 (98%)


********** Época: 8 **********

Train set: Loss: 0.0664

Validation set: Avg. loss: 0.0940, Accuracy: 9733/10000 (97%)


Test set: Avg. loss: 0.0814, Accuracy: 9782/10000 (98%)


********** Época: 9 **********

Train set: Loss: 0.0410

Validation set: Avg. loss: 0.0681, Accuracy: 9805/10000 (98%)


Test set: Avg. loss: 0.0673, Accuracy: 9824/10000 (98%)


********** Época: 10 **********

Train set: Loss: 0.0365

Validation set: Avg. loss: 0.0763, Accuracy: 9814/10000 (98%)


Test set: Avg. loss: 0.0758, Accuracy: 9823/10000 (98%)
```

```
****************** Época: 11 ********************

Train set: Loss: 0.0714

Validation set: Avg. loss: 0.1116, Accuracy: 9750/10000 (98%)


Test set: Avg. loss: 0.1037, Accuracy: 9748/10000 (97%)


****************** Época: 12 ********************

Train set: Loss: 0.0457

Validation set: Avg. loss: 0.0859, Accuracy: 9805/10000 (98%)


Test set: Avg. loss: 0.0885, Accuracy: 9819/10000 (98%)


****************** Época: 13 ********************

Train set: Loss: 0.0382

Validation set: Avg. loss: 0.0772, Accuracy: 9826/10000 (98%)


Test set: Avg. loss: 0.0886, Accuracy: 9808/10000 (98%)


****************** Época: 14 ********************

Train set: Loss: 0.0446

Validation set: Avg. loss: 0.0832, Accuracy: 9803/10000 (98%)


Test set: Avg. loss: 0.0773, Accuracy: 9816/10000 (98%)


****************** Época: 15 ********************

Train set: Loss: 0.0420

Validation set: Avg. loss: 0.0860, Accuracy: 9825/10000 (98%)
```

```
Test set: Avg. loss: 0.0866, Accuracy: 9840/10000 (98%)
```

### 1.7 Evaluando los modelos

[44]:
```
if (SGD_early_stopper.early_stop):
  print(f"Ha habido una parada temprana en el modelo de SGD con un coste de␣
 ↪{SGD_early_stopper.val_loss_min}\nÉpoca número:{SGD_early_stopper.
 ↪stop_iter}\n")
else:
  print(f"No ha habido early stop en SGD, siendo el menor coste:␣
 ↪{SGD_early_stopper.val_loss_min}\n")

if (SGDM_early_stopper.early_stop):
  print(f"Ha habido una parada temprana en el modelo de SGD con momento con un␣
 ↪coste de {SGDM_early_stopper.val_loss_min}\nÉpoca número:{SGDM_early_stopper.
 ↪stop_iter}\n")
else:
  print(f"No ha habido early stop en SGD con momento, siendo el menor coste:␣
 ↪{SGDM_early_stopper.val_loss_min}\n")

if (ADAM_early_stopper.early_stop):
  print(f"Ha habido una parada temprana en el modelo de ADAM con una precisión␣
 ↪de {ADAM_early_stopper.val_loss_min}\nÉpoca número:{ADAM_early_stopper.
 ↪stop_iter}\n")
else:
  print(f"No ha habido early stop en ADAM, siendo el menor coste:␣
 ↪{ADAM_early_stopper.val_loss_min}\n")
```

```
No ha habido early stop en SGD, siendo el menor coste: 0.06296864151954651

No ha habido early stop en SGD con momento, siendo el menor coste:
0.04994836822152138

Ha habido una parada temprana en el modelo de ADAM con una precisión de
0.06714840978384018
Época número:2
```

# Apéndice C

# Clasificador de imágenes CIFAR-10

# Clasificador del conjunto de datos CIFAR-10

## 1 Classificador de imágenes (Cifar-10 Database)

### 1.1 Librerías y clases necesarías

[1]:
```
import torch
import torchvision

import matplotlib.pyplot as plt
import os, datetime
from itertools import cycle

#Tensorboard
from torch.utils.tensorboard import SummaryWriter
import numpy as np
```

[2]:
```
class EarlyStopping:
    """Parada temprara del entrenamiento si el coste de validación no mejora␣
 ↪tras {patience} iteraciones"""
    def __init__(self, ident, path, patience=7, verbose=True, delta=0):

        self.ident = ident
        self.path = path
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta
        self.stop_iter = 0

    def __call__(self, val_loss, model, i):

        score = -val_loss

        if self.best_score is None:
            self.best_score = score
```

```
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta:
            self.counter += 1
            #print(f'EarlyStopping counter: {self.counter} de {self.patience}')
            if self.counter >= self.patience:
              if not self.early_stop:
                self.early_stop = True
                self.stop_iter = i - self.patience
                if self.verbose:
                  print(f'Mejora estancada por {self.patience} épocas ({self.
↪val_loss_min:.6f} --> {val_loss:.6f}).  Activando early stop ...')

        else:
          self.save_checkpoint(val_loss, model)
          self.best_score = score
          self.counter = 0

    def save_checkpoint(self, val_loss, model):
        '''Guarda el valor de los parámetros en checkpoint'''
        torch.save(model.state_dict(), os.path.join(data_path,␣
↪f'checkpoint-{self.ident}.pt'))
        self.val_loss_min = val_loss
```

### 1.1.1 Conectar google colab a sistema de archivos en la nube (Google Drive)

[3]:
```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/gdrive
```

[4]:
```
data_path = os.path.join(os.getcwd(),'gdrive', 'My Drive','Colab Notebooks',␣
 ↪'data', 'cifar-10')
log_path = os.path.join(os.getcwd(),'gdrive', 'My Drive','Colab Notebooks',␣
 ↪'logs', 'cifar-10')
os.makedirs(data_path, exist_ok=True)
os.makedirs(log_path, exist_ok=True)
tensorboard_path = f"'{log_path}'"
```

## 1.2 Cargar TensorBoard

### 1.2.1 Utilizar ngrok para ver tensorboard en un navegador

[5]:
```
!pkill tensorboard
!pkill ngrok
```

[6]:
```
%load_ext tensorboard
```

[7]:
```
%tensorboard --logdir $tensorboard_path --host localhost --port 6007
```

```
<IPython.core.display.Javascript object>
```

[8]:
```
get_ipython().system_raw('./ngrok http 6007 &')
```

[9]:
```
! curl -s http://localhost:4040/api/tunnels | python3 -c \
    "import sys, json; print(json.load(sys.stdin)['tunnels'][0]['public_url'])"
```

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/usr/lib/python3.6/json/__init__.py", line 299, in load
    parse_constant=parse_constant, object_pairs_hook=object_pairs_hook, **kw)
  File "/usr/lib/python3.6/json/__init__.py", line 354, in loads
    return _default_decoder.decode(s)
  File "/usr/lib/python3.6/json/decoder.py", line 339, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
  File "/usr/lib/python3.6/json/decoder.py", line 357, in raw_decode
    raise JSONDecodeError("Expecting value", s, err.value) from None
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

## 1.3 Preparando el conjunto de datos

[10]:
```
n_epochs = 40

batch_size_train = 25

learning_rate = 0.0005
sgd_learning_rate= 0.001

momentum = 0.9
log_interval = 50
batch_test_interval = 30
patience_value = 3

validation_set_size = 10000
```

```
# Fijando la semilla siempre nos dara los mismo resultados cada vez que lo␣
 ↪corramos
random_seed = 1
```

### 1.3.1 Datos de entrenamiento y testeo

**Manejo de los DataLoaders**

Enlace a la documentación

[11]:
```
data_set =   torchvision.datasets.CIFAR10(data_path, train=True, download=True,
                            transform=torchvision.transforms.Compose([
                              torchvision.transforms.ToTensor()]))

test_set = torchvision.datasets.CIFAR10(data_path, train=False, download = True,
                            transform=torchvision.transforms.Compose([
                              torchvision.transforms.ToTensor()]))

train_set, val_set = torch.utils.data.random_split(data_set, [len(data_set) -␣
 ↪validation_set_size , validation_set_size])
validation_loader = torch.utils.data.DataLoader(val_set,␣
 ↪batch_size=validation_set_size, shuffle=False)
no_batch_train_loader = torch.utils.data.DataLoader(train_set,batch_size=␣
 ↪len(data_set) - validation_set_size, shuffle=False)
train_loader = torch.utils.data.DataLoader(train_set,␣
 ↪batch_size=batch_size_train, shuffle=False)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=len(test_set),␣
 ↪shuffle=False)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
/content/gdrive/My Drive/Colab Notebooks/data/cifar-10/cifar-10-python.tar.gz
```

```
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0), HTML(value='')))
```

```


Extracting /content/gdrive/My Drive/Colab
Notebooks/data/cifar-10/cifar-10-python.tar.gz to /content/gdrive/My Drive/Colab
Notebooks/data/cifar-10
Files already downloaded and verified
```

### 1.3.2 Cargar imágenes en tensorboard

[13]:
```
SGD_wr = SummaryWriter(os.path.join(log_path, 'images'))
```

[14]:
```
def load_images(wr, loader, set_name):
  print(f"Saving {set_name}...")
  for idx, (images, labels) in enumerate(loader):
    batch_grid = torchvision.utils.make_grid(images)
```

```
    wr.add_image(f"{set_name}/Lote-{idx}", batch_grid)
  print(f"{set_name} saved.")
```

[15]:
```
load_images(SGD_wr, train_loader, 'Training data')
load_images(SGD_wr, test_loader, 'Test data')
load_images(SGD_wr, validation_loader, 'Validation data')
```

```
Saving Training data...
Training data saved.
Saving Test data...
Test data saved.
Saving Validation data...
Validation data saved.
```

### 1.3.3 Muestra de los datos

[16]:
```
test_examples = enumerate(test_loader)
test_batch_idx, (example_test_data, example_test_targets) = next(test_examples)

train_examples = enumerate(train_loader)
train_batch_idx, (example_train_data, example_train_targets) =␣
 ↪next(train_examples)
```

[17]:
```
len(train_loader)
```

[17]: 1600

[18]:
```
print(f"Dimensión de un batch del conjunto de datos de test es␣
 ↪{example_test_data.shape}")
```

```
Dimensión de un batch del conjunto de datos de test es torch.Size([10000, 3, 32,
32])
```

[19]:
```
print(f"Dimensión de un batch del conjunto de datos de entrenamiento es␣
 ↪{example_train_data.shape}")
```

```
Dimensión de un batch del conjunto de datos de entrenamiento es torch.Size([25,
3, 32, 32])
```

[20]:
```
print(f"Dimensión de los conjuntos:\nEntrenamiento: {len(data_set) -␣
 ↪validation_set_size}\nValidación: {validation_set_size}\nTest:␣
 ↪{len(test_set)}")
```

```
Dimensión de los conjuntos:
Entrenamiento: 40000
Validación: 10000
Test: 10000
```

## 1.4 Construyendo la red neuronal

[21]:
```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

[22]:
```
class Net(nn.Module):
    def __init__(self, name):
        super(Net, self).__init__()
        self.name = name
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

### 1.4.1 Creación de modelos a comparar fijación de criterios de optimización

Optimización –> - Gradiente descendiente estocástico por lotes - Gradiente descendente estocástico por lotes con momento - ADAM

[23]:
```
torch.manual_seed(random_seed)
SGD_modelo = Net("Modelo con gradiente descendente estocástico")
SGD_optimizer = optim.SGD(SGD_modelo.parameters(), lr=sgd_learning_rate)
SGD_early_stopper = EarlyStopping('SGD', data_path, patience=patience_value)
SGD_wr = SummaryWriter(os.path.join(log_path,'SGD'))
SGD_val_wr = SummaryWriter(os.path.join(log_path,'SGD_val'))

torch.manual_seed(random_seed)
SGDM_modelo = Net("Modelo con gradiente descendente estocástico con momento")
SGDM_optimizer = optim.SGD(SGDM_modelo.parameters(), lr=learning_rate,␣
 ↪momentum=momentum)
SGDM_early_stopper = EarlyStopping('SGD', data_path, patience=patience_value)
SGDM_wr = SummaryWriter(os.path.join(log_path,'SGDM'))
SGDM_val_wr = SummaryWriter(os.path.join(log_path,'SGDM_val'))
```

```
torch.manual_seed(random_seed)
ADAM_modelo = Net("Modelo con ADAM")
ADAM_optimizer = optim.Adam(ADAM_modelo.parameters(), lr=learning_rate, betas=␣
↪(momentum, 0.99))
ADAM_early_stopper = EarlyStopping('ADAM', data_path, patience=patience_value)
ADAM_wr = SummaryWriter(os.path.join(log_path,'ADAM'))
ADAM_val_wr = SummaryWriter(os.path.join(log_path,'ADAM_val'))
```

[24]:
```
p = sum(p.numel() for p in SGD_modelo.parameters() if p.requires_grad)
print(f'El número de parámetros entrenables es: {p}')
```

```
El número de parámetros entrenables es: 62006
```

### 1.5 Entrenando la red neuronal

Función de coste > $Cross\_entropy = nll\_loss(log\_softmax(x))$

Negative log likelihood loss (nll)

- Función de activación Softmax –> comúnmente utilizada para problemas de aprendizaje multiclase donde un conjunto de características puede asociaser a 1 de K clases.

[25]:
```
def train(epoch, modelo, optimizer, wr:SummaryWriter):

  for batch_idx, (data, target) in enumerate(train_loader):

    modelo.train()
    optimizer.zero_grad()
    output = modelo(data)
    loss = F.nll_loss(output, target, reduction='mean')
    loss.backward()
    optimizer.step()
    wr.add_scalar('Loss/train', loss.item(), batch_idx + (epoch - 1) *␣
↪len(train_loader))
```

[26]:
```
def train_error(epoch, modelo, wr):
  modelo.eval()
  correct = 0
  train_loss = 0
  with torch.no_grad():
    for data, target in no_batch_train_loader:
      output = modelo(data)
      train_loss += F.nll_loss(output, target, reduction='mean').item()
    wr.add_scalar('Loss/early', train_loss, epoch)
  print(f'\nTrain set: Avg. loss: {train_loss:.4f}')
```

[27]:
```
def validation(epoch, modelo, wr, es):
  modelo.eval()
```

```
  correct = 0
  validation_loss = 0
  with torch.no_grad():
    for data, target in validation_loader:
      output = modelo(data)
      validation_loss += F.nll_loss(output, target, reduction='mean').item()
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
    es(validation_loss, modelo, epoch)
    wr.add_scalar('Loss/early', validation_loss, epoch)
    wr.add_scalar('Accuracy/validation', 100. * correct / len(validation_loader.
 ↪dataset), epoch)
  print(f'\nValidation set: Avg. loss: {validation_loss:.4f}, Accuracy:␣
 ↪{correct}/{len(validation_loader.dataset)} ({100. * correct /␣
 ↪len(validation_loader.dataset):.0f}%)')
```

[28]:
```
def test(epoch, modelo, wr):
  modelo.eval()
  correct = 0
  test_loss = 0
  with torch.no_grad():
    for data, target in test_loader:
      output = modelo(data)
      # Reduction mean --> Calcula la función de coste por cada ejemplo y hace␣
 ↪la media
      test_loss += F.nll_loss(output, target, reduction='mean').item()
      # pred --> Devuelve el index del valor maximo (predicción)
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
    wr.add_scalar('Loss/test', test_loss, epoch)
    wr.add_scalar('Accuracy/test', 100. * correct / len(test_loader.dataset),␣
 ↪epoch)
  print(f'\nTest set: Avg. loss: {test_loss:.4f}, Accuracy: {correct}/
 ↪{len(test_loader.dataset)} ({100. * correct / len(test_loader.dataset):.
 ↪0f}%)\n')
```

### 1.5.1 Rutina de entrenamiento

[29]:
```
print("Training with SGD\n***************** Sin entrenamiento␣
 ↪**********************")
train_error(0, SGD_modelo, SGD_wr)
validation(0, SGD_modelo, SGD_val_wr, SGD_early_stopper)
test(0, SGD_modelo, SGD_wr)

for epoch in range(1, n_epochs + 1):
  print(f'***************** Época: {epoch} ********************')
```

```
  train(epoch, SGD_modelo, SGD_optimizer, SGD_wr)

  train_error(epoch, SGD_modelo, SGD_wr)
  validation(epoch, SGD_modelo, SGD_val_wr, SGD_early_stopper)
  test(epoch, SGD_modelo, SGD_wr)
```

```
Training with SGD
****************** Sin entrenamiento **********************

Train set: Avg. loss: 2.3035

Validation set: Avg. loss: 2.3032, Accuracy: 1125/10000 (11%)

Test set: Avg. loss: 2.3034, Accuracy: 1080/10000 (11%)

****************** Época: 1 **********************

Train set: Avg. loss: 2.3025

Validation set: Avg. loss: 2.3023, Accuracy: 1123/10000 (11%)

Test set: Avg. loss: 2.3025, Accuracy: 1086/10000 (11%)

****************** Época: 2 **********************

Train set: Avg. loss: 2.3017

Validation set: Avg. loss: 2.3016, Accuracy: 1129/10000 (11%)

Test set: Avg. loss: 2.3016, Accuracy: 1059/10000 (11%)

****************** Época: 3 **********************

Train set: Avg. loss: 2.3007

Validation set: Avg. loss: 2.3008, Accuracy: 1147/10000 (11%)

Test set: Avg. loss: 2.3007, Accuracy: 1094/10000 (11%)

****************** Época: 4 **********************

Train set: Avg. loss: 2.2995

Validation set: Avg. loss: 2.2996, Accuracy: 1581/10000 (16%)

Test set: Avg. loss: 2.2994, Accuracy: 1592/10000 (16%)
```

```
********** Época: 5 **********
Train set: Avg. loss: 2.2975
Validation set: Avg. loss: 2.2977, Accuracy: 1279/10000 (13%)
Test set: Avg. loss: 2.2974, Accuracy: 1297/10000 (13%)
********** Época: 6 **********
Train set: Avg. loss: 2.2942
Validation set: Avg. loss: 2.2946, Accuracy: 1076/10000 (11%)
Test set: Avg. loss: 2.2941, Accuracy: 1111/10000 (11%)
********** Época: 7 **********
Train set: Avg. loss: 2.2882
Validation set: Avg. loss: 2.2888, Accuracy: 1060/10000 (11%)
Test set: Avg. loss: 2.2880, Accuracy: 1058/10000 (11%)
********** Época: 8 **********
Train set: Avg. loss: 2.2761
Validation set: Avg. loss: 2.2769, Accuracy: 1189/10000 (12%)
Test set: Avg. loss: 2.2758, Accuracy: 1191/10000 (12%)
********** Época: 9 **********
Train set: Avg. loss: 2.2455
Validation set: Avg. loss: 2.2470, Accuracy: 1531/10000 (15%)
Test set: Avg. loss: 2.2451, Accuracy: 1495/10000 (15%)
********** Época: 10 **********
Train set: Avg. loss: 2.1650
Validation set: Avg. loss: 2.1678, Accuracy: 1978/10000 (20%)
Test set: Avg. loss: 2.1648, Accuracy: 1950/10000 (20%)
```

```
********** Época: 11 **********
Train set: Avg. loss: 2.0684
Validation set: Avg. loss: 2.0704, Accuracy: 2381/10000 (24%)
Test set: Avg. loss: 2.0674, Accuracy: 2417/10000 (24%)
********** Época: 12 **********
Train set: Avg. loss: 2.0316
Validation set: Avg. loss: 2.0332, Accuracy: 2522/10000 (25%)
Test set: Avg. loss: 2.0287, Accuracy: 2585/10000 (26%)
********** Época: 13 **********
Train set: Avg. loss: 2.0111
Validation set: Avg. loss: 2.0130, Accuracy: 2570/10000 (26%)
Test set: Avg. loss: 2.0067, Accuracy: 2629/10000 (26%)
********** Época: 14 **********
Train set: Avg. loss: 1.9941
Validation set: Avg. loss: 1.9959, Accuracy: 2631/10000 (26%)
Test set: Avg. loss: 1.9885, Accuracy: 2706/10000 (27%)
********** Época: 15 **********
Train set: Avg. loss: 1.9783
Validation set: Avg. loss: 1.9799, Accuracy: 2668/10000 (27%)
Test set: Avg. loss: 1.9717, Accuracy: 2779/10000 (28%)
********** Época: 16 **********
Train set: Avg. loss: 1.9629
Validation set: Avg. loss: 1.9644, Accuracy: 2770/10000 (28%)
Test set: Avg. loss: 1.9558, Accuracy: 2825/10000 (28%)
```

```
****************** Época: 17 ********************
Train set: Avg. loss: 1.9457
Validation set: Avg. loss: 1.9470, Accuracy: 2828/10000 (28%)
Test set: Avg. loss: 1.9387, Accuracy: 2924/10000 (29%)
****************** Época: 18 ********************
Train set: Avg. loss: 1.9261
Validation set: Avg. loss: 1.9273, Accuracy: 2925/10000 (29%)
Test set: Avg. loss: 1.9191, Accuracy: 3017/10000 (30%)
****************** Época: 19 ********************
Train set: Avg. loss: 1.9036
Validation set: Avg. loss: 1.9048, Accuracy: 3024/10000 (30%)
Test set: Avg. loss: 1.8971, Accuracy: 3131/10000 (31%)
****************** Época: 20 ********************
Train set: Avg. loss: 1.8777
Validation set: Avg. loss: 1.8793, Accuracy: 3151/10000 (32%)
Test set: Avg. loss: 1.8719, Accuracy: 3238/10000 (32%)
****************** Época: 21 ********************
Train set: Avg. loss: 1.8490
Validation set: Avg. loss: 1.8508, Accuracy: 3277/10000 (33%)
Test set: Avg. loss: 1.8436, Accuracy: 3343/10000 (33%)
****************** Época: 22 ********************
Train set: Avg. loss: 1.8186
Validation set: Avg. loss: 1.8208, Accuracy: 3396/10000 (34%)
Test set: Avg. loss: 1.8136, Accuracy: 3444/10000 (34%)
```

```
********** Época: 23 **********
Train set: Avg. loss: 1.7864
Validation set: Avg. loss: 1.7893, Accuracy: 3522/10000 (35%)
Test set: Avg. loss: 1.7822, Accuracy: 3568/10000 (36%)
********** Época: 24 **********
Train set: Avg. loss: 1.7532
Validation set: Avg. loss: 1.7572, Accuracy: 3677/10000 (37%)
Test set: Avg. loss: 1.7501, Accuracy: 3697/10000 (37%)
********** Época: 25 **********
Train set: Avg. loss: 1.7200
Validation set: Avg. loss: 1.7255, Accuracy: 3790/10000 (38%)
Test set: Avg. loss: 1.7178, Accuracy: 3815/10000 (38%)
********** Época: 26 **********
Train set: Avg. loss: 1.6879
Validation set: Avg. loss: 1.6952, Accuracy: 3889/10000 (39%)
Test set: Avg. loss: 1.6861, Accuracy: 3939/10000 (39%)
********** Época: 27 **********
Train set: Avg. loss: 1.6559
Validation set: Avg. loss: 1.6652, Accuracy: 4025/10000 (40%)
Test set: Avg. loss: 1.6544, Accuracy: 4038/10000 (40%)
********** Época: 28 **********
Train set: Avg. loss: 1.6263
Validation set: Avg. loss: 1.6374, Accuracy: 4086/10000 (41%)
Test set: Avg. loss: 1.6249, Accuracy: 4145/10000 (41%)
```

********** Época: 29 **********

Train set: Avg. loss: 1.6002

Validation set: Avg. loss: 1.6134, Accuracy: 4165/10000 (42%)

Test set: Avg. loss: 1.5992, Accuracy: 4232/10000 (42%)

********** Época: 30 **********

Train set: Avg. loss: 1.5776

Validation set: Avg. loss: 1.5928, Accuracy: 4235/10000 (42%)

Test set: Avg. loss: 1.5778, Accuracy: 4323/10000 (43%)

********** Época: 31 **********

Train set: Avg. loss: 1.5579

Validation set: Avg. loss: 1.5746, Accuracy: 4274/10000 (43%)

Test set: Avg. loss: 1.5588, Accuracy: 4369/10000 (44%)

********** Época: 32 **********

Train set: Avg. loss: 1.5407

Validation set: Avg. loss: 1.5581, Accuracy: 4350/10000 (44%)

Test set: Avg. loss: 1.5424, Accuracy: 4424/10000 (44%)

********** Época: 33 **********

Train set: Avg. loss: 1.5248

Validation set: Avg. loss: 1.5434, Accuracy: 4406/10000 (44%)

Test set: Avg. loss: 1.5274, Accuracy: 4478/10000 (45%)

********** Época: 34 **********

Train set: Avg. loss: 1.5104

Validation set: Avg. loss: 1.5301, Accuracy: 4442/10000 (44%)

Test set: Avg. loss: 1.5136, Accuracy: 4523/10000 (45%)

********** Época: 35 **********

Train set: Avg. loss: 1.4967

Validation set: Avg. loss: 1.5174, Accuracy: 4497/10000 (45%)

Test set: Avg. loss: 1.5008, Accuracy: 4578/10000 (46%)

********** Época: 36 **********

Train set: Avg. loss: 1.4827

Validation set: Avg. loss: 1.5045, Accuracy: 4537/10000 (45%)

Test set: Avg. loss: 1.4877, Accuracy: 4623/10000 (46%)

********** Época: 37 **********

Train set: Avg. loss: 1.4701

Validation set: Avg. loss: 1.4929, Accuracy: 4587/10000 (46%)

Test set: Avg. loss: 1.4759, Accuracy: 4658/10000 (47%)

********** Época: 38 **********

Train set: Avg. loss: 1.4582

Validation set: Avg. loss: 1.4817, Accuracy: 4639/10000 (46%)

Test set: Avg. loss: 1.4650, Accuracy: 4689/10000 (47%)

********** Época: 39 **********

Train set: Avg. loss: 1.4464

Validation set: Avg. loss: 1.4704, Accuracy: 4670/10000 (47%)

Test set: Avg. loss: 1.4543, Accuracy: 4733/10000 (47%)

********** Época: 40 **********

Train set: Avg. loss: 1.4360

Validation set: Avg. loss: 1.4609, Accuracy: 4714/10000 (47%)

Test set: Avg. loss: 1.4451, Accuracy: 4752/10000 (48%)

[30]:
```
print("Training with SGD with momentum\n********** Sin entrenamiento␣
 ↪**********")
train_error(0, SGDM_modelo, SGDM_wr)
validation(0, SGDM_modelo, SGDM_val_wr, SGDM_early_stopper)
test(0, SGDM_modelo, SGDM_wr)

for epoch in range(1, n_epochs + 1):
  print(f'********** Época: {epoch} **********')
  train(epoch, SGDM_modelo, SGDM_optimizer, SGDM_wr)

  train_error(epoch, SGDM_modelo, SGDM_wr)
  validation(epoch, SGDM_modelo, SGDM_val_wr, SGDM_early_stopper)
  test(epoch, SGDM_modelo, SGDM_wr)
```

```
Training with SGD with momentum
********** Sin entrenamiento **********

Train set: Avg. loss: 2.3035

Validation set: Avg. loss: 2.3032, Accuracy: 1125/10000 (11%)

Test set: Avg. loss: 2.3034, Accuracy: 1080/10000 (11%)

********** Época: 1 **********

Train set: Avg. loss: 2.2978

Validation set: Avg. loss: 2.2980, Accuracy: 1247/10000 (12%)

Test set: Avg. loss: 2.2977, Accuracy: 1262/10000 (13%)

********** Época: 2 **********

Train set: Avg. loss: 2.1962

Validation set: Avg. loss: 2.1984, Accuracy: 1761/10000 (18%)

Test set: Avg. loss: 2.1957, Accuracy: 1761/10000 (18%)

********** Época: 3 **********

Train set: Avg. loss: 1.9934

Validation set: Avg. loss: 1.9953, Accuracy: 2623/10000 (26%)

Test set: Avg. loss: 1.9872, Accuracy: 2726/10000 (27%)
```

```
********** Época: 4 **********
Train set: Avg. loss: 1.9152
Validation set: Avg. loss: 1.9173, Accuracy: 2987/10000 (30%)
Test set: Avg. loss: 1.9083, Accuracy: 3053/10000 (31%)
********** Época: 5 **********
Train set: Avg. loss: 1.7923
Validation set: Avg. loss: 1.7947, Accuracy: 3460/10000 (35%)
Test set: Avg. loss: 1.7877, Accuracy: 3497/10000 (35%)
********** Época: 6 **********
Train set: Avg. loss: 1.7003
Validation set: Avg. loss: 1.7033, Accuracy: 3787/10000 (38%)
Test set: Avg. loss: 1.6961, Accuracy: 3876/10000 (39%)
********** Época: 7 **********
Train set: Avg. loss: 1.6171
Validation set: Avg. loss: 1.6260, Accuracy: 4122/10000 (41%)
Test set: Avg. loss: 1.6146, Accuracy: 4220/10000 (42%)
********** Época: 8 **********
Train set: Avg. loss: 1.5543
Validation set: Avg. loss: 1.5676, Accuracy: 4354/10000 (44%)
Test set: Avg. loss: 1.5543, Accuracy: 4426/10000 (44%)
********** Época: 9 **********
Train set: Avg. loss: 1.5018
Validation set: Avg. loss: 1.5191, Accuracy: 4548/10000 (45%)
Test set: Avg. loss: 1.5066, Accuracy: 4576/10000 (46%)
```

```
********** Época: 10 **********
Train set: Avg. loss: 1.4589
Validation set: Avg. loss: 1.4799, Accuracy: 4738/10000 (47%)
Test set: Avg. loss: 1.4708, Accuracy: 4717/10000 (47%)
********** Época: 11 **********
Train set: Avg. loss: 1.4225
Validation set: Avg. loss: 1.4482, Accuracy: 4834/10000 (48%)
Test set: Avg. loss: 1.4414, Accuracy: 4814/10000 (48%)
********** Época: 12 **********
Train set: Avg. loss: 1.3900
Validation set: Avg. loss: 1.4207, Accuracy: 4954/10000 (50%)
Test set: Avg. loss: 1.4154, Accuracy: 4908/10000 (49%)
********** Época: 13 **********
Train set: Avg. loss: 1.3635
Validation set: Avg. loss: 1.3997, Accuracy: 5011/10000 (50%)
Test set: Avg. loss: 1.3962, Accuracy: 4995/10000 (50%)
********** Época: 14 **********
Train set: Avg. loss: 1.3282
Validation set: Avg. loss: 1.3711, Accuracy: 5096/10000 (51%)
Test set: Avg. loss: 1.3687, Accuracy: 5098/10000 (51%)
********** Época: 15 **********
Train set: Avg. loss: 1.3055
Validation set: Avg. loss: 1.3554, Accuracy: 5175/10000 (52%)
Test set: Avg. loss: 1.3538, Accuracy: 5162/10000 (52%)
```

```
****************** Época: 16 ********************
Train set: Avg. loss: 1.2830
Validation set: Avg. loss: 1.3391, Accuracy: 5220/10000 (52%)
Test set: Avg. loss: 1.3380, Accuracy: 5247/10000 (52%)
****************** Época: 17 ********************
Train set: Avg. loss: 1.2603
Validation set: Avg. loss: 1.3242, Accuracy: 5254/10000 (53%)
Test set: Avg. loss: 1.3238, Accuracy: 5292/10000 (53%)
****************** Época: 18 ********************
Train set: Avg. loss: 1.2349
Validation set: Avg. loss: 1.3068, Accuracy: 5333/10000 (53%)
Test set: Avg. loss: 1.3068, Accuracy: 5328/10000 (53%)
****************** Época: 19 ********************
Train set: Avg. loss: 1.2184
Validation set: Avg. loss: 1.2979, Accuracy: 5380/10000 (54%)
Test set: Avg. loss: 1.2990, Accuracy: 5360/10000 (54%)
****************** Época: 20 ********************
Train set: Avg. loss: 1.1980
Validation set: Avg. loss: 1.2858, Accuracy: 5455/10000 (55%)
Test set: Avg. loss: 1.2876, Accuracy: 5414/10000 (54%)
****************** Época: 21 ********************
Train set: Avg. loss: 1.1778
Validation set: Avg. loss: 1.2728, Accuracy: 5523/10000 (55%)
Test set: Avg. loss: 1.2757, Accuracy: 5480/10000 (55%)
```

```
********** Época: 22 **********
Train set: Avg. loss: 1.1599
Validation set: Avg. loss: 1.2630, Accuracy: 5578/10000 (56%)
Test set: Avg. loss: 1.2670, Accuracy: 5531/10000 (55%)
********** Época: 23 **********
Train set: Avg. loss: 1.1366
Validation set: Avg. loss: 1.2476, Accuracy: 5624/10000 (56%)
Test set: Avg. loss: 1.2517, Accuracy: 5565/10000 (56%)
********** Época: 24 **********
Train set: Avg. loss: 1.1132
Validation set: Avg. loss: 1.2342, Accuracy: 5665/10000 (57%)
Test set: Avg. loss: 1.2379, Accuracy: 5622/10000 (56%)
********** Época: 25 **********
Train set: Avg. loss: 1.0956
Validation set: Avg. loss: 1.2256, Accuracy: 5720/10000 (57%)
Test set: Avg. loss: 1.2282, Accuracy: 5671/10000 (57%)
********** Época: 26 **********
Train set: Avg. loss: 1.0743
Validation set: Avg. loss: 1.2152, Accuracy: 5739/10000 (57%)
Test set: Avg. loss: 1.2176, Accuracy: 5706/10000 (57%)
********** Época: 27 **********
Train set: Avg. loss: 1.0569
Validation set: Avg. loss: 1.2092, Accuracy: 5766/10000 (58%)
Test set: Avg. loss: 1.2118, Accuracy: 5722/10000 (57%)
```

```
********** Época: 28 **********
Train set: Avg. loss: 1.0387
Validation set: Avg. loss: 1.2028, Accuracy: 5807/10000 (58%)
Test set: Avg. loss: 1.2049, Accuracy: 5767/10000 (58%)
********** Época: 29 **********
Train set: Avg. loss: 1.0238
Validation set: Avg. loss: 1.1993, Accuracy: 5829/10000 (58%)
Test set: Avg. loss: 1.2011, Accuracy: 5808/10000 (58%)
********** Época: 30 **********
Train set: Avg. loss: 1.0077
Validation set: Avg. loss: 1.1953, Accuracy: 5835/10000 (58%)
Test set: Avg. loss: 1.1963, Accuracy: 5826/10000 (58%)
********** Época: 31 **********
Train set: Avg. loss: 0.9948
Validation set: Avg. loss: 1.1932, Accuracy: 5844/10000 (58%)
Test set: Avg. loss: 1.1947, Accuracy: 5862/10000 (59%)
********** Época: 32 **********
Train set: Avg. loss: 0.9782
Validation set: Avg. loss: 1.1894, Accuracy: 5848/10000 (58%)
Test set: Avg. loss: 1.1903, Accuracy: 5875/10000 (59%)
********** Época: 33 **********
Train set: Avg. loss: 0.9648
Validation set: Avg. loss: 1.1874, Accuracy: 5853/10000 (59%)
Test set: Avg. loss: 1.1867, Accuracy: 5871/10000 (59%)
```

```
********** Época: 34 **********

Train set: Avg. loss: 0.9531

Validation set: Avg. loss: 1.1897, Accuracy: 5857/10000 (59%)

Test set: Avg. loss: 1.1875, Accuracy: 5885/10000 (59%)

********** Época: 35 **********

Train set: Avg. loss: 0.9403

Validation set: Avg. loss: 1.1890, Accuracy: 5885/10000 (59%)

Test set: Avg. loss: 1.1867, Accuracy: 5887/10000 (59%)

********** Época: 36 **********

Train set: Avg. loss: 0.9295
Mejora estancada por 3 épocas (1.187384 --> 1.190809).  Activando early stop ...

Validation set: Avg. loss: 1.1908, Accuracy: 5905/10000 (59%)

Test set: Avg. loss: 1.1885, Accuracy: 5887/10000 (59%)

********** Época: 37 **********

Train set: Avg. loss: 0.9173

Validation set: Avg. loss: 1.1922, Accuracy: 5914/10000 (59%)

Test set: Avg. loss: 1.1902, Accuracy: 5925/10000 (59%)

********** Época: 38 **********

Train set: Avg. loss: 0.9058

Validation set: Avg. loss: 1.1959, Accuracy: 5881/10000 (59%)

Test set: Avg. loss: 1.1937, Accuracy: 5926/10000 (59%)

********** Época: 39 **********

Train set: Avg. loss: 0.8957

Validation set: Avg. loss: 1.1992, Accuracy: 5899/10000 (59%)

Test set: Avg. loss: 1.1990, Accuracy: 5923/10000 (59%)
```

```
********** Época: 40 **********

Train set: Avg. loss: 0.8908

Validation set: Avg. loss: 1.2071, Accuracy: 5873/10000 (59%)

Test set: Avg. loss: 1.2077, Accuracy: 5912/10000 (59%)
```

[31]:
```
print("Training with ADAM\n********** Sin entrenamiento␣
 ↪**********")
train_error(0, ADAM_modelo, ADAM_wr)
validation(0, ADAM_modelo, ADAM_val_wr, ADAM_early_stopper)
test(0, ADAM_modelo, ADAM_wr)

for epoch in range(1, n_epochs + 1):
  print(f'********** Época: {epoch} **********')
  train(epoch, ADAM_modelo, ADAM_optimizer, ADAM_wr)

  train_error(epoch, ADAM_modelo, ADAM_wr)
  validation(epoch, ADAM_modelo, ADAM_val_wr, ADAM_early_stopper)
  test(epoch, ADAM_modelo, ADAM_wr)
```

```
Training with ADAM
********** Sin entrenamiento **********

Train set: Avg. loss: 2.3035

Validation set: Avg. loss: 2.3032, Accuracy: 1125/10000 (11%)

Test set: Avg. loss: 2.3034, Accuracy: 1080/10000 (11%)

********** Época: 1 **********

Train set: Avg. loss: 1.6467

Validation set: Avg. loss: 1.6494, Accuracy: 4033/10000 (40%)

Test set: Avg. loss: 1.6464, Accuracy: 4127/10000 (41%)

********** Época: 2 **********

Train set: Avg. loss: 1.5320

Validation set: Avg. loss: 1.5428, Accuracy: 4506/10000 (45%)

Test set: Avg. loss: 1.5376, Accuracy: 4532/10000 (45%)
```

```
****************** Época: 3 **********************
Train set: Avg. loss: 1.4204
Validation set: Avg. loss: 1.4415, Accuracy: 4842/10000 (48%)
Test set: Avg. loss: 1.4362, Accuracy: 4883/10000 (49%)
****************** Época: 4 **********************
Train set: Avg. loss: 1.3268
Validation set: Avg. loss: 1.3623, Accuracy: 5165/10000 (52%)
Test set: Avg. loss: 1.3537, Accuracy: 5175/10000 (52%)
****************** Época: 5 **********************
Train set: Avg. loss: 1.2600
Validation set: Avg. loss: 1.3104, Accuracy: 5384/10000 (54%)
Test set: Avg. loss: 1.2988, Accuracy: 5395/10000 (54%)
****************** Época: 6 **********************
Train set: Avg. loss: 1.2148
Validation set: Avg. loss: 1.2798, Accuracy: 5539/10000 (55%)
Test set: Avg. loss: 1.2663, Accuracy: 5521/10000 (55%)
****************** Época: 7 **********************
Train set: Avg. loss: 1.1598
Validation set: Avg. loss: 1.2398, Accuracy: 5649/10000 (56%)
Test set: Avg. loss: 1.2256, Accuracy: 5682/10000 (57%)
****************** Época: 8 **********************
Train set: Avg. loss: 1.1197
Validation set: Avg. loss: 1.2156, Accuracy: 5756/10000 (58%)
Test set: Avg. loss: 1.1984, Accuracy: 5817/10000 (58%)
```

```
****************** Época: 9 **********************
Train set: Avg. loss: 1.0799
Validation set: Avg. loss: 1.1907, Accuracy: 5867/10000 (59%)
Test set: Avg. loss: 1.1770, Accuracy: 5910/10000 (59%)
****************** Época: 10 **********************
Train set: Avg. loss: 1.0270
Validation set: Avg. loss: 1.1554, Accuracy: 5964/10000 (60%)
Test set: Avg. loss: 1.1405, Accuracy: 6019/10000 (60%)
****************** Época: 11 **********************
Train set: Avg. loss: 0.9925
Validation set: Avg. loss: 1.1352, Accuracy: 6050/10000 (60%)
Test set: Avg. loss: 1.1231, Accuracy: 6070/10000 (61%)
****************** Época: 12 **********************
Train set: Avg. loss: 0.9651
Validation set: Avg. loss: 1.1287, Accuracy: 6073/10000 (61%)
Test set: Avg. loss: 1.1188, Accuracy: 6111/10000 (61%)
****************** Época: 13 **********************
Train set: Avg. loss: 0.9446
Validation set: Avg. loss: 1.1258, Accuracy: 6080/10000 (61%)
Test set: Avg. loss: 1.1172, Accuracy: 6138/10000 (61%)
****************** Época: 14 **********************
Train set: Avg. loss: 0.9167
Validation set: Avg. loss: 1.1176, Accuracy: 6128/10000 (61%)
Test set: Avg. loss: 1.1087, Accuracy: 6151/10000 (62%)
```

```
****************** Época: 15 **********************

Train set: Avg. loss: 0.9004

Validation set: Avg. loss: 1.1223, Accuracy: 6123/10000 (61%)

Test set: Avg. loss: 1.1126, Accuracy: 6178/10000 (62%)

****************** Época: 16 **********************

Train set: Avg. loss: 0.8883

Validation set: Avg. loss: 1.1272, Accuracy: 6115/10000 (61%)

Test set: Avg. loss: 1.1179, Accuracy: 6178/10000 (62%)

****************** Época: 17 **********************

Train set: Avg. loss: 0.8629
Mejora estancada por 3 épocas (1.117644 --> 1.123808).  Activando early stop ...

Validation set: Avg. loss: 1.1238, Accuracy: 6165/10000 (62%)

Test set: Avg. loss: 1.1150, Accuracy: 6182/10000 (62%)

****************** Época: 18 **********************

Train set: Avg. loss: 0.8565

Validation set: Avg. loss: 1.1331, Accuracy: 6124/10000 (61%)

Test set: Avg. loss: 1.1229, Accuracy: 6173/10000 (62%)

****************** Época: 19 **********************

Train set: Avg. loss: 0.8389

Validation set: Avg. loss: 1.1410, Accuracy: 6155/10000 (62%)

Test set: Avg. loss: 1.1307, Accuracy: 6178/10000 (62%)

****************** Época: 20 **********************

Train set: Avg. loss: 0.8248

Validation set: Avg. loss: 1.1473, Accuracy: 6129/10000 (61%)
```

```
Test set: Avg. loss: 1.1376, Accuracy: 6196/10000 (62%)

********** Época: 21 **********

Train set: Avg. loss: 0.8293

Validation set: Avg. loss: 1.1703, Accuracy: 6093/10000 (61%)

Test set: Avg. loss: 1.1602, Accuracy: 6133/10000 (61%)

********** Época: 22 **********

Train set: Avg. loss: 0.8192

Validation set: Avg. loss: 1.1863, Accuracy: 6066/10000 (61%)

Test set: Avg. loss: 1.1740, Accuracy: 6125/10000 (61%)

********** Época: 23 **********

Train set: Avg. loss: 0.8035

Validation set: Avg. loss: 1.2023, Accuracy: 6079/10000 (61%)

Test set: Avg. loss: 1.1873, Accuracy: 6132/10000 (61%)

********** Época: 24 **********

Train set: Avg. loss: 0.7852

Validation set: Avg. loss: 1.2097, Accuracy: 6105/10000 (61%)

Test set: Avg. loss: 1.1944, Accuracy: 6137/10000 (61%)

********** Época: 25 **********

Train set: Avg. loss: 0.7927

Validation set: Avg. loss: 1.2385, Accuracy: 6056/10000 (61%)

Test set: Avg. loss: 1.2216, Accuracy: 6090/10000 (61%)

********** Época: 26 **********

Train set: Avg. loss: 0.7870

Validation set: Avg. loss: 1.2566, Accuracy: 6017/10000 (60%)
```

```
Test set: Avg. loss: 1.2422, Accuracy: 6075/10000 (61%)

********** Época: 27 **********

Train set: Avg. loss: 0.7870

Validation set: Avg. loss: 1.2771, Accuracy: 5995/10000 (60%)

Test set: Avg. loss: 1.2615, Accuracy: 6074/10000 (61%)

********** Época: 28 **********

Train set: Avg. loss: 0.7795

Validation set: Avg. loss: 1.2933, Accuracy: 6010/10000 (60%)

Test set: Avg. loss: 1.2807, Accuracy: 6061/10000 (61%)

********** Época: 29 **********

Train set: Avg. loss: 0.7659

Validation set: Avg. loss: 1.3058, Accuracy: 6012/10000 (60%)

Test set: Avg. loss: 1.2949, Accuracy: 6049/10000 (60%)

********** Época: 30 **********

Train set: Avg. loss: 0.7618

Validation set: Avg. loss: 1.3366, Accuracy: 6027/10000 (60%)

Test set: Avg. loss: 1.3237, Accuracy: 6036/10000 (60%)

********** Época: 31 **********

Train set: Avg. loss: 0.7828

Validation set: Avg. loss: 1.3739, Accuracy: 5954/10000 (60%)

Test set: Avg. loss: 1.3562, Accuracy: 5979/10000 (60%)

********** Época: 32 **********

Train set: Avg. loss: 0.7662

Validation set: Avg. loss: 1.3895, Accuracy: 5966/10000 (60%)
```

```
Test set: Avg. loss: 1.3733, Accuracy: 5993/10000 (60%)

********** Época: 33 **********

Train set: Avg. loss: 0.7308

Validation set: Avg. loss: 1.3978, Accuracy: 6011/10000 (60%)

Test set: Avg. loss: 1.3821, Accuracy: 5992/10000 (60%)

********** Época: 34 **********

Train set: Avg. loss: 0.7075

Validation set: Avg. loss: 1.4097, Accuracy: 5985/10000 (60%)

Test set: Avg. loss: 1.3891, Accuracy: 6005/10000 (60%)

********** Época: 35 **********

Train set: Avg. loss: 0.7164

Validation set: Avg. loss: 1.4386, Accuracy: 5938/10000 (59%)

Test set: Avg. loss: 1.4165, Accuracy: 5936/10000 (59%)

********** Época: 36 **********

Train set: Avg. loss: 0.7200

Validation set: Avg. loss: 1.4764, Accuracy: 5943/10000 (59%)

Test set: Avg. loss: 1.4548, Accuracy: 5911/10000 (59%)

********** Época: 37 **********

Train set: Avg. loss: 0.7755

Validation set: Avg. loss: 1.5485, Accuracy: 5879/10000 (59%)

Test set: Avg. loss: 1.5234, Accuracy: 5851/10000 (59%)

********** Época: 38 **********

Train set: Avg. loss: 0.7798

Validation set: Avg. loss: 1.5916, Accuracy: 5830/10000 (58%)
```

```
Test set: Avg. loss: 1.5619, Accuracy: 5835/10000 (58%)

********** Época: 39 **********

Train set: Avg. loss: 0.7870

Validation set: Avg. loss: 1.6189, Accuracy: 5809/10000 (58%)

Test set: Avg. loss: 1.5953, Accuracy: 5800/10000 (58%)

********** Época: 40 **********

Train set: Avg. loss: 0.7738

Validation set: Avg. loss: 1.6376, Accuracy: 5813/10000 (58%)

Test set: Avg. loss: 1.6146, Accuracy: 5787/10000 (58%)
```

### 1.6 Evaluando el modelo

[32]:
```
if (SGD_early_stopper.early_stop):
  print(f"Ha habido una parada temprana en el modelo de SGD con un coste de␣
 ↪{SGD_early_stopper.val_loss_min}\nÉpoca número:{SGD_early_stopper.
 ↪stop_iter}\n")
else:
  print(f"No ha habido early stop en SGD, siendo el menor coste:␣
 ↪{SGD_early_stopper.val_loss_min}\n")

if (SGDM_early_stopper.early_stop):
  print(f"Ha habido una parada temprana en el modelo de SGD con momento con␣
 ↪coste de {SGDM_early_stopper.val_loss_min}\nÉpoca número:{SGDM_early_stopper.
 ↪stop_iter}\n")
else:
  print(f"No ha habido early stop en SGD con momento, siendo el menor coste:␣
 ↪{SGDM_early_stopper.val_loss_min}\n")

if (ADAM_early_stopper.early_stop):
  print(f"Ha habido una parada temprana en el modelo de ADAM con coste de␣
 ↪{ADAM_early_stopper.val_loss_min}\nÉpoca número:{ADAM_early_stopper.
 ↪stop_iter}\n")
else:
  print(f"No ha habido early stop en ADAM, siendo el menor coste:␣
 ↪{ADAM_early_stopper.val_loss_min}\n")
```

```
No ha habido early stop en SGD, siendo el menor coste: 1.4608945846557617
```

```
Ha habido una parada temprana en el modelo de SGD con momento con coste de
1.1873843669891357
Época número:33

Ha habido una parada temprana en el modelo de ADAM con coste de
1.117644190788269
Época número:14
```

# Glosario

Álgoritmo de aprendizaje, 9

Adagrad, 41
ADAM, 42
Algoritmo de optimización, 26
Aprendizaje automático, 9
Aprendizaje no supervisado, 10
Aprendizaje por refuerzo, 10
Aprendizaje supervisado, 10

Backpropagation, 27

Conjunto de datos de validación, 55
Convolución, 37

Deep Learning, 17

Early stopping, 54
Entrenamiento, 9

Función de coste, 24

Google Colaboratory, 45
Gradiente descendente, 26
Gradiente descendente estocástico con momento, 39

Hiperparámetros, 28

Inteligencia artificial, 8

Objetivo del proyecto, 3

Peso, 24
Pooling, 37
Precisión de un modelo, 32
Puntos de silla, 38
Pytorch, 43

Redes neuronales artificiales, 11
Redes neuronales convolucionales, 37
Regularización, 55
RMSProp, 41

Sesgo, 24
Sobreajuste, 54

Tasa de aprendizaje, 26
Tensorboard, 45