

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

**INTRODUCCIÓN PRÁCTICA A LAS REDES NEURONALES CON
PYTHON**

Trabajo de Titulación para optar al Título de
Ingeniero de Ejecución en Control e
Instrumentación Industrial

Alumno:

Cristian Alexis Galvez Flores

Profesor 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: INTRODUCCIÓN PRÁCTICA A LAS REDES NEURONALES CON PYTHON

Nombre del candidato(a): Cristian Alexis Galvez Flores

Carrera / Grado:

Ingeniería Ejecución en Control e Instrumentación Industrial

Campus: 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: 15/07/2026

Firma: Guelis Montenegro Z.

Estudiante o Candidato(a):

Fecha: 15/07/2026

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: REDES NEURONALES, PYTHON, APRENDIZAJE AUTOMÁTICO, MACHINE LEARNING.

El presente trabajo de título está destinado a generar una guía didáctica para un lector principiante en el área de Machine Learning y con ciertos conocimientos en el lenguaje de programación Python. El trabajo de título entrega al lector que posea la inquietud de comprender cómo funcionan las Redes Neuronales, su finalidad y usos. Esto se hace mediante diferentes ejemplos prácticos, explicados detalladamente y de manera simple. Con el uso de Keras y Tensor Flow como herramientas básicas para la creación y entrenamiento de modelos. Adicionalmente se entregan soluciones simples a problemas que se presentan habitualmente en este tipo de sistemas, como lo son el Overfitting y el Underfitting.

ÍNDICE

INTRODUCCIÓN.....	1
1. ¿QUÉ ES UNA RED NEURONAL BIOLÓGICA?	3
1.1. TIPOS DE NEURONAS BIOLÓGICAS	3
1.1.1. Neuronas sensoriales	3
1.1.2. Neuronas motoras.....	3
1.1.3. Interneuronas	3
1.2. NEURONA ARTIFICIAL Y REDES NEURONALES	4
1.2.1. El Perceptrón	5
1.2.2. Entradas (X_n)	6
1.2.3. Pesos (W_n).....	6
1.2.4. Sesgo (b).....	6
1.2.5. Suma ponderada	6
1.2.6. Capas	7
1.3. FUNCIONES DE ACTIVACIÓN	7
1.3.1. Función Lineal.....	8
1.3.2. Función Escalón	8
1.3.3. Función Sigmoide o logística.....	8
1.3.4. Función Tangente Hiperbólica	8
1.3.5. Función ReLU	9
1.4. FUNCIONES DE PÉRDIDA (L)	9
1.4.1. Error Cuadrático Medio (MSE).....	10
1.4.2. Error absoluto medio (MAE)	10
1.4.3. Entropía Cruzada Binaria (BCE).....	11
1.4.4. Entropía Cruzada Categórica (CCE)	11
1.5. ALGORITMOS DE OPTIMIZACIÓN	12

1.5.1.	Descenso de gradiente y variantes	12
1.5.2.	ADAM	12
1.5.2.1	Tasa de aprendizaje (η)	13
1.6.	PROCESOS DE APRENDIZAJE.....	13
1.6.1.	Feedforward (FNN).....	13
1.6.2.	Backpropagation	13
1.7.	TIPOS DE APRENDIZAJE.....	14
1.7.1.	Aprendizaje supervisado	14
1.7.1.1	Regresión	14
1.7.1.2	Clasificación.....	14
1.7.2.	Aprendizaje no supervisado	15
1.7.3.	Aprendizaje por refuerzo (RL).....	15
1.8.	TIPOS DE REDES NEURONALES.....	15
1.8.1.	Redes Neuronales Artificiales (ANN).....	15
1.8.2.	Redes Neuronales Convolucionales (CNN).....	15
1.8.3.	Redes Neuronales Recurrentes (RNN)	16
1.8.3.1	Redes tipo LSTM y GRU.....	16
1.9.	MATEMÁTICA DE LA NEURONA	16
1.9.1.	Derivada del error respecto a la salida (predicción).....	17
1.9.2.	Derivada del error respecto a la suma ponderada (δ).....	17
1.9.3.	Derivada del error respecto al peso	17
1.9.4.	Derivada del error con respecto a la entrada (X_n)	18
1.9.5.	Derivada del error con respecto al sesgo (b).....	18
1.9.6.	Actualización de pesos (w)	18
1.9.7.	Actualización de sesgo (b)	19
1.9.8.	Ejemplo aplicación del descenso por gradiente	20
1.10.	APLICACIONES DE LAS REDES NEURONALES.....	23
1.11.	OBJETIVOS	23
1.11.1.	Objetivo General	23

- 1.11.2. Objetivos Específicos.....24
- 2. HERRAMIENTAS PARA EL DESARROLLO EN PYTHON 26
 - 2.1. PYTHON Y REDES NEURONALES.....26
 - 2.1.1. ¿Qué es Python?.....26
 - 2.1.2. Librerías de Python 27
 - 2.1.2.1 Numpy.....27
 - 2.1.2.2 Pandas.....27
 - 2.1.2.3 Matplotlib y Seaborn.....27
 - 2.1.2.4 Scikit- learn 27
 - 2.1.2.5 TensorFlow.....28
 - 2.1.2.6 Keras.....28
 - 2.1.2.7 PyTorch 28
 - 2.2. ENTORNOS DE DESARROLLO 28
 - 2.2.1. Visual Studio Code.....28
 - 2.2.2. Jupyter Notebook 29
 - 2.3. PROBLEMAS COMUNES EN EL ENTRENAMIENTO 29
 - 2.3.1. Overfitting (sobreajuste) 30
 - 2.3.1.1 Cómo prevenir el Overfitting 31
 - 2.3.2. Underfitting (subajuste).....31
 - 2.3.2.1 Cómo prevenir el Underfitting 32
 - 2.4. TÉCNICAS DE OPTIMIZACIÓN 32
 - 2.4.1. Early Stopping (Detención temprana).....33
 - 2.4.2. Dropout.....33
 - 2.4.3. Data Augmentation (aumento de datos).....33
 - 2.4.3.1 ¿Qué es el Data Augmentation? 33
 - 2.4.3.2 Tipos de Data Augmentation.....34
 - 2.5. IMPORTANCIA DEL HARDWARE.....35
 - 2.5.1. La CPU en redes neuronales 36
 - 2.5.2. La GPU en redes neuronales 36

2.5.3. Uso de RAM en redes neuronales.....	36
2.5.4. Uso de la VRAM en redes neuronales	36
3. IMPLEMENTACIÓN PRÁCTICA DE REDES NEURONALES	38
3.1. RED NEURONAL LINEAL SIMPLE	38
3.1.1. Modelación de una red neuronal lineal simple (Regresión).....	38
3.1.1.1 Carga de librerías para red neuronal simple.....	38
3.1.1.2 Configuración de parámetros	39
3.1.1.3 Compilación y entrenamiento del modelo	40
3.1.1.4 Gráfica de magnitud de pérdida v/s epochs	41
3.1.1.5 Generar predicción de datos.....	41
3.2. RED NEURONAL CONVOLUCIONAL (CNN)	43
3.2.1. Modelación de una red neuronal convolucional	44
3.2.2. Ejemplo de modelación de una red neuronal convolucional (CNN)	45
3.2.2.1 Carga de librerías	45
3.2.2.2 Configuración de parámetros y aumento de datos	46
3.2.2.3 Normalización de imágenes y carga de datos	47
3.2.2.4 Modelo base preentrenado (Transfer Learning).....	48
3.2.2.5 Construcción del modelo final	49
3.2.2.6 Compilación	50
3.2.2.7 Entrenamiento	50
3.2.2.8 Predicción.....	52
3.3. RED NEURONAL RECURRENTE (RNN).....	56
3.3.1. Modelación de una red neuronal recurrente.....	56
3.3.1.1 Estado oculto y Salida.....	57
3.3.2. Ejemplo de modelación de una red neuronal recurrente (RNN).....	59
3.3.2.1 Carga de librería.....	59
3.3.2.2 Generación de datos simulados.....	60
3.3.2.3 Adecuación de datos mediante el método de “ventanas deslizantes”.61	
3.3.2.4 Preparación de datos	62

3.3.2.5 Definición del tipo de modelo de red neuronal 63

3.3.2.6 Compilación y entrenamiento 63

3.3.2.7 Predicción.....64

3.3.2.8 Cálculo de error.....64

3.3.2.9 Visualización de predicción de carga 65

3.3.2.10 Gráfica de error 66

CONCLUSIÓN 67

BIBLIOGRAFÍA.....68

ANEXOS.....70

ÍNDICE DE FIGURAS

Figura 1-1 Diseño de una neurona biológica	4
Figura 1-2 Comparación entre neurona biológica y Perceptrón	5
Figura 1-3 Estructura de un Perceptrón.....	5
Figura 1-4 Organización de redes neuronales mediante capas.....	7
Figura 1-5 Funciones de activación	9
Figura 1-6 Ejemplo de perceptrón con dos entradas	20
Figura 2-1 Logo Python	26
Figura 2-2 Logo de Visual Studio Code.....	29
Figura 2-3 Logo Jupyter Notebook	29
Figura 2-4 Ejemplo conceptual de gráfica con y sin Overfitting	30
Figura 2-5 Ejemplo conceptual de gráficas con y sin Underfitting.....	32
Figura 2-6 Ejemplo de Data Augmentation en imágenes.....	34
Figura 2-7 Ejemplo de Data Augmentation de Texto.....	35
Figura 3-1 Modelación de red neuronal lineal : Carga de librerías.....	39
Figura 3-2 Configuración de parámetros de la red.....	39
Figura 3-3 Representación gráfica de Neurona lineal simple	40
Figura 3-4 Modelación de red neuronal básica: Modo de operación	40
Figura 3-5 Gráfica de magnitud de pérdida	41
Figura 3-6 Gráfico magnitud de pérdida v/s número de epochs	42
Figura 3-7 Comandos para visualizar variación de pesos y sesgos v/s epochs...	42
Figura 3-8 Descomposición de colores de una fotografía.....	44
Figura 3-9 Ejemplo de etapas en una red Convolutacional.....	45
Figura 3-10 Librerías que se deben descargar al inicio de modelación	46
Figura 3-11 Configuración de imágenes y aumento de datos	47
Figura 3-12 Normalización y carga de datos.....	48
Figura 3-13 Modelo base preentrenado (Transfer Learning)	49
Figura 3-14 Construcción del modelo final.....	50
Figura 3-15 Compilación	50
Figura 3-16 Entrenamiento del modelo.....	52
Figura 3-17 Preparación y carga de modelo.....	53
Figura 3-18 Procesamiento de imagen a analizar.....	54
Figura 3-19 Imágenes a testear.....	55

Figura 3-20 Resultados de la predicción 55

Figura 3-21 Arquitectura de una red Recurrente 56

Figura 3-22 Tipos de Redes Neuronales Recurrentes.....59

Figura 3-23 Importación de Librerías.....60

Figura 3-24 Generación de datos simulados 61

Figura 3-25 Uso de técnica “Ventana deslizante” 62

Figura 3-26 Generación de datos de entrenamiento 63

Figura 3-27 Definición de modelo LSTM.....63

Figura 3-28 Compilación y entrenamiento del modelo 64

Figura 3-29 Predicción del modelo 64

Figura 3-30 Cálculo de MSE del modelo 65

Figura 3-31 Líneas de comando para generar gráficas.....65

Figura 3-32 Gráfica de predicción de carga 66

Figura 3-33 Gráfica del error.....66

ÍNDICE DE FÓRMULAS

Ecuación 1-1 Función matemática básica de un perceptrón	6
Ecuación 1-2 Ecuación del error cuadrático medio	10
Ecuación 1-3 Ecuación del Error Absoluto Medio.....	10
Ecuación 1-4 Ecuación Entropía cruzada Binaria.....	11
Ecuación 1-5 Ecuación Entropía cruzada categórica	12
Ecuación 1-6 Derivada del error en función de la salida(predicción).....	17
Ecuación 1-7 Ecuación del error respecto a la suma ponderada	17
Ecuación 1-8 Cálculo de la derivada del error con respecto al peso	18
Ecuación 1-9 Derivada del error con respecto a la entrada.....	18
Ecuación 1-10 Derivada del error con respecto al sesgo	18
Ecuación 1-11 Actualización de pesos	19
Ecuación 1-12 Actualización de sesgo	19
Ecuación 3-1 Algoritmo de RNN	57
Ecuación 3-2 Cálculo de la salida y_t	58

ÍNDICE DE TABLAS

Tabla 1-1 Tabla de interacciones	23
Tabla 3-1 Comparación de datos	43

SIGLAS Y SIMBOLOGÍAS

A. SIGLAS:

ADAM	:	Adaptive Moment Estimation (Estimación Adaptativa de Momentos)
CNN	:	Convolutional Neural Networks (Redes neuronales Convolucionales)
CPU	:	Central Processing Unit (Unidad de Procesamiento Central)
DL	:	Deep Learning (Aprendizaje Profundo)
FNN	:	Feedforward Neural Network (Red neuronal de retroalimentación)
GRU	:	Gated Recurrent Unit (Unidad recurrente cerrada)
GPU	:	Graphics Processing Unit (Unidad de Procesamiento Gráfico)
IA	:	Inteligencia Artificial
LSTM	:	Long Short-term memory (Memoria a corto-largo plazo)
ML	:	Machine Learning (Aprendizaje de máquina)
MSE	:	Mean Squared Error (Error cuadrático medio)
RAM	:	Random Access Memory (Memoria de Acceso Aleatorio)
ReLU	:	Rectified Linear Unit (Unidad lineal rectificada)
RL	:	Reinforcement Learning (Aprendizaje por refuerzo)
RNA	:	Red Neuronal Artificial

RNN : Recurrent Neural Network (Redes Neuronales Recurrentes)

VRAM : Video Random Access Memory (Memoria de Acceso Aleatorio de Video)

B. SIMBOLOGÍA:

s : Segundo

m/s : metros por segundo

INTRODUCCIÓN

Las redes neuronales artificiales son un modelo básico basado en las redes neuronales biológicas, inspiradas en cómo el cerebro procesa la información, imitando así el funcionamiento de las neuronas cerebrales. De lo anterior, la importancia que poseen las redes neuronales artificiales en los sistemas computacionales actuales para el análisis de grandes cantidades de datos y la toma de decisiones de forma inteligente es de vital importancia, esto se debe a que pueden observar patrones no necesariamente tan claros, hacer generalizaciones, sacar conclusiones, generar aprendizaje autónomo y autocorrectivo, etc. Todas las características anteriores hacen de esta una herramienta básica la cual abarca la mayoría de los aspectos del diario vivir, ejemplos como: detección anticipada de enfermedades catastróficas, pronóstico del tiempo, análisis de datos, identificación de rostros, predicción de fallas en equipos o sistemas, predicción de movilidad de pasajeros en terminales de trenes y una infinidad de otras aplicaciones en diferentes áreas. En el desarrollo de este trabajo se intentará simplificar el entendimiento de lo que son las redes neuronales con Python, entregando ejemplos prácticos con las redes neuronales simples, redes neuronales convolucionales y redes neuronales recurrentes, cómo funcionan, sus características más importantes y cómo mejorar los sistemas con ciertas técnicas relativamente simples.

CAPÍTULO 1: ¿QUÉ ES UNA RED NEURONAL BIOLÓGICA?

1. ¿QUÉ ES UNA RED NEURONAL BIOLÓGICA?

El cerebro humano posee alrededor de cien millones de neuronas, las cuales mediante impulsos eléctricos y señales químicas se comunican entre ellas. Los diferentes tipos de neuronas manejan o ejercen distintas acciones, esto según su especialidad.

1.1. TIPOS DE NEURONAS BIOLÓGICAS

A continuación, se detallarán algunos de los principales tipos de neuronas biológicas y su función .

1.1.1. Neuronas sensoriales

Encargadas de recibir información desde los tejidos del cuerpo y los órganos sensoriales (ojos, piel, etc.) para enviarlos a la médula espinal. Ejemplo: Al tomar un cubo de hielo con la mano, esta transmite la información al cerebro de que está muy frío.

1.1.2. Neuronas motoras

La cual, mediante la obtención de información de otras neuronas, transmiten órdenes a los músculos, órganos y glándulas. Ejemplo: Cuando la persona toma el cubo de hielo, en un momento dado la mano lo soltará por el frío quemante.

1.1.3. Interneuronas

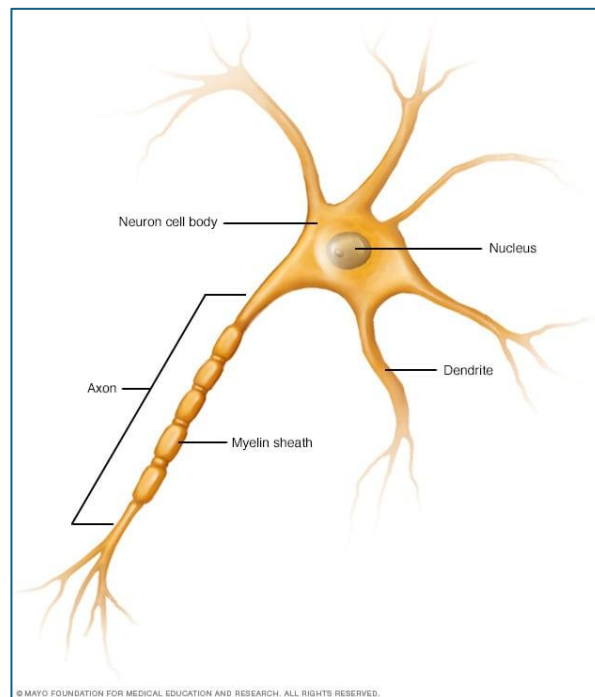
Dedicadas a conectar una neurona con otra, como el ejemplo del hielo, al estar muy frío reciben la información de las neuronas sensoriales, luego envían la información tanto a los músculos (neuronas motoras) como a la médula espinal para que las neuronas correspondientes entreguen la información de dolor al cerebro.

En la Figura 1-1 se muestra el diseño de una neurona biológica, la cual mediante las dendritas recibe la información proveniente de otra neurona o un órgano sensorial, luego de esto la señal es transformada y si estas señales son lo suficientemente grandes para generar un potencial de activación en el axón, éste generará un impulso que viajará

hasta la parte terminal del axón, posteriormente puede conectarse a otras neuronas vía sinapsis o músculos.

Se debe tener en consideración que la velocidad de transferencia de los mensajes de activación oscila entre 1 m/s y 100m/s, dependiendo del largo del axón.

La mielina en este caso trabaja como un recubrimiento aislante del axón, provocando así mayor rapidez en la transmisión de señales neuronales. La falta de mielina en las neuronas provoca enfermedades como esclerosis múltiple.

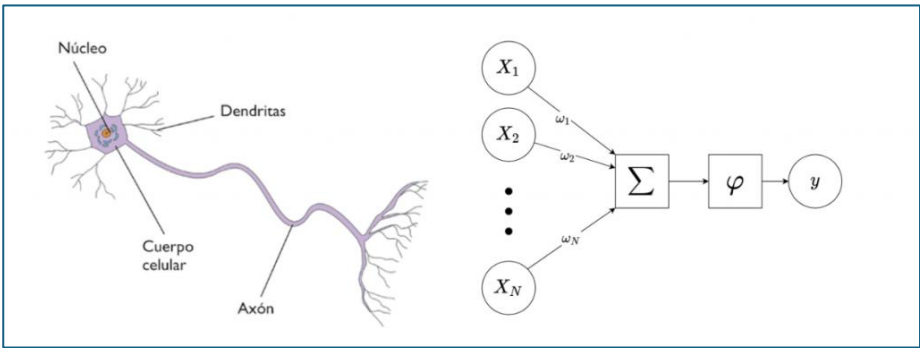


Fuente: <https://www.mayoclinic.org/es/nerve-cell-neuron/img-20007830>

Figura 1-1 Diseño de una neurona biológica

1.2. NEURONA ARTIFICIAL Y REDES NEURONALES

Una red neuronal artificial es un modelo computacional que intenta modelar el complejo sistema de toma de decisiones del cerebro humano, la red neuronal está compuesta, en su forma básica por el **perceptrón**, el cual es la base de las redes neuronales artificiales, estas redes pueden estar compuestas por unos pocos perceptrones, hasta miles, los cuales forman capas, las que en su conjunto son llamadas redes neuronales, como muestra la Figura 1-2. En donde: X_n son las entradas, W_n los pesos, Σ la suma ponderada, ϕ la función de activación y finalmente, la salida del sistema y .

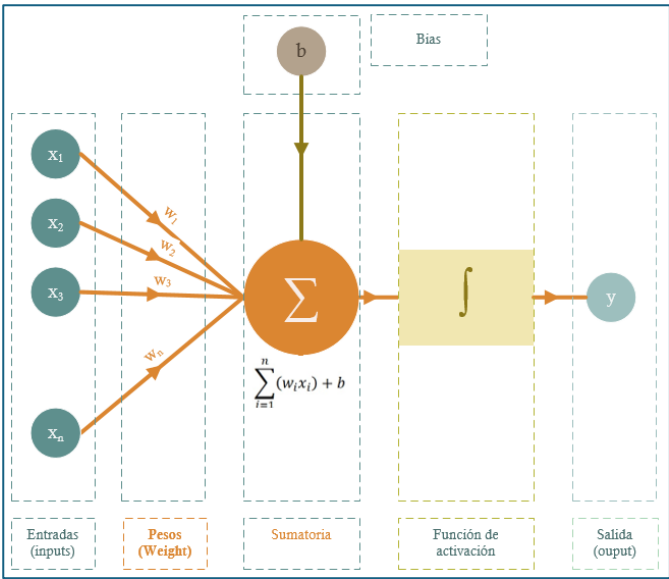


Fuente: blog.damavis.com

Figura 1-2 Comparación entre neurona biológica y Perceptrón

1.2.1. El Perceptrón

Creado en 1957 por el psicólogo estadounidense Frank Rosenblatt, basándose en los conceptos del primer modelo neuronal de McCulloch y Pitts (1943), es el primer modelo computacional de una neurona artificial capaz de aprender, ya que bastaba con enseñarle a reconocer figuras y de a poco iba aprendiendo a reconocerlas de una manera binaria. Mediante una función matemática la neurona artificial recepciona datos, los pondera, los evalúa y posteriormente una función no lineal, llamada función de activación presenta un resultado. En la figura 1-3 se muestra la estructura de un perceptrón y en la ecuación 1-1 se muestra la función matemática básica de un perceptrón.



Fuente: www.linkedin.com/pulse/el-perceptr%C3%B3n-la-unidad-b%C3%A1sica-de-las-redes-alvaro-m-chambi-reinaga/

Figura 1-3 Estructura de un Perceptrón

$$y = f \left[\sum_{i=1}^n (x_i * w_i) + b \right]$$

Fuente : <https://www.geeksforgeeks.org/deep-learning/what-is-perceptron-the-simplest-artificial-neural-network/>

Ecuación 1-1 Función matemática básica de un perceptrón

Donde:

- $X_1, X_2, \dots, X_n$: Son las entradas al sistema
- $W_1, W_2, W_3 \dots W_n$: Son los Pesos asociados a cada entrada
- f : Función de activación (ReLu, TanH, lineal, escalón, etc.)
- b : Sesgo aplicado.

1.2.2. Entradas (X_n)

Son los datos que ingresan al perceptrón, estos pueden ser valores numéricos provenientes de datos o desde otras neuronas.

1.2.3. Pesos (W_n)

Los *pesos* son el coeficiente por el cual se multiplica el dato de la entrada, es la “importancia” que se le da a esa entrada, la cual se va ajustando a medida que el sistema va ‘aprendiendo’.

1.2.4. Sesgo (b)

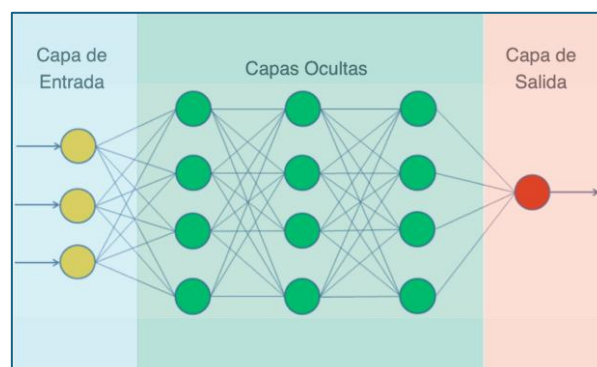
Es independiente de los pesos, pero su funcionamiento es similar, ya que predispone a la neurona a su activación o no, esto le brinda una flexibilidad adicional para modelar patrones complejos de datos en la entrada .

1.2.5. Suma ponderada

Es la suma de las entradas, multiplicada por los pesos correspondientes, más el sesgo. El resultado de lo anterior pasa a la función de activación.

1.2.6. Capas

Las neuronas artificiales se organizan en capas, éstas pueden ser de entrada, ocultas o de salida, como muestra la figura 1-4, las capas de entrada son las encargadas de recibir los datos, mientras que las ocultas son las que procesan los datos y su número depende del objetivo. Luego está la capa de salida la cual proporciona el resultado final del procesamiento de datos. Dependiendo del contexto en que se encuentre, clasificación o regresión, estará compuesta por una o más neuronas. Esto se debe a que en la clasificación el objetivo es asignar una salida binaria o multiclase (perro ,gato...), en cambio en la regresión la salida es un valor numérico (valor de una casa, temperatura...). Toda red neuronal debe de tener a lo menos una capa de entrada y una de salida.



Fuente: <https://aprendeia.com/que-son-las-redes-neuronales-artificiales/>

Figura 1-4 Organización de redes neuronales mediante capas

1.3. FUNCIONES DE ACTIVACIÓN

Es una de las partes más importantes de una neurona artificial, ya que agrega el componente *no lineal* (a excepción de la función identidad) a la red neuronal, con lo anterior es posible el aprendizaje de patrones mediante entrenamiento, clasificación de datos, y predicción de eventos futuros. Sin éstas, las redes neuronales serían solo una repetición de valores lineales. El uso de las funciones de activación tiene que ver con el acercamiento con una respuesta de la red más cercana la realidad, más allá del **sí** o **no**, **1** o **0**, **bueno** o **malo**.

Otra forma de entender cómo trabaja la función de activación (en el ámbito electrónico), sería una analogía al imaginar la *serie de Fourier*, la cual a medida que se van sumando más componente senos y cosenos, la forma de onda resultante tiende a ser similar a la onda original. En este caso, la función de activación va “acomodando” los

sesgos y pesos para obtener una salida semejante a la entrada, todo lo anterior con el entrenamiento supervisado.

Existen variadas funciones de activación y su uso depende de las necesidades de cada caso, a continuación, se detallan alguna de ellas y sus principales características.

1.3.1. Función Lineal

También conocida como función identidad, se utiliza cuando se desea que la salida sea un número, tal como: avalúo de una propiedad, cambio de escalas, etc. Tiene la limitante de no reconocer patrones no lineales o complejos, lo cual limita su uso, su gráfica se presenta en la figura 1-5.

1.3.2. Función Escalón

Usada para clasificar la salida en modo binario: Si o no, bueno o malo, de fácil implementación. Su problemática radica en su cambio extremo entre 0 y 1. La función escalón no es diferenciable, por lo cual no es apta para el uso de algoritmos de aprendizaje. En la figura 1-5 se puede observar la gráfica de la función y su ecuación correspondiente.

1.3.3. Función Sigmoide o logística

Con un rango entre 0 y 1, es la función que más se utiliza, es diferenciable esto es debido a la curva que posee, y con lo cual es posible tener una salida interpretable como probabilidad (0.1, 0.5, 0.8). Una falencia es la saturación con valores muy altos o bajos, esto provoca que, durante el entrenamiento de gradiente descendente, los gradientes sean demasiado pequeños, es decir, que al momento de “aprender” mediante Backpropagation las derivadas serán muy pequeñas afectando así los ajustes de pesos y sesgos, los cuales serán ajustes pequeños o nulos, provocando así que las capas aprendan muy lenta o simplemente dejará de hacerlo en las primeras capas. En la figura 1-5 se observa la gráfica de la función y la ecuación correspondiente.

1.3.4. Función Tangente Hiperbólica

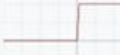
Con un rango de salida entre **-1** y **1** y simétrica en torno al origen (centrada en cero), con el tiempo se ha reemplazado por el uso de otras funciones de activación como ReLU, es una función no lineal, por lo cual es ideal para modelar relaciones no lineales de datos, tiene la misma problemática que la función logística con los valores extremos,

aparte de necesitar una cantidad elevado de cálculo en comparación con otras. En la figura 1-5 se muestra la gráfica de la función y la ecuación correspondiente.

1.3.5. Función ReLU

Una de las funciones de activación más usadas actualmente en Redes Neuronales Convolucionales (CNN), así como su uso en capas ocultas. El atractivo de esta función de activación proviene de la característica que posee , la cual a un valor de entrada menor que cero, muestra cero , en cambio , si el valor de entrada es mayor o igual a cero, muestra el valor de entrada. Con un costo computacional bajo y una derivada simple. La problemática es con números menores a cero, la salida será siempre cero, generando así neuronas “muertas” y la correspondiente lentitud del aprendizaje.

En la figura 1-5 se muestra la forma de onda de cada una de las funciones de activación mencionadas anteriormente.

Name	Plot	Function, $g(x)$	Derivative of $g, g'(x)$	Range
Identity		x	1	$(-\infty, \infty)$
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	0	$\{0, 1\}$
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1 + e^{-x}}$	$g(x)(1 - g(x))$	$(0, 1)$
Hyperbolic tangent (tanh)		$\tanh(x) \doteq \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - g(x)^2$	$(-1, 1)$
Rectified linear unit (ReLU) ^[14]		$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max(0, x) = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$	$[0, \infty)$

Fuente : <https://machinelearninggeek.com/activation-functions/>

Figura 1-5 Funciones de activación

1.4. FUNCIONES DE PÉRDIDA (L)

Es la forma numérica de cuantificar el margen de error entre la predicción del modelo y el valor real, es una guía para el proceso de aprendizaje. En Machine Learning, dependiendo del tipo de problema (regresión o clasificación) se usan diferentes funciones de pérdida, a continuación, se detallarán algunos de los más utilizados .

1.4.1. Error Cuadrático Medio (MSE)

Es la función más utilizada para problemas de regresión (MSE por sus siglas en inglés) debido a que calcula el promedio de los cuadrados de las diferencias entre el valor esperado y el valor real, es continua, diferenciable, los errores se elevan al cuadrado por lo cual amplifica las diferencias. Como desventaja tiene la particularidad de ser sensible a valores atípicos. En la ecuación 1-2 se muestra el cálculo del error cuadrático medio.

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

Ecuación 1-2 Ecuación del error cuadrático medio

Donde :

MSE : Error cuadrático medio

n : Número total de muestras

y_i : Valores esperados

$\hat{y}_i$: Valores predichos por la red

1.4.2. Error absoluto medio (MAE)

El error absoluto medio (MAE por sus siglas en inglés), mide la diferencia absoluta promedio entre el valor predicho y el valor real , es más robusta frente a valores atípicos, y al igual que la anterior, es utilizada en problemas de regresión. Posee la problemática de que su derivada no es continua en cero. En la ecuación 1-3 se muestra el cálculo del error absoluto medio.

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

Ecuación 1-3 Ecuación del Error Absoluto Medio

Donde :

- MAE : Error Absoluto medio
- n : Número total de muestras
- y_i : Valores esperados
- $\hat{y}_i$: Valores predichos por la red

1.4.3. Entropía Cruzada Binaria (BCE)

La entropía cruzada binaria (BCE por sus siglas en inglés) es utilizada cuando es un problema de clasificación con dos opciones (si/no, perro/gato , Spam/No Spam) , su resultado es una predicción con valor de probabilidad entre 0 y 1 de que pertenezca a una clase o categoría. Su uso se asocia con una función de activación sigmoide en la última capa.

En la ecuación 1-4 se muestra el cálculo de la entropía cruzada binaria.

$$BCE = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log (1 - \hat{y}_i)]$$

Ecuación 1-4 Ecuación Entropía cruzada Binaria

Donde :

- BCE : Entropía Cruzada Binaria
- n : Número total de muestras
- y_i : Valores esperados (entre 0 y 1)
- $\hat{y}_i$: Valores predichos por la red

1.4.4. Entropía Cruzada Categórica (CCE)

La entropía cruzada categórica (CCE por sus siglas en inglés) es utilizada para la clasificación multiclase, es de uso estándar para la clasificación de imágenes , funciona muy bien con la función de activación Softmax (que trabaja entre 0 y 1), con la problemática de consumir más memoria si hay muchas clases. En la ecuación 1-5 se muestra el cálculo de la entropía cruzada categórica.

$$CCE = - \sum_{c=1}^c y_i \log(\hat{y}_i)$$

Ecuación 1-5 Ecuación Entropía cruzada categórica

Donde :

CCE : Entropía Cruzada Categórica

c : Número total de clases

y_i : Valores esperados para la clase c

$\hat{y}_i$: Probabilidad predicha por la red para la clase c

1.5. ALGORITMOS DE OPTIMIZACIÓN

Los algoritmos de optimización son métodos matemáticos que ajustan los pesos y sesgos de una red neuronal, esto se hace con la finalidad de minimizar la función de pérdida (L) y así mejorar su rendimiento, a continuación, se detallan algunos algoritmos de optimización más utilizados.

1.5.1. Descenso de gradiente y variantes

Es uno de los algoritmos de optimización más básicos y ampliamente utilizado para el entrenamiento de redes neuronales artificiales buscando el mínimo de pérdida . Este método consiste en calcular la salida del dato desde la neurona de entrada ,se calcula el error, el cual se devuelve hacia atrás ajustando pesos y sesgos, esto se hace mediante el uso de los gradientes calculados por el *Backpropagation*. Es de uso simple y fácil de implementar, pero puede ser lento y quedar atrapado en mínimos locales.

1.5.2. ADAM

Es el algoritmo de optimización más popular (ADAM por sus siglas en inglés), combina ventajas de varios otros algoritmos (Momentum y RMSprop), por lo cual es particularmente usado por su capacidad para manejar grandes cantidades de datos y modelos con millones de parámetros. Tienen la ventaja de ser muy rápido, estable y su

funcionalidad con la mayoría de los problemas. Pero tiene la problemática de caer en el sobreajuste.

1.5.2.1 Tasa de aprendizaje (η)

La tasa de aprendizaje es un concepto fundamental en el Machine Learning, es un hiperparámetro que determina cuánto “se mueve” en el descenso de gradiente. Es un número muy pequeño, del orden del 0.01-0.001 y con él se ajustan los pesos en cada iteración. Este factor no puede ser muy grande ni muy pequeño, esto debido a que al ser muy grande puede oscilar alrededor de la solución óptima al momento de encontrarla, y en cambio, si es muy pequeña, puede quedar atascado en soluciones locales que no son las óptimas globales, su correcta configuración puede marcar la diferencia entre un modelo que converge rápidamente y uno que falla por divergencia o underfitting. En la sección 3 se verá su aplicación en los ejemplos planteados.

1.6. PROCESOS DE APRENDIZAJE

En el siguiente apartado se explicarán algunos procesos de aprendizaje más utilizados en la actualidad.

1.6.1. Feedforward (FNN)

También llamada Propagación hacia adelante, es un tipo de arquitectura de una red neuronal en donde la información fluye en una sola dirección (desde la entrada hasta la salida), en su estructura básica posee una capa de entrada, una capa oculta y una capa de salida. Este tipo de redes son utilizadas en tareas de clasificación y regresión. Matemáticamente se podría clasificar en etapas:

- a. Cálculo de la suma ponderada
- b. Cálculo de la Función de activación aplicada
- c. Cálculo del error

1.6.2. Backpropagation

Algoritmo que se utiliza para el entrenamiento de redes neuronales artificiales, es parte esencial dentro del método de *descenso por gradiente*, su propósito es el cálculo del gradiente de los pesos y sesgos, para que luego estos sean ajustados y allí conseguir el

mínimo error. Todo lo anterior lo realiza mediante el cálculo de la derivada con el uso del método de la ***Regla de la cadena***. En pocas palabras, este algoritmo calcula las derivadas (gradientes), datos que son usados en el descenso por gradiente.

1.7. **TIPOS DE APRENDIZAJE**

En machine Learning existen variados métodos de aprendizaje, algunos de ellos son descritos a continuación.

1.7.1. Aprendizaje supervisado

Es el tipo de aprendizaje en el cual se necesitan datos históricos de entrada y salida, o “cargar” con la información de entrada y lo que se espera en la salida, con este tipo de aprendizaje la red neuronal se va ajustando a medida que va aprendiendo. Este tipo de aprendizaje se utiliza en el reconocimiento de imágenes u objetos, análisis predictivo, detección de spam, predicción de precios inmobiliarios, encontrar factores de riesgo de una enfermedad, predecir fallos en piezas mecánicas. Entre sus desventajas se encuentran que en ciertos casos requieran mucho tiempo de entrenamiento, los datos ingresados posean errores humanos con lo cual el algoritmo aprenda equivocadamente.

Dentro del aprendizaje supervisado existen dos tareas o problemas que son abordados comúnmente, estos son:

1.7.1.1 Regresión

Es un tipo de problema de aprendizaje supervisado, el cual tiene como finalidad predecir valores numéricos continuos, a partir de un conjunto de datos de entrada. Responde a la pregunta “¿Cuánto?” y su uso se destina para la predicción de valores (de acciones, venta de propiedades, etc.)

1.7.1.2 Clasificación

Es otro tipo de problema del aprendizaje supervisado, el cual tiene como finalidad el predecir una clase o categoría discreta a partir de los datos de entrada. Responde a la pregunta “¿Qué es esto?” (perro/gato , bueno/malo , etc.)

1.7.2. Aprendizaje no supervisado

A diferencia del anterior, en este caso no necesita de datos etiquetados en la entrada y salida, sino que la máquina va buscando patrones ocultos sin la necesidad de intervención humana. En este tipo de aprendizaje el modelo explora los datos por sí mismo y aprende a organizarlos de manera lógica, es útil para agrupar segmentación de clientes, realizar análisis de datos iniciales, clustering, análisis de datos meteorológicos, etc.

1.7.3. Aprendizaje por refuerzo (RL)

Es un método de aprendizaje por “recompensas y castigos”, a diferencia de los métodos anteriores donde se intenta reducir el error, esta técnica se enfoca en aprender a partir de su propia experiencia, con errores y aciertos mediante interacción con el entorno recibe retroalimentación en forma de recompensas o castigos (sumar o restar puntos) este método es útil cuando la finalidad es optimizar la toma de decisiones secuenciales en entornos dinámicos y complejos.

1.8. TIPOS DE REDES NEURONALES

Existen varios tipos de redes neuronales, las cuales se pueden clasificar según su finalidad de uso, el número de capas, tipo y grado de conexiones o por el tipo de arquitectura. En los siguientes párrafos se explicará resumidamente los tipos de redes neuronales y usos más frecuentes.

1.8.1. Redes Neuronales Artificiales (ANN)

Compuestas por capas de nodo conectados por pesos, este tipo de redes son usadas generalmente en problemas de regresión (predicción de precios), reconocimiento de patrones (escritura a mano) y clasificación (detección de correo spam).

1.8.2. Redes Neuronales Convolucionales (CNN)

Utilizadas en el reconocimiento de objetos, clasificación de imágenes y reconocimiento de patrones, estas redes utilizan el álgebra lineal, especialmente la multiplicación de matrices para identificar patrones dentro de una imagen. Lo anterior lo hacen aplicando un filtro (kernel) sobre la imagen para generar un mapa de características

1.8.3. Redes Neuronales Recurrentes (RNN)

Una red neuronal recurrente es un tipo de neurona artificial que utiliza datos secuenciales o datos de series de tiempo, estos se utilizan habitualmente para reconocimiento de voz, subtítulos de imágenes o traducción de idiomas, un ejemplo es *Siri* en búsquedas por voz y *Google Translate*. Su característica se basa en su “memoria”, ya que tienen la capacidad de mantener información sobre entradas anteriores gracias a su estructura de bucles.

1.8.3.1 Redes tipo LSTM y GRU

Un problema que tienen las RNN simples, es la “memoria” a largo plazo y el problema de desvanecimiento de gradiente, esto se solucionó en cierta forma con las redes tipo LSTM (Long Short-Term Memory) y tipo GRU (Gated Recurrent Unit).

- **Redes tipo LSTM**

Son redes tipo RNN con una arquitectura para aprender a largo plazo, esto lo hace mediante compuertas que controlan el flujo de información (puerta de entrada, puerta de olvido y puerta de salida). Son utilizadas en tareas de traducción automática y predicción de series temporales.

- **Redes tipo GRU**

La red neuronal de unidades recurrentes cerradas, son una simplificación de las redes LSTM, pero con menor complejidad computacional. Con dos puertas (actualización y reinicio) es más rápida y menos compleja que una LSTM, pero su memoria a largo plazo es menor en comparación. Su aplicación se centra en análisis de series temporales, procesamiento de lenguaje natural y reconocimiento de voz.

1.9. MATEMÁTICA DE LA NEURONA

A continuación, se detallan ciertos parámetros matemáticos básicos para un mejor entendimiento de la forma de trabajar de las redes neuronales, así como un ejemplo aplicado.

1.9.1. Derivada del error respecto a la salida (predicción)

Es la medida de cuanto varía el error ante una pequeña variación de la predicción, la cual se obtiene derivando la función de pérdida con respecto a la salida $\hat{y}$. El siguiente ejemplo es aplicable cuando se utiliza MSE para cálculo del error. En la ecuación 1-6 se muestra cómo la variación de $\hat{y}$ afecta en la variación del error.

$$\frac{\partial L}{\partial \hat{y}} = (\hat{y} - y)$$

Ecuación 1-6 Derivada del error en función de la salida(predicción)

Donde :

y : Valor deseado

$\hat{y}$: Valor entregado (predicción)

1.9.2. Derivada del error respecto a la suma ponderada (δ)

Conocido como “error local” de la neurona, determina cuánto varía el error total ante un pequeño cambio en la suma ponderada (z), antes de que pase por la función de activación. En la ecuación 1-7 se muestra el cálculo del error respecto a la suma ponderada.

$$\delta = \frac{\partial L}{\partial z} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z}$$

Ecuación 1-7 Ecuación del error respecto a la suma ponderada

1.9.3. Derivada del error respecto al peso

Es utilizada para conocer la contribución del peso (w_n) en el error (L) total, usada en descenso por gradiente. En la ecuación 1-8 se muestra el cálculo de la derivada del error con respecto al peso.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial w} = \left(\frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} \right) * \frac{\partial z}{\partial w} = (\hat{y} - y) * (\hat{y} * (1 - \hat{y})) * x_i$$

Ecuación 1-8 Cálculo de la derivada del error con respecto al peso

1.9.4. Derivada del error con respecto a la entrada (X_n)

Es utilizada para conocer cómo afecta la entrada al error, usada en redes profundas. En la ecuación 1-9 se muestra el cálculo de la derivada del error con respecto a la entrada.

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial x} = \left(\frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} \right) * \frac{\partial z}{\partial x} = -(y - \hat{y}) * (\hat{y} * (1 - \hat{y})) * w_n$$

Ecuación 1-9 Derivada del error con respecto a la entrada

1.9.5. Derivada del error con respecto al sesgo (b)

Este dato muestra la importancia de los cambios en el sesgo que afectan al error, es decir, que tanto influye el error en función del sesgo. En la ecuación 1-10 se muestra el cálculo de la derivada del error con respecto al sesgo.

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial b} = -(y - \hat{y}) * (\hat{y} * (1 - \hat{y})) * 1$$

Ecuación 1-10 Derivada del error con respecto al sesgo

1.9.6. Actualización de pesos (w)

En la ecuación 1-11 se muestra el cálculo para actualizar el peso, con los datos obtenidos anteriormente.

$$w_n = w_{old} - \eta * \frac{\partial L}{\partial w}$$

Ecuación 1-11 Actualización de pesos

Donde :

w_n : Nuevo peso calculado

w_{old} : Peso antiguo entregado

η : Tasa de aprendizaje (valor entre 0 y 1)

$\frac{\partial L}{\partial w}$: Gradiente de error con respecto a w

1.9.7. Actualización de sesgo (b)

La ecuación 1-12 se muestra el cálculo para actualizar el sesgo.

$$b_n = b_{old} - \eta * \frac{\partial L}{\partial b}$$

Ecuación 1-12 Actualización de sesgo

Donde :

b_n : Nuevo sesgo

b_{old} : Sesgo anterior

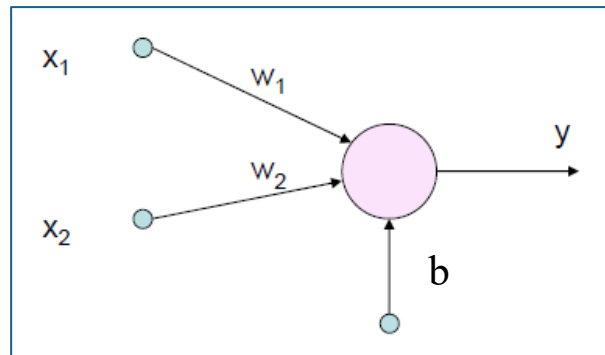
η : Tasa de aprendizaje (valor entre 0 y 1)

$\frac{\partial L}{\partial b}$: Derivada del error con respecto al sesgo

1.9.8. Ejemplo aplicación del descenso por gradiente

A modo de ejemplo numérico se mostrará el cálculo de una red simple con el uso de descenso por gradiente, lo que implica *Feedforward*, *Backpropagation* y otros términos explicados anteriormente.

Se utilizará como ejemplo un ejercicio con dos entradas y una salida, como muestra la figura 1-6.



Fuente : <https://flutter.droppages.com>

Figura 1-6 Ejemplo de perceptrón con dos entradas

Datos :

$$X_1 = 2 \quad X_2 = 3 \quad w_1 = 0.1 \quad w_2 = 0.3 \quad y_{\text{esperada}} = 1$$

$$b = 0 \quad \text{Error : MSE} \quad F. \text{ activación : Sigmoid} \quad \alpha = 0.1$$

- a. En primer lugar, se calcula la salida con los datos entregados (suma ponderada)**

$$\text{Entonces : } z = x_1 * w_1 + x_2 * w_2 + b = 2 (0.1) + 3(0.3) + 0 = 1.1$$

- b. Luego se aplica la función de activación :** Este sería el valor de la predicción

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-1.1}} \approx \mathbf{0.7503}$$

Esto indica que la proyección es bastante lejana al valor deseado ($y=1$)

c. Posteriormente se calcula la derivada del error , en este caso se usará MSE (Error cuadrático medio)

$$\frac{\partial L}{\partial \hat{y}} = (\hat{y} - y) = (0.7503 - 1) \approx \mathbf{-0.2497}$$

El valor y el signo que entrega como resultado es muy importante , ya que cuando el signo del gradiente es positivo, indica que el error y el peso van en la misma dirección, es decir, cuando aumenta el peso, el error aumenta. Por el contrario, cuando el signo es negativo, si aumenta el peso, el error disminuye. Un número bajo indica que se está cerca de un mínimo local o que la red ya aprendió (convergencia).

d. Luego se obtiene la derivada de la función de activación , que en este caso es sigmoide.

$$\text{Si: } \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\text{Entonces: } \sigma'(z) = \hat{y} * (1 - \hat{y}) = 0.7503 * (1 - 0.7503) = \mathbf{0.1873}$$

e. Derivada de la suma respecto al peso

En este paso , se deriva la suma ponderada con respecto al peso

Como :

$$z = x_1 * w_1 + x_2 * w_2 + b$$

$$\text{Entonces: } \frac{\partial z}{\partial w_n} = X_n$$

$$\frac{\partial z}{\partial w_1} = x_1 = 2 \quad ; \quad \frac{\partial z}{\partial w_2} = x_2 = 3$$

f. Finalmente es posible realizar el cálculo de gradientes (aplicación de Backpropagation) , que es la derivada del error con respecto al peso

En este paso se toman los valores calculados en los puntos **c, d** y **e**

$$\frac{\partial L}{\partial w_n} = (\hat{y} - y) * \sigma' * X_n = (\hat{y} - y) * (\hat{y} * (1 - \hat{y})) * X_n$$

Entonces :

$$\frac{\partial L}{\partial w_1} = -0.2497 * (0.1873) * 2 \approx -0.0935$$

$$\frac{\partial L}{\partial w_2} = -0.2497 * (0.1873) * 3 \approx -0.1403$$

El signo negativo de los resultados indica que, para reducir el error, se deben aumentar los pesos y el número tan pequeño indica cercanía con el mínimo, si el error fuese demasiado grande, este número sería mucho mayor.

g. Actualización de pesos

En este paso se aplica la fórmula de actualización de pesos (ecuación 1-11)

$$w_1 = w_{old\ 1} - (\alpha * \frac{\partial L}{\partial w_1}) = 0.1 - (0.1 * -0.0935) = 0.1093$$

$$w_2 = w_{old\ 1} - (\alpha * \frac{\partial L}{\partial w_1}) = 0.1 - (0.1 * -0.1403) = 0.3140$$

Con estos nuevos de pesos, se vuelve a realizar todo el cálculo otra vez, hasta que llegue al error mínimo posible y la predicción sea lo más cercana al valor buscado, como se muestra en la tabla 1-1, la evolución de los resultados se muestra en la interacción número 20, en donde la predicción llega muy cerca del valor ideal (= 1), y el error baja a medida que avanzan las interacciones.

Tabla 1-1 Tabla de interacciones

Interacción	Predicción	Error
0	0.7503	0.2497
Actual	0.7712	0.2288
2	0.7915	0.2085
10	0.9150	0.085
15	0.9612	0.0388
20	0.9885	0.0115

1.10. APLICACIONES DE LAS REDES NEURONALES

El campo de las aplicaciones de las redes neuronales va desde diagnósticos médicos mediante imágenes, análisis de datos, detección de objetos, reconocimiento facial, traducción automática, chatbots y asistentes virtuales, generación de textos, transcripción de voz a textos, en las plataformas de streaming se usan para recomendar películas y series, en finanzas detección de fraudes y predicción de los mercados , control de calidad en líneas de producción, predicción de fallos, conducción autónoma y un largo etc.

1.11. OBJETIVOS

En este apartado se describen los objetivos tanto general como específicos, los cuales se deben lograr en el proceso de este trabajo de título.

1.11.1. Objetivo General

El trabajo de título tiene como finalidad confeccionar una guía práctica para el entrenamiento de redes neuronales con el software Python, esto se realizará mediante la explicación simplificada de ciertos parámetros que están insertos en los sistemas mencionados.

1.11.2. Objetivos Específicos

Como objetivos específicos se abordarán los conceptos:

- Describir en qué consiste una red neuronal
- Presentar modelos matemáticos de una red neuronal
- Programar una red neuronal mediante Python
- Presentar códigos base en Python para el uso de redes neuronales en diferentes ámbitos
- Obtener resultados prácticos en los códigos bases de Python.

CAPÍTULO 2: HERRAMIENTAS PARA EL DESARROLLO EN PYTHON

2. HERRAMIENTAS PARA EL DESARROLLO EN PYTHON

En este apartado se explica el entorno de desarrollo, en el cual es posible preparar los proyectos de redes neuronales, tales como: Cómo crear un perceptrón simple mediante Python, implementar una red neuronal y el cálculo del Backpropagation.

Como consejo al lector, es recomendable la instalación de un software con una versión adecuada de Python para las librerías que se usarán a futuro, ya que puede existir la complejidad de instalación debido al soporte de dicha librería con la versión de Python instalada. Por lo anterior es recomendable verificar las compatibilidades de las versiones a instalar. Otra opción es el trabajo con el editor *Google Colab* (<https://colab.google.com>), página donde es posible desarrollar y ejecutar Python en línea, así como sus librerías sin la necesidad de descargar ningún tipo de archivo.

2.1. PYTHON Y REDES NEURONALES

En los siguientes puntos se realizará una breve introducción a Python, sus usos y el motivo por el cual es uno de los lenguajes de programación preferidos.

2.1.1. ¿Qué es Python?

Python es un lenguaje de programación de alto nivel interpretado, esto quiere decir, que no necesita ser compilado, es de código abierto orientado a objetos. Creado por el programador holandés Guido van Rossum en 1990. El nombre viene dado por un programa de televisión favorito del creador. Python es preferido por gran parte de desarrolladores, gracias a su fácil lectura, su reducida línea de códigos, la amplia oferta de soluciones gracias a su librería, en las cuales existen soluciones para el desarrollo web, ciencia de datos, Machine Learning (ML), la interoperabilidad con otros lenguajes, una comunidad de soporte en línea, soporte multiplataformas, entre otros. En la figura 2-1 se muestra el logo oficial de Python .



Fuente : <https://www.python.org/>

Figura 2-1 Logo Python

2.1.2. Librerías de Python

Las librerías o biblioteca de Python son uno de los puntos fuertes por lo cual es elegido por desarrolladores, ya que les permite acceder a colecciones de funciones, clases y métodos predefinidos que hacen de este software una poderosa herramienta, a continuación, se describen algunas de las librerías más populares.

2.1.2.1 Numpy

Librería de código abierto, orientada para el cálculo matemático, para operaciones con matrices y algebra lineal, es recomendada para proyectos de Data Science.

- Permite Trabajar con arrays multidimensionales
- Convertir listas en array, matriz ndarray, cadena o CSV
- Incluye funciones para matemáticas, lógica, estadísticas.

2.1.2.2 Pandas

Es una librería especializada en el manejo y análisis de estructuras de datos , sus principales características son:

- Permite leer y escribir ficheros en formato CSV, EXCEL y bases de datos en SQL.
- Acceder a los datos mediante índices o nombres
- Trabajar series temporales.

2.1.2.3 Matplotlib y Seaborn

Ambas son potentes librerías para la visualización de datos y crear gráficos visualmente atractivos, mientras que el primero ofrece mayores opciones de personalización y control, la segunda entrega facilidad en el código de ingreso.

2.1.2.4 Scikit- learn

Es una librería de código abierto y herramienta para Machine Learning en Python, es una herramienta básica para realizar análisis predictivo, clasificaciones, algoritmos de clúster.

2.1.2.5 TensorFlow

Librería que permite crear y entrenar redes neuronales para detectar patrones y razonamientos, aparte de lo anterior TensorFlow tiene la característica de trabajar en multiplataformas.

2.1.2.6 Keras

Una de las API más utilizadas para el desarrollo y pruebas de redes neuronales, se ejecuta sobre frameworks como Theano y TensorFlow. Facilita la creación de capas para las redes neuronales o configuración de arquitecturas complejas.

2.1.2.7 PyTorch

Es un marco de aprendizaje profundo de código abierto, se basa en software para crear redes neuronales, otra característica es la capacidad de ejecutar y probar partes de un código en tiempo real, en lugar de esperar a que se implemente el código completo en el ámbito de Data Science.

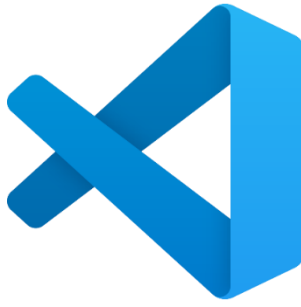
2.2. ENTORNOS DE DESARROLLO

Como se ha mencionado anteriormente, Python es un lenguaje de alto nivel y muy accesible, pero requiere un software tipo “intérprete” para que el código fuente pueda ser ejecutado, gracias a las herramientas que poseen dichos softwares, la programación en el entorno es más rápida y efectiva, herramientas tales como: autocompletado de código, herramientas de depuración, ejecución de partes del código, uso de librerías, etc.

A continuación, se mencionan algunos de los softwares utilizados.

2.2.1. Visual Studio Code

Es un editor de código gratuito multiplataformas, desarrollado por Microsoft, es una herramienta robusta que es utilizada para programar en Python, este editor posee variadas herramientas al momento de programar, tanto para principiantes como para profesionales, gracias a autocompletado inteligente, su Marketplace de extensiones, una interfaz intuitiva y capacidad de personalización. En la figura 2-2 se muestra el logo oficial de Visual Studio Code.



Fuente : <https://code.visualstudio.com/>

Figura 2-2 Logo de Visual Studio Code

2.2.2. Jupyter Notebook

Aplicación web de código abierto, la cual permite crear, leer y compartir documentos creados en Python, además de ser compatible con otras plataformas, tales como : Ruby, Julia, Perl, Matlab y R. Entre otras características, es posible probar líneas de código separadas, es decir, no es necesario cargar todo el programa para corroborar su funcionalidad, sino que es posible hacerlo por partes. En la figura 2-2 se muestra el logo oficial de Jupyter Notebook.



Fuente : <https://jupyter.org/>

Figura 2-3 Logo Jupyter Notebook

2.3. PROBLEMAS COMUNES EN EL ENTRENAMIENTO

Al momento de entrenar un modelo, estos pueden carecer de un asertividad mínimo, necesaria, lo que puede ser a causa de dos conceptos que son claves en Machine Learning: el Sobreajuste (Overfitting) y Subajuste (underfitting), estos conceptos se

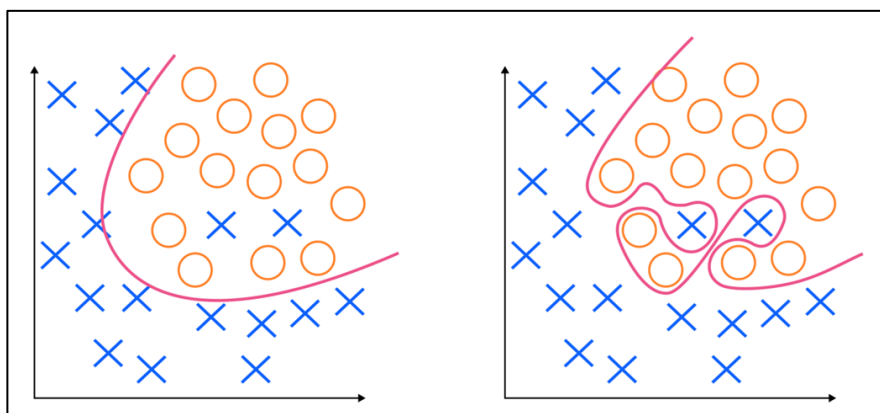
pueden relacionar con el rendimiento que tiene un modelo. El comprender estos conceptos permiten determinar qué tipo de arquitectura o modelo se debe utilizar, si es mejor una red neuronal o red convolucional, un mayor o menor número de capas, o la cantidad de neuronas por capas. Antes de explicar los conceptos mencionados, es necesario sumar un par de ideas adicionales, esto, con la finalidad de comprender las ideas generales.

2.3.1. Overfitting (sobreajuste)

El sobreajuste ocurre cuando el modelo no generaliza los datos entregados, por lo que se “pierde” en su propio entrenamiento, una de las características de este problema es que, al momento de comparar la precisión del entrenamiento y la validación, el porcentaje de precisión en el entrenamiento es alto, pero es muy bajo al momento de validación, por lo cual el modelo está aprendiendo de los datos de entrenamiento, pero no logra extraer las características más importantes (generalizar), sino más bien las características particulares. Este problema se produce habitualmente cuando :

- Un modelo es demasiado complejo
- Pocos datos de entrenamiento

En la figura 2-4 se observa un ejemplo conceptual de las diferencias entre una gráfica correcta y una con Overfitting.



Fuente : <https://www.ibm.com/think/topics/overfitting-vs-underfitting>

Figura 2-4 Ejemplo conceptual de gráfica con y sin Overfitting

2.3.1.1 Cómo prevenir el Overfitting

Para prevenir o disminuir este problema es posible aplicar varias estrategias independientes o combinadas, tales como:

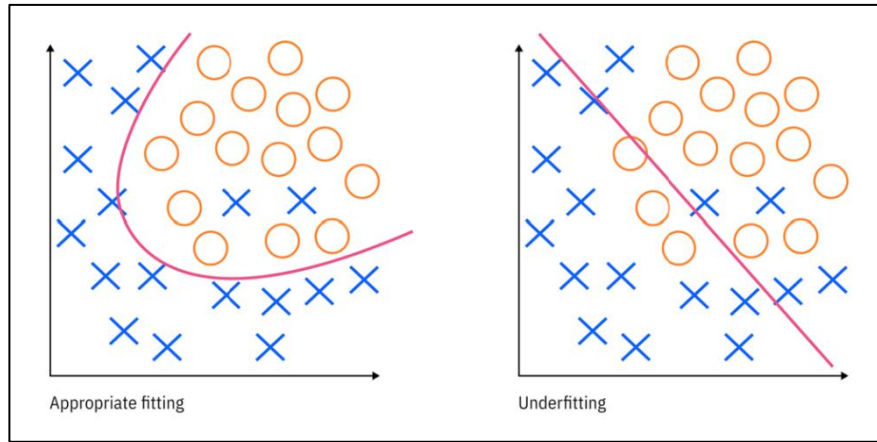
- Obtención de más datos: Aumentar el conjunto de entrenamiento con nueva información o utilizando técnicas de ***data augmentation*** (creación de datos).
- Simplificar el modelo: Reducir el número de capas neuronales o neuronas en las redes .
- Early Stopping (parada temprana): Detener el entrenamiento cuando el error de validación comienza a empeorar.
- Dropout : Apagar aleatoriamente neuronas durante el entrenamiento
- Eliminar características irrelevantes o redundantes: Elegir variables más relevantes para el modelo, evitando el ruido.

2.3.2. Underfitting (subajuste)

El underfitting se presenta cuando un modelo es demasiado simple, provocando que no logre extraer las características básicas de los datos, una de las características de este problema se presenta al momento del entrenamiento y validación, donde en ambos el rendimiento es muy bajo. Las causas del underfitting:

- Modelo demasiado simple: El modelo puede no tener suficientes capas o neuronas por capa.
- Número de interacciones insuficientes: El número de interacciones no logra extraer las características principales de los datos entregados .
- Arquitectura inadecuada: El uso de una Red Neuronal en vez de una Convolutiva para imágenes.

En la figura 2-5 se muestran un ejemplo conceptual de una gráfica apropiada, versus una gráfica con underfitting.



Fuente: <https://www.ibm.com/think/topics/overfitting-vs-underfitting>

Figura 2-5 Ejemplo conceptual de gráficas con y sin Underfitting

2.3.2.1 Cómo prevenir el Underfitting

Existen variadas técnicas que rectifican o previenen el underfitting, las cuales trabajan de manera independiente o en conjunto, éstas son algunas:

- Uso de modelos más complejos: En una Red neuronal , aumentar el número de neuronas o el número de capas ,de regresión lineal a una polinomial o modelo ensamble
- Añadir más características (features): En ciertas circunstancias se pueden crear nuevas características de datos ya existentes (ej.: si se tiene el peso y talla de personas, se puede crear una característica tipo “Índice de masa corporal”).
- Reducción de la regularización: Este mecanismo evita que el modelo se sobre ajuste (Overfitting) pero el exceso genera el underfitting.
- Mas tiempo de entrenamiento: Aumentar epochs, ajustar tasa de aprendizaje.

2.4. TÉCNICAS DE OPTIMIZACIÓN

Las técnicas de optimización son esenciales para optimizar el rendimiento de los modelos de Deep Learning y Machine Learning, para así evitar problemas como el Overfitting y el underfitting, en la siguiente sección se entregan algunas técnicas y cómo implementarlas.

2.4.1. Early Stopping (Detención temprana)

Esta técnica consiste en la interrupción del entrenamiento en el instante en que se visualice un aumento en el valor de error de validación, con esta técnica se logra entrenar al modelo para que el error sea el mínimo y, por consecuencia, la precisión sea máxima, disminuyendo así el Overfitting.

2.4.2. Dropout

Es una técnica de regulación, que tiene como objetivo final el prevenir el sobreajuste (Overfitting) durante el proceso de entrenamiento del modelo. El funcionamiento de esta técnica se basa en la desactivación aleatoria de neuronas en el momento del entrenamiento.

Con la desactivación aleatoria de neuronas en una capa , mientras se produce el entrenamiento de una red neuronal, evita que el modelo dependa excesivamente de combinaciones específicas de neuronas, instando a una representación más generalizada de los datos. Esta técnica es utilizada mayormente en modelos de redes neuronales profundas (Deep Learning), tales como CNN o RNN.

2.4.3. Data Augmentation (aumento de datos)

En el ámbito del Deep Learning, la necesidad de grandes cantidades de datos etiquetados para el entrenamiento es crucial, pero la obtención de estos puede ser costoso, lento o inviable. Es en este escenario que el *Data Augmentation* aparece como una estrategia eficiente que tienen los modelos al momento de generalizar los datos, sin la necesidad de recopilar datos adicionales.

2.4.3.1 ¿Qué es el Data Augmentation?

El Data Augmentation o aumento de datos, es una técnica que consiste en aumentar artificialmente el conjunto de datos para el entrenamiento, esto se hace mediante el uso de datos ya existentes. Lo anterior contrasta con los **datos sintéticos**, los cuales son creados artificialmente sin utilizar el conjunto de datos originales. La técnica de aumento se utiliza en diferentes áreas, tales como: imágenes, textos, series de tiempo y audio. Esta técnica ayuda a reducir el sobre ajuste, mejorar la generalización del modelo y aumentar de manera artificial el tamaño del dataset.

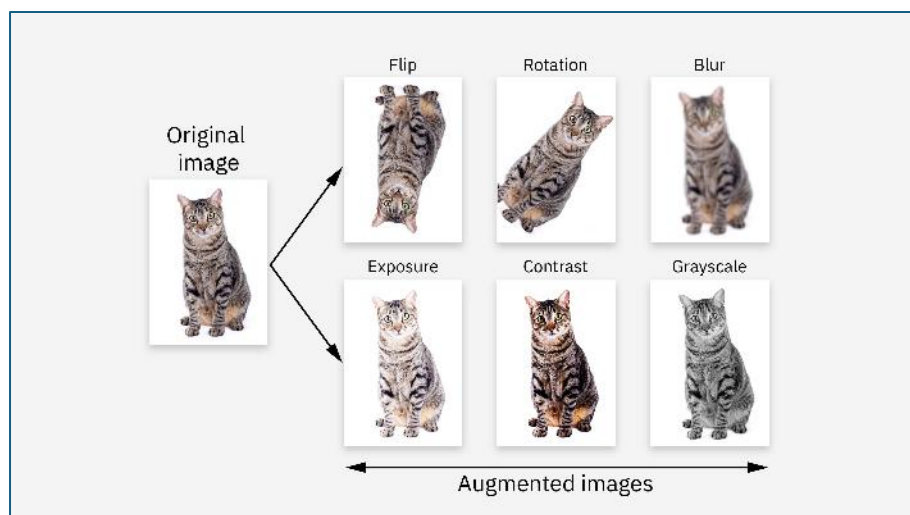
2.4.3.2 Tipos de Data Augmentation

En esta sección se describirá una lista de tipos de Data Augmentation en diferentes áreas y sus características más relevantes .

- Imágenes :

En la figura 2-6 , es posible observar un ejemplo de aumento de datos de imágenes, esto mediante diferentes técnicas de manejo de imágenes. Este tipo de técnicas es posible utilizarlas en el área de la salud, donde el adquirir imágenes médicas y el etiquetado del conjunto de datos lleva mucho tiempo y es costoso, además de la necesidad de un especialista en la materia que valide la información.

- ✓ Transformaciones geométricas: Zoom aleatorio, recortes, rotación y estiramiento de imágenes.
- ✓ Transformaciones de espacio de color : Cambia aleatoriamente los canales de color RGB, el contraste, brillo, nitidez, saturación o ruido.
- ✓ Borrado aleatorio : Suprime alguna parte de la imagen inicial.
- ✓ Mezcla de imágenes: Combina y mezcla varias imágenes.

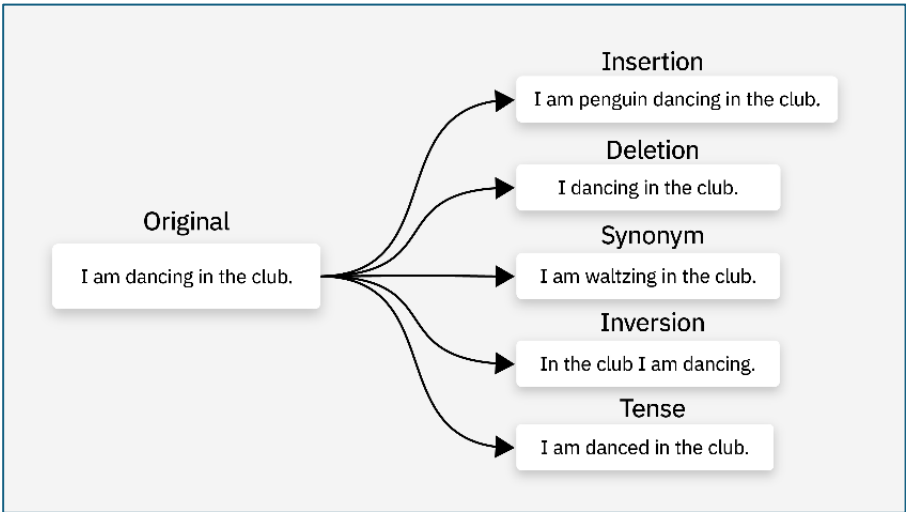


Fuente : <https://www.ibm.com/es-es/think/topics/data-augmentation>

Figura 2-6 Ejemplo de Data Augmentation en imágenes

- Texto:
- ✓ Sustitución de palabras: Trabajo con sinónimos, se busca la palabra y es reemplazada por su sinónimo.
- ✓ Sequence Manipulation (manipulación de secuencia): Esta técnica altera la estructura de la oración para generar nuevas variantes.
- ✓ Eliminación /adición de palabras aleatorias: Esta técnica permite eliminar o agregar palabras al azar.

En la figura 2-7 es posible observar un ejemplo de Data Augmentation aplicado en texto.



Fuente : <https://www.ibm.com/es-es/think/topics/data-augmentation>

Figura 2-7 Ejemplo de Data Augmentation de Texto

2.5. IMPORTANCIA DEL HARDWARE

Para el entrenamiento de redes neuronales profundas es necesaria la capacidad de computar enormes cantidades de datos (en ocasiones miles de millones), por lo cual el hardware es clave para el entrenamiento de los modelos. Es así como las memorias RAM, CPU, GPU,VRAM se vuelven preponderantes en el uso y su entrenamiento (pero también

más costosas). En los siguientes párrafos se explicará, resumidamente, la importancia de cada parte en el procesamiento de datos.

2.5.1. La CPU en redes neuronales

La Unidad Central de Procesamiento (CPU por sus siglas en inglés) es el “cerebro” que controla el flujo de datos, se encarga de prepararlos y coordinar el envío de información hacia la GPU. Actúa como el controlador del sistema, la CPU no se encarga de los cálculos pesados, sino del entorno del entrenamiento.

2.5.2. La GPU en redes neuronales

La Unidad de Procesamiento Gráfico (GPU por sus siglas en inglés) es la que realiza el trabajo principal del Deep Learning, ejecuta cálculos masivos de forma paralela, cualidad que la diferencia con la CPU, su uso es especialmente necesario en la multiplicación de matrices y operaciones tensoriales en paralelo durante el entrenamiento. Esto lo puede hacer, debido a los miles de núcleos que posee, en comparación a los pocos que posee la CPU.

2.5.3. Uso de RAM en redes neuronales

La Memoria de Acceso Aleatorio (RAM por sus siglas en inglés), es la encargada principal del almacenaje temporal del sistema, almacena los conjuntos de datos antes que se envíen a la GPU. Permite cargar datasets y mantener la información lista para que la CPU y la GPU trabajen más rápidamente. Se debe tener en cuenta que, si la RAM es insuficiente, el sistema comienza a utilizar el disco duro, con lo cual el proceso se ralentiza drásticamente.

2.5.4. Uso de la VRAM en redes neuronales

La Memoria de Acceso Aleatorio de video (VRAM por sus siglas en inglés) es la memoria de la GPU y es la encargada de almacenar los pesos del modelo, las activaciones de cada capa y los gradientes durante el entrenamiento. Se debe tener en consideración que el entrenamiento de modelos demasiado grandes, requieren mucha VRAM por lo que se debe considerar el uso de GPUs de alto rango o múltiples GPUs.

CAPITULO 3 : IMPLEMENTACIÓN PRÁCTICA DE REDES NEURONALES

3. IMPLEMENTACIÓN PRÁCTICA DE REDES NEURONALES

En esta sección, se implementarán tres ejemplos de redes neuronales, ejemplos prácticos y funcionales.

3.1. RED NEURONAL LINEAL SIMPLE

Como ejemplo a desarrollar, se mostrará la creación de una red neuronal simple, en este caso se desarrollará e irá detallando paso a paso las líneas de programación para una mayor comprensión por parte del lector.

Para un trabajo más fluido, es recomendable la descarga de las librerías mencionadas anteriormente (Tensorflow, Numpy ,Pandas , entre otros).

3.1.1. Modelación de una red neuronal lineal simple (Regresión)

En el presente ejemplo se muestra el desarrollo de la configuración de una red neuronal simple en Python, mediante el uso de Visual Studio Code y Jupyter. En el ejemplo se propone entrenar una red neuronal simple para que “aprenda”, mediante datos de entrada y de salida (aprendizaje supervisado) a transformar de escala Celsius a Fahrenheit, si bien es conocida la fórmula de conversión, para este ejemplo básico se tomará como desconocida, es decir, que sólo se tendrán los datos de entrada y de salida como información.

3.1.1.1 Carga de librerías para red neuronal simple

En la primera celda de código de programación de la figura 3-1 se hace el llamado a cargar las librerías **TensorFlow** y **Numpy**, esto con la finalidad de entrenar la red neuronal y trabajar con arreglos numéricos respectivamente.

En la segunda celda de código , mostrada en la figura 3-1, se declaran las variables de entrada y salida, con lo cual se crea un array a partir de los datos de entrada (Celsius) y los datos de salida (Fahrenheit) estos datos serán para que la red neuronal pueda ser entrenada, sumado a lo anterior se debe indicar el tipo de dato que se ingresó (*dtype*).

```
import tensorflow as tf
import numpy as np

[72] ✓ 0.0s

celsius = np.array([-35,-20,-15,0,5,10,15,20,35,], dtype= float) #entradas
fahrenheit = np.array([-31,-4,5,32,41,50,59,68,95,], dtype=float) #salidas

[79] ✓ 0.0s
```

Fuente: Elaboración propia, basada en página web

Figura 3-1 Modelación de red neuronal lineal : Carga de librerías

3.1.1.2 Configuración de parámetros

En la figura 3-2 que se muestra a continuación, en la primera celda de código, se crea una variable llamada “capa” y se usará la librería Keras para crear redes neuronales de manera más fácil y sin tantas líneas de programación .En la línea de código se especifica la capa será una capa “densa”, es decir, cada neurona tiene contacto con todas las neuronas a la capa siguiente, pero en este ejemplo solo tendrá una neurona (*units=1*), con *input_shape* le indica la dimensión de entrada (1) de los datos. En la siguiente línea se usa *sequential* para definir un modelo secuencial, el cual se caracteriza por estructura de pila lineal de capas, en donde los datos fluyen en una sola dirección (desde la entrada hacia la salida).

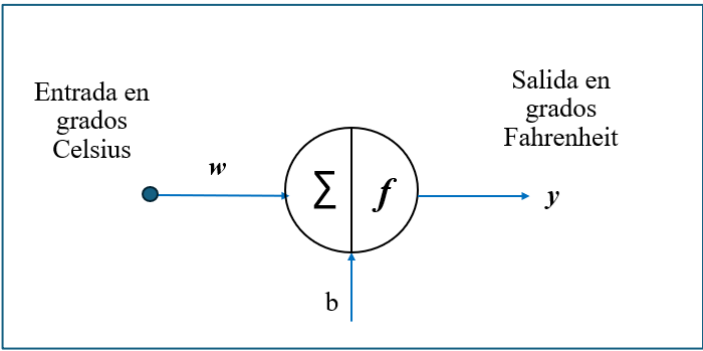
```
capa = tf.keras.layers.Dense(units=1, input_shape=[1])
modelo =tf.keras.Sequential([capa])

[81] ✓ 0.0s
```

Fuente: Elaboración propia, basada en página web

Figura 3-2 Configuración de parámetros de la red

En la figura 3-3 y a modo de representación gráfica del ejercicio, se muestra lo que sería una representación del ejercicio , el cual es bastante simple, ya que se programó con solo una neurona.



Fuente : Elaboración propia mediante MS Power Point, basada en página web

Figura 3-3 Representación gráfica de Neurona lineal simple

3.1.1.3 Compilación y entrenamiento del modelo

En la Figura 3-4 se muestra (en primera línea de programación) el optimizador *Adam* (Sección 1.5.2) el cual es un optimizador usado regularmente para el aprendizaje automático basado en el descenso de gradiente para el entrenamiento. El número asociado (0.1) es la tasa de aprendizaje (Sección 1.5.2.1). La siguiente línea de código indica la *función de pérdida*, la cual se utiliza para medir el error entre la predicción y los valores objetivos, en este ejemplo es utilizado el *error cuadrático medio* (MSE). En la siguiente línea, con *modelo.fit* se da comienzo al entrenamiento, durante este proceso se ajustan sus pesos mediante la reducción de la función de pérdida, en la línea de código los datos a entregar son: datos de entrada, datos de salida, *epochs* (número de veces que el modelo recorre a través de todos los datos de entrenamiento) y el dato *verbose* el cual indica en pantalla las veces del recorrido de datos, lo cual se hace muy poco útil al momento de ingresar 1000 epochs y verlos reflejados en pantalla. Anterior y posteriormente a lo descrito se inserta *print* para mostrar el mensaje indicando el proceso en curso a modo de orientación.

```
curso python > neurona.ipynb > print("comenzando entrenamiento")
+ Code + Markdown | ▶ Run All ⏮ Restart ⏭ Clear All Outputs | 📄 Jupyter Variables 📄 Outline ...
Show Table Of Contents (Outline View)

modelo.compile(
    optimizer=tf.keras.optimizers.Adam(0.1),
    loss="mean_squared_error"
)
[5]

print("comenzando entrenamiento")

historial = modelo.fit(celsius, fahrenheit , epochs=1000, verbose=False)

print("Modelo entrenado")
[6]

... comenzando entrenamiento
Modelo entrenado
```

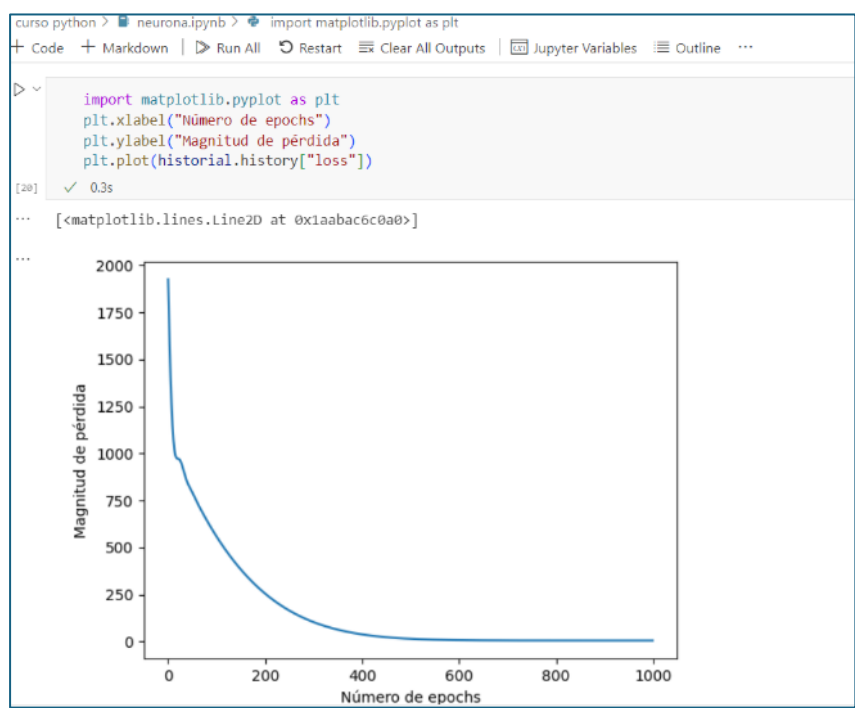
Fuente: Elaboración propia, basada en página web

Figura 3-4 Modelación de red neuronal básica: Modo de operación

3.1.1.4 Gráfica de magnitud de pérdida v/s epochs

En la figura 3-5, se presenta en forma gráfica los resultados de la función de pérdida, en este caso se indica que “tan mal” es el error. Inicialmente el error es alto, pero a medida que el número de épocas va aumentando el error va disminuyendo, casi quedando estacionado desde la epoch número 600 en adelante.

En las líneas de programación es posible observar la carga de la librería *plot*, como primera línea , elemento básico para graficar en Python. Posteriormente en las líneas de comando siguientes se muestran los detalles gráficos que es posible modificar, y finalmente el objeto que se desea graficar (*loss*).



Fuente: Elaboración propia, basada en página web

Figura 3-5 Gráfica de magnitud de pérdida

3.1.1.5 Generar predicción de datos

En la primera y tercera línea de programación de la figura 3-6, se observa la función *print*, con la finalidad de mostrar el mensaje que se desee visibilizar en pantalla, pero es la línea central la que es más preponderante, ya que es la entrada para probar el sistema modelado. En esta línea de código se utiliza *modelo.predict* de Keras para generar predicciones de un modelo entrenado, *np.array [50, 25,33]* crea un array de Numpy con los valores ingresados y con *reshape (3,1)* transforma la forma del array para compatibilizarla con el modelo, en este caso 3 son la cantidad de datos (número de filas)

que serán procesados de forma independiente y el número 1 indica el número de columnas que tendrá el array.

```
print("poner a prueba ")
resultado = modelo.predict(np.array([50, 25,33]).reshape(3,1))
print("El resultado es : " + str(resultado) + " Fahrenheit")

[8]  ✓ 0.0s

... poner a prueba
1/1 ██████████ 0s 38ms/step
El resultado es : [[121.98212]
 [ 76.97992]
 [ 91.38062]] Fahrenheit
```

Fuente: Elaboración propia, basada en página web

Figura 3-6 Gráfico magnitud de pérdida v/s número de epochs

A modo de información, se puede agregar líneas de programación para visualizar cómo evoluciona la corrección de los pesos y sesgos en función de las epochs. Tal como muestra la figura 3-7

```
for epoch in range(1000):
    modelo.fit(celsius, fahrenheit, epochs=1, verbose=False)

    if epoch % 100 == 0:
        pesos = capa.get_weights()
        w = pesos[0][0][0]
        b = pesos[1][0]

        salida = modelo.predict(np.array([[20]]), verbose=0)[0][0]

        print(f"Epoch: {epoch} | w: {w:.4f} | b: {b:.4f} | Entrada de (20°C): {salida:.2f}")
```

Fuente: Elaboración propia, basada en página web

Figura 3-7 Comandos para visualizar variación de pesos y sesgos v/s epochs

Con las líneas de comandos anteriores es posible obtener la información para completar la tabla 3-1, en la cual se muestra la información de la evolución de la salida según el número de epochs. Mediante la variación del peso (w) y el sesgo (b) la neurona va “acomodando” los valores , hasta llegar al valor de salida, que idealmente debería ser 68° Fahrenheit a una entrada de 20° Celsius.

Tabla 3-1 Comparación de datos

Epoch :	w:	b:	Entrada °C	Salida: ° F
0	1.523	0.1000	20	30.55
100	1.8907	9.4953	20	47.31
200	1.8584	17.1434	20	54.31
300	1.8357	22.9085	20	59.62
400	1.8201	26.8890	20	63.29
500	1.8103	29.3765	20	65.58
600	1.8048	30.7747	20	66.87
700	1.8020	31.4806	20	67.52
800	1.8008	31.8006	20	67.82
900	1.8003	31.9309	20	67.94

Fuente: Elaboración propia, basada en datos de programa

Si bien la fórmula de transformación entre grados Fahrenheit y grados Celcius ya es conocida, el ejercicio presentado entrega un ejemplo simple, práctico y detallado de cómo funciona una red neuronal simple. Ya que con datos de entrada y salida, más una programación relativamente simple, es posible generar un aprendizaje del sistema y llegar a una solución de una incógnita mediante una sola neurona. Lo anterior también prepara al lector para los ejemplos venideros, los cuales aumentan en complejidad, pero con una base similar.

3.2. RED NEURONAL CONVOLUCIONAL (CNN)

En este punto se desarrollará un ejemplo básico de una red neuronal recurrente (CNN por sus siglas en inglés), este tipo de redes se caracteriza por su uso preferente en el reconocimiento de imágenes u objetos.

En el presente ejemplo se desarrollará un identificador de **gatos y perros**, con la particularidad del uso de *Transfer Learning* (aprendizaje por transferencia), técnica de aprendizaje que permite el uso de un modelo preentrenado y reutilizarlo en otra tarea a fin.

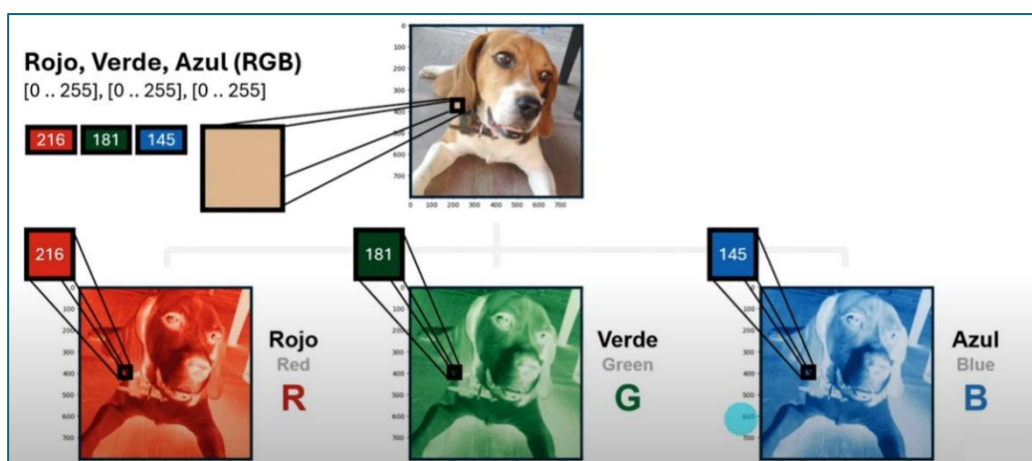
3.2.1. Modelación de una red neuronal convolucional

Las redes neuronales convolucionales utilizan datos tridimensionales para clasificación de imágenes y reconocer los objetos. A grandes rasgos, poseen las mismas características que la red neuronal descrita anteriormente, como son: arquitectura mediante capas, el aprendizaje supervisado, Backpropagation, función de activación. El uso de convolución, el *Pooling* (reducción dimensional de los mapas) y su particular desempeño en el procesamiento de datos con estructura espacial (imágenes o videos) hacen que este tipo de redes sean utilizadas para el procesamiento de imágenes, reconocimiento de objetos, reconocimiento facial, etc.

Una fotografía a color está formada por tres capas de colores **Red, Green y Blue (R,G,B)** gracias a esto es posible generar todos los colores del espectro. Cada píxel de cada capa posee un número específico que va desde el 0 al 255. Es decir, si se toma un píxel, éste está compuesto de tres capas de los colores mencionados, pero gracias a la “graduación” de cada color (rojo, azul y verde) este píxel muestra el color generado en particular. Como muestra la figura 3-8, el color “arena” del píxel seleccionado está formado por tres capas con diferentes graduaciones del mismo píxel.

En el caso de las fotografías en blanco y negro, éstas están compuestas de solo una capa y cada píxel tiene una graduación que va de cero (para el negro absoluto) hasta el 255 (blanco absoluto).

Con el uso de un filtro (kernel), éste se posa sobre toda la fotografía para generar un “escaneo”, generando así un mapa de características para la siguiente capa convolucional, estos filtros pueden ser matrices de 3x3, 5x5 o 7x7 y son usados en la búsqueda de patrones en la fotografía, estos patrones pueden ser texturas, cambio de contrastes, relieves, etc.

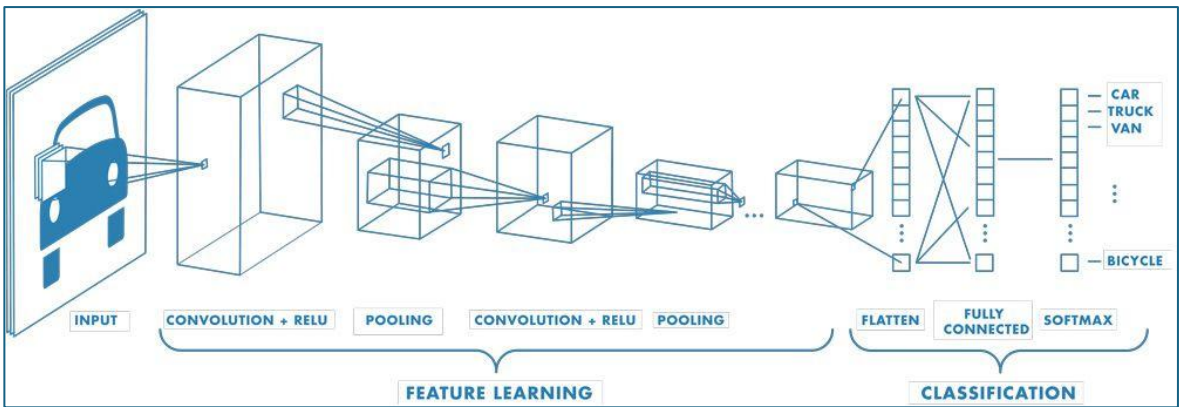


Fuente: <https://medium.datadriveninvestor.com/handling-different-data-types-in-machine-learning-8abb5d0dc43a>

Figura 3-8 Descomposición de colores de una fotografía

Posteriormente, el resultado pasa por la función ReLU, la cual transforma todo número negativo en cero y solo deja los valores positivos (si el valor es -30 lo deja en 0 y si es 15 lo deja igual). Luego se aplica el **Pooling**, método por el cual se reduce la dimensión del mapa de muestreo eligiendo el número mayor de la muestra (Max Pooling) o un promedio (*Average Pooling*). El paso siguiente es repetir varias veces estos procesos hasta llegar a la etapa del “aplanado” (flatten) de los mapas de características, con lo cual deja un solo vector que pasará la información por capas completamente conectadas (FC) similares a las redes neuronales tradicionales. Finalmente, seguirá a la capa de salida, donde suele ser una capa con la función **sigmoide** o **softmax** (clasificación binaria o multiclase respectivamente) con lo cual entregará la probabilidad que la imagen pertenezca a alguna clase.

En la figura 3-9 se muestra, a modo de ejemplo, el desarrollo de lo explicado anteriormente.



Fuente: <https://es.mathworks.com/discovery/convolutional-neural-network.html>

Figura 3-9 Ejemplo de etapas en una red Convolucional

3.2.2. Ejemplo de modelación de una red neuronal convolucional (CNN)

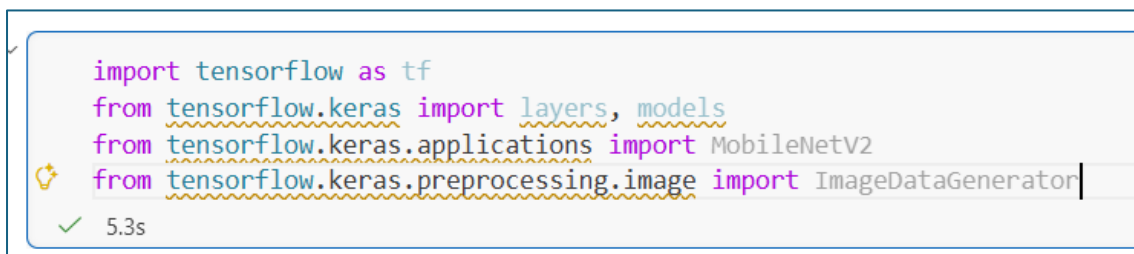
En el presente ejemplo se detallará el proceso para un identificador de animales, específicamente de perros o gatos.

3.2.2.1 Carga de librerías

Al igual que en la modelación anterior, y en todos los casos, en el inicio se cargan las librerías que serán usadas, en este caso:

- **TensorFlow** : Conocida librería para construir y entrenar redes neuronales.
- **Layers, models** : Importa módulos específicos de Keras para crear capas neuronales y modelos secuenciales, con la finalidad de construir la arquitectura de la red.
- **MobileNetV2** : Red neuronal preentrenada con un gran dataset de imágenes , esto ayuda al sistema a no partir de cero en el entrenamiento .
- **ImageDataGenerator** : Herramienta que ayuda a cargar imágenes desde carpetas, procesarlas y generar variaciones de esta.

En la figura 3-10 se muestran los elementos descritos anteriormente, para la modelación de una red neuronal convolucional.



```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Fuente : Elaboración propia , basado en página web

Figura 3-10 Librerías que se deben descargar al inicio de modelación

3.2.2.2 Configuración de parámetros y aumento de datos

En la figura 3-11 las dos primeras líneas de programación indican el redimensionamiento de las imágenes con las que se trabajarán (160x160) y el número de imágenes que procesarán simultáneamente durante el entrenamiento (32). Las siguientes líneas de código describen lo que se denomina **Data Augmentation** (visto en la sección 2.4.3), el cual es un método de aumento de datos artificialmente, esto lo consigue con voltear las imágenes, aplicar zoom aleatorio, girar hasta en 20° aleatoriamente, recortar la imagen, cambio de brillo, etc. esto se realiza con la finalidad de crear datos de forma artificial (con los mismos datos existentes) y así entrenar los modelos sin tanta cantidad de nuevos datos. Es así como:

- **IMG_SIZE = 160** : Define el tamaño en que se redimensionarán todas las imágenes

- ***BATCH_SIZE = 32*** : Indica el número de imágenes que se procesarán en cada iteración.
- ***rescale*** : Realiza una normalización de píxeles de 0-255 a 0-1
- ***rotation* , *zoom* y *flip*** : Crean variaciones artificiales de las imágenes, rotando, realizando zoom y girando las imágenes.

```
# Parámetros
IMG_SIZE = 160
BATCH_SIZE = 32

# Data augmentation
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    zoom_range=0.2,
    horizontal_flip=True
)
```

Fuente: Elaboración Propia , basada en página web

Figura 3-11 Configuración de imágenes y aumento de datos

3.2.2.3 Normalización de imágenes y carga de datos

La figura 3-12 la primera línea de comando realiza una normalización de las imágenes de 0-255 a 1-0 con el comando ***val_datagen***, se debe aclarar que esta normalización de imágenes no es una redundancia con las líneas de comando anteriores, ya que este generador trabaja con otro conjunto de imágenes que serán destinadas para validación. Posteriormente se tiene:

- ***train_generator = train_datagen.flow_from_directory*** : crea un generador de entrenamiento y carga imágenes desde la carpeta indicada en la ruta .
- ***target_size*** : Redimensiona las imágenes a 160x160.
- ***batch_size*** : Cargará 32 imágenes por lote.
- ***class_mode='binary'*** : Como se tienen solo 2 categorías , el modelo usará 0 y 1 para clasificarlos.
- ***val_generator = val_datagen.flow_from_director*** : Generador de dataset para validación .
- ***dataset/validation*** : Carpeta con imágenes para evaluar el modelo .

Las posteriores líneas de comando tienen las mismas configuraciones que el entrenamiento, pero sin *Augmentation*.

```
val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    r'C:\Users\xino_\Downloads\curso python\dataset\train',
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='binary'
)

val_generator = val_datagen.flow_from_directory(
    'dataset/validation',
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='binary'
)
```

Fuente: Elaboración Propia , basada en página web

Figura 3-12 Normalización y carga de datos

3.2.2.4 Modelo base preentrenado (Transfer Learning)

En la siguiente parte de la programación, mediante las líneas de código, se configura el uso de la arquitectura *MobileNetV2*, como se mencionó, es un modelo preentrenado ideal para ser usado como extractor de características, sumado a su rapidez y menor uso de memoria al momento de entrenar, acorta el tiempo de entrenamiento. En la figura 3-13 se muestra las líneas de códigos descritas a continuación.

- ***base_model*** : Indicador de cual modelo se utilizará.
- ***input_shape*** : Dimensiones de las imágenes en las que trabajará 160x160x3 (3 canales en color R,G,B).
- ***include_top=False*** : En este paso, se deja al modelo solo como extractor de características genérico, ya que fue diseñado para clasificar 1000 objetos distintos, pero en este ejercicio solo se clasificarán 2.
- ***weights='imagenet'*** : Le indica al modelo que cargue entrenamiento “aprendido” (pesos) con *dataset ImageNet*. *ImageNet* es una base de datos con más de 14 millones de imágenes.

- ***base_model.trainable = False*** : Le indica al modelo que mantenga “congelado” los pesos de las capas ya entrenadas, y no se modificaran con el nuevo entrenamiento.

```
# Modelo base preentrenado
base_model = MobileNetV2(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,
    weights='imagenet'
)

base_model.trainable = False # Congelamos pesos
```

✓ 1.0s

Fuente: Elaboración Propia , basada en página web

Figura 3-13 Modelo base preentrenado (Transfer Learning)

3.2.2.5 Construcción del modelo final

El siguiente paso de la modelación de la red neuronal es la construcción del modelo final y la definición de sus características. En la figura 3-14 se muestra las líneas de códigos descritas a continuación.

- ***model = models.Sequential*** : Crea un modelo secuencial (capas una detrás de otra).
- ***base_model*** : MobileNetV2 extrae las características de las imágenes.
- ***layers.GlobalAveragePooling2D()*** : Realiza un promedio de las características de los datos , sin perder la información importante (3D en vector 1D).
- ***layers.Dropout(0.3)*** : Desactiva el 30% aleatoriamente durante el entrenamiento, evita el sobreajuste.
- ***layers.Dense(1, activation='sigmoid')*** : Capa final con 1 neurona y con función de activación Sigmoide (salida entre 0 y 1).

```
# Construcción del modelo final
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dropout(0.3),
    layers.Dense(1, activation='sigmoid')
])
```

Fuente: Elaboración Propia , basada en página web

Figura 3-14 Construcción del modelo final

3.2.2.6 Compilación

A continuación, como muestra la figura 3-15, se configura el modelo para el entrenamiento, el “cómo va a aprender” de los errores.

- ***optimizer='adam'*** : Algoritmo de optimización eficiente y adaptativo.
- ***loss='binary_crossentropy'*** : Función de pérdida para clasificación binaria.
- ***metrics=['accuracy']*** : Es la precisión que se visualiza, el desarrollo de los aciertos en pantalla.

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

Fuente: Elaboración Propia , basada en página web

Figura 3-15 Compilación

3.2.2.7 Entrenamiento

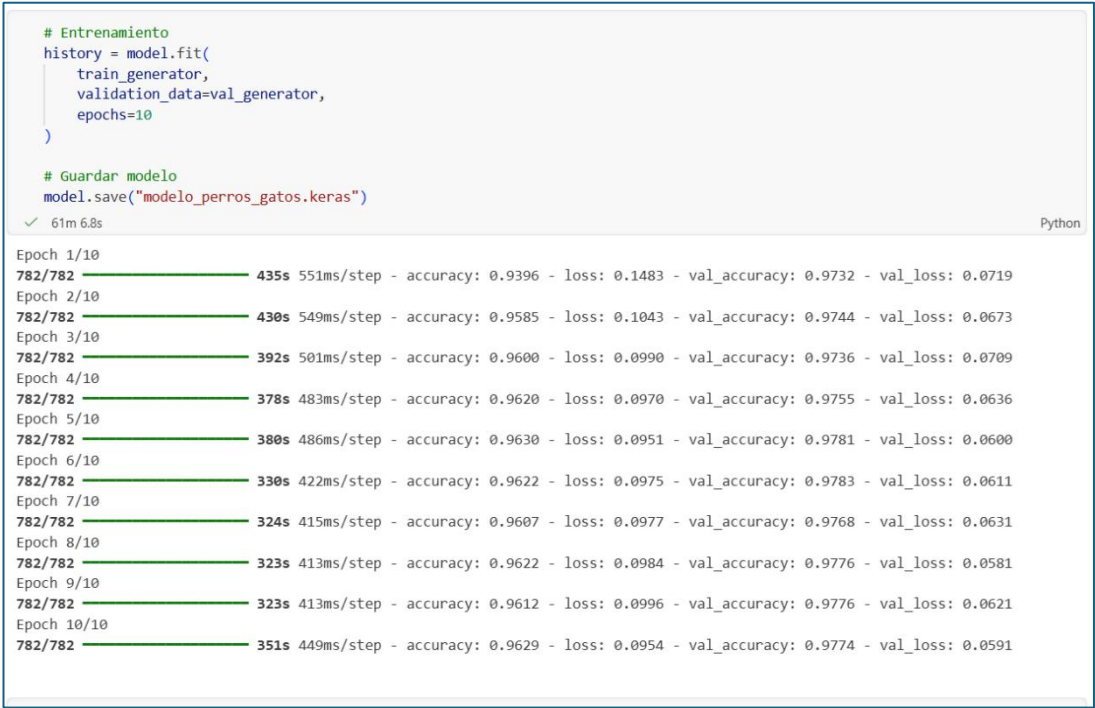
En esta parte el modelo es llamado a entrenarse con las imágenes entregadas, esto lo hace revisando las imágenes (perros y gatos), predice, compara, aprende de sus errores y repite. En la figura 3-16 se muestra las líneas de códigos descritas a continuación.

- ***history = model.fit*** : Función que inicia el proceso de aprendizaje de la red y guardando los resultados en una variable llamada ***history***.
- ***train_generator*** : En esta línea de programación es en donde se le entregan las imágenes para entrenamiento .
- ***validation_data=val_generator*** : Evalúa el modelo con los datos de validación después de cada época .
- ***epochs=10*** : Con esta línea de comando , se le pide al set de entrenamiento que revise cada imagen 10 veces , con lo que el error descenderá de a poco en cada pasada .
- ***model.save("modelo_perros_gatos.keras")*** : El modelo entrenado se guarda en un archivo con el nombre descrito, para luego ser llamado .

Las líneas de información que son mostradas posteriormente entregan datos relevantes del proceso, tales como:

- **Epoch 1/10** : Muestra en qué número de época está el proceso, en este caso son 10 las épocas programadas, de allí del (1/10 , 2/10, etc.)
- **782** : Es el número de batch (pasos) que el modelo da para ver todas las fotografías de entrenamiento , 32 fotografías por batch (son las fotografías “ve” de una sola vez), de un total de 25.000, eso da $781.25 \approx 782$.
- ***accuracy*** : Indica la precisión con que evalúa las fotografías 93.96%, 95.85%.(son valores excelentes), el valor debe ir en aumento, lo que nos indica que el modelo mejora el porcentaje de asertividad.
- ***loss*** : Mide el error cometido en las imágenes de entrenamiento , el valor debe ir bajando . Un valor de 0.3 hacia abajo se considera bien , un valor de 0.1 es excelente y valores cercanos a 0.01 es casi perfecto.
- ***Val_accuracy*** :Es uno de los valores más importante, ya que indica el acierto del modelo con imágenes nuevas , no con las de entrenamiento.
- ***Val_loss*** : Es el otro valor importante , este indicador muestra el error en imágenes que el modelo nunca ha visto . Si este valor aumenta y ***loss*** baja , se está en presencia de Overfitting.

En el ejemplo descrito anteriormente se utilizó el dataset de Kangle (www.kangle.com), el cual es un inmenso repositorio de datos de uso público propiedad de Google. Es en esa página en donde se pueden bajar miles de archivos para entrenar modelos de IA.



Fuente: Elaboración Propia , basada en página web

Figura 3-16 Entrenamiento del modelo

3.2.2.8 Predicción

En este último tramo, como se muestra en la figura 3-17 se detalla el script de predicción, en este caso los bloques de programación están destinadas al uso del modelo ya entrenado, con la finalidad de reconocer una nueva imagen (perro o gato) .

Como ya se vio en scripts anteriores, se inicia con la carga de librerías , en este caso son :

- *numpy* : Librería matemática , con esta librería es posible procesar las imágenes para transformarlas en matrices de números .
- *from tensorflow.keras.preprocessing import image* : Herramienta que sirve para la carga y preparación de imágenes, antes de su uso en un modelo.

- ***load_model*** : Llamado al modelo entrenado ("modelo_perros_gatos.keras").

```
import numpy as np
from tensorflow.keras.preprocessing import image

# Cargar modelo
model = tf.keras.models.load_model("modelo_perros_gatos.keras")
```

Elaboración Propia , basada en página web

Figura 3-17 Preparación y carga de modelo

En el siguiente block de programación en la figura 3-18, es posible observar pasos ya descritos anteriormente.

- ***img_path = r"C:\Users\xino_\Downloads\curso python\gato1.jpeg"***: Guarda la ruta de la imagen que se desea analizar.
- ***img = image.load_img(img_path, target_size=(160, 160))*** : Carga la imagen y luego la redimensiona a 160x160 pixeles.
- ***img_array = image.img_to_array(img)*** : Convierte la imagen PIL de 160x160 (Python imaging Library) a un array NumPy (matriz numérica).
- ***img_array = img_array / 255.0*** : Normalización de los pixeles que van desde 0 a 255, para transformarlos en valores que van desde 0 a 1, con esto la red neuronal no se “sature” con números tan grandes.
- ***img_array = np.expand_dims(img_array, axis=0)*** : Es un “truco” que se utiliza, ya que Keras está diseñado para procesar lotes (batches) de imágenes y no solo una. Con esto se logra que el sistema funcione con solo una imagen.
- ***prediction = model.predict(img_array)*** : Presenta la imagen al modelo y se obtiene la predicción.
- ***print("Porcentaje de predicción :", prediction)*** : Imprime el porcentaje de predicción en pantalla .

- **if prediction[0][0] > 0.5: print("Es un PERRO"):** Accede al valor de la predicción, si el valor es mayor a 0.5, imprimirá la frase que se encuentra entre paréntesis “Es un PERRO”.
- **else: print("Es un GATO ") :** Si el valor es menor o igual a 0.5 , imprimirá “Es un GATO”.

```
# Ruta de la imagen a probar
img_path = r"C:\Users\xino_\Downloads\curso python\gato.jpeg"
# Cargar imagen
img = image.load_img(img_path, target_size=(160, 160))
img_array = image.img_to_array(img)

# Normalizar
img_array = img_array / 255.0

# Agregar dimensión
img_array = np.expand_dims(img_array, axis=0)

# Predicción
prediction = model.predict(img_array)

print("Porcentaje de predicción :", prediction)

if prediction[0][0] > 0.5:
    print("Es un PERRO 🐶")
else:
    print("Es un GATO 🐱")
```

Fuente: Elaboración Propia , basada en página web

Figura 3-18 Procesamiento de imagen a analizar

3.2.2.9 Resultados

Luego del entrenamiento y la validación del sistema, es posible realizar pruebas con imágenes, tal como se describe en el punto anterior (ruta de imagen), en donde se debe añadir la ruta del archivo en cuestión. En la figura 3-19 se muestran un set de imágenes con las cuales se realiza el testeó, y en la figura 3-20 se muestra el resultado de la predicción con cada una de las imágenes descritas, con el porcentaje de predicción inserto en ellas. Se debe recordar (como aparece en la figura 3-18) que si el porcentaje de predicción es mayor a 0.5, se imprimirá “Es un PERRO”, en caso contrario, si el valor entregado por el sistema es menor o igual a 0.5, el sistema deberá mostrar en pantalla “Es un GATO”.

Como consejo al lector, ante este ejercicio, se recomienda realizar el entrenamiento en la página de Google mencionada en ocasiones anteriores (<https://colab.research.google.com/>), esto aplica cuando no se cuenta con un equipo muy potente en capacidad de memoria o de CPU, o simplemente el usuario no desea saturar el computador. Posteriormente es posible realizar el llamado al modelo entrenado y hacer las pruebas descritas en el párrafo anterior.



Fuente: Elaboración Propia, basada en página web

Figura 3-19 Imágenes a testear

La figura 3-20 nos muestra el resultado de las predicciones realizadas.

```
... 1/1 ██████████ 1s 1s/step
Porcentaje de predicción : [[0.00374571]]
Es un GATO 🐱

1/1 ██████████ 1s 982ms/step
Porcentaje de predicción : [[0.00054709]]
Es un GATO 🐱

1/1 ██████████ 1s 925ms/step
Porcentaje de predicción : [[0.99791026]]
Es un PERRO 🐶

1/1 ██████████ 1s 989ms/step
Porcentaje de predicción : [[0.9954689]]
Es un PERRO 🐶

1/1 ██████████ 1s 1s/step
Porcentaje de predicción : [[0.9995098]]
Es un PERRO 🐶

1/1 ██████████ 1s 1s/step
Porcentaje de predicción : [[8.213484e-05]]
Es un GATO 🐱
```

Fuente: Elaboración Propia, basada en página web

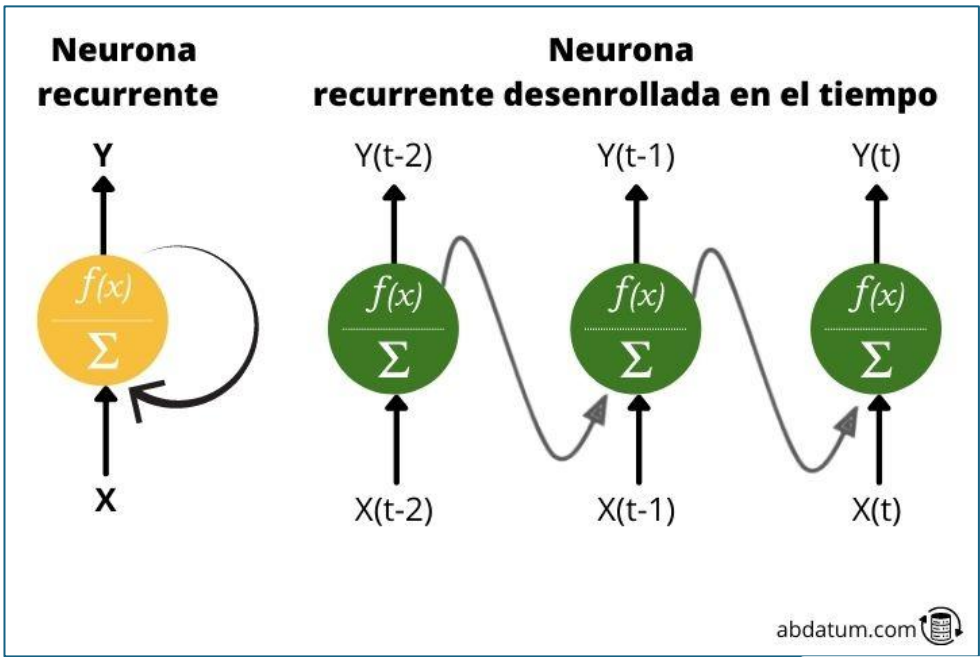
Figura 3-20 Resultados de la predicción

3.3. RED NEURONAL RECURRENT (RNN)

Una red neuronal recurrente (RNN por sus siglas en inglés), es un tipo de red neuronal utilizada habitualmente para el procesamiento de secuencia de datos o series temporales . Se distingue esencialmente de otros tipos de redes por su conexión en forma cíclica, lo cual les permite tener “memoria” de las entradas anteriores para ser utilizada en la salida actuales. Con esta característica es posible procesar textos, audios, videos, entre otros. Está compuesta básicamente por: Capa de entrada (recibe los datos), capa oculta (procesa la información y mantiene el estado anterior) y la capa de salida (entrega datos procesados).

3.3.1. Modelación de una red neuronal recurrente

Las redes recurrentes se destacan por la “memoria” que poseen, una de la característica esencial al momento de procesar y entender datos secuenciales en donde el orden de los datos es primordial. En contraste con otras redes neuronales en donde los datos tienen una sola dirección (entrada y salida) las redes recurrentes retienen información y poseen una retroalimentación, en la figura 3-19 se muestra un ejemplo sencillo de una red neuronal recurrente, el lado izquierdo (figura en amarillo) se puede observar la conexión recurrente (bucle) de la neurona, indicando así la retroalimentación del sistema.



Fuente: <https://abdatum.com/tecnologia/redes-neuronales-recurrentes>

Figura 3-21 Arquitectura de una red Recurrente

La otra parte de la imagen muestra las retroalimentaciones en función del tiempo, en este caso **la misma neurona** (o capa) está siendo alimentada por datos actuales (x) y retroalimentada con datos anteriores (y).

3.3.1.1 Estado oculto y Salida

Para entender de mejor manera, se debe aclarar que la salida y la retroalimentación (hidden state) **no son lo mismo**, pero están directamente ligados.

- El estado oculto (**$h_{(t)}$**) es la “memoria” interna que almacena información de entradas pasadas, se actualiza en cada paso de tiempo (**t**).
- La salida (**$y_{(t)}$**) es el resultado final que la RNN produce en cada paso de tiempo (**t**), se calcula a partir del estado oculto.

Matemáticamente se expresa de la siguiente forma:

- **Estado oculto (Hidden State) :**

En la ecuación 3-1 se observa el algoritmo de RNN.

$$h_{(t)} = g \left(w_{(h)} h_{(t-1)} + w_{(t)} x_{(t)} + b_{(h)} \right)$$

Ecuación 3-1 Algoritmo de RNN

Donde :

- $h_{(t)}$** : Estado oculto en tiempo actual (t)
- $g(t)$** : Función de activación (ReLU o Tanh)
- $w_{(h)}$** : Matriz de pesos estado anterior
- $h_{(t-1)}$** : Estado oculto en tiempo anterior
- $w_{(t)}$** : Matriz de pesos que cambia la entrada actual
- $x_{(t)}$** : Entrada en el tiempo actual

$b_{(h)}$: Sesgo

De lo anterior $w_{(h)}$ y $h_{(t-1)}$ son los términos claves para que la RNN tenga “memoria” de los tiempos anteriores.

- **Salida :**

En la ecuación 3-2 se muestra la salida $y_{(t)}$

$$y_{(t)} = f(w_{(y)} h_{(t)} + b_{(y)})$$

Ecuación 3-2 Cálculo de la salida $y_{(t)}$

Donde :

$y_{(t)}$: Salida generada por la RNN en el tiempo (t)

$f(t)$: Función de activación Softmax (habitualmente usada en clasificación)

$w_{(y)}$: Matriz de peso para la salida

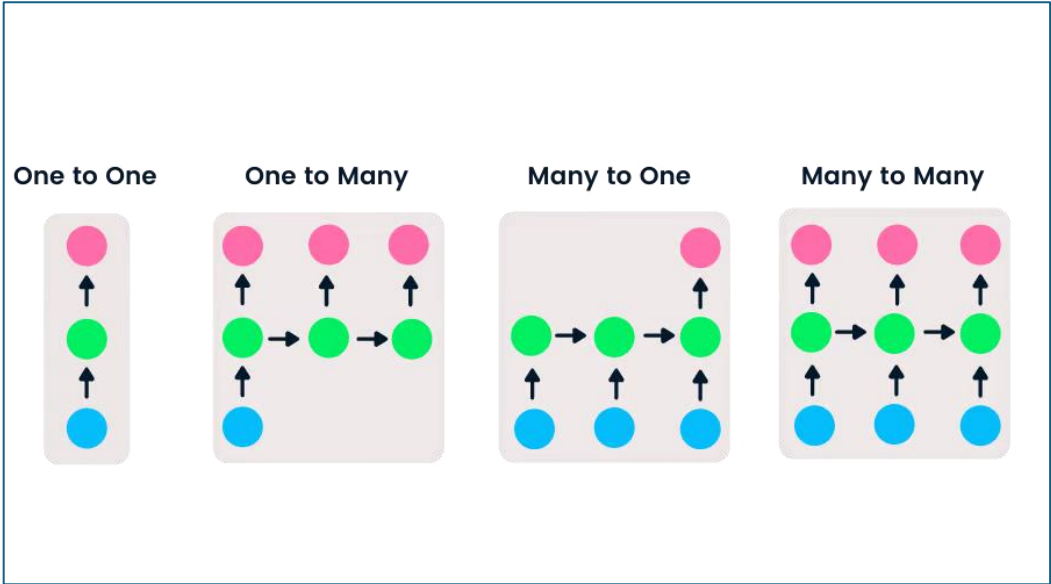
$h_{(t)}$: Estado oculto actual

$b_{(y)}$: Sesgo

Existen diferentes arquitecturas de redes recurrentes, estas varían según su uso y arquitectura, en la siguiente figura 3-20 se muestran ejemplos conceptuales de los tipos de arquitecturas.

- One to one: Una entrada una salida, utilizado en modelación de sistemas dinámicos simples como clasificación de imágenes.
- One to many: Una entrada y múltiples salidas, empleado mayormente en la generación de texto a partir de una imagen.
- Many to one: Múltiples entradas y una salida, dedicada al análisis de los sentimientos insertos en un texto.

- Many to many: Múltiples entradas y múltiples salidas, usada en traducción automática.



Fuente : <https://www.datacamp.com/es/tutorial/tutorial-for-recurrent-neural-network>

Figura 3-22 Tipos de Redes Neuronales Recurrentes

3.3.2. Ejemplo de modelación de una red neuronal recurrente (RNN)

En el presente ejemplo se desarrollará un ejercicio determinado “Predicción de carga de baterías mediante energía solar ” con una red LSTM (Sección 1.8.3.1), en el cual serán usados datos de entrada con un generador de datos simulados, para posteriormente ingresarlos en la red neuronal.

3.3.2.1 Carga de librería

Como muestra la figura 3-21, y al igual que en los ejemplos presentados anteriormente, al inicio del ejercicio es necesario cargar las librerías que serán usadas en el proyecto. Tal como se observa en figura 3-21, la única novedad en comparación con los ejemplos anteriores es la carga de la librería LSTM de Tensorflow en la línea de comando, con la cual es posible mejorar la “memoria” y error en el sistema.

```
# 1 Importar librerías
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
```

[32] ✓ 0.0s

Fuente : Elaboración propia , basado en página de internet

Figura 3-23 Importación de Librerías

3.3.2.2 Generación de datos simulados

En la figura 3-22 se muestra la generación de datos ficticios para generación de energía y consumo de energía , los cuales serán usados para este ejemplo .En este caso se crearán datos aleatorios de: generación solar, consumo de energía y carga de baterías será la diferencia acumulativa entre la generación y consumo, esto lo hace con las siguientes líneas de comando:

- ***generate_energy_data(n_samples=500) :***

Genera 500 datos simulados de generación de energía, consumo y carga de una batería para el problema.

- ***np.random.seed(42) :***

Esta línea de comando establece que son los mismos datos en cada ejecución.

- ***solar_generation = np.sin(np.linspace(0, 50, n_samples)) + np.random.normal(0, 0.1, n_samples) :***

En el ítem de “generación de energía” utiliza la función seno para simular la variabilidad de energía solar en el tiempo, con ***np.linspace*** genera ***n_samples*** de valores distanciados de forma uniforme entre 0 y 50. El siguiente tramo de la línea de comando ***np.random.normal(0, 0.1, n_samples)*** genera valores aleatorios, extrayéndolo de una distribución normal (gaussiana) con valor medio de 0 y una desviación estándar de 0.1, para sumarlos a la onda sinusoidal, con esto logra simular las fluctuaciones (ruido) en la generación de energía solar.

- ***energy_consumption = np.cos(np.linspace(0, 50, n_samples)) + np.random.normal(0, 0.1, n_samples) :***

En línea de consumo de energía, es similar a la anterior, con la salvedad de utilizar el coseno en vez del seno , con las mismas características de ruido aleatorio.

- ***battery_charge = np.cumsum(solar_generation - energy_consumption) :***

Calcula la diferencia entre la generación de energía y el consumo. Con ***np.cumsum*** calcula la suma acumulada de las diferencias, simulando así el aumento en carga de una batería cuando la generación es mayor al consumo, o la descarga de la batería cuando el consumo es mayor a la carga.

- ***return solar_generation, energy_consumption, battery_charge***

Devuelve los arrays generación solar, consumo de energía y carga de batería ya simulados.

```
# 2 Generación de datos simulados
def generate_energy_data(n_samples=500):
    np.random.seed(42)
    solar_generation = np.sin(np.linspace(0, 50, n_samples)) + np.random.normal(0, 0.1, n_samples)
    energy_consumption = np.cos(np.linspace(0, 50, n_samples)) + np.random.normal(0, 0.1, n_samples)
    battery_charge = np.cumsum(solar_generation - energy_consumption)
    return solar_generation, energy_consumption, battery_charge
```

Fuente : Elaboración propia , basado en página de internet

Figura 3-24 Generación de datos simulados

3.3.2.3 Adecuación de datos mediante el método de “ventanas deslizantes”

En el siguiente paso, como se muestra en la figura 3-23, se describe la función que toma los datos creados anteriormente (generación solar, consumo y carga de baterías) y los adecúa para el entrenamiento de la red RNN con LSTM, con el uso de la técnica “ventanas deslizantes”.

La técnica de “ventanas deslizantes” es usada para la predicción , esto se hace mediante datos históricos. Esta técnica mueve la ventana que tiene un tamaño fijo a través de una estructura de datos para luego predecir el dato siguiente ,con la finalidad de reducir la complejidad de analizar secuencias muy largas.

Ejemplo :

En una secuencia de números como : [1, 2 , 3 , 4 , 5 , 6]

Y con una ventana de : 3

Entonces la entrada sería [1 , 2, 3] y lo esperado (predicción) sería 4

En [2, 3, 4] la salida esperada sería 5

En [3, 4, 5] la salida esperada sería 6, y así sucesivamente con letras o palabras.

Como se muestra en la primera línea de programación la figura 3-23, toma los datos creados anteriormente y los adecúa para el entrenamiento de la red RNN.

Posteriormente se crean listas X e Y para ser llenadas con los datos calculados con la técnica de ventanas deslizante, donde X son los valores de **generación solar** y **consumo**, por otro lado, en Y el queda el resultado denominado **carga de batería**. Finalmente, con *return np.array(X), np.array(y)* convierte las listas en arrays para el entrenamiento.

```
# 3 Crear dataset con "ventanas deslizantes"
def create_dataset(solar, consumption, charge, time_steps=10):
    X, y = [], []
    for i in range(len(solar) - time_steps):
        X.append(np.column_stack((solar[i:i+time_steps], consumption[i:i+time_steps])))
        y.append(charge[i+time_steps])
    return np.array(X), np.array(y)
```

[35] ✓ 0.0s

Fuente : Elaboración propia , basado en página de internet

Figura 3-25 Uso de técnica “Ventana deslizante”

3.3.2.4 Preparación de datos

En este caso , como muestra la figura 3-24, las líneas de comando indican que los datos generados anteriormente son transformados para el entrenamiento, con un formato adecuado para una red neuronal.

Con *X , y = create_dataset(solar_gen, energy_con, battery_ch)* convierte los datos en secuencias de entrada y etiquetas, posteriormente con *reshape* se redimensionan el array X para que sean compatibles con entrada en una red **LSTM**

```
# 4 Generar datos simulados para entrenamiento
solar_gen, energy_con, battery_ch = generate_energy_data()
X, y = create_dataset(solar_gen, energy_con, battery_ch)
X = X.reshape((X.shape[0], X.shape[1], 2)) # Redimensionar para LSTM
] ✓ 0.0s
```

Fuente : Elaboración propia , basado en página de internet

Figura 3-26 Generación de datos de entrenamiento

3.3.2.5 Definición del tipo de modelo de red neuronal

El paso siguiente en el ejercicio es la elección del tipo de red a usar, en este caso se usará LSTM. Como muestra la figura 2-25, la primera línea de programación indica el modelo secuencial, el cual permite construir capa por capa, la primera capa con 100 neuronas LSTM, estas neuronas tienen memoria de datos. Como función de activación se usa ReLU, **return_sequences=True** devolverá la secuencia completa de salidas, **input_shape** indica la forma de entrada de los datos y finalmente **Dense(1)** que establece que la es una capa densa completamente conectada con 1 unidad, que equivale a la salida.

```
# 5 Definición del modelo LSTM
model = Sequential([
    LSTM(100, activation='relu', return_sequences=True, input_shape=(X.shape[1], X.shape[2])),
    Dense(1)
])
```

Fuente: Elaboración propia, basada en página web

Figura 3-27 Definición de modelo LSTM

3.3.2.6 Compilación y entrenamiento

En la figura 3-26 es posible observar las líneas de programación del sistema que compila y entrena el modelo. Al igual que en los ejemplos anteriores, se utiliza el optimizador ADAM (Adaptative Moment Estimation) el cual adapta eficientemente los pesos, la función de pérdida usada es MSE (Error cuadrático medio). Posteriormente, en la siguiente línea de programación se entrena el modelo con los datos X e Y, **history** almacena el historial del entrenamiento, con el objeto **epochs** en 50, indica que entrenará 50 épocas , con **verbose = 0** se ocultan los mensajes de entrenamiento en pantalla y con **validation_split=0.2** se reserva el 20% de los datos para validación.

```
#6 Compilar y entrenar modelo
model.compile(optimizer='adam', loss='mse')
history = model.fit(X, y, epochs=50, verbose=0, validation_split=0.2)
✓ 8.0s
```

Fuente : Elaboración propia , basado en página de internet

Figura 3-28 Compilación y entrenamiento del modelo

3.3.2.7 Predicción

En la línea de código que se muestra en la figura 3-27, es usada para realizar predicciones del sistema, con los datos del array de entrada X (que contiene las secuencias de generación y consumo), lo pasa a través de la RNN para generar la predicción, donde la variable ***predictions*** almacena el resultado (nivel de carga de batería)

```
# 7 Predicciones
predictions = model.predict(X)
✓ 0.4s
16/16 ————— 0s 12ms/step
```

Fuente : Elaboración propia , basado en página de internet

Figura 3-29 Predicción del modelo

3.3.2.8 Cálculo de error

La figura 3-28 muestra el cálculo del error cuadrático medio (MSE), en donde:

- **np.mean**

Este comando está destinado a calcular el promedio de los errores.

- **predictions.flatten() - y) ** 2**

Convierte el array de predicciones en un array unidimensional, luego calcula la diferencia entre la predicción (-y) y el valor real para posteriormente elevarlo al cuadrado (**2).

- **print(f"Error cuadrático medio (MSE): {mse}")**

Imprime el valor que se observa en la imagen, en este caso 0.790...

```
| #8 Calculo de error
mse = np.mean((predictions.flatten() - y) ** 2)
print(f"Error cuadrático medio (MSE): {mse}")
✓ 0.0s
Error cuadrático medio (MSE): 0.790194382755043
```

Fuente : Elaboración propia , basado en página de internet

Figura 3-30 Cálculo de MSE del modelo

3.3.2.9 Visualización de predicción de carga

En la figura 3-29, se observan los comandos para generar las gráficas de carga de batería y la predicción de carga, se tiene que recordar que la forma sinusoidal es meramente referencial a modo de ejercicio. En ambos casos (#9 y #10) las líneas de comando indican el tamaño de la gráfica, los títulos y otros elementos gráficos modificables.

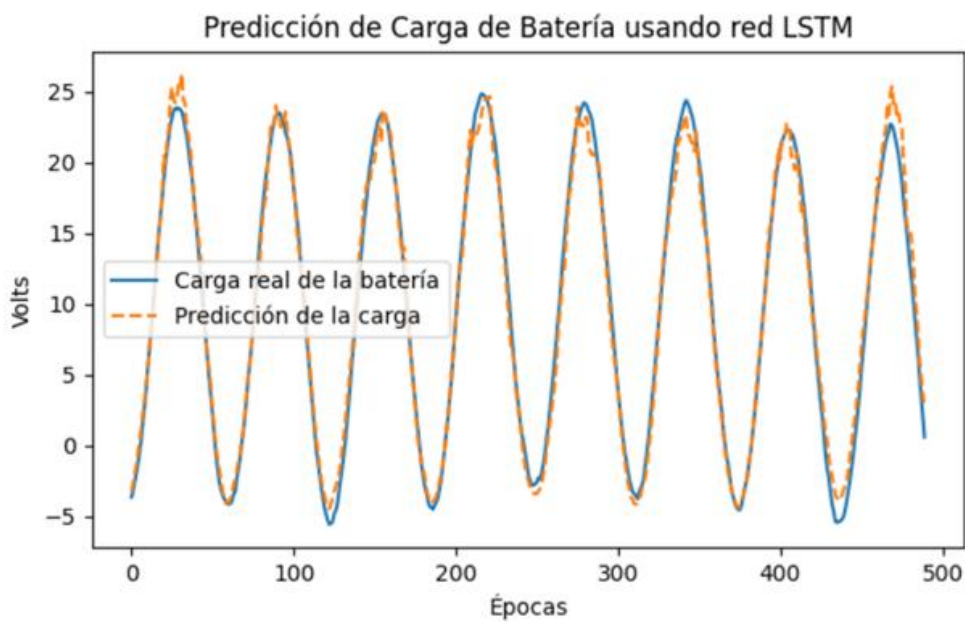
```
# 9 Visualización de resultados en pantalla
plt.figure(figsize=(7, 4))
plt.plot(battery_ch[10:], label="Carga real de la batería")
plt.plot(predictions, label="Predicción de la carga", linestyle='dashed')
plt.xlabel('Épocas')
plt.ylabel('Volts')
plt.legend()
plt.title("Predicción de Carga de Batería usando red LSTM")
plt.show()

# 10 Visualización de la evolución del error
plt.figure(figsize=(7, 3))
plt.plot(history.history['loss'], label='Error de entrenamiento')
plt.plot(history.history['val_loss'], label='Error de validación')
plt.legend()
plt.title("Evolución del Error durante el Entrenamiento")
plt.show()
```

Fuente: Elaboración propia, basada en página web

Figura 3-31 Líneas de comando para generar gráficas

La figura 3-30 se muestra la gráfica de predicción de carga versus el número de épocas.

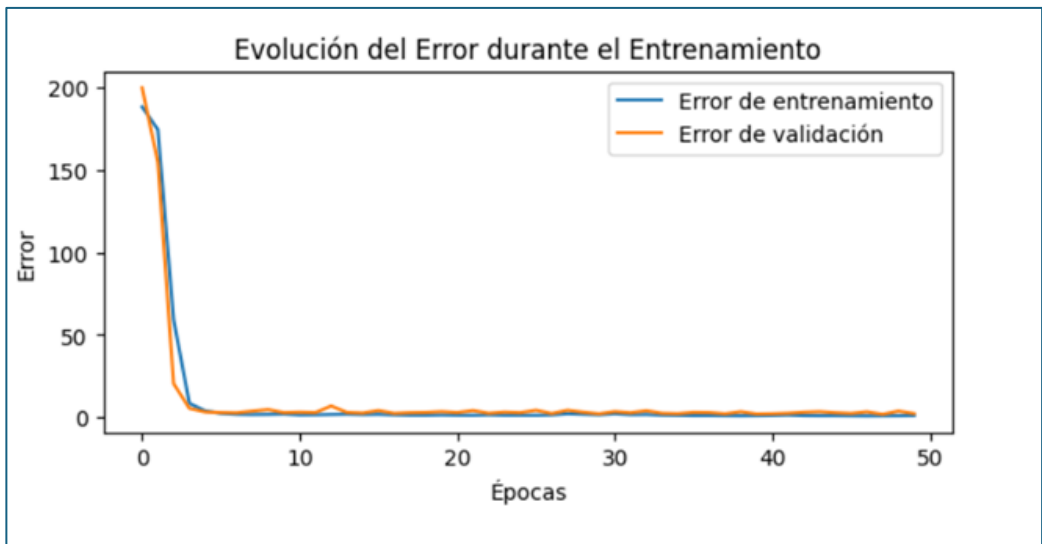


Fuente: Elaboración propia, basada en página web

Figura 3-32 Gráfica de predicción de carga

3.3.2.10 Gráfica de error

En la figura 3-31 se observa la evolución del error a medida que el sistema va entrenando, es posible apreciar que después de las 10 o 20 epochs, el sistema no varía mayormente y se asienta en un valor relativamente estable.



Fuente: Elaboración propia, basada en página web

Figura 3-33 Gráfica del error

CONCLUSIÓN

En el desarrollo de este trabajo de título, fue posible ampliar el conocimiento sobre las Redes Neuronales, tanto las Redes Neuronales simples, como las Redes Convolucionales (CNN), así como las Recurrentes (RNN). Lo anterior sienta las bases para un desarrollo de otras áreas de las redes neuronales.

Gracias a este trabajo fue posible entender y describir cómo funciona básicamente una neurona artificial y su inspiración, la neurona biológica. Sumado a lo anterior fue posible entender un poco las matemáticas inmersas en una red neuronal, así como comprender lo que es el Feedforward, el Backpropagation (de forma básica), las funciones de activación (ReLU, Sigmoide), la importancia del Descenso por Gradiente en las Redes Neuronales. Adicionalmente se pudieron desarrollar ejemplos con redes un poco más complejas, como son las Redes Neuronales Convolucionales y las Redes Neuronales Recurrentes. Adicionalmente fue posible entregar información práctica para lograr mayor eficiencia en el modelo, esto mediante técnicas de optimización básicas y simples, tal como el manejo de la tasa de aprendizaje o (un poco más complejo) el uso y aplicación del Dropout, el Early Stopping y el Data Augmentation. Todo lo anterior con la finalidad de evitar la aparición del Overfitting y el Underfitting.

Es necesario mencionar que, debido a los alcances del presente trabajo, no fue posible abarcar con mayor profundidad los temas aquí descritos, así como tampoco la posibilidad de un mayor número de ejemplos.

Es de esperar que el presente trabajo sirva al lector como base introductoria a un campo tan amplio y en constante desarrollo como son las redes neuronales artificiales.

BIBLIOGRAFÍA

KHAN ACADEMY. El sistema nervioso : [En Línea]

<<https://es.khanacademy.org/science/2-secundaria-cyt/xe87a282d8d8e1339:funcion-de-relacion/xe87a282d8d8e1339:sistema-nervioso-humano/v/crash-course-biology-125>> [consulta: 2024]

AMAZON WEB SERVICE. ¿Qué es una red neuronal?[En Línea]

<<https://aws.amazon.com/es/what-is/neural-network/>> [consulta: 2024]

IBM.COM. Neural networks [En Línea]

< <https://www.ibm.com/es-es/topics/neural-networks> > [consulta: 2024]

DATASCIENTEST.COM. Perceptrón: ¿qué es y para qué sirve? [En Línea]

<<https://datascientest.com/es/perceptron-que-es-y-para-que-sirve>> [consulta: 2024]

YADAV , Amit . Backpropagation vs. Gradient Descent [En Línea]

<<https://medium.com/biased-algorithms/backpropagation-vs-gradient-descent-19e3f55878a6>> [consulta: 2024]

INTERACTIVECHAOS.COM. Backpropagation [En Línea]

<<https://interactivechaos.com/es/manual/tutorial-de-machine-learning/backpropagation>> [consulta: 2024]

RINGATECH. (10 noviembre 2020). Cómo funcionan las redes neuronales - Inteligencia Artificial [archivo de video] . Youtube
<<https://www.youtube.com/watch?v=CU24iC3grq8&t=364s>> [consulta: 2024]

RINGATECH. (7 de julio de 2021). Tu primera red neuronal en Python y Tensorflow [archivo de video] . Youtube
<<https://www.youtube.com/watch?v=CU24iC3grq8&t=364s>> [consulta: 2024]

VASQUEZ, Irving (22 de marzo de 2024) . Backpropagation (Retropropagación)
[archivo de video] Youtube
<<https://www.youtube.com/watch?v=bJFM9rvpwaY>>[consulta: 2024]

IBM.COM. Neural networks [En Línea]

< <https://www.ibm.com/es-es/topics/data-augmentation> > [consulta: 2025]

IBM.COM. Neural networks [En Línea]

< <https://www.ibm.com/es-es/topics/overfitting-vs-underfitting>> [consulta:
2025]

APRENDEMACHINELEARNING.COM [En Línea]

<<https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/>>[consulta: 2025]

CODIFICANDOBITS.COM [En Línea] <<https://codificandobits.com/blog/underfitting-y-overfitting/>> [consulta: 2025]

DEVELOPERS.GOOGLE.COM [En Línea] <https://developers.google.com/machine-learning/crash-course/overfitting/overfitting?hl=es-419> > [consulta: 2025]

GARCIA-FERREIRA.ES [En Línea] < <https://www.garcia-ferreira.es/conoce-todo-sobre-el-dropout/>> [consulta: 2025]

TOWARDSDATASCIENCE.COM [En Línea] < <https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9/>> [consulta:
2025]

AWS.AMAZON.COM [En Línea] < <https://aws.amazon.com/es/what-is/data-augmentation/>> [consulta: 2025]

ANEXOS

Anexo 1 : Modelo de Red Neuronal Lineal (simple)



```

1  import tensorflow as tf
2  import numpy as np
3
4  celsius = np.array([-35,-20,-15,0,5,10,15,20,35,], dtype= float) #entradas
5  fahrenheit = np.array([-31,-4,5,32,41,50,59,68,95,], dtype=float) #salidas
6
7  capa = tf.keras.layers.Dense(units=1, input_shape=[1])
8  modelo =tf.keras.Sequential([capa])
9
10 modelo.compile(
11     optimizer=tf.keras.optimizers.Adam(0.1),
12     loss ="mean_squared_error"
13 )
14
15 print("comenzando entrenamiento")
16
17 historial = modelo.fit(celsius, fahrenheit , epochs=1000, verbose=False)
18
19 print("Modelo entrenado")
20
21 import matplotlib.pyplot as plt
22 plt.xlabel("Número de epochs")
23 plt.ylabel("Magnitud de pérdida")
24 plt.plot(historial.history["loss"])
25
26 print("poner a prueba ")
27 resultado = modelo.predict(np.array([50, 25,33]).reshape(3,1))
28 print("El resultado es : " + str(resultado) + " Fahrenheit")

```

Anexo 2 : Modelo Red Neuronal Convolutacional (CNN)

```

1 import tensorflow as tf
2 from tensorflow.keras import layers, models
3 from tensorflow.keras.applications import MobileNetV2
4 from tensorflow.keras.preprocessing.image import ImageDataGenerator
5 # Parámetros
6 IMG_SIZE = 160
7 BATCH_SIZE = 32
8
9 # Data augmentation
10 train_datagen = ImageDataGenerator(
11     rescale=1./255,
12     rotation_range=20,
13     zoom_range=0.2,
14     horizontal_flip=True
15 )
16
17 val_datagen = ImageDataGenerator(rescale=1./255)
18
19 train_generator = train_datagen.flow_from_directory(
20     r'C:\Users\xino\Downloads\curso python\dataset\train',
21     target_size=(IMG_SIZE, IMG_SIZE),
22     batch_size=BATCH_SIZE,
23     class_mode='binary'
24 )
25
26 val_generator = val_datagen.flow_from_directory(
27     'dataset/validation',
28     target_size=(IMG_SIZE, IMG_SIZE),
29     batch_size=BATCH_SIZE,
30     class_mode='binary'

```

```

1 class_mode='binary'
2 )
3
4 val_generator = val_datagen.flow_from_directory(
5     'dataset/validation',
6     target_size=(IMG_SIZE, IMG_SIZE),
7     batch_size=BATCH_SIZE,
8     class_mode='binary'
9 )
10 # Modelo base preentrenado
11 base_model = MobileNetV2(
12     input_shape=(IMG_SIZE, IMG_SIZE, 3),
13     include_top=False,
14     weights='imagenet'
15 )
16
17 base_model.trainable = False # Congelamos pesos
18 # Construcción del modelo final
19 model = models.Sequential([
20     base_model,
21     layers.GlobalAveragePooling2D(),
22     layers.Dropout(0.3),
23     layers.Dense(1, activation='sigmoid')
24 ])
25 # Compilación
26 model.compile(
27     optimizer='adam',
28     loss='binary_crossentropy',
29     metrics=['accuracy']
30 )
31 # Entrenamiento

```



```

1  # Entrenamiento
2  history = model.fit(
3      train_generator,
4      validation_data=val_generator,
5      epochs=10
6  )
7
8  # Guardar modelo
9  model.save("modelo_perros_gatos.keras")
10 # Entrenamiento
11 history = model.fit(
12     train_generator,
13     validation_data=val_generator,
14     epochs=10
15 )
16
17
18 import numpy as np
19 from tensorflow.keras.preprocessing import image
20
21 # Cargar modelo
22 model = tf.keras.models.load_model("modelo_perros_gatos.keras")
23
24 # Ruta de la imagen a probar
25 img_path = r"C:\Users\xino\Downloads\curso python\gato.jpeg"
26 # Cargar imagen
27 img = image.load_img(img_path, target_size=(160, 160))
28 img_array = image.img_to_array(img)

```



```

1  # Normalizar
2  img_array = img_array / 255.0
3
4  # Agregar dimensión
5  img_array = np.expand_dims(img_array, axis=0)
6
7  # Predicción
8  prediction = model.predict(img_array)
9
10 print("Porcentaje de predicción :", prediction)
11
12 if prediction[0][0] > 0.5:
13     print("Es un PERRO 🐕")
14 else:
15     print("Es un GATO 🐈")

```

Anexo 3 : Modelo de Red Neuronal Recurrente (RNN)

```

1  # 1 Importar librerías
2  import numpy as np
3  import tensorflow as tf
4  from tensorflow.keras.models import Sequential
5  from tensorflow.keras.layers import LSTM, Dense
6  import matplotlib.pyplot as plt
7
8  # 2 Generación de datos simulados
9  def generate_energy_data(n_samples=500):
10     np.random.seed(42)
11     solar_generation = np.sin(np.linspace(0, 50, n_samples)) + np.random.normal(0, 0.1, n_samples)
12     energy_consumption = np.cos(np.linspace(0, 50, n_samples)) + np.random.normal(0, 0.1, n_samples)
13     battery_charge = np.cumsum(solar_generation - energy_consumption)
14     return solar_generation, energy_consumption, battery_charge
15
16
17 # 3 Crear dataset con "ventanas deslizantes"
18 def create_dataset(solar, consumption, charge, time_steps=10):
19     X, y = [], []
20     for i in range(len(solar) - time_steps):
21         X.append(np.column_stack((solar[i:i+time_steps], consumption[i:i+time_steps])))
22         y.append(charge[i+time_steps])
23     return np.array(X), np.array(y)
24
25 # 4 Generar datos simulados para entrenamiento
26 solar_gen, energy_con, battery_ch = generate_energy_data()
27 X, y = create_dataset(solar_gen, energy_con, battery_ch)
28 X = X.reshape((X.shape[0], X.shape[1], 2)) # Redimensionar para LSTM
29
30 # 5 Definición del modelo LSTM
31 model = Sequential([
32     LSTM(100, activation='relu', input_shape=(X.shape[1], X.shape[2])),
33     Dense(1) # Predicción de carga de batería
34 ])
35
36 #6 Compilar y entrenar modelo
37 model.compile(optimizer='adam', loss='mse')
38 history = model.fit(X, y, epochs=50, verbose=0, validation_split=0.2)
39
40 # 7 Predicciones
41 predictions = model.predict(X)
42
43 #8 Error cuadrático medio
44 mse = np.mean((predictions.flatten() - y) ** 2)
45 print(f"Error cuadrático medio (MSE): {mse}")
46
47
48 # 9 Visualización de resultados en pantalla
49 plt.figure(figsize=(7, 4))
50 plt.plot(battery_ch[10:], label="Carga real de la batería")
51 plt.plot(predictions, label="Predicción de la carga", linestyle='dashed')
52 plt.legend()
53 plt.title("Predicción de Carga de Batería usando red LSTM")
54 plt.show()
55
56 # 10 Visualización de la evolución del error
57 plt.figure(figsize=(7, 3))
58 plt.plot(history.history['loss'], label='Error de entrenamiento')
59 plt.plot(history.history['val_loss'], label='Error de validación')
60 plt.legend()
61 plt.title("Evolución del Error durante el Entrenamiento")
62 plt.show()
63

```