

**UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA**  
**SEDE VIÑA DEL MAR – JOSÉ MIGUEL CARRERA**

**ESTUDIO E IMPLEMENTACIÓN DE DEEP LEARNING CON PYTHON**

Trabajo de Titulación para optar al  
Título de Ingeniero de Ejecución en  
CONTROL E INSTRUMENTACIÓN  
INDUSTRIAL

Alumno:

Fernando Andrés Constanzo Garcés

Profesora Guía:

Mag. Guelis Montenegro Zamora.



## CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

### 1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción):  Memoria o trabajo de título  Tesis de Postgrado

Título del trabajo: Estudio e implementación de deep learning con python.

Nombre del candidato(a): Fernando Andrés Constanza Garcés

Carrera / Grado: Ingeniería de ejecución en control e instrumentación industrial.

Campus: Sede Viña del Mar Departamento: Electrotecnia e informática

### 2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Guelis Montenegro Zamora, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

### 3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (marcar una opción):

6 meses  12 meses  2 años  3 años  5 años  10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

---

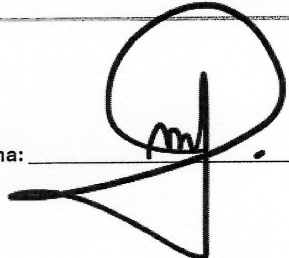
---

---

### 4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 07-04-2026

Firma: 

Estudiante o Candidato(a):

Fecha: 06-04-2026

Firma: 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

## RESUMEN

**KEYWORDS:** Inteligencia Artificial, Machine Learning, Deep Learning, Redes neuronales, Python.

En el presente trabajo de título, titulado *“Estudio e implementación de inteligencia artificial con Python”*, se abordará el estudio teórico y práctico de los principales conceptos que conforman el campo de la inteligencia artificial (IA). El objetivo es revisar y comprender los métodos y subcampos que integran esta disciplina, destacando entre ellos: Machine Learning, Análisis Predictivo, Visión Artificial, Procesamiento del Lenguaje Natural, Computación Evolutiva, Aprendizaje por Transferencia, Aprendizaje por Refuerzo, Sistemas Expertos, Deep Learning y Reconocimiento de Voz.

En una primera etapa, se definirán y relacionarán estos conceptos, permitiendo al lector tener una visión general del ecosistema de la inteligencia artificial. A continuación, se profundizará especialmente en el subcampo del deep learning, desarrollando ejercicios prácticos de entrenamiento, tanto de forma manual como mediante su implementación en Python, utilizando un ejemplo que facilite su comprensión.

Finalmente, se llevará a cabo un proyecto aplicado que empleará los conocimientos adquiridos sobre las redes neuronales artificiales. Este incluirá la evaluación de su rendimiento y aplicabilidad en un caso práctico, analizando además las fortalezas del sistema implementado y proponiendo posibles mejoras para su optimización.

## ÍNDICE

<b>INTRODUCCIÓN.....</b>	<b>1</b>
<b>CAPÍTULO 1: ANTECEDENTES GENERALES .....</b>	<b>2</b>
<b>1 ANTECEDENTES GENERALES .....</b>	<b>3</b>
<b>1.1 PLANTEAMIENTO DEL PROBLEMA.....</b>	<b>3</b>
<b>1.2 REQUERIMIENTOS .....</b>	<b>3</b>
1.2.1 Requerimientos teóricos .....	3
1.2.2 Requerimientos prácticos.....	4
1.2.3 Requerimientos tecnológicos.....	4
<b>1.3 ESTADO DEL ARTE DE LA INTELIGENCIA ARTIFICIAL .....</b>	<b>4</b>
1.3.1 Inteligencia artificial .....	6
1.3.2 Machine learning.....	6
1.3.2.1 Aprendizaje supervisado.....	7
1.3.2.2 Aprendizaje no supervisado .....	7
1.3.2.3 Aprendizaje semi supervisado .....	8
1.3.2.4 Aprendizaje por transferencia .....	8
1.3.3 Análisis predictivo.....	8
1.3.3.1 Modelo predictivo .....	9
1.3.3.2 Series temporales .....	9
1.3.4 Reinforcement learning.....	9
1.3.4.1 Aprendizaje por refuerzo.....	10
1.3.4.2 Q-learning.....	10
1.3.4.3 Política de gradiente.....	10

1.3.5	Sistemas expertos.....	11
1.3.5.1	Sistemas basados en reglas .....	11
1.3.5.2	Sistemas basados en casos .....	11
1.3.6	Deep learning .....	12
1.3.6.1	Redes neuronales profundas.....	12
1.3.6.2	Redes neuronales convolucionales .....	13
1.3.6.3	Redes neuronales recurrentes.....	13
1.3.7	Procesamiento del lenguaje natural .....	13
1.3.7.1	Reconocimiento de voz .....	14
1.3.7.2	Traducción automática .....	14
1.3.7.3	Modelado del lenguaje .....	14
1.3.8	Visión artificial .....	14
1.3.8.1	Reconocimiento de imágenes .....	15
1.3.8.2	Detección de objetos .....	15
1.3.8.3	Segmentación de imágenes.....	16
1.3.9	Computación evolutiva.....	16
1.3.9.1	Algoritmos genéticos .....	17
1.3.9.2	Programación genética .....	17
1.3.9.3	Optimización evolutiva .....	18
<b>1.4</b>	<b>REFLEXIÓN GENERAL .....</b>	<b>18</b>
<b>1.5</b>	<b>OBJETIVOS.....</b>	<b>19</b>
1.5.1	objetivo general.....	19
1.5.2	objetivos específicos.....	19

<b>CAPÍTULO 2: DESARROLLO DEL PROYECTO.....</b>	<b>21</b>
<b>2 DESARROLLO DEL PROYECTO .....</b>	<b>22</b>
<b>2.1 HISTORIA Y CONTEXTO DE LAS REDES NEURONALES .....</b>	<b>22</b>
<b>2.2 FUNCIONAMIENTO GENERAL DE UNA RED NEURONAL BIOLÓGICA .....</b>	<b>23</b>
2.2.1 Dendritas .....	23
2.2.2 Cuerpo celular o soma .....	23
2.2.3 Axón .....	24
<b>2.3 FUNCIONAMIENTO GENERAL DE UNA RED NEURONAL ARTIFICIAL .....</b>	<b>25</b>
2.3.1 Neurona artificial .....	26
2.3.2 Capas.....	27
2.3.2.1 Capa de entrada.....	27
2.3.2.2 Capas ocultas .....	27
2.3.2.3 Capa de salida .....	27
2.3.3 Pesos y sesgos.....	28
2.3.4 Funciones de activación .....	28
2.3.4.1 Relu .....	28
2.3.4.2 Sigmoide .....	29
2.3.4.3 Tanh .....	29
2.3.5 Funciones de pérdida .....	30
2.3.5.1 Error cuadrático medio.....	30
2.3.5.2 Entropía cruzada.....	31
2.3.6 Algoritmo de optimización .....	32
<b>2.4 COMPARACIÓN ENTRE UNA NEURONA BIOLÓGICA Y UNA NEURONA ARTIFICIAL... 32</b>	

<b>2.5</b>	<b>TIPOS DE REDES NEURONALES.....</b>	<b>34</b>
2.5.1	Redes neuronales artificiales (ANN).....	35
2.5.2	Redes neuronales convolucionales (CNN).....	35
2.5.3	Redes neuronales recurrentes (RNN).....	35
2.5.4	Lstm y gru .....	36
2.5.5	Transformers .....	36
2.5.6	Autoencoders .....	37
<b>2.6</b>	<b>ENTRENAMIENTO DE UNA RED NEURONAL.....</b>	<b>37</b>
2.6.1	Propagación hacia adelante .....	38
2.6.2	Cálculo del error .....	39
2.6.3	Propagación hacia atrás .....	39
2.6.4	Actualización de los pesos.....	39
2.6.5	Épocas.....	40
<b>2.7</b>	<b>PYTHON .....</b>	<b>40</b>
<b>2.8</b>	<b>EJERCICIO PRÁCTICO.....</b>	<b>42</b>
2.8.1	Datos iniciales.....	43
2.8.2	Función de activación.....	45
2.8.3	Propagación hacia adelante .....	45
2.8.4	Función de pérdida.....	47
2.8.5	Propagación hacia atrás .....	48
2.8.5.1	Gradiente de la pérdida con respecto a la salida de la capa de salida.....	49
2.8.5.2	Gradiente de la salida de la capa de salida con respecto a su entrada $z^{(2)}$ .....	50
2.8.5.3	Gradiente de la pérdida con respecto a la entrada de la capa de salida $z^{(2)}$ .....	51

2.8.5.4	Gradiente de la pérdida con respecto a los pesos de la capa de salida $W_n^{(2)}$ .....	52
2.8.5.5	Gradiente de la pérdida con respecto al sesgo de la capa de salida $b^{(2)}$ .....	53
2.8.5.6	Gradiente de la entrada de la capa de salida $z^{(2)}$ con respecto a la salida de la capa oculta $a_n^{(1)}$ .....	54
2.8.5.7	Gradiente de la salida de la capa oculta $a_n^{(1)}$ con respecto a su entrada $z_n^{(1)}$ .....	55
2.8.5.8	Gradiente de la pérdida con respecto a la entrada de la capa oculta $z_n^{(1)}$ .....	56
2.8.5.9	Gradiente de la pérdida con respecto a los pesos de la capa de entrada $W_n^{(1)}$ .....	57
2.8.5.10	Gradiente de la pérdida con respecto a los sesgos de la capa oculta $b_n^{(1)}$ .....	58
2.8.6	Actualización de parámetros mediante descenso de gradiente.....	60
2.8.6.1	Actualización de pesos de la capa de salida .....	61
2.8.6.2	Actualización del sesgo de la capa de salida .....	61
2.8.6.3	Actualización de pesos de la capa oculta .....	62
2.8.6.4	Actualización de los sesgos de la capa oculta.....	63
2.8.6.5	Nueva propagación hacia adelante.....	63
2.8.6.6	Nueva función de pérdida .....	64
2.8.6.7	Comparación de resultados .....	65
<b>2.9</b>	<b>EJERCICIO COMPUTACIONAL .....</b>	<b>66</b>
2.9.1	Bibliotecas.....	66
2.9.2	Función de activación sigmoide y su derivada .....	67
2.9.3	Datos iniciales de entrenamiento.....	68
2.9.4	Pesos y sesgos iniciales.....	69
2.9.5	Parámetros de entrenamiento.....	70

2.9.6	Ciclo de entrenamiento .....	70
2.9.7	Cálculo de la pérdida .....	71
2.9.8	Propagación hacia atrás .....	72
2.9.9	Actualización de pesos y sesgos .....	74
2.9.10	Prueba del modelo entrenado .....	75
2.9.11	Resultados obtenidos del entrenamiento.....	77
2.9.12	Red neuronal funcional .....	78
2.9.13	Gráfica de la función de pérdida .....	79
2.9.14	Evolución de la pérdida durante el entrenamiento .....	80
<b>2.10</b>	<b>REFLEXIÓN FINAL DEL SEGUNDO CAPÍTULO .....</b>	<b>82</b>
	<b>CAPÍTULO 3: PUESTA EN MARCHA DEL PROYECTO .....</b>	<b>84</b>
<b>3</b>	<b>PUESTA EN MARCHA DEL PROYECTO .....</b>	<b>85</b>
<b>3.1</b>	<b>RECOLECCIÓN Y PREPARACIÓN DE DATOS .....</b>	<b>85</b>
3.1.1	Normalización.....	85
3.1.2	Codificación one-hot de las etiquetas .....	86
<b>3.2</b>	<b>ARQUITECTURA DEL MODELO.....</b>	<b>86</b>
3.2.1	Capa de entrada .....	87
3.2.2	Primera capa convolucional .....	87
3.2.3	Primera capa de submuestreo .....	88
3.2.4	Segunda capa convolucional .....	89
3.2.5	Segunda capa de submuestreo .....	89
3.2.6	Capa de aplanamiento.....	90
3.2.7	Capas densas .....	90

3.2.8	Capa de salida.....	91
<b>3.3</b>	<b>CONSIDERACIONES PREVIAS AL ENTRENAMIENTO.....</b>	<b>94</b>
3.3.1	Función de pérdida.....	94
3.3.2	Optimizador.....	95
3.3.3	Métricas de evaluación.....	96
3.3.4	Hiperparámetros.....	97
3.3.5	Épocas.....	97
3.3.6	Batch size.....	98
3.3.7	División de datos para validación.....	99
3.3.8	Regularización.....	99
<b>3.4</b>	<b>IMPLEMENTACIÓN PRÁCTICA DE UNA RED NEURONAL CONVOLUCIONAL.....</b>	<b>99</b>
3.4.1	Importación de librerías.....	100
3.4.2	Carga y procesamiento de datos.....	102
3.4.3	Construcción del modelo CNN.....	103
3.4.4	Compilación y entrenamiento.....	105
3.4.5	Evaluación y guardado del modelo.....	106
3.4.6	Importación de librerías y carga del modelo entrenado.....	107
3.4.7	Configuración de la ventana y superficie de trabajo.....	108
3.4.8	Función para dibujar.....	110
3.4.9	Limpieza de lienzo.....	111
3.4.10	Función para predecir el número dibujado.....	112
3.4.11	Botones.....	114
<b>3.5</b>	<b>INTERFAZ GRÁFICA.....</b>	<b>115</b>

3.5.1	Funciones principales de la interfaz gráfica .....	115
<b>3.6</b>	<b>EVALUACIÓN DEL MODELO.....</b>	<b>116</b>
3.6.1	Curva de pérdida .....	116
3.6.2	Curva de precisión (accuracy).....	118
3.6.3	Matriz de confusión.....	119
<b>3.7</b>	<b>VISUALIZACIÓN DE RESULTADOS .....</b>	<b>120</b>
3.7.1	Predicciones correctas.....	121
3.7.2	Predicciones incorrectas.....	121
3.7.3	Análisis general.....	122
	<b>CONCLUSIÓN.....</b>	<b>124</b>
	<b>BIBLIOGRAFÍA. ....</b>	<b>126</b>
	<b>ANEXOS .....</b>	<b>127</b>

## ÍNDICE DE FIGURAS

Figura 1-1	Estado del arte de la inteligencia artificial .....	6
Figura 2-1	Evolución histórica de las redes neuronales artificiales .....	23
Figura 2-2	Neurona biológica .....	24
Figura 2-3	Estructura básica de una red neuronal artificial .....	25
Figura 2-4	Esquema de una neurona artificial .....	26
Figura 2-5	Gráficos de las funciones de activación relu, sigmoide y tanh .....	30
Figura 2-6	Gráficos de las funciones pérdidas.....	32
Figura 2-7	Esquema de comparación entre una neurona biológica y artificial .....	34
Figura 2-8	Esquema del proceso de entrenamiento de una RNA.....	38
Figura 2-9	Iconos de herramientas de desarrollo para rna en Python .....	42
Figura 2-10	Estructura de la red neuronal requerida .....	45
Figura 2-11	Propagación hacia delante.....	47
Figura 2-12	Retropropagación de una red neuronal artificial .....	60
Figura 2-13	Bibliotecas .....	66
Figura 2-14	Función de activación sigmoide y su derivada.....	67
Figura 2-15	Datos iniciales de entrenamiento .....	68
Figura 2-16	Pesos y sesgos iniciales .....	69
Figura 2-17	Parámetros de entrenamiento .....	70
Figura 2-18	Ciclo de entrenamiento .....	71
Figura 2-19	Cálculo de la pérdida.....	72
Figura 2-20	Propagación hacia atrás.....	73

Figura 2-21	Actualización de pesos y sesgos.....	75
Figura 2-22	Prueba del modelo entrenado.....	76
Figura 2-23	Resultados obtenidos en el entrenamiento .....	77
Figura 2-24	Red neuronal funcional.....	79
Figura 2-25	Gráfica de la función de pérdida.....	79
Figura 2-26	Evolución de la pérdida durante el entrenamiento.....	80
Figura 3-1	Estructura general de una red neuronal convolucional.....	87
Figura 3-2	Ejemplo de conjunto MNIST .....	100
Figura 3-3	Importación de librerías.....	101
Figura 3-4	Carga y procesamiento de datos.....	102
Figura 3-5	Construcción del modelo CNN .....	103
Figura 3-6	Compilación y entrenamiento.....	105
Figura 3-7	Evaluación y guardado del modelo .....	107
Figura 3-8	Importación de librerías y carga del modelo entrenado .....	108
Figura 3-9	Configuración de la ventana de trabajo.....	109
Figura 3-10	Función para dibujar .....	110
Figura 3-11	Limpieza de lienzo.....	111
Figura 3-12	Función para predecir el numero dibujado .....	112
Figura 3-13	Limpieza de lienzo.....	114
Figura 3-14	Interfaz gráfica .....	115
Figura 3-15	Curva de pérdida.....	117
Figura 3-16	Curva de precisión.....	118
Figura 3-17	Matriz de confusión .....	119

Figura 3-18	Predicciones correctas .....	121
Figura 3-19	Predicciones incorrectas .....	122

## ÍNDICE DE TABLAS

Tabla 2-1	Resultados de las predicciones del modelo.....	78
Tabla 3-1	Exponenciales de los logits antes de aplicar la función de activación.....	93

## SIGLAS Y SIMBOLOGÍA

### A. SIGLAS:

ADAM:	Adaptive Moment Estimation (Estimación adaptativa de momentos).
BERT:	Bidirectional Encoder Representation from Transformers (Representaciones de codificador bidireccional a partir de transformers).
CNN:	Convolutional Neural Network (Red neuronal convolucional).
DNN:	Deep Neural Network (Red neuronal profunda).
ECM:	Error Cuadrático Medio.
GB:	Giga Byte.
GPT:	Generative Pre-trained Transformer (Transformador generativo preentrenado).
GRU:	Gated Recurrent Unit (Unidad recurrente cerrada).
IA:	Inteligencia Artificial.
LIDAR:	Light Detection and Ranging (Sensor óptico de detección por luz y distancia).
LSTM:	Long Short-Term Memory (Memoria a corto-largo plazo).
MNIST:	Modified National Institute of Standard and Technology.
MSE:	Mean Squared Error (Error cuadrático medio).
PIL:	Python Imaging Library (librería para manipulación de imágenes).
RAM:	Random Access Memory (Memoria de acceso aleatorio).
ReLU:	Rectified Linear Unit (Unidad lineal rectificada).
ResNet:	Residual Network (Arquitectura de redes neuronales profundas con conexiones residuales).
RNA:	Red Neuronal Artificial.
RNN:	Recurrent Neural Network (Red neuronal recurrente).

## **B. SIMBOLOGÍA:**

- a: Activación.
- b: Sesgo.
- f: Función.
- k: Número de clases, en clasificación.
- L: Función de pérdida (loss).
- n: Número de muestras.
- W: Pesos.
- X: Entrada.
- y: Salida esperada.
- z: Suma ponderada de neurona.
- $\partial$ : Derivada.
- $\Sigma$ : Sumatoria.

## **INTRODUCCIÓN**

La inteligencia Artificial (IA) ha revolucionado diversos sectores, desde la economía y la medicina, hasta el entretenimiento y el transporte, posicionándose como una de las tecnologías más influyentes de la actualidad.

En este trabajo de título se plantea explorar en profundidad los subcampos y métodos de la inteligencia artificial junto con una aplicación práctica de Deep Learning utilizando el lenguaje de programación Python.

El desarrollo comenzará con una revisión del estado del arte de la inteligencia artificial, abordando sus fundamentos teóricos. Posteriormente, se avanzará hacia la implementación práctica, enfocándose en el uso de redes neuronales artificiales, lo que permitirá demostrar cómo los conceptos aprendidos pueden aplicarse en proyectos reales. Python será la herramienta central a lo largo de este trabajo, facilitando la construcción, entrenamiento y evaluación de modelos de IA.

## **CAPÍTULO 1: ANTECEDENTES GENERALES**

## **1 ANTECEDENTES GENERALES**

En este capítulo se plantea la problemática a tratar en el trabajo de título, brindando un contexto, también se explicará cómo se resolverá, además de establecer los objetivos.

### **1.1 PLANTEAMIENTO DEL PROBLEMA**

El rápido crecimiento de la inteligencia artificial (IA) ha dado lugar a diversos subcampos como la visión artificial, el procesamiento del lenguaje natural y la computación evolutiva, todos impulsados por métodos de machine learning como el aprendizaje supervisado, no supervisado, por refuerzo, por transferencia entre otros.

No obstante, la gran cantidad de conceptos y su complejidad técnica dificultan su entendimiento y aplicación, especialmente para quienes están iniciando en el área. Este trabajo aborda dicha problemática mediante un análisis claro y estructurado del estado del arte de la inteligencia artificial, reforzado con ejercicios prácticos de entrenamiento de redes neuronales artificiales, tanto de forma manual como utilizando Python. Finalmente se implementa un caso aplicado que analiza la escritura de números naturales como ejemplo de uso avanzado de redes neuronales.

### **1.2 REQUERIMIENTOS**

Antes de avanzar en el desarrollo del proyecto, es importante contar con ciertos conocimientos previos que permitan comprender de mejor manera los conceptos que se abordarán. A continuación, se detallan los principales requerimientos desde una perspectiva teórica, práctica y tecnológica.

#### **1.2.1 Requerimientos teóricos**

Es ideal tener una comprensión básica de los fundamentos de la inteligencia artificial, así como de los principales métodos de aprendizaje automático, como el aprendizaje supervisado, no

supervisado, semi supervisado y por transferencia. Además, se requiere manejo básico de matemáticas aplicadas, especialmente álgebra lineal (vectores y matrices) y cálculo diferencial (derivadas parciales y gradientes), ya que estos conocimientos son esenciales para entender cómo funcionan los algoritmos de IA y las redes neuronales artificiales.

### 1.2.2 Requerimientos prácticos

Se recomienda tener nociones básicas de lógica de programación y experiencia previa en programación estructurada o modular, preferentemente en Python, debido a su amplio uso en inteligencia artificial. Python ofrece diversas bibliotecas útiles como NumPy (para operaciones matemáticas), Pandas (para manejo de datos), Matplotlib (para visualización) y especialmente TensorFlow y Keras, que facilitan la construcción y entrenamiento de redes neuronales, además de otras bibliotecas ampliamente utilizadas como PyTorch, Scikit-learn especiales para modelos clásicos de machine learning y OpenCV en aplicaciones de visión por computador. Estas bibliotecas permiten implementar, entrenar y evaluar modelos de manera eficiente, según los requerimientos del proyecto.

### 1.2.3 Requerimientos tecnológicos

En cuanto a los recursos técnicos, es recomendable contar con un equipo con buen rendimiento, al menos 8 GB de RAM si se planea trabajar con redes neuronales profundas, además de suficiente espacio de almacenamiento para datasets de tamaño mediano. El entorno de desarrollo utilizado será PyCharm, junto conexión a internet para acceder a bibliotecas, documentación oficial y descargar conjuntos de datos como MNIST.

## 1.3 ESTADO DEL ARTE DE LA INTELIGENCIA ARTIFICIAL

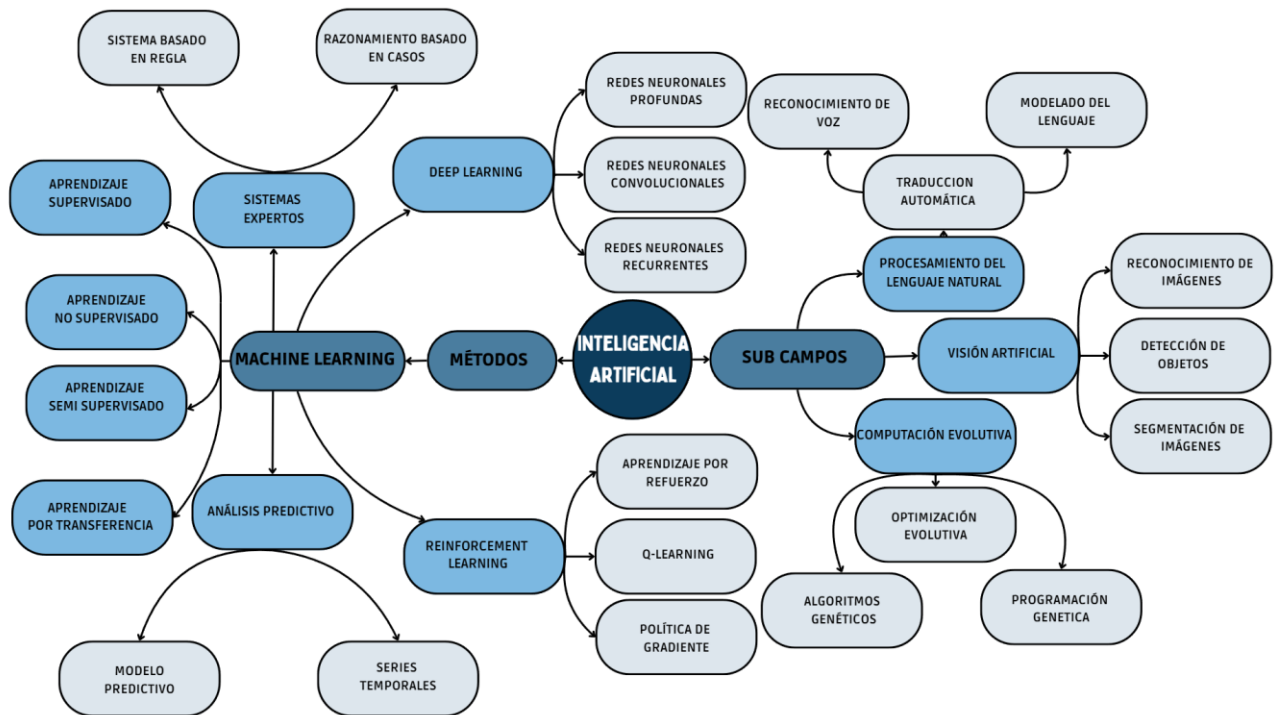
El estado del arte de la inteligencia artificial (IA) es altamente dinámico y se encuentra en constante evolución, impulsado por avances en algoritmos, mayor capacidad de procesamiento y

una creciente disponibilidad de datos. En la actualidad, sus aplicaciones están transformando múltiples industrias y aspectos de la vida cotidiana. Entre los desarrollos más destacados se encuentran:

- **Aprendizaje profundo (Deep Learning):** Las redes neuronales profundas, incluyendo redes convolucionales (CNN) y recurrentes (RNN), han demostrado un rendimiento sobresaliente en tareas como el reconocimiento de imágenes, el procesamiento del lenguaje natural y el reconocimiento de voz.
- **Medicina:** La IA ha revolucionado el diagnóstico médico, especialmente mediante el análisis de imágenes como radiografías y resonancias magnéticas. Además, ha optimizado el desarrollo de nuevos fármacos al acelerar la identificación de compuesto viables.
- **Conducción autónoma:** Empresas como Tesla, Waymo (filial de Google) y Cruise (subsidiaria de General Motors) han desarrollado vehículos que utilizan IA para interpretar el entorno mediante cámaras y sensores LIDAR, tomando decisiones en tiempo real mediante algoritmos de percepción, planificación y control.
- **Finanzas:** El trading algorítmico ha sido potenciado por la IA gracias a su capacidad para operar con rapidez y precisión. Asimismo, se ha implementado en la detección de fraudes, mediante sistemas que identifican patrones anómalos en tiempo real.
- **Ética:** se han dado avances importantes en torno a la privacidad de los datos, abordando como se recopilan, almacenan y utilizan. Este aspecto es clave para garantizar el desarrollo responsable de tecnologías basadas en IA.
- **Impacto social:** La automatización y el uso de sistemas inteligentes plantean desafíos en torno al empleo, la economía y el rol de los humanos en los procesos productivos, lo cual motiva debates sobre el futuro del trabajo.

El futuro de la inteligencia artificial es prometedor. Se proyecta que esta tecnología será capaz de realizar muchas tareas cognitivas propias del ser humano, mejorando a su vez la interacción hombre-máquina y optimizando continuamente sus algoritmos.

A continuación, en la figura 1-1, se presenta una visión general del estado del arte de la inteligencia artificial. Posteriormente, se analizarán en detalle sus principales subcampos y Métodos.



Fuente: Elaboración propia en Canva

Figura 1-1 Estado del arte de la inteligencia artificial.

### 1.3.1 Inteligencia artificial

La inteligencia artificial puede entenderse a través de dos aspectos fundamentales:

- **Métodos:** Corresponden a las herramientas, algoritmos o técnicas utilizadas para el desarrollo de soluciones basadas en IA. Es decir, representan “*el cómo se hacen*”.
- **Subcampos:** Son las áreas de aplicación donde se implementan estos métodos para resolver problemas específicos. Representan “*el para qué se hacen*”.

Esta distinción permite organizar el estudio de la IA en torno a sus fundamentos técnicos y sus aplicaciones prácticas.

### 1.3.2 Machine learning

También conocido como aprendizaje automático, es una rama de la inteligencia artificial enfocado en el desarrollo de algoritmos capaces de aprender a partir de datos. Esta característica permite que los sistemas puedan mejorar su rendimiento sin necesidad de ser programados

explícitamente para cada tarea. En lugar de depender de reglas fijas definidas por un programador, los modelos aprenden a partir de grandes volúmenes de información.

Dentro de este campo, se distinguen varias técnicas fundamentales:

#### 1.3.2.1 Aprendizaje supervisado

En el aprendizaje supervisado, el modelo es entrenado con un conjunto de datos que incluye tanto las entradas (características) como las salidas deseadas (etiquetas). Es decir, se le entrega al algoritmo una serie de ejemplos con las respuestas correctas, permitiéndole aprender la relación entre los datos y sus resultados.

Ejemplo: predecir el precio de una casa en base a sus características como tamaño, número de habitaciones, año de construcción, ubicación, entre otros. El modelo aprende a predecir el valor a partir de datos históricos.

Dentro del aprendizaje supervisado, existen dos grandes tipos de problemas:

- Clasificación: El modelo agrupa los datos en categorías. Por ejemplo, clasificar correos como “spam” o “no spam”, o distinguir entre diferentes especies de animales.
- Regresión: El modelo predice valores continuos. Un ejemplo, estimar el precio de una casa, como se mencionó anteriormente.

#### 1.3.2.2 Aprendizaje no supervisado

En este enfoque, el algoritmo trabaja únicamente con las características, sin etiquetas asociadas, el objetivo es encontrar patrones ocultos o estructuras en los datos, como grupos.

Ejemplo: Clasificar especies animales basándose solo en características como número de patas, pesos, temperatura corporal o presencia de alas, sin conocer realmente a que especie pertenecen. El algoritmo, al identificar similitudes entre las características, agrupa los datos de manera automática.

### 1.3.2.3 Aprendizaje semi supervisado

Este tipo de aprendizaje combina elementos de enfoque supervisado y no supervisado. Se utiliza un pequeño conjunto de datos etiquetados junto a una gran cantidad de datos sin etiquetar para entrenar el modelo.

Este enfoque es especialmente útil cuando el etiquetado manual es costoso o difícil de obtener, pero existe una gran disponibilidad de datos. En muchos casos, mejora el rendimiento del modelo reduciendo el esfuerzo necesario en el proceso de etiquetado.

Ejemplo: Un modelo puede entrenarse con unos pocos documentos legales ya clasificados como contratos, resoluciones, demandas, etc. y miles sin clasificar. Con esta pequeña base etiquetada, el modelo aprende y luego identifica patrones en los no etiquetados, mejorando su precisión, sin la necesidad de etiquetar todo manualmente.

### 1.3.2.4 Aprendizaje por transferencia

El aprendizaje por transferencia permite reutilizar el conocimiento adquirido por un modelo en una tarea previa, aplicándolo a una tarea nueva pero relacionada.

Ejemplo: Un modelo previamente entrenado comienza a reconocer objetos en imágenes generales, este puede ser adaptado para identificar enfermedades en radiografías, incluso con una base de datos más limitada. De esta forma, se acelera el entrenamiento y se mejora la precisión sin necesidad de construir el modelo desde cero.

### 1.3.3 Análisis predictivo

La inteligencia artificial predictiva utiliza algoritmos matemáticos y análisis estadísticos para identificar comportamientos, patrones o tendencias con el fin de anticipar eventos futuros. Este tipo de inteligencia permite realizar cálculos de manera más rápida y precisa, especialmente al disponer de grandes volúmenes de datos. Cuanta más información se proporciona, mayor será la precisión de las predicciones. Dentro del análisis predictivo destacan dos herramientas claves: los modelos predictivos y las series temporales.

### 1.3.3.1 Modelo predictivo

También conocido como modelo de predicción, corresponde a un conjunto de técnicas estadísticas orientadas a anticipar un comportamiento antes de que ocurra. Al igual que en el aprendizaje supervisado, sus algoritmos principales son los de clasificación y regresión, pero enfocados en el análisis de escenarios futuros.

- Modelo predictivo de clasificación: se basa en datos históricos para clasificar o categorizar información. Su enfoque suele ser binario, es decir, determina si un elemento pertenece o no a una categoría específica. Es ampliamente utilizado en el ámbito de salud, como apoyo a diagnósticos médicos.
- Modelo predictivo de regresión: permite estimar valores numéricos futuros. Aprende a partir de datos pasados para proyectar comportamientos, siendo comúnmente aplicado en logística, donde se anticipa la demanda de productos.

### 1.3.3.2 Series temporales

Consiste en el análisis de datos recogidos en distintos momentos y ordenados cronológicamente. Este tipo de modelo resulta esencial cuando el tiempo es un factor determinante, ya que permite identificar tendencias y variaciones a lo largo de un período. Para obtener resultados significativos, es necesario contar con una cantidad de datos suficientes. Se aplica principalmente en áreas como la economía, donde se puede proyectar el producto interno bruto o la tasa de desempleo, y en meteorología, en la predicción del clima o patrones climáticos.

### 1.3.4 Reinforcement learning

El aprendizaje por refuerzo (Reinforcement Learning) es una rama del machine learning en la que un agente aprende a tomar decisiones a través de la interacción con un entorno. Este campo se debe subdividir en tres componentes fundamentales: aprendizaje por refuerzo clásico, el algoritmo Q-learning y la política de gradiente.

#### 1.3.4.1 Aprendizaje por refuerzo

En este enfoque, un agente (como un robot o un programa) toma decisiones dentro de un entorno, recibiendo recompensas o castigos según sus acciones. A lo largo del tiempo, el agente aprende qué acciones conducen a mejores resultados, con el objetivo de maximizar la recompensa acumulada. Este método es ideal cuando el agente no conoce inicialmente la mejor acción a tomar, pero puede descubrirlo mediante prueba y error. Una forma de entenderlo es compararlo con entrenar a una mascota: se le permite explorar, y cuando realiza una acción correcta, recibe una recompensa.

#### 1.3.4.2 Q-learning

Q-learning es un algoritmo dentro del aprendizaje por refuerzo que utiliza una tabla de valores (Q-table) para registrar qué tan buena es cada acción en un determinado estado. Cada vez que el agente actúa, actualiza la tabla en base a la recompensa recibida, mejorando su capacidad de decisión futura. Es especialmente eficaz en entornos discretos o estructurados, como laberintos o escenarios simples con un número limitado de acciones posibles.

#### 1.3.4.3 Política de gradiente

A diferencia del Q-learning, la política de gradiente no utiliza tablas, sino que enseña directamente al agente una estrategia de decisión (una política) que define qué acción tomar en cada situación. Este enfoque es más adecuado cuando el espacio de acciones es muy amplio y continuo, como tareas complejas donde no es práctico utilizar tablas, por ejemplo, en el control preciso de un brazo robótico. El agente mejora su política ajustándola gradualmente para maximizar la recompensa esperada con el tiempo.

### 1.3.5 Sistemas expertos

Los modelos expertos son programas de inteligencia artificial diseñados para resolver problemas específicos mediante el uso de conocimiento especializado, emulando el razonamiento de un experto humano. Se utiliza en áreas como la medicina, la ingeniería y las finanzas. Su funcionamiento consiste en una base de conocimiento, donde se almacenan hechos y reglas, y en un motor de inferencia que aplica ese conocimiento para tomar decisiones o resolver problemas. Los enfoques más comunes son los sistemas basados en reglas y casos.

#### 1.3.5.1 Sistemas basados en reglas

En este tipo de sistema, el conocimiento se representa mediante reglas del tipo “*Si... Entonces...*”, que vincula una condición con una conclusión o acción. Por ejemplo: “*Si el paciente presenta fiebre alta y tos, entonces podría tener un resfrío*”. El motor de inferencia aplica estas reglas para analizar esta situación y llegar a una conclusión. Son adecuados para dominios donde el conocimiento puede expresarse de forma clara y lógica, aunque su eficiencia puede verse limitada cuando deben manejar muchas variables o incertidumbre.

#### 1.3.5.2 Sistemas basados en casos

En lugar de utilizar reglas explícitas, estos sistemas almacenan experiencias pasadas en forma de casos (problemas y sus respectivas soluciones), y utilizan esta información para enfrentar nuevos problemas. Por ejemplo, si un mecánico se encuentra con una falla en un vehículo similar a una que ya resolvió, puede aplicar la misma solución adaptándola al nuevo contexto. El sistema compara el caso actual con los almacenados, selecciona el más similar y lo adapta. Este enfoque es útil cuando el conocimiento no puede definirse fácilmente mediante reglas, pero existe una base sólida de experiencias prácticas acumuladas.

### 1.3.6 Deep learning

El Deep learning o aprendizaje profundo es una técnica avanzada dentro del aprendizaje automático (machine learning), que se basa en el uso de redes neuronales para resolver problemas complejos como el reconocimiento de imágenes, la traducción de idiomas o la generación de texto. Su principal característica es el uso de redes neuronales profundas, es decir, redes con múltiples capas ocultas que permiten aprender representaciones jerárquicas y altamente precisas en los datos.

Estas redes están inspiradas en la estructura y funcionamiento del cerebro humano pero diseñadas matemáticamente para procesar información y descubrir patrones o relaciones en grandes volúmenes de datos. El auge del Deep learning ha sido posible gracias al incremento de la capacidad computacional y a la disponibilidad de grandes conjuntos de datos. Este enfoque ha transformado significativamente áreas como la visión por computadora, el procesamiento de lenguaje natural y la inteligencia artificial en general.

Dentro del Deep learning se destacan tres subtramas fundamentales: redes neuronales profundas (DNN), las redes neuronales convolucionales (CNN) y las redes neuronales recurrentes (RNN).

#### 1.3.6.1 Redes neuronales profundas

Las redes neuronales profundas se caracterizan por tener múltiples capas ocultas entre la capa de entrada y la capa de salida. Esta estructura permite que el modelo aprenda patrones complejos y representaciones abstractas de los datos. Cada capa transforma la salida de la capa anterior para detectar características más complejas.

Por ejemplo, una red que analiza imágenes, las primeras capas pueden identificar bordes y colores; las capas intermedias aprenden formas y texturas; mientras que las últimas capas combinan toda esa información para reconocer objetos o realizar clasificaciones.

Este enfoque jerárquico es muy potente, aunque también requiere un alto poder computacional y grandes volúmenes de datos para su entrenamiento eficaz.

### 1.3.6.2 Redes neuronales convolucionales

Las redes neuronales convolucionales están diseñadas específicamente para trabajar con datos que tienen una estructura espacial, como imágenes o videos. Utilizan capas convolucionales que aplican filtros sobre pequeñas regiones del dato de entrada, permitiendo detectar características locales como bordes, texturas y formas.

Estas capas extraen representaciones claves de la imagen, facilitando el reconocimiento de patrones espaciales complejos. Gracias a esta capacidad, las CNN son ampliamente utilizadas en tareas como el reconocimiento facial, la detección de objetos y la segmentación de imágenes.

### 1.3.6.3 Redes neuronales recurrentes

Las redes neuronales recurrentes están diseñadas para procesar datos secuenciales, donde el orden de los elementos es relevante como el texto, audio o series temporales. A diferencia de las redes neuronales tradicionales, las RNN incorporan conexiones que permiten conservar información del pasado, lo que les da la capacidad de mantener el contexto a lo largo de una secuencia.

Esto es especialmente útil en tareas donde la dependencia temporal o contextual es clave, como la traducción automática o la generación de texto.

### 1.3.7 Procesamiento del lenguaje natural

El procesamiento del lenguaje natural es un subcampo de la inteligencia artificial que se enfoca en permitir que los computadores comprendan, interpreten, generen y respondan al lenguaje humano, ya sea hablado o escrito. Sus aplicaciones abarcan asistentes virtuales, chatbots, motores de búsqueda, sistemas de corrección gramatical, entre otros.

### 1.3.7.1 Reconocimiento de voz

El reconocimiento de voz convierte señales de audio en texto. Para ello, el sistema analiza fragmentos de audio, identifica fonemas, y mediante modelos de Deep learning, asocia patrones sonoros con palabras. Desafíos como acentos o ruidos de fondo, se abordan con redes neuronales profundas y recurrentes. Esta tecnología se usa en asistentes como Alexa o Google assistant, así como también en sistemas de transcripción automática.

### 1.3.7.2 Traducción automática

La traducción automática transforma texto o voz de un idioma a otro. Los sistemas actuales utilizan redes neuronales que analizan el contexto completo de una frase para generar traducciones coherentes y precisas, superando la simple traducción palabra por palabra. Es empleada en herramientas como Google Translate y plataformas con traducción en tiempo real como Skype o Zoom.

### 1.3.7.3 Modelado del lenguaje

El modelado del lenguaje permite a las máquinas predecir y generar texto con sentido, comprendiendo la relación entre palabras y su contexto. Modelos como GPT, basados en redes neuronales profundas, aprenden patrones complejos a partir de grandes volúmenes de datos. Sus aplicaciones incluyen chatbots, correctores gramaticales, asistentes de escritura y motores de búsqueda predictivo.

### 1.3.8 Visión artificial

La visión artificial es uno de los subcampos más relevante dentro de la inteligencia artificial y el machine learning, su objetivo principal consiste en imitar las capacidades de la visión humana, y va más allá al replicar la percepción y la capacidad cognitiva de los seres humanos para interpretar lo que observan. Este campo se fundamenta principalmente en el reconocimiento de patrones. Los

métodos y técnicas de Deep learning han revolucionado profundamente la visión artificial, en particular las redes neuronales convolucionales, las cuales permiten procesar grandes volúmenes de imágenes a nivel de pixel. Estas tecnologías se enfocan en el desarrollo de sistemas digitales y algoritmos capaces de procesar, analizar e interpretar datos visuales de manera similar a como lo haría un ser humano.

#### 1.3.8.1 Reconocimiento de imágenes

El reconocimiento de imágenes es una de las tareas más fundamentales dentro de la visión artificial. Esta técnica se basa en la clasificación e identificación de objetos o elementos presentes en una imagen. El proceso consiste en analizar imágenes para detectar patrones o características relevantes, como bordes, formas, colores o texturas. Posteriormente, un modelo de aprendizaje profundo, como una red neuronal convolucional, procesa dicha información y clasifica la imagen en una categoría específica.

Entre sus aplicaciones más comunes se encuentran el reconocimiento facial, permitiendo identificar personas en fotografías o videos, como ocurre en sistemas de desbloqueo facial; la clasificación automática de imágenes, donde puede distinguir si la imagen contiene un perro, un gato o un árbol; y el diagnóstico médico, donde este tipo de tecnología puede detectar enfermedades en imágenes como radiografías o resonancias magnéticas, entre muchas otras aplicaciones.

#### 1.3.8.2 Detección de objetos

La detección de objetos va un paso más allá del reconocimiento de imágenes, ya que no sólo identifica qué objetos están presentes en una imagen o video, sino que también determina su ubicación exacta. Para ello, se emplean modelos que se conocen como “*cajas delimitadoras*”, que permiten indicar visualmente la posición de cada objeto dentro del marco visual.

Este enfoque es ampliamente utilizado en áreas como la conducción autónoma, permitiendo detectar vehículo, peatones o señales de tránsito, lo que facilita la toma de decisiones en tiempo

real. También tiene aplicaciones relevantes en seguridad y vigilancia, al ser capaz de identificar intrusos o comportamientos sospechosos en grabaciones de video.

### 1.3.8.3 Segmentación de imágenes

La segmentación o seguimiento de imágenes se enfoca en rastrear un objetivo específico a lo largo del tiempo dentro de una secuencia de video. Inicialmente, un sistema de detección localiza el objeto en un cuadro de video y lo segmenta. Luego, algoritmos de seguimientos predicen el movimiento del objeto en los cuadros siguientes, ajustando continuamente su posición. Diversas técnicas de Deep learning permiten mantener un seguimiento preciso, incluso si el objeto cambia de tamaño, forma o posición.

Entre sus aplicaciones más destacadas se encuentra el ámbito deportivo, donde se puede rastrear una pelota o jugadores durante un partido; la videovigilancia, donde se puede seguir a una persona en tiempo real; y la realidad aumentada, que requiere mantener objetivos virtuales correctamente alineados con el entorno real.

Para integrar estos tres conceptos, un ejemplo práctico sería un sistema de cámaras de tránsito. En este caso, el reconocimiento de imágenes identifica los tipos de vehículos presentes en el cuadro, como automóviles, camiones o motocicletas. La detección de objetos determina su ubicación exacta en la imagen, y finalmente, la segmentación de imágenes permite rastrear a cada vehículo en tiempo real, facilitando la predicción de movimientos y superponiendo información relevante como velocidad, número de patente, tipo o color del vehículo, entre muchos otros datos.

### 1.3.9 Computación evolutiva

Este subcampo de machine learning está inspirado en el proceso de evolución natural para resolver diversos tipos de problemas. Utiliza conceptos fundamentales como la selección natural, la mutación y la reproducción, con el objetivo de encontrar soluciones óptimas o casi óptimas en contextos donde otras técnicas tradicionales podrían fallar.

### 1.3.9.1 Algoritmos genéticos

Los algoritmos genéticos constituyen una herramienta dentro de la computación evolutiva, utilizada para encontrar las mejores soluciones a un problema mediante la prueba y combinaciones de múltiples opciones, simulando el comportamiento de los genes. Básicamente, este enfoque comienza generando varias soluciones al azar, las cuales son evaluadas en función de su desempeño o aptitud. Posteriormente, las mejores soluciones son cruzadas (combinadas) y modificadas ligeramente (mutadas). Este proceso se repite reiteradamente hasta encontrar una solución adecuada.

Un ejemplo clásico sería un algoritmo diseñado para encontrar la mejor ruta para un repartidor que debe visitar múltiples domicilios. A través de múltiples generaciones de rutas, el algoritmo podría descubrir trayectos más eficientes que los diseñados manualmente.

### 1.3.9.2 Programación genética

La programación genética es una extensión de los algoritmos genéticos que, en lugar de enfocarse en buscar números o rutas, se orienta a desarrollar programas o fórmulas que resuelven problemas específicos. Al igual que en los algoritmos genéticos, se parte de programas simples generados aleatoriamente, los cuales se evalúan según su capacidad de resolver el problema propuesto. Los programas con mejor desempeño son cruzados y ligeramente modificados, dando lugar a nuevas generaciones de soluciones.

Por ejemplo, podría desarrollarse un programa capaz de diseñar circuitos eléctricos orientados a un objetivo determinado. Dichos diseños comenzarían con combinaciones aleatorias de componentes básicos como resistencias, capacitores y transistores. Cada circuito será evaluado en término de costo, eficiencia y funcionalidad. Los mejores diseños se seleccionarían y combinarían para crear nuevas versiones optimizadas, permitiendo la generación de soluciones innovadoras que podrían resultar difíciles de imaginar manualmente.

### 1.3.9.3 Optimización evolutiva

La optimización evolutiva es una técnica que busca mejorar programas, fórmulas o algoritmos existentes mediante procesos inspirados en la evolución natural. Al igual que en los algoritmos genéticos, se basa en principios como la selección natural, el cruce y la mutación. El procedimiento parte de un modelo vigente considerado como solución inicial, al cual se le introducen nuevas propuestas generadas desde cero. Las mejores alternativas se seleccionan y combinan con una solución original, incorporando pequeñas mejoras. Este ciclo se repite hasta alcanzar el nivel deseado de optimización.

Un ejemplo ilustrativo podría ser la optimización evolutiva en el diseño de motores eléctricos. Supongamos que ya se dispone de un modelo funcional. Para mejorar su eficiencia energética, el programa generaría múltiples variantes del diseño original, explorando distintas combinaciones de materiales, tamaños de componentes y configuraciones internas. Las versiones más eficientes serían seleccionadas y cruzadas con el diseño principal, introduciendo pequeñas mejoras sucesivas hasta lograr una versión optimizada del motor.

## 1.4 **REFLEXIÓN GENERAL**

El estado del arte de la inteligencia artificial evidencia cómo esta disciplina ha evolucionado hacia un ecosistema diverso de subcampos y metodologías. Desde enfoques tradicionales como el aprendizaje supervisado y no supervisado, hasta técnicas más avanzadas como el deep learning, la IA ha transformado radicalmente la forma en que se procesan y utilizan los datos.

Subcampos como la visión artificial destacan por su capacidad para resolver problemas complejos, generando un impacto significativo en industrias como la salud, la logística y la tecnología en general. Asimismo, herramientas específicas como las redes neuronales profundas (Deep Neural Networks), las redes neuronales recurrentes (Recurrent Neural Networks), y las redes neuronales convolucionales (Convolutional Neural Networks) han permitido abordar tareas sofisticadas, tales como el reconocimiento de imágenes, la traducción automática y la predicción de secuencias temporales.

El procesamiento del lenguaje natural y la computación evolutiva, por su parte, abren nuevas oportunidades en la interacción humano-máquina, al ofrecer soluciones que emulan procesos cognitivos y evolutivos propios del ser humano. En conjunto, estos avances no sólo redefinen la manera en que se interactúa con la tecnología, sino que también proporcionan soluciones innovadoras frente a los desafíos del presente y del futuro, guiando el camino hacia un mundo más eficiente, automatizado e interconectado.

## **1.5 OBJETIVOS.**

En esta sección se presentan los objetivo general y específicos que se espera alcanzar al finalizar el desarrollo de este trabajo.

### **1.5.1 Objetivo general**

Analizar los principales conceptos de la inteligencia artificial e implementar ejercicios con redes neuronales, tanto de forma manual como utilizando el lenguaje de programación Python.

### **1.5.2 Objetivos específicos**

- Analizar el estado del arte de la inteligencia artificial, identificando sus principales métodos y subcampos.
- Describir y explicar los fundamentos teóricos de las redes neuronales artificiales, incluyendo su estructura, funcionamiento y principios matemáticos básicos.
- Implementar ejercicios prácticos de entrenamientos de redes neuronales artificiales, tanto de forma manual como programada en Python.
- Desarrollar una aplicación práctica basada en redes neuronales, para el reconocimiento de escritura de números naturales.
- Integrar una interfaz gráfica que permita al usuario dibujar números manualmente y observar su predicción en tiempo real.

- Obtener resultados prácticos derivados de la programación en Python aplicando redes neuronales artificiales.

## **CAPÍTULO 2: DESARROLLO DEL PROYECTO**

## **2 DESARROLLO DEL PROYECTO**

En este capítulo se explorarán las redes neuronales artificiales, un modelo computacional inspirado en el cerebro humano que es capaz de aprender y procesar datos. Además, se explicará el funcionamiento y ejecución del programa desarrollado en Python enfocado en esta área, el cual incluye un ejercicio práctico orientado al entrenamiento de una red neuronal que representa una compuerta lógica AND. En primera instancia, dicho ejercicio se abordará de manera teórica, posteriormente, este análisis se complementa mediante el desarrollo de un código en Python, implementado en el entorno de PyCharm y utilizando las bibliotecas TensorFlow y Keras. Finalmente, se presentará una reflexión sobre ambos enfoques.

### **2.1 HISTORIA Y CONTEXTO DE LAS REDES NEURONALES**

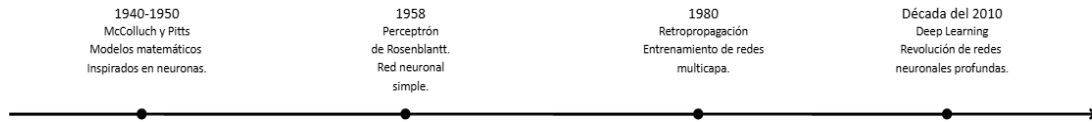
Las redes neuronales artificiales constituyen uno de los pilares fundamentales en el campo de la inteligencia artificial (IA), y están inspiradas en la estructura y el funcionamiento del cerebro humano. Estas redes están compuestas por nodos interconectados que simulan el comportamiento de una neurona biológica.

La idea de las redes neuronales surgió entre las décadas de 1940 y 1950, cuando investigadores como Warren McCulloch y Walter Pitts propusieron modelos matemáticos inspirados en la actividad de las neuronas humanas. Posteriormente, en 1958, Frank Rosenblatt desarrolló el perceptrón, una de las primeras redes neuronales capaces de aprender patrones simples.

Durante las décadas siguientes, el desarrollo de las redes neuronales experimentó altibajos. Sin embargo, en la década de 1980 resurgió el interés gracias al desarrollo del algoritmo de retropropagación (backpropagation), el cual permitió entrenar redes neuronales con múltiples capas.

Fue en la última década cuando las redes neuronales vivieron una verdadera revolución, impulsadas por el crecimiento exponencial del poder de cómputo, la disponibilidad de grandes volúmenes de datos y los avances en hardware especializado. En la Figura 2-1 se presenta una línea de tiempo con los hitos más relevantes en la evolución de las redes neuronales artificiales.

## EVOLUCIÓN HISTÓRICA DE LAS REDES NEURONALES ARTIFICIALES.



Fuente: Elaboración propia

Figura 2-1 Evolución histórica de las redes neuronales artificiales.

## **2.2 FUNCIONAMIENTO GENERAL DE UNA RED NEURONAL BIOLÓGICA**

Una neurona biológica es una célula del sistema nervioso encargada de recibir, procesar y transmitir señales tanto eléctricas como químicas. Estas células son fundamentales para el funcionamiento del cerebro y del sistema nervioso en general, ya que permiten la comunicación entre distintas partes del cuerpo, así como el procesamiento de la información sensorial, motora y cognitiva. La estructura de una neurona se compone de tres partes principales, tal como se muestra en la Figura 2-2.

### **2.2.1 Dendritas**

Las dendritas son prolongaciones ramificadas que se extienden desde el cuerpo celular de la neurona. Su función principal es recibir señales químicas o eléctricas provenientes de otras neuronas. Estas señales llegan en forma de neurotransmisores, los cuales son liberados en las sinapsis, permitiendo así la transmisión de información entre células nerviosas.

### **2.2.2 Cuerpo celular o soma**

El cuerpo celular, también conocido como soma, constituye el núcleo funcional de la neurona. En él se alberga el material genético y es el lugar donde se integran todas las señales recibidas desde

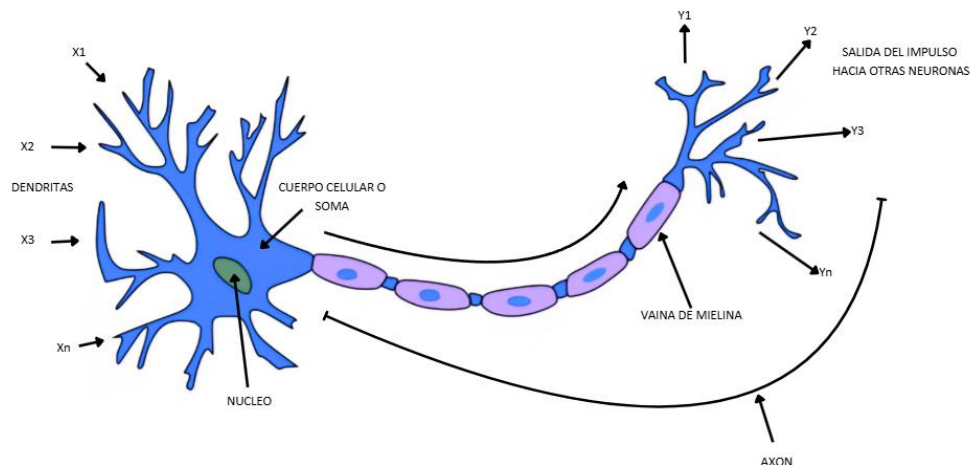
las dendritas. Si la suma de estas señales alcanza un determinado umbral, se genera un potencial de acción que se propagará a lo largo del axón.

### 2.2.3 Axón

El axón es una prolongación larga que transmite el impulso nervioso desde el soma hacia otras neuronas, músculos o glándulas. Está recubierto por una vaina de mielina, la cual acelera la conducción del impulso eléctrico. Al final del axón se encuentran los terminales sinápticos, donde se liberan neurotransmisores que permiten la comunicación con otras células.

La sinapsis es el punto de conexión entre dos neuronas, en el cual la señal eléctrica se convierte en una señal química mediante la liberación de neurotransmisores, permitiendo así la transmisión de información de una célula a otra.

Las neuronas no sólo se encargan de transmitir información, sino que también son capaces de procesarla y modificarla a través de mecanismos como la plasticidad sináptica, lo que posibilita funciones complejas como el aprendizaje y la memoria. A diferencia de otras células del cuerpo, las neuronas no se reproducen fácilmente, por lo que los daños que sufren pueden tener efectos duraderos.



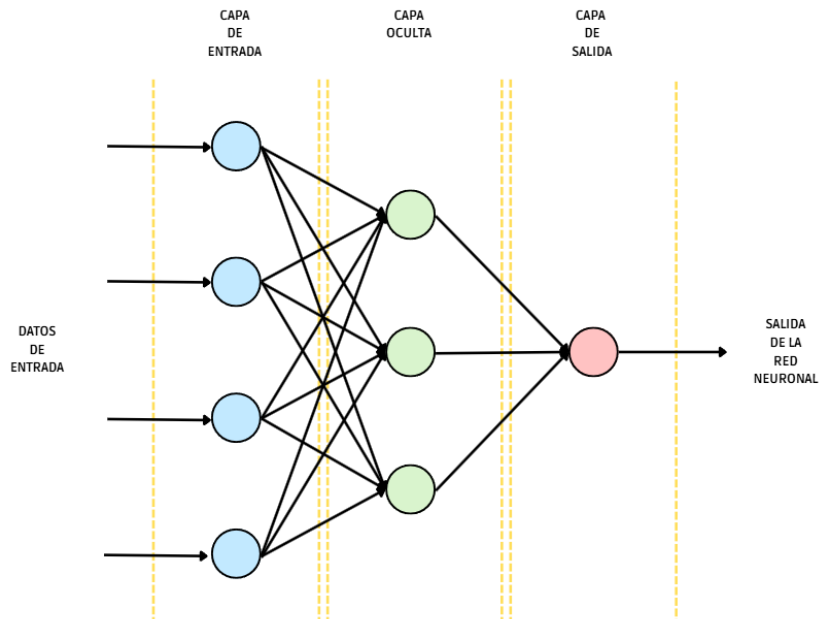
Fuente: Elaboración propia

Figura 2-2 Neurona biológica.

## 2.3 FUNCIONAMIENTO GENERAL DE UNA RED NEURONAL ARTIFICIAL

Una red neuronal artificial (RNA) consiste en un conjunto de capas formadas por neuronas interconectadas, de manera similar a una red neuronal biológica. Cada neurona realiza un cálculo simple: recibe entradas en forma de datos numéricos (que pueden representar imágenes, texto, sonido, entre otros), los multiplica por un conjunto de pesos, suma un término conocido como sesgo y luego aplica una función de activación. Esta función permite que la red aprenda patrones complejos. El resultado se transmite a la siguiente capa, y este proceso se repite sucesivamente hasta llegar a la capa de salida.

La red finaliza generando una predicción, clasificación, puntuación u otra salida, dependiendo de la tarea específica. El objetivo es que, mediante ajustes en los pesos, la red aprenda a transformar las entradas en las salidas deseadas con la mayor precisión posible. Este proceso se denomina propagación hacia adelante (forward propagation). En la Figura 2-3 se muestra la estructura básica de una red neuronal artificial.



Fuente:

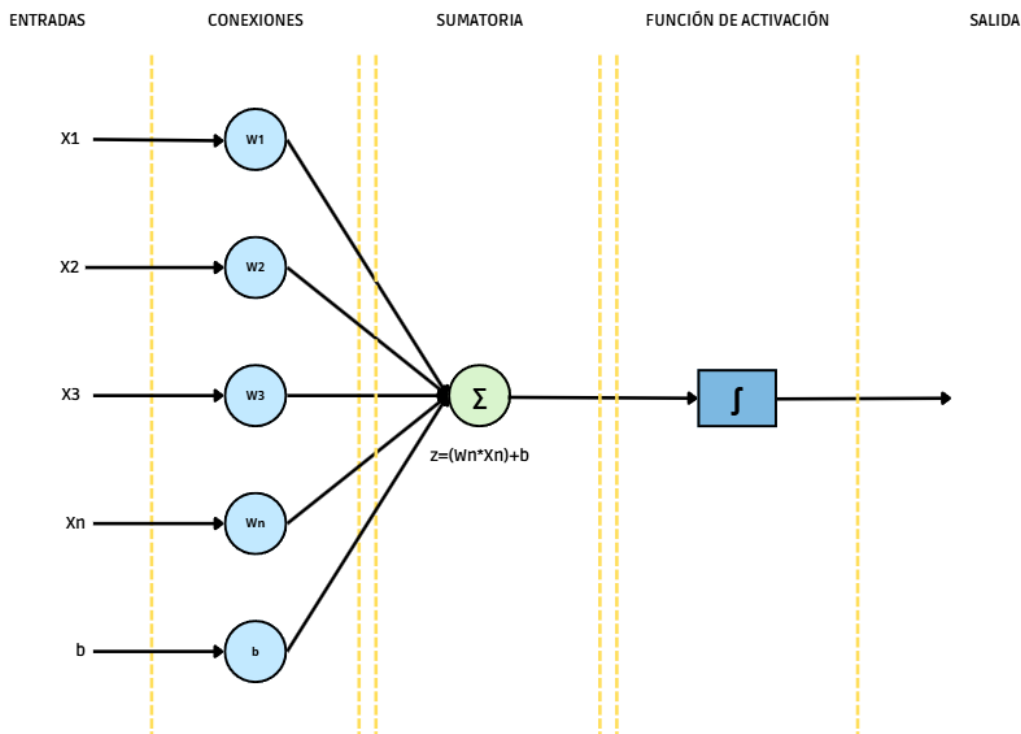
Fuente: Elaboración propia

Figura 2-3 Estructura básica de una red neuronal artificial.

Las redes neuronales artificiales (RNA) están compuestas por diversos elementos esenciales que trabajan en conjunto para procesar la información y permitir el aprendizaje a partir de los datos. A continuación, se describen los componentes más relevantes:

### 2.3.1 Neurona artificial

Las neuronas artificiales son las unidades básicas de procesamiento en una red neuronal. Cada una de ellas recibe un conjunto de entradas ( $X$ ), que se combinan de forma ponderada mediante los pesos ( $W$ ) y un término llamado sesgo ( $b$ ). El resultado de esta combinación se transforma a través de una función de activación, lo que permite introducir no linealidad al modelo. Las salidas generadas se transmiten posteriormente a las neuronas de las capas siguientes. En la Figura 2-4 se presenta un esquema representativo de una neurona artificial.



Fuente: Elaboración propia

Figura 2-4 Esquema de una neurona artificial.

### 2.3.2 Capas

Las capas de una red neuronal se dividen en tres tipos: capa de entrada, capas ocultas y capa de salida. Cada una de ellas cumple un rol específico en el flujo y la transformación de los datos dentro de la red, permitiendo el procesamiento progresivo de la información desde las entradas iniciales hasta la obtención del resultado final.

#### 2.3.2.1 Capa de entrada

La capa de entrada es la primera capa de una red neuronal y tiene la función de recibir los datos originales en forma numérica. Esta capa no realiza procesamiento alguno; su propósito es simplemente entregar los datos al resto de la red para que puedan ser transformados y analizados por las capas posteriores.

#### 2.3.2.2 Capas ocultas

Las capas ocultas son aquellas capas intermedias que se encargan de transformar los datos a través de múltiples combinaciones de pesos, sesgos y funciones de activación. Son responsables del procesamiento interno de la información y permiten que la red aprenda representaciones complejas de los datos. La cantidad de capas ocultas, conocida como profundidad de la red, es lo que determina si se trata de una red superficial o de una red profunda. Una red se considera profunda cuando posee dos o más capas ocultas entre la capa de entrada y la de salida. En cambio, si solo tiene una capa oculta se clasifica como una red superficial.

#### 2.3.2.3 Capa de salida

La capa de salida es la capa final de la red neuronal artificial. Entrega el resultado o la predicción generada tras el procesamiento completo de la información a lo largo de la red. El formato y la naturaleza de esta salida dependerán de los requerimientos específicos del problema, como clasificación, regresión o detección de patrones.

### 2.3.3 Pesos y sesgos

Los pesos determinan la importancia de cada conexión entre neuronas dentro de la red, mientras que los sesgos permiten desplazar la función de activación, otorgando al modelo mayor flexibilidad para ajustarse a diferentes patrones. Ambos parámetros son fundamentales en el funcionamiento de una red neuronal y se modifican durante el proceso de entrenamiento con el objetivo de minimizar el error entre la salida esperada y la salida generada por el modelo.

### 2.3.4 Funciones de activación

Una función de activación es una función matemática que determina la salida de una neurona en función de su entrada. Su uso es especialmente relevante en modelos no lineales, ya que permite que la red neuronal aprenda patrones y relaciones complejas en los datos proporcionados.

Algunas de las funciones de activación más comunes son:

#### 2.3.4.1 ReLU

La función ReLU (Rectified Linear Unit o unidad lineal rectificadora) se define como.

$$f(x) = \max(0, x)$$

Esta función devuelve 0 cuando la entrada es negativa y el valor de  $x$  cuando la entrada es positiva.

Aunque su forma es simple, ReLU permite que los modelos aprendan más rápidamente en comparación con otras funciones de activación. A pesar de ser lineal en el tramo positivo, al aplicarse en múltiples capas dentro de una red profunda, posibilita el aprendizaje de relaciones no lineales complejas gracias a su combinación con otras neuronas.

### 2.3.4.2 Sigmoide

La función sigmoide se define como:

$$f(x) = \frac{1}{(1 + e^{(-x)})}$$

Esta función transforma cualquier valor numérico en un resultado comprendido entre 0 y 1, lo que la hace especialmente útil en problemas de clasificación binaria, como, por ejemplo: “¿es spam o no?”, “¿sí o no?”.

Debido a que el término disminuye  $e^{(-x)}$  rápidamente a medida que  $x$  aumenta, los valores negativos grandes producen resultados cercanos a 0, mientras que los valores positivos grandes generan salidas cercanas a 1. Así, cuando  $x$  es muy grande, la salida de la función se aproxima a 1; y cuando  $x$  es muy negativo, se aproxima a 0.

En resumen, la función sigmoide convierte cualquier número real en un valor entre 0 y 1, lo que la hace ideal para representar probabilidades.

### 2.3.4.3 Tanh

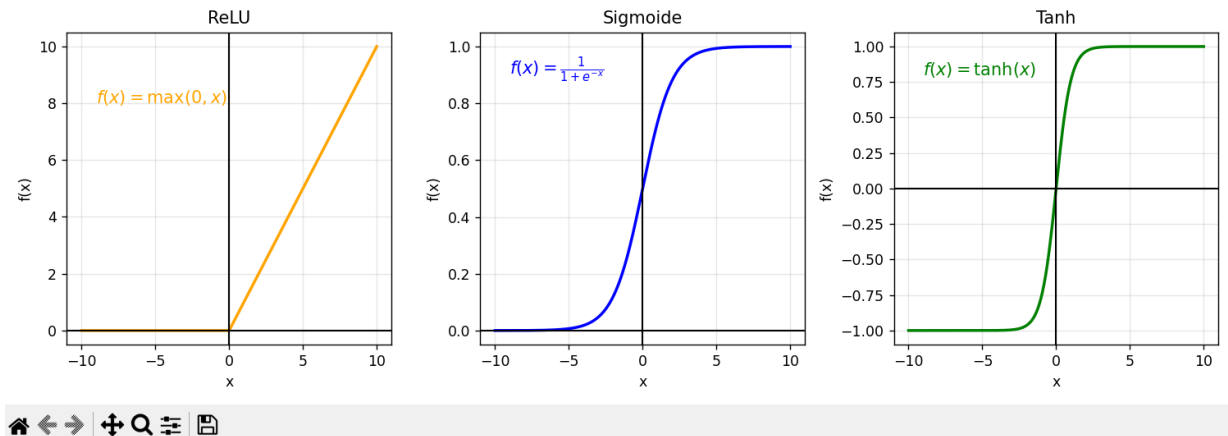
La tangente hiperbólica, comúnmente conocida como  $\tanh$ , es una función de activación cuya expresión matemática es:

$$f(x) = \frac{(e^x - e^{(-x)})}{(e^x + e^{(-x)})}$$

Esta función transforma los valores de entrada en un rango comprendido entre -1 y 1. Cuando  $x$  es grande, la salida se aproxima a 1; cuando  $x$  es muy negativo, se aproxima a -1; y si  $x=0$ , el resultado es exactamente 0.

Debido a su naturaleza centrada en cero,  $\tanh$  resulta especialmente útil en tareas de procesamiento de lenguaje natural, como el análisis de sentimientos. Por ejemplo, un modelo de red neuronal artificial utiliza la función  $\tanh$  para clasificar opiniones, un valor cercano a 1 podría indicar un sentimiento muy positivo (“me encantó el producto”), un valor cercano a -1 reflejaría un sentimiento muy negativo (“fue una pésima experiencia”), mientras que un valor cercano a 0

señalaría una opinión neutral (“estuvo bien”). En la figura 2-5 se observan los gráficos correspondientes a las funciones de activación analizadas anteriormente:



Fuente: Elaboración propia

Figura 2-5 Gráficos de las funciones de activación ReLU, Sigmoide y Tanh.

### 2.3.5 Funciones de pérdida

Las funciones de pérdida son herramientas fundamentales en el entrenamiento de una red neuronal, ya que permiten medir qué tan errónea es la salida generada por la red en comparación con la respuesta esperada. Esta medida de error guía el proceso de ajuste de los pesos durante el aprendizaje.

Entre las funciones de pérdida más utilizadas se encuentran el error cuadrático medio (MSE) y la entropía cruzada (cross-entropy).

#### 2.3.5.1 Error cuadrático medio

El error cuadrático medio (MSE, por sus siglas en inglés) se calcula tomando el promedio de los errores al cuadrado entre las predicciones del modelo y los valores reales. Al elevar al cuadrado las diferencias, los errores grandes tienen un mayor impacto en el resultado final, lo que permite identificar desviaciones significativas.

Un valor de MSE más bajo indica que el modelo realiza predicciones con mayor precisión, por lo que esta métrica es especialmente útil en tareas de regresión.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Donde  $y_i$  es el valor real,  $\hat{y}_i$  es la predicción, y  $n$  es el número total de observaciones o cantidad de ejemplos que se están utilizando para calcular el error.

### 2.3.5.2 Entropía cruzada

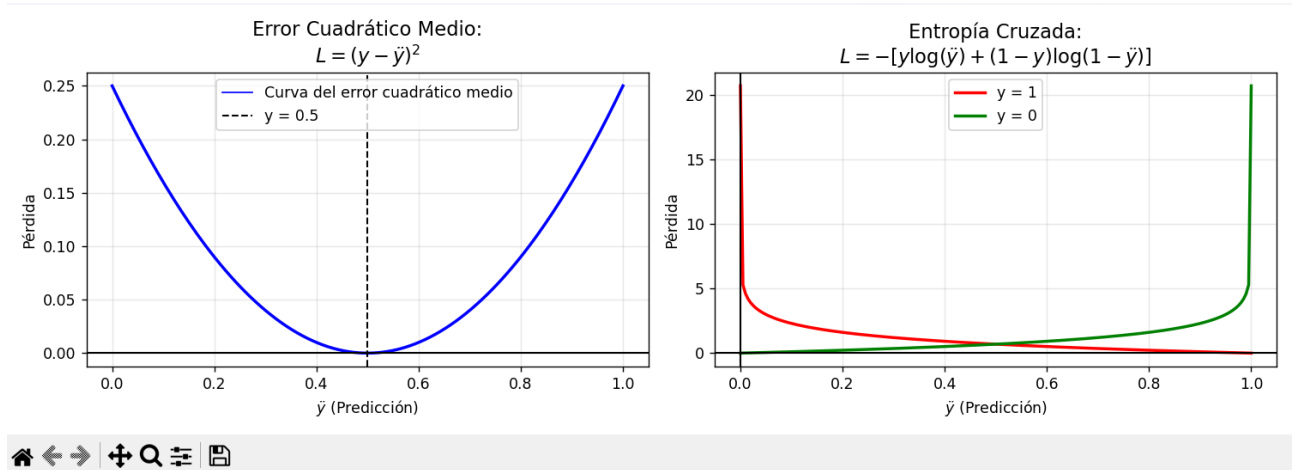
La entropía cruzada es una función de pérdida utilizada comúnmente en tareas de clasificación. Mide qué tan buenas son las predicciones de un modelo cuando se trabaja con probabilidades, evaluando cuán cercanas están las distribuciones predichas respecto a las verdaderas.

$$L = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^k y_{ij} \log(\hat{y}_{ij})$$

La entropía cruzada mide qué tan bien las probabilidades predichas ( $\hat{y}$ ) coinciden con las verdaderas etiquetas ( $y$ ). En esta función,  $n$  representa el número de muestras,  $k$  el número de clases posibles,  $y_{ij}$  el valor real (1 si la muestra  $i$  pertenece a la clase  $j$ , y 0 en caso contrario), y  $\hat{y}_{ij}$  la probabilidad predicha de que la muestra  $i$  pertenezca a la clase  $j$ . Cuanto menor sea el valor de la pérdida  $L$ , mejor será el desempeño del modelo.

Por ejemplo, si el modelo predice correctamente una clase con un 90 % de confianza, se considera un buen resultado. Sin embargo, si predice con un 90 % de seguridad una clase incorrecta, la penalización será alta. Este comportamiento incentiva al modelo a ajustar sus probabilidades, aprendiendo a no estar demasiado seguro cuando no debe, lo cual contribuye a mejorar sus predicciones progresivamente.

En la Figura 2-6 se muestran los gráficos correspondientes a las funciones de pérdida.



Fuente: Elaboración propia

Figura 2-6 Gráficos de las funciones pérdida.

### 2.3.6 Algoritmo de optimización

En una red neuronal, el algoritmo de optimización es el método utilizado para ajustar los pesos con el fin de minimizar la función de pérdida o el error. Este algoritmo es fundamental para que la red pueda aprender a partir de los datos y realizar un entrenamiento automático.

Uno de los algoritmos más utilizados es el gradiente descendente, considerado la base de muchos otros métodos más avanzados. Este algoritmo consiste en calcular el gradiente (o derivada) de la función de pérdida con respecto a los pesos, y luego ajustar dichos pesos en la dirección contraria al gradiente. De esta manera, se busca minimizar la pérdida o el error del modelo en cada iteración.

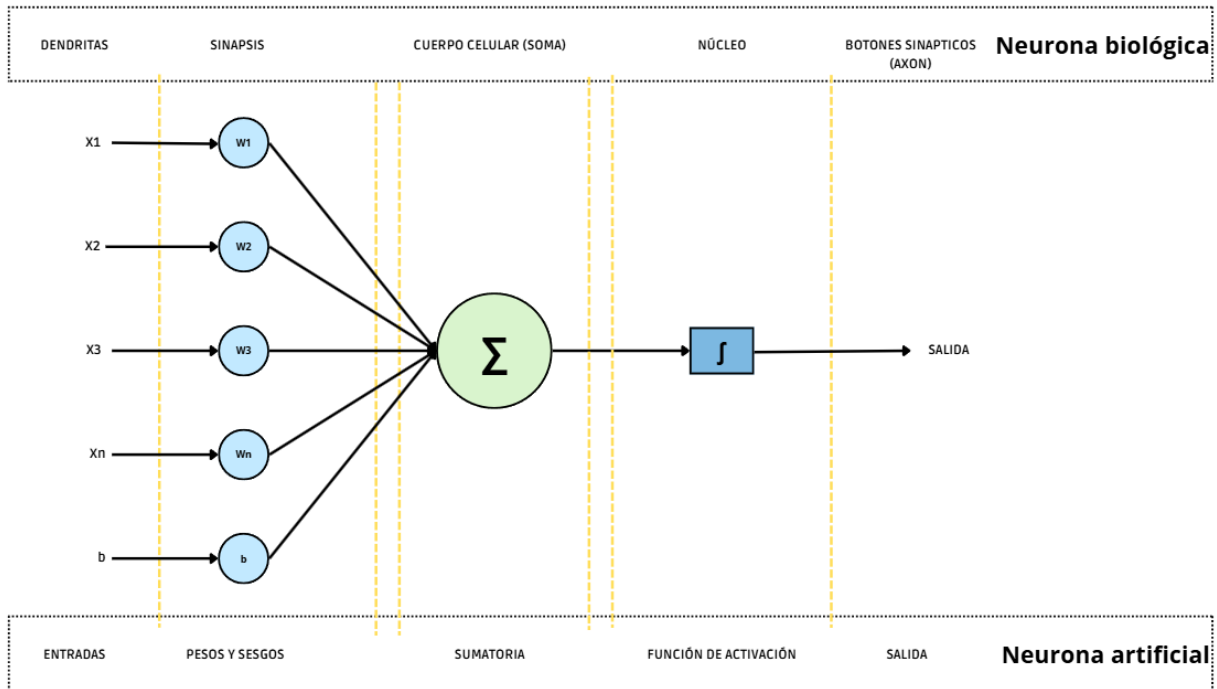
## 2.4 COMPARACIÓN ENTRE UNA NEURONA BIOLÓGICA Y UNA NEURONA ARTIFICIAL

Una neurona biológica es una célula especializada del sistema nervioso cuya función principal es recibir, procesar y transmitir información mediante impulsos eléctricos y señales químicas. De manera análoga, una neurona artificial es una unidad computacional inspirada en la neurona biológica, diseñada para procesar información numérica dentro de una red neuronal artificial.

- Las dendritas de la neurona biológica son las encargadas de recibir señales provenientes de otras neuronas, corresponden en la neurona artificial a las entradas  $X_1, X_2 \dots X_n$ , que representan los datos de entrada, como los píxeles de una imagen o valores numéricos de sensores.
- Las sinapsis, puntos de conexión donde se transmiten señales eléctricas y químicas mediante neurotransmisores, tienen como equivalente artificial a los pesos  $W_1, W_2 \dots W_n$  y al sesgo  $b$ . Estos determinan la importancia relativa de cada entrada en la salida de la neurona; un peso mayor implica una señal más influyente.
- El soma o cuerpo celular de la neurona biológica integra todas las señales recibidas. En la neurona artificial, esta integración se realiza mediante una suma ponderada de las entradas multiplicadas por sus respectivos pesos, más un valor adicional llamado sesgo. Esta operación genera un valor de activación  $z$ , calculado como:

$$z = \sum_{i=1}^n X \times W + b$$

- El núcleo de la neurona biológica toma la decisión de activar o no la neurona. En la neurona artificial, esta función se realiza mediante una función de activación, que determina si el valor de  $z$  es suficientemente significativo para generar una salida. Las funciones de activación más comunes son ReLU, sigmoide y tanh.
- Finalmente, el axón es la estructura de la neurona biológica encargada de transmitir la señal eléctrica a otras neuronas o células. Su equivalente artificial es la salida  $y$ , que representa el valor que se transmite a la siguiente neurona o a la siguiente capa de la red neuronal artificial. En la Figura 2-7 se presenta una comparación entre el funcionamiento fundamental de una neurona biológica y una neurona artificial.



Fuente: Elaboración propia

Figura 2-7 Esquema de comparación entre una neurona biológica y artificial.

## 2.5 TIPOS DE REDES NEURONALES

Existen diversos tipos de redes neuronales, las cuales se clasifican principalmente según su arquitectura y el propósito para el que fueron diseñadas. Entre las más importantes se encuentran:

- Redes neuronales artificiales (ANN)
- Redes neuronales convolucionales (CNN)
- Redes neuronales recurrentes (RNN), junto con sus variantes como LSTM (Long Short-Term Memory) y GRU (Gated Recurrent Unit)
- Redes basadas en transformadores (Transformers)
- Redes autoencoders

Cada una de estas arquitecturas posee características particulares que las hacen más adecuadas para determinados tipos de tareas, como clasificación de imágenes, procesamiento de texto, generación de secuencias o compresión de datos.

#### 2.5.1 Redes neuronales artificiales (ANN)

Las redes neuronales artificiales (Artificial Neural Networks, ANN) representan la forma más básica de red neuronal. Están compuestas por capas de neuronas interconectadas entre sí, generalmente organizadas en una estructura de capa de entrada, capas ocultas y una capa de salida.

Este tipo de red se utiliza comúnmente en tareas sencillas de predicción y clasificación, como el reconocimiento de patrones o la estimación de valores numéricos.

#### 2.5.2 Redes neuronales convolucionales (CNN)

Las redes neuronales convolucionales (Convolutional Neural Networks, CNN) están diseñadas específicamente para trabajar con datos que presentan una estructura en forma de rejilla, como las imágenes. Estas redes utilizan filtros o convoluciones para detectar automáticamente características relevantes, tales como bordes, formas, texturas u otras estructuras espaciales presentes en los datos.

Gracias a esta capacidad, las CNN son ampliamente utilizadas en aplicaciones de visión por computadora, como el reconocimiento de objetos, la clasificación de imágenes y la detección de rostros.

#### 2.5.3 Redes neuronales recurrentes (RNN)

Las redes neuronales recurrentes (Recurrent Neural Networks, RNN) están especializadas en el procesamiento de datos secuenciales, es decir, aquellos que se presentan uno tras otro de forma sucesiva, como el audio, el texto o las series de tiempo.

Su característica distintiva es la capacidad de mantener una memoria interna, lo que les permite utilizar información de pasos anteriores para influir en las decisiones presentes. Esta propiedad las

hace especialmente útiles en tareas como traducción automática, análisis de sentimientos y predicción de secuencias.

#### 2.5.4 LSTM Y GRU

Las redes LSTM (Long Short-Term Memory) y GRU (Gated Recurrent Unit) son variantes avanzadas de las redes neuronales recurrentes (RNN), diseñadas para superar las limitaciones de memoria a corto plazo que presentan las RNN tradicionales. Estas arquitecturas permiten a la red recordar información relevante durante periodos más largos y olvidar datos irrelevantes, mejorando su desempeño en tareas secuenciales complejas.

Ambos modelos son ampliamente utilizados en aplicaciones de procesamiento de lenguaje natural, traducción automática y análisis de series temporales. La principal diferencia entre ambas radica en su estructura:

- GRU (Unidad Recurrente con Puerta): Presenta una arquitectura más simple, con menos parámetros, lo que le permite entrenar más rápido. Sin embargo, esta simplicidad puede llevar a una ligera pérdida de precisión en tareas más complejas.
- LSTM (Memoria a Corto y Largo Plazo): Posee una arquitectura más detallada y flexible, capaz de manejar secuencias largas y dependencias más complejas, aunque con un mayor costo computacional.

#### 2.5.5 Transformers

Las redes neuronales Transformers representan una de las arquitecturas más avanzadas en la actualidad. A diferencia de las redes neuronales recurrentes, que procesan los datos secuencialmente paso a paso, los Transformers son capaces de procesar secuencias completas en paralelo, lo que mejora significativamente la eficiencia y velocidad del entrenamiento.

Su elemento distintivo es el uso de métodos que permiten al modelo conectarse con la información más importante de los datos al realizar una tarea específica, lo que mejora su capacidad para procesar y comprender patrones complejos de manera eficiente.

Los Transformers son ampliamente utilizados en modelos de procesamiento de lenguaje natural como BERT (Bidirectional Encoder Representations from Transformers) que es un modelo que entiende el contexto de las palabras tanto desde la izquierda como desde la derecha en una oración, GPT (Generative Pre-trained Transformer) es un modelo generativo que predice texto y puede crear contenido a partir de un contexto dado, aunque también han demostrado un alto rendimiento en tareas relacionadas con imágenes, audio y datos estructurados.

### 2.5.6 Autoencoders

Las redes neuronales autoencoders se caracterizan por su capacidad para comprimir y luego reconstruir los datos, permite reducir la dimensionalidad de estos mismos, conservar las características más relevantes, eliminar ruido innecesario y facilitar tareas posteriores como análisis, visualización, almacenamiento eficiente o detectar patrones importantes. Su estructura está compuesta por dos partes principales: un codificador, que transforma los datos originales en una representación de menor dimensión, y un decodificador, que intenta reconstruir los datos originales a partir de dicha representación, no garantiza una reconstrucción perfecta; su objetivo es aproximar los datos originales lo mejor posible según la función de pérdida que utilice.

Además, los autoencoders se emplean en detección de anomalías, ya que, al aprender el comportamiento esperado del conjunto de datos, pueden identificar fácilmente aquellos puntos que se desvían significativamente de la norma, lo cual es útil en áreas como la ciberseguridad, mantenimiento predictivo y análisis financieros.

## **2.6 ENTRENAMIENTO DE UNA RED NEURONAL**

Entrenar una red neuronal artificial consiste en enseñarle a realizar predicciones correctas, ajustando sus conexiones internas, los pesos y sus sesgos a través de un proceso iterativo. Este procedimiento es similar a entrenar a una persona para resolver un problema: mediante práctica, retroalimentación y correcciones constantes.

Como se muestra en la Figura 2-8, el entrenamiento implica una serie de pasos que se repiten continuamente para mejorar el desempeño del modelo. A continuación, se describen las etapas más importantes de este proceso.



Fuente: Elaboración propia

Figura 2-8 Esquema del proceso de entrenamiento de una RNA.

### 2.6.1 Propagación hacia adelante

La propagación hacia adelante (forward propagation) es el primer paso del proceso de entrenamiento de una red neuronal. En esta etapa, los datos ingresan por la capa de entrada y atraviesan sucesivamente las capas ocultas, donde se realizan cálculos matemáticos combinando los datos con los pesos y sesgos actuales. Estos cálculos permiten determinar la importancia relativa de cada entrada en la predicción final.

A modo de ejemplo, si se introduce una imagen de un perro, la red procesará la información visual a través de sus capas y, al llegar a la capa de salida, generará una predicción. Por ejemplo: 20 % probabilidad de que sea un perro y 80 % de que sea otro animal.

### 2.6.2 Cálculo del error

En esta etapa, la red neuronal compara la predicción obtenida con el valor real o esperado. Continuando con el ejemplo, si la imagen correspondía a un perro, pero la red predijo un 80 % de probabilidad de que fuera otro animal, se considera que ha cometido un error significativo.

Este error se cuantifica utilizando alguna de las funciones de pérdida presentadas anteriormente, como la entropía cruzada, más utilizada en clasificación, ya que mide la diferencia entre la distribución de probabilidades predicha por la red y la distribución verdadera.

El objetivo principal de este paso es medir qué tan lejos está la predicción del valor correcto para luego poder ajustar los pesos y reducir ese error en las siguientes iteraciones.

### 2.6.3 Propagación hacia atrás

En esta etapa ocurre el proceso de aprendizaje, la red toma el error calculado y lo distribuye en sentido inverso, capa por capa, determinando cuánto contribuyó cada peso al error final. Este proceso se basa en el cálculo diferencial, específicamente en el uso de derivadas, para identificar en qué dirección y con qué magnitud se deben ajustar los pesos con el objetivo de reducir el error.

Puede compararse con una revisión en equipo: si se comete un error, se analiza quién tuvo mayor responsabilidad y se asigna a cada integrante su parte de la “culpa” para que todos puedan mejorar. De manera análoga, cada conexión en la red ajusta sus parámetros en función de su aporte al error.

### 2.6.4 Actualización de los pesos

Una vez que se ha calculado cómo cada peso contribuyó al error, la red utiliza esta información para ajustarlos mediante un algoritmo de optimización. Uno de los más utilizados es el descenso de gradiente, el cual actualiza los pesos con el objetivo de reducir el error.

En términos simples:

- Si un peso provocó un aumento en el error, se disminuye su valor.

- Si un peso ayudó a reducir el error, se mantiene o ajusta ligeramente hacia arriba, dependiendo de la magnitud del error.

Este proceso de ajuste no ocurre una sola vez, sino que se repite cientos o incluso miles de veces, hasta que la red logre un nivel aceptable de precisión.

### 2.6.5 Épocas

Una época es el momento en que la red neuronal procesa todo el conjunto de datos de entrenamiento, en otras palabras, cuando el modelo ha visto y ajustado sus parámetros usando cada ejemplo del conjunto de entrenamiento una vez, se considera que ha finalizado una época.

Este proceso se repite múltiples veces (a menudo decenas, cientos o incluso miles) con el objetivo de que la red mejore progresivamente su capacidad de realizar predicciones correctas. En cada época, los pesos y sesgos se actualizan en función de los errores cometidos, lo que permite que el modelo aprenda gradualmente.

No obstante, entrenar durante demasiadas épocas puede conducir a un fenómeno llamado sobreajuste (overfitting), en el cual la red memoriza los datos del entrenamiento en lugar de generalizar correctamente nuevos datos nunca vistos.

Además, es importante considerar que este proceso de entrenamiento implica un alto costo computacional, especialmente cuando se trabaja con grandes volúmenes de datos o modelos profundos. Por esta razón, es común utilizar unidades de procesamiento gráfico (GPU), las cuales permiten realizar operaciones matemáticas en paralelo, acelerando significativamente el tiempo de entrenamiento de la red neuronal.

## 2.7 PYTHON

Python es un lenguaje de programación de propósito general, caracterizado por una sintaxis clara y sencilla, lo que lo convierte en una opción accesible tanto para principiantes como para desarrolladores experimentados. Fue creado por Guido van Rossum en 1991 y, desde entonces, se ha consolidado como uno de los lenguajes más utilizados a nivel mundial. Su versatilidad permite

su aplicación en diversas áreas, tales como el desarrollo web, el análisis de datos, la automatización de tareas y la inteligencia artificial.

Python cuenta con un amplio ecosistema de bibliotecas que facilitan el desarrollo de aplicaciones complejas. En el ámbito de las redes neuronales y el aprendizaje profundo, destacan bibliotecas como TensorFlow, desarrollada por Google; PyTorch, desarrollada por Meta; y Keras, una interfaz de alto nivel que simplifica la construcción de modelos sobre TensorFlow. Estas bibliotecas permiten construir, entrenar y evaluar modelos de redes neuronales de manera eficiente y flexible.

Para el desarrollo práctico del proyecto se utiliza PyCharm, un entorno de desarrollo integrado para Python. Este software ofrece funcionalidades como autocompletado inteligente, detección de errores en tiempo real, integración con bibliotecas populares y herramientas de apoyo a la programación, lo que facilita la escritura de código estructurado, eficiente y con menor propensión a errores, especialmente en proyectos relacionados con redes neuronales.

En la Figura 2-9 se presentan los íconos representativos de las principales herramientas utilizadas en el desarrollo del proyecto. En particular, se muestra Python como lenguaje de programación base, TensorFlow y Keras como bibliotecas orientadas a la construcción y entrenamiento de redes neuronales, y PyCharm es el entorno de desarrollo utilizado en el proyecto. Estas herramientas conforman el entorno tecnológico sobre el cual se desarrollan los ejercicios prácticos presentados en este trabajo.



Fuente: Elaboración propia

Figura 2-9 Íconos de herramientas de desarrollo para RNA en Python.

## **2.8 EJERCICIO PRÁCTICO**

Para comprender en profundidad el funcionamiento de las redes neuronales artificiales, se desarrollará un ejercicio práctico realizado a mano, en el cual se entrenará una red neuronal simple para aprender una función lógica básica: Compuerta AND.

La red estará compuesta por:

- Dos neuronas en la capa de entrada,
- Dos neuronas en la capa oculta,
- Una neurona en la capa de salida.

Este ejemplo permitirá visualizar paso a paso el comportamiento de la red durante el entrenamiento, desde la propagación hacia adelante hasta la retropropagación del error y la actualización de los pesos. Para facilitar la comprensión, se utilizarán las siguientes variables estándar en el contexto de redes neuronales:

- $W$ : pesos
- $b$ : sesgos
- $a$ : activación de una neurona
- $L$ : error o función de pérdida
- $y$ : salida esperada
- $x$ : entrada

### 2.8.1 Datos iniciales

La fórmula de una neurona artificial es la representación matemática de cómo procesa información. Se inspira en el funcionamiento de las neuronas biológicas, pero de una manera simplificada. En esencia, su objetivo es tomar varias entradas, combinarlas y producir una única salida.

$$\text{Fórmula general neurona artificial} = z^{(1)} = W_n^{(1)}x + b^{(1)}$$

Para este ejercicio práctico se utilizará como entrada una compuerta lógica AND, es decir:

La red neuronal recibe dos valores de entrada, ambos iguales a 1.

- $\text{Entrada } (x) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

El valor que se espera obtener a la salida de la red.

- $\text{Salida esperada } (y) = [1]$

Se asignan valores pequeños aleatorios o definidos manualmente a los pesos  $W$  y los sesgos  $b$ .

- Los pesos representan la intensidad de la conexión entre neuronas de capas consecutivas. En este ejemplo, los valores iniciales de cada peso se establecen de manera aleatoria.
- Los sesgos son valores que se suman a la entrada de cada neurona, para darle mayor flexibilidad al modelo.

La conexión entre la capa de entrada y la capa oculta de la red cuenta con dos neuronas en la capa de entrada y dos neuronas en la capa oculta, por lo que los pesos de esta conexión se separan en una matriz de tamaño de  $2 \times 2$ :

- $Pesos = W^{(1)} = \begin{bmatrix} 0,1 & 0,4 \\ -0,2 & 0,3 \end{bmatrix}$

En esta matriz las filas representan las neuronas de entrada y las columnas representan las neuronas de la capa oculta.

- $W_{1,1}^{(1)} = 0,1$  es el peso que conecta la entrada 1 con la neurona oculta 1.
- $W_{1,2}^{(1)} = 0,4$  es el peso que conecta la entrada 1 con la neurona oculta 2.
- $W_{2,1}^{(1)} = -0,2$  es el peso que conecta la entrada 2 con la neurona oculta 1.
- $W_{2,2}^{(1)} = 0,3$  es el peso que conecta a la entrada 2 con la neurona oculta 2.

Los sesgos de la capa oculta se representan como un vector:

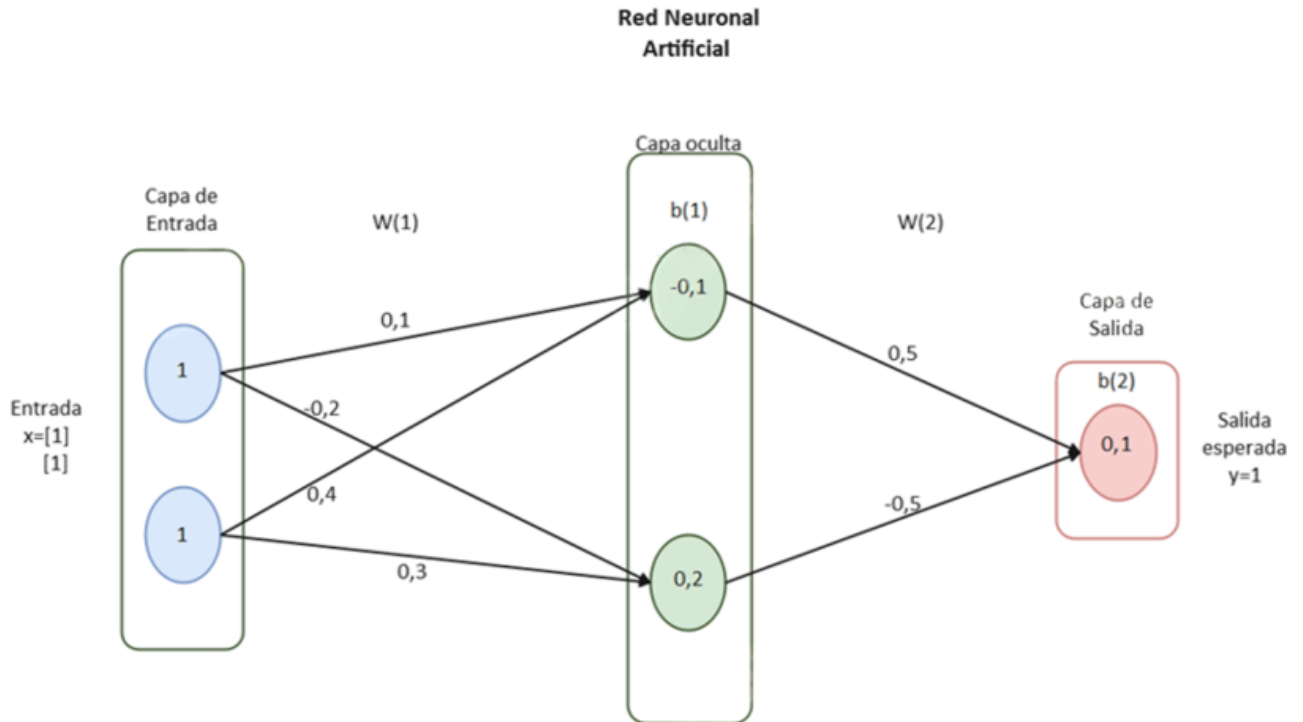
- $Sesgos = b^{(1)} = \begin{bmatrix} -0,1 \\ 0,2 \end{bmatrix}$

Conexión entre la capa oculta y la capa de salida, la capa oculta tiene dos neuronas y la capa de salida tiene una neurona, los pesos de esta conexión se representan como una fila de dos elementos:

- $Pesos = W^{(2)} = [0,5 \quad -0,5]$
- $W_1^{(2)} = 0,5$ : conecta la neurona oculta uno con la salida.
- $W_2^{(2)} = -0,5$ : Conecta la neurona oculta dos con la salida.

El sesgo de la capa de salida es un único valor:

- $Sesgo = b^{(2)} = [0,1]$



Fuente: Elaboración propia

Figura 2-10 Estructura de la red neuronal requerida.

## 2.8.2 Función de activación

Se utilizará la función sigmoide como función de activación para las neuronas, ya que éste es un problema de clasificación binaria, como lo es una compuerta lógica.

La función sigmoide transforma cualquier valor real de entrada en un valor de salida entre cero y uno, lo que permite interpretar el resultado como una probabilidad.

Su expresión matemática es:

$$\text{Función de activación sigmoide} = f(z) = \frac{1}{1 + e^{-z}}$$

## 2.8.3 Propagación hacia adelante

En este punto se alimentará la red neuronal artificial, donde se espera como resultado final un 1, ya que se trata de una compuerta AND con entradas 1 y 1.

Primero se calculará la salida de la capa oculta  $a^{(1)}$ , para lo cual es necesario obtener la entrada ponderada de la capa oculta  $z^{(1)}$ . Esta se calcula multiplicando la entrada por los pesos correspondientes y sumando el sesgo:

$$z^{(1)} = W_n^{(1)} \times x + b^{(1)} = \begin{bmatrix} 0,1 & 0,4 \\ -0,2 & 0,3 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -0,1 \\ 0,2 \end{bmatrix} = \begin{bmatrix} (0,1 \times 1) + (0,4 \times 1) - 0,1 \\ (-0,2 \times 1) + (0,3 \times 1) + 0,2 \end{bmatrix}$$

$$z^{(1)} = \begin{bmatrix} 0,4 \\ 0,3 \end{bmatrix}$$

Luego se aplica la función de activación sigmoide para obtener la salida de la capa oculta:

$$a^{(1)} = f(z^{(1)}) = \begin{bmatrix} f(0,4) \\ f(0,3) \end{bmatrix} = \begin{bmatrix} \frac{1}{1 + e^{-0,4}} \\ \frac{1}{1 + e^{-0,3}} \end{bmatrix} \approx \begin{bmatrix} 0,5986 \\ 0,5744 \end{bmatrix}$$

$$a^{(1)} \approx \begin{bmatrix} 0,5986 \\ 0,5744 \end{bmatrix}$$

A continuación, se calcula la entrada ponderada de la capa de salida  $z^{(2)}$ , multiplicando la salida de la capa oculta por los pesos hacia la capa de salida y sumando el sesgo:

$$z^{(2)} = W^{(2)} * a^{(1)} + b^{(2)} = [0,5 \quad -0,5] * \begin{bmatrix} 0,5986 \\ 0,5744 \end{bmatrix} + [0,1]$$

$$z^{(2)} = [(0,5 * 0,5986) + (-0,5 * 0,5744) + 0,1] = [0,2993 - 0,2872 + 0,1]$$

$$z^{(2)} = 0,1121$$

Aplicando la función sigmoide a  $z^{(2)}$  se obtiene la salida final de la red:

$$y^{(1)} = a^{(2)} = f(z^{(2)}) = [0,1121] = \left[ \frac{1}{1 + e^{-0,1121}} \right] \approx [0,5279]$$

Con los pesos y sesgos iniciales, la red neuronal artificial predice una salida aproximada de 0,5279 para la entrada [1, 1]. Dado que la salida esperada para una compuerta AND es 1, será necesario ajustar los pesos y sesgos para que la red se aproxime mejor al valor correcto mediante retropropagación.

### 2.8.4 Función de pérdida

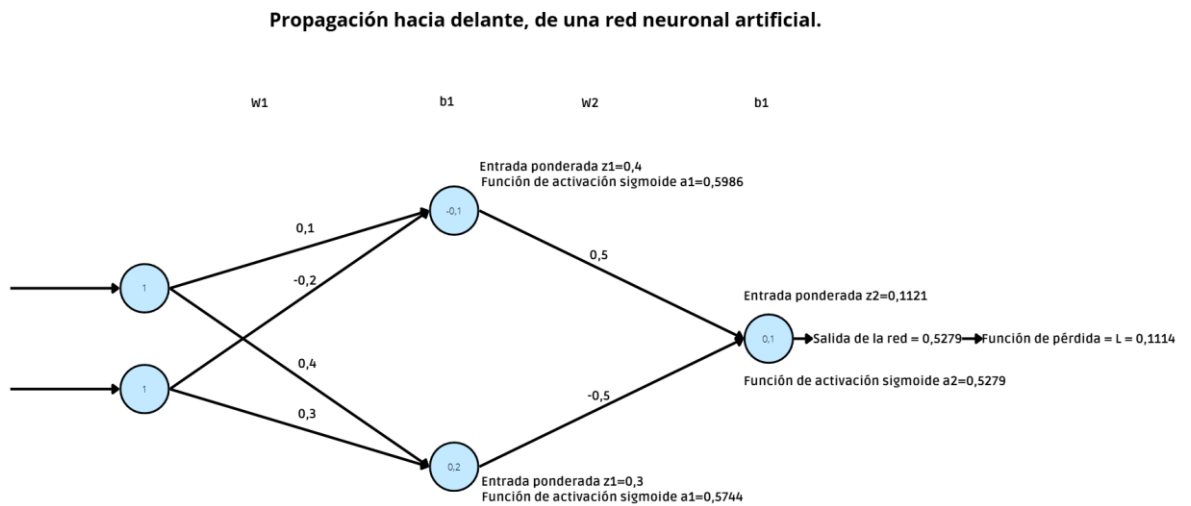
Para cuantificar la diferencia entre la salida predicha por la red  $y^{(1)}$  y la salida esperada  $y$ , se utiliza la función de pérdida. En este caso, se aplicará el error cuadrático medio (MSE), que permite medir de forma sencilla qué tan lejos está la predicción del valor real:

$$L(y^{(1)}, y) = \frac{1}{2}(y^{(1)} - y)^2$$

Sustituyendo los valores:

$$L = \frac{1}{2}(0,5279 - 1)^2 = \frac{1}{2}(-0,4721)^2 \approx 0,1114$$

El valor de la función de pérdida es aproximadamente 0,1114, lo cual indica que aún existe una diferencia considerable entre la predicción de la red y el valor esperado, a continuación, se expone una imagen con el resumen de la propagación hacia delante en la figura 2-11.



Fuente: Elaboración propia

Figura 2-11 Propagación hacia delante.

El valor de la función de pérdida es aproximadamente 0,1114, lo cual indica que aún existe una diferencia considerable entre la predicción de la red y el valor esperado. Ahora que se cuenta con el resultado de la función de pérdida, es posible comenzar el proceso de entrenamiento de la red. Esto se realizará a través del algoritmo de retropropagación, con lo cual se calcularán los gradientes de la pérdida respecto a todos los pesos y sesgos. Luego, dichos parámetros serán actualizados utilizando el método de descenso de gradiente. Con el objetivo de minimizar la pérdida o error y mejorar la predicción del modelo.

### 2.8.5 Propagación hacia atrás

La propagación hacia atrás (retropropagación) es el proceso, mediante el cual se calcula el gradiente de la función de pérdida con respecto a los pesos y sesgos de la red. Estos gradientes indican la dirección en la que deben ajustarse los parámetros para reducir el error del modelo. Este procedimiento se realiza desde la capa de salida hacia las capas anteriores, aplicando la regla de la cadena.

Recapitulando los resultados obtenidos en la propagación hacia adelante:

- Entrada  $(x) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Salida esperada  $(y) = [1]$
- Salida de la capa oculta  $a^{(1)} \approx \begin{bmatrix} 0,5986 \\ 0,5744 \end{bmatrix}$
- Salida de la red  $y^{(1)} \approx [0,5279]$
- Función de pérdida  $L \approx 0,1114$

A continuación, se procederá a calcular los gradientes parciales de la función de pérdida con respecto a los parámetros  $W_n^{(2)}$ ,  $b^{(2)}$ ,  $W_n^{(1)}$ ,  $b_n^{(1)}$ , lo que permitirá ajustarlos usando descenso de gradiente futura etapa de entrenamiento.

### 2.8.5.1 Gradiente de la pérdida con respecto a la salida de la capa de salida

El primer paso en el proceso de retropropagación es calcular cuánto cambia la función de pérdida ante un pequeño cambio en la salida de la red. Esto se obtiene derivando la función de pérdida con respecto a la salida predicha por el modelo.

Se utilizará la función de pérdida del error cuadrático medio (MSE), definida como:

$$L = \frac{1}{2}(y^{(1)} - y)^2$$

Donde:

- $y^{(1)}$ : es la salida predicha por la red neuronal.
- $y$ : es la salida real o esperada.
- $L$ : representa que tan lejos está la predicción del valor real.

Para obtener el gradiente con respecto a  $y^{(1)}$ , se deriva la función:

$$\frac{\partial L}{\partial y^{(1)}} = \frac{\partial}{\partial y^{(1)}} \left[ \frac{1}{2}(y^{(1)} - y)^2 \right]$$

Aplicando la regla de la cadena:

- El 2 baja por la potencia, y se reduce el exponente a 1.
- Se multiplica por la derivada interna de  $(y^{(1)} - y)$ , que es 1 (ya que  $y$  es constante).

$$\frac{\partial L}{\partial y^{(1)}} = [(y^{(1)} - y)]$$

Sustituyendo los valores:

$$\frac{\partial L}{\partial y^{(1)}} = (y^{(1)} - y) = 0,5279 - 1 = -0,4721$$

Este resultado indica que la salida predicha está por debajo del valor esperado, y que se debe incrementar la predicción para reducir el error.

### 2.8.5.2 Gradiente de la salida de la capa de salida con respecto a su entrada $z^{(2)}$

Este paso calcula el gradiente de la función de activación de la neurona de salida, es decir, cómo varía la salida activada  $y^{(1)}$  frente a un pequeño cambio en su entrada  $z^{(2)}$ .

- $z^{(2)}$ : es la entrada de la neurona de salida.
- $y^{(1)} = f(z^{(2)})$ : es la salida, la activación de esa neurona usando la función sigmoide.

La función sigmoide está definida como:

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

Para derivar, primero la escribimos como una potencia negativa:

$$f(z) = (1 + e^{-z})^{-1}$$

Luego aplicamos la regla de la cadena, ya que se trata de una función compuesta. La derivada general de una función del tipo  $(g(z))^{-1}$ :

$$\frac{\partial}{\partial z} (g(z))^{-1} = -1 \times (g(z))^{-2} \times g'(z)$$

En nuestro caso,  $g(z) = 1 + e^{-z}$ , al derivar  $e^{-z}$ , el exponente  $-z$  aporta un signo negativo por la regla de la cadena, por lo tanto:

$$g'(z) = \frac{\partial}{\partial z} (1 + e^{-z}) = -e^{-z}$$

Sustituyendo en la fórmula:

$$f'(z) = -1 \times (1 + e^{-z})^{-2} \times (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Ahora bien, esta expresión también puede escribirse de forma más compacta en función de la propia sigmoide. Sabemos que:

$$\text{Función sigmoide: } f(z) = \frac{1}{1 + e^{-z}}$$

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

Multiplicamos ambas expresiones:

$$f(z) \times (1 - f(z)) = \left( \frac{1}{1 + e^{-z}} \right) \times \left( \frac{e^{-z}}{1 + e^{-z}} \right) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Por lo tanto, la derivada de la función sigmoide se puede expresar de forma simple como:

$$f'(z) = f(z)(1 - f(z))$$

Aplicando esta derivada a la activación de la capa de salida, se tiene:

$$\frac{\partial y^{(1)}}{\partial z^{(2)}} = f'(z^{(2)}) = f(z^{(2)}) (1 - f(z^{(2)})) = y^{(1)}(1 - y^{(1)})$$

Con los valores calculados anteriormente:

$$y^{(1)} = 0,5279$$

Entonces:

$$\frac{\partial y^{(1)}}{\partial z^{(2)}} = 0,5279 \times (1 - 0,5279) = 0,5279 \times 0,4721 \approx 0,2492$$

Este valor representa el gradiente de la activación respecto a su entrada, y es clave para continuar con el cálculo de la retropropagación en la red neuronal.

### 2.8.5.3 Gradiente de la pérdida con respecto a la entrada de la capa de salida $z^{(2)}$

En este paso se aplica la regla de la cadena para combinar los dos gradientes calculados anteriormente: el gradiente de la función de pérdida con respecto a la salida activación  $y^{(1)}$ , y el gradiente de la salida activada con respecto a su entrada  $z^{(2)}$ . Esto permite determinar cómo cambia la pérdida cuando hay una variación en la entrada de la función de activación de la capa de salida.

$$\frac{\partial L}{\partial z^{(2)}} = \frac{\partial L}{\partial y^{(1)}} \times \frac{\partial y^{(1)}}{\partial z^{(2)}}$$

Sustituyendo los valores previamente obtenidos:

$$\frac{\partial L}{\partial y^{(1)}} = -0,4721, \quad \frac{\partial y^{(1)}}{\partial z^{(2)}} = 0,2492$$

Entonces:

$$\frac{\partial L}{\partial z^{(2)}} = -0,4721 \times 0,2492 = -0,1176$$

Este valor indica cómo afecta un pequeño cambio en la entrada de la neurona de salida (antes de aplicar la función de activación) al valor de la pérdida. Es uno de los elementos clave para calcular los gradientes de los pesos y sesgos conectados a esta neurona durante el proceso de retropropagación.

#### 2.8.5.4 Gradiente de la pérdida con respecto a los pesos de la capa de salida $W_n^{(2)}$

Los pesos  $W_n^{(2)}$  conectan la salida de la capa oculta  $a^{(1)}$  con la entrada de la neurona de salida  $z^{(2)}$ , según la relación:

$$z^{(2)} = W_n^{(2)} \times a^{(1)} + b^{(2)}$$

Para determinar cómo cambia la pérdida  $L$  en función de los pesos  $W^{(2)}$ , se utiliza nuevamente la regla la cadena:

$$\frac{\partial L}{\partial W_n^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial W_n^{(2)}}$$

La derivada de  $z^{(2)}$  con respecto a  $W_n^{(2)}$  corresponde al vector de activación de la capa oculta, transpuesto para que las dimensiones sean compatibles en la multiplicación:

$$(a^{(1)})^T = \frac{\partial z^{(2)}}{\partial W_n^{(2)}} = \frac{\partial (W_n^{(2)} \times a^{(1)} + b^{(2)})}{\partial W_n^{(2)}} = a^{(1)T} = [0,5986 \quad 0,5744]$$

La notación  $T$  indica que se está utilizando el vector transpuesto, es decir, como fila en lugar de columna, lo cual es necesario para que la operación con el escalar  $\frac{\partial L}{\partial z^{(2)}}$  sea válida.

Sustituyendo los valores:

$$\frac{\partial L}{\partial W_n^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial W_n^{(2)}}$$

$$\frac{\partial L}{\partial W_n^{(2)}} = -0,1176 \times [0,5986 \quad 0,5744]$$

$$\frac{\partial L}{\partial W_n^{(2)}} = [(-0,1176 \times 0,5986) \quad (-0,1176 \times 0,5744)]$$

$$\frac{\partial L}{\partial W_1^{(2)}} = -0,0703 \qquad \frac{\partial L}{\partial W_2^{(2)}} = -0,0675$$

Estos vectores representan el gradiente de la pérdida con respecto a cada uno de los pesos que conectan las neuronas de la capa oculta con la neurona de salida. Cada valor indica cuánto debe ajustarse cada peso para minimizar la función de pérdida durante el entrenamiento.

#### 2.8.5.5 Gradiente de la pérdida con respecto al sesgo de la capa de salida $b^{(2)}$

El sesgo  $b^{(2)}$  también influye directamente en la entrada de la neurona de salida  $z^{(2)}$ , ya que forma parte de la operación:

$$z^{(2)} = W_n^{(2)} \times a^{(1)} + b^{(2)}$$

Como el sesgo se suma directamente a la salida de la capa oculta, un pequeño cambio en  $b^{(2)}$  afecta a  $z^{(2)}$  en la misma proporción. Por tanto, la derivada de  $z^{(2)}$  respecto al sesgo es simplemente:

$$\frac{\partial z^{(2)}}{\partial b^{(2)}} = 1$$

Aplicando la regla de la cadena:

$$\frac{\partial L}{\partial b^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial b^{(2)}} = -0,1176 \times 1 = -0,1176$$

Este valor indica cuánto debe ajustarse el sesgo de la neurona de salida para reducir el error durante el entrenamiento. Un valor negativo sugiere que disminuir ligeramente el sesgo puede ayudar a minimizar la función de pérdida.

#### 2.8.5.6 Gradiente de la entrada de la capa de salida $z^{(2)}$ con respecto a la salida de la capa oculta

$a_n^{(1)}$

En este paso, comienza la retropropagación del error hacia la capa oculta. Para ello, primero se determina cómo cambia la entrada de la neurona de salida  $z^{(2)}$  con respecto a la salida de la capa oculta  $a_n^{(1)}$ .

Recordando la ecuación de la neurona de salida:

$$z^{(2)} = W_n^{(2)} \times a_n^{(1)} + b^{(2)}$$

Derivamos esta expresión respecto a  $a^{(1)}$ :

$$\frac{\partial z^{(2)}}{\partial a_n^{(1)}} = \frac{\partial (W_n^{(2)} \times a_n^{(1)} + b^{(2)})}{\partial a_n^{(1)}} = (\partial W_n^{(2)})^T$$

En este caso, los pesos de la capa de salida son:

$$W_n^{(2)} = [0,5 \quad -0,5]$$

Por lo tanto, su transpuesta (necesaria para conservar la consistencia en las dimensiones al retropropagar el gradiente) es:

$$(W_n^{(2)})^T = \begin{bmatrix} 0,5 \\ -0,5 \end{bmatrix}$$

$$\frac{\partial z^{(2)}}{\partial a_1^{(1)}} = 0,5 \quad \frac{\partial z^{(2)}}{\partial a_2^{(1)}} = -0,5$$

Esto indica que la tasa de cambio de  $z^{(2)}$  respecto a cada componente de  $a_n^{(1)}$  está dada directamente por los pesos asociados, ya que la salida de la capa oculta es multiplicada por estos valores en la propagación hacia adelante.

#### 2.8.5.7 Gradiente de la salida de la capa oculta $a_n^{(1)}$ con respecto a su entrada $z_n^{(1)}$

Este paso es similar al realizado en el punto 2.8.5.2, donde calculamos la derivada de la función de activación sigmoide con respecto a su entrada. Como la capa oculta contiene dos neuronas, obtendremos una matriz diagonal que contiene las derivadas individuales para cada neurona.

- $z_n^{(1)}$  representa la entrada de la función de activación sigmoide en la capa oculta.
- $a_n^{(1)} = f(z^{(1)})$  es la salida activa de dicha capa.

La derivada de la función sigmoide es:

$$f'(z) = f(z)(1 - f(z))$$

Debido a que hay dos neuronas, se debe calcular la derivada para cada una por separado.

Para la neurona 1:

$$z_1^{(1)} = 0,4 \rightarrow f(0,4) = \frac{1}{1 + e^{0,4}} \rightarrow f(0,4) \approx 0,5986$$

$$f'(0,4) \approx 0,5986 \times (1 - 0,5986) \approx 0,2402$$

Para la neurona 2:

$$z_2^{(1)} = 0,3 \rightarrow f(0,3) = \frac{1}{1 + e^{0,3}} \rightarrow f(0,3) \approx 0,5744$$

$$f'(0,3) \approx 0,5744 \times (1 - 0,5744) \approx 0,2444$$

Como cada neurona actúa de manera independiente, no existe interacción entre ellas en este paso, por lo tanto, el resultado se representa en forma de matriz diagonal:

$$\frac{\partial a_n^{(1)}}{\partial z_n^{(1)}} = \begin{bmatrix} f'(z_1^{(1)}) & 0 \\ 0 & f'(z_2^{(1)}) \end{bmatrix} = \begin{bmatrix} 0,2402 & 0 \\ 0 & 0,2444 \end{bmatrix}$$

Esta matriz permite propagar el error hacia atrás desde la capa de salida, modulando cómo se deben ajustar los pesos de la capa oculta, según el impacto individual de cada neurona en el error final.

#### 2.8.5.8 Gradiente de la pérdida con respecto a la entrada de la capa oculta $z_n^{(1)}$

Este paso es importante para entender cómo cada neurona de la capa oculta contribuye al error total de la red. Para esto se aplica nuevamente la regla de la cadena, combinando las tres derivadas obtenidas en los puntos anteriores.

$$\frac{\partial L}{\partial z_n^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial a_n^{(1)}} \times \frac{\partial a_n^{(1)}}{\partial z_n^{(1)}}$$

Los valores ya calculados son:

- $\frac{\partial L}{\partial z^{(2)}} = -0,1176$ . Resultado calculado en el punto 2.8.5.3.
- $\frac{\partial z^{(2)}}{\partial a_n^{(1)}} = (W^{(2)})^T = \begin{bmatrix} 0,5 \\ -0,5 \end{bmatrix}$ . Resultado calculado en el punto 2.8.5.6.
- $\frac{\partial a_n^{(1)}}{\partial z_n^{(1)}} = \begin{bmatrix} 0,2402 & 0 \\ 0 & 0,2444 \end{bmatrix}$ . Resultado calculado en el punto 2.8.5.7.

Se multiplica la matriz por el vector y luego por el escalar.

$$\frac{\partial L}{\partial z_n^{(1)}} = \begin{bmatrix} 0,2402 & 0 \\ 0 & 0,2444 \end{bmatrix} \times \begin{bmatrix} 0,5 \\ -0,5 \end{bmatrix} \times (-0,1176)$$

Primero la multiplicación matriz-vector:

$$\frac{\partial L}{\partial z_n^{(1)}} = \begin{bmatrix} 0,2402 \times 0,5 + 0 \times (-0,5) \\ 0 \times 0,5 + 0,2444 \times (-0,5) \end{bmatrix} = \begin{bmatrix} 0,1201 \\ -0,1222 \end{bmatrix}$$

Luego se multiplica por el escalar:

$$\frac{\partial L}{\partial z_n^{(1)}} = \begin{bmatrix} 0,1201 \\ -0,1222 \end{bmatrix} \times (-0,1176) \quad \frac{\partial L}{\partial z_1^{(1)}} = -0,0141 \quad \frac{\partial L}{\partial z_2^{(1)}} = 0,0144$$

Este resultado indica que la primera neurona de la capa oculta tiene un pequeño impacto negativo en el error mientras que la segunda neurona tiene un pequeño impacto positivo en el error. Esto permitirá ajustar sus respectivos pesos y sesgos en la siguiente etapa de retropropagación.

#### 2.8.5.9 Gradiente de la pérdida con respecto a los pesos de la capa de entrada $W_n^{(1)}$

Los pesos  $W_n^{(1)}$  conectan la capa de entrada  $x$  con la entrada de la capa oculta  $z_n^{(1)}$ . Por lo tanto, el gradiente:

$$\frac{\partial L}{\partial W_n^{(1)}}$$

Indica cuanto influye cada uno de estos pesos en el error total de la red. Este gradiente se obtiene usando nuevamente la regla de la cadena.

$$\frac{\partial L}{\partial W_n^{(1)}} = \frac{\partial L}{\partial z_n^{(1)}} \times \frac{\partial z_n^{(1)}}{\partial W_n^{(1)}}$$

De pasos anteriores tenemos:

- $\frac{\partial L}{\partial z_n^{(1)}} = \begin{bmatrix} -0,0141 \\ 0,0144 \end{bmatrix}$ . Resultado calculado en el punto 2.8.5.8, (cada punto representa el gradiente para cada neurona oculta).
- $\frac{\partial z_n^{(1)}}{\partial W_n^{(1)}} = x^T$ , y como  $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ , su transpuesta es:

$$x^T = [1 \quad 1]$$

Multiplicamos el vector columna por el vector fila:

$$\frac{\partial L}{\partial W_n^{(1)}} = \begin{bmatrix} -0,0141 \\ 0,0144 \end{bmatrix} \times [1 \quad 1] = \begin{bmatrix} -0,0141 \times 1 & -0,0141 \times 1 \\ 0,0144 \times 1 & 0,0144 \times 1 \end{bmatrix} = \begin{bmatrix} -0,0141 & -0,0141 \\ 0,0144 & 0,0144 \end{bmatrix}$$

$$\frac{\partial L}{\partial W_{1\ 1}^{(1)}} = -0,0141 \quad \frac{\partial L}{\partial W_{1\ 2}^{(1)}} = -0,0141 \quad \frac{\partial L}{\partial W_{2\ 1}^{(1)}} = 0,0144 \quad \frac{\partial L}{\partial W_{2\ 2}^{(1)}} = 0,0144$$

La matriz resultante tiene 2 filas (una por cada neurona de la capa oculta) y 2 columnas (una por cada entrada).

La fila 1, columna 1 y 2: indican que los pesos que conectan la entrada 1 y 2 con la neurona oculta 1 deberían reducirse ligeramente, ya que contribuyen negativamente al error.

La fila 2, columna 1 y 2: muestran que los pesos que conectan ambas entradas con la neurona oculta 2 deberían aumentarse ligeramente, por que ayudaron a reducir el error.

### 2.8.5.10 Gradiente de la pérdida con respecto a los sesgos de la capa oculta $b_n^{(1)}$

El sesgo  $b_n^{(1)}$  también influye directamente en la entrada de la capa oculta  $z_n^{(1)}$ , ya que sabemos que:

$$z^{(1)} = W_n^{(1)} \times x + b_n^{(1)}$$

Esto quiere decir que el sesgo se suma directamente al resultado del producto entre los pesos y las entradas. Por lo tanto, si aumentamos el sesgo en una unidad, el valor de  $z_n^{(1)}$  también aumentara en una unidad. En términos de derivada:

$$\frac{\partial z_n^{(1)}}{\partial b_n^{(1)}} = [1 \quad 1]$$

Aplicando la regla de la cadena:

$$\frac{\partial L}{\partial b_n^{(1)}} = \frac{\partial L}{\partial z_n^{(1)}} \times \frac{\partial z_n^{(1)}}{\partial b_n^{(1)}}$$

De pasos anteriores ya tenemos:

- $\frac{\partial L}{\partial z_n^{(1)}} = \begin{bmatrix} -0,0141 \\ 0,0144 \end{bmatrix}$ . Resultado obtenido en el punto 2.8.5.8.

- $\frac{\partial z_n^{(1)}}{\partial b_n^{(1)}} = [1 \quad 1]$ . Resultado obtenido en este punto.

Entonces:

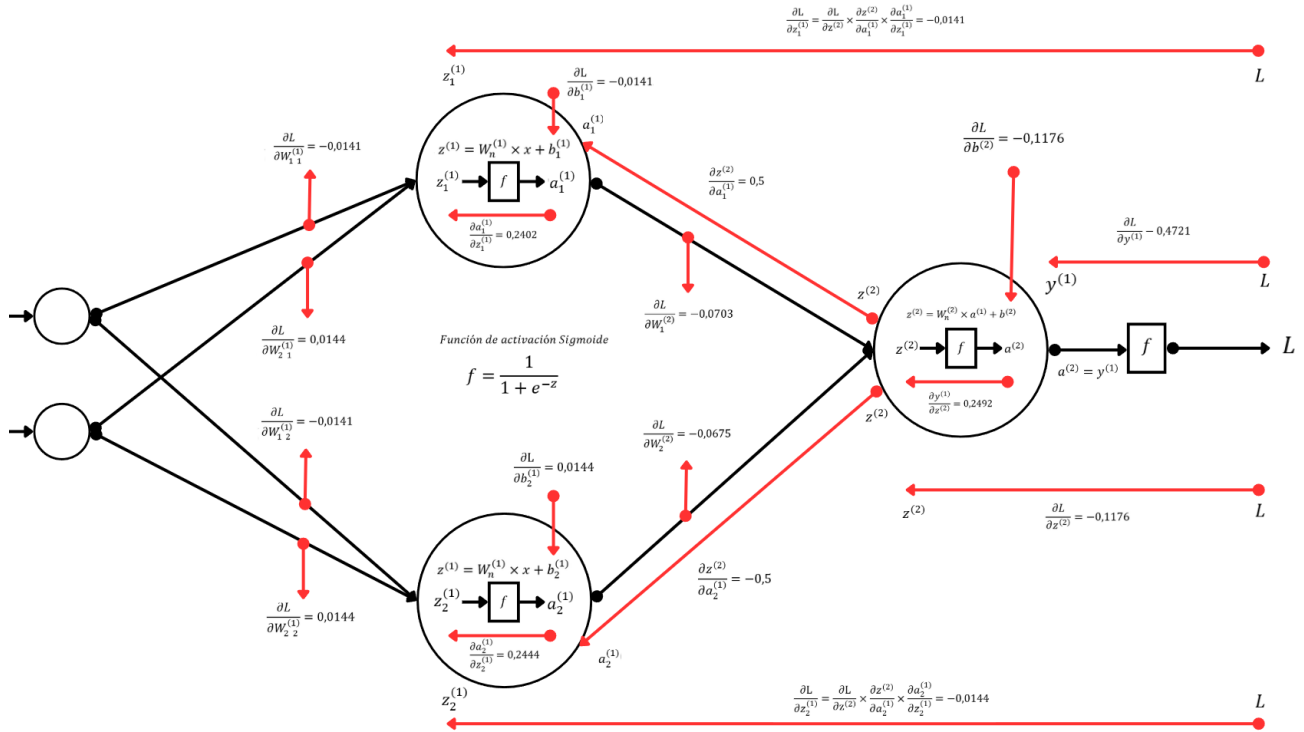
$$\frac{\partial L}{\partial b_2^{(1)}} = 0,0144 [-0,0141 \quad 0,0144] \times [1 \quad 1]$$

$$\frac{\partial L}{\partial b_n^{(1)}} = 0,0144[-0,0141 \times 1 \quad 0,0144 \times 1] = [-0,0141 \quad 0,0144]$$

El resultado indica que el sesgo de la neurona uno en la capa oculta ( $-0,0141$ ) debería reducirse levemente, mientras que el de la neurona 2 ( $0,0144$ ) debería incrementarse un poco. Esto muestra cómo cada sesgo puede ser ajustado para ayudar a reducir el error total en la salida.

Con esta explicación completa del proceso de propagación hacia adelante y retropropagación, se pudo observar como una red neuronal simple ajusta sus pesos y sesgos para reducir el error entre su predicción y el resultado esperado, a continuación, en la figura 2-12, se muestra un resumen de cómo afecta la retropropagación en una red neuronal artificial.

Retropropagación, de una red neuronal artificial.



Fuente: Elaboración propia.

Figura 2-12 Retropropagación de una red neuronal artificial.

2.8.6 Actualización de parámetros mediante descenso de gradiente

Una vez calculados los gradientes de la función de pérdida respecto a los pesos y sesgos mediante el proceso de retropropagación, se procede a la actualización de estos parámetros con el objetivo de reducir el error de la red neuronal. Este ajuste se realiza utilizando el algoritmo de descenso de gradiente, el cual modifica los parámetros en la dirección opuesta al gradiente de la función de pérdida. De esta forma se busca minimizar progresivamente el valor del error, mejorando la capacidad de la red para aproximar la salida esperada. La magnitud del ajuste está controlada por la tasa de aprendizaje  $\eta$ , que determina cuanto se modifican los parámetros en cada iteración del entrenamiento.

Este proceso constituye una etapa fundamental en el entrenamiento de redes neuronales, ya que permite que el modelo aprenda a partir de los errores cometidos y ajuste sus predicciones de manera iterativa.

Para esto se utiliza la siguiente regla:

$$W_{nuevo} = W_{viejo} \eta \frac{\delta L}{\delta W}$$

$$b_{nuevo} = b_{viejo} \eta \frac{\delta L}{\delta b}$$

Donde  $\eta = 0,1$

### 2.8.6.1 Actualización de pesos de la capa de salida

Gradientes:

$$\frac{\delta L}{\delta W^2} = [-0,0703 \quad -0,0675]$$

Pesos originales:

$$W^2 = [0,5 \quad -0,5]$$

Actualización con  $\eta = 0,1$ :

$$W_{nuevo}^2 = W^2 - \eta \frac{\delta L}{\delta W^2}$$

$$W_1^2 = 0,5 - 0,1 \times (-0,0703) = 0,50703$$

$$W_2^2 = -0,5 - 0,1 \times (-0,0675) = -0,49325$$

Nuevo peso de la capa de salida:

$$W^2 = [0,50703 \quad -0,49325]$$

### 2.8.6.2 Actualización del sesgo de la capa de salida

Gradiente:

$$\frac{\delta L}{\delta b^2} = -0,1176$$

Sesgo original:

$$b^2 = 0,1$$

Nuevo sesgo de la capa de salida:

$$b_{nuevo}^2 = 0,1 - 0,1 [-0,1176] = 0,11176$$

### 2.8.6.3 Actualización de pesos de la capa oculta

Gradiente:

$$\frac{\delta L}{\delta W^1} = \begin{bmatrix} -0,0141 & -0,0141 \\ 0,0144 & 0,0144 \end{bmatrix}$$

Pesos originales:

$$W^1 = \begin{bmatrix} 0,1 & 0,4 \\ -0,2 & 0,3 \end{bmatrix}$$

Actualización con  $\eta = 0,1$ :

$$W_{nuevo}^1 = W^1 - \eta \frac{\delta L}{\delta W^1}$$

Cálculo elemento a elemento:

$$0,1 - 0,1 \times (-0,0141) = 0,10141$$

$$0,4 - 0,1 \times (-0,0141) = 0,40141$$

$$-0,2 - 0,1 \times (-0,0144) = -0,20144$$

$$0,3 - 0,1 \times (-0,0144) = 0,29856$$

Nuevo peso de la capa oculta:

$$W^1 = \begin{bmatrix} 0,1014 & 0,4014 \\ -0,2014 & 0,2986 \end{bmatrix}$$

#### 2.8.6.4 Actualización de los sesgos de la capa oculta

Gradiente:

$$\frac{\delta L}{\delta b^1} = [-0,0141 \quad 0,0144]$$

Sesgos originales:

$$b^1 = [-0,1 \quad 0,2]$$

Actualización de sesgos:

$$b_{nuevo}^1 = b^1 - \eta \frac{\delta L}{\delta b^1}$$

$$-0,1 - 0,1 \times (-0,0141) = -0,09859$$

$$0,2 - 0,1 \times (-0,0144) = 0,19856$$

Nuevos sesgos de la capa oculta:

$$b^1 = [-0,0986 \quad 0,19856]$$

#### 2.8.6.5 Nueva propagación hacia adelante

Entrada:

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Capa oculta:

$$z^1 = W^1 \times x + b^1$$

Neurona 1:

$$z_1 = 0,1014 + 0,4014 - 0,0986 = 0,4042$$

Neurona 2:

$$z_2 = -0,2014 + 0,2986 + 0,1986 = 0,2958$$

Función sigmoide:

$$a(z) = \frac{1}{1 + e^{-z}}$$

$$a_1 = \frac{1}{1 + e^{-0,4042}} = 0,5997$$

$$a_1 = \frac{1}{1 + e^{-0,2958}} = 0,5734$$

$$a^1 = \begin{bmatrix} 0,5997 \\ 0,5734 \end{bmatrix}$$

Capa de salida:

$$z^2 = W^2 \times a^1 + b^2$$

$$z^2 = 0,50703 \times (0,5997) - 0,49325 \times (0,5734) + 0,11176$$

$$z^2 = 0,3040 - 0,2827 + 0,1118 = 0,1331$$

Función sigmoide de la nueva salida de la red:

$$y^1 = \frac{1}{1 + e^{-z}}$$

$$y^1 = \frac{1}{1 + e^{-0,1331}}$$

$$y^1 = 0,5332$$

#### 2.8.6.6 Nueva función de pérdida

$$L = \frac{1}{2} \times (y^1 - y)^2$$

$$L = \frac{1}{2} \times (0,5332 - 1)^2$$

$$L = 0,1089$$

### 2.8.6.7 Comparación de resultados

Pérdida inicial:

$$L = 0,1114$$

Pérdida actual:

$$L = 0,1089$$

Se observa que la pérdida disminuye y la salida se aproxima al valor esperado, lo que confirma que el algoritmo de retropropagación, junto con el método de descenso de gradiente, permite ajustar correctamente los parámetros de la red neuronal. Para verificar el correcto funcionamiento del algoritmo de entrenamiento, se realizó una actualización de los pesos y sesgos utilizando descenso de gradiente con una tasa de aprendizaje  $\eta = 0,1$ . Posteriormente, se repitió la propagación hacia adelante con los parámetros actualizados, obteniéndose una salida de 0,5332, en comparación con el valor inicial de 0,5279. La función de pérdida disminuyó de 0,1114 a 0,1089, lo que demuestra que el ajuste de los parámetros reduce el error y mejora la aproximación al valor esperado.

Este ejercicio fue realizado intencionalmente utilizando una sola muestra de entrenamiento, una única iteración y una red neuronal con solo dos neuronas en la capa oculta, con fines prácticos y educativos. En un escenario real, el entrenamiento de una red neuronal se realiza utilizando grandes cantidades de datos y múltiples iteraciones, denominadas épocas, repitiendo este mismo procedimiento de propagación hacia adelante, cálculo del error, retropropagación y actualización de parámetros. El objetivo principal de este desarrollo fue demostrar el principio fundamental del aprendizaje en redes neuronales, el cual consiste en utilizar el gradiente de la función de pérdida para guiar el ajuste progresivo de los pesos y sesgos internos de la red.

Como siguiente etapa, este mismo procedimiento será implementado en Python, lo que permitirá automatizar el proceso de entrenamiento y observar cómo la red neuronal mejora gradualmente sus predicciones hasta alcanzar una solución adecuada.

## 2.9 EJERCICIO COMPUTACIONAL

En este punto se llevará a cabo un ejercicio computacional utilizando el lenguaje de programación Python. El objetivo es entrenar una red neuronal artificial para que aprenda el comportamiento lógico de una compuerta AND, esta vez de forma automática, aprovechando la capacidad de las computadoras para realizar miles de ciclos de entrenamiento en pocos segundos.

A través de este proceso computacional, se podrían encontrar los pesos y sesgos óptimos de manera eficiente, permitiendo que la red ajuste sus parámetros internos hasta lograr una predicción precisa, cabe destacar que la red es muy susceptible a pequeños cambios en los pesos y sesgos, lo cual provoca una infinidad de combinaciones que cumplen con el objetivo final.

A continuación, se presentará y explicará paso a paso el código utilizado para el entrenamiento de esta red neuronal artificial, abordando desde la definición de las capas hasta el cálculo de la función de pérdida, pasando por el proceso de retropropagación y actualización de pesos y sesgos.

Este ejercicio servirá como una extensión práctica de todo lo que se explicó previamente en forma manual.

### 2.9.1 Bibliotecas

Para comenzar con el desarrollo del programa, se incorporan dos bibliotecas fundamentales que facilitarán el manejo de datos numéricos y la visualización de resultados, tal como se observa en la Figura 2-13.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
```

Fuente: Elaboración propia.

Figura 2-13 Bibliotecas.

- Numpy: Es una biblioteca esencial en Python para el trabajo con vectores y matrices. Permite realizar operaciones matemáticas complejas de manera rápida y eficiente, como multiplicación de matrices, sumas, productos, derivadas, entre otras. En el contexto de este proyecto, resulta indispensable ya que las entradas, pesos y sesgos de la red neuronal se manejan como matrices y vectores. Durante la propagación hacia adelante y el proceso de retropropagación, se realizan

numerosas operaciones matriciales, las cuales son gestionadas de forma óptima por esta biblioteca.

- Matplotlib.pyplot: Esta biblioteca está diseñada para la visualización de datos y gráficos. Se utilizan aquí para graficar la evolución del error (función de pérdida) durante el entrenamiento de la red. Gracias a estas graficas, podemos observar visualmente como disminuye el error con cada época, confirmando que la red neuronal está aprendiendo correctamente.

### 2.9.2 Función de activación sigmoide y su derivada

En esta sección se define la función de activación sigmoide, junto con su derivada, elementos esenciales para el funcionamiento y aprendizaje de la red neuronal como se muestra en la Figura 2-14.

```

4  # Función de activación sigmoide y su derivada
5  def sigmoid(z): 4 usages
6      return 1 / (1 + np.exp(-z))
7
8  def sigmoid_deriv(a): 2 usages
9      return a * (1 - a)
10

```

Fuente: Elaboración propia

Figura 2-14 Función de activación sigmoide y su derivada.

La función  $\text{sigmoid}(z)$  transforma cualquier valor real de entrada  $z$  en un valor entre 0 y 1.

Se logra mediante la fórmula:

$$\text{sigmoide}(z) = \frac{1}{(1 + \text{np.exp}(-z))} = \frac{1}{1 + e^{-z}}$$

- Si  $z$  es muy grande, la salida se acerca a 1.
- Si  $z$  es muy negativa, la salida se aproxima a 0.
- Si  $z$  es cercana a 0, la salida será cercana a 0,5.

Este comportamiento es útil para modelar probabilidades y generar salidas continuas en redes neuronales. La derivada de la sigmoide es necesaria para aplicar el algoritmo de retropropagación,

ya que indica qué tan sensible es la salida ante pequeños cambios en la entrada  $z$ . Esta derivada tiene forma:

$$\text{sigmoid}'(z) = \text{sigmoid}(z) \times (1 - \text{sigmoid}(z))$$

En el código se simplifica escribiendo la derivada como  $\text{sigmoid\_deriv}(a)$ , siendo  $a = \text{sigmoid}(z)$  previamente calculado.

Esto mejora la eficiencia del código y evita recalcular la función varias veces, esta función es fundamental para actualizar los pesos durante el entrenamiento, permitiendo que la red aprenda a través de los gradientes.

### 2.9.3 Datos iniciales de entrenamiento

En este apartado se define el conjunto de datos que se utilizará para entrenar la red neuronal. Se trata de un caso de aprendizaje supervisado, donde a la red se le enseña a comportarse como una compuerta lógica AND como se muestra en la figura 2-15.

```

11 # Datos de entrenamiento para la compuerta AND
12 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
13 y = np.array([[0], [0], [0], [1]])
14

```

Fuente: Elaboración propia

Figura 2-15 Datos iniciales de entrenamiento.

Como se observa, la matriz de entrada  $x$  contiene las cuatro combinaciones posibles de entrada binaria (0 y 1) para una compuerta AND.

El vector  $y$  representa las salidas esperadas para cada combinación de entrada.

La compuerta AND solo devuelve un 1 cuando ambas entradas son 1, por lo tanto:

- AND (0,0) → 0
- AND (0,1) → 0
- AND (1,0) → 0
- AND (1,1) → 1

### 2.9.4 Pesos y sesgos iniciales

Esta parte del código permite entender como inicia el entrenamiento de la red neuronal, ya que define los pesos y sesgos con los que la red comenzará a aprender. Estos valores son asignados de forma aleatoria, pero con un pequeño ajuste para que no sean ni muy grandes ni muy pequeños, como se puede ver en la Figura 2-16.

```

15 # Pesos y sesgos iniciales
16 np.random.seed(42) # Numeros iniciales aleatorios
17 W1 = np.random.rand(2, 2) - 0.5 # 2 neuronas de entrada a 2 neuronas ocultas
18 b1 = np.random.rand(1, 2) - 0.5
19 W2 = np.random.rand(2, 1) - 0.5 # 2 neuronas ocultas a 1 neurona de salida
20 b2 = np.random.rand(1, 1) - 0.5
21

```

Fuente: Elaboración propia

Figura 2-16 Pesos y sesgos iniciales.

- $np.random.seed(42)$ : Esta función establece una semilla aleatoria fija, lo que permite que los mismos números aleatorios se generen cada vez que se ejecute el programa. Esto es útil para que los resultados sean reproducibles.
- $W1 = np.random.rand(2,2) - 0,5$ : Define los pesos de entrada y la capa oculta. La forma (2,2) se debe a que hay dos neuronas de entrada y dos neuronas en la capa oculta. Se genera un valor aleatorio entre 0 y 1, al cual se le resta 0,5 para obtener valores entre -0,5 y 0,5, evitando saturar la función sigmoide.
- $b1 = np.random.rand(1,2) - 0,5$ : Define el sesgo para la capa oculta, con una fila y dos columnas. Al igual que en los pesos, se generan valores entre -0,5 y 0,5.
- $W2 = np.random.rand(2,1) - 0,5$ : Define los pesos entre la capa oculta y la capa de salida. Hay 2 neuronas ocultas conectadas a una neurona de salida.
  - $b2 = np.random.rand(1,1) - 0,5$ : Define el sesgo de la neurona de salida, también inicializando con un valor aleatorio pequeño.

Con esta inicialización, la red está lista para comenzar a realizar la propagación hacia adelante, calcular el error, y luego ajustar los pesos mediante retropropagación.

### 2.9.5 Parámetros de entrenamiento

En estas líneas de código se establecen los parámetros iniciales para configurar como se entrenará la red neuronal. Tal como se muestra en la figura 2-17, estos valores son clave para controlar la forma en que el modelo aprende.

```
22 # Parametros de configuracion del entrenamiento
23 learning_rate = 0.1
24 epochs = 6000
25 losses = []
26
```

Fuente: Elaboración propia

Figura 2-17 Parámetros de entrenamiento.

- *learning\_rate = 0,1*: Define la tasa de aprendizaje, qué controla que tan grande es el ajuste que se le hace a los pesos en cada ciclo. Un valor demasiado pequeño (como 0,001) haría que el aprendizaje sea muy lento, y uno muy grande (como 1.0) puede hacer que el modelo se vuelva inestable, en este caso se escoge un valor moderado (0,1), adecuado para una red simple.
- *epochs = 6000*: Define el número de épocas, es decir, cuantas veces se repetirá el entrenamiento recorriendo todos los datos. Se elige 6000 por que es suficiente para que la red neuronal aprenda correctamente el comportamiento de una compuerta lógica AND, este proceso se repite muchas veces para que la red aprenda gradualmente.
- *losses = []*: Se crea una lista vacía que almacenará los valores del error (o pérdida) en cada época. Esto permitirá luego visualizar si la red está aprendiendo, viendo como el error disminuye a medida que aumenta el entrenamiento.

### 2.9.6 Ciclo de entrenamiento

El ciclo de entrenamiento incluye el proceso de propagación hacia adelante donde la red neuronal calcula sus salidas basándose en las entradas y los valores actuales de los pesos y sesgos. Como se muestra en la figura 2-18.

```

27 # ciclo de entrenamiento
28 for epoch in range(epochs):
29     # propagación hacia adelante
30     z1 = np.dot(X, W1) + b1
31     a1 = sigmoid(z1)
32     z2 = np.dot(a1, W2) + b2
33     y_pred = sigmoid(z2)
34

```

Fuente: Elaboración propia

Figura 2-18 Ciclo de entrenamiento.

- *for epoch in range(epochs)*: Es un bucle que repite el proceso de entrenamiento tantas veces como se definió en *epochs* (en este caso 6000). Cada repetición representa una época.
- $z1 = np.dot(X, W1) + b1$ : Calcula la entrada total de  $z1$  para cada neurona de la capa oculta.
  - $X$ : son las entradas del conjunto de entrenamiento.
  - $W1$ : son los pesos que conectan las entradas con las neuronas ocultas.
  - $b1$ : es el sesgo aplicado a las neuronas de las capas ocultas.
  - $np.dot(X, W1)$ : realiza la multiplicación matricial entre las entradas y los pesos.
  - Luego se suma  $b1$  para ajustar la salida de cada neurona.
- $a1 = sigmoid(z1)$ : aplica la función de activación sigmoide sobre  $z1$ , obteniendo  $a1$ , que representa la salida de la capa oculta.
- $z2 = np.dot(a1, W2) + b2$ : calcula la entrada total  $z2$  de la neurona de salida. Usa  $a1$  como entrada,  $W2$  como pesos hacia la salida y  $b2$  como sesgo de salida.
- $y\_pred = sigmoid(z2)$ : aplica nuevamente la sigmoide a  $z2$  para obtener la salida final de la red,  $y\_pred$  para cada una de las entradas.

Este proceso permite que, en cada época, la red produzca una predicción basada en sus pesos y sesgos actuales, los cuales serán ajustados en los siguientes pasos usando retropropagación.

### 2.9.7 Cálculo de la pérdida

En este punto del ciclo de entrenamiento se calcula el error de la red neuronal, es decir, qué tan lejos estuvo la predicción del valor esperado. Esto se hace en cada época para poder llevar un

seguimiento del aprendizaje del modelo. En la figura 2-19 se puede observar cómo se realiza este cálculo.

```
35 # Cálculo de la pérdida (Error Cuadrático Medio)
36 loss = np.mean((y_pred - y)**2)
37 losses.append(loss)
38
```

Fuente: Elaboración propia

Figura 2-19 Cálculo de la pérdida.

Primero se usa la expresión  $loss = np.mean((y\_pred - y) ** 2)$ . Lo que hace esta línea es restar las salidas predichas por la red ( $y\_pred$ ) con las salidas reales del conjunto de entrenamiento ( $y$ ). Esta diferencia representa el error individual para cada una de las combinaciones de entrada. Luego, cada uno de estos errores se eleva al cuadrado para evitar que los errores negativos se cancelen con los positivos. Después con la función  $np.mean$ , se calcula el promedio de todos estos errores al cuadrado, lo que nos entrega el error cuadrático medio (ECM), también conocido como Mean Squared Error (MSE).

Este valor obtenido se almacena en la variable  $loss$ , y representa el error total que cometió la red en esa época.

Luego, con la instrucción  $losses.append(loss)$ , se guarda ese valor en una lista llamada  $losses$ . Esta lista se va llenando a medida que avanza el entrenamiento, lo cual es muy útil porque al final permitirá graficar como va disminuyendo el error, lo que nos demuestra si realmente la red neuronal está aprendiendo correctamente o no, este proceso de cálculo de pérdida se repite en cada una de las épocas de entrenamiento.

### 2.9.8 Propagación hacia atrás

Esta sección del código implementa el algoritmo de retropropagación, una de las partes fundamentales del aprendizaje en redes neuronales artificiales, como se puede observar en la figura 2-20, la retropropagación permite ajustar los pesos y sesgos de la red a partir del error cometido, para que en la próxima época la red pueda hacerlo mejor.

```

39 # propagación hacia atrás.
40 error_output = y_pred - y
41 d_y_pred = sigmoid_deriv(y_pred)
42 d_z2 = error_output * d_y_pred
43 d_W2 = np.dot(a1.T, d_z2)
44 d_b2 = np.sum(d_z2, axis=0, keepdims=True)
45
46 error_hidden = np.dot(d_z2, W2.T)
47 d_a1 = sigmoid_deriv(a1)
48 d_z1 = error_hidden * d_a1
49 d_W1 = np.dot(X.T, d_z1)
50 d_b1 = np.sum(d_z1, axis=0, keepdims=True)
51

```

Fuente: Elaboración propia

Figura 2-20 Propagación hacia atrás.

Este proceso se compone de varias etapas:

- Retropropagación del error desde la capa de salida,  $error\_output = y\_pred - y$ : aquí se calcula el error de la capa de salida, restando lo que predijo la red ( $y\_pred$ ) menos el valor real ( $y$ ). Esto nos dice que tan mal estuvo la predicción en cada uno de los ejemplos de entrada, el prefijo  $d\_$  significa derivada de.
- $d\_y\_pred = sigmoid\_deriv(y\_pred)$ : Luego se calcula la deriva de la función de activación sigmoide aplicada a la salida ( $y\_pred$ ). Esta derivada es necesaria para poder usar la regla de la cadena, que nos permite calcular cómo cambia el error con respecto a las entradas de las neuronas.
- $d\_z2 = error\_output * d\_y\_pred$ : Multiplica el error por la derivada de la función de activación, obteniendo así el gradiente del error con respecto a la entrada de la neurona de salida.
- $d\_W2 = np.dot(a1.T, d\_z2)$ : Calcula cuánto deben ajustarse los pesos entre la capa oculta y la salida. Se usa la transpuesta de  $a1$  (que es la salida de la capa oculta) y se multiplica por el gradiente anterior.
- $d\_b2 = np.sum(d\_z2, axis = 0, keepdims = true)$ : Aquí se calcula el ajuste necesario en el sesgo de la capa de salida. Sumando los errores por cada neurona de salida. El parámetro  $keepdims = true$  mantiene la forma en dos dimensiones para que sea compatible con el resto de los cálculos,  $axis = 0$  realiza operaciones a lo largo de las filas (verticalmente).

- Retropropagación del error hacia la capa oculta,  $error\_hidden = np.dot(d\_z2, W2.T)$ : Este paso propaga el error hacia la capa oculta, utilizando la transpuesta de los pesos  $W2$  que conectan la capa oculta con la salida.
- $d\_a1 = sigmoid\_deriv(a1)$ : Calcula la derivada de la función de activación sigmoide aplicada a la capa oculta. Esto permite saber cómo cambia la salida de la capa oculta respecto a su entrada.
- $d\_z1 = error\_hidden * d\_a1$ : Obtiene el gradiente del error con respecto a la entrada de la capa oculta, multiplicando el error propagado con la derivada calculada.
- $d\_W1 = np.dot(X.T, d\_z1)$ : Ahora se calcula cuánto deben ajustarse los pesos que conectan la entrada con la capa oculta. Se usa la transpuesta de  $X$  (la matriz de entrada) multiplicada por el gradiente del error de la capa oculta.
- $d\_b1 = np.sum(d\_z1, axis = 0, keepdims = true)$ : Finalmente, se calcula el ajuste del sesgo de la capa oculta, sumando los errores por cada neurona oculta, manteniendo el formato adecuado.

Este bloque completo del código es responsable de calcular como debe ajustarse cada parámetro (pesos y sesgos) de la red neuronal para reducir el error. Es el corazón del aprendizaje automático dentro de una red.

### 2.9.9 Actualización de pesos y sesgos

En esta etapa del código se realiza la actualización de los parámetros de la red neuronal, es decir, se modifican los pesos y sesgos con el objetivo de reducir el error cometido durante la predicción. Como se muestra en la figura 2-21, este proceso es esencial, ya que representa el momento en que la red realmente “aprende” a partir de sus errores.

Este ajuste se lleva a cabo mediante el algoritmo de descenso por gradiente, el cual consiste en actualizar los parámetros entrenables de la red, los pesos y sesgos, en la dirección opuesta al gradiente de la función de pérdida. Si bien los pesos sinápticos determinan la intensidad de las conexiones entre neuronas, los sesgos también influyen en el comportamiento de la neurona,

permitiendo ajustar el umbral de activación. Por esta razón, ambos parámetros son optimizados durante el entrenamiento.

```

52 # Actualización de pesos y sesgos
53 W2 -= learning_rate * d_W2
54 b2 -= learning_rate * d_b2
55 W1 -= learning_rate * d_W1
56 b1 -= learning_rate * d_b1
57

```

Fuente: Elaboración propia

Figura 2-21 Actualización de pesos y sesgos.

Cada línea actualiza los parámetros correspondientes:

- $W2$  y  $b2$ : son los pesos y sesgos de la capa de salida.
- $W1$  y  $b1$ : corresponden a los pesos y sesgos de la capa oculta.
- $d_W2$ ,  $d_b2$ ,  $d_W1$  y  $d_b1$ : representan los gradientes calculados en la retropropagación, es decir, cuanto y en qué dirección deben ajustarse esos valores para mejorar.
- *learning\_rate* (tasa de aprendizaje): controla el tamaño del paso que da la red en cada actualización. Un valor alto hace que los cambios sean más grandes y rápidos, pero también más riesgosos; un valor muy pequeño hace que el aprendizaje sea más lento, pero más estable.

Este paso es clave por que aplica los ajustes necesarios a la red basados en la información obtenida del error, permitiendo que, con el tiempo, las predicciones sean cada vez más precisas. Es como si la red entendiera en que se equivocó y realizara pequeños cambios dirigidos a corregirse para la próxima vez.

### 2.9.10 Prueba del modelo entrenado

En esta penúltima parte del código se realiza la evaluación del modelo después de haber sido entrenado, como se muestra en la figura 2-22. El objetivo es comprobar si la red neuronal ha aprendido correctamente a representar el comportamiento de una compuerta lógica AND.

```

58 # Prueba del modelo entrenado
59 print("Pesos de la capa oculta (W1):\n", W1)
60 print("Sesgos de la capa oculta (b1):\n", b1)
61 print("Pesos de la capa de salida (W2):\n", W2)
62 print("Sesgo de la capa de salida (b2):\n", b2)
63
64 print("\nPredicciones después del entrenamiento:")
65 for i in range(len(X)):
66     prediction = sigmoid(np.dot(sigmoid(np.dot(X[i], W1) + b1), W2) + b2)
67     print(f"Entrada: {X[i]}, Predicción: {prediction.flatten()[0]:.4f}, Esperado: {y[i][0]}")
68

```

Fuente: Elaboración propia

Figura 2-22 Prueba del modelo entrenado.

Primero, se imprimen los valores finales de los pesos y sesgos, estos valores ( $W1, W2, b1, b2$ ) son el resultado de todas las actualizaciones realizadas durante las épocas de entrenamiento, representan el conocimiento final que adquirió la red. Son precisamente estos parámetros los que permiten a la red hacer predicciones.

Luego, se realiza una propagación hacia adelante para cada entrada de prueba del conjunto  $x$ , utilizando los pesos y sesgos aprendidos. Se evalúa cómo responde la red comparando sus predicciones con los valores esperados.

Las predicciones después del entrenamiento:

- Se toma cada combinación de entrada  $X[i]$ .
- Se realiza el cálculo hacia adelante (forward propagation) aplicando la función de activación y los productos matriciales.
- Finalmente, se imprime la entrada número  $i$ , la predicción correspondiente (redondeada con 4 decimales) y el valor esperado asociado, el resultado se imprime con formato claro: se muestra la entrada, la predicción hecha por la red y el valor esperado según la tabla de verdad de la compuerta AND.

Este paso permite verificar visualmente si la red ha aprendido correctamente. Por ejemplo, para la entrada  $[1, 1]$ , la predicción debería estar cercana a 1, y para las otras combinaciones debería estar cercana a 0.

En resumen, esta sección permite validar el conocimiento adquirido por la red neuronal y comprobar si ha logrado generalizar correctamente la función lógica AND.

### 2.9.11 Resultados obtenidos del entrenamiento

Como se muestra en la figura 2-23, luego del proceso de entrenamiento, la red neuronal ha ajustado sus parámetros internos y ha aprendido una configuración de pesos y sesgos que le permiten resolver correctamente la compuerta lógica AND.

```

Pesos de la capa oculta (W1):
[[-2.65781413  2.35866732]
 [-2.67663861  2.33806572]]
Sesgos de la capa oculta (b1):
[[ 3.64017137 -3.11861378]]
Pesos de la capa de salida (W2):
[[-6.05987318]
 [ 5.28122979]]
Sesgo de la capa de salida (b2):
[[-0.55361704]]

Predicciones después del entrenamiento:
Entrada: [0 0], Predicción: 0.0020, Esperado: 0
Entrada: [0 1], Predicción: 0.0362, Esperado: 0
Entrada: [1 0], Predicción: 0.0363, Esperado: 0
Entrada: [1 1], Predicción: 0.9470, Esperado: 1

```

Fuente: Elaboración propia

Figura 2-23 Resultados obtenidos en el entrenamiento.

Resultados obtenidos tras el proceso de entrenamiento, cabe destacar que estos valores son una de las muchas configuraciones posibles que una red neuronal puede encontrar para resolver este problema. Cada entrenamiento puede generar valores diferentes, pero siempre apuntando a minimizar el error y cumplir con la lógica esperada.

- Pesos de la capa oculta ( $W1$ ).

$$\begin{bmatrix} -2,6578 & 2,3586 \\ -2,6766 & 2,3380 \end{bmatrix}$$

- Sesgos de la capa oculta ( $b1$ ):

$$[3,6401 \quad -3,1186]$$

- Pesos de la capa de salida ( $W2$ ):

$$\begin{bmatrix} -6,0598 \\ 5,2812 \end{bmatrix}$$

- Sesgo de la capa de salida ( $b2$ ):

$$[-0,5536]$$

Además, la red realiza predicciones muy cercanas a los valores esperados:

Tabla 2-1 Resultado de las predicciones del modelo.

Entrada	Predicción	Valor esperado
[0,0]	0,0020	0
[0,1]	0,0362	0
[1,0]	0,0363	0
[1,1]	0,9470	1

Fuente: Creación propia

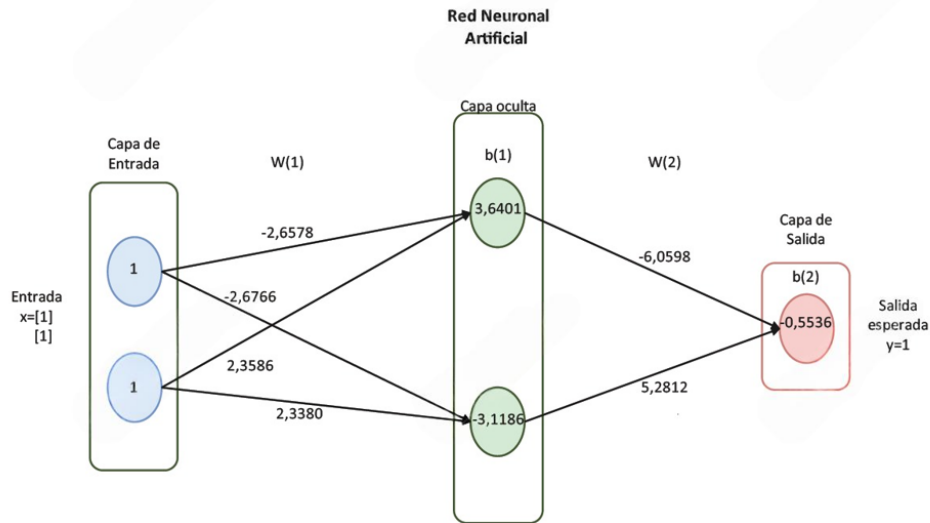
Esto demuestra que el modelo ha aprendido con éxito a clasificar correctamente los datos de entrada de una compuerta AND, entregando salidas cercanas a 0 para entradas que no cumplen la condición lógica, y una salida cercana a 1 para la entrada [1, 1].

### 2.9.12 Red neuronal funcional

Concluyendo esta etapa, se puede afirmar que la red neuronal entrenada ha logrado aprender correctamente el comportamiento lógico de una compuerta AND. Tal como se espera, la red solo entrega una salida cercana a 1 cuando ambas entradas son 1, y produce valores muy próximos a 0 en los demás casos.

Aunque las predicciones no son exactamente 0 o 1, los resultados obtenidos son lo suficientemente precisos. Esto se debe a que la función de activación utilizada en la salida es la sigmoide, la cual entrega valores continuos entre 0 y 1.

En la figura 2-24 se muestra la representación de cómo quedó finalmente la red neuronal con los valores aprendidos para los pesos y sesgos, lo que permite entender visualmente cómo procesa las entradas y genera las salidas esperadas. Esto demuestra que el proceso de entrenamiento fue exitoso y que la red quedó completamente funcional para resolver este problema lógico.



Fuente: Elaboración propia

Figura 2-24 Red neuronal funcional.

### 2.9.13 Gráfica de la función de pérdida

En esta última parte del código, se generará una gráfica que muestra cómo fue disminuyendo el error del modelo a lo largo de las épocas de entrenamiento. Como se observa en la figura 2-25.

```

69 # Graficar la función de pérdida
70 plt.plot(*args: range(epochs), losses)
71 plt.xlabel('Época')
72 plt.ylabel('Pérdida (MSE)')
73 plt.title('Evolución de la Pérdida durante el Entrenamiento')
74 plt.grid(True)
75 plt.show()

```

Fuente: Elaboración propia

Figura 2-25 Código de gráfica de la función de pérdida.

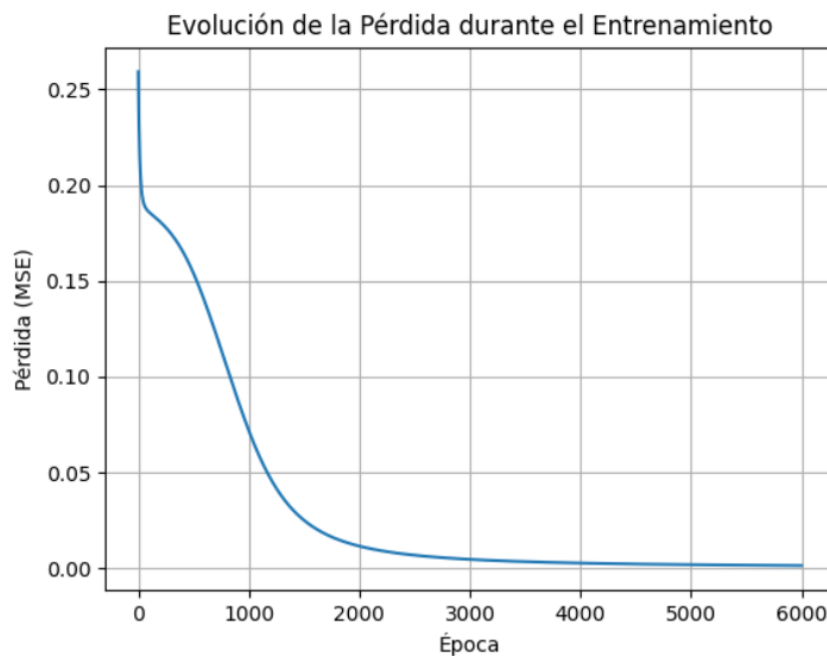
En el eje  $X$  se presentarán las épocas, es decir, cuántas veces se recorrió el conjunto completo de entrenamiento. En el eje  $Y$  se muestra el error cuadrático medio (MSE), que indica qué tan lejos están las predicciones del modelo con respecto a los valores reales.

Se utiliza la función `plt.plot()` para dibujar la curva del error a lo largo del tiempo, mientras que las funciones `xlabel()`, `ylabel()`, y `title()` añaden etiquetas y títulos gráficos. Finalmente, `plt.grid(True)` agrega una cuadrícula que facilita la lectura del gráfico y `plt.show()` lo despliega en la pantalla.

Este gráfico es clave para verificar que el error efectivamente disminuyó de forma progresiva, lo que indica que la red neuronal fue aprendiendo correctamente durante el entrenamiento.

#### 2.9.14 Evolución de la pérdida durante el entrenamiento

En la figura 2-26 se muestra cómo evolucionó la pérdida (error cuadrático medio, MSE) durante el entrenamiento de la red neuronal diseñada para aprender el comportamiento de una compuerta lógica AND.



Fuente: Elaboración propia

Figura 2-26 Evolución de la pérdida durante el entrenamiento.

Al comienzo del entrenamiento, la pérdida es considerablemente alta, lo que indica que la red comete errores en sus primeras predicciones. A medida que avanzan las épocas, la red ajusta progresivamente sus pesos y sesgos mediante el algoritmo de retropropagación del error, reduciendo gradualmente el valor de la función de pérdida. Se observa que alrededor de las 3.000 épocas la curva presenta un descenso pronunciado y posteriormente comienza a estabilizarse, alcanzando valores cercanos a cero hacia las 5.000 épocas. Este comportamiento indica que la red ha logrado capturar la relación lógica de la compuerta AND, la cual entrega un valor de salida igual a 1 únicamente cuando ambas entradas son iguales a 1.

No obstante, afirmar que la red “ha aprendido” en un punto específico del entrenamiento es, en cierta medida, un criterio subjetivo, ya que depende del objetivo práctico del modelo y del nivel de error aceptable definido para la aplicación. En este caso particular, al tratarse de una red neuronal de carácter académico compuesta por 2 neuronas en la capa de entrada, 2 en la capa oculta y 1 en la capa de salida, el propósito principal no fue la optimización del rendimiento, sino la comprensión y validación del proceso matemático subyacente. Esto incluye la propagación hacia adelante, el cálculo de la función de pérdida, la retropropagación del error y la actualización de pesos y sesgos.

Debido a la extensión que implica el desarrollo manual de estos cálculos, el entrenamiento fue posteriormente implementado en Python con el objetivo de dar continuidad al proceso y verificar que la lógica matemática efectivamente conduce a una convergencia estable. En este contexto, se puede apreciar que, una vez superadas aproximadamente las 3.000 épocas, la red deja de mostrar mejoras significativas en la pérdida, por lo que continuar el entrenamiento solo implica un mayor consumo de recursos computacionales sin un beneficio relevante en la precisión final del modelo.

En problemas reales, existen mecanismos formales para detener el entrenamiento de una red neuronal. Entre los más utilizados se encuentran el early stopping, que detiene el entrenamiento cuando la métrica de validación deja de mejorar durante un número determinado de épocas, y el criterio basado en umbral de error, donde el entrenamiento se finaliza una vez que la función de pérdida alcanza un valor previamente definido. Sin embargo, en este ejercicio dichos mecanismos no fueron implementados, ya que el enfoque es estrictamente pedagógico. El objetivo principal fue observar el proceso completo de aprendizaje y demostrar que, a partir de la formulación matemática, la red neuronal es capaz de converger hacia una solución estable. En este sentido, la definición de criterios de detención también puede considerarse subjetiva, dado que depende del propósito del modelo, el contexto de aplicación y las restricciones del sistema. Finalmente, se utilizaron 6.000 iteraciones con el fin de visualizar de manera clara la convergencia del modelo y la estabilización de la función de pérdida, aun cuando el problema planteado podría resolverse en un menor número de épocas.

## **2.10 REFLEXIÓN FINAL DEL SEGUNDO CAPÍTULO**

El ejercicio de entrenamiento de una red neuronal artificial ha representado una excelente oportunidad para comprender en profundidad los fundamentos del aprendizaje automático, particularmente en lo que respecta a las redes neuronales.

A lo largo del desarrollo, se demostró cómo un modelo sencillo puede aprender funciones lógicas como la compuerta AND de forma efectiva, utilizando primero un enfoque manual y luego una implementación automatizada en Python.

En el enfoque manual, se abordó el entrenamiento paso a paso, lo que permitió visualizar y entender claramente cómo funciona el algoritmo de retropropagación y actualización de pesos a nivel fundamental. Se observó con detalle cada etapa del proceso: desde la propagación hacia adelante, pasando por el cálculo del error, hasta la retropropagación del mismo. Esta aproximación facilitó la comprensión del mecanismo mediante el cual la red aprende corrigiendo sus pesos en función del error. Sin embargo, también se evidenciaron sus limitaciones, como la alta probabilidad a errores humanos en los cálculos y la dificultad de escalar este método a redes más grande o complejas. Además, se hizo evidente la magnitud del entrenamiento: lograr una red estable requiere miles de épocas, incluso en este ejemplo simple.

Por otro lado, al implementar el mismo ejercicio en Python, el proceso resultó mucho más eficiente y preciso. La automatización de los cálculos permitió observar cómo la red actualiza sus pesos y sesgos de forma progresiva para minimizar el error, logrando una convergencia estable hacia las 5.000 épocas. Esta implementación reafirma el poder de las herramientas computacionales modernas para desarrollar soluciones de aprendizaje automático de manera práctica.

En conjunto, ambos enfoques (manual y automatizado) permitieron consolidar el entendimiento de los conceptos esenciales de las redes neuronales artificiales. El primero para profundizar en la lógica interna del aprendizaje, y el segundo para aplicar dicho conocimiento en entornos reales donde se requieren resultados rápidos, repetibles y efectivos.

Finalmente, es importante contextualizar el entorno en el cual se realizaron las pruebas. El entrenamiento del modelo fue ejecutado en un computador portátil ASUS ROG Strix G15, equipado con un procesador AMD Ryzen 7 serie 4000, 16 GB de memoria RAM y una tarjeta gráfica dedicada NVIDIA GeForce RTX 3050 con memoria VRAM asociada. Este tipo de hardware permite acelerar

significativamente los procesos de entrenamiento, especialmente cuando se utilizan modelos más complejos o grandes volúmenes de datos, gracias a la capacidad de procesamiento paralelo de la GPU.

### **CAPÍTULO 3: PUESTA EN MARCHA DEL PROYECTO**

### **3 PUESTA EN MARCHA DEL PROYECTO**

En este capítulo se plantea el problema principal del proyecto: desarrollar un modelo basado en redes neuronales capaz de reconocer números naturales del 0 al 9. Además, se abordan las etapas de implementación y las pruebas prácticas del funcionamiento del programa.

#### **3.1 RECOLECCIÓN Y PREPARACIÓN DE DATOS**

Para entrenar el modelo de reconocimiento de números naturales manuscritos, se utilizará el conjunto de datos MNIST (Modified National Institute of Standards and Technology). Este dataset es ampliamente utilizado en problemas de clasificación de imágenes y aprendizaje profundo, y contiene un total de 70.000 imágenes de dígitos manuscritos del 0 al 9, serán distribuidas en 60.000 para entrenamiento y 10.000 para prueba.

Cada imagen tiene una resolución de 28x28 píxeles y está en escala de grises, lo que permite representarla como una matriz de 784 valores. Los valores de los píxeles varían entre 0 (negro absoluto) y 255 (blanco absoluto).

Antes de alimentar con estas imágenes a la red neuronal, es necesario aplicar ciertos pasos de preprocesamiento para adaptar adecuadamente los datos al modelo.

##### **3.1.1 Normalización**

Las imágenes del dataset MNIST presentan valores en píxel en el rango de 0 a 255. Sin embargo, las redes neuronales se entrenan de manera más eficientes cuando las entradas están normalizadas en un rango reducido, como  $[0, 1]$ . Esta transformación mejora la estabilidad numérica del modelo y acelera el proceso de convergencia durante el entrenamiento, evitando que los valores altos afecten desproporcionadamente el ajuste de los pesos.

En la práctica, esto se logra dividiendo cada valor de píxel por 255, transformado así el rango de intensidades de  $[0 - 255]$  a  $[0 - 1]$ .

### 3.1.2 Codificación One-Hot de las etiquetas

Las etiquetas originales del dataset son enteros del 0 al 9, que indican el número correspondiente en cada imagen. No obstante, las redes neuronales interpretan mejor las clases categóricas cuando se representan mediante codificación one-hot.

Esta técnica transforma cada número en un vector binario de 10 posiciones, donde solo una posición (la correspondiente a la clase) tiene un valor de 1, y el resto son ceros. Por ejemplo, la etiqueta 4 se representa como:

$$[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]$$

Este tipo de codificación evita que el modelo asuma relaciones numéricas erróneas entre clases (como suponer que el 9 es “mayor” que el 1), ya que en clasificación cada etiqueta representa una categoría distinta, no un valor ordinal. Además, esta representación es compatible con funciones como softmax en la capa de salida y la función de pérdida categorical crossentropy.

## 3.2 ARQUITECTURA DEL MODELO

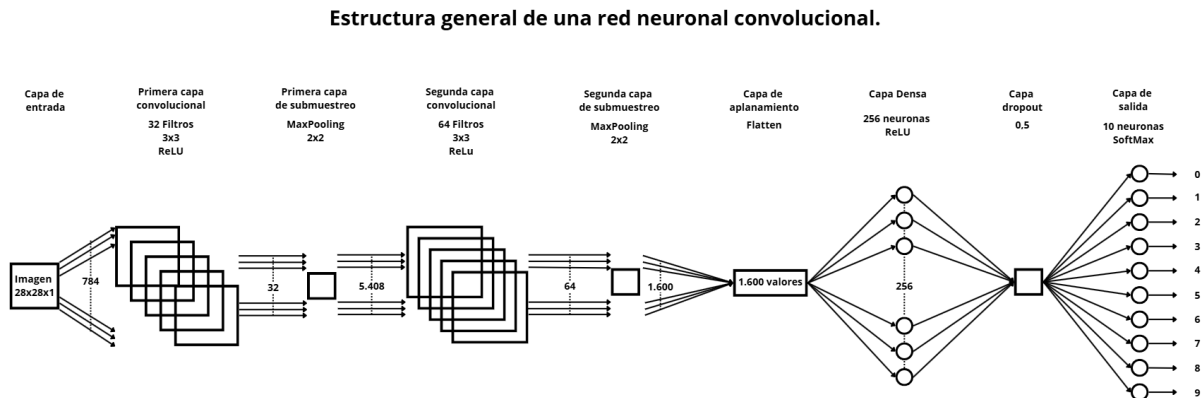
La arquitectura del modelo se basa en redes neuronales convolucionales (CNN), una técnica especialmente eficaz para procesar imágenes, como las del conjunto de datos MNIST, que contiene dígitos manuscritos. Las CNN son capaces de identificar patrones espaciales como bordes, curvas y formas, reutilizando pesos mediante filtros que se desplazan sobre la imagen (convoluciones), lo que reduce significativamente la cantidad de patrones. En las capas convolucionales se utilizan filtros con un stride de 1 y sin aplicación de padding, lo que corresponde a una convolución válida. Este tipo de red está diseñada para trabajar con datos que presentan una estructura espacial en forma de cuadrícula, como las imágenes.

El modelo implementado consta de varias capas: entrada, primera capa convolucional, primera capa de submuestreo, segunda capa convolucional, segunda capa de submuestreo, de aplanamiento, densa, dropout y capa de salida, como se muestra en la figura 3-1.

Las capas convolucionales extraen características de la imagen; las de submuestreo reducen la dimensionalidad y resaltan la información más relevante; y las capas densas procesan dichas características para realizar la clasificación. Esta estructura es ideal para tareas de reconocimiento

de patrones visuales, ya que aprovecha la naturaleza bidimensional de las imágenes, sin necesidad de aplanarlas desde el inicio.

A continuación, se detalla la estructura de una CNN, compuesta por capas especializadas que procesan la información en distintas etapas, desde lo más simple hasta lo más complejo.



Fuente: Elaboración propia.

Figura 3-1 Estructura general de una red neuronal convolucional basada en IBM, <https://www.ibm.com/es-es/think/topics/convolutionalneural-networks>.

### 3.2.1 Capa de entrada

Esta capa recibe directamente las imágenes en formato  $28 \times 28 \times 1$ , corresponde a imágenes de  $28 \times 28$  píxeles en escala de grises (1 canal). Aunque no realiza cálculos, es fundamental porque define la forma de los datos de entrada para toda la red.

### 3.2.2 Primera capa convolucional

Su objetivo es extraer características locales como líneas, bordes y formas simples, generando mapas de activación. Esto se logra mediante la aplicación de filtros (Kernels) utilizando la función *Conv2D*, que se desliza sobre la imagen.

Además, se emplea la función de activación ReLU, que introduce no linealidad, y permite a la red aprender patrones complejos, como los trazos curvos característicos del número 3. El stride, o

paso, representa el número de píxeles que se desplaza el filtro cada vez que avanza sobre la imagen. Habitualmente se usa un valor de 1, ya que así se obtiene un análisis más detallado de las características de la imagen.

Ejemplo:

- Se aplican 32 filtros de tamaño:  $3(K) \times 3(K)$ .
- Imagen de entrada:  $28(H) \times 28(W) \times 1$  (canal).
- Stride (paso): 1.
- Padding: 0 (convolución válida, sin relleno de bordes)

Cálculo de salida:

En el caso de una convolución sin padding y con stride igual a 1, la dimensión espacial de la salida de una capa convolucional se calcula como:

$$Salida\ conv2D = \left(\frac{H - K}{stride} + 1\right) \times \left(\frac{W - K}{stride} + 1\right)$$

Aplicando los valores del ejemplo:

$$Salida\ conv2D = (28 - 3 + 1) \times (28 - 3 + 1) = 26 \times 26$$

Cada filtro produce una salida de  $26 \times 26$ , y al aplicar 32 filtros, se obtienen 32 mapas de activación, resultando en una salida de dimensiones:

$$26 \times 26 \times 32$$

### 3.2.3 Primera capa de submuestreo

Estas capas aplican una operación de reducción espacial, comúnmente *MaxPooling2D*, sobre los mapas de activación generados previamente. El objetivo es reducir el tamaño de las representaciones (ancho y alto), conservar las características más significativas y descartar la información redundante.

El *MaxPooling* divide el mapa en regiones pequeñas (por ejemplo, de  $2 \times 2$ ) y selecciona el valor máximo de cada región. Este proceso reduce la cantidad de datos, acelerando el entrenamiento.

Ejemplo:

Aplicando *MaxPooling*  $2 \times 2$  a la salida anterior:

$$\text{Salida MaxPooling2D} = 13,13,32$$

Acá se reduce el tamaño tomando el valor máximo de cada bloque de  $2 \times 2$ , esto reduce el tamaño espacial a la mitad.

### 3.2.4 Segunda capa convolucional

Después de la primera etapa de extracción de características, se añade una segunda capa convolucional con el fin de detectar patrones más complejos y jerárquicos. Mientras que la primera capa identifica bordes o contornos simples, esta segunda capa aprende combinaciones de esas formas, como curvas, intersecciones o trazos característicos que representan mejor los dígitos escritos.

En esta capa se aplican 64 filtros de tamaño  $3 \times 3$ , cada uno recorriendo la salida anterior ( $13 \times 13 \times 32$ ) con un stride de 1, usando nuevamente la función de activación ReLU para mantener la no linealidad del modelo.

Ejemplo:

Aplicando *Conv2D* con 64 filtros sobre una entrada de  $13 \times 13 \times 32$ :

$$\text{Salida Conv2D} = \left( \left( \frac{13 - 3}{1} + 1 \right) \times \left( \frac{13 - 3}{1} + 1 \right) \right) = 11 \times 11$$

Por lo tanto, el resultado será una salida de tamaño  $11 \times 11 \times 64$ , donde cada uno de los 64 filtros genera un mapa de activación distinto.

### 3.2.5 Segunda capa de submuestreo

Del mismo modo que en la primera etapa, se aplica una capa de *Maxpooling2D* con ventana  $2 \times 2$  sobre la salida de la segunda convolución. El propósito sigue siendo reducir la dimensionalidad de los mapas de activación, conservar las características más representativas y disminuir la carga computacional.

El proceso selecciona el valor máximo de cada región de  $2 \times 2$  en cada mapa de activación resultando en una salida reducida de:

$$\left(\frac{11}{2}\right) = 5,5 \rightarrow 5$$

Por lo tanto, el tamaño de salida final de la capa será  $5 \times 5 \times 64$

Esto significa que ahora el modelo tiene 64 mapas de activación más compactos, pero que contienen información de alto nivel sobre la estructura de la imagen.

### 3.2.6 Capa de aplanamiento

La capa de aplanamiento (Flatten) convierte los mapas tridimensionales resultantes de la última etapa de submuestreo en un vector unidimensional, necesario para conectar la red convolucional con las capas densas, las cuales procesan las características extraídas y realizan la clasificación final mediante la capa de salida con 10 neuronas, una por cada dígito posible (0-9).

En este caso, la salida anterior tiene dimensiones  $5 \times 5 \times 64$ .

$$5 \times 5 \times 64 = 1.600$$

Por lo cual el proceso de aplanamiento genera un vector de 1.600 valores o unidades de activación, cada uno de estos valores se comporta como una neurona de entrada para la siguiente capa densa.

### 3.2.7 Capas densas

En esta etapa, cada neurona está conectada con todas las neuronas de la capa anterior, por lo que también se les denomina capas totalmente conectadas (Fully Connected Layers). Aunque las CNN extraen características locales mediante convoluciones y pooling, es necesario que una capa densa integre estas características para realizar una predicción final.

La salida del Flatten (vector unidimensional) se conecta con las neuronas densas, cada una de las cuales calcula su salida mediante la fórmula clásica de una red neuronal.

$$y = f(w^1 * x^1 + w^2 * x^2 + \dots + w^n * x^n + b)$$

Donde:

- $w^i$ : son los pesos.
- $x^i$ : es el vector unidimensional, es decir la entrada de estas neuronas.
- $b$ : es el sesgo.
- $f$ : es la función de activación.
- $y$ : es la salida.

Ejemplo:

Se definen 256 neuronas, cada una conectada con los 1.600 valores provenientes de la capa de aplanamiento.

### 3.2.8 Capa de salida

Esta es la capa final de la red, encargada de realizar la predicción. Está compuesta por 10 neuronas, una por cada clase (dígitos del 0 al 9). Cada neurona calcula un valor conocido como logit, en el contexto de las redes neuronales, el término logit se refiere al valor real que produce una neurona antes de aplicar la función de activación. Estos valores corresponden a combinaciones lineales de las entradas y los pesos, más el sesgo. Los logits no representan probabilidades por sí mismos, sino que contribuyen una medida de la evidencia que la red asigna a cada clase. Posteriormente, al aplicar la función de activación softmax, estos valores se transforman en una distribución de probabilidades interpretables.

Los logits están definidos por:

$$z_j = w_j^T x + b_j$$

Donde:

- $z_j$ : es el valor antes de aplicar la función de activación, llamado logit.
- $w_j$ : son los pesos de la neurona j.
- $x$ : es el vector de entrada.
- $b_j$ : es el sesgo de la neurona j.

Al tener 10 neuronas, se obtienen 10 valores reales  $z_0, z_1, \dots, z_9$ , estos 10 logits se normalizan con la función *softmax*, que convierte los valores en probabilidades.

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=0}^9 e^{z_k}}$$

- $z_j$ : representa los logits.
- $k$ : es el índice de suma, este recorre todas las clases posibles.
- $z_k$ : representa al logic antes de aplicar la función de activación.

La función softmax es una función de activación utilizada comúnmente en la capa de salida de redes neuronales para problemas de clasificación multiclase. Su objetivo es transformar los logits obtenidos por cada neurona en una distribución de probabilidades, donde cada valor se encuentra en el rango  $[0, 1]$  y la suma total de las probabilidades es igual a 1. De este modo, la salida del modelo puede interpretarse directamente como la probabilidad de pertenencia a cada clase.

Ejemplo:

Supongamos que los logits obtenidos en nuestras 10 neuronas son las siguientes:

$$z_k = [2.1, 0.3, 1.0, -0.5, 3.2, 0.1, 1.5, 0.9, -1.0, 0.0]$$

En la tabla 3-1 se calculan las exponenciales.

Tabla 3-1 Exponenciales de logits antes de aplicar la función de activación.

$k$	$z_k$	$e^{z_k}$
0	2.1	8.16
1	0.3	1.34
2	1.0	2.71
3	-0.5	0.6
4	3.2	24.53
5	0.1	1.1
6	1.5	4.48
7	0.9	2.45
8	-1.0	0,36
9	0.0	1

Fuente: Creación propia.

Ejemplo:

Suma total de exponenciales de  $e^{z_k}$ :

$$\sum_{k=0}^9 e^{z_k} \approx 46.73$$

Se calcula la probabilidad de la red para la clase 4:

$$\text{softmax}(z_j) = \frac{24.53}{46.73} = 0.524$$

Por lo tanto, la red predice que la imagen corresponde a la clase 4 con una probabilidad del 52,4%.

### **3.3 CONSIDERACIONES PREVIAS AL ENTRENAMIENTO.**

En esta etapa del desarrollo del modelo es esencial definir los componentes clave que permitirán optimizar el aprendizaje de la red neuronal convolucional. Estos elementos determinan cómo se ajustan los pesos, cómo se mide el rendimiento y qué técnicas se aplican para mejorar la generalización del modelo, a continuación, se analizará la función de pérdida, el optimizador, la métrica de evaluación, los hiperparámetros y el regularizador.

En conjunto, estos elementos establecen la base para que el entrenamiento de la CNN sea eficiente, preciso y capaz de adaptarse a datos de entrada no vistos anteriormente.

#### **3.3.1 Función de pérdida**

En problemas de clasificación multiclase, como el abordado en este proyecto, es fundamental contar con una función de pérdida capaz de cuantificar que tan diferente es la predicción del modelo respecto a la etiqueta real. Para ello se utilizará la función Categorical Crossentropy (entropía cruzada categórica), que mide la distancia entre dos distribuciones de probabilidad:

- Distribución predicha: salida de la capa softmax del modelo.
- Distribución real: vector one-hot que representa la clase correcta.

Un vector one-hot es aquel en el que todos los valores son cero excepto el correspondiente a la clase correcta, que es uno.

Por ejemplo, para representar la clase 3:

$$[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$$

Suponiendo que la red neuronal produce las siguientes probabilidades (salida de la función Softmax).

$$[0.02, 0.05, 0.01, 0.83, 0.03, 0.01, 0.01, 0.02, 0.01, 0.01]$$

La fórmula general de la entropía cruzada es:

$$Loss = - \sum_{i=0}^9 y_i \times \log(\hat{y}_i)$$

Donde:

$y_i$ : es el valor correspondiente en el vector one-hot para la clase  $i$  (0 o 1).

$\hat{y}_i$ : representa la probabilidad predicha por la red para la clase  $i$ .

Dado que en un vector one-hot la clase correcta tiene valor de 1, la formula se reduce a:

$$Loss = -\log(\hat{y}_{clase\ correcta})$$

Para la clase 3 ( $\hat{y}_3 = 0,83$ ):

$$Loss = -\log(\hat{y}_3) = \log(0.83)$$

$$Loss = -0.08092$$

Un valor de pérdida bajo indica que el modelo asignó una alta probabilidad en la clase correcta.

Si la red hubiera asignado una probabilidad de 0,05 a la clase correcta.

$$Loss = -\log(0,05) = 2,9957$$

En este caso la pérdida es mucho mayor, lo que refleja que la función penaliza fuertemente las predicciones erróneas y recompensa al modelo cuando asigna mayor probabilidad a la clase correcta. Estas recompensas y castigos son implícitos, es decir, los pesos se ajustan en mayor medida cuando la pérdida es alta (castigo) y en menor medida cuando la pérdida es baja (recompensa).

### 3.3.2 Optimizador

Para el entrenamiento del modelo se empleará el optimizador Adam (Adaptive Moment Estimation), un algoritmo de optimización ampliamente utilizado en aprendizaje profundo. Adam se basa en el descenso por gradiente y en el uso de tasas de aprendizaje adaptativas, lo que permite ajustar automáticamente la magnitud de las actualizaciones de los pesos y sesgos durante el entrenamiento. Este optimizador calcula dos estimaciones clave: el primer momento y el segundo

momento del gradiente. El primer momento puede entenderse como un promedio del gradiente a lo largo del tiempo, es decir, una especie de “dirección promedio” en la que los pesos deberían ajustarse. Esto ayuda a suavizar las actualizaciones y evita cambios bruscos en direcciones inconsistentes. Por otro lado, el segundo momento corresponde a un promedio de los valores al cuadrado del gradiente, lo que se relaciona con la “magnitud” o variabilidad de los cambios. En términos simples, permite medir qué tan grandes o inestables son los ajustes. Gracias a esto, Adam puede reducir el tamaño de las actualizaciones cuando los gradientes son muy grandes y aumentarlo cuando son pequeños, logrando así un entrenamiento más equilibrado.

La combinación de ambos momentos permite que el algoritmo adapte automáticamente la tasa de aprendizaje para cada parámetro, contribuyendo a un proceso de aprendizaje más estable y eficiente. Gracias a estas características, Adam logra una convergencia más rápida y confiable al minimizar la función de pérdida.

El optimizador se configura con una tasa de aprendizaje (learning rate) de 0,001. Este parámetro determina el grado de ajuste de los pesos en cada iteración; valores más altos pueden acelerar el aprendizaje, pero también aumentar el riesgo de una convergencia inestable o incorrecta. El algoritmo Adam se encarga de actualizar los pesos y los sesgos de la red neuronal convolucional con el objetivo de minimizar la función de pérdida, ajustando los parámetros en la dirección que permita reducir el error de manera eficiente. Adam destaca por requerir una configuración mínima y por su capacidad de converger rápidamente, lo que lo convierte en una opción ampliamente utilizada en tareas de aprendizaje profundo.

### 3.3.3 Métricas de evaluación

Durante el entrenamiento y la validación del modelo se utilizará la métrica accuracy (precisión) como principal indicador de rendimiento. Esta métrica mide el porcentaje de predicciones correctas realizadas por la red neuronal artificial.

Se define matemáticamente como:

$$Accuracy = \frac{\text{Número de predicciones correctas}}{\text{Número total de predicciones}}$$

Cabe mencionar que existen otras métricas de evaluación comúnmente utilizadas en problemas de clasificación, como *precision*, *recall* y *F1-score*, las cuales permiten un análisis más detallado del desempeño del modelo. No obstante, en este trabajo se utiliza principalmente la métrica accuracy, dado que el conjunto de datos empleado presenta clases balanceadas.

En el contexto de este trabajo, enfocado en la clasificación de dígitos del 0 al 9, la métrica accuracy se calcula comparando la clase predicha (neurona de salida con mayor probabilidad obtenida mediante la función softmax) con la etiqueta real (codificada en formato one-hot). Esta métrica resulta apropiada cuando las clases están balanceadas, es decir, cuando cada dígito aparece aproximadamente el mismo número de veces en el conjunto de datos.

#### 3.3.4 Hiperparámetros

Los hiperparámetros son configuraciones externas al modelo que define cómo se llevará a cabo el entrenamiento de la red neuronal. Estos parámetros son establecidos previamente por el desarrollador, influyendo de manera directa en el rendimiento y la capacidad de generalización del modelo.

A continuación, se analizan los utilizados en este trabajo:

#### 3.3.5 Épocas

Las épocas representan el número de veces que la red neuronal recorre por completo el conjunto de datos de entrenamiento. En el contexto de este trabajo, el modelo se entrena durante 15 épocas.

Aumentar el número de épocas puede mejorar la precisión del modelo, ya que permite que este aprenda más patrones de los datos. Sin embargo, entrenar durante demasiadas épocas puede provocar sobreajuste (*overfitting*), donde el modelo se adapta excesivamente a los datos de entrenamiento y pierde capacidad de generalización sobre datos nuevos.

La elección de un número relativamente bajo de épocas, en comparación con el ejercicio de la compuerta lógica AND, donde se utilizaron 6000 épocas, se debe a diferencias significativas en el

enfoque de entrenamiento. En este caso, se emplea el optimizador Adam, que ajusta de manera adaptativa la tasa de aprendizaje para cada parámetro, permitiendo una convergencia mucho más rápida y eficiente. Además, el modelo utilizado es considerablemente más complejo y se entrena sobre un conjunto de datos amplio como MNIST, lo que proporciona una mayor cantidad de información en cada época. También se incorpora una capa densa con 256 neuronas junto con una técnica de regularización Dropout de 0,5, lo que ayuda a prevenir el sobreajuste y mejora la capacidad de generalización del modelo.

En contraste, el ejercicio de la compuerta AND utilizaba una red muy simple, con pocas neuronas y una tasa de aprendizaje fija, lo que requería un mayor número de épocas para lograr la convergencia. Por tanto, en este contexto, 15 épocas son suficientes para alcanzar un buen rendimiento sin incurrir en sobreajuste.

### 3.3.6 Batch size

El batch size es un hiperparámetro que define cuántas muestras se procesan antes de actualizar los pesos de la red neuronal.

Un tamaño de lote grande permite entrenar más rápido, pero puede ocasionar que el modelo aprenda muy bien los ejemplos vistos y no se desempeñe correctamente con datos nuevos, fenómeno conocido como mala generalización. Esto significa que el modelo no logra aplicar lo aprendido en situaciones no vistas previamente.

Por el contrario, lotes más pequeños suelen producir un entrenamiento más estable y con mejor capacidad de generalización, aunque el proceso suele ser más lento.

En función del rendimiento del hardware, es común utilizar tamaños de lote como 32, 64, 128 o 256 ya que los procesadores modernos están optimizados para trabajar con potencias de 2 en operaciones paralelas.

### 3.3.7 División de datos para validación

Durante el entrenamiento, se ha reservado el 20% del conjunto de datos de entrenamiento con el fin de realizar una validación. Este proceso permite monitorear el rendimiento del modelo con datos no vistos, lo que ayuda a detectar sobre ajuste (overfitting), evaluar su capacidad de generalización, y tomar decisiones como detener el entrenamiento antes de tiempo (early stopping), ajustar el número de épocas o modificar otros parámetros para optimizar el desempeño.

La selección adecuada de estos hiperparámetros es fundamental para optimizar la eficiencia del entrenamiento y mejorar la predicción del modelo.

### 3.3.8 Regularización

Durante el entrenamiento de redes neuronales profundas, uno de los principales desafíos es el sobre ajuste (overfitting), el cual ocurre cuando el modelo aprende en exceso los datos de entrenamiento, perdiendo la capacidad de generalizar frente a datos nuevos.

Para mitigar este problema, se incorpora una técnica de regularización denominada dropout. Ésta consiste en desactivar aleatoriamente un porcentaje de neuronas durante cada época. Comúnmente, se emplea una tasa de 0,5, lo que significa que el 50% de las neuronas de esa capa se “apagarán” temporalmente.

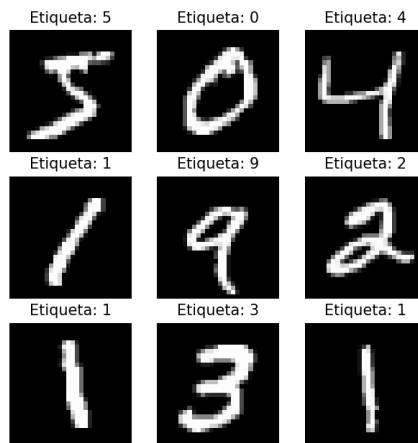
Este mecanismo obliga a la red a funcionar sin ciertas neuronas en cada iteración, evitando que dependa demasiado de alguna en particular. De esta forma el aprendizaje se distribuye de manera más equilibrada y robusta. Favoreciendo que el modelo no memorice los datos, sino que aprenda patrones generales útiles para nuevos escenarios.

## **3.4 IMPLEMENTACIÓN PRÁCTICA DE UNA RED NEURONAL CONVOLUCIONAL.**

En este apartado se desarrolla la implementación práctica de una red neuronal convolucional, aplicada al conjunto de datos MNIST. Este dataset constituye un estándar de referencia en el ámbito del aprendizaje profundo, ya que contiene 60.000 imágenes de entrenamiento y 10.000 imágenes

de prueba de dígitos manuscritos entre 0 y 9, cada una con un tamaño de 28x28 píxeles en escala de grises.

El conjunto MNIST como se ve en la figura 3-2, ha sido utilizado durante décadas como un punto de partida en la investigación de algoritmos de clasificación de imágenes, debido a su simplicidad y disponibilidad pública. En este trabajo, se emplea con el objetivo de demostrar de manera clara como una CNN puede aprender patrones visuales básicos (bordes, trazos y curvas) y utilizarlos para identificar con gran precisión los dígitos escritos a mano.



Fuente: Dataset MNIST

Figura 3-2 Ejemplo de conjunto MNIST.

En esta fase, se diseña la arquitectura de la red neuronal convolucional y se entrena utilizando el conjunto de datos de dígitos manuscritos. Durante el entrenamiento, la CNN ajusta sus parámetros internos (pesos y sesgos) mediante retropropagación, con el objetivo de minimizar la función de pérdida y aumentar la precisión de clasificación.

#### 3.4.1 Importación de librerías

Antes de construir y entrenar una red neuronal convolucional (CNN), es necesario importar las librerías y módulos que proporcionan las funciones y clases fundamentales para el trabajo con el dataset MNIST y el modelado en *Keras/TensorFlow*, en la figura 3-3 se muestra la importación de librerías. En este contexto, una librería es un conjunto de herramientas predefinidas que facilitan

el trabajo con datos y modelos, una función es un bloque de código que realiza una acción específica, por ejemplo, el `mnist.load_data()` para cargar los datos; una clase es un modelo para crear objetos, como `dense` o `secuencial`, que representan capas o modelos en la red neuronal.

```

1 import tensorflow as tf
2 from tensorflow.keras.datasets import mnist
3 from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
4 from tensorflow.keras.models import Sequential
5 from tensorflow.keras.utils import to_categorical

```

Fuente: Elaboración propia

Figura 3-3 Importación de librerías.

`import tensorflow as tf` importa la librería TensorFlow, que es el framework principal de aprendizaje profundo utilizado en este proyecto, provee todas las herramientas para crear, entrenar, evaluar y guardar redes neuronales.

`from tensorflow.keras.dataset import mnist` importa el dataset MNIST directamente desde `keras`, este dataset contiene imágenes de dígitos escritos a mano (0-9), de tamaño 28x28 píxeles en escala de grises, es la base de datos sobre la cual se entrena y evalúa la CNN.

`from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout` importa las capas necesarias para la creación de la CNN, `Conv2D` introduce la capa de convolución, detecta patrones como bordes y formas, `MaxPooling2D` reduce la dimensionalidad de las imágenes, conservando características importantes, `Flatten` transforma los mapas de características en un vector unidimensional, `Dense` capa totalmente conectada, usada para la clasificación final, `Dropout` técnica de regularización que desconecta neuronas aleatoriamente para evitar sobre ajuste, con estas capas se define la arquitectura de la CNN, cada capa cumple con un rol en el aprendizaje jerárquico de características, desde los más simples (bordes) hasta lo más abstractos (números completos).

`from tensorflow.keras.models import Sequential` importa el modelo Sequential, que permite aplicar capas de manera lineal y en orden, la CNN se construye como una secuencia de capas agregadas paso a paso, Sequential es la opción más eficiente para introducir arquitectura de redes, ideal para proyectos académicos.

`from tensorflow.keras.utils import to_categorical` importa la función `to_categorical`, que convierte etiquetas enteras (0-9) en vectores binarios (one-hot encoding), transforma las etiquetas de los dígitos en un formato compatible con la función de pérdida `categorical_crossentropy`, este paso es importante por que las redes neuronales no entienden números de clases directamente, sino distribuciones de probabilidad sobre las clases.

### 3.4.2 Carga y procesamiento de datos

En esta sección se prepara el dataset MNIST para que pueda ser utilizado por la red neuronal convolucional. Este proceso incluye la carga de los datos, su normalización, el ajuste de dimensiones y la codificación de las etiquetas, como se muestra en la figura 3-4.

```

7      # Carga y procesamiento de datos
8      (x_train, y_train), (x_test, y_test) = mnist.load_data()
9      x_train = x_train.astype("float32") / 255.0
10     x_test = x_test.astype("float32") / 255.0
11
12     x_train = x_train.reshape((-1, 28, 28, 1))
13     x_test = x_test.reshape((-1, 28, 28, 1))
14
15     y_train = to_categorical(y_train, 10)
16     y_test = to_categorical(y_test, 10)
17

```

Fuente: Elaboración propia

Figura 3-4 Carga y procesamiento de datos.

`(x_train, y_train), (x_test, y_test) = mnist.load_data()` carga el dataset MNIST en dos subconjuntos, `x_train, y_train` imágenes y etiquetas para entrenamiento, `x_test, y_test` imágenes y etiquetas para pruebas, permitiendo dividir el proceso de entrenamiento en dos fases (entrenamiento y evaluación) evitando que el modelo memorice, asegurando una evaluación objetiva.

`x_train = x_train.astype("float32") / 255.0`

`x_test = x_test.astype("float32") / 255.0` convierte los valores de píxeles (originalmente enteros entre 0 y 255) a números decimales entre 0 y 1, facilitando el entrenamiento de las redes neuronales, ya que éstas trabajan mejor con datos normalizados mejorando así la estabilidad numérica y acelerando la convergencia durante el entrenamiento.

```
x_train = x_train.reshape((-1, 28, 28, 1))
```

*x\_test* = *x\_test.reshape*((-1, 28, 28, 1)) cambia la forma de las imágenes redimensionando, los valores -1 define el número de imágenes (indefinido), 28 el alto, 28 el ancho y 1 canal (escala de grises) ya que las capas convolucionales Conv2D esperan datos con la estructura (alto, ancho canales).

```
y_train = to_categorical(y_train, 10)
```

*y\_test* = *to\_categorical*(*y\_test*, 10) convierte las etiquetas enteras (0-9) en vectores binarios de 10 posiciones (one-hot encoding), estas se transforman en un formato compatible con la función de pérdida *categorical\_crossentropy*, el one-hot encoding permite que la red neuronal aprenda a asignar probabilidades a cada clase.

### 3.4.3 Construcción del modelo CNN

En este apartado se define la arquitectura de la red neuronal convolucional (CNN), utilizando el modelo secuencial de Keras. Cada capa cumple con una función específica en el proceso de extracción de características y clasificación de imágenes, como se muestra a continuación en la figura 3-5.

```

19 model = Sequential([
20     tf.keras.Input(shape=(28, 28, 1)),
21     Conv2D(32, (3, 3), activation='relu'),
22     MaxPooling2D(pool_size=(2, 2)),
23     Conv2D(64, (3, 3), activation='relu'),
24     MaxPooling2D(pool_size=(2, 2)),
25     Flatten(),
26     Dense(256, activation='relu'),
27     Dropout(0.5),
28     Dense(10, activation='softmax')
29 ])

```

Fuente: Elaboración propia

Figura 3-5 Construcción del modelo CNN.

```
model = Sequential([
```

*tf.keras.Input*(*shape* = (28, 28, 1)), se inicializa un modelo *Sequential*, que permite aplicar capas de manera lineal, la entrada se define con imágenes de tamaño 28x28 y un canal (escala de grises) además establece la base de la CNN y especifica el tamaño de los datos de entrada.

*Conv2D(32, (3, 3), activation = 'relu')*, primera capa convolucional con 32 filtros de tamaño 3x3 y activación ReLU, las convoluciones reducen la complejidad al reutilizar pesos y la activación ReLU introduce no linealidad mejorando así la capacidad de aprendizaje.

*MaxPooling2D(pool\_size = (2, 2))*, capa de submuestreo máximo con ventana de 2x2, reduce la resolución espacial de las imágenes conservando las características más relevantes. Disminuye el número de parámetros y previene el sobreajuste, además de acelerar el entrenamiento.

*Conv2D(64, (3, 3), activation 'relu')*, segunda capa convolucional con 64 filtros de 3x3 y activación ReLU, permite detectar patrones más complejos (curvas, formas, texturas). Al aumentar el número de filtros, la red aprende representaciones jerárquicas más profundas de los dígitos.

*MaxPooling2D(pool\_size(2, 2))*, segunda capa de maxpooling con ventana de 2x2, reduce nuevamente el tamaño espacial, conservando las características aprendidas, ayuda a generalizar el aprendizaje y disminuye el riesgo de sobreajuste.

*Flatten()*, convierte la salida 2D de las capas convolucionales en un vector 1D, preparando los datos para ser procesados por las capas densas. El aplanamiento permite conectar las características extraídas con la etapa de clasificación.

*Dense (256, activation = 'relu')*, capa totalmente conectada con 256 neuronas y activación ReLU, combina las características extraídas actuando como puente entre la parte convolucional y la clasificación, integrando la información para la predicción.

*Dropout (0.5)*, técnica de regularización que apaga aleatoriamente el 50% de las neuronas durante el entrenamiento, previniendo el sobreajuste asegurando que la red no aprenda de neuronas específicas mejorando la capacidad de generalización del modelo.

*Dense(10, activation = 'softmax')*, capa de salida con 10 neuronas, y activación softmax, genera una distribución de probabilidad sobre las diez clases, la función softmax es ideal para problemas de clasificación multiclase ya que garantiza que la suma de probabilidades sea uno.

### 3.4.4 Compilación y entrenamiento

Una vez definida la arquitectura de la CNN, el siguiente paso es preparar al modelo para que aprenda y luego entrenarlo con los datos, como se muestra en la figura 3-6; esta parte corresponde al “cerebro” del aprendizaje automático donde la CNN pasa de ser una estructura vacía a un modelo entrenado capaz de reconocer y clasificar imágenes de dígitos escritos a mano.

```

31 # Compilación
32 model.compile(
33     optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
34     loss='categorical_crossentropy',
35     metrics=['accuracy']
36 )
37
38 # Entrenamiento
39 history = model.fit(
40     x_train, y_train,
41     epochs=15,
42     batch_size=32,
43     validation_split=0.2,
44     verbose=1
45 )
46

```

Fuente: Elaboración propia

Figura 3-6 Compilación y entrenamiento.

*model.compile()*: Se configuran las “reglas de aprendizaje” antes de entrenar, acá se define que algoritmo de optimización se usara, que función de pérdida y que métricas.

*optimizer = tf.keras.optimizers.Adam(learning\_rate = 0.001)*: Se utiliza el optimizador Adam que ajusta los pesos de la red para reducir el error además es muy eficiente en redes profundas por que se adapta dinámicamente a la tasa de aprendizaje de cada parámetro; *learning\_rate = 0.001* es un valor estándar que controla el tamaño de los pasos en la búsqueda de la solución.

*loss = 'categorical\_crossentropy'*: Esta función de pérdida mide que tan diferentes es la predicción del modelo respecto al valor real, como el problema es de clasificación multiclase, la función adecuada es la entropía cruzada categórica ya que penaliza fuertemente predicciones con

alta confianza en la clase equivocada, favoreciendo que el modelo aprenda a generar distribuciones de probabilidad precisa.

*metrics = ['accuracy']*: La métrica *accuracy* (exactitud) calcula el porcentaje de predicciones correctas, sirve para monitorear el desempeño del modelo durante el entrenamiento.

*history = model.fit()*: El resultado queda almacenado en *history*, el cual guarda la evolución de la pérdida y la exactitud, siendo útil para graficar el progreso.

*epochs = 15*: Define la cantidad de epochs, una época es un ciclo completo en el que el modelo ve todos los datos de entrenamiento una vez, aquí el modelo pasara 15 veces por todo el dataset, refinando sus parámetros en cada interacción.

*batch\_size = 32*: Los datos no se procesan todos de una vez, sino que en pequeños lotes (batches), cada batch tiene 32 imágenes lo cual resulta más eficiente en memoria y acelera el entrenamiento.

*validation\_split = 0.2*: Reserva el 20% de los datos para validación, estos datos no son usados para ajustar pesos, solo para medir rendimiento, ayudando a detectar sobreajuste (overfitting), permitiendo que el modelo no memorizar si no que a aprender.

*verbose = 1*: Controla el nivel de detalle mostrado durante el entrenamiento, muestra una barra de progreso por época, si fuera 0 no mostraría nada, si fuera 2 muestra solo una línea.

#### 3.4.5 Evaluación y guardado del modelo

Tras el proceso de entrenamiento, es fundamental evaluar el rendimiento del modelo en datos que nunca ha visto (conjunto de prueba), con el fin de medir su capacidad de generalización. Además, una vez validado su desempeño, se procede a guardar el modelo entrenado para utilizarlo sin la necesidad de entrenarlo nuevamente, lo cual ahorra tiempo y recursos computacionales.

```

47 # Evaluación
48 test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
49 print(f"Test accuracy: {test_accuracy:.4f}")
50 print(f"Test loss: {test_loss:.4f}")
51
52 # Guardar el modelo
53 model.save('CNN_para_prediccion_de_numeros.keras')
54

```

Fuente: Elaboración propia

Figura 3-7 Evaluación y guardado del modelo.

`test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose = 0)`: Evalúa el modelo utilizando los datos de prueba (`x_test`, `y_test`), permitiendo obtener métricas objetivas sobre que tan bien la red neuronal clasifica los dígitos en un conjunto diferente al entrenamiento.

`print(f"Test accuracy: {test_accuracy:.4f}")`

`print(f"test loss: {test_loss:.4f}")`: Imprime en consola la precisión (`accuracy`) y la pérdida (`loss`) del modelo sobre el conjunto de prueba, con 4 decimales de precisión. Ofreciendo una retroalimentación clara y cuantitativa del rendimiento.

`model.save('CNN_para_prediccion_de_numeros.keras')`: Guarda el modelo entrenado en un archivo con extensión `.keras`, permitiendo reutilizar el modelo sin necesidad de volver a entrenarlo, cargándolo directamente en futuras ejecuciones.

#### 3.4.6 Importación de librerías y carga del modelo entrenado

Una vez que el modelo ha sido entrenado y guardado, el siguiente paso es integrarlo en una aplicación interactiva que permite al usuario probarlo de manera práctica. Para ello, se utilizan librerías que permiten construir interfaces gráficas y manejar imágenes complementadas con la carga del modelo previamente entrenado en TensorFlow/Keras.

```

55 #Aplicación gráfica para predecir
56 import tkinter as tk
57 from PIL import Image, ImageDraw, ImageOps
58 import numpy as np
59
60 # Cargar el modelo entrenado
61 model = tf.keras.models.load_model("CNN_para_prediccion_de_numeros.keras")
62

```

Fuente: Elaboración propia

Figura 3-8 Importación de librerías y carga del modelo entrenado.

*import tkinter as tk* librería estándar de Python para crear interfaces gráficas, agregándole el alias de tk a esta librería.

*from PIL import Image, ImageDraw, ImageOps*: *Pil* es una librería, *Image* es el módulo principal, permite manipular imágenes (crear, abrir, guardar), *ImageDraw* permite dibujar sobre las imágenes (líneas, círculos etc), *ImageOps* permite invertir colores, recortar etc.

*Import numpy as np* esta librería facilita el manejo de arreglos numéricos, necesario para preparar la imagen antes de ser enviada al modelo, agregándole el alias de np a esta librería.

Estas librerías se utilizan para construir la ventana de dibujo, capturar los trazos realizados y convertirlo en un formato que el modelo pueda interpretar, permite crear aplicaciones completas que integran la teoría del aprendizaje automático con la práctica interactiva.

*model = tk.keras.models.load\_model("CNN\_para\_prediccion\_de\_numeros.keras")* carga el modelo previamente entrenado y guardado, la carga de modelos entrenados es clave en el ciclo de vida de proyectos de Machine Learning, ya que posibilita la implementación en entornos reales y la reutilización del trabajo previamente realizado.

### 3.4.7 Configuración de la ventana y superficie de trabajo

En esta sección se define la ventana principal de la aplicación y el área donde el usuario podrá dibujar los dígitos. Para ello se combina la librería *tkinter* para la interfaz gráfica (Canvas) y *Pil* para manejar la imagen en la que se guardará el trazo realizado en canvas, como se muestra en la figura 3-9.

```

63 # Configuración de ventana
64 ventana = tk.Tk()
65 ventana.title("Reconocimiento de Dígitos - MNIST")
66
67 anchura, altura = 280, 280
68 canvas = tk.Canvas(ventana, width=anchura, height=altura, bg='white')
69 canvas.grid(row=0, column=0, columnspan=4)
70
71 imagen_pil = Image.new("L", (anchura, altura), color=255)
72 dibujante = ImageDraw.Draw(imagen_pil)
73

```

Fuente: Elaboración propia

Figura 3-9 Configuración de la ventana de trabajo.

*ventana = tk.Tk()* crea una ventana principal usando *tkinter*, ésta es el contenedor de todos los elementos gráficos de la interfaz (canvas, botones, etiquetas).

*ventana.title("Reconocimiento de Dígitos – MNIST")* asigna un título a la venta principal.

*anchura, altura = 280, 280* define las dimensiones ancho y alto del área de dibujo en píxeles, aunque el modelo CNN trabaja con imágenes de 28x28, se usa un lienzo mayor (280x280) para facilitar la interacción y luego se reduce el tamaño al procesar la imagen.

*canvas = tk.Canvas(ventana, width = anchura, height = altura, bg = ' white')* crea un objeto *canvas* de *tkinter* dentro de la ventana, con las dimensiones definidas y un fondo blanco.

*canvas.grid(row = 0, column = 0, columnspan = 4)* organiza la posición del canvas dentro de la ventana utilizando el gestor de cuadrícula *grid*, *row = 0* coloca a canvas en la primera fila de la cuadrícula, *column = 0* coloca a canvas en la primera columna de esa fila y *columnspan = 4* hace que canvas ocupe 4 columnas de ancho, lo suficiente para poder dibujar números.

*imagen\_pil = Image.new("L", (anchura, altura), color = 255)* crea una nueva imagen con las dimensiones anteriormente vistas, se usa "L" (luminancia), la imagen tiene un solo canal con valores entre 0 y 255 (escala de grises).

*dibujante = ImageDraw.Draw(imagen\_pil)* crea un objeto de tipo *ImageDraw* asociado a la imagen generada, asegurando que cada trazo realizado por el usuario quede registrado como datos de imagen procesables, es el puente entre la interfaz gráfica y el procesamiento de imágenes, indispensable para convertir los trazos del usuario en datos numéricos.

### 3.4.8 Función para dibujar

En esta sección se define cómo el usuario puede dibujar con el mouse sobre el área interactiva (Canvas). Cada movimiento del mouse con el botón presionado genera un trazo que se refleja tanto en la interfaz gráfica como en la imagen interna PIL, como se muestra en la figura 3-10.

```

74 # Función para dibujar
75 def dibujar(event):
76     x, y = event.x, event.y
77     r = 8 # grosor del trazo
78     canvas.create_oval(x - r, y - r, x + r, y + r, fill="black")
79     dibujante.ellipse([x - r, y - r, x + r, y + r], fill=0)
80
81 canvas.bind("<B1-Motion>", dibujar)
82

```

Fuente: Elaboración propia

Figura 3-10 Función para dibujar.

*def dibujar(event)*: define la función dibujar, que recibe un objeto (*event*), este contiene la información del movimiento del mouse (posición X,Y) permitiendo capturar en tiempo real la interacción del usuario y traducirla en coordenadas gráficas que luego se convierten en trazos.

$x, y = event.x, event.y$  obtiene las coordenadas actuales del mouse dentro del canvas, la relación entre coordenadas y trazos es fundamental para digitalizar el dibujo en un formato procesable.

$r = 8$  define el radio del trazo a dibujar en píxeles.

*canvas.create\_oval(x - r, y - r, x + r, y + r, fill = "black")* dibuja un óvalo (círculo) en canvas en la posición (X, Y) con el grosor definido permitiendo que el usuario vea visualmente el trazo en la pantalla

*dibujante.ellipse([x - r, y - r, x + r, y + r], fill = 0)* dibuja un círculo equivalente en la imagen PIL asociada, con color negro (*fill = 0*) asegurando que el trazo quede guardado en la imagen digital interna, no solo en la interfase grafica.

*canvas.bind(<B1-Motion>, dibujar)* vincula la función dibujar al evento de mover el mouse con el botón izquierdo presionado (<B1-Motion>)

### 3.4.9 Limpieza de lienzo

Este fragmento del código se encarga de reiniciar el área de dibujo canvas y la imagen asociada en la memoria PIL, de modo que el usuario pueda ingresar un nuevo dígito sin interferencia del trazo anterior, además se restablece la etiqueta del resultado anterior para evitar confusiones, como se aprecia en la figura 3-11.

```

83 # Limpiar el lienzo
84 def limpiar():
85     canvas.delete("all")
86     dibujante.rectangle([0, 0, anchura, altura], fill=255)
87     etiqueta_resultado.config(text="")
88

```

Fuente: Elaboración propia.

Figura 3-11 Limpieza de lienzo.

*def limpiar ()*: define la función limpiar, que será ejecutada al presionar el botón correspondiente, encapsula la lógica necesaria para reiniciar tanto la interfaz visual como la imagen digital interna, separar ésta acción en una función mejora la organización del código y facilita la reutilización.

*canvas.delete("all")* elimina todos los elementos gráficos de Canvas, borrando el dibujo visible que realizó el usuario en la interfaz gráfica.

*dibujante.rectangle([0,0,anchura,altura], fill = 255)* dibuja un rectángulo blanco que cubre toda la superficie de la imagen PIL asociada, reiniciando la representación digital del lienzo rellenándola con un valor 255 (blanco), es fundamental que la imagen interna también se reinicie, ya que es la que se pasa al modelo de predicción. Si solo se limpiara el canvas, quedarían datos previos que afectarían la predicción.

*etiqueta\_resultado.config(text = "")* borra el texto de la etiqueta donde se muestra el resultado de la predicción asegurando que el lienzo también desaparezca el último dígito reconocido y su probabilidad.

### 3.4.10 Función para predecir el número dibujado

La función `predecir` toma el dibujo realizado por el usuario en el lienzo (canvas) lo procesa para que tenga las mismas características del dataset MNIST y luego lo pasa al modelo previamente entrenado. Finalmente, muestra en pantalla el dígito más probable junto con su porcentaje de certeza como se muestra en la figura 3-12.

```

89 # Predecir número dibujado
90 def predecir():
91     imagen_reducida = imagen_pil.resize((28, 28))
92     imagen_invertida = ImageOps.invert(imagen_reducida)
93     imagen_np = np.array(imagen_invertida) / 255.0
94     imagen_np = imagen_np.reshape(1, 28, 28, 1)
95
96     prediccion = model.predict(imagen_np, verbose=0)
97     clase = np.argmax(prediccion)
98     probabilidad = np.max(prediccion)
99
100     etiqueta_resultado.config(
101         text=f"Predicción: {clase} ({probabilidad*100:.2f}%)"
102     )
103

```

Fuente: Elaboración propia.

Figura 3-12 Función para predecir el numero dibujado.

`def predecir ()`: define la función `predecir`, que será llamada al presionar el botón “Predecir”, centraliza todo el proceso de transformación de la imagen y predicción con la CNN, al encapsular este flujo en una función mejora la modularidad y la claridad del programa.

`imagen_reducida = imagen_pil.resize((28,28))` redimensiona la imagen PIL de 280x280 píxeles a 28x28 adaptando la entrada al tamaño exacto.

`imagen_invertida = ImageOps.invert(imagen_reducida)` invierte los colores de la imagen (lo blanco pasa a negro y viceversa) MNIST representa los dígitos en blanco con fondo negro, esta transformación es necesaria para mantener los datos de entrenamiento y las entradas reales.

`imagen_np = np.array(imagen_invertida) / 255.0` convierte a la imagen en un arreglo de *NumPy*, normalizando los valores de píxel al rango [0, 1], preparando la imagen en el formato que

espera *TensorFlow/Keras*, la normalización acelera y estabiliza la inferencia del modelo tal como ocurrió en el entrenamiento.

*imagen\_np = imagen\_np.reshape(1, 28, 28, 1)* cambia la forma del arreglo para adaptarlo al formato de entrada de la CNN donde:

- 1=tamaño del batch (una sola imagen).
- 28, 28= dimensiones de la imagen.
- 1= canal (escala de grises).

Garantiza que la CNN reciba el dato en la estructura correcta, el modelo no acepta una simple matriz 28x28 sino un tensor con forma (batch, altura, anchura, canales).

*prediccion = model.predict(imagen\_np, verbose = 0)* pasa la imagen procesada al modelo, éste realiza una inferencia real usando la CNN entrenada, es el paso central donde la IA analiza el trazo y determina el dígito más probable.

*clase = np.argmax(prediccion)* obtiene el índice de la probabilidad más alta del vector, ese índice corresponde al dígito predicho (0-9), determina el número que el modelo “cree” que el usuario dibujó.

*probabilidad = np.max(prediccion)* extrae el valor máximo de probabilidad asociado a la clase predicha, muestra el nivel de certeza del modelo en su decisión.

*etiqueta\_resultado.config(*

*text = f"Predicción: {clase} ({probabilidad × 100: .2f}%)"*) actualiza la etiqueta de la interfaz para mostrar el dígito predicho y su porcentaje de confianza, entregando una retroalimentación inmediata al usuario permitiendo así poder visualizar el desempeño del modelo en tiempo real.

### 3.4.11 Botones

Esta parte del código conecta la interfaz con las funciones definidas para los botones, además de establecer el bucle principal que mantiene activa la ventana, como se muestra en la figura 3-13.

```

104 # Botones
105 boton_predecir = tk.Button(ventana, text="Predecir", command=predecir)
106 boton_predecir.grid(row=1, column=0)
107
108 boton_limpiar = tk.Button(ventana, text="Limpiar", command=limpiar)
109 boton_limpiar.grid(row=1, column=1)
110
111 etiqueta_resultado = tk.Label(ventana, text="", font=("Helvetica", 16))
112 etiqueta_resultado.grid(row=1, column=2, columnspan=2)
113
114 ventana.mainloop()

```

Fuente: Elaboración propia.

Figura 3-13 Limpieza de lienzo.

*boton\_predecir = tk.button(ventana, text = "Predecir", command = predecir)*

*boton\_predecir.grid(row = 1, column = 0)* crea un botón con el texto “Predecir” y lo ubica en la fila 1 columna 0 de la ventana, al presionar este botón se ejecuta la función *predecir*, la cual analiza el dibujo y muestra el número reconocido.

*boton\_limpiar = tk.Button (ventana, text = "Limpiar", command = limpiar)*

*boton\_limpiar.grid(row = 1, column = 1)* crea un botón con el texto “Limpiar” y lo ubica en la fila 1 columna 1, al presionar este botón se ejecuta la función *limpiar* que borra el lienzo y reinicia la etiqueta del resultado.

*etiqueta\_resultado = tk.Label(ventana, text = "", font = ("Helvetica", 16))*

*etiqueta\_resultado.grid(row = 1, column = 2, columnspan = 2)* crea una etiqueta vacía que luego se actualizará con el resultado de la predicción, se ubica en la fila 1, columna 2 ocupando dos columnas de ancho, sirve como área de salida visual, donde se mostrará el número detectado y su probabilidad. Helvética es el nombre de la tipografía (fuente) que se utiliza para mostrar texto en la interfaz gráfica, y el número 16 indica el tamaño de la fuente.

`ventana.mainloop()` inicial el bucle principal de la aplicación que mantiene la ventana activa y esperando interacciones (clics y movimiento del mouse), sin esta línea la ventana abriría y cerraría de inmediato, ya que el programa terminaría su ejecución.

### 3.5 INTERFAZ GRÁFICA

La implementación de una interfaz de usuario permite interactuar de manera directa con el modelo entrenado, facilitando su uso por personas sin conocimientos técnicos. En este proyecto se empleó Tkinter, la librería estándar de Python para el desarrollo de interfaces gráficas junto con Pil para el manejo de imágenes como se muestra en la figura 3-14.



Fuente: Elaboración propia.

Figura 3-14 Interfaz gráfica.

#### 3.5.1 Funciones principales de la interfaz gráfica

- Superficie de dibujo (Canvas): permite al usuario escribir un dígito con el mouse, simulando el entorno de prueba MNIST.
- Conversión y procesamiento de la imagen: el trazo se guarda como imagen, se convierte a escala de grises, se redimensiona a 28x28 píxeles y se normaliza para ser compatible con la entrada de la CNN.

- Botón de predicción: envía la imagen procesada al modelo, mostrando el dígito estimado y el nivel de confianza de la predicción.
- Botón de limpieza: reinicia el lienzo para permitir nuevas pruebas sin necesidad de reiniciar la aplicación.
- Etiqueta de resultado: muestra en pantalla la clase predicha (0-9), y su probabilidad.

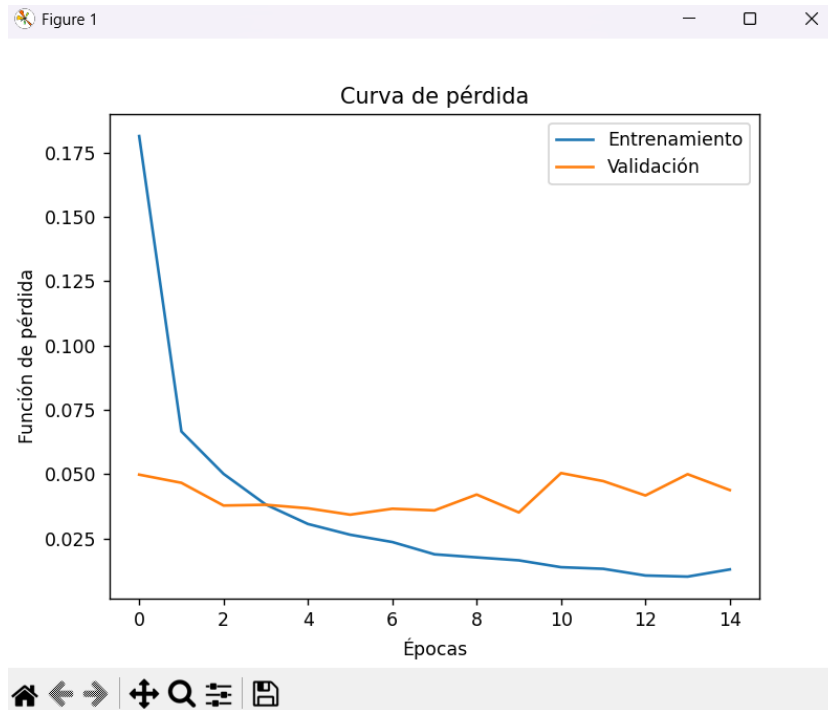
La interfaz gráfica no solo cumple con el rol práctico, si no también pedagógico y de validación permitiendo comprobar de forma interactiva que el modelo CNN funciona correctamente, además permite demostrar la conexión entre teoría (entrenamiento del modelo) y práctica (uso real por parte del usuario), haciendo al proyecto más accesible, acercando la inteligencia artificial a un contexto cotidiano y amigable.

### **3.6 EVALUACIÓN DEL MODELO**

El análisis del desempeño de un modelo CNN no debe limitarse únicamente a reportar una métrica global de precisión. Para comprender el detalle del comportamiento del modelo, es fundamental observar cómo aprende durante el entrenamiento, cómo se desempeña en la validación y qué errores específicos comete al momento de clasificar. En este caso, se utilizan tres herramientas complementarias: la curva de pérdida, la curva de precisión y la matriz de confusión. El estudio de estos tres elementos en conjunto ofrece una visión integral del rendimiento de la CNN entrenada sobre el conjunto de datos de MNIST, permitiendo evaluar tanto la eficacia como la capacidad de generalización.

#### **3.6.1 Curva de pérdida**

La curva de pérdida es una de las primeras métricas a considerar, ya que muestra directamente cómo evoluciona el error cometido por la red neuronal a lo largo de las épocas de entrenamiento. Este gráfico es esencial para identificar si el modelo está aprendiendo de manera eficiente, si converge hacia valores estables, en la figura 3-15 se observa esta gráfica.



Fuente: Elaboración propia.

Figura 3-15 Curva de pérdida.

El eje horizontal indica el número de épocas recorridas durante el entrenamiento y el eje vertical indica el valor de la función de pérdida (loss).

La línea azul (entrenamiento) desciende con rapidez en las primeras épocas, pasando de un error inicial relativamente alto ( $\sim 0.18$ ) a valores cercanos a 0.01, lo que evidencia que el modelo está aprendiendo progresivamente a reducir sus errores en los datos de entrenamiento.

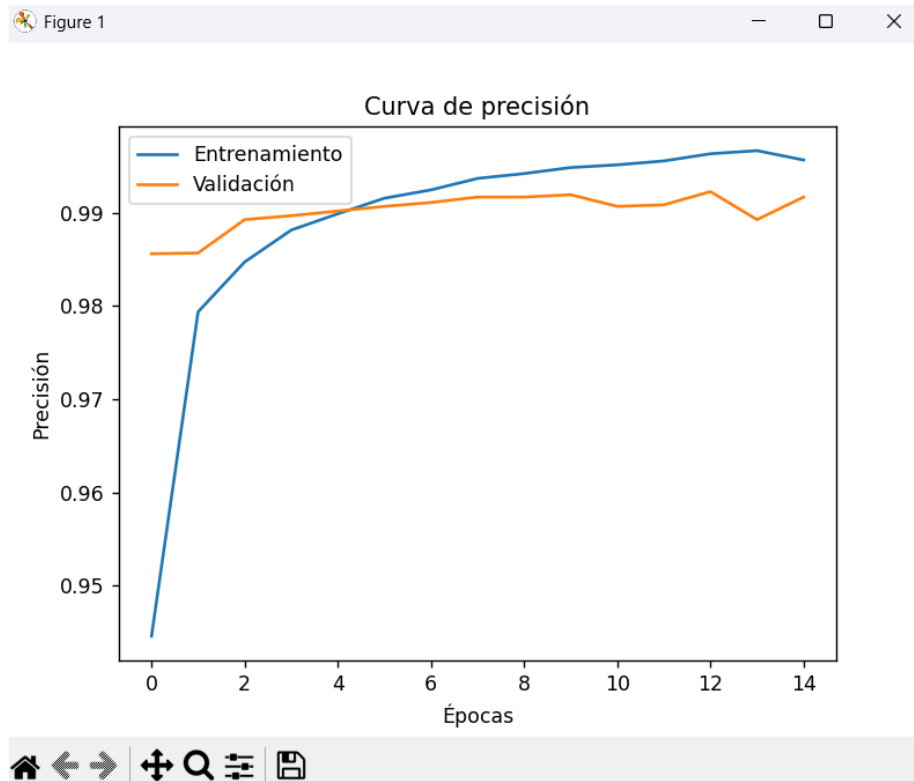
La línea naranja (validación), comienza en valores también bajos ( $\sim 0.05$ ), lo cual es un indicio de que el modelo está generalizando de forma adecuada y no se limita únicamente a memorizar los datos de entrenamiento.

Aunque la validación muestra pequeñas oscilaciones en las últimas épocas, se mantiene dentro de un rango controlado y cercano al entrenamiento.

La curva de pérdida confirma que la CNN logra un aprendizaje sólido y estable, con una disminución sostenida del error. Esto significa que el modelo no solo aprende correctamente de los datos de entrenamiento, sino que también mantiene un buen rendimiento en los datos de validación, garantizando su equilibrio adecuado entre precisión y generalización.

### 3.6.2 Curva de precisión (accuracy)

La curva de precisión complementa la información de la pérdida al mostrar directamente el porcentaje de aciertos obtenidos durante el entrenamiento y la validación. Este gráfico es crucial para evaluar la eficiencia real del modelo, ya que refleja en términos prácticos cuántas predicciones son correctas, en la figura 3-16 se observa la curva de precisión.



Fuente: Elaboración propia

Figura 3-16 Curva de precisión.

El eje horizontal indica que el número de épocas y el eje vertical indica el valor de la precisión (accuracy).

La línea azul (entrenamiento) inicia en un valor cercano al 94% y muestra un incremento muy rápido durante las primeras épocas, superando el 99% de la precisión. Esto indica que la CNN aprende de manera eficiente y logra clasificar correctamente la mayoría de los dígitos en el conjunto de entrenamiento.

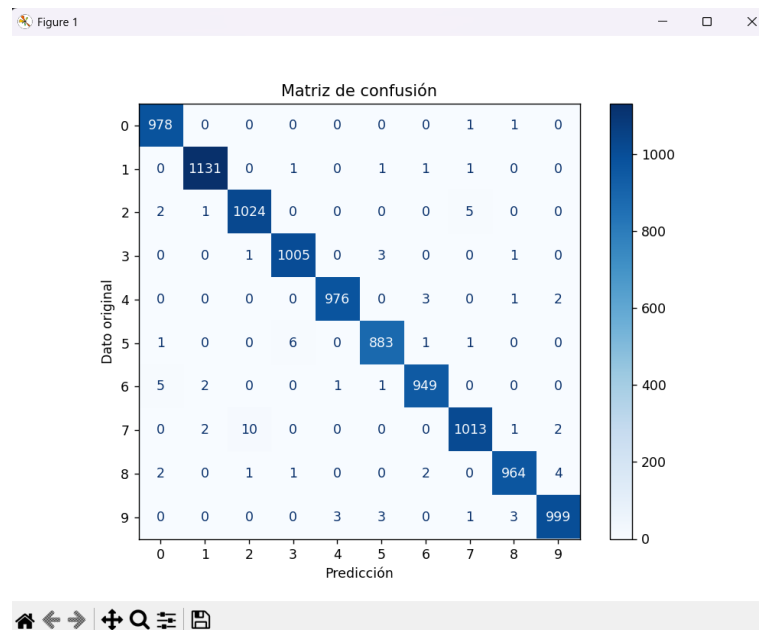
La línea naranja (validación) arranca en un valor ya alto (~98%) y se mantiene muy cercana a la línea azul durante todo el proceso, lo que refleja que el modelo no depende exclusivamente de los datos de entrenamiento y generaliza correctamente datos nuevos.

Se observan ligeras variaciones en la validación, lo cual es normal en este tipo de procesos, pero no afecta significativamente el rendimiento global.

La curva de precisión evidencia que el modelo alcanza un desempeño sobresaliente, con niveles de aciertos cercanos al 99% tanto en entrenamiento como en validación. La cercanía entre ambas curvas confirma que la CNN es capaz de reconocer patrones generales que permiten clasificar dígitos con alta exactitud en datos no vistos.

### 3.6.3 Matriz de confusión

La matriz de confusión ofrece un análisis más detallado, permitiendo visualizar qué tan bien se comporta el modelo en cada una de las clases específicas (los dígitos del 0 al 9). A diferencia de las métricas globales, esta herramienta muestra los aciertos y errores de forma desglosada, ayudando a detectar qué dígitos resultan más difíciles de reconocer para el modelo, en la figura 3-17 se expone la matriz de confusión.



Fuente: Elaboración propia.

Figura 3-17 Matriz de confusión.

El eje vertical indica el dato original, los dígitos reales (0-9) y el eje horizontal indica la predicción del modelo.

La diagonal principal concentra la mayoría de los valores en azul oscuro, lo que representa predicciones correctas, ejemplo: el dígito 1 fue clasificado correctamente en 1131 casos, el dígito 0 fue identificado correctamente en 978 casos y el dígito 9 se reconoció correctamente en 999 ocasiones.

Fuera de la diagonal se encuentran los errores, algunos dígitos 7 fueron confundidos con 2 en 10 casos, lo que indica cierta similitud en la escritura manuscrita de estas clases, el dígito 9 fue confundido con el 4 en 3 casos, mostrando otro punto débil en la diferenciación. En general, los errores son escasos en comparación con los aciertos.

La escala de color indica un azul oscuro al mayor número de aciertos y el azul claro lo contrario.

La matriz de confusión confirma que la CNN tiene un rendimiento muy equilibrado en todas las clases, con un alto número de aciertos en cada dígito y con errores residuales mínimos. Los pocos errores se concretan en pares de dígitos visualmente similares, lo cual es esperado en un problema de reconocimiento manuscrito.

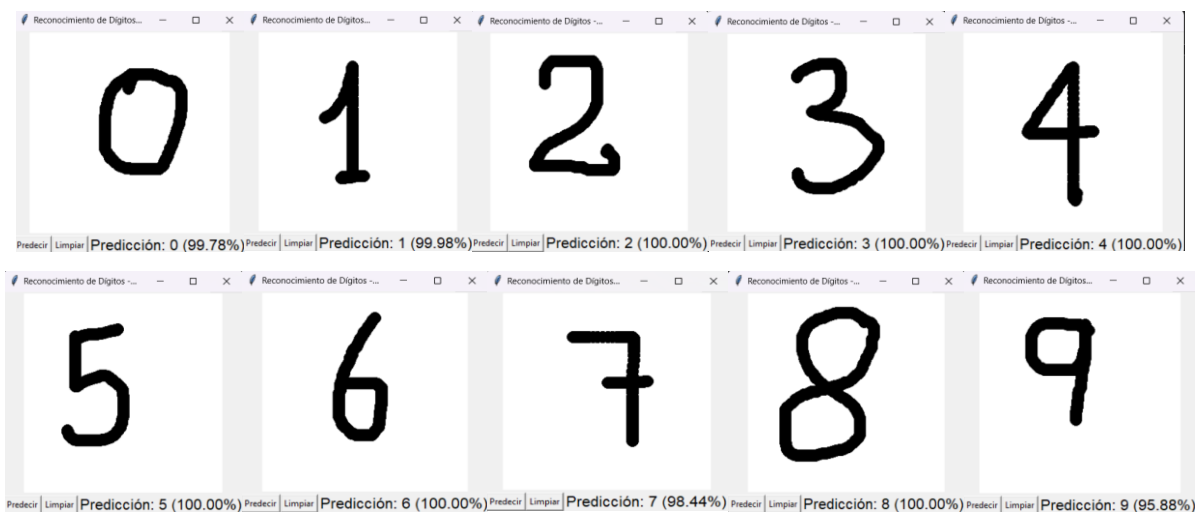
En conjunto, estos resultados permiten concluir que la CNN entrenada es altamente efectiva para la tarea de reconocimiento de dígitos manuscritos, alcanzando un equilibrio entre precisión, robustez, y capacidad de generalización.

### **3.7 VISUALIZACIÓN DE RESULTADOS**

En esta sección se presentan los resultados obtenidos a través de la interfaz gráfica de la red neuronal convolucional. El objetivo es visualizar cómo el modelo clasifica correcta e incorrectamente, los dígitos manuscritos (del 0 al 9) en tiempo real. Esta demostración complementa las métricas cuantitativas (precisión, recall, matriz de confusión) con ejemplos visuales, permitiendo confirmar de manera práctica la capacidad de generalización del modelo frente a entradas nuevas.

### 3.7.1 Predicciones correctas

Al ingresar dígitos del cero al nueve, la CNN fue capaz de identificar con alta precisión, mostrando un porcentaje de confianza asociado a cada predicción, como se observan a continuación en la figura 3-18. Los dígitos 0, 1, 2, 3, 4, 5, 6 y 8 fueron clasificados correctamente, con 100% de precisión (o valores muy cercanos, como 99.78% y 99.98%), lo que evidencia que los patrones de trazos de estos números son altamente distinguibles y la red los reconoce con total seguridad. El dígito 7 también fue clasificado correctamente, con una confianza de 98.44% reflejando igual un alto nivel de certeza en la predicción. Por otro lado, el dígito 9 obtuvo una predicción correcta, pero con una confianza algo menor (95.88%), lo que indica que, aunque el modelo identificó correctamente el número, reconoció cierta ambigüedad en sus trazos, ya que manuscritos de 9 pueden asemejarse a otros dígitos, como un 4 mal escrito.



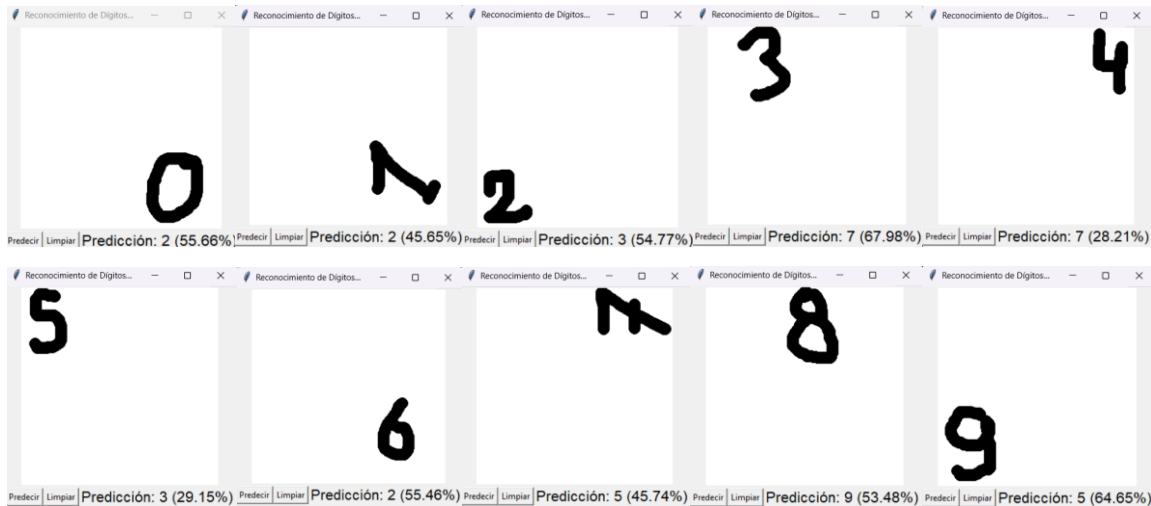
Fuente: Elaboración propia.

Figura 3-18 Predicciones correctas.

### 3.7.2 Predicciones incorrectas

La visualización de predicciones erróneas permite observar como el modelo CNN, a pesar de su alto rendimiento general, puede cometer confusiones significativas cuando se enfrenta a condiciones atípicas en la entrada, como en la figura 3-19. En este caso, al ubicar de manera intencional algunos dígitos en las esquinas del tablero de dibujo, algunos con cierta rotación se

generaron predicciones alejadas de la etiqueta real, alcanzando incluso porcentajes de confianza relativamente altos en la clase equivocada.



Fuente: Elaboración propia

Figura 3-19 Predicciones incorrectas.

Estos errores se deben principalmente a la posición y trazado de los dígitos, al estar desplazados hacia las esquinas, los números reflejan parte de la escritura central que el modelo aprendió durante el entrenamiento, el cual está basado en imágenes centradas (dataset MNIST). Esto provoca que la CNN interprete patrones incompletos o distorsionados, llevándola a confundir dígitos con otras formas.

Además del posicionamiento espacial, se podrían provocar errores en las predicción debido a la similitud entre dígitos (por ejemplo, un nueve mal escrito que parece 4), a las variaciones en el estilo de escritura propias de cada persona, a trazos incompletos, a diferencias de escala o tamaño que hacen perder detalles al reescalar y/o marcas adicionales al dibujo que confunden al modelo. Estos factores generan ambigüedad en la representación visual y explican por qué una CNN, pese a su alta precisión, pueden cometer errores en casos específicos.

### 3.7.3 Análisis general

Las predicciones correctas e incorrectas evidencian tanto las fortalezas como las limitaciones de la CNN entrenada. Los resultados correctos muestran un alto nivel de precisión y confianza,

confirmando que el modelo aprendió eficazmente los patrones característicos de los dígitos manuscritos y que su rendimiento se mantiene sólido en la mayoría de los casos.

Por otro lado, las predicciones incorrectas revelan que la red es sensible a factores externos como la posición o el trazo del dígito, ya que números desplazados o pocos centrados en el tablero de dibujo pueden alterar la representación interna de características, generando confusiones entre clases con similitudes estructurales.

En conjunto, estos resultados reflejan que la CNN es un modelo robusto y aplicable en contextos reales, siempre que las condiciones de entrada estén alineadas con el patrón de entrenamiento. Al mismo tiempo, los errores identificados ofrecen un punto de mejora.

## **CONCLUSIÓN.**

El presente trabajo de título abordó el diseño, implementación y evaluación de un sistema de reconocimiento de dígitos manuscritos mediante redes neuronales convolucionales (CNN), integrando fundamentos teóricos con una aplicación práctica a través de una interfaz gráfica interactiva.

En el capítulo 1 se estableció el marco teórico, introduciendo la inteligencia artificial y sus principales subcampos, como machine learning y deep learning, junto con la relevancia de la visión artificial como área clave para el procesamiento de datos visuales.

El capítulo 2 se centró en los fundamentos de las redes neuronales tradicionales, desarrollando un ejemplo práctico mediante la compuerta lógica AND. Este ejercicio permitió comprender en profundidad los procesos de propagación hacia adelante, cálculo de la función de pérdida y retropropagación, sentando las bases para el entendimiento de modelos más complejos.

En el capítulo 3 se desarrolló la implementación de una CNN aplicada al dataset MNIST, abarcando las etapas de procesamiento de datos, diseño de la arquitectura, entrenamiento y evaluación del modelo. La evaluación se realizó mediante métricas como la precisión (accuracy), curvas de pérdida y matriz de confusión, permitiendo analizar el desempeño del modelo. Los resultados obtenidos evidencian que la red es capaz de reconocer dígitos manuscritos con alta exactitud, demostrando una adecuada capacidad de generalización.

El trabajo de título demuestra que las CNN constituyen una herramienta eficiente para el reconocimiento de patrones complejos. La incorporación de una interfaz gráfica facilita su uso, ampliando su aplicabilidad incluso a usuarios sin conocimientos técnicos. Asimismo, este tipo de soluciones puede extenderse a otras aplicaciones de visión por computador, como el reconocimiento de caracteres en documentos o sistemas de entrada manuscrita. En cuanto a mejoras futuras, se propone ampliar y diversificar el dataset para incluir mayor variabilidad en la escritura, optimizar la arquitectura mediante modelos más avanzados como ResNet, y aplicar técnicas de aumento de datos. Además, el uso de aprendizaje por transferencia con modelos preentrenados y la implementación de sistemas de evaluación continua podrían contribuir a mejorar la robustez y el rendimiento del modelo.

En síntesis, el trabajo confirma que las redes neuronales convolucionales son una solución sólida y efectiva para problemas de reconocimiento de patrones visuales. Más allá de los resultados obtenidos, el valor del proyecto radica en el proceso metodológico seguido, que permitió comprender tanto los fundamentos teóricos como los desafíos prácticos del deep learning. Los resultados alcanzados validan la aplicabilidad de estas técnicas en escenarios reales y establecen una base para futuras mejoras e investigaciones.

**BIBLIOGRAFÍA.**

MORENO PARRA, R. A. (2018) Redes neuronales: Parte 1  
<https://ia902807.us.archive.org/33/items/RedesNeuronales1Moreno/Redes%20Neuronales.%20P arte%201%20-%20Moreno.pdf> [En línea][Consulta: Diciembre 2025]

HILERA, J. R., & MARTÍNEZ, V. J. (2010). Redes neuronales artificiales: Fundamentos modelos y aplicaciones.  
[https://www.researchgate.net/publication/44343683\\_Redес\\_neuronales\\_artificiales\\_fundamento\\_s\\_modelos\\_y\\_aplicaciones\\_Jose\\_Ramon\\_Hilera\\_Gonzalez\\_Victor\\_Jose\\_Martinez\\_Hernando](https://www.researchgate.net/publication/44343683_Redес_neuronales_artificiales_fundamento_s_modelos_y_aplicaciones_Jose_Ramon_Hilera_Gonzalez_Victor_Jose_Martinez_Hernando) [En línea][Consulta: Diciembre 2025]

Universidad Nacional Autónoma de México (UNAM). (s.f.) Fundamentos de redes neuronales artificiales. [https://conceptos.sociales.unam.mx/conceptos\\_final/598trabajo.pdf](https://conceptos.sociales.unam.mx/conceptos_final/598trabajo.pdf) [En línea][Consulta: Diciembre 2025]

Universidad Politécnica de Madrid (UPM). (s.f.) Programación de una red neuronal y ajuste de sus parámetros. [https://oa.upm.es/75938/1/TFG\\_JUAN\\_BERENGUER\\_TRIANA.pdf](https://oa.upm.es/75938/1/TFG_JUAN_BERENGUER_TRIANA.pdf) [En línea][Consulta: Diciembre 2025]

Arquitectura general de redes neuronales convolucionales descritas en IBM, 2024, <https://www.ibm.com/es-es/think/topics/convolutional-neural-networks> [En línea][Consulta: Diciembre 2025]

**ANEXO**

Gráficos de las funciones de activación: ReLU, Sigmoide y Tanh.

```
import numpy as np
import matplotlib.pyplot as plt

# Rango de valores de entrada
x = np.linspace(-10, 10, 400)

# Definición de funciones
def relu(x):
    return np.maximum(0, x)

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

def tanh(x):
    return np.tanh(x)

# Cálculo de salidas
y_relu = relu(x)
y_sigmoid = sigmoid(x)
y_tanh = tanh(x)

# Crear figura
plt.figure(figsize=(12, 4))

# ReLU
plt.subplot(1, 3, 1)
plt.plot(x, y_relu, color='orange', linewidth=2)
```

```
plt.axhline(0, color='black', linewidth=1.2)
plt.axvline(0, color='black', linewidth=1.2)
plt.title('ReLU')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True, alpha=0.3)
plt.text(-9, 8, r'$f(x) = \max(0, x)$', fontsize=12, color='orange')

# Sigmoide
plt.subplot(1, 3, 2)
plt.plot(x, y_sigmoid, color='blue', linewidth=2)
plt.axhline(0, color='black', linewidth=1.2)
plt.axvline(0, color='black', linewidth=1.2)
plt.title('Sigmoide')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True, alpha=0.3)
plt.text(-9, 0.9, r'$f(x) = \frac{1}{1 + e^{-x}}$', fontsize=12, color='blue')

# Tanh
plt.subplot(1, 3, 3)
plt.plot(x, y_tanh, color='green', linewidth=2)
plt.axhline(0, color='black', linewidth=1.2)
plt.axvline(0, color='black', linewidth=1.2)
plt.title('Tanh')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True, alpha=0.3)
plt.text(-9, 0.8, r'$f(x) = \tanh(x)$', fontsize=12, color='green')
```

```
plt.tight_layout()  
plt.show()
```

Gráficos de las funciones de pérdida: error cuadrático medio y entropía cruzada.

```
import numpy as np
import matplotlib.pyplot as plt

# --- Valores predichos ---
y_pred = np.linspace(0, 1, 200)

# --- Valor real (elegimos 0.5 para que se vea la U del MSE) ---
y_true = 0.5

# --- Funciones de pérdida ---
def mse(y, y_hat):
    return (y - y_hat) ** 2

def cross_entropy(y, y_hat):
    eps = 1e-9
    return -(y * np.log(y_hat + eps) + (1 - y) * np.log(1 - y_hat + eps))

# --- Cálculo de pérdidas ---
loss_mse = mse(y_true, y_pred)
loss_ce_y1 = cross_entropy(1, y_pred)
loss_ce_y0 = cross_entropy(0, y_pred)

# --- Gráficas ---
plt.figure(figsize=(12, 4))

# --- Error Cuadrático Medio ---
plt.subplot(1, 2, 1)
plt.plot(y_pred, loss_mse, color='blue', linewidth=2)
```

```

plt.plot(y_pred, loss_mse, color='blue', linewidth=1, label='Curva del error cuadrático medio')
plt.axhline(0, color='black', linewidth=1.2)
plt.axvline(y_true, color='black', linestyle='--', linewidth=1.2, label=f'y = {y_true}')
plt.title(r'Error Cuadrático Medio: ' '\n' r'$L = (y - \hat{y})^2$', fontsize=13)
plt.xlabel(r'$\hat{y}$ (Predicción)')
plt.ylabel('Pérdida')
plt.legend()
plt.grid(True, alpha=0.3)

# --- Entropía Cruzada ---
plt.subplot(1, 2, 2)
plt.plot(y_pred, loss_ce_y1, color='red', linewidth=2, label='y = 1')
plt.plot(y_pred, loss_ce_y0, color='green', linewidth=2, label='y = 0')
plt.axhline(0, color='black', linewidth=1.2)
plt.axvline(0, color='black', linewidth=1.2)
plt.title(r'Entropía Cruzada: ' '\n' r'$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$', fontsize=13)
plt.xlabel(r'$\hat{y}$ (Predicción)')
plt.ylabel('Pérdida')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

```

Compuerta lógica AND y gráfica de función de pérdida.

```
import numpy as np
import matplotlib.pyplot as plt

# Función de activación sigmoide y su derivada
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_deriv(a):
    return a * (1 - a)

# Datos de entrenamiento para la compuerta AND
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [0], [0], [1]])

# Pesos y sesgos iniciales
np.random.seed(42)
W1 = np.random.rand(2, 2) - 0.5
b1 = np.random.rand(1, 2) - 0.5
W2 = np.random.rand(2, 1) - 0.5
b2 = np.random.rand(1, 1) - 0.5

# Parámetros de entrenamiento
learning_rate = 0.1
epochs = 6000
losses = []

# ciclo de entrenamiento
for epoch in range(epochs):
    # propagación hacia adelante
```

```
z1 = np.dot(X, W1) + b1
a1 = sigmoid(z1)
z2 = np.dot(a1, W2) + b2
y_pred = sigmoid(z2)

# pérdida (MSE)
loss = np.mean((y_pred - y)**2)
losses.append(loss)

# propagación hacia atrás
error_output = y_pred - y
d_y_pred = sigmoid_deriv(y_pred)
d_z2 = error_output * d_y_pred
d_W2 = np.dot(a1.T, d_z2)
d_b2 = np.sum(d_z2, axis=0, keepdims=True)

error_hidden = np.dot(d_z2, W2.T)
d_a1 = sigmoid_deriv(a1)
d_z1 = error_hidden * d_a1
d_W1 = np.dot(X.T, d_z1)
d_b1 = np.sum(d_z1, axis=0, keepdims=True)

# actualización
W2 -= learning_rate * d_W2
b2 -= learning_rate * d_b2
W1 -= learning_rate * d_W1
b1 -= learning_rate * d_b1

# Prueba del modelo
print("\nPesos de la capa oculta (W1):\n", W1)
```

```
print("Sesgos de la capa oculta (b1):\n", b1)
print("Pesos de la capa de salida (W2):\n", W2)
print("Sesgo de la capa de salida (b2):\n", b2)

print("\nPredicciones después del entrenamiento:")
for i in range(len(X)):
    prediction = sigmoid(np.dot(sigmoid(np.dot(X[i], W1) + b1), W2) + b2)
    print(f"Entrada: {X[i]}, Predicción: {prediction.flatten()[0]:.4f}, Esperado: {y[i][0]}")

# gráfico
plt.plot(range(len(losses)), losses)
plt.xlabel('Época')
plt.ylabel('Pérdida (MSE)')
plt.title('Evolución de la Pérdida durante el Entrenamiento')
plt.grid(True)
plt.show()
```

Extracto de dataset MNIST.

```
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

# Cargar dataset MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Mostrar las primeras 9 imágenes
plt.figure(figsize=(6,6))
for i in range(9):
    plt.subplot(3,3,i+1) # 3 filas, 3 columnas
    plt.imshow(x_train[i], cmap="gray") # Escala de grises
    plt.title(f"Etiqueta: {y_train[i]}")
    plt.axis("off")
plt.show()
```

CNN números naturales del 0 al 9 e interfaz gráfica.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical

# Carga y preprocesamiento de datos
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

x_train = x_train.reshape((-1, 28, 28, 1))
x_test = x_test.reshape((-1, 28, 28, 1))

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Construcción del modelo CNN
model = Sequential([
    tf.keras.Input(shape=(28, 28, 1)),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

```
# Compilación
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Entrenamiento
history = model.fit(
    x_train, y_train,
    epochs=15,
    batch_size=32,
    validation_split=0.2,
    verbose=1
)

# Evaluación
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test accuracy: {test_accuracy:.4f}")
print(f"Test loss: {test_loss:.4f}")

# Guardar el modelo
model.save('CNN_para_prediccion_de_numeros.keras')

import pickle

# Guardar historial de entrenamiento
with open("history.pkl", "wb") as f:
    pickle.dump(history.history, f)
```

```
#Aplicación gráfica para predecir
import tkinter as tk
from PIL import Image, ImageDraw, ImageOps
import numpy as np

# Cargar el modelo entrenado
model = tf.keras.models.load_model("CNN_para_prediccion_de_numeros.keras")

# Configuración de ventana
ventana = tk.Tk()
ventana.title("Reconocimiento de Dígitos - MNIST")

anchura, altura = 280, 280
canvas = tk.Canvas(ventana, width=anchura, height=altura, bg='white')
canvas.grid(row=0, column=0, columnspan=4)

imagen_pil = Image.new("L", (anchura, altura), color=255)
dibujante = ImageDraw.Draw(imagen_pil)

# Función para dibujar
def dibujar(event):
    x, y = event.x, event.y
    r = 8 # grosor del trazo
    canvas.create_oval(x - r, y - r, x + r, y + r, fill="black")
    dibujante.ellipse([x - r, y - r, x + r, y + r], fill=0)

canvas.bind("<B1-Motion>", dibujar)

# Limpiar el lienzo
```

```

def limpiar():
    canvas.delete("all")
    dibujante.rectangle([0, 0, anchura, altura], fill=255)
    etiqueta_resultado.config(text="")

# Predecir número dibujado
def predecir():
    imagen_reducida = imagen_pil.resize((28, 28))
    imagen_invertida = ImageOps.invert(imagen_reducida)
    imagen_np = np.array(imagen_invertida) / 255.0
    imagen_np = imagen_np.reshape(1, 28, 28, 1)

    prediccion = model.predict(imagen_np, verbose=0)
    clase = np.argmax(prediccion)
    probabilidad = np.max(prediccion)

    etiqueta_resultado.config(
        text=f"Predicción: {clase} ({probabilidad*100:.2f}%"
    )

# Botones
boton_predecir = tk.Button(ventana, text="Predecir", command=predecir)
boton_predecir.grid(row=1, column=0)

boton_limpiar = tk.Button(ventana, text="Limpiar", command=limpiar)
boton_limpiar.grid(row=1, column=1)

etiqueta_resultado = tk.Label(ventana, text="", font=("Helvetica", 16))
etiqueta_resultado.grid(row=1, column=2, columnspan=2)

```

```
ventana.mainloop()
```

CNN ya entrenada.

```
import tensorflow as tf
import tkinter as tk
from PIL import Image, ImageDraw, ImageOps
import numpy as np

# Cargar el modelo entrenado
model = tf.keras.models.load_model("CNN_para_prediccion_de_numeros.keras")

# Configuración de ventana
ventana = tk.Tk()
ventana.title("Reconocimiento de Dígitos - MNIST")

anchura, altura = 280, 280
canvas = tk.Canvas(ventana, width=anchura, height=altura, bg='white')
canvas.grid(row=0, column=0, columnspan=4)

imagen_pil = Image.new("L", (anchura, altura), color=255)
dibujante = ImageDraw.Draw(imagen_pil)

# Función para dibujar
def dibujar(event):
    x, y = event.x, event.y
    r = 8 # grosor del trazo
    canvas.create_oval(x - r, y - r, x + r, y + r, fill="black")
    dibujante.ellipse([x - r, y - r, x + r, y + r], fill=0)

canvas.bind("<B1-Motion>", dibujar)
```

```

# Limpiar el lienzo
def limpiar():
    canvas.delete("all")
    dibujante.rectangle([0, 0, anchura, altura], fill=255)
    etiqueta_resultado.config(text="")

# Predecir número dibujado
def predecir():
    imagen_reducida = imagen_pil.resize((28, 28))
    imagen_invertida = ImageOps.invert(imagen_reducida)
    imagen_np = np.array(imagen_invertida) / 255.0
    imagen_np = imagen_np.reshape(1, 28, 28, 1)

    prediccion = model.predict(imagen_np, verbose=0)
    clase = np.argmax(prediccion)
    probabilidad = np.max(prediccion)

    etiqueta_resultado.config(
        text=f"Predicción: {clase} ({probabilidad*100:.2f}%"
    )

# Botones
boton_predecir = tk.Button(ventana, text="Predecir", command=predecir)
boton_predecir.grid(row=1, column=0)

boton_limpiar = tk.Button(ventana, text="Limpiar", command=limpiar)
boton_limpiar.grid(row=1, column=1)

etiqueta_resultado = tk.Label(ventana, text="", font=("Helvetica", 16))
etiqueta_resultado.grid(row=1, column=2, columnspan=2)

```

```
ventana.mainloop()
```

Curva de pérdida.

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

```
import pickle
```

```
# Cargar history
```

```
with open("history.pkl", "rb") as f:
```

```
    history = pickle.load(f)
```

```
# Gráfica de la curva de pérdida.
```

```
plt.plot(history["loss"], label="Entrenamiento")
```

```
plt.plot(history["val_loss"], label="Validación")
```

```
plt.title("Curva de pérdida")
```

```
plt.xlabel("Épocas")
```

```
plt.ylabel("Función de pérdida")
```

```
plt.legend()
```

```
plt.show()
```

Curva de precisión.

```
import matplotlib.pyplot as plt
import pickle

# Cargar del historial
with open("history.pkl", "rb") as f:
    history = pickle.load(f)

# Gráfico de la función de pérdida
plt.plot(history["accuracy"], label="Entrenamiento")
plt.plot(history["val_accuracy"], label="Validación")
plt.title("Curva de precisión")
plt.xlabel("Épocas")
plt.ylabel("Precisión")
plt.legend()
plt.show()
```

Matriz de confusión.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import numpy as np

# Cargar dataset
(_, _), (x_test, y_test) = mnist.load_data()
x_test = x_test.astype("float32") / 255.0
x_test = x_test.reshape((-1, 28, 28, 1))

# Cargar modelo entrenado
model = tf.keras.models.load_model("CNN_para_prediccion_de_numeros.keras")

# Predicciones
y_pred = model.predict(x_test, verbose=0)
y_pred_classes = np.argmax(y_pred, axis=1)

# Matriz de confusión
cm = confusion_matrix(y_test, y_pred_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.arange(10))

fig, ax = plt.subplots(figsize=(8, 6))
disp.plot(cmap=plt.cm.Blues, values_format="d", ax=ax, colorbar=True)

ax.set_xlabel("Predicción")
ax.set_ylabel("Dato original")
```

```
plt.title("Matriz de confusión")
```

```
plt.show()
```