CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS
AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

SECCIÓN DE ELECTRÓNICA DEL ESTADO SÓLIDO

"Aprendizaje en Máquina Aplicado a la Detección de Fallas Mecánicas"

TESIS

Que presenta

Ing. Luis Elias Salgado Solano

Para obtener el grado de

MAESTRO EN CIENCIAS
EN LA ESPECIALIDAD DE INGENIERÍA ELÉCTRICA

Directores de tesis: Dr. Felipe Gómez Castañeda
Dr. José Antonio Moreno Cadenas

Ciudad de México, México DICIEMBRE, 2022

Agradecimientos

A mis familiares.

Este trabajo fue desarrollado con mucho esfuerzo y sacrificio no solo míos sino también de mis seres queridos. Quiero agradecer a mis padres Teófila Solano Sánchez y Gonzalo Salgado Sánchez por brindarme su apoyo en todo momento para continuar desarrollándome profesionalmente y también quiero dedicar un especial agradecimiento a mi tía, Olga Solano Sánchez quien me cuido de pequeño y me apoyo a cumplir mis metas con el afán que lo hace una madre.

A mis asesores.

Para llevar a cabo este trabajo tuve la fortuna de tener como mis asesores al Doctor Felipe Gómez Castañeda, al Doctor José Antonio Moreno Cadenas y al Maestro en Ciencias Luis Martin Flores Nava, cuyas aportaciones fueron de gran importancia, y a quienes les agradezco el trato humano y profesional recibido.

A mis sinodales.

Dedico un especial agradecimiento a mis sinodales, el Doctor Mario Alfredo Reyes Barranca y el Doctor Álvaro Anzueto Ríos, por su disposición a contribuir con sus observaciones de una manera cordial y profesional.

Al CONACYT.

Le agradezco al CONACYT, una noble institución, por el financiamiento recibido.

Al CINVESTAV.

Le agradezco al CINVESTAV, una institución de excelencia, por la educación recibida.

Índice general

0.1. RESUMEN

En este trabajo se ha realizado un estudio sobre la aplicación de la inteligencia artificial para la extracción y reconocimiento de información de señales de vibración mecánica. Las señales de vibración mecánica han sido estudiadas tradicionalmente mediante modelados matemáticos y tratamientos de tipo Gaussiano en diferentes disciplinas como el análisis sismológico y el monitoreo de salud mecánica. En años recientes nuevas tendencias de computación han mejorado la eficiencia y adaptabilidad de los algoritmos de inteligencia artificial, los cuales pueden ser implementados para ayudar a resolver muchas de las necesidades humanas. En el procesamiento digital de señales y reconocimiento de patrones mediante inteligencia artificial, la información es tratada desde un enfoque cognitivo, cuyo coste computacional es menor que el de los algoritmos tradicionales. En *Machine Learning* existen dos ramas del aprendizaje, el aprendizaje supervisado y el no supervisado; en este trabajo nos enfocaremos en ambos, construyendo un autoencoder profundo (Aprendizaje no supervisado), entrenando el modelo y extrayendo la información de la estructura interna, esta metodología también es conocida como *clustering*. Finalmente, la información comprimida (La mayoría de los autoencoders tienen una representación interna reducida de la entrada) es clasificada mediante un modelo de aprendizaje supervisado llamado *Extreme Learning Machine*, en dos clases. En este trabajo se usó una base de datos proporcionada por la Universidad *Case Western Reserve* en su sitio web. Las señales fueron medidas a través de un acelerómetro a una tasa de muestreo de 12 mil muestras por segundo, radialmente dispuesto en un rodamiento. Las mediciones fueron realizadas para cuatro casos, el primero fue una señal vibratoria saludable y los siguientes tres correspondientes a baleros con fallas inducidas intencionalmente en el canal interno, canal externo y el balín respectivamente. Se trabajó en el entorno basado en *Python*, *Jupyter*, el cual puede ejecutar herramientas de *Machine Learning* como *Tensor Flow* y *Keras*; también se construyó el modelo. En este trabajo se propuso un sistema basado en hardware para imitar el modelo computacional en un dispositivo *Field Programmable Gate Array.* (*FPGA*, pos sus siglas en inglés), importando los parámetros numéricos al entorno de desarrollo *Vivado* de *AMD Xilinx*, dando lugar a un sistema de monitoreo de salud mecánica en tiempo real.

0.2. ABSTRACT

In this work, we have realized a study about Artificial Intelligence for the extraction and recognition of mechanical vibration information, that were obtained by electronic instruments.
Mechanical vibration signals have been studied traditionally by mathematical models and Gaussian analysis in many fields like seismic signals and mechanical health monitoring. In recent years, novel trends in computing have improved the efficiency and adaptability of Artificial Intelligence algorithms, which can be implemented to help solving some of the mankind needs. In signal processing and pattern recognition by Artificial Intelligence, the information is treated from a cognitive approach, whose computational cost is lower than traditional signal processing.
In Machine Learning there are two learning branches: supervised and unsupervised learning; in this work we focused in both, building a Deep Autoencoder architecture(Unsupervised architecture), training the model and extracting the internal structure information, also known as clustering. Finally, the compressed information (Most Autoencoders have a reduced input representation in the internal structure) is classified by a supervised model Extreme Learning Machine, in two classes.

We used a data set provided by the Case Western Reserve University in its web site. The signals were measured by sampling an accelerometer at *12kS/s*, radially placed in a bearing. The measures were made for four cases the first was a vibrational signal in a healthy bearing, and three bearings with intentionally induced failures in the inner race, outer race and the ball respectively.

We worked in the *Python* language environment *Jupyter*, which can perform Machine Learning tools like *Tensor Flow* and *Keras*; we built and trained the model as well. We proposed hardware based system to mimic the computational model in FPGA devices, importing the numerical parameters to the *AMD Xilinx* development environment *Vivado*, giving rise to a possible real time health monitoring system.

Introducción

La estructura de este trabajo la constituyen fundamentos teóricos y aplicaciones prácticas, con la finalidad de transmitir al lector un contenido integral. En el capítulo 1, se dará una breve introducción sobre el mercado y los campos de aplicación de elementos giratorios de sistemas mecánicos, como es el caso que se abordó en esta tesis, usando rodamientos. Se realizará un análisis de los factores que ponen en riesgo su estado; se hará un recuento de todas las metodologías existentes dedicadas al diagnóstico del estado de los elementos giratorios y finalmente se realizará una descripción de la base de datos empleada. En el capítulo 2 se mencionarán conceptos fundamentales de las dos ramas de la ciencias de la computación que se involucran en este trabajo, específicamente la inteligencia artificial y la ciencia de datos, también conocida como *Big Data*. Se abordarán temas específicos de cada una de ellas y, además, se realizará una explicación detallada del esquema de Internet de las Cosas, tratando de mostrar el vínculo entre dichas tecnologías, cuya conjunción da lugar a una herramienta con grandes prestaciones. En el capítulo 3, se introducirá al lector hacia los conceptos específicos de Aprendizaje en Máquina utilizados, relacionados al aprendizaje supervisado y no supervisado. Se realizará un recuento de las arquitecturas existentes y se profundizará en las metodologías de reducción de dimensión y clasificación, desde el punto de vista matemático y computacional. Esto a través de planteamientos matemáticos y descripciones en lenguaje *Python* de modelos neuronales, haciendo uso de los paquetes *Tensor Flow* y *Keras*. En el capítulo 4 se hará un estudio de las arquitecturas seleccionadas para llevar a cabo este trabajo, así como la influencia de la estructura del entrenamiento, mediante la selección de métodos de optimización y configuración de hiperparámetros. En el capítulo 5 se hace un análisis del desempeño de la arquitectura final, evaluando la calidad de reconstrucción y la exactitud de la clasificación. En el capítulo 6 se hace una propuesta de un sistema digital que emula el comportamiento de una neurona, mediante la descripción en lenguaje *Very High Speed Integrated Circuit Hardware Description Language* (*VHDL*, por sus siglas en inglés) demostrando las capacidades cognitivas de los modelos neuronales.

0.3. Objetivos

Este trabajo de maestría tiene como objetivo particular mostrar un método de inteligencia artificial basado en técnicas de aprendizaje en máquina, del inglés: *Machine Learning*, el cual permita detectar con un alto grado de certidumbre fallas específicas de un sistema mecánico. Estas fallas están asociadas al desgaste de rodamientos o baleros, en los cuales un sistema de transducción de vibración aporta los valores en tiempo real de vectores del estado o grado de desgaste. Se propone que los valores obtenidos de magnitud y unidades arbitrarias puedan ser procesados por un arreglo de dos redes neuronales conectadas en serie, un autoencoder y un clasificador. Ambas redes deben ser entrenadas con métodos de redes neuronales profundas para lograr un desempeño aceptable.

0.4. Objetivos Particulares

• Definir una arquitectura de dos redes neuronales y métodos de entrenamiento en la tarea de detección temprana de fallas en sistemas mecánicos de rodamiento.

• Mostrar de forma sistemática el método de preparación de datos, desde una base experimental disponible de un laboratorio especializado de mecánica.

• Proponer un sistema electrónico de tipo numérico que pueda integrar estas redes para su aplicación en tiempo real en la detección de fallas mecánicas.

Acrónimos

AI: **A**rtificial **I**ntelligence

*ASIC:***A**pplication **S**pecific **I**ntegrated **C**ircuit

CAEs: **C**onvolutional **A**utoencoder**s**

CNN: **C**onvolutional **N**eural **N**etwork

DAE: **D**enoising **A**utoencoder

ELBO: **E**vidence **L**ower **Bo**und

ELM: **E**xtreme **L**earning **M**achine

FPGA: **F**ield **P**rogramable **G**ate **A**rray

FFT: **F**ast **F**ourier **T**ransform

GANs: **G**enerative **A**dversarial **N**etwork**s**

HDL: **H**ardware **D**escription **L**anguage

IoT: **I**nternet **o**f **T**hings

IRT: **I**nfra**r**ed **T**hermography

KCM: **K**-means **C**lustering **M**odel

MAE: **M**ean **A**bsolute **E**rror

OOMS: **O**nline **O**il **M**ulti-parameters monitoring **S**ystem

PCA: **P**rincipal **C**omponent **A**nalysis

PHM: **P**rognostics and **H**ealth **M**anagement

RBF: **R**adial **B**asis **F**unction

SA: **S**tate **A**ssessment

SGD: **S**tochastic **G**radient **D**escent

SLFNs: **S**ingle **L**ayer **F**eedforward **N**eural **N**etwork**s**

SVD: **S**ingular **V**alue **D**ecomposition.

VHDL: **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

CAPÍTULO 1

Fallas mecánicas en sistemas con rodamientos

Se estima que cada año son manufacturados 10 billones de rodamientos en todo el mundo, el 90 % de ellos son colocados en maquinaria o equipo en el cual permanecen de por vida, 9.5 % de ellos son reemplazados de manera preventiva, y el 0.5 % de ellos son reemplazados por razones correctivas, esto es descrito en la Figura 1.1 por el **Grupo SKF©**, en adelante **SKF**, una empresa de manufactura de rodamientos especializados, con presencia en el mercado mundial.

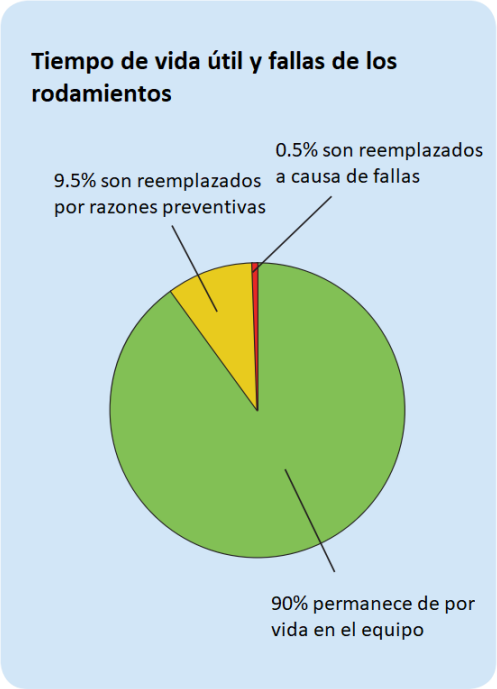

Figura 1.1: Gráfico de estimación de fallas en rodamientos, adaptado de **SKF**.

1.1. Factores de falla en rodamientos

Existen diferentes factores que pueden causar una falla en un rodamiento, algunas de ellas están relacionadas a las condiciones operativas a las cuales se somete el rodamiento. La Ecuación 1.1, proporcionada por **SKF**, estima el número de revoluciones en las cuales un rodamiento trabajaría en condiciones óptimas:

$$L_{nm} = a_1 * a_{SKF} \left(\frac{C}{P}\right)^p \tag{1.1}$$

Donde:

L_{mn} = Indice de vida **SKF** (millones de revoluciones)
a_1 = Ajuste de vida para confiabilidad
a_{SKF} = Factor de modificación de vida **SKF**
C = Indice de carga básica dinámica[kN]
P = Carga dinámica equivalente del rodamiento [kN]
p = Exponente de la ecuación

Como se observa en la Ecuación 1.1, el tiempo de vida depende de las condiciones operativas como la carga, alineación, etc. Existen más factores que influyen en el estado de los rodamientos; en la Figura 1.2, se observa una gráfica que muestra las causas más comunes que ocasionan una falla en un rodamiento según la empresa **Bearing & Drive Systems©**, una empresa estadounidense que manufactura rodamientos especializados.

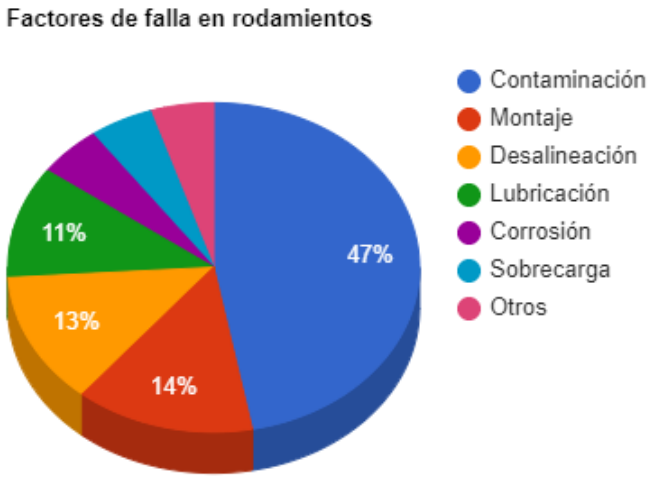

Figura 1.2: Gráfico de causas de falla, adaptado de **Bearing & Drive Systems**.

1.1.1. Causas comunes de falla en los rodamientos

A continuación se enlistan 13 de las causas más comunes de falla en los rodamientos:

1.- Lubricación inapropiada

La lubricación inadecuada podría ser una de las causa principales de inducción de falla en rodamientos. La temperatura operativa de un rodamiento está asociada con la viscosidad del lubricante; una lubricación pobre o una sobre-lubricación pueden causar un fallo en la operación del elemento giratorio.

2.- Daño de la jaula

El daño en la jaula del rodamiento puede ser la causa de un fallo en la operación del rodamiento. Este daño es ocasionado principalmente por vibración, operar a mayor RPM para las que el rodamiento está diseñado, desgaste y bloqueo.

3.- Contaminación y corrosión

Hay diversos contaminantes, que ocasionan fallos en la operación del rodamiento tales como el polvo, agua, agentes químicos etc. La presencia de estos elementos contaminantes ocasiona una modificación en la viscosidad, corrosión y erosión.

4.- Arco eléctrico

La falla por erosión de arco eléctrico ocurre cuando el rodamiento conduce una corriente eléctrica y ocasiona un aumento de la temperatura en las áreas de contacto superficial. Este fenómeno ocasiona fisuras localizadas en los canales y los elementos giratorios.

5.- Ajuste inadecuado

Es importante que las dimensiones del rodamiento y del lugar de montaje coincidan con exactitud, de no ser así, podría ocasionar dos anomalías:

Alojamiento sub-dimensionado

Esto puede ocasionar baja holgura interna en los elementos de giro.

Alojamiento sobre-dimensionado

En este caso el rodamiento tenderá a deslizarse y como consecuencia se desgastará y generará calor y vibración.

6.- Fatiga

La fatiga en los rodamientos ocurre cuando se crea una fractura en las superficies de contacto; dicha fractura tiende a desprender partículas residuales de metal.
Esta falla ocurre normalmente al final de la vida útil del rodamiento o cuando ha sido sometido a carga excesiva; una característica distintiva de este tipo de falla es el aumento de la vibración.

7.- *Brinelling*

Este tipo de falla ocurre cuando las cargas exceden el límite elástico del material del anillo; su nombre se debe a la escala de dureza de Brinell, la cual caracteriza la dureza de deformación de los materiales a través de la escala de penetración de un punzón, cargado en una pieza de material de prueba.
Esta falla puede ser identificada a través de marcas permanentes de hendiduras en los canales de giro, los cuales incrementan la vibración. Existen dos categorías dentro de este tipo de falla:

1.- ***Brinelling* verdadero**
Esta categoría se refiere al *Brinelling* ocasionado por cargas excesivas que exceden el límite elástico del material.

2.- ***Brinelling* Falso**
Esta categoría hace referencia al *Brinelling* ocasionado por vibración u oscilaciones entre los elementos del rodamiento.

8.- Desalineación

La desalineación, trae consigo vibración y fuerzas de carga para las cuales no está diseñado el rodamiento. La desalineación se debe principalmente a ejes excéntricos, contaminación en el cuerpo de contacto del eje o la carcasa, entre otras.

9.- Patrones de trayectoria

Los patrones de trayectoria proporcionan información sobre las condiciones operativas del rodamiento en análisis, que se encuentra en servicio. Al ser sometidos a análisis los patrones de trayectoria, proporcionan información sobre las condiciones operativas del rodamiento.

10.- Selección y mantenimiento de sellos

Los sellos garantizan que la contaminación no ingrese al interior del rodamiento, lo cual permite que la lubricación no se modifique.

11.- Sobrecarga

La sobrecarga es una causa común de daño en los rodamientos, al operar bajo condiciones para las cuales no han sido diseñados.

12.- Defectos de almacenaje

Los rodamientos tienden a fallar cuando han sido expuestos a condiciones de intemperie; condiciones como humedad, polvo y temperaturas cambiantes son factores que influyen en el estado de los rodamientos.
La incorrecta manipulación por el personal de almacenaje también es un factor de riesgo, especialmente por una potencial ruptura de los sellos.

13.- Holgura inadecuada

La holgura en un rodamiento ocasiona un incremento en la temperatura, e induce fallas como modificación de la lubricación y fricción interna.

1.2. Métodos Alternativos de selección de fallas

El monitoreo de la salud mecánica de los elementos giratorios ha sido estudiado ampliamente desde diferentes enfoques, cuyos procesos experimentales pueden ser simples, como es el caso de los acelerómetros o sensores ultrasónicos hasta experimentos rigurosos con análisis microscópico. En esta sección se dará una breve explicación de los sistemas empleados para monitoreo de salud mecánica.

1.2.1. Análisis vibracional de alta frecuencia

El análisis vibracional es el estudio del movimiento introducido por una perturbación del cuerpo, relacionado con la posición de equilibrio. La perturbación puede ser:

a)Movimiento del cuerpo a gran escala.
b)Perturbación de una pequeña sección de material dentro del cuerpo, a pequeña escala.

La información más importante a recabar, es la referente a las ondas de estrés. Es importante colocar los transductores muy cerca de la fuente de vibración (preferentemente en la zona de carga radial). La obtención de la información vibracional puede hacerse mediante dos sensores: Acelerómetro y sensor ultrasónico.

Metodología de reducción de datos

El análisis dominante de vibración ocurre en el dominio espectral de la frecuencia. Esto requiere que una forma de onda discreta bajo el dominio del tiempo sea transformada, con el fin de obtener su representación en el dominio de la frecuencia; el análisis de Fourier es uno de los tratamientos más empleados para el estudio espectral. Este análisis se puede aplicar a cualquier función periódica real mediante la transformada de Fourier. Por otra parte, para señales discretas existe el algoritmo numérico *Fast Fourier Transform* (**FFT**, por sus siglas en inglés) el cual es usado para realizar dicho análisis enfocado hacia señales discretas. La función de auto correlación obtenida por el análisis *FFT* proporciona al analista información sobre los eventos periódicos que están sucediendo en el sistema. Para estudiar una forma de onda es necesario por lo menos obtener la información de 15 revoluciones; esto debido al ancho de banda de los instrumentos. Comúnmente, los acelerómetros y sensores ultrasónicos suelen trabajar en bajas frecuencias por lo que los fenómenos que sensan son de escala macroscópica y como las condiciones experimentales se encuentran entre 700 y 3600 revoluciones por minuto (RPM) [3, p. 3], para obtener una señal con alto grado de exactitud se usa una razón de muestreo de 50 mil muestras por segundo, esto hace que el número de muestras a analizar sea significativo; para ejemplificar esto, se muestra en la Ecuación 1.2 el número de muestras que se pueden obtener de una revolución, con una velocidad angular de 700 RPM.

$$T_{rev} = \frac{60\,s}{700\,rev} = 0.085\,s/rev \tag{1.2}$$

Donde T_{rev} es el periodo en segundos en el que se completa una revolución. El periodo de muestreo se determina en la Ecuación 1.3.

$$T_{muestra} = \frac{1}{50000\,muestras/s} = 0.00002\,s/muestra \tag{1.3}$$

Donde $T_{muestra}$ es el periodo de la muestra en segundos.
De 1.2 y 1.3 se tiene que el número de muestras por revolución se determina mediante la relación entre en periodo de revolución y de muestreo como se observa en la Ecuación 1.4:

$$N_{muestras} = \frac{T_{rev}}{T_{muestra}} = \frac{0.085\,s/rev}{0.00002\,s/muestras} = 4250\,muestras/rev \tag{1.4}$$

De 1.4 se puede determinar que el número de muestras obtenidas de 15 revoluciones es de 63750, brindando así una resolución aceptable.

Existen dos métodos para extraer la información vibracional de las formas de onda, los cuales son demodulación de amplitud y detección de valor pico. Este último desarrollado por grupo **Emerson Reliability Solutions©** [3]. Dichos métodos consisten en un tratamiento de la señal en diferentes etapas; a continuación se muestran su metodología:

1) Demodulación de Amplitud:

    1) Filtro pasa altas.
    2) Rectificador de onda completa.
    3) Filtro pasa bajas (para separar la señal modulada de la señal "portadora").
    4) Convertir la señal resultante en una señal digital con un ancho de banda que excede la más alta frecuencia de falla, con una duración suficiente para muestrear 15 revoluciones.

2) Valor pico.

    1) Filtro pasa altas.
    2) Rectificador de onda completa.
    3) Convertidor de valor pico absoluto.

Estas etapas de procesamiento permiten enfocar el análisis de la señal en las ondas de estrés, atenuando la información que no esta relacionada. En la Figura 1.3, se ilustran dos gráficas de la metodología, obtenidas de un sensor ultrasónico; la gráfica púrpura corresponde a un valor de demodulación pico-pico de 0.27 $Gal(cm/s^2) * s$ y la gráfica azul corresponde a un valor de demodulación de 1.24 $Gal * s$, de un estado normal y de alerta respectivamente.

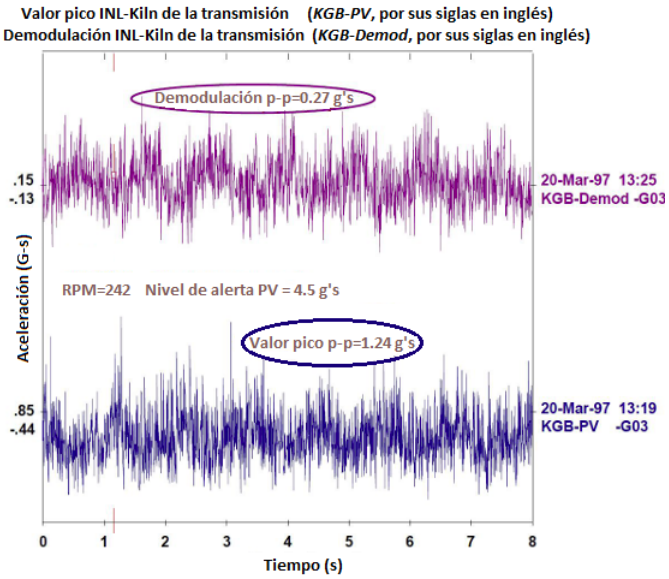

Figura 1.3: Gráficas del análisis en alta frecuencia. **Gráfica purpura:** Señal de baja vibración. **Gráfica azul:** Señal de alta vibración. Adaptado de [3].

1.2.2. Análisis de lubricante

En los elementos giratorios, el lubricante proporciona información valiosa sobre el estado del sistema. El análisis de partículas de desgaste es uno de los métodos más precisos para determinar el estado de salud de un elemento giratorio. Los nuevos sistemas de análisis se centran en el estudio de la correspondencia entre los parámetros del lubricante y el estado de los elementos de giro. Sistemas de diagnóstico como el sistema *Online Oil Multi-parameters Monitoring System* (**OOMS**, por sus siglas en inglés) es un ejemplo de esta técnica, la cual es ampliamente usada, pero resulta muy compleja de implementar, además de necesitar de condiciones controladas de experimentación. Los parámetros que se analizan son: Residuos micro-metálicos, contaminación, viscosidad, agua, indicadores de densidad y señales vibracionales.

Los métodos de análisis de lubricante usualmente analizan las características de las partículas dispersas en el lubricante, como es el caso de la metodología *K-means Clustering Model* (**KCM**, por sus siglas en inglés) que estudia la separación entre ellas. Ésta es una métrica usada para agrupar y clasificar las partículas, siendo un punto de referencia para determinar el estado de salud de un elemento giratorio; un ejemplo ilustrativo de esta metodología se observa en la Figura 1.4, donde se muestran 5 ejemplos de agrupamiento; los puntos negros son el centro de los grupos y los enlistados (A-E) son los elementos miembros de los grupos.

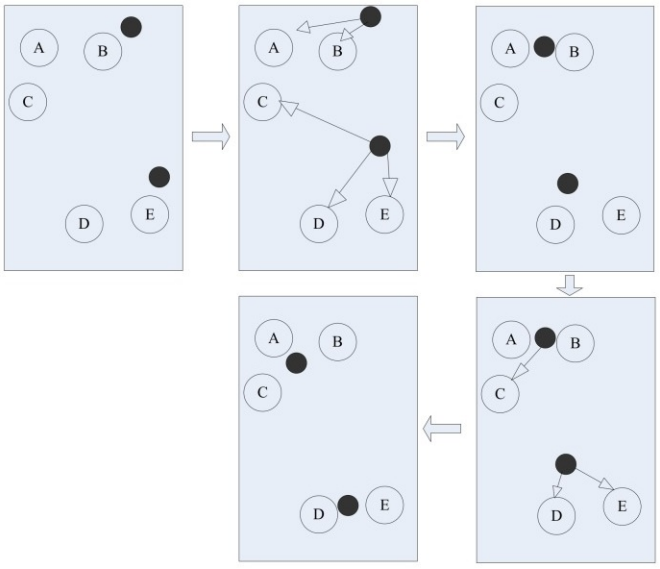

Figura 1.4: Metodología de *clustering* del modelo *KCM*. Adaptado de [13].

La metodología mencionada en conjunción con otros parámetros como temperatura, viscosidad, densidad, agua, suelen usarse como criterios para determinar el estado de elementos giratorios mediante una relación de correlación entre ellos [13].

1.2.3. Análisis infrarrojo

La técnica *Infrared Thermography* (**IRT**, por sus siglas en inglés) actualmente es utilizada en los sistemas como *Prognostics and Health Management* (**PHM**, por sus siglas en inglés) y *State Assessment* (**SA**, por sus siglas en inglés) entre otros, cuya metodología consiste en la determinación del estado de salud de la maquinaria mediante el procesamiento de señales adquiridas de cámaras térmicas; en la Figura 1.5, se aprecia un ejemplo del banco de pruebas genérico que es empleado en este análisis.

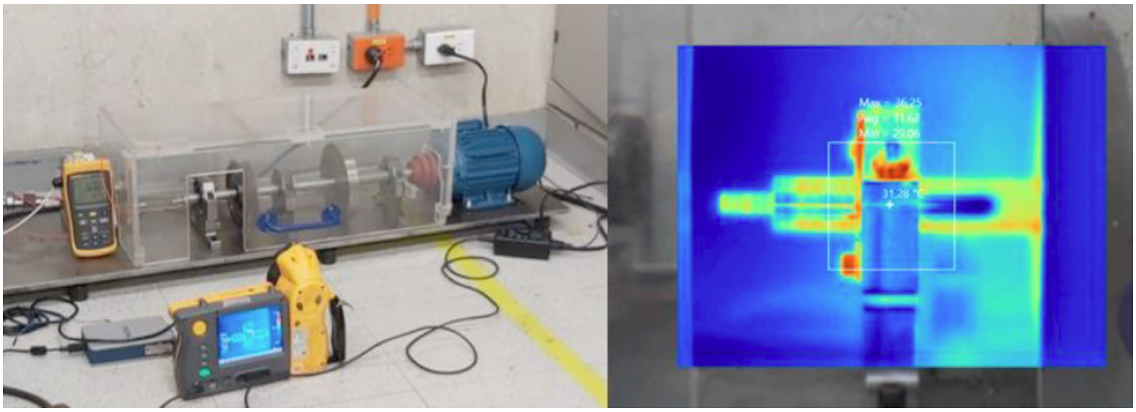

Figura 1.5: Banco de pruebas para monitoreo infrarrojo. Tomado de [11].

Esta metodología se basa en el principio de captura de señal térmica mediante un arreglo de sensores dispuestos de manera matricial, la información obtenida es clasificada dependiendo del valor de cada píxel muestreado, como se puede observar en la Figura 1.6.

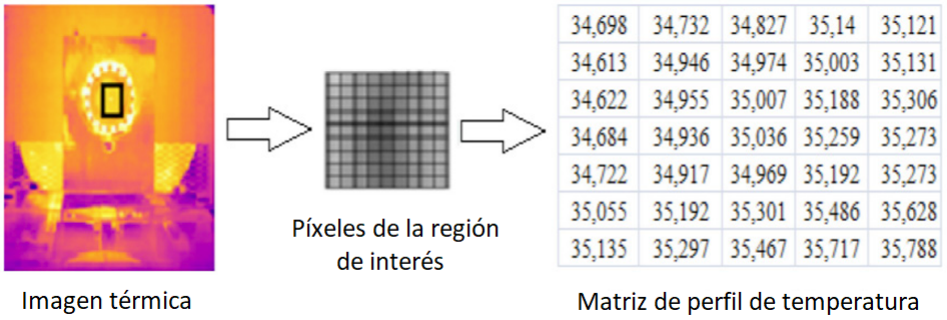

Figura 1.6: Esquema funcional de la captura de información. Adaptado de [11].

Esta metodología puede dividirse en dos tipos de análisis: el estacionario y el transitorio, los cuales son explicados a continuación.

Estacionario:

El análisis estacionario se realiza mediante dos criterios:

Cualitativo:

Este criterio consiste en determinar la temperatura relativa entre la temperatura máxima de la maquinaria y la temperatura ambiente, En la Ecuación 1.5 se aprecia la aplicación de este criterio.

$$\Delta T_{quant} = T_m - T_a \tag{1.5}$$

Donde T_m es la temperatura máxima del sistema y T_a es la temperatura ambiente.

Cuantitativo:

Este criterio consiste en calcular la temperatura entre el sistema de prueba y otro sistema saludable operando bajo condiciones similares, En la Ecuación 1.6, se aprecia la forma matemática de la aplicación de este criterio.

$$\Delta T_{qual} = T_m - T_h \tag{1.6}$$

Donde T_m es la máxima temperatura del sistema de prueba y T_h es la máxima temperatura del sistema saludable.

Transitorio:

Análisis Transitorio [11] consiste en modelar el comportamiento de transferencia térmica de los sistemas mecánicos cuando se ejerce cierto trabajo en ellos. En la Ecuación 1.7 se observa el modelo matemático de dicho comportamiento.

$$c_p T(t)' = W' - hA(T(t) - t_a) \tag{1.7}$$

Donde c_p es la capacidad calorífica del sistema, W' es el trabajo mecánico, h es el coeficiente de transferencia térmica por convección, A es el área superficial, donde toma lugar la convección, T_a es la temperatura ambiente.

La solución de la Ecuación 1.7 es analítica y tiene la forma:

$$T(t) = \frac{K_2}{K_1} - c_1 e^{-K_1 t} \tag{1.8}$$

Donde:

$$K_1 = \frac{hA}{c_p},\ K_2 = \frac{hAT_a + W'}{c_p}$$

Haciendo un análisis para la solución encontrada para dos posibles casos, el estado inicial ($t = 0$) y el estado estacionario ($t \to \infty$), se tiene:

Estado inicial:

$$T(t = 0) = T_a = \frac{K_2}{K_1} - c_1 e^0 = \frac{K_2}{K_1} - c_1 \tag{1.9}$$

Estado estacionario:

$$T(t \to \infty) = \frac{K_2}{K_1} - C_1 e^{-\infty} = \frac{K_2}{K_1} \tag{1.10}$$

De 1.9, se obtiene:

$$c_1 = \frac{K_2}{K_1} - T_a = T(\infty) - T_a \tag{1.11}$$

De la Ecuación 1.11 se concluye que c_p es la temperatura correspondiente al análisis estacionario con criterio cuantitativo (véase la Ecuación 1.5); por lo tanto la relación K_2/K_1 así como c_p son indicadores del estado de la maquinaria, pues una variación en las mediciones podría indicar alguna anomalía en el sistema.

1.2.4. Contribución a los sistemas precedentes

En este trabajo, el objetivo es encontrar un sistema compacto, asequible, capaz de determinar el estado de salud de un sistema giratorio, el cual sea factible de implementar en un *Aplication Specific Integrated Circuit* (*ASIC*, pos sus siglas en Inglés), para poder emplear dichos sistemas en toda clase de entornos, proporcionando una herramienta de diagnóstico que no necesita de sistemas de análisis complejos, así como instrumentos de medición sofisticados. Esto se hace posible gracias a las tecnologías de aprendizaje en máquina, cuyas características hacen posible la extracción de información por medio de procesos de cálculo, cuyas arquitecturas son viables para su implementación en dispositivos lógicos programables.

1.3. Base de Datos empleada

La base de datos fue descargada del sitio de la Universidad *Case Western Reverse*, el cual fue financiado por el centro científico de **Rockwell Automation©**, y la oficina de investigación naval de EEUU; en la Figura 1.7 se aprecian los logotipos de las instituciones mencionadas.

Figura 1.7: **Izquierda** Logotipo de la Universidad *Case Western Reserve*. **Derecha** Emblema de la oficina de investigaciones Navales de EEUU. **Abajo** Logotipo de la Compañia de automatización *Rockwell Automation*©

1.3.1. Descripción de la base de datos

La base de datos consta de un conjunto de archivos que contienen información numérica de las vibraciones de un rodamiento, medidas bajo condiciones normales y de falla intencionalmente inducidas. Cada archivo contiene la información vibracional correspondiente a una única condición experimental. Los archivos pueden clasificarse según la región en donde fue inducida la falla así como el diámetro de la misma. Las fallas fueron intencionalmente provocadas mediante descarga eléctrica mecanizada en un solo punto, con diámetros de 7mil(0.1778mm), 14mil(0.3556mm) y 21mil(0.5334mm). En la Tabla 1.1, se asocian los archivos con sus respectivas condiciones experimentales, para un rodamiento rígido de bolas, de la marca **SKF** y modelo `6205-2RS JEM`.

Diámetro de la falla	Carga mecánica (HP)	Velocidad del motor (RPM)	Área de la falla inducida	Nombre del archivo	Tamaño (MB)
0.007"	0	1797	Canal interno	`IR007_0`	2.33 MB
0.014"	0	1797	Canal interno	`IR014_0`	2.37 MB
0.021"	0	1797	Canal interno	`IR021_0`	2.33 MB
0.007"	0	1797	Canal externo[1]	`OR007@6_0`	2.32 MB
0.014"	0	1797	Canal externo	`OR014@6_0`	2.38 MB
0.021"	0	1797	Canal externo	`OR021@6_0`	2.36 MB
0.007"	0	1797	Balin	`B007_0`	2.38 MB
0.014"	0	1797	Balin	`B014_0`	2.39 MB
0.021"	0	1797	Balin	`B021_0`	2.37 MB
-	0	1797	Ninguna[2]	`Normal_0`	4.66 MB

Tabla 1.1: Tabla descriptiva de los archivos. Adaptado de [10].

1.3.2. Condiciones experimentales

Los rodamientos de prueba soportan el eje del motor; en la Figura 1.8 se aprecia el banco de pruebas usado para las mediciones experimentales, el cual consta de 3 etapas fundamentales: la generación de fuerza electromotriz, medición de las magnitudes mecánicas y carga radial.

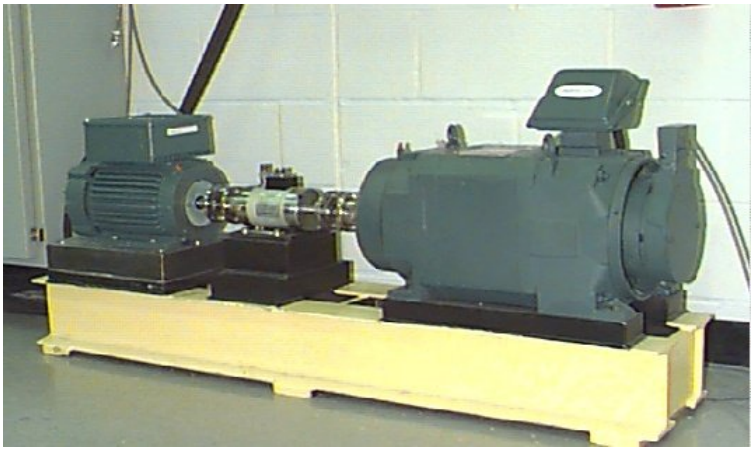

Figura 1.8: Banco de experimentación. **Izquierda:** Motor de 2 caballos de fuerza. **Centro:** transductor/codificador de torque. **Derecha:** Dinamómetro. Tomado de [10].

1.4. Conclusiones del capítulo

En este capítulo se realizó un breve análisis de la importancia del diagnóstico temprano de sistemas mecánicos giratorios, pues estos se encuentran inmersos en la mayoría de los procesos productivos, y el análisis de su estado es vital para prevenir perdida de producción, especialmente en nuestra era donde la manufactura en serie y a gran escala es crítica para satisfacer las demandas de una sociedad en desarrollo y con crecientes necesidades. El diagnóstico temprano de fallas en sistemas giratorios ha sido abordado desde diferentes enfoques, como los son las mediciones de parámetros químicos en el lubricante, la captura de información térmica y el análisis vibracional. De esta manera se concluye que es necesario contribuir al desarrollo de la sociedad, aplicando las tecnologías emergentes en las áreas productivas, mediante la propuesta de un posible sensor asequible al cual puedan tener acceso no solo las industrias altamente automatizadas sino también las iniciativas de producción a pequeña y mediana escala.

[1] La falla de canal externo inducida en el centro de la región de carga radial.
[2] Archivo correspondiente a un rodamiento saludable.

CAPÍTULO 2

Tratamiento de información y ciencias de la computación

Las Ciencias de la Computación son disciplinas jóvenes históricamente hablando, pues carecen de precedentes en comparación con las ciencias como Matemáticas, Física y Química, cuyo estudio ha sido abordado desde las primeras civilizaciones. El desarrollo de las Ciencias de la Computación ha sido tan rápido que el modo de vida es radicalmente diferente en la actualidad que en épocas pasadas[1].

Históricamente, cuando las personas comenzaron a construir máquinas de cálculo, es decir en los inicios de la Computación, las partes que constituían la máquina eran concebidas como unidades específicas que formaban parte de la organización de la misma como un todo[9]; posteriormente, se observó que la interacción entre diferentes unidades era sustancialmente compleja y de esta manera surgió la necesidad de crear métodos sistemáticos de organización.

Los métodos sistemáticos empleados tenían un enfoque algebraico, motivo por el cual las ciencias de la computación aun retienen dicha metodología en múltiples tareas. Con el auge de los circuitos integrados, la computación se trasladó al campo del álgebra discreta; posteriormente se plantearon *homomorfismos*[1] entre computadoras, llegando así hasta la computación moderna.

En este capítulo se hace una introducción a algunos conceptos importantes del área de computación y ciencia de datos, mediante teoría y algunos desarrollos matemáticos, con la intención de ofrecer al lector una inducción a dichas ciencias y facilitar así la comprensión de este trabajo.

[1] El significado de *homomorfismo* proviene de la composición de dos palabras en griego antiguo: *Homo* que significa "mismo" y *morfe* que significa "forma".

2.1. Conceptos de aprendizaje en máquina

El aprendizaje en máquina, también conocido en inglés como *Machine Learning*, es una rama de la inteligencia artificial, la cual data de 1940, sin embargo, muchos autores la consideran un área tecnológica nueva, pero esto solo es cierto porque fue bastante impopular en los años que precedieron su éxito. Existen tres épocas fundamentales que definieron la inteligencia artificial, como se observa en la Tabla 2.1.

Periodo	Nombre de referencia	Autores principales
1940-1960	Cibernética	McCulloch and Pitts (1943), Hebb (1949), Rosenblatt (1958)
1980–1995	Conexionismo	Rumelhart et al (1986)
2006-Actualidad	Machine Learning & Deep Learning	Hinton et al (2006) , Bengio (2007), Ranzato (2007)

Tabla 2.1: Tabla de épocas de auge de la Inteligencia Artificial.

En la Tabla 2.1 se muestran los nombres con los que se referían a la hoy conocida *Artificial Intelligence* (*AI*, por sus siglas en inglés), de acuerdo a *Google Books*. La primera época fue impulsada por el desarrollo de las teorías de aprendizaje biológico, además de la introducción del primer modelo de aprendizaje, el perceptrón que permitió el entrenamiento de una sola neurona. La segunda época se basó en la introducción del algoritmo de entrenamiento *back propagation* que permitió entrenar modelos de más de una capa. Finalmente, la actual y tercera época tiene una fuerte influencia de las neurociencias, las cuales han ayudado a crear modelos neuronales que intentan imitar el comportamiento del cerebro humano, pero estos modelos aun se encuentran alejados de la comprensión del funcionamiento de un cerebro natural.

El aprendizaje en máquina, representa modelos computacionales bio-inspirados que involucran ramas de las ciencias naturales e ingeniería. Álgebra Lineal, Cálculo Vectorial, Probabilidad y Estadística, Teoría de la Información y Computación Numérica, son algunos ejemplos de ello.
Los componentes primordiales que componen un modelo de aprendizaje son los siguientes:

1.- El conjunto de datos, del cual se aprenderá.
2.- Un modelo que transformará los datos.
3.- Una función objetivo que cuantifique cuán satisfactoriamente aprende el modelo.
4.- Un algoritmo para ajustar los parámetros del modelo, que optimice la función objetivo.

2.1.1. Conjunto de datos

El conjunto de datos es la base sobre la cual todo el modelo aprenderá a realizar la tarea que se le asigne, generalmente entre mayor sea el conjunto de datos disponible, el proceso de aprendizaje será mejor. Es importante señalar que un criterio importante para la selección del conjunto de datos es la calidad de la información; ésta no debe contener errores de ninguna índole, de igual manera debe estar suficientemente enriquecida con todas las variantes del problema a analizar, para así proporcionar al modelo información confiable y adecuada en términos estadísticos.

2.1.2. Modelos

La mayoría de los modelos de *Machine Learning* involucran una transformación de algún tipo de los datos. Los modelos de interés son del tipo estadístico; dichos modelos consisten en transformaciones sucesivas de los conjuntos de datos, cuyo tratamiento facilita la realización de tareas que implican aprendizaje.

2.1.3. Funciones Objetivo

Una función objetivo puede interpretarse como una función que registra un evento o los valores de variables a los cuales se asigna intuitivamente un "costo", el cual es un número real asociado con dicho evento o variable. La función objetivo puede ser tanto una función de pérdida, como su complemento, es decir función de recompensa (también llamada función de beneficio, función de utilidad, función de aptitud), en cuyos casos los métodos de optimización tienden a minimizar y maximizar respectivamente. La función objetivo evalúa la precisión en la salida de la red, calculando una tasa de aserción entre los valores deseados y los calculados. Al evaluar la función objetivo se obtiene un valor numérico que es interpretado como una métrica que informa qué tan bien está aprendiendo el modelo neuronal, con respecto a un valor ideal.

2.1.4. Métodos de Optimización

En un problema de *Deep Learning* se define usualmente una *función objetivo*. Una vez que se tiene la función objetivo, se puede usar un algoritmo de optimización con la finalidad de minimizar la pérdida. En la optimización, una *función objetivo* también puede ser llamada *función de pérdida*. Tradicionalmente y por convención la mayoría de los algoritmos de optimización se ocupan de la minimización.

Objetivo de la optimización

Aunque la optimización proporciona una manera de minimizar la *función de pérdida* para *Deep Learning*, en esencia, los objetivos de la optimización y de *Deep Learning* son fundamentalmente diferentes. El primero se ocupa principalmente de minimizar un objetivo y el segundo se ocupa de encontrar un modelo adecuado, dado un conjunto de datos finito. Dicho lo anterior, el error de entrenamiento y el error de generalización generalmente difieren: la función objetivo del algoritmo de optimización es usualmente una pérdida basada en el conjunto de datos de entrenamiento, el objetivo de la optimización es reducir el error de entrenamiento [20]. Sin embargo, el objetivo de *Deep Learning* es reducir el error de generalización (generalmente conocido como inferencia estadística).

Los Algoritmos de Optimización son diversos con muchas técnicas especializadas, y algunas técnicas generales, están divididos en diferentes tipos, incluidos los basados en gradiente, algoritmos libres de gradiente como los algoritmos evolutivos y meta heurísticos inspirados en la naturaleza.

Métodos de optimización basados en gradiente

Los Métodos basados en gradiente son métodos iterativos que usan extensamente información diferencial de la función objetivo, durante las iteraciones.

A) Método de Newton

Para la minimización o maximización de una función invariante $f(x)$, es equivalente a encontrar las raíces de su gradiente $g(x) = f'(x) = 0$; El algoritmo de búsqueda de raíces de Newton se describe a continuación, en la Ecuación 2.1:

$$x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{2.1}$$

Los métodos basados en gradiente son muy eficientes; dado que la solución final tiende a depender del los valores iniciales. Si los valores iniciales son lejanos a la solución óptima, el algoritmo puede obtener una solución diferente en cada iteración para problemas multi-modales y/o fallar.

B) Gradiente descendiente

Hay tres variantes de gradiente descendiente, las cuales difieren en cuánta información es usada para procesar el gradiente de la función objetivo. Dependiendo del conjunto de datos, se realiza una compensación entre la exactitud del parámetro de actualización y el tiempo que toma para realizar una actualización.

C) Gradiente descendiente por lote

Este algoritmo calcula el gradiente de la función costo a los parámetros θ para el conjunto de datos completo de entrenamiento.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{2.2}$$

Como es necesario calcular los gradientes para la base de datos completa para calcular la única actualización, el método gradiente descendiente por lotes puede ser muy lento y no es viable para conjuntos de datos que no entrenan en memoria. Este método tampoco permite actualizar el modelo en línea.

D) Gradiente descendiente estocástico SGD

Stochastic Gradient Descent (*SGD*, por sus siglas en inglés) realiza una actualización de parámetros para cada ejemplo de entrenamiento $x^{(i)}$ y etiqueta $y^{(i)}$, como se observa en la Ecuación 2.3.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.3}$$

El gradiente descendiente por lote realiza iteraciones redundantes para grandes conjuntos de datos, así como re calcula gradientes para muestras similares, antes de cada actualización de parámetros. SGD termina con esta redundancia al realizar una actualización a la vez. Por lo tanto, es más rápido y también puede ser usado para aprendizaje en línea. SGD realiza actualizaciones frecuentes con una alta varianza que ocasiona una alta fluctuación en la función objetivo.
Mientras el gradiente descendiente por lotes converge a un mínimo de los parámetros colocados, las fluctuaciones en SGD permiten saltar hacia un mejor mínimo local; por otra parte las fluctuaciones también representan un problema, que complica la convergencia hacia un mínimo exacto; se ha visto que el tiempo de convergencia de SGD es muy largo comparado con otros métodos.

Métodos de optimización libres de gradiente

Los métodos de optimización basados en gradiente son eficientes y ampliamente usados; por definición dichos métodos necesitan calcular derivadas de manera iterativa. Para algunos casos en los que se estudian funciones objetivo discontinuas, no es posible calcular sus derivadas, por lo que se usan métodos alternativos que no necesitan calcularlas, como los que se mostrarán a continuación.

A) Método de Nelder–Mead

Este método consta de un algoritmo descendiente simple, desarrollado en 1965 por *Nelder* y *Mead*. Este algoritmo busca el mínimo o el máximo de una función objetivo en el espacio n-dimensional.

Primer paso

Se construye un conjunto inicial *n-simplex* con $n + 1$ vértices y se evalúa la función objetivo en los vértices; una vez hecho esto, los vértices se re ordenan de menor a mayor, lo que equivale a decir de mejor a peor desempeño, respectivamente, obteniendo un reagrupamiento como se muestra en la Ecuación 2.4.

$$f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{n+1}) \tag{2.4}$$

Segundo paso

Se calcula el centroide x del actual arreglo *n-simplex*, excluyendo al último vértice x_{n+1}, mediante la Ecuación 2.5

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.5}$$

Tercer paso

Usando el centroide como punto de referencia, se buscará el punto de reflexión del peor vértice x_{n+1} mediante la Ecuación 2.6.

$$x_r = \bar{x} + \alpha(\bar{x} - x_{n+1}) \quad (\alpha > 0) \tag{2.6}$$

La aceptación o el rechazo de la solución y la manera en que se actualizan los nuevos vértices depende directamente de la función objetivo en el punto x_r, de la cual se derivan tres posibilidades.

• Si $fx_1 \leq f(x_r) < f(x_n)$, entonces reemplaza el peor vértice x_{n+1} por x_r

• Si $f(x_r) < f(x_1)$, significa que el objetivo se mejoró, por lo tanto se busca realizar un movimiento con la intención de mejorar aún más la respuesta, moviendo o expandiendo el vértice hacia adelante a lo largo de la línea de reflexión hacia una nueva solución de prueba, como se observa en la Ecuación 2.7.

$$x_e = x_r + \beta(x_r + \bar{x}) \tag{2.7}$$

Donde $\beta = 2$. Posteriormente, al comprobar la solución de prueba, si $f(x_e)$ ha mejorado aún más. Si $f(x_e) < f(x_r)$, entonces se acepta y se actualiza el vértice x_{n+1} asignándole el valor de x_e; si el resultado no mejoró entonces se actualiza el valor de x_{n+1} asignándole el valor de la reflexión x_r.

• Si no hay mejora, si $f(x_r) > f(x_n)$, entonces es necesario reducir el tamaño de los vértices, manteniendo sin cambios a los mejores; a este proceso se le conoce como contracción. En la Ecuación 2.8 se observa su expresión matemática.

$$x_c = x_{n+1} + \gamma(\bar{x} - x_{n+1}) \tag{2.8}$$

Donde $0 < \gamma < 1$, cuyo valor típico es de $1/2$. Si $f(x_c) < f(x_{n+1})$, entonces se actualiza x_{n+1} x_c.

Si todos los pasos fallan, entonces se necesita reducir las dimensiones de los vértices hacia el mejor vértice x_1. La reducción se lleva a cabo mediante la Ecuación 2.9.

$$x_i = x_1 + \delta(x_i - x_1) \quad (i = 2, 3, ..., n + 1) \tag{2.9}$$

Una vez actualizados los vértices, se regresa al primer paso comenzando el proceso iterativo de optimización.

b) Algoritmo genético

El algoritmo genético, es un modelo de comportamiento inspirado en la teoría evolutiva de *Charles Darwin* llamada la selección natural; este modelo fue desarrollado en la década de 1960 por *John Holland* y sus colaboradores. El algoritmo genético es un algoritmo del tipo evolutivo ampliamente usado en optimización y aunque muchas de sus características son bio-inspiradas, algunas de ellas están basadas en algoritmos precedentes.
La metodología de este modelo comienza con la codificación de la función objetivo como arreglos de bits o caracteres, los cuales representan a los cromosomas. Posteriormente, mediante operadores genéticos se editan los arreglos de bits o caracteres para finalmente seleccionar los miembros mejor adaptados para dar solución al problema dado.
El algoritmo se ejecuta típicamente en la siguiente secuencia de pasos [18].

Paso 1

Codificación de las soluciones en arreglos de caracteres.

Paso 2

Se define una función de adaptabilidad y un criterio de selección.

Paso 3

Se crea una población de individuos y se evalúa su adaptabilidad.

Paso 4

Se induce la evolución en la población generando nuevas soluciones mediante el cruce, la mutación y reproducción proporcional a la adaptabilidad.

Paso 5

Se seleccionan nuevas soluciones dependiendo de su adaptabilidad y se reemplaza la población antigua por mejores individuos.

Paso 6

Se decodifican los resultados para obtener la solución.

2.2. Inteligencia artificial y *Big Data*

En el campo de las Ciencias de la Computación la *AI* está aprovechando las características de la nueva tendencia de la información *Big Data*; en conjunción ambas tecnologías tienen el potencial de ser una herramienta que ayude a la humanidad a resolver algunos de los problemas que actualmente enfrenta. *Machine Learning* desde sus inicios tiene un enfoque algorítmico y estadístico, pero en ese entonces no se disponía de suficiente información para poder explotar eficientemente sus capacidades; en la Tabla 2.2, se observa una comparación cualitativa de la *AI* antes y después de tener *Big Data*.

AI antes de *Big Data*	*AI* con *Big Data*
Disponibilidad limitada de conjuntos de datos	Disponibilidad creciente de conjuntos de datos
Modelos que no son capaces de analizar datos en tiempos cortos	Modelos con capacidad de análisis de datos en milésimas de segundo
Modelos enfocados a un conjunto específico de datos	Modelos enfocados a operar en tiempo real
Curva de aprendizaje lenta	Curva de aprendizaje acelerada
Fuentes de datos limitadas	Múltiples y heterogéneas fuentes de datos
Limitación en el tamaño de muestra	Tamaños de muestra masivos, que dan lugar a un modelo más preciso
Modelos con baja exactitud	Modelos precisos con el enriquecimiento proporcionado por grandes conjuntos de datos.
Modelos basados mayormente en bases de datos estructuradas	Modelos basados en bases de datos estructuradas, no estructuradas y semi-estructuradas

Tabla 2.2: Capacidades de modelos de *AI*, antes y después de *Big Data*. Tomado de [2].

Big Data ha sido posible gracias a las nuevas metodologías de almacenaje de información; un ejemplo de ello es *IoT*, cuyo funcionamiento se basa en la recopilación y análisis de información.

La tecnología *Big Data* no significa mejores algoritmos de *AI*, pero sí mas información. El término *Big Data*, hace referencia a grandes volúmenes de información, siendo el volumen y tres puntos más los que describen cualitativamente las ventajas de *Big Data*, como se muestra a continuación.

Volumen:

El volumen representa la cantidad de información que se encuentra en continuo crecimiento. La democratización de las tecnologías de la información, ha permitido que se pueda recopilar información de diferentes fuentes, a través de diferentes interfaces humano-máquina.

Velocidad:

La velocidad hace referencia a la cantidad de información generada con respecto al tiempo, y la necesidad de analizar dicha información conforme es generada, con la finalidad de realizar alguna tarea critica.

Diversidad:

Diversidad, específicamente en el formato de los datos. Actualmente se cuenta con una gran variedad de formatos, tales como imágenes, vídeos, archivos de audio etc.

Valor:

La información solamente es valiosa cuando es usada para generar conocimiento que lleve a realizar una acción.

Las nuevas tecnologías, ahora se encuentran centradas en la recopilación de datos para su análisis; en la Figura 2.1, se ilustra la tendencia creciente de la generación de información conforme avanza el tiempo.

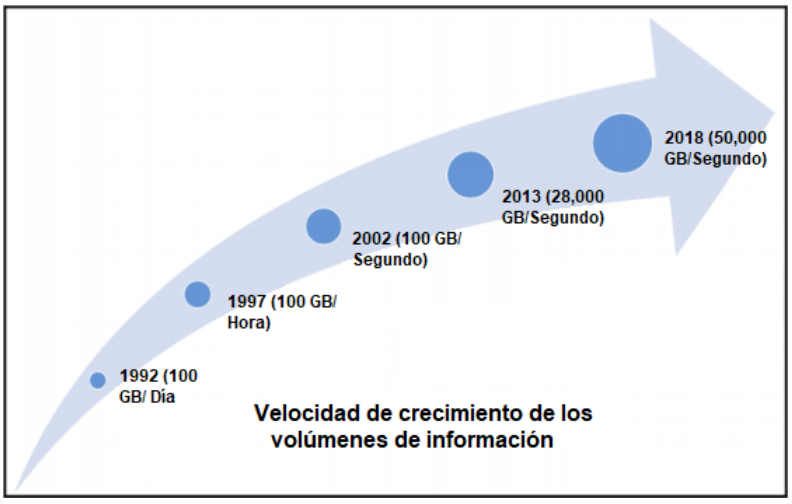

Figura 2.1: Tendencia de crecimiento de los volúmenes de información. Adaptado de [2]

Un ejemplo de la importancia de *Big Data* es la tecnología 5G. Esta tecnología de comunicación incrementa las capacidades de transferencia de información; algunos autores mencionan "*5G no es para ti*", esto se basa en el hecho que el 5G estará involucrado mayormente en máquinas, cuyas áreas de aplicación son la industria inteligente, *IoT*, navegación autónoma entre otras, las cuales harán uso de las características de dicha tecnología.

2.3. Propiedades de redes neuronales

El cerebro humano consta aproximadamente de diez billones de células nerviosas densamente interconectadas, llamadas neuronas, cada neurona tiene conexión con alrededor de 10E3 neuronas, con 60 trillones de conexiones sinápticas entre ellas.

El cerebro humano es la fuente de inspiración para la creación de modelos cognitivos computacionales; su funcionamiento resulta más rápido y eficiente que los procesadores neuromórficos actuales. La neurona ha sido modelada como una unidad elemental de procesamiento, por lo cual el cerebro es concebido como una red biológica con un alto grado de complejidad, no lineal y paralela de procesamiento de información; en la Figura 2.2 se observan las partes que constituyen una neurona desde el punto de vista fisiológico.

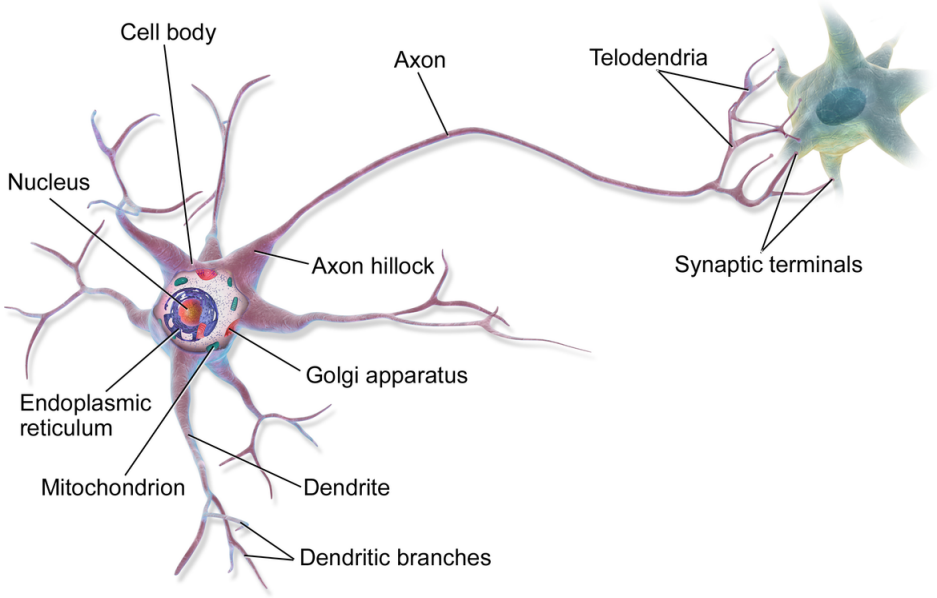

Figura 2.2: Esquema biológico de una neurona. Tomado de [17].

La manera en que las neuronas biológicas pueden aprender resulta de interés científico, por lo cual hay diversas aproximaciones matemáticas que intentan emular su comportamiento en computadoras.

En una red neuronal, la neuronas se relacionan entre sí mediante conexiones dirigidas; estos dos elementos constituyen la topología de la red. Cada conexión tiene asociado un peso, el cual representa la influencia de la conexión entre dos neuronas, el valor numérico de este peso también cuenta con un significado. Un valor mayor que cero representa refuerzo, mientras que un valor menor que cero significa inhibición. En una red neuronal artificial los pesos determinan el comportamiento de la red. Comúnmente una neurona artificial tiene varias entradas y su correspondiente salida para su función de activación. En la Tabla 2.3 se realiza un análisis comparativo entre los elementos constituyentes de una neurona biológica y artificial.

Neurona biológica	Neurona artificial
Soma	Suma + Función de activación
Dendrita	Entrada
Axón	Salida
Sinapsis	Peso

Tabla 2.3: Tabla de equivalencias entre modelos biológicos y numéricos de una neurona.

En la Ecuación 2.10 se observa el modelo matemático de una neurona artificial cuyo esquema se ilustra en la Figura 2.3.

$$Y = \Phi\left(\sum_{i=1}^{n} x_n * w_n + \theta\right) \tag{2.10}$$

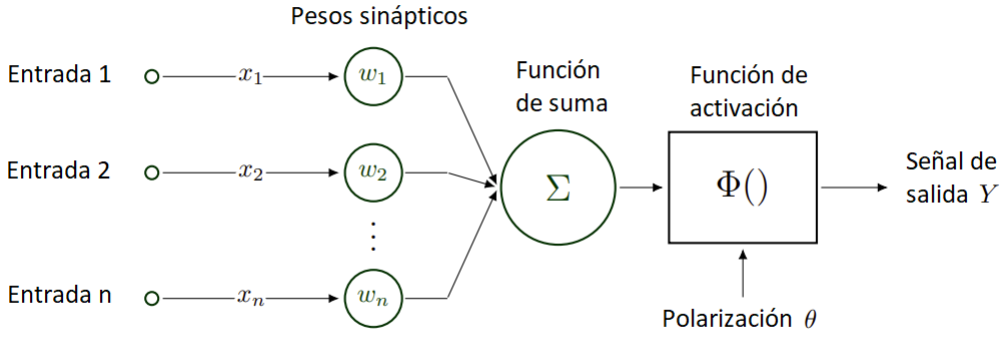

Figura 2.3: Modelo matemático de una neurona artificial.

Las propiedades de las redes neuronales artificiales pueden sintetizarse en cuatro puntos, los cuales se exponen a continuación:

Propiedades de red

Una red neuronal es una arquitectura la cual está constituida por un conjunto de neuronas que operan conjuntamente para responder a una entrada. Las neuronas se encuentran organizadas por niveles jerárquicos; cada nivel es llamado capa. En modelos ya construidos al presentar una entrada a la red, se obtiene una salida, de esta manera no se observa el rol que desempeñan las capas internas, por lo que son consideradas capas ocultas. Las redes neuronales son divididas en dos categorías: si la arquitectura tiene ciclos dirigidos, se le denomina *recurrente*, si no tiene, se considera de *alimentación hacia adelante*.

Propiedades de la neurona

Cada neurona calcula una sola salida ó activación. Las entradas y activaciones pueden ser discretas, continuas o continuas por intervalos; para ejemplificar lo ya mencionado, en la Ecuación 2.11 se observa la función de activación de escalón binario, la cual tiene valores de salida discretos. En la Ecuación 2.12 se observa la función de activación sigmoide, la cual es una función real con valores de salida continuos y finalmente, en la Ecuación 2.13 se observa la función de activación llamada unidad lineal de rectificación, la cual tiene valores de salida continuos por secciones.

$$\textit{Escalón binario} \quad \phi(x) = \begin{cases} 0 & \text{para } x \leq 0 \\ 1 & \text{para } x > 0 \end{cases} \tag{2.11}$$

$$\textit{Sigmoide} \quad \phi(x) = \frac{1}{1+e^{-x}} \tag{2.12}$$

$$\textit{Unidad lineal de rectificación} \quad \phi(x) = \begin{cases} 0 & \text{para } x < 0 \\ x & \text{para } x \geq 0 \end{cases} \tag{2.13}$$

La activación de una neurona se obtiene de las activaciones de las neuronas con las que se encuentra relacionada, multiplicadas por los pesos de sus respectivas conexiones. Todas las neuronas (con excepción de las neuronas de entrada) calculan una activación como la suma de las activaciones, presentando una nueva activación a las neuronas del siguiente nivel jerárquico.

Propiedades dinámicas:

Un modelo neuronal debe contener un cierto grado de sincronía; cada capa, y proceso de cálculo tienen que ejecutarse de manera subsecuente, con el fin de obtener una salida consistente con el modelo. En los modelos de *Alimentación hacia adelante*, cada neurona calcula su salida en un orden fijo; cada neurona actualiza su activación antes de que la otra comience el proceso de cálculo. En este caso el sistema logra un comportamiento estacionario con el fin de lograr un ordenamiento coherente de los procesos de cálculo.

Propiedades cognitivas

Todas las redes neuronales necesitan ser entrenadas para que respondan a las entradas. Dicho proceso está asociado con un algoritmo de *Machine Learning*. Las técnicas de entrenamiento en *Machine Learning* están dividas en aprendizaje *supervisado* y *no supervisado*. En el aprendizaje supervisado, la metodología de entrenamiento está basada en un *maestro* y un *ejemplo*, el *maestro* es una métrica que evalúa de qué manera se está desempeñando el modelo; por otra parte, el *ejemplo* consta de una rutina en la que se muestra de qué manera se espera que el modelo responda a cierto tipo de estímulo. En el aprendizaje *no supervisado* no existe ninguna métrica que evalué el desempeño del modelo, solo una medida que estima la calidad de la reconstrucción de la entrada; en estos modelos se suele esperar un alto grado de generalización que extraiga el mayor número de características del sistema a estudiar[19].

Propiedades descriptivas

La descripción de un modelo de *Machine Learning* en lenguajes de cómputo cuenta con diversas propiedades, Las distintas metodologías empleadas para construir modelos cognitivos difieren dependiendo del software que se use. En el caso de *MATLAB*© y *Python* ambos tratan los modelos y sus propiedades como objetos; por otra parte, la metodología de procesamiento en MATLAB, se realiza desde el punto de vista matricial, en contraste con *Python*, cuyo uso del paquete *TensorFlow*[16] permite realizar el procesamiento mediante tensores, un tipo de grafos que se han empleado por su bajo coste de cómputo. Con el fin de introducir al lector en el entorno de programación usado en este trabajo, se explicarán algunos conceptos empleados en la descripción de una red neuronal y paralelamente se construirá y entrenará un modelo en lenguaje *Python*. Dicho modelo es una red neuronal para identificación de números escritos, mediante imágenes como la que se muestra en la Figura 2.4, el cual fue adaptado del sitio de Aprendizaje de *TensorFlow* [14],[15].

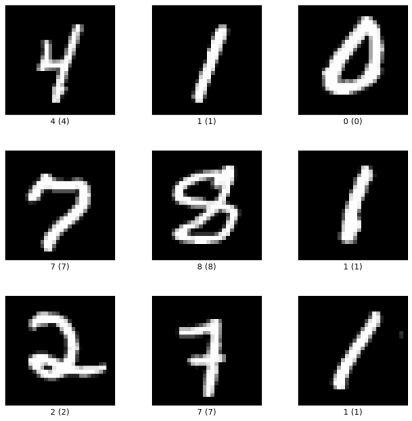

Figura 2.4: Dígitos escritos a mano de la base de datos MNIST. Tomado de [14].

Inclusión de bibliotecas

En la Sección de código 2.1 se muestra la inclusión de la biblioteca *TensorFlow*, la inclusión de conjuntos de datos de *TensorFlow* y la conversión de números enteros a flotante, los cuales fueron extraídos de la clase `mnist` (Variable a la que fueron transferidos los atributos de la base de datos) mediante el atributo `.load_ data()`.

```
   import tensorflow as tf # Se importa la bilbioteca con la
       sentencia ‘‘import’’ y se le asigna el nombre ‘‘tf’’ al objeto
       que guardara sus atributos
   mnist = tf.keras.datasets.mnist # Carga el conjunto de datos de la
       localidad ‘‘mnist’’
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0 # Los numeros
    enteros resguardados en el conjunto de datos se convierten a
    punto flotante
```

Sección de código 2.1: Inclusión de bibliotecas, base de datos y adaptación de la información.

Modelo

La creación de un modelo, es básicamente la inclusión de una clase alojada en una biblioteca; a continuación se observa la descripción de esta rutina en Sección de código 2.2.

```
   model = tf.keras.models.Sequential()
```

Sección de código 2.2: Creación del modelo.

En este ejemplo, la declaración `models.Sequential()` extrae la clase correspondiente a una arquitectura secuencial; el atributo `Sequential()` permite crear una arquitectura por capas apiladas. Todos los tributos de la clase ya mencionada son transferidos al objeto `model`.

Capas

Las capas en una arquitectura de *Machine Learning* representan un conjunto de neuronas, las cuales se relacionan con otras neuronas de otras capas, o entre sí mismas, dependiendo de la topología empleada. En la Tabla 2.4, se muestran algunos ejemplos del tipo de capas así como las arquitecturas en las que se emplean.

Tipos de capa	Ejemplos	Arquitecturas que las emplean
Core layers	*Dense layer*, *Activation layer*	*Neural Networks*
Convolution layers	*Conv1D layer*, *Conv2D layer*	*Convolutional Neural Networks*
Pooling layers	*MaxPooling1D layer*, *MaxPooling2D layer*	*Convolutional Neural Networks*
Recurrent layers	*LSTM layer*, *GRU layer*	*Recurrent neural networks*
Preprocessing layers	*Text preprocessing*, *Numerical features preprocessing layers*	*Generative adversarial networks*

Tabla 2.4: Tabla de tipos de capa.

La declaración y apilamiento de las capas se realizan de diferentes maneras, en la Sección de código 2.3 se ejemplifica la creación y apilamiento de capas mediante el atributo `add`.

```
 # Configuracion de las dimensiones de entrada
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
# Primera capa neuronal
model.add(tf.keras.layers.Dense(128, activation='relu'))
# Segunda capa neuronal
model.add(tf.keras.layers.Dropout(0.2))
# Tercera capa neuronal
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

Sección de código 2.3: Adición de las capas.

Cada capa cuenta con parámetros como dimensiones de entrada, número de neuronas y función de activación.

Compilación

La compilación del modelo se lleva a cabo mediante el comando `compile`, esta sentencia direcciona hacia una clase en la que es necesario especificar parámetros de entrada como función de optimización y función de pérdida, como se ejemplifica en la Sección de código 5.7. El proceso de compilación permite que el modelo ya esté listo para entrenar.

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

Sección de código 2.4: Compilación del modelo.

Entrenamiento

La etapa de entrenamiento se realiza una vez que el modelo ya ha sido compilado pues, el modelo ya tiene definidos los criterios bajo los cuales será evaluado. En la Sección de código 2.5 se muestra la descripción del entrenamiento.

```
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test, verbose=2)
```

Sección de código 2.5: Entrenamiento del modelo.

En la Sección de código 2.5 se observa que, para realizar el entrenamiento, es necesario transferir los parámetros de entrenamiento como lo son datos de entrenamiento, datos de prueba y número de épocas, estas últimas hacen referencia al número de veces que el modelo se entrenará, re-usando el aprendizaje de los entrenamientos previos, una vez los datos de entrenamiento y prueba hayan sido usados completamente.
Durante el entrenamiento se obtiene una salida que reporta los valores de desempeño de la red, como tiempo de cálculo, valores de pérdida de entrenamiento, exactitud y el número correspondiente a la época que se está ejecutando. En la Sección de código 2.5 se ejecutan dos comandos, el primero con extensión `.fit` genera los primeros 11 renglones que se observan en la Figura 2.5, reportando los resultados obtenidos en cada época iterada; el segundo comando con extensión `.evaluate` genera el último renglón con los valores finales de pérdida y aserción del modelo.

```
Epoch 1/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0662 - accuracy: 0.9786
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0570 - accuracy: 0.9815
Epoch 3/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0526 - accuracy: 0.9827
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0477 - accuracy: 0.9844
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0438 - accuracy: 0.9850
313/313 - 0s - loss: 0.0689 - accuracy: 0.9792 - 450ms/epoch - 1ms/step
[0.06887895613908768, 0.979200005531311]
```

Figura 2.5: Salida obtenida del entrenamiento.

El valor de pérdida depende de la función objetivo configurada, en este ejemplo se usó la función entropía cruzada categórica, la cual calcula la pérdida mediante distribuciones de probabilidad. Su expresión matemática se observa en la Ecuación 2.14.

$$Loss = -\sum_{i=1}^{n} y_i * Log(\hat{y}_i) \tag{2.14}$$

Donde n es el número de eventos, y_i es la distribución de probabilidad real de las clases y $\hat{y}_i$ es la distribución de probabilidad generada por el modelo. Su valor de salida ideal es de `0`; en nuestro ejemplo el valor obtenido fue de `0.067`. La precisión (el segundo valor obtenido de la evaluación), se calcula de manera simple mediante la Ecuación 2.15.

$$Accuracy = \frac{\sum_{i=1}^{n} p_i}{n} \tag{2.15}$$

Donde n es el número de elementos a evaluar y p_i es 1 para valores calculados correctos y 0 para valores incorrectos.

Una vez entrenados los modelos se puede acceder a la información del mismo, mediante comandos, en la Tabla 2.5 se enlistan algunos de los atributos de dicho modelo.

Comando	Descripción
`.get_layer`	Este atributo regresa un objeto con la propiedades de la capa especificada.
`.load_weights`	Carga una variable con una *lista* que contiene las matrices de pesos y polarizaciones de la red.
`.predict`	Predice una salida para una entrada proporcionada, de un modelo previamente entrenado.

Tabla 2.5: Extracción de parámetros mediante atributos.

2.4. Esquemas *IoT*

El esquema del Internet de las Cosas (**IoT**, por sus siglas en inglés) es visto frecuentemente como un cambio del internet tradicional, hacia un escenario donde todas las “cosas"se encuentran conectadas a través de internet. *IoT* ha forzado el desarrollo de nuevos estándares *IoT* para incluir dispositivos y protocolos heterogéneos bajo una misma normatividad, para una interacción armoniosa entre ellos.
El IoT es una nueva tendencia en computación con una filosofía alternativa a la tradicional, permitiendo interacciones entre el mundo físico y cibernético. El término fue acuñado en el 2009, por Kevin Ashton, cuando presentaba una propuesta para la cadena de suministros de la empresa P&G©, en la cual él argumentaba que colocar etiquetas de radiofrecuencia y algunos sensores a los productos de la empresa, permitiría monitorearlos mediante la fusión de dos cadenas, la de suministro y la de bits(Internet).
Esta persona hacía énfasis en la importancia de las tecnologías, pero aún más la importancia de los seres humanos y las cosas, siendo las tecnologías de la información un vínculo entre ellos.

En 1991, Mark Weiser, creó una hipótesis de la orientación de mecanismos implícitos de la tecnología basado en acciones, estímulos y reacciones; de esta manera, se puede actuar o reaccionar a ciertos eventos o estímulos.

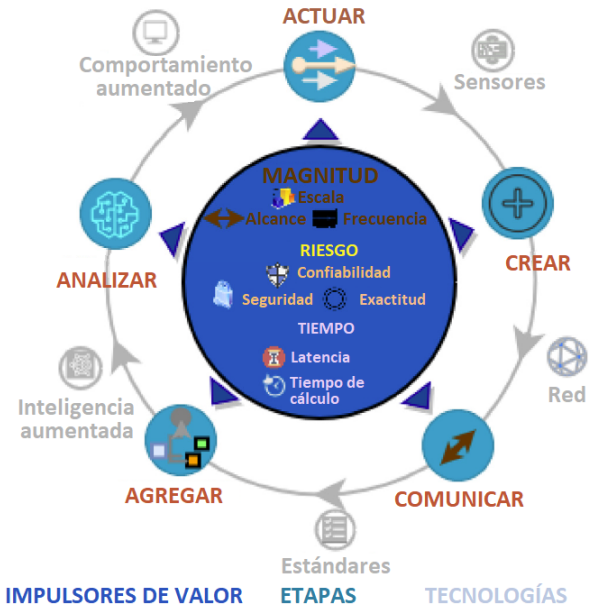

Figura 2.6: Modelo de Weiser.

Weiser describió la "*computación ubicua*" como: "Un ecosistema con objetos de todo tipo que tienen la capacidad de sentir de manera autónoma, comunicándose, analizando y actuando/reaccionando a las personas y a otras máquinas", y propuso "el modelo de Weiser", para mostrar los mecanismos involucrados en el proceso [12]. En la Tabla 2.6 se observa un esquema que plasma el proceso del *IoT*, en el cual la información pasa a través de varias etapas.

Crear →	Comunicar →	Agregar →	Analizar →	Actuar →
Sensores	Red	Integración	Inteligencia aumentada	Comportamiento aumentado
Dispositivos físicos - Sensores	**Conectividad** - Elementos de procesamiento - Elementos de conectividad - Estructuras de conectividad	**Administración** - Software intermedio **Acumulación** - Ingesta de datos - Almacenamiento de datos - Abstracción de datos **Información existente** - Información estructurada - Información no estructurada	**Procesamiento** - Unidades de procesamiento / Estructuras - Mensajería de datos **Análisis** - Análisis de transmisión - Análisis de banco, *Machine learning*	**Aplicaciones** - Aplicaciones de usuario - Reporte ***Edge Computing*** - Compuertas inteligentes - Computación en niebla
Estándares				
Seguridad				

Tabla 2.6: Adquisición y tratamiento de la información, según el esquema *IoT*. Adaptado de [5]

En la Tabla 2.6 se observan principalmente cinco etapas; en la primera se encuentra la etapa de sensado, la cual algunos autores describen como la conexión entre el mundo físico y el cibernético, la segunda es la etapa de comunicación, en esta etapa es donde se transmiten los datos recopilados a través de Internet, mediante algún protocolo de Internet(Wi-Fi, Ethernet, 3G, etc.); la tercera etapa consiste en el respaldo remoto de la información en alguna localidad; la cuarta etapa es la etapa de análisis, en esta etapa los datos son procesados con el fin de extraer información; la quinta etapa consiste en la reacción inteligente y es la etapa donde se genera una realimentación a la información proporcionada.

2.4.1. Sistemas de monitoreo

El *IoT* ha sido incluido en diversos entornos, algunos de ellos son: dispositivos móviles, el hogar, industrias, entre otras. Este trabajo se enfoca en proponer un sistema de monitoreo para la industria, aunque podría extenderse a áreas como el hogar y los automóviles.

Las tecnologías IoT involucran servicios de hardware y software; el uso de estos sistemas se ha

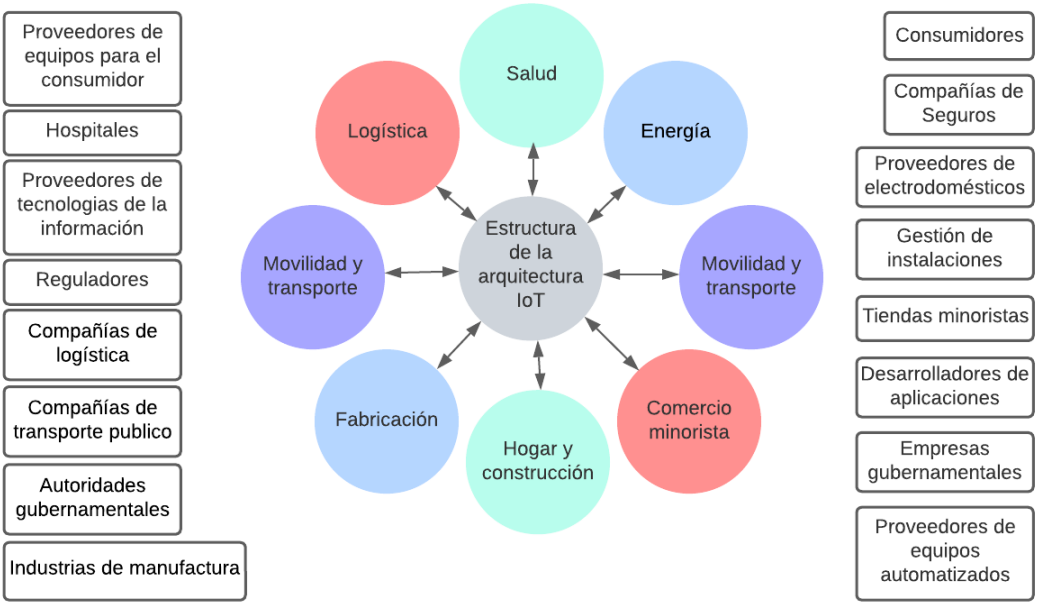

Figura 2.7: Áreas de aplicación del IoT.

incrementado rápidamente en los últimos años. La reparación automatizada, una aplicación de IoT, es una tendencia que han adoptado múltiples industrias en distintas áreas. Un ejemplo de ello es el monitoreo de la salud estructural en infraestructuras criticas; dichos sistemas ayudan a salvar vidas y ahorrar dinero, con una inversión significativa.

Un ejemplo de la importancia de los sistemas IoT, es el caso de la localización de hospitales dedicados a tratar el COVID-19; en la ciudad de México entidades de gobierno y desarrolladores independientes se dieron a la tarea de implementar sistemas *IoT* para proporcionar información a la población acerca de hospitales disponibles, a través de una aplicación que determinaba el lugar más conveniente en función de la ubicación, dada la alta demanda de servicios médicos.

2.5. Conclusiones del capítulo

En este capítulo se explicaron algunos de los conceptos fundamentales en el campo del aprendizaje en máquina, la cual se encuentra íntimamente asociada con las ciencias de la computación, siendo ambas, unas de las ramas más nuevas de estudio que existen en la actualidad. Se puede concluir que el aprendizaje en maquina pretende emular el proceso cognitivo presente en la naturaleza, mediante modelos matemáticos cuyas metodologías pueden variar entre sí, dando lugar a dos principales ramas del aprendizaje en máquina, el supervisado y no supervisado, cada una de las cuales con características adecuadas para la solución de diferentes problemas. Con la rápida expansión de las ciencias de aprendizaje en máquina, también han surgido herramientas de computo, que hacen posible la creación de modelos neuronales, como los son *MATLAB* y *Tensor Flow* cuyas metodologías de descripción son diferentes así como sus procesos numéricos de computo.
Con la democratización de las tecnologías, se ha logrado un incremento en la captura de información, pues esta es obtenida casi de cualquier parte del mundo, dando lugar a la ciencia de datos conocida como *Big Data*, de esta manera se puede concluir que la gran cantidad de información disponible en la actualidad ha permitido el surgimiento de nuevas tendencias tecnológicas como lo son IoT y aprendizaje en máquina, las cuales aprovechan las ventajas de conectividad y almacenaje de datos inherentes de Internet y le dan una aplicación útil para la sociedad.

CAPÍTULO 3

Metodología de reducción de dimensión y clasificación

En años recientes, grandes conjuntos de datos como imágenes, audio y señales, se han convertido en un estándar; uno de los problemas que han surgido de la manipulación de grandes conjuntos de datos, se conoce como *maldición de dimensionalidad*. Dado que el volumen del espacio de datos se incrementa exponencialmente rápido con la dimensión, se tiende a perder información[5].

La reducción de dimensionalidad es un paso de pre-procesamiento común para clasificación. Debido a que el aprendizaje para entradas de baja dimensionalidad es rápido, la reducción de dimensionalidad puede ayudar principalmente a tener un clasificador mejor entrenado, esto se logra gracias al efecto regulador de esta técnica, que puede evitar el sobre-ajuste. La reducción de dimensionalidad elimina dos tipos de ruido aleatorio.

Ruido independiente aleatorio

Este ruido se presenta en todas las entradas y no tiene correlación entre las entradas y sus respectivas etiquetas.

Grados de libertad no deseados

Estos factores posiblemente no sean lineales, a lo largo del cual la entrada cambia pero la etiqueta no lo hace [5].

Ventajas

La reducción de dimensionalidad es un proceso que reduce el número de variables aleatorias de estudio, y es importante en el análisis de grandes conjuntos de datos; algunas de las ventajas de usar esta técnica son:

- Evitan la reducción de eficiencia con datos de alta dimensionalidad, ya que la eficiencia y la exactitud decaen rápidamente conforme la dimension se incrementa.
- Bajo consumo computacional.
- Ayuda a evitar el sobre-entrenamiento.

Desventajas

- Son algoritmos altamente especializados.
- Cuellos de botella estrechos pueden ocasionar que el modelo ignore información importante.

Muchos métodos tradicionales de reducción de dimensionalidad han sido ampliamente usados, entre ellos los métodos basados en *Singular Value Decomposition* (*SVD*, por sus siglas en inglés), incluidos *Principal Component Analysis* (*PCA*, por sus siglas en inglés) e *Isomap*. Existen muchos algoritmos para implementar esta técnica; los cuales pueden ser divididos en dos categorías.

Extracción de características:
En esta metodología, todas las variables son empleadas y el conjunto de datos es convertido mediante transformaciones lineales en un conjunto con dimensiones espaciales reducidas.

Selección de características.
Esta metodología es empleada para encontrar el subconjunto mas óptimo e importante de características y usualmente elimina los datos irrelevantes y redundantes.

Con la introducción de las redes neuronales artificiales, se ha propuesto una arquitectura llamada *Autoencoder*, la cual puede desempeñar el papel de algoritmo de reducción de dimensionalidad, y ha sido adoptada rápidamente debido a su alto grado de flexibilidad.

3.1. Red Autoencoder

Los autoencoder juegan un papel fundamental en el ramo del aprendizaje no supervisado y arquitecturas de *Deep Learning* para transferencia de aprendizaje así como otros temas.

Los *Autoencoders* son redes de aprendizaje simples, las cuales tienden a transformar entradas en salidas con la menor distorsión posible. Estos fueron introducidos por primera vez en la década de 1980, enfocados hacia el problema *"backpropagation without a teacher"*, usando la entrada como función objetivo.

Junto con las reglas de aprendizaje tipo Hebbian, los autoencoders proveen uno de los paradigmas fundamentales para el aprendizaje no supervisado y para comenzar a considerar los fundamentos de cómo los cambios sinápticos inducidos por eventos bioquímicos locales, pueden ser coordinados de una manera auto organizada, para producir un aprendizaje global y un comportamiento inteligente. Los autoencoders han retomado nuevamente su papel central, como *arquitectura profunda*.

3.1.1. Tipos de autoencoder

El estudio de los autoencoders ha sido parte del panorama de las redes neuronales por décadas. Tradicionalmente los autoencoders eran empleados para tareas de reducción de dimensionalidad y extracción de características. Por razones conceptuales, la teoría de los autoencoders ha convergido con arquitecturas avanzadas recientemente introducidas como el campo de *Generative Adversarial Networks* (*GAN*, por sus siglas en inglés). El modelado de los autoencoders ha sido adoptado por múltiples ramas de la *AI*, con diferentes aplicaciones, métodos de entrenamiento y arquitecturas, con lo que se puede enfatizar la importancia del aprendizaje no supervisado, como lo menciona en un artículo publicado en la revista *Forbes©*, el *CEO* de *Helm.ai©*, una empresa dedicada a descubrir modelos de negocio usando herramientas de *Machine Learning*, "el futuro de la *AI* es no supervisado" [4]. Con el fin de expandir la idea del autoencoder al lector, a continuación se explicarán los fundamentos de algunas de las arquitecturas mas empleadas en la literatura [6].

Autoencoder elemental

La Arquitectura de un autoencoder elemental lo constituye una capa de entrada, una capa oculta y una de salida; en la Figura 3.1, se ilustra dicha arquitectura.

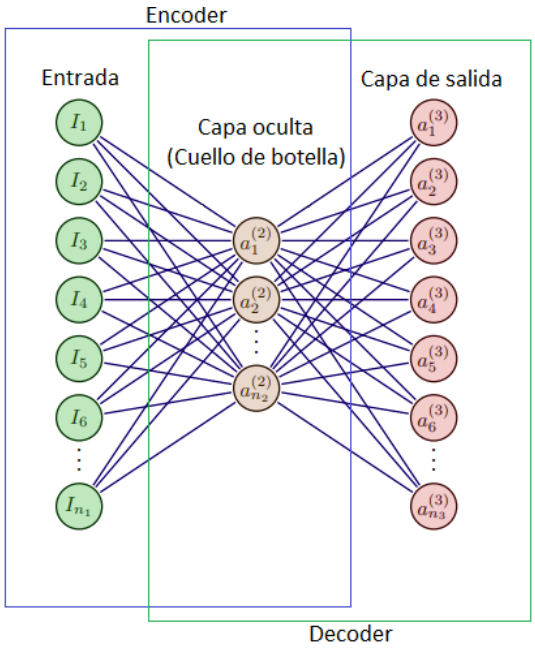

Figura 3.1: Topología de un autoencoder simple.

La expresión matemática de la arquitectura se muestra en la ecuación 3.1.

$$\hat{x} = g(f(x)) \tag{3.1}$$

Donde

$f(x)$ es la función vectorial de extracción de características, también conocida como representación oculta $\boldsymbol{h}$, la cual es descrita en la Ecuación 3.2.

$$\boldsymbol{h} = f(x) = s_f(\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}_h) : \mathbb{R}^n \longrightarrow \mathbb{R}^p \quad \text{con } n > p \tag{3.2}$$

Donde
$s_f()$ es la función de activación empleada en la capa oculta, $\boldsymbol{W}^1$ es la matriz de pesos sinápticos que relaciona a la entrada con la capa oculta, $\boldsymbol{x}$ es el vector de entrada, $\boldsymbol{b}_h$ es el vector de polarizaciones de la capa oculta, n es el tamaño de entrada y p es número de neuronas en la capa oculta.

$\hat{x}$, es la reconstrucción de la entrada dada por la función de decodificación $g(f(x))$, cuyas expresiones se correlacionan, como se muestra en la Ecuación 3.3.

$$\hat{x} = g(h) = s_g(\boldsymbol{W}^2\boldsymbol{h} + \boldsymbol{b}_y) : \mathbb{R}^p \longrightarrow \mathbb{R}^n \tag{3.3}$$

Donde
$s_g()$ es la función de activación empleada en la capa de salida, $\boldsymbol{W}^2$ es la matriz de pesos sinápticos que relaciona a la capa oculta con la capa de salida, $\boldsymbol{h}$ es la representación oculta de $\boldsymbol{x}$ y $\boldsymbol{b}_y$ es el vector de polarizaciones de la capa de salida.

Autoencoder profundo

Un autoencoder profundo está constituido por dos ó más capas ocultas; esta arquitectura tiene la ventaja de obtener una mejor reconstrucción a la salida, así como una mejor extracción de características, esto se debe a la transferencia de aprendizaje que se lleva a cabo entre las capas ocultas, de manera jerárquica. Su arquitectura general se muestra en la figura 3.2.
Un autoencoder apilado puede ser modelado como una arreglo de autoencoders sucesivamente conectados, en donde cada capa puede ser la capa oculta de un autoencoder anterior y la capa de entrada de un autoencoder subsecuente.

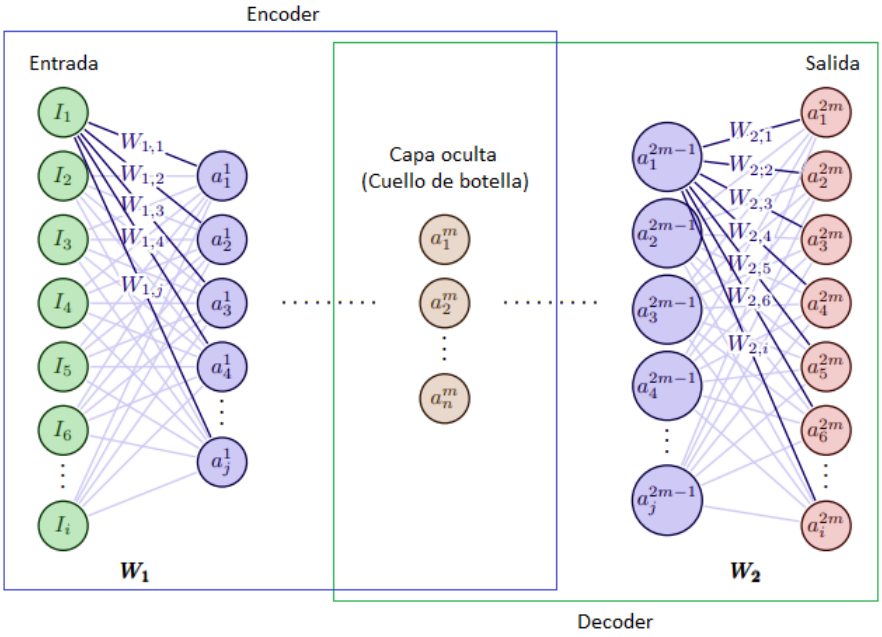

Figura 3.2: Topología de un autoencoder profundo.

En 3.5,3.6 y 3.7 se modela el conjunto de ecuaciones correspondientes a un autoencoder con n capas ocultas las cuales realizan una conversión de dimensión de las entradas $\mathbb{R}^{H_{in}} \longrightarrow \mathbb{R}^{H_{out}}$, siendo $\mathbb{R}^{H_{in}}$ la dimensión de entrada y $\mathbb{R}^{H_{out}}$ la dimensión de salida. En la Ecuación 3.7 se describe la ecuación que representa la reconstrucción cuya dimensión de entrada es la correspondiente a la *n-ésima* capa y cuya dimensión de salida es igual a la de la información de entrada.

$$h_1 = \phi^1\left(\sum_j w^1 x_j + b^1\right) : \quad \mathbb{R}^P \longrightarrow \mathbb{R}^{H_1} \tag{3.4}$$

$$h_2 = \phi^2\left(\sum_j w^2 h_{1,j} + b^2\right) : \quad \mathbb{R}^{H_1} \longrightarrow \mathbb{R}^{H_2} \tag{3.5}$$

$$\vdots$$

$$h_n = \phi^n\left(\sum_j w^n h_{n-1,j} + b^n\right) : \quad \mathbb{R}^{H_{n-1}} \longrightarrow \mathbb{R}^{H^n} \tag{3.6}$$

$$y_i = \phi^{n+1}\left(\sum_j w^{n+1} h_{n,j} + b^{n+1}\right) : \quad \mathbb{R}^{H_n} \longrightarrow \mathbb{R}^P \tag{3.7}$$

Donde $h_1 \dots h_n$ son la representación de las capas ocultas, $\phi^1 \dots \phi^n, \phi^{n+1}$ son la funciones de activación de las capas[1], $w^1 \dots w^n, w^{n+1}$ son los pesos sinápticos de las capas, $b_1 \dots b^n, b^{n+1}$ son las polarizaciones y y es la salida reconstruida.

[1]Esta notación es usada con el propósito de tener en cuenta que en las capas que constituyen la red neuronal, no necesariamente son iguales las funciones de activación.

Autoencoder supresor de ruido

Las arquitecturas de los *Denoising Autoencoders* (*DAE's*, por sus siglas en inglés) en esencia pueden tener la forma de las vistas en la Subseccion 3.1.1.1 y 3.1.1.2, sólo que en vez de agregar una penalización a la función objetivo, se puede forzar al modelo que aprenda rasgos importantes cambiando el término del error de reconstrucción de la función objetivo.
Tradicionalmente los Autoencoders minimizan una función, que tiene la forma de la Ecuación 3.8.

$$L\left(x, g(f(x))\right) \tag{3.8}$$

Donde L, es la función de pérdida que penaliza a $g(f(x))$ al ser diferente a x, así como L^2 es la norma de su diferencia. Esto obliga a $g \cdot f$ a ser simplemente una función identidad, si tienen la capacidad de hacerlo. Por otra parte los *DAE's* minimizan funciones de la forma mostrada en Ecuación 3.9.

$$L(x, g(f(\tilde{x}))) \tag{3.9}$$

Donde $\tilde{x}$ es una copia de x que ha sido corrompida con algún tipo de ruido. El proceso de supresión de ruido obliga implícitamente a g y f a aprender la estructura de los datos de entrada. En la Figura 3.3 se muestra la arquitectura de un autoencoder profundo de 5 capas, supresor de ruido.

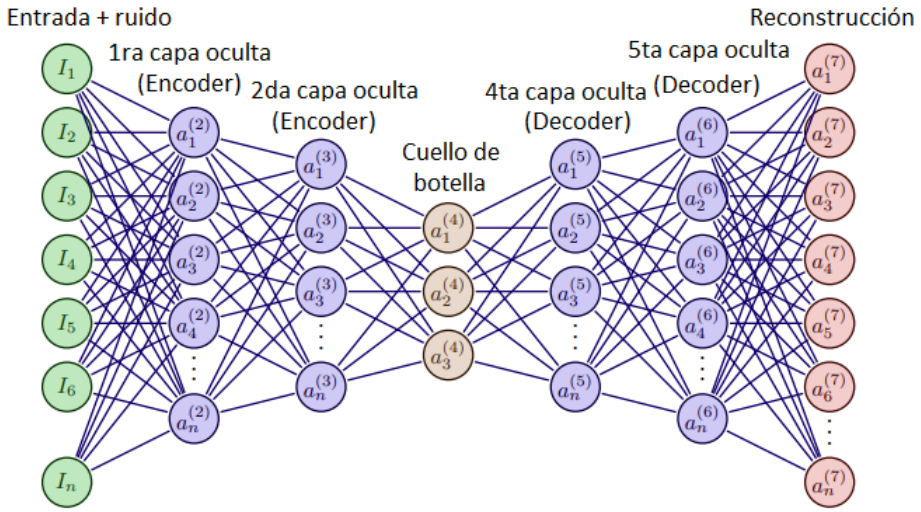

Figura 3.3: Topología de un autoencoder profundo con cinco capas ocultas, supresor de ruido.

Autoencoder convolucional

Los *Convolutional Autoencoders* (*CAEs*, por sus siglas en inglés) son un enfoque de las *Convolutional Neural Networks* (*CNN*, por sus siglas en inglés). La arquitectura convencional de una *CNN* está compuesta de las siguientes capas: convolución, *pooling*, y desconvolución, opcionalmente acompañadas de una capa *fully connected*[2]. En los autoencoders elementales, profundos y supresores de ruido se ignora la estructura de la información de entrada, forzando cada sesgo a ser global; los *CAEs* tienen la característica de reconocer sesgos localizados, cuando la información en la entrada es adaptada mediante un proceso de "suavizado" Gaussiano; dada la separabilidad de la función Gaussiana bidimensional, su convolución puede ser fácilmente calculada [8].
La operación convolucional se aplica extrayendo campos receptivos nativos alrededor de cada vector de entrada para crear un mapa de actividad de las entradas. Este mapa también se conoce como mapa de sesgos.
El operador convolucional permite filtrar una señal de entrada para extraer una fase considerable de su contenido. Los CAEs utilizan el operador convolucional para tomar ventaja del hecho de que una señal puede ser vista como una conjunto de otras señales. Los autoencoders convolucionales descubren cómo cifrar la contribución de las entradas a un conjunto de señales básicas y luego intentan rehacer la contribución de esas señales. El modelo del autoencoder se optimizará, mediante filtros adecuados a la señal de entrada, los cuales disminuyen el error en la reconstrucción. Una vez entrenado el modelo, los filtros son aplicados a la entrada para recuperar su respectiva información. En la Figura 3.4 se observa la arquitectura general de un CAE.

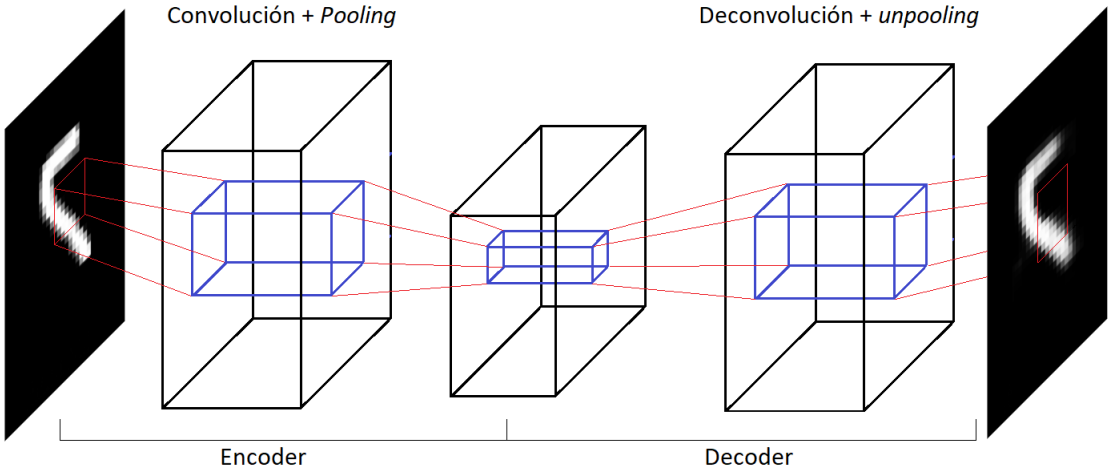

Figura 3.4: Topología de un autoencoder convolucional.

[2]Para mayor información sobre las capas de las *CNN*, véase [20, Capitulo 7]

La representación de la información interna está dada por la ecuación 3.10.

$$h^k = \psi\left(x * W^k + b^k\right) \tag{3.10}$$

Donde h^k es la información de la *k-ésima* capa, ψ es la función de activación, x es la entrada, $*$ denota el operador de convolución bidimensional, W^k es la matriz de pesos sinápticos y b^k es la polarización.
La reconstrucción de la entrada $\boldsymbol{x}$ está dada por la ecuación 3.11.

$$y^k = \psi\left(\sum_{k\in H} \boldsymbol{h^k} * \bar{W}^k + c\right) \tag{3.11}$$

Donde c representa una polarización por canal de entrada, H hace referencia al grupo de mapas de características internas, $\bar{W}$ es la operación de reconstrucción en las dos dimensiones de la matriz de pesos sinápticos.

Autoencoder variacional

El autoencoder variacional se construye mediante una arquitectura profunda y es básicamente un algoritmo más que un modelo; su definición matemática se muestra en la ecuación 3.12, la cual es conocida como *Evidence Lower Bound*[7] (*ELBO*, por sus siglas en inglés). Esta ecuación es no convexa y representa un modelo de las redes *Generative Adversarial Network* (*GAN*, por sus siglas en inglés).

$$\mathcal{L} = \int q(z) log\left(p(x|z)\right) dz - \int q(z) log\left(\frac{p(z)}{q(z)}\right) dz \tag{3.12}$$

El límite inferior de la Ecuación 3.12, se muestra a continuación en la Ecuación 3.13.

$$\mathcal{L} = \mathbb{E}_{q(z)}[log(p(x|z))] - KL(q(z)||p(z)) \tag{3.13}$$

Donde $\mathcal{L}$ es el término *ELBO* y el primer y segundo factores del lado derecho de la igualdad representan decoder y encoder respectivamente, como se explica a continuación.

Decoder (Error de reconstrucción)

En la ecuación 3.13 el primer factor del lado derecho de la igualdad $p(x|z)$ representa la salida de la pérdida *log-likehood*, en función de la información observada x dada una representación interna z. Este término es una métrica que evalúa con que calidad representa $q(z)$ la información de entrada x. Este término tiene como función principal reconstruir a x mediante z, por lo cual es conocido como el termino de error de reconstrucción, cuya funcionalidad hace que su equivalencia en una arquitectura autoencoder sea el decoder.

Encoder (Regularización)

El segundo término, consiste en realizar una relación $z \sim q(z|x)$, representando un encoder. Este término se encarga de que la compresión de la información no se desvíe de las "creencias[3]" anteriores, $p(z)$, y es llamado término de regularización, definido mediante la divergencia KL entre q y el anterior $p(z)$.

En la Figura 3.5 se observa un autoencoder variacional, la primera etapa es el encoder, el cual realiza una inferencia de una variable z a partir de la entrada x. El segundo es un decoder que reconstruye $\tilde{x}$ de una variable oculta z. El encoder y el decoder, cuyas operaciones son denotadas mediante q_ϕ y p_θ respectivamente, son entrenados mediante el algoritmo *backpropagation*.

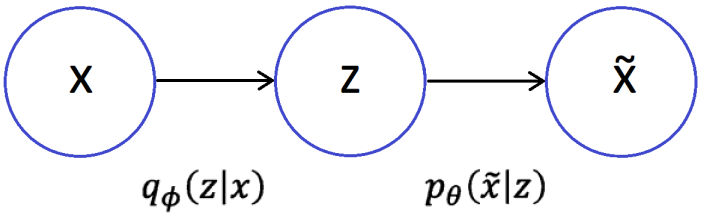

Figura 3.5: Topología de un autoencoder variacional.

3.2. Red *Extreme Learning Machine*

La arquitectura *Extreme Learning Machine* (*ELM*, por sus siglas en inglés) es un algoritmo de *Machine Learning* de las *Single Layer Feedforward Neural Networks* (*SLFNs* por sus siglas en inglés), que elige aleatoriamente nodos ocultos y determina analíticamente los pesos de salida de la *SLFNs*. En teoría, este algoritmo tiende a proveer un buen desempeño de generalización, mientras mantiene una velocidad de aprendizaje extremadamente rápida [21].Para un aproximador de funciones en una conjunto de entrenamiento finito, *SLFNs* con máximo N nodos ocultos y casi cualquier función de activación no lineal, puede aprender exactamente N distintas mediciones. *ELM* es una solución efectiva para *SLFNs*. Dado que la arquitectura *ELM* tiene gran velocidad para clasificación, ésta es ampliamente aplicada en clasificación en canales de flujo de datos. Las arquitecturas *ELM* tienen dos principales tipos de arquitectura:

- *SLFNs* con nodos ocultos aditivos.
- *SLFNs* con *Radial basis function* (*RBF*, por sus siglas en inglés), la cual aplica nodos *RBF* en la capa oculta.

[3]Las creencias son un término usado en modelado generativo que hace referencia a cualidades de aprendizaje estocástico.

La función de red de una *SLFN* con d nodos ocultos se muestra en la Ecuación 3.14.

$$f_d(x) = \sum_{i=1}^{d} \beta_i g_i(x), x \in \mathbb{R}, \beta_i \in \mathbb{R} \tag{3.14}$$

Donde g_i la función de salida del i-ésimo nodo oculto.
Dos funciones de salida comúnmente usadas para los nodos de salida g_i. En la Ecuación 3.15 se muestra la salida, para nodos aditivos.

$$g_i(x) = g\left(\langle W_i, x\rangle + b_i\right) = g\left(W_i^T x + b_i\right), w_i \in \mathbb{R}^n, b_i \in \mathbb{R} \tag{3.15}$$

en la ecuación 3.16 se muestra la salida, para nodos *RBF*.

$$g_i(x) = g\left(\frac{||x - a_i||}{b_i}\right), a_i \in \mathbb{R}^n, b_i \in \mathbb{R}_+ \tag{3.16}$$

donde a_i y b_i son el factor de centro e impacto del i-ésimo nodo *RBF* y son los pesos que conectan el i-ésimo nodo oculto con el nodo de salida.
En otras palabras, la función de salida de una **SLFN** con d nodos aditivos y d nodos **RBF**, pueden ser representadas como se muestra en las Ecuaciones 3.17 y 3.18, respectivamente.

$$f_d(x) = \sum_{i=1}^{d} \beta_i g\left(w_i^T x + b_i\right) \in \mathbb{R} \tag{3.17}$$

$$f_d(x) = \sum_{i=1}^{d} \beta_i g\left(\frac{||x - a_i||}{b_i}\right), a_i \in \mathbb{R}^n \in \mathbb{R} \tag{3.18}$$

Fue demostrado por Hornik que si la función de activación es continua, delimitada, y no constante, entonces mapeos continuos pueden ser aproximados mediante **SLFNs** con nodos ocultos aditivos y con una función de activación no polinomial puede aproximar cualquier función objetivo continua.
Considérese una **SLFN** w con pesos $\beta_1, ..., \beta_d$. Si la **SLFN** produce N muestras distintas, (x_i, y_i), $i = 1, ..., N$ donde $x_i = [x_{i1}, ..., x_{in}]^T \in \mathbb{R}^n$ y $y_i = [y_{i1}, ..., y_{im}]^T \in \mathbb{R}^m$, entonces las **SLFNs** con d nodos ocultos y función de activación $g(x)$ son matemáticamente modeladas como.

$$\begin{cases} \sum_{i=1}^{d} \beta_i g\left(w_i^T x_1 + b_i\right) = y_1, \\ \vdots \\ \sum_{i=1}^{d} \beta_i g\left(w_i^T x_N + b_i\right) = y_N \end{cases} \tag{3.19}$$

Donde $w_i = [w_{i1}, ..., w_{in}]^T \in \mathbb{R}^n$ es el vector de pesos que conecta el i-ésimo nodo oculto y los nodos de salida, b_i es el umbral del i-ésimo nodo oculto.

El sistema de ecuaciones mostrado en 3.19, puede ser transformado en una versión matricial, como se muestra a continuación:

$$\mathbf{HB} = \mathbf{Y}, \tag{3.20}$$

Donde:

$$\mathbf{H} = \begin{bmatrix} g\left(w_1^T x_1 + b_1\right) & \dots & g\left(w_d^T x_1 + b_d\right) \\ \vdots & \ddots & \vdots \\ g\left(w_1^T x_N + b_1\right) & \dots & g\left(w_d^T x_N + b_d\right) \end{bmatrix} \in \mathbb{R}^{Nxd}, \tag{3.21}$$

$$\mathbf{B} = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_d^T \end{bmatrix} = \begin{bmatrix} \beta_{11} & \dots & \beta_{1m} \\ \vdots & \ddots & \vdots \\ \beta_{d1} & \dots & \beta_{dm} \end{bmatrix} \in \mathbb{R}^{dxm}, \tag{3.22}$$

$$\mathbf{Y} = \begin{bmatrix} y_1^T \\ \vdots \\ y_N^T \end{bmatrix} = \begin{bmatrix} y_{11} & \dots & y_{1m} \\ \vdots & \ddots & \vdots \\ y_{N1} & \dots & y_{Nm} \end{bmatrix} \in \mathbb{R}^{Nxm}, \tag{3.23}$$

La Solución de la Ecuación 3.20, tiene la forma:

$$\mathbf{B} = \mathbf{H}^{\dagger}\mathbf{Y} \tag{3.24}$$

Donde $\mathbf{H}^{\dagger}$ es la inversa generalizada de *Moore-Penrose* [21, Capitulo 1].

En la Figura 3.6, se observa la arquitectura *ELM*, donde se aprecia que la capa de entrada se relaciona con la intermedia mediante w_i y la intermedia con la de salida mediante β_i.

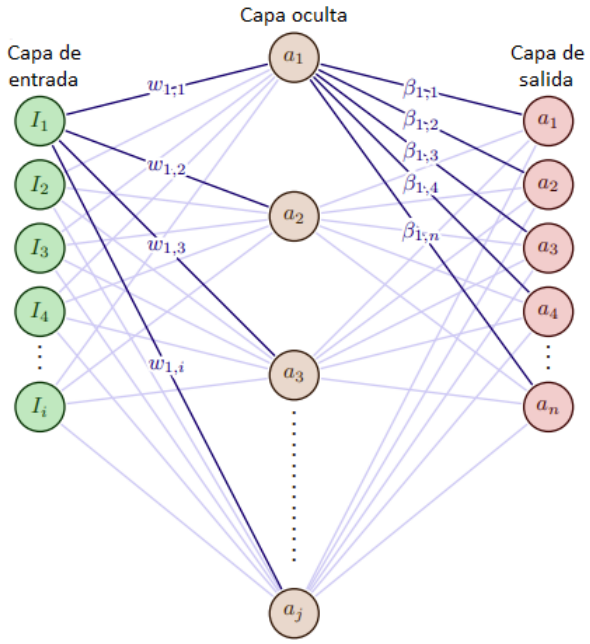

Figura 3.6: Arquitectura de una *ELM*, con i entradas, j nodos ocultos y n clases.

Algoritmo ELM

El algoritmo de entrenamiento y prueba de la *ELM*, se desarrollará a continuación con ayuda de una descripción en *Python* del mismo; éste consta de 4 pasos.

Paso 1:
Se selecciona un conjunto de entrenamiento $\{(x_i, y_i)|x_i \in \mathbb{R}^n, y_i \in \mathbb{R}^m, i = 1, ..., N\}$, una función de activación $g(x)$ y el número de nodos ocultos d. Para realizar la descripción en lenguaje *Python*, primeramente es necesario incluir las bibliotecas que serán empleadas. En la Sección de código 3.1 se aprecia la declaración de las bibliotecas que serán empleadas.

```
import tensorflow as tf
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
```

Sección de código 3.1: Creación del conjunto de datos y configuración de parámetros.

La descripción en *Python* del paso mencionado se muestra en la Sección de código 3.2, haciendo un conjunto de datos de 50 vectores de entrenamiento tridimensionales; 25 vectores pertenecen a la clase 1 y 25 a la clase 2.

```
N=50 # numero de vectores de entrenamiento
d=15 # numero de nodos ocultos
x_t = np.zeros((N,3)) # vectores de entrada
y_t = np.zeros((N,1)) # vector de etiqueta
for i in range(0,N): # rutina de generacion de valores aleatorios
  if i < N//2:
    mean = 1; y_t[i] = 0 # clase 1
  else:
    mean = -1; y_t[i] = 1 # clase 2
  x_t[i] = tf.random.normal(shape =
      [1,3],mean=mean,stddev=0.5,dtype=tf.dtypes.float32,seed=None,name=None)
```

Sección de código 3.2: Creación del conjunto de datos y configuración de parámetros.

En la Figura 3.7, se observan los dos conjuntos generados, los cuales constan de dos agrupaciones centradas en $[1, 1, 1]$ (vectores rojos) y $[-1, -1, -1]$ (vectores azules).

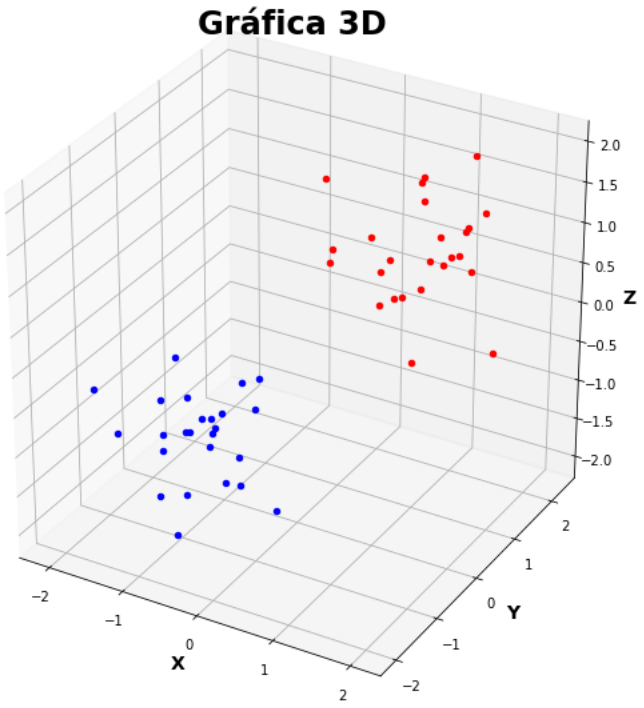

Figura 3.7: Visualización en el espacio, de los vectores de entrenamiento cuyas agrupaciones en color rojo y azul corresponden a las clases 1 y 2 respectivamente.

Paso 2:

Se inicializan los vectores de pesos w_i y polarizaciones b_i con valores aleatorios, siendo $i = 1, ..., d$, como se observa en la Sección de código 3.3.

```
w = tf.random.uniform(shape = [d,3],minval=-0.2,maxval=0.2,
                      dtype=tf.dtypes.float32,seed=None,name=None)
b = tf.random.uniform(shape = [d,1],minval=-0.2,maxval=0.2,
                      dtype=tf.dtypes.float32,seed=None,name=None)
```

Sección de código 3.3: Inicialización de los vectores de pesos w_i y polarizaciones b_i.

Paso 3:

3.1.- Mediante la Ecuación 3.21, se calcula H.
3.2.- Mediante la Ecuación 3.23, se calcula Y.
3.3 Se calcula la norma de mínimos cuadrados, primeramente obteniendo B, de la Ecuación 3.24, posteriormente se obtiene el vector de pesos β_i de $B^T = [\beta_1, ..., \beta_d]$.
En la Sección de código 3.4, se realiza el proceso de cálculo de los pasos mencionados.

```
# paso 3.1
H = np.zeros((N,d)) # se reserva espacio para la matriz H
def sig(x): # se define la funcion de activacion sigmoide
  return 1/(1 + np.exp(-x)) # se realizan las operaciones matriciales
for i in range(0,N):
  for j in range(0,d):
    H[i,j] = sig(np.dot(np.transpose(w[j]),x_t[i]) + b[j])
# paso 3.2
onehotencoder = OneHotEncoder(categories='auto')
Y = onehotencoder.fit_transform(y_t).toarray() # transforma el vector de clases a
    arreglo binario
# paso 3.3
B = np.matmul(np.linalg.pinv(H), Y) # se determina la solucion numerica para la
    matriz B
```

Sección de código 3.4: Cálculo de las matrices de la arquitectura.

En la Sección de código 3.4 se usaron algunos comandos numéricos, de los cuales se realizará una breve descripción a continuación:

`onehotencoder.fit_transform(y_t).toarray()`

El objeto `onehotencoder` contiene los atributos de la clase `OneHotEncoder`, cuyos parámetros configurados son `categories='auto'`. Este comando se encarga de convertir una columna de clases proporcionada (en este caso `y_t`) en una matriz binaria con tantas columnas como clases, en cuyas filas solo existe un valor verdadero situado en la columna correspondiente al número de la clase.

`np.matmul(a,b)`

Este comando proviene de la biblioteca numérica de *Python*, `numpy` y hace referencia a una multiplicación matricial entre los elementos `a` y `b`.

`np.linalg.pinv(H)`

Este comando se encuentra en la biblioteca numérica de *Python*, y calcula la Matriz pseudo-inversa (Referida primeramente como matriz generalizada de *Moore-Penrose*) de la matriz proporcionada.

Paso 4:

Mediante la Ecuación 3.25 se determina la salida.

$$y = \sum_{i=1}^{d} \beta_i g\left(w_i^T x + b_i\right) \tag{3.25}$$

En la Sección de código 3.5 se determina la salida mediante dos ciclos, los cuales iteran los cálculos, cuya cantidad de cálculos está determinada por el número de vectores de prueba y el número de neuronas en la capa intermedia.

```
n = np.shape(x_t)[0]  # Declaracion de arreglos
y_test = np.zeros((d,2))
y_out = np.zeros((n,1))

for j in range(0,n): # Implementacion de la sumatoria en un ciclo
  for i in range(0,d):
    y_test[i] = B[i]*sig(np.dot(np.transpose(w[i]),x_t[j]) + b[i])
  if np.sum(y_test, axis=0)[0] > np.sum(y_test, axis=0)[1]:
    y_out[j] = 0
  else:
    y_out[j] = 1
```

Sección de código 3.5: Cálculo de la salida del modelo.

En la Figura 3.8 se observan los vectores de prueba (en este ejemplo son los mismos que se usaron para el entrenamiento); los vectores amarillos y verdes corresponden a los vectores correctamente identificados por el modelo de la clase 1 y 2 respectivamente, y los vectores rojos (no observados en la Figura 3.8) corresponden a los vectores incorrectamente identificados.

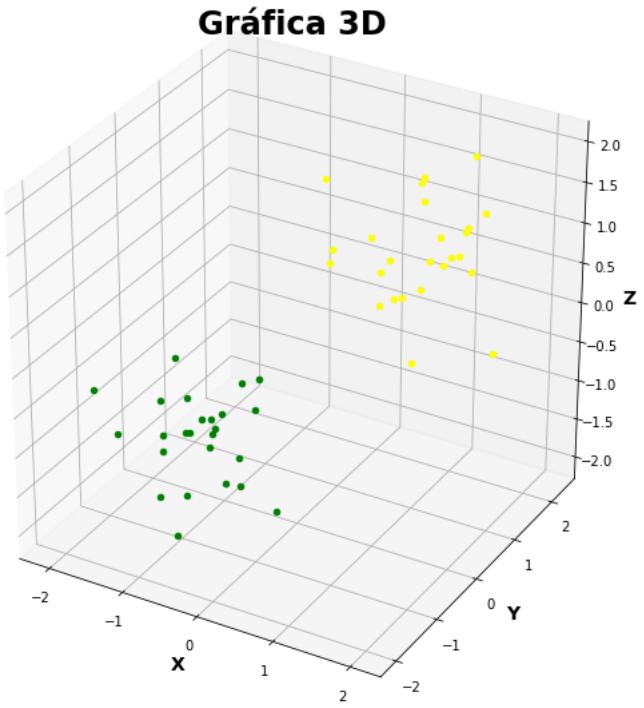

Figura 3.8: Visualización de los vectores clasificados. **Vectores amarillos:** Corresponden a vectores pertenecientes a la clase 1, correctamente identificados por el modelo. **Vectores verdes:** Corresponden a vectores de la clase 2 correctamente identificados.

Con el propósito de ejemplificar un caso de error de clasificación, se modificará el parámetro `stddev=0.5` de la Sección de código 3.2, cambiándolo a `stddev=1.2`, lo que hará que la distribución normal de valores aleatorios tenga un mayor índice de dispersión, creando un cierto grado de incertidumbre que dificultará la tarea de clasificación.
En la Figura 3.9 se observa el conjunto de datos creado y como se observa, los vectores se encuentran mayormente distribuidos en el espacio. En la Figura 3.10 se aprecian los vectores correctamente clasificados, de color amarillo y verde, así como los erróneamente clasificados en color rojo.

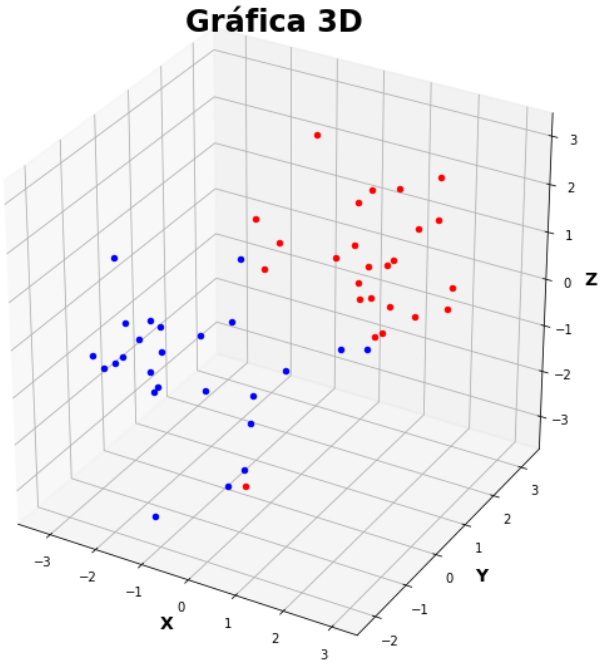

Figura 3.9: Visualización de los conjuntos creados. **Vectores rojos:** Corresponden a la clase 1. **Vectores azules:** Corresponden a la clase 2.

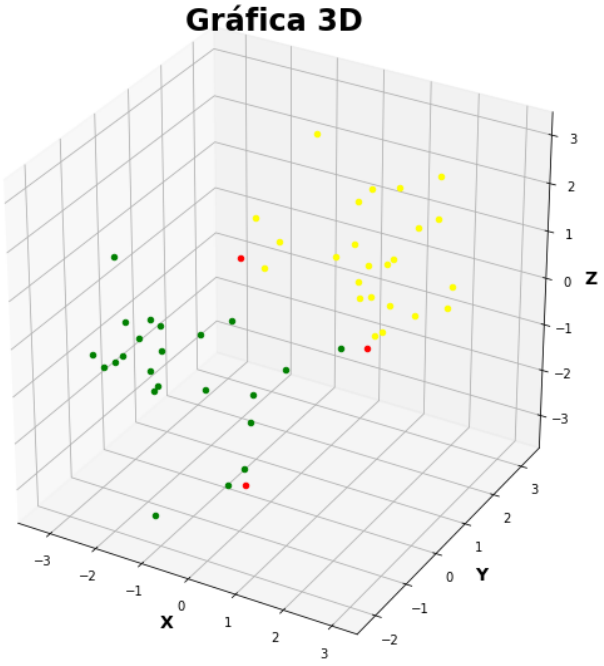

Figura 3.10: Visualización de los vectores clasificados. **Vectores amarillos:** Corresponden a vectores pertenecientes a la clase 1, correctamente identificados por el modelo. **Vectores verdes:** Corresponden a vectores de la clase 2 correctamente identificados. **Vectores rojos:** Vectores erróneamente clasificados.

3.3. Conclusiones del capítulo

En este capítulo se estudió la metodología de reducción de dimensión mediante autoencoders, siendo esta una de las principales aplicaciones del aprendizaje no supervisado. La atenuación del efecto de ruido aleatorio y la reducción del sobreajuste en datos comprimidos son algunas de sus principales ventajas que la convierten en una opción atractiva para la solución de problemas que tratan con datos en grandes volúmenes. Se concluye que la arquitectura autoencoder ha sido ampliamente usada en redes neuronales y adoptada en ramales avanzados de la inteligencia artificial, como lo son redes neuronales convolucionales y modelos generativos.

Con respecto al aprendizaje supervisado se estudió la arquitectura *ELM*, la cual es un modelo de clasificación eficiente, cuyo coste computacional en el entrenamiento es considerablemente alto, pero cuyo desempeño fuera de línea es muy satisfactorio y conveniente para sistemas embebidos de computo en tiempo real, con lo que se puede concluir que esta metodología de clasificación es adecuada para los propósitos de este trabajo.

También se concluye que las herramientas de construcción de modelos de aprendizaje en máquina de *Tensor Flow*, representan una ventaja en el flujo de trabajo, pues se puede construir un modelo de una manera muy detallada, además de la capacidad de acceder en cualquier momento a todos los parámetros del modelo, así como la configuración del entrenamiento.

CAPÍTULO 4

Propuesta de Sistema Digital para Detección de Fallas

La propuesta del sistema digital se realizará mediante un modelo de lógica programable que está constituido de bloques aritméticos de cálculo, así como de procesos secuenciales sincronizados. A modo de ejemplo se creará una propuesta para modelar una neurona desde el punto de vista digital, siendo está la base de los modelos cognitivos de cómputo; dicha neurona contará con propiedades sincrónicas que la hacen escalable y a su vez permitirán la creación de modelos con diversas organizaciones neuronales. En este trabajo se empleó el paquete de descripción *HDL* desarrollado por la empresa *AMD Xilinx©*(véase la Figura 4.1), *Vivado Design Suite©*, el cual es un entorno de desarrollo integrado con herramientas de síntesis, implementación, simulación y configuración de dispositivos. Entre las familias de *FPGA's* que soporta el paquete son *Virtex-7, Kintex-7, Artix-7, y Zynq-7000*. El dispositivo usado como ejemplo para este trabajo es de la familia *Artix-7*, con el chip `XC7A100T-CSG324`.

Figura 4.1: Logotipo de *Xilinx©*, propiedad de la empresa *AMD©*.

4.1. Extracción de datos estadísticos en la base de datos, para la estimación de recursos

Con la finalidad de obtener un mejor aprovechamiento de los recursos del dispositivo de prueba (FPGA), se hace una estimación de los valores máximos y mínimos de la base de datos. En la Sección de código 4.1 se aprecia la descripción en *Python* de la extracción de dichos parámetros.

```
import numpy as np
import pandas as pd

file_names = ["97.csv",
          "inner_race_007.csv","inner_race_014.csv","inner_race_021.csv",
          "outer_race_007.csv","outer_race_014.csv","outer_race_021.csv",
          "ball_007.csv","ball_014.csv","ball_021.csv"]

dataset_statics = np.zeros((10,2))

for i in range(10):
    dataframe = pd.read_csv(file_names[i],header=None)
    raw_data = dataframe.values
    dataset_statics[i] = np.amax(raw_data, axis = 0),np.amin(raw_data, axis = 0)
```

Sección de código 4.1: Extracción de máximos y mínimos de los conjuntos de datos.

La salida obtenida de la descripción 4.1 se muestra en la Sección de código 4.2.

```
[[ 0.31125415 -0.28663754]
 [ 1.7390305 -1.37988643]
 [ 1.93427752 -2.01078447]
 [ 3.78798723 -3.37174721]
 [ 3.63042515 -3.4087012 ]
 [ 0.55056036 -0.55043856]
 [ 6.65293683 -6.41172066]
 [ 0.60393381 -0.60702008]
 [ 2.27815269 -1.96351585]
 [ 1.65959972 -1.61785389]]
```

Sección de código 4.2: Máximos y mínimos de los conjuntos.

Con la información obtenida de la Sección de código 4.2 se determina que ninguna entrada en la parte entera excede 7, por lo cual puede representarse con 3 bits; como los datos serán tratados como signados, la representación a 16 bits será: 1 bit de signo, 3 bits en la parte entera y 12 en la parte fraccionaria. A continuación, se realiza una explicación de la representación de cantidades reales en el dominio discreto, tratados como cantidades de punto fijo con una cantidad de ocho bits.

$$\text{Una cadena de 8 bits: } \underbrace{0101.0101}_{\text{Representación de un número de punto fijo a 8 bits}}$$

$$\text{El primer bit es de signo: } \underbrace{0}_{1=\text{-},0=+} \quad 101.0101$$

$$\text{Los siguientes bits representan la parte entera: } 0 \quad \underbrace{101}_{2^2*1+2^1*0+2^0*1} \quad .0101$$

$$\text{Los siguientes bits representan la parte fraccionaria: } 0101. \quad \underbrace{0101}_{2^{-1}*0+2^{-2}*1+2^{-3}*0+2^{-4}*1}$$

Por lo tanto el valor de la cadena de 8 bits, con el punto decimal en el bit 4 es:

$$01010101 = +3.3125$$

El uso de números de punto fijo resulta conveniente, desde el punto de vista numérico, en comparación con las operaciones de punto flotante, cuya resolución se ajusta continuamente dependiendo de la parte en donde se necesite de mayor exactitud.

El tratamiento de estos datos a nivel de hardware es posible mediante el uso de bibliotecas dedicadas.

4.2. Manejo de la biblioteca `FIXED_PKG`

La biblioteca `FIXED_PKG` fue creada para la operación de números en representación de punto fijo. En el área de la inteligencia artificial el uso de valores reales es necesario. Dicha biblioteca es compatible con el estándar IEEE-754, y su soporte para síntesis y simulación, es mayor cada vez.

Declaración del paquete y tipos de números en punto fijo

La declaración del paquete se hace primeramente declarando la biblioteca `IEEE_PROPOSED`, para posteriormente declarar el uso del paquete, como se muestra en la Sección de código 4.3.

```
library ieee_proposed;
use ieee_proposed.fixed_pkg.all;
```

Sección de código 4.3: Inclusión de bibliotecas.

Existen 2 tipos de números de punto fijo, los cuales se describen a continuación:

`ufixed` Número de punto fijo no signado, declarado en la Sección de código 4.4.

```
type ufixed is array (integer range <>) of std_logic;
```

Sección de código 4.4: Declaración de un número de punto fijo no signado.

`sfixed` Número de punto fijo signado, declarado en la Sección de código 4.5.

```
type sfixed is array (integer range <>) of std_logic;
```

Sección de código 4.5: Declaración de un número de punto fijo signado.

Funciones de conversión

Existen 2 funciones principales que se encargan de realizar la conversión de un número real a su representación en punto fijo, las cuales se muestran a continuación.

Conversión a punto fijo no signado, declarado en la Sección de código 4.6.

```
to_ufixed( number, a, -b)
```

Sección de código 4.6: Conversión a punto fijo no signado.

Conversión a punto fijo signado, declarado en la Sección de código 4.7.

```
to_sfixed( number, a, -b)
```

Sección de código 4.7: Conversión a punto fijo signado.

Donde `number` es el número real que se quiere representar y `a`, `b` son un números enteros mayores que cero, los cuales representan el número de bits asignados para la parte entera (mas el bit de signo) y decimal respectivamente.

También existen funciones de conversión entre tipos de números, es decir, entre tipos de variables creadas con el fin de transferir de dominio ciertas variables y de esta manera crear un entorno más flexible numéricamente.

Conversión a `std_logic_vector`
En la Sección de código 4.8, se muestra la sintaxis de dicha conversión.

```
to_std_logic_vector(var), to_slv(var)
```

Sección de código 4.8: Conversión de formato a `std_logic`.

Donde `var` es la variable convertida y puede ser del tipo entero, punto fijo o flotante.

Conversión a tipo entero
En la Sección de código 4.9 se observa la sintaxis de la conversión a entero.

```
to_integer(var)
```

Sección de código 4.9: Conversión de formato a tipo entero.

Donde `var` es la variable convertida y puede ser de punto fijo o flotante.

Operadores

Los operadores que pueden ser usados a través de esta biblioteca son los siguientes.

Operadores lógicos.

`and, nand, or, nor, xor, xnor, not`

Operadores aritméticos.

`+, -, *, /, rem, mod, abs`

Operadores de comparación.

`=, >=, <=, <, >, / =`

Operadores de funciones de intercambio.

`sll, srl, rol, ror, sla, sra`

4.3. Ejemplo introductorio

A manera de ejemplo, en la Sección de código 4.10 se realiza una descripción en lenguaje *VHDL*, para una multiplicación de dos números reales no signados de 24 bits cada uno, con resultado de 48 bits, haciendo uso de los operadores ya descritos.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;-- Declaracion de la bibliotecas de proposito general

library IEEE_PROPOSED;
use IEEE_PROPOSED.FIXED_PKG.ALL;-- Declaracion de la bibliotecas para operaciones de
    punto fijo

entity simple_opperation is
Port (port_in1: in std_logic_vector(23 downto 0);
      port_in2: in std_logic_vector(23 downto 0);-- Declaracion de los puertos de entrada
      port_out: out std_logic_vector(47 downto 0));-- Declaracion de los puertos de salida
end simple_opperation;

architecture Behavioral of simple_opperation is

begin
m_1<=to_ufixed(port_in1,11,-12); -- Conversion de std_logic_vector a real no signado
m_2<=to_ufixed(port_in2,11,-12);

port_out <= to_slv(m_1*m_2); -- Multiplicacion y conversion del resultado real a
    std_logic_vector

end Behavioral;
```

Sección de código 4.10: Multiplicación de dos números de punto fijo, mediante el uso de la biblioteca `fixed_pkg`.

Para comprobar que los resultados son los esperados se procede a realizar la simulación de la descripción mediante un banco de pruebas en *VHDL*, también conocido como *test bench*, el cual se muestra en la Sección de código 4.11.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library IEEE_PROPOSED;
use IEEE_PROPOSED.FIXED_PKG.ALL;

entity sim_mul is
end sim_mul;

architecture Behavioral of sim_mul is

     signal port_in1: std_logic_vector(23 downto 0);
     signal port_in2: std_logic_vector(23 downto 0);
     signal port_out: std_logic_vector(47 downto 0);

begin

UUT: entity work.simple_opperation port map(port_in1=>port_in1, port_in2=>port_in2,
    port_out=>port_out); -- Instanciacion de los puertos de entrada en la descripcion
    principal, en la simulacion

process
begin
port_in1 <= x"2EE7B0"; -- 750.48
port_in2 <= x"15E730"; -- 350.45
wait for 10us;
wait;

end process;
end Behavioral;
```

Sección de código 4.11: Banco de pruebas descrito en *VHDL*, para la simulación de una multiplicación.

En la Figura 4.2, se observa el resultado de la multiplicación 750.48∗350.45, obtenido de la simulación funcional descrita en la sección de código 4.11; cabe mencionar que existe la pérdida de exactitud dependiendo de la resolución con que se trabaje.

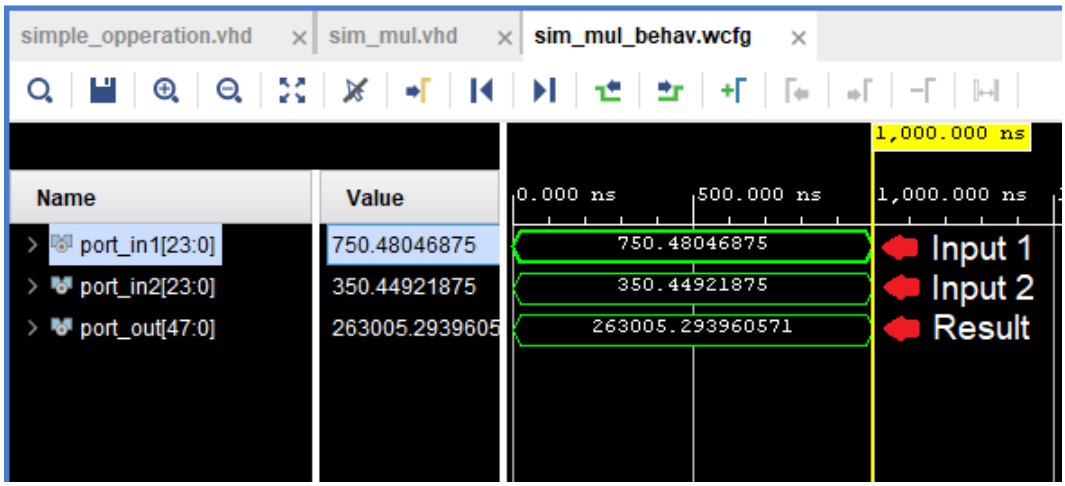

Figura 4.2: Resultado de la simulación lógica.

4.4. Propuesta de modelo digital de una neurona

Para poder implementar una red neuronal en dispositivos lógicos programables en campo, es necesario modelar la secuencia de cálculo, como las que se llevan a cabo en lenguajes de programación para sistemas basados en procesador; en este trabajo se realizó la descripción de una sola neurona cuyo funcionamiento individual (con algunas modificaciones) puede escalarse para crear capas interconectadas, dando lugar a redes neuronales implementadas en *hardware*.

En la Figura 4.3, se observan las terminales de una neurona digital propuesta.

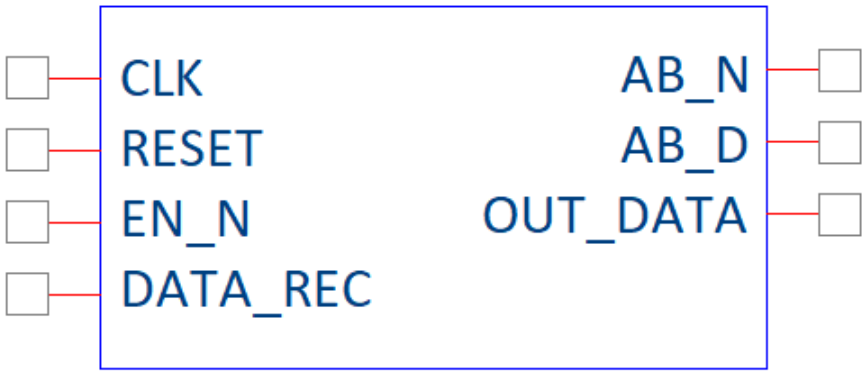

Figura 4.3: Modelo esquemático de una neurona digital.

En la Tabla 4.1, se realiza la descripción de los puertos del modelo esquemático de la Figura 4.3.

Puerto	Tipo	Descripción
`CLK`	`IN`	Entrada de reloj
`RESET`	`IN`	Entrada de re establecimiento
`EN_N`	`IN`	Entrada que habilita la secuencia de procesamiento de la neurona
`DATA_REC`	`IN`	Entrada que notifica que la información de salida ha sido recolectada
`AB_N`	`OUT`	Salida que notifica el estatus de la neurona: Disponible/No Disponible
`AB_D`	`OUT`	Salida que notifica cuando el procesamiento ha concluido y el resultado se encuentra disponible en la salida
`OUT_DATA`	`OUT-BUS`	Bus de salida

Tabla 4.1: Descripción de los puertos empleados en el modelo propuesto.

4.4.1. Explicación del modelo propuesto

Para poder modelar correctamente el proceso de cálculo que se lleva a cabo dentro de una unidad de procesamiento neuronal, es necesario sincronizar los procesos que se llevan a cabo en dicha unidad. Como se vio en la sección 2.3, las propiedades dinámicas de las redes neuronales son un factor de importancia a la hora de modelar un sistema neuronal. Para poder relacionar los procesos secuencialmente, se propuso una máquina de estados cuyas transiciones se encuentran condicionadas por las señales de estímulo y el estado interno de las unidades de cálculo. En la Figura 4.4, se muestran los 3 estados propuestos y las condicionales para la transición entre ellos.

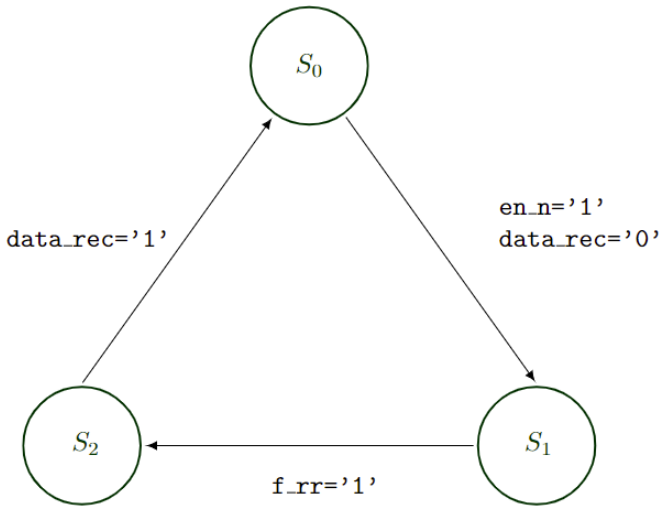

Figura 4.4: Máquina de estados propuesta para el proceso de cálculo de una neurona digital.

Los estados `S0, S1` y `S2` se encuentran relacionados mediante las señales de entrada ya descritas `en_n,data_rec`, además de una señal interna `f_rr`, correspondiente a un subproceso[1].

Metodología empleada en el algoritmo

Estado inicial `S0`

Es el estado donde la neurona se encuentra lista para realizar el proceso de cálculo.

Estado de las señales de salida:

`AB_N='1'`: Indica que la neurona está disponible.
`AB_D='0'`: Indica que el resultado aún no está disponible.

Transición:

La transición del estado `S0` a `S1`, se realiza una vez que la señal de habilitación `en_n` es verdadera y la entrada de información recibida `data_rec` es falsa.

Estado de procesamiento `S1`

Es el estado en el cual se lleva a cabo el proceso de cálculo; es decir, multiplicar los pesos sinápticos por las entradas, realizar la sumatoria de todos los productos, sumar la polarización y aplicar la función de activación.

Estado de las señales de salida:

`AB_N='0'`: Indica que la neurona no está disponible.
`AB_D='0'`: Indica que el resultado aún no está disponible.

Transición:

La transición del estado de cálculo al estado de confirmación, se efectúa una vez que la señal interna de conclusión de cálculo `f_rr` es verdadera.

[1] En *VHDL*, un subproceso es un módulo de una organización jerárquica, la cual es usada en programación estructural.

Estado de confirmación `S2`

Es el estado en el que la neurona espera a que se le notifique que la información que calculó ha sido recibida correctamente(por lo que la neurona no estará disponible mientras no reciba dicha confirmación).

Estados de las señales de salida:

`AB_N='0'`: Indica que la neurona no está disponible.
`AB_D='1'`: Indica que el resultado ya está disponible.

Transición:

La transición se hace una vez que la señal de confirmación `data_rec` es verdadera, informando que la información ha sido correctamente recopilada.

La descripción completa de la máquina de estados en lenguaje *VHDL*, se encuentra disponible en el apéndice.

Simulación del sistema propuesto

Para constatar que el comportamiento de la descripción realizada, en el apéndice es correcta, se procedió a realizar una simulación funcional del modelo descrito; a continuación se muestra la descripción del banco de pruebas en lenguaje *VHDL*:

Declaración de bibliotecas y entidad

El primer paso para elaborar un banco de pruebas, consiste en declarar las bibliotecas necesarias para la simulación de la descripción y crear una identidad vacía, es decir, sin puertos de entrada-salida, como se muestra en la Sección de código 4.12.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all; -- Declaracion de las bibliotecas

entity state_tb is -- Empty entity -- Entidad vacia
end state_tb;
```

Sección de código 4.12: Declaración de bibliotecas.

Declaración de la arquitectura y las señales a usar

Una vez declarada la arquitectura, se crean las señales internas a simular; éstas deben de estar asociadas a los puertos de entrada-salida del modelo a simular y por lo tanto deben ser del mismo tipo y dimensiones, como se ilustra en la Sección de código 4.13.

```
architecture Behavioral of state_tb is

signal clk   :   std_logic;
signal reset :   std_logic;
signal en_n  :     std_logic;
signal data_rec :  std_logic;
```

Sección de código 4.13: Declaración de bibliotecas.

Creación del vínculo entre el banco de pruebas y el modelo:

La creación del vínculo entre el banco de pruebas y el modelo se realiza una vez que la descripción ha comenzado, después del comando `begin`, haciendo referencia al nombre del archivo que contiene el modelo en el banco de pruebas y asignando los puertos del modelo a las señales internas del banco de pruebas, como se muestra en la Sección de código 4.14.

```
begin
UUT: entity work.moore port map(clk=>clk,reset=>reset,
                              en_n=>en_n,data_rec=>data_rec);
```

Sección de código 4.14: Importación de variables de la descripción principal al banco de pruebas.

Creación de la secuencia de simulación:

En esta parte se crean los estímulos temporizados, los cuales asignan valores discretos a las señales que están asociadas a entradas del modelo mediante procesos que se llevan a cabo bajo el dominio del tiempo; en la sección de código 4.15 se realiza la descripción de dos procesos, uno correspondiente a la señal de reloj y otro a los valores de las entradas para la transición de estados.

```
process -- Periodo de reloj de dos microsegundos
begin
clk <= '0'; wait for 1us;
clk <= '1'; wait for 1us;
end process;

reset <= '0';

process -- Senales de control de la maquina de estados
begin
data_rec <= '0';
en_n <= '0'; wait for 5us;
en_n <= '1'; wait for 35us;
data_rec <= '1'; wait for 1us;
end process;
```

Sección de código 4.15: Creación de la secuencia de simulación del banco de pruebas.

Para analizar el comportamiento temporal del sistema descrito, se procederá a correr la simulación del sistema; a continuación, se muestra la respuesta del sistema en una ventana de análisis lógico del paquete *Vivado Design Suite*.En la Figura 4.5 se observa la simulación general del sistema.

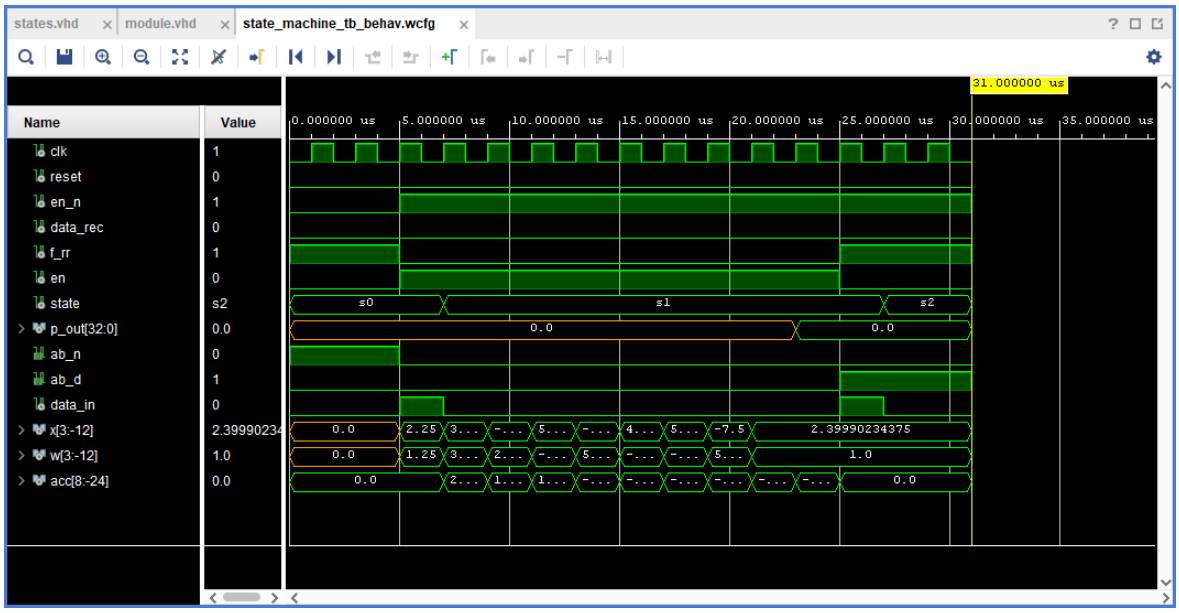

Figura 4.5: Simulación general del sistema.

Estado de espera (Estado `S0`)

A continuación, se describirán los procesos secuenciales de la descripción, haciendo un análisis funcional, como se observa en la Figura 4.6.

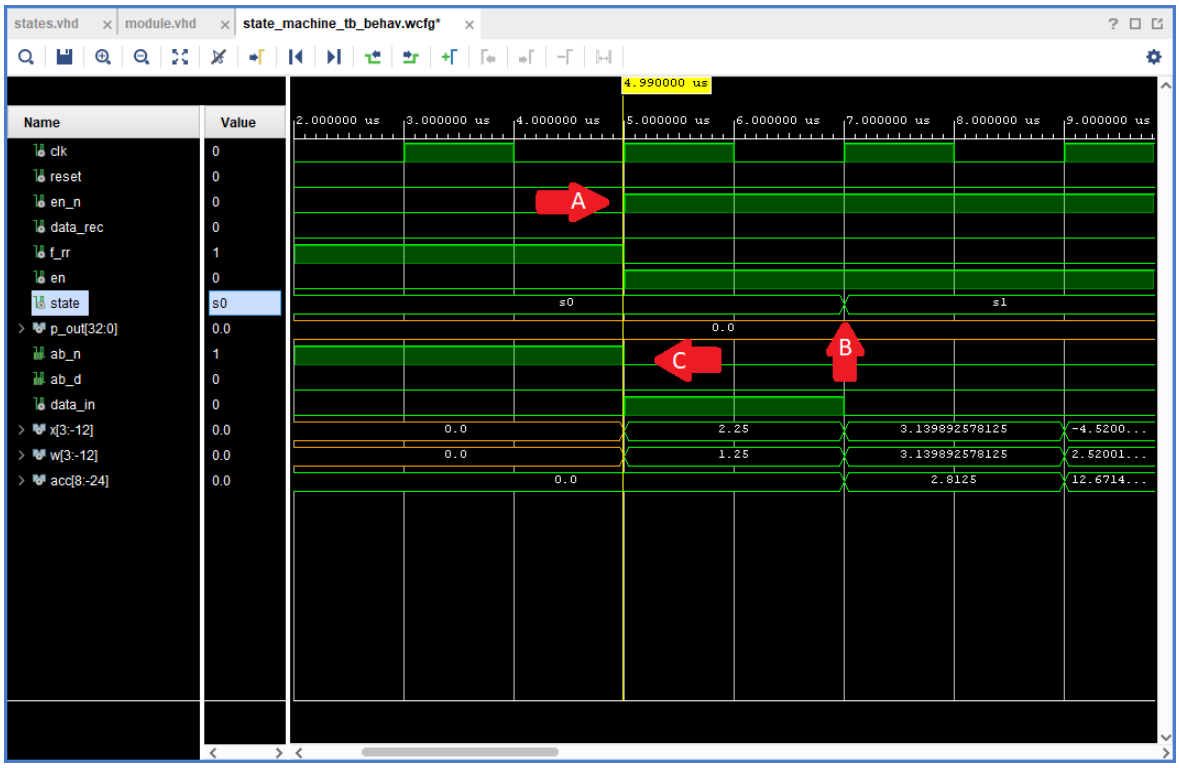

Figura 4.6: Simulación de la transición de `S0` a `S1`.

Indicador `A`:

> En este indicador se enfatiza el comienzo del proceso de razonamiento digital propuesto. La señal `en_n` tiene una transición de estado bajo a alto en $t = 5\mu s$, siendo este evento el disparador del proceso de cálculo.

Indicador `B`:

> En este indicador se hace referencia a la transición de estados; se observa que una vez que ha sido habilitada la señal `en_n`, en el siguiente ciclo de reloj se efectúa la transición del estado de `S0` al estado `S1`.

Indicador `C`:

> En este indicador se señaliza el cambio de nivel lógico de la salida `ab_n`, indicando así que la neurona no se encuentra disponible.

Análisis Aritmético (Estado `S1`):

Para el análisis del proceso aritmético, es necesario configurar la herramienta de simulación, con el fin de observar resultados que puedan ser interpretados fácilmente. Para la interpretación de números en punto fijo, el simulador de *Vivado Design Suite* cuenta con herramientas de análisis real. El procedimiento para cambiar la forma en que se muestra el valor de un registro, es necesario hacer click derecho en la señal de estudio y seleccionar ***Radix/Real Settings***, como se muestra en la Figura 4.7.

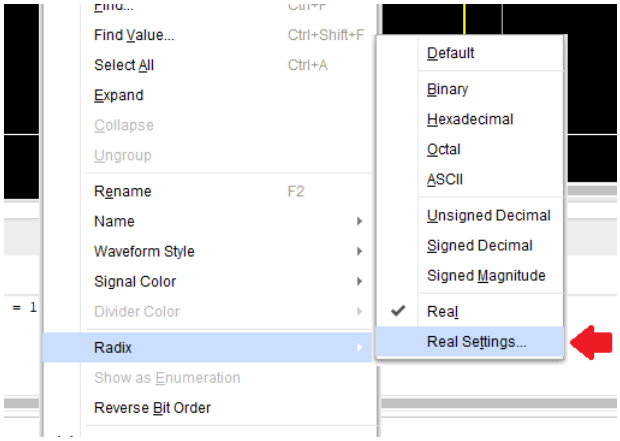

Figura 4.7: Configuración de las propiedades de visualización.

Posteriormente, en la ventana emergente, es necesario indicar las propiedades del número de punto fijo; si es signado/no signado y la ubicación del punto decimal, como se muestra en la Figura 4.8.

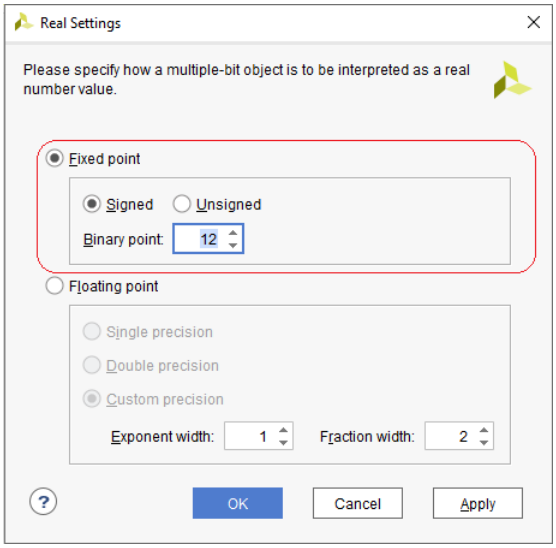

Figura 4.8: Configuración de las propiedades de la variable de punto fijo.

En la Figura 4.9, se señalan tres indicadores referentes al proceso de multiplicación. La señal `acc`, es un registro que almacena en cada iteración el producto de las dos entradas actuales.

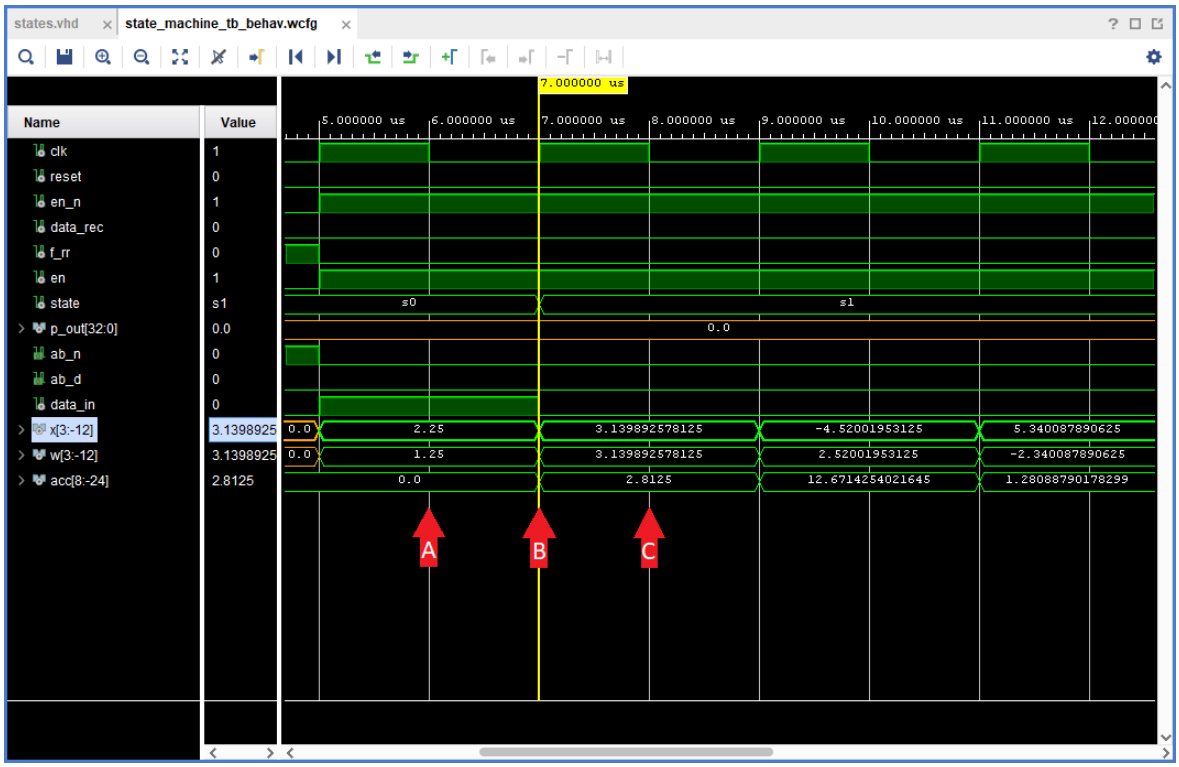

Figura 4.9: Simulación de la transición de `S0` a `S1`.

Indicador `A`:

> En este indicador se hace referencia principalmente a tres registros: el primero, la señal `x` que corresponde a las entradas de la neurona, el segundo `w`, correspondiente a los pesos sinápticos, y el tercero, la señal `acc` al acumulador. Las señales `x` y `w` tienen un valor asignado una vez que la neurona ha sido habilitada.

Indicador `B`:

> Este indicador señala la transición en la que se realizan dos procesos:
> - Se transfiere al acumulador `acc` el valor de la multiplicación entre `x` y `w`.
> - Se actualizan los valores de `x` y `w`.

Indicador `C`:

> En este indicador se enfatiza la actualización de los valores de `x` y `w`, además de la suma acumulativa que se lleva a cabo en la señal `acc`.

En la Figura 4.10 se observa el proceso final de estado `S1`.

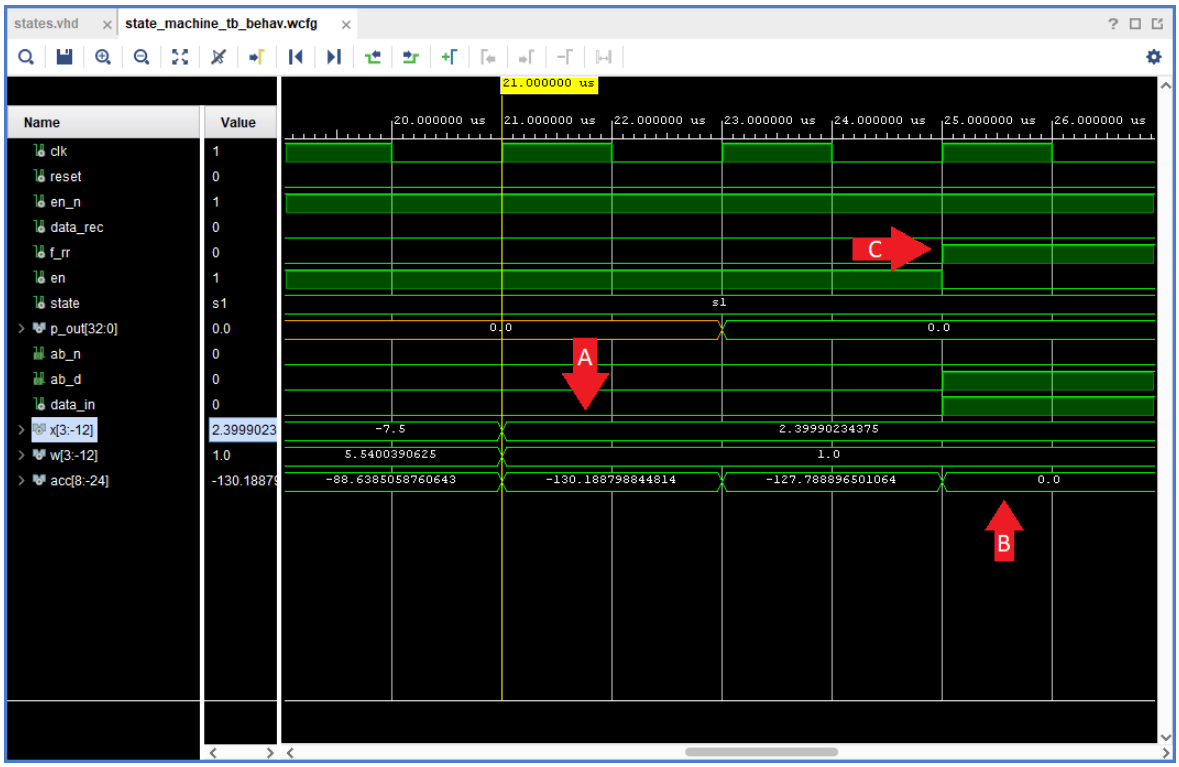

Figura 4.10: Secuencia final de la transición de `S0` a `S1`.

Indicador `A`:

> En este indicador se hace referencia al proceso de suma de polarización, el cual se introduce a la suma acumulativa, multiplicando el valor de la polarización por uno, ahorrando así un proceso secuencial extra.

Indicador `B`:

> En este indicador se señala la aplicación de la función de activación *RELU*; como el resultado es negativo, al ser aplicada la función de activación, la salida es cero.

Indicador `C`:

> En este indicador se hace notar el funcionamiento de la señal interna `f_rr`, cuyo valor en alto indica que el proceso aritmético ha finalizado, pues la suma acumulativa ha concluido y su valor ha sido evaluado en la función de activación.

Estado de espera de confirmación (Estado `S2`):

En la Figura 4.11, se observa la simulación funcional del estado de espera de confirmación.

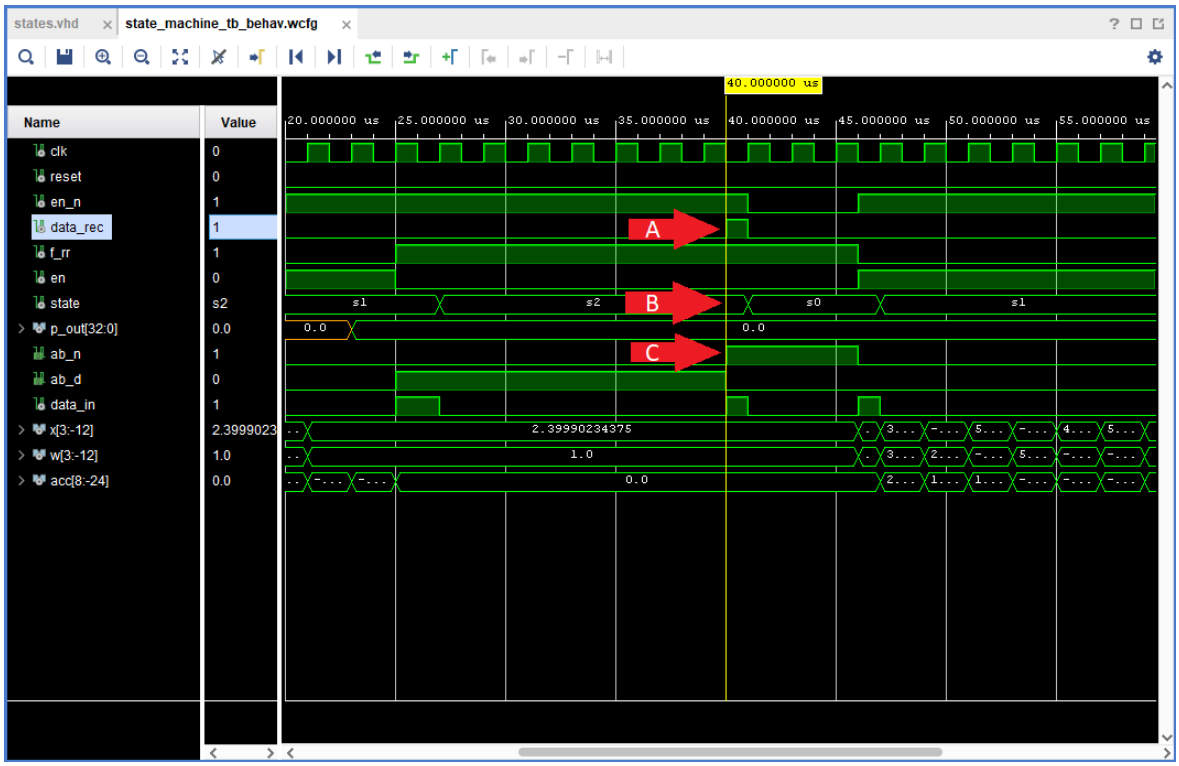

Figura 4.11: Simulación del estado de confirmación.

Indicador `A`:

> En este indicador se muestra que la señal de confirmación `data_rec` es verdadera, lo cual notifica que la información proporcionada por la neurona ha sido recolectada correctamente.

Indicador `B`:

> En este indicador se enfatiza el cambio de estado, el cual se lleva a cabo una vez que la señal de confirmación `data_rec` es verdadera.

Indicador `C`:

> Este indicador hace referencia a la señal `ab_n`, la cual es una señal de salida que notifica que la neurona ha terminado el proceso anterior y se encuentra disponible para ejecutar un nuevo cálculo.

4.4.2. Estimación del error de computo.

Como se observó en la Figura 4.11, el resultado final de suma de productos es `-127.788896501064`. Para poder hacer una estimación del error generado por la metodología de punto fijo, se realizó una simulación en el entorno de trabajo *Simulink*© perteneciente a la empresa *MATLAB*©, en la cual se hace una comparación cuantitativa en el cálculo de las mismas operaciones para punto fijo y para punto flotante a 32 bits. Dicha simulación puede apreciarse en el Apéndice. En las Tablas 4.2 y 4.3, se observan los resultados de la simulación, donde se aprecia el error obtenido en cada cálculo.

Iteración	Primer factor	Segundo factor	Resultado	Suma acumulativa
1	2.25	1.25	2.8125	2.8125
2	3.14	3.14	9.859	12.672
3	−4.52	2.52	−11.390	1.281
4	5.34	−2.34	−12.495	−11.213
5	−4.14	5.14	−21.279	−32.493
6	4.84	−4.84	−23.425	−55.919
7	5.72	−5.72	−32.718	−88.637
8	−7.5	5.54	−41.55	−130.187
9	1.0	2.4	2.4	−127.787

Tabla 4.2: Resultados del cálculo numérico en punto flotante.

Iteración	Primer factor	Segundo factor	Resultado	Suma acumulativa
1	2.25	1.25	2.8125	2.8125
2	3.14	3.14	9.858	12.671
3	−4.52	2.52	−11.390	1.280
4	5.34	−2.34	−12.496	−11.215
5	−4.14	5.14	−21.278	−32.493
6	4.84	−4.84	−23.426	−55.920
7	5.72	−5.72	−32.718	−88.638
8	−7.5	5.54	−41.550	130.188
9	1.0	2.4	2.399	−127.788

Tabla 4.3: Resultados del cálculo numérico en punto fijo.

Para calcular el error entre ambas metodologías, se procedió a determinar el error medio, mediante la Ecuación 4.1.

$$E = \frac{|V_{fixed} - V_{float}|}{V_{float}} * 100 = \frac{|(-127.788) - (-127.787)|}{-127.787} * 100 = 0.001 \tag{4.1}$$

Donde V_{float} es considerado el valor deseado y V_{fixed} el valor obtenido. Los resultados obtenidos del cálculo del error nos indican que la metodología de computo propuesta para una neurona de 8 entradas y una polarización, es aceptable.

4.4.3. Sumario de recursos lógicos.

Los resursos necesarios para implementar el modelo descrito se muestran en la Tabla 4.4.

Elemento	Unidades usadas	unidades disponibles
LUTs	7	63400
Registros	8	126800
DSP	0	240

Tabla 4.4: Resultados del cálculo numérico en punto fijo.

Los recursos mostrados en la Tabla 4.4, son considerados para la descripción de la neurona[2], implementada en el *chip FPGA* `xc7a100tcsg324-1`. Cabe mencionar que la utilización total en el *chip* es inferior al 0.01 %, pero es importante tener en cuenta que conforme se incrementa el numero de neuronas, también lo hacen la cantidad de bits necesarios para su operación, causando una demanda aún mayor de recursos, conforme escala el numero de capas de un modelo neuronal.

4.5. Conclusiones del capítulo

En este capítulo se describió un modelo secuencial-aritmético que fue propuesto con la intensión de emular una neurona en dispositivos lógicos programables. Se hizo uso de la biblioteca de manejo de operaciones en punto fijo `FIXED_PKG`, debido a sus ventajas con respecto a las operaciones en punto flotante, que, desde el punto de vista de la implementación, consumen mayor cantidad de recursos. Se realizó la simulación del proceso de cálculo de una neurona, lo cual implica considerar el proceso de multiplicación, la suma acumulativa, la suma de polarización y la aplicación de la función de activación, del cual se obtuvieron resultados satisfactorios. Finalmente, se realizó un estudio comparativo, mediante la herramienta de simulación *Simulink*©, en el cual se midió el error obtenido en el modelo propuesto con respecto a un modelo de la misma resolución, pero con metodología flotante. Se determinó que el modelo es viable, pues el error obtenido es inferior al 0.01 %, con lo cual se concluye que los modelos neuronales son factibles de implementar en dispositivos lógicos programables, respetando sus propiedades y fusionando las ventajas de aprendizaje de los modelos neuronales y las capacidades de eficiencia de computo de los dispositivos lógicos programables.

[2] Véase la descripción completa en el apéndice.

CAPÍTULO 5

Experimentación y despliegue de resultados

En este capítulo se estudiará el proceso experimental realizado en este trabajo de manera cuantitativa, mediante el análisis del desempeño de las diferentes configuraciones de modelado y entrenamiento. En las características del modelo se estudiará el desempeño de una arquitectura óptima, cuyo rendimiento cognitivo sea aceptable y cuyo coste computacional sea viable para su ejecución fuera de línea. De igual manera, se analizarán las características matemáticas del proceso de aprendizaje de la red, mediante la selección de métodos de optimización, funciones objetivo, así como la manipulación de los *hiperparámetros*[1], seleccionando aquella relación que ofrezca un grado de aserción conveniente para este trabajo.

5.1. Construcción de la arquitectura

En esta sección se tratarán los aspectos relacionados a la construcción de una arquitectura autoencoder. Como uno de los objetivos de este trabajo es crear la arquitectura más compacta posible, con el fin de poder sintetizar el modelo a una descripción en *Hardware Description Language* (*HDL*, por sus siglas en inglés), se experimentará básicamente con dos configuraciones de autoencoders, los elementales y los profundos.

[1]Por definición los hiperpárametros son aquellos valores que están involucrados en el entrenamiento de un modelo neuronal y son ajenos al modelo, como lo son la razón de aprendizaje y el número de épocas.

5.1.1. Cálculo del tamaño de vector

Para poder experimentar con las arquitecturas es necesario determinar el tamaño de entrada, es decir, cuál será la longitud de cada vector de entrenamiento que se ingresará al modelo, por lo que se tienen básicamente dos criterios, el primero es que la entrada debe ser de la menor dimensión posible, esto con el fin de economizar los recursos del modelo; el segundo criterio es el de incluir la mayor cantidad de información posible para que la red pueda extraer sus sesgos característicos con un mínimo de pérdida de información. Tomando esto en cuenta, se hizo uso de una herramienta de visualización del espectro en frecuencia de la señal presente en la base de datos, con la finalidad de observar en qué rangos de frecuencia se encuentra la mayor cantidad de información.
Este análisis s realizó mediante la descripción de un programa en *Python* (Véase el apéndice en la descripción `fft_experiment`) el cual extrae 1000 muestras de 4 vectores de la base de datos y posteriormente extrae la transformada discreta de Fourier; como resultado se obtuvo la gráfica que se observa en la Figura 5.1.

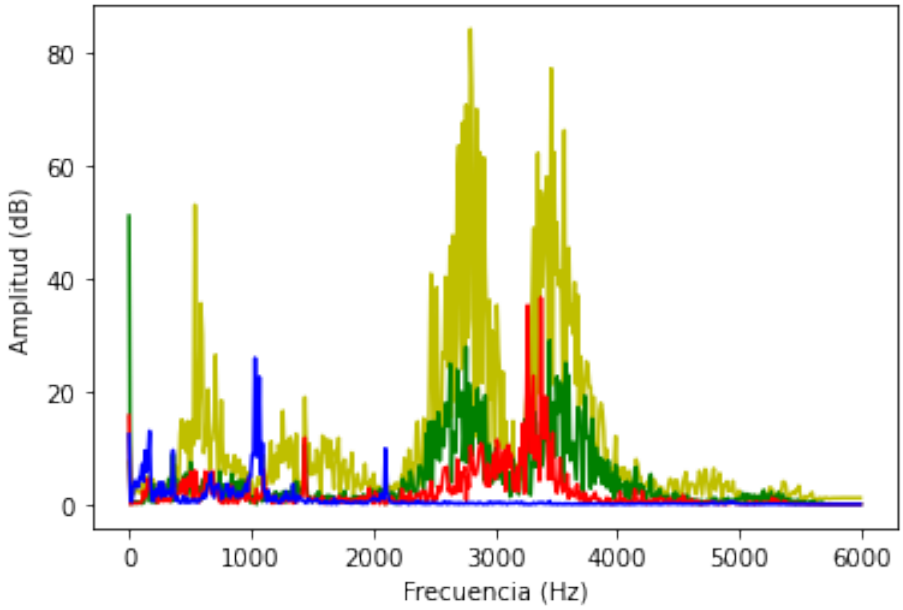

Figura 5.1: Espectros en frecuencia de la base de datos `Normal_0`, `B007_0`, `IR014_0` y `OR021@6_0`.

Con la información visualizada en la figura 5.1, se determinó que un vector de 100 muestras, proporciona un periodo de $83.3\mu s$ a una frecuencia de muestreo de $12kHz$, lo cual da lugar a una frecuencia de vector de 120Hz, suficiente como para incluir a la información de alta frecuencia que se encuentra después de los 1000Hz.

5.1.2. Criterios de desempeño del modelo

Existen muchos criterios para evaluar el desempeño de un modelo neuronal; en el aprendizaje no supervisado éstos suelen ser principalmente dos, la pérdida de validación y de entrenamiento. Ambas métricas calculan el ajuste de las predicciones proporcionadas por el modelo, la diferencia radica en los datos con los que es evaluado dicho modelo. En el caso de la pérdida de entrenamiento, ésta se obtiene evaluando los datos de entrenamiento en el modelo, por lo tanto la pérdida de validación se obtiene evaluando los datos de prueba. Comúnmente ambas medidas se plasman en una gráfica en función del número de épocas, como se observa en la Figura 5.2.

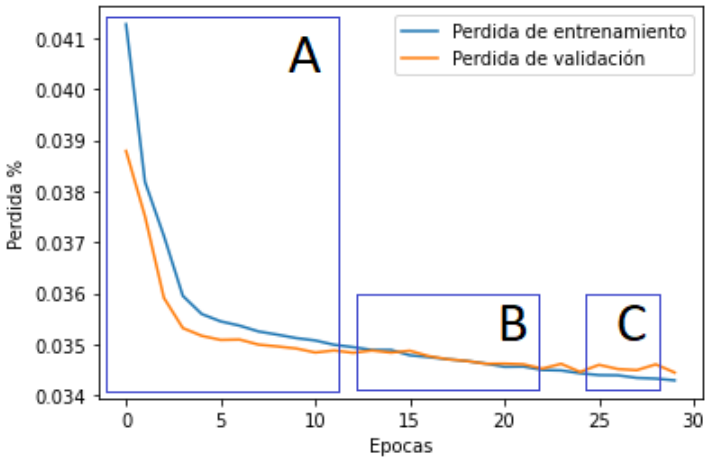

Figura 5.2: Curvas de desempeño. **A)** Desajuste. **B)** Ajuste optimo. **C)** Sobre-ajuste.

Las unidades de la pérdida(unidades del eje vertical del gráfico), dependen de la función objetivo empleada en el entrenamiento. En el ejemplo de la Figura 5.2 se usó la función *Mean Absolute Error* (*MAE*, por sus siglas en inglés), cuya expresión matemática se observa en la Ecuación 5.1.

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j| \tag{5.1}$$

Donde n es el número de muestras, y_i es la i-ésima muestra del conjunto de datos y $\hat{y}_i$ es la predicción del modelo.

El desempeño de un modelo se puede extrapolar de un gráfico de curvas de pérdida; se dice que un modelo está desajustado cuando las curvas de entrenamiento y validación divergen. Por otra parte, se dice que un modelo está aprendiendo eficientemente cuando las curvas de entrenamiento y validación tienden a converger y el sobre ajuste aparece cuando las gráficas sufren una bifurcación después de haber convergido; esto sucede cuando el modelo se avoca a aprender de la información de entrenamiento y ya no es capaz de reconocer datos ajenos a dicho conjunto, en este caso los de validación; en la Figura 5.2 se observan los casos de desajuste (**A**), Ajuste (**B**) y sobreajuste (**C**).

5.1.3. Construcción de un autoencoder elemental

Esta arquitectura está constituida de tres capas: capa de entrada, capa intermedia y capa de salida. En la Figura 5.3, se aprecia el esquema de dicha arquitectura.

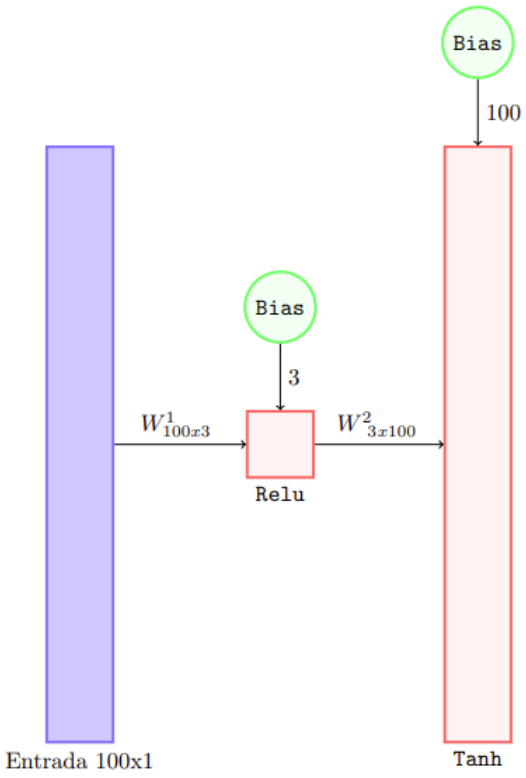

Figura 5.3: Esquema de un autoencoder elemental.

Esta arquitectura tiene la característica de ofrecer un gran aprovechamiento de los recursos, además de proveer un grado de generalización aceptable. La arquitectura tiene los parámetros que se muestran en la Tabla 5.1.

Capa	Número de neuronas	Función de activación	Polarización
Entrada	100	Lineal	No
Oculta	3	*ReLU*	Si
Salida	100	*tanh*	Si

Tabla 5.1: Parámetros de la arquitectura del autoencoder elemental.

Con base en los parámetros especificados en la Tabla 5.1, se realizó la descripción del modelo en lenguaje *Python* de la arquitectura, como se muestra la Sección de código 5.1.

```
latent_dim = 3 # se configura el numero de neuronas en la capa oculta.

class Autoencoder(Model): # se crea la clase que alojara al modelo
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([layers.Flatten(), # capa de entrada
      layers.Dense(latent_dim, activation='relu')]) # capa intermedia
    self.decoder = tf.keras.Sequential([
      layers.Dense(100, activation='tanh')]) # capa de salida

  def call(self, x):
    encoded = self.encoder(x)    # funcion del encoder
    decoded = self.decoder(encoded) # funcion del decoder
    return decoded

autoencoder = Autoencoder(latent_dim)
```

Sección de código 5.1: Descripción de un autoencoder elemental.

Una vez descrito el modelo, es necesario llevar a cabo el proceso de compilación en el cual se especifica el método de optimización que se empleará para su entrenamiento, así como la función objetivo que se usará en el proceso de entrenamiento. Este proceso se realiza mediante el atributo `.compile` de modelo neuronal construido como se muestra en la Sección de código 5.2.

```
autoencoder.compile(optimizer=keras.optimizers.Adam(1e-4), loss='mae')
```

Sección de código 5.2: Compilación del modelo.

Como se observa en la Sección de código 5.2, se configuró el método de optimización *Adam*[20, Capítulo 12.10] y la función objetivo *MAE*, cuya expresión matemática tiene la forma de la Ecuación 5.1.

Metodología de entrenamiento iterativo

Dada la naturaleza aleatoria de los procesos de aprendizaje computacional, cuando se realiza mas de un entrenamiento a un modelo, los resultados de cada entrenamiento suelen diferir sustancialmente en los resultados de uno con respecto al otro, por lo que para comprobar el desempeño de una arquitectura, es necesario estimar un comportamiento promedio a través de diversos entrenamientos. Para llevar a cabo este proceso estadístico, se realizo un experimento para entrenar la red descrita, durante 10 veces con la finalidad de evaluar los valores de pérdida.

En los modelos de *Tensor Flow* no es posible realizar el proceso de entrenamiento de manera iterativa simplemente ingresando las instrucciones de entrenamiento en un lazo cíclico, debido a que cuando se hace esto el programa reutiliza los parámetros obtenidos del entrenamiento previo, por lo que es necesario restablecer la memoria de la descripción volviendo a ejecutarla. Para solucionar este problema se optó por hacer uso del atributo `.set_weights` el cual le asigna al modelo un método[2] que contiene las matrices y vectores de polarización con valores determinados. En la Sección de código 5.3 se observa las instrucción de asignación de pesos.

```
autoencoder.set_weights(new_weights)
```

Sección de código 5.3: Asignación de parámetros al objeto `autoencoder`.

Las matrices de pesos y los vectores de polarización deben tener valores aleatorios uniformemente distribuidos. Para crear los valores con las distribuciones mencionadas se hizo uso de la instrucción de *Tensor Flow* `tf.random.uniform`. En la Sección de código 5.4 se observa la creación de una matriz de dimensiones `shape`, cuyos valores no serán menores que `minval` ni mayores a `maxval`; los valores son del tipo flotante de 32 bits.

```
w_1 = tf.random.uniform(shape,minval=minval,maxval=minval,dtype=tf.dtypes.float32,
seed=None,name=None)
```

Sección de código 5.4: Declaración de una lista para configurar los parámetros.

El uso de las dos instrucciones mencionadas anteriormente, hizo posible la creación de una clase que pudiera realizar el proceso de entrenamiento sin necesidad de restablecer el *kernel*[3].

[2]Un método en *Python* hace referencia a una función que pertenece a un objeto.

[3]El *Kernel* es un programa primitivo que se encuentra en el núcleo del sistema operativo y controla todos los procesos dentro de él.

En la Figura 5.4 se plasma una familia de curvas de la pérdida de entrenamiento obtenida en 10 diferentes entrenamientos, cuyo comportamiento promedio es remarcado con puntos azules.

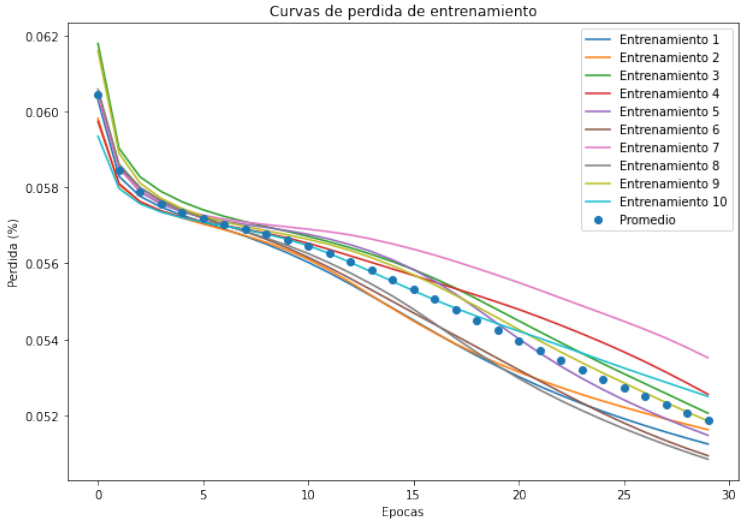

Figura 5.4: Familia de curvas obtenidas de los valores de pérdida en el entrenamiento para 10 casos.

De manera análoga a la imagen anterior, en la Figura 5.5 se aprecia una familia de curvas correspondiente a la pérdida de validación.

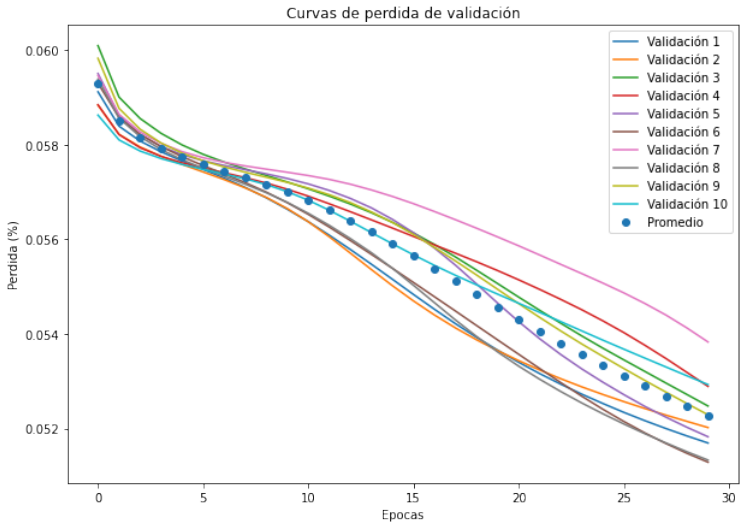

Figura 5.5: Familia de curvas obtenidas de los valores de pérdida en la validación para 10 casos.

En el proceso iterativo se obtuvieron valores promedio para entrenamiento y validación de `0.0518` y `0.0522` respectivamente cuyos mejores valores son `0.0508` y `0.0512`.

5.1.4. Construcción de un autoencoder profundo

Por definición, un autoencoder profundo está constituido de más de una capa oculta, motivo por cual es referido por algunos autores como una variante de perceptrón multicapa. En este trabajo será modelado un autoencoder profundo basado en la primera arquitectura. Para construir un autoencoder profundo en este trabajo, uno de los criterios a considerar es que los tamaños de las capas de entrada y oculta se mantienen constantes, por lo que las dimensiones a calcular son las de las capas apiladas en los flancos del cuello de botella. Mediante experimentación se concluyó que apilar dos pares de capas en los flancos del cuello de botella es la mejor opción, en términos de generalización y error de reconstrucción.

Para determinar el número de neuronas de las dos capas posteriores a la de entrada se creó una relación exponencial entre el número de neuronas de la capa intermedia y la de salida, como se observa en la Figura 5.6. La magnitud de interés a calcular es el dominio, puesto que el contradominio se modeló como se muestra en las Ecuaciones 5.2 y 5.3.

$$e^{x_{hidden}} = 3 \Rightarrow \quad x_{hidden} = ln(3) = 1.098 \tag{5.2}$$

$$e^{x_{in}} = 100 \Rightarrow \quad x_{in} = ln(100) = 4.605 \tag{5.3}$$

Haciendo la diferencia entre n_{hidden} y n_{in} se obtuvo la distancia entre ambos puntos. Como se muestra en la Ecuación 5.4.

$$l_{total} = x_{in} - x_{hidden} = 3.506 \tag{5.4}$$

para hacer que los cuatro puntos sean equidistantes se dividió la distancia total entre tres y usando 5.4, se determinaron los dominios como se muestra en la ecuaciones 5.5. y 5.6.

$$x_{\text{1st sublayer}} = \frac{l_{total}}{3} + x_{hidden} = 2.267 \tag{5.5}$$

$$x_{\text{2nd sublayer}} = \frac{l_{total} * 2}{3} + x_{hidden} = 3.436 \tag{5.6}$$

Por lo tanto el número de neuronas es calculado mediante las Ecuaciones 5.7 y 5.8.

$$n_{\text{1st layer}} = e^{x_{\text{1st sublayer}}} = 9.654 \tag{5.7}$$

$$n_{\text{2nd layer}} = e^{x_{\text{2nd sublayer}}} = 31.072 \tag{5.8}$$

De 5.7 y 5.8 se determina que las dimensiones de las capas se redondean a 10 y 31 respectivamente; en la Figura 5.6 se aprecia la relación exponencial considerada, con separaciones equidistantes en el eje horizontal.

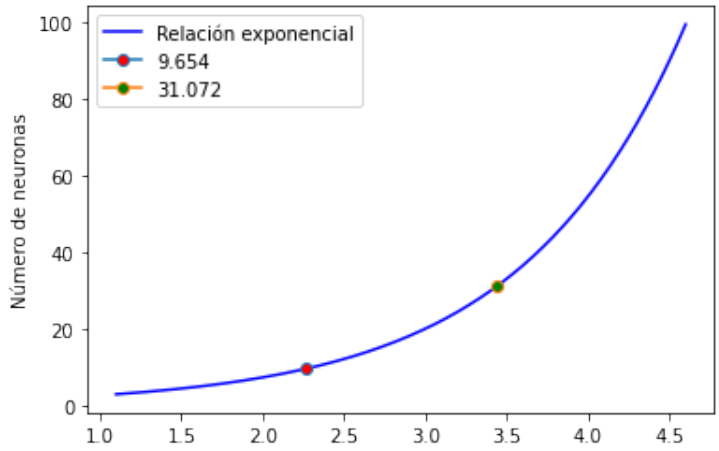

Figura 5.6: Relación exponencial entre el número de neuronas de las capas apiladas.

Una vez determinadas las dimensiones de todas las capas de la arquitectura se procede a construirla con la información obtenida. En la Tabla 5.2 se muestra el sumario de recursos y parámetros que se emplearán en el modelo.

Capa	Número de neuronas	Función de activación	Polarización
Entrada	`100`	Lineal	No
Capa oculta 1	`31`	*ReLU*	Si
Capa oculta 2	`10`	*ReLU*	Si
Capa oculta 3 (cuello de botella)	`3`	*ReLU*	Si
Capa oculta 4	`10`	*ReLU*	Si
Capa oculta 5	`31`	*ReLU*	Si
Salida	`100`	*Tanh*	Si

Tabla 5.2: Tabla de tipos de capa.

Con base en los datos mostrados en la Tabla 5.2, se construyó un modelo cuya arquitectura general se muestra en la Figura 5.7; ahí se aprecia la organización de las capas (rectángulos), sus funciones de activación (parte inferior de la capa), de las matrices de pesos sinápticos que las relacionan denotadas por W^i_{nxm}, así como sus respectivos vectores de polarización (círculos).

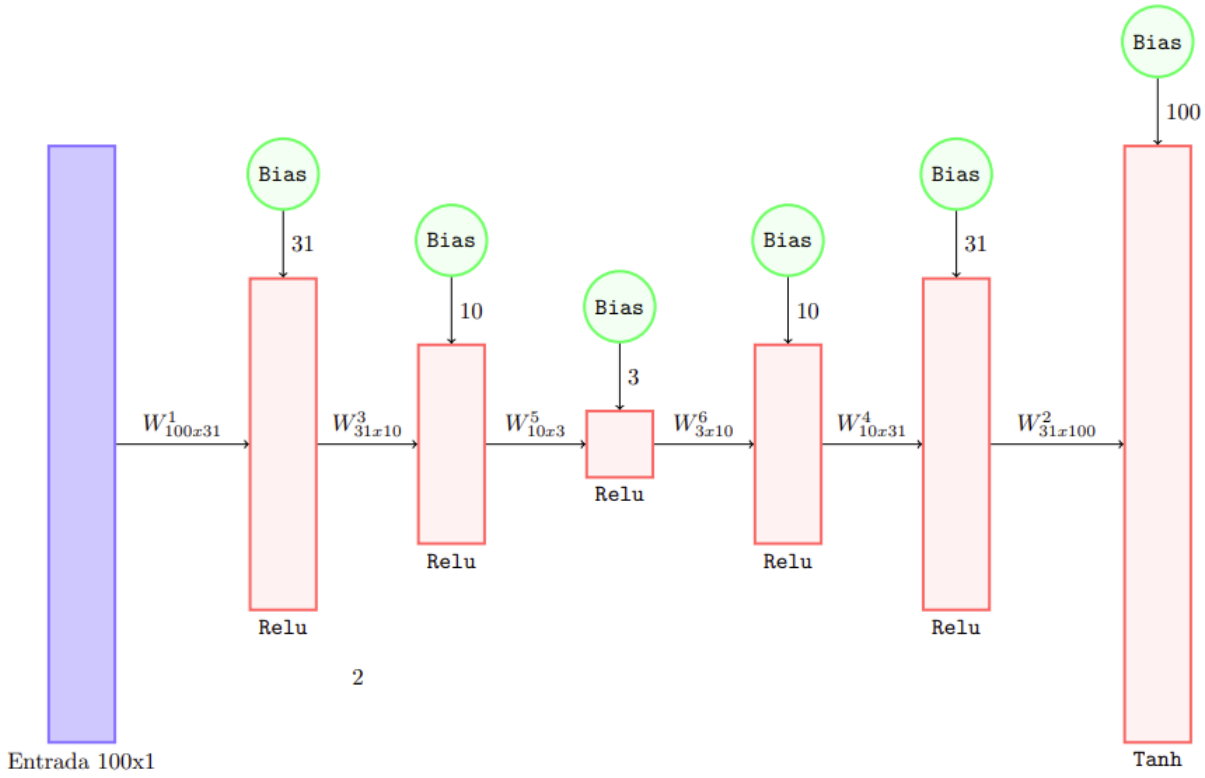

Figura 5.7: Arquitectura general de autoencoder profundo.

La descripción del modelo considerará la descripción de la arquitectura mediante la unión de dos modelos, el encoder y el decoder, con la intención de poder extraer fácilmente la información de la estructura interna; el modelo del decoder contará con las 4 primeras capas, desde la capa de entrada hasta el cuello de botella y el decoder contendrá tres capas, desde la cuarta capa hasta la capa de salida. Con el fin de obtener un mayor grado de organización en el entorno de trabajo, primeramente se crea una lista[4] con pares de valores que corresponden al número de neuronas y función de activación de cada capa, como se observa en la Sección de código 5.5.

```
# Se crea una lista con el tamano y funcion de activacion de cada capa
network_parameters = [[31,'relu'], # parametros de la capas 1 y 5
                      [10,'relu'], # parametros de la capas 2 y 4
                      [3,'relu'], # parametros de la capa 3
                      [100,'tanh']]# parametros de la capa de salida
```

Sección de código 5.5: Declaración de una lista para configurar los parámetros.

[4]En *Python* una lista es un arreglo n-dimensional que puede contener diferentes tipos de datos como caracteres y números enteros.

En la Sección de código 5.6 se observa la descripción del modelo en *Python*.

```
# Construir el modelo
class Autoencoder(Model): # Se crea el modelo neuronal
 def __init__(self, network_parameters): # Se importa la lista con los parametros
     super(Autoencoder, self).__init__()
     self.latent_dim = network_parameters[2][0]
     self.encoder = tf.keras.Sequential([ # se construye el submodelo encoder
       layers.Flatten(),
       layers.Dense(network_parameters[0][0], activation=network_parameters[0][1]),
       layers.Dense(network_parameters[1][0], activation=network_parameters[1][1]),
       layers.Dense(network_parameters[2][0], activation=network_parameters[2][1])])
     self.decoder = tf.keras.Sequential([ # se construye el submodelo decoder
       layers.Dense(network_parameters[1][0], activation=network_parameters[1][1]),
       layers.Dense(network_parameters[0][0], activation=network_parameters[0][1]),
       layers.Dense(network_parameters[3][0], activation=network_parameters[3][1])])
 def call(self, x):
     encoded = self.encoder(x) # se declara el submodelo encoder
     decoded = self.decoder(encoded) # se declara el submodelo decoder
     return decoded
```

Sección de código 5.6: Descripción de un autoencoder profundo.

Después de que el modelo ha sido creado, se procede a compilarlo. En este paso se le especifica el método de optimización que será empleado en el entrenamiento, la función objetivo a minimizar y el índice de aprendizaje; en este ejemplo se usó el método de optimización *Adam*, con una función objetivo *MAE*, y un índice de aprendizaje de `0.01`, como se muestra en la Sección de código 5.7.

```
autoencoder.compile(optimizer=keras.optimizers.Adadelta(0.001), loss='mae')
```

Sección de código 5.7: Compilación del modelo.

El entrenamiento del modelo se realiza como se muestra en la Sección de código 5.8.

```
autoencoder.fit(train_data, train_data,epochs=30,batch_size=12,
                validation_data=(test_data,test_data),shuffle=True)
```

Sección de código 5.8: Entrenamiento del modelo.

El proceso de entrenamiento se realizó 10 veces con el fin de obtener un comportamiento promedio. Las curvas de pérdida de entrenamiento y validación se muestran en las Figuras 5.8 y 5.9, respectivamente.

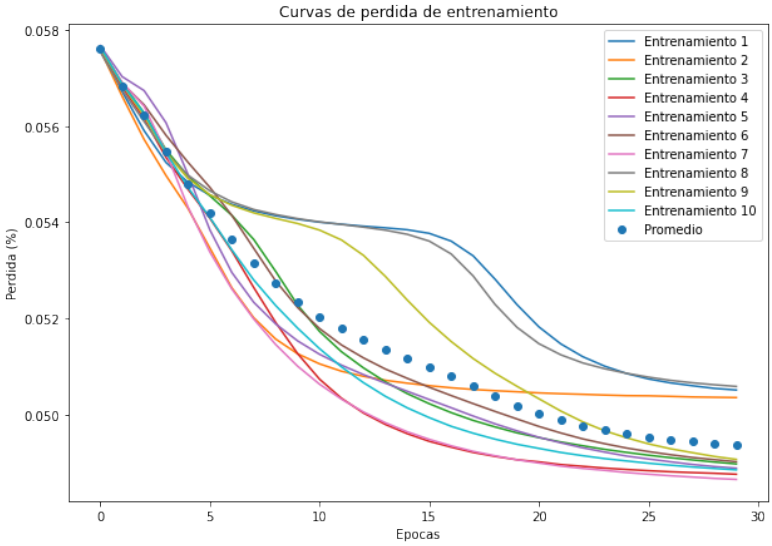

Figura 5.8: Familia de curvas obtenidas de los valores de pérdida en el entrenamiento para 10 casos.

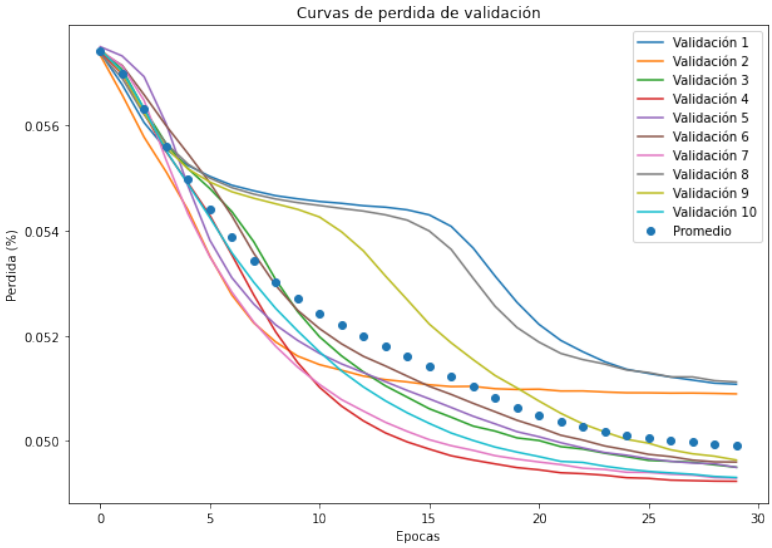

Figura 5.9: Familia de curvas obtenidas de los valores de pérdida en la validación para 10 casos.

Las curvas de entrenamiento y validación nos proporcionan valores promedio finales de `0.0493` y `0.0499`, respectivamente, cuyos mejores valores (valores mínimos) son `0.0486` y `0.0492`. Esta familia de curvas se generó usando el primer conjunto de datos, es decir `Normal_0`, `B007_0`, `IR007_0` y `OR007@6_0`.

Método de transferencia de aprendizaje y sintonización fina.

Como se observó en las Figuras 5.8 y 5.9, el aprendizaje del modelo tiende a ser un poco inestable, esto, debido a la inicialización de los pesos sinápticos y polarizaciones; en cada entrenamiento a las matrices de pesos sinápticos y los vectores de polarización se les asignan valores aleatorios, con determinadas distribuciones[5] y por tanto en cada entrenamiento se tienen condiciones iniciales diferentes, lo que ocasiona que en algunos entrenamientos la optimización se estanque en algún mínimo local. Esta clase de comportamientos es normal en las arquitecturas profundas debido a la secuencia del algoritmo *Back Proagation*, el cual prioriza el aprendizaje en las capas externas, haciendo que la influencia del entrenamiento se atenúe conforme el entrenamiento avanza hacia las capas más internas. Los métodos de transferencia de aprendizaje y ajuste fino tienden a mejorar el comportamiento del entrenamiento mediante una serie de pasos, que a continuación se explicarán.

1) Se crea una primera arquitectura con las capas de entrada y salida y se configura como capa intermedia la penúltima de la arquitectura deseada.

Con la intención de construir el modelo secuencialmente con parámetros que coincidan exactamente con los de la arquitectura final.

2) Se entrena la arquitectura creada con un índice de aprendizaje alto[6].

Se entrena la primer arquitectura con un índice de aprendizaje alto, pues se desea que el entrenamiento inicialice los parámetros de pesos sinápticos y polarizaciones con valores afines al los conjuntos de entrenamiento y prueba, por lo que en este paso obtener buenos valores de pérdida no es una prioridad.

3) Se añaden capas extra al modelo creado, y se mantienen inalterados los parámetros obtenidos en entrenamiento previo.

Esto con el fin de transferir el aprendizaje obtenido del entrenamiento previo, a las capas recién añadidas, éstas se apilan a la arquitectura inicial, obteniendo así información con cierto grado de discernimiento.

[5] En *TensorFlow* suelen ser distribuciones normales y uniformes.

[6] El índice de aprendizaje tiene como valor máximo 1 y 0 como valor mínimo, idealmente hablando por lo que un valor alto es cercano a 1.

Cabe mencionar que la adición de las capas se debe planear de tal manera que las matrices de pesos y los vectores de polarización coincidan en dimensiones con las capas subsecuentes, con el fin de hacer coincidir las formas de todas las capas de la arquitectura deseada.

4) Se entrenan las capas recién añadidas, manteniendo inalterados los parámetros de las capas pre-entrenadas.

Las capas recién añadidas se entrenan con un índice de aprendizaje menor con respecto al usado en las capas previas.

5) Se repiten los pasos 3 y 4 hasta obtener la arquitectura deseada.

6) Se realiza el ajuste fino a la arquitectura.

El ajuste fino consiste en liberar los parámetros obtenidos en los entrenamientos previos, haciéndolos modificables y se entrena la arquitectura completa con un índice de aprendizaje pequeño.

La construcción por etapas del modelo de la Figura 5.7, se realizará siguiendo los pasos ya mencionados. En la Sección de código 5.9 se observa la creación de el primer modelo, el cual está constituido de la capas de entrada, oculta de 31 neuronas y salida, como lo indica el paso **1**.

```
class Autoencoder_1(Model):
  def __init__(self):
    super(Autoencoder, self).__init__()
    self.encoder = tf.keras.Sequential([layers.Flatten(),
      layers.Dense(31, activation='relu')])
    self.decoder = tf.keras.Sequential([
      layers.Dense(100, activation='tanh')])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder_1 = Autoencoder_1()
```

Sección de código 5.9: Descripción del primer modelo.

En la Figura 5.10 se observa el esquema de la arquitectura descrita en la Sección de código 5.9.

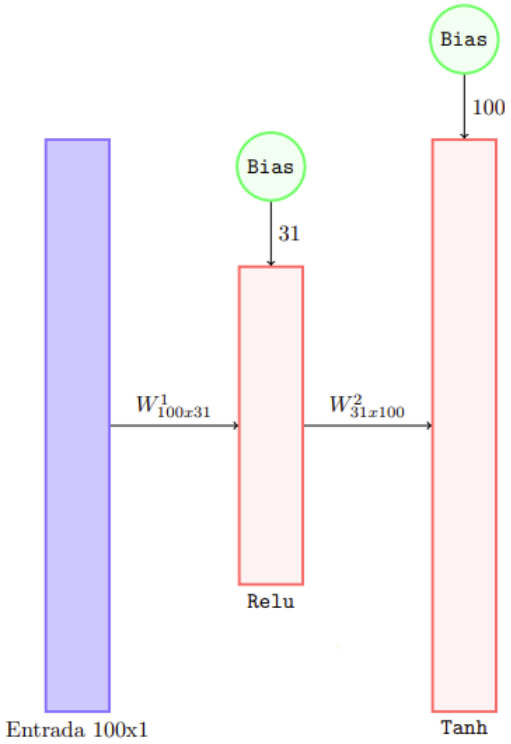

Figura 5.10: Primer autoencoder.

Siguiendo el paso **2**, se realizará el entrenamiento, primeramente compilando el modelo como se observa en la Sección de código 5.10, configurando un método de optimización basado en gradiente, por sus características de transferencia de información diferencial.

```
autoencoder_1.compile(optimizer=keras.optimizers.Adadelta(0.001), loss='mae')
```

Sección de código 5.10: Compilación del primer modelo.

Posteriormente, se entrena el modelo como se observa en la Sección de código 5.11.

```
autoencoder_1.fit(train_data, train_data,epochs=30,batch_size=12,
              validation_data=(test_data,test_data),shuffle=True)
```

Sección de código 5.11: Entrenamiento del primer modelo.

Siguiendo lo indicado en el paso **3**, primeramente se procede a declarar los parámetros obtenidos del primer entrenamiento, inalterados; esto se consigue mediante el atributo `trainable`, del objeto que aloja al modelo mediante la asignación de un valor binario, siendo *False* la indicación de congelación de parámetros y *True* la indicación de su liberación. La inhabilitación de la alterabilidad de los parámetros del primer modelo se describe en la Sección de código 5.12.

```
autoencoder_1.encoder.trainable = False
autoencoder_1.decoder.trainable = False
```

Sección de código 5.12: Congelamiento de parámetros del primer modelo.

Unas vez congelados los parámetros del primer modelo, se procede a crear un segundo modelo, el cual hará uso de las capas y los parámetros del primero, apilando un nuevo par de capas, como lo indicado en el paso **3**. En la Sección de código 5.13 se muestra la descripción del apilamiento de las capas.

```
class Autoencoder_2(Model):

def __init__(self):
    super(Autoencoder_2, self).__init__()
    # Se usa como primera capa el encoder del primer modelo
    self.encoder = tf.keras.Sequential([autoencoder.encoder,
    # Se apila una nueva capa de 10 neuronas al encoder
    layers.Dense(10, activation='relu')])
    # Se apila una nueva capa de 31 neuronas al decoder
    self.decoder = tf.keras.Sequential([layers.Dense(31,activation='relu'),
    # Se usa como capa de salida el decoder del primer modelo
    autoencoder.decoder])

def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder_2 = Autoencoder_2()
```

Sección de código 5.13: Descripción del segundo modelo.

En la Figura 5.11 se observa el esquema del modelo creado en la Sección de código 5.13, donde se aprecian los parámetros inalterables de pesos y polarizaciones en color azul y los nuevos (entrenables) en color negro.

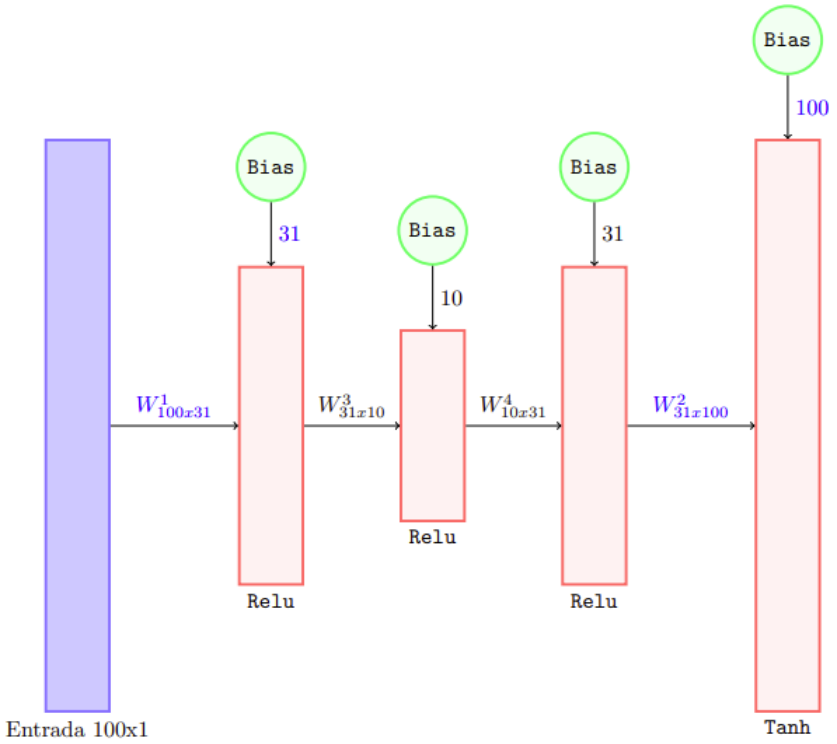

Figura 5.11: Segundo modelo. **Parámetros color azul:** Variables congeladas. **Parámetros color negro:** Variables entrenables.

Se compila el segundo modelo configurando con un índice de aprendizaje menor que en el modelo anterior, como se observa en la Sección de código 5.14.

```
autoencoder_2.compile(optimizer=keras.optimizers.Adadelta(0.001), loss='mae')
```

Sección de código 5.14: Compilación del segundo modelo.

Posteriormente se entrena el modelo como se observa en la Sección de código 5.15.

```
autoencoder_2.fit(train_data, train_data,epochs=30,batch_size=12,
                  validation_data=(test_data,test_data),shuffle=True)
```

Sección de código 5.15: Entrenamiento del segundo modelo.

Una vez construido y entrenado el segundo modelo, se procede a agregar dos capas, lo cual creará la arquitectura deseada. Para construir el modelo, primero es necesario congelar los parámetros del segundo modelo como se muestra en la Sección de código 5.16.

```
autoencoder_2.encoder.trainable = False
autoencoder_2.decoder.trainable = False
```

Sección de código 5.16: Congelamiento de parámetros del segundo modelo.

La descripción del tercer modelo se observa en la Sección de código 5.17.

```
class Autoencoder_3(Model):

def __init__(self):
    super(Autoencoder_3, self).__init__()
    # Se usa como base el encoder del segundo modelo
    self.encoder = tf.keras.Sequential([autoencoder_2.encoder,
    # Se apila una nueva capa de 3 neuronas
    layers.Dense(3, activation='relu')])
    # Se apila una nueva capa de 10 neuronas al decoder
    self.decoder = tf.keras.Sequential([layers.Dense(10,activation='relu'),
    # Se usa como capas de salida el decoder del segundo modelo
    autoencoder_2.decoder])

def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder_3 = Autoencoder_3()
```

Sección de código 5.17: Descripción del tercer modelo.

El esquema del modelo descrito en la Sección de código 5.17, se observa en la Figura 5.12, con las variables congeladas en color azul y las variables entrenables en color negro.

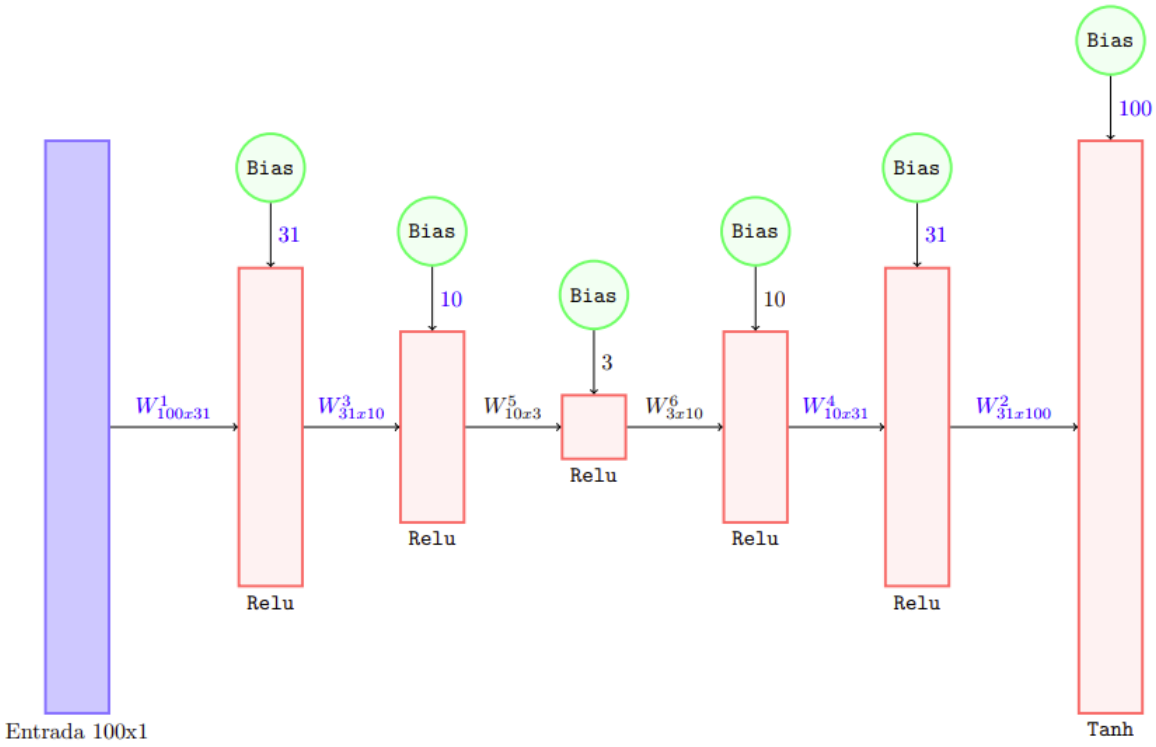

Figura 5.12: Tercer modelo. **Parámetros color azul:** Variables congeladas. **Parámetros color negro:** Variables entrenables.

Nuevamente se entrena el modelo recién creado, como se observa en la Sección de código 5.18.

```
autoencoder_3.compile(optimizer=keras.optimizers.Adadelta(0.001), loss='mae')
```

Sección de código 5.18: Compilación del tercer modelo.

Posteriormente se entrena el tercer modelo como se observa en la Sección de código 5.15.

```
autoencoder_3.fit(train_data, train_data,epochs=30,batch_size=12,
                  validation_data=(test_data,test_data),shuffle=True)
```

Sección de código 5.19: Entrenamiento del tercer modelo.

Una vez entrenadas las capas recién agregadas al modelo deseado, se liberan los parámetros mediante el atributo `trainable = True`, como se observa en la Sección de código 5.20.

```
autoencoder_3.encoder.trainable = True
autoencoder_3.decoder.trainable = True
```

Sección de código 5.20: Liberación de parámetros de la arquitectura deseada.

```
autoencoder_3.compile(optimizer=tf.keras.optimizers.Adam(5e-4), loss='mae')
```

En la figura 5.13 se observa la curva de los 4 entrenamientos, cuyos comportamientos de las curvas son decrecientes, lo que indica que se está llevando a cabo correctamente la minimización.

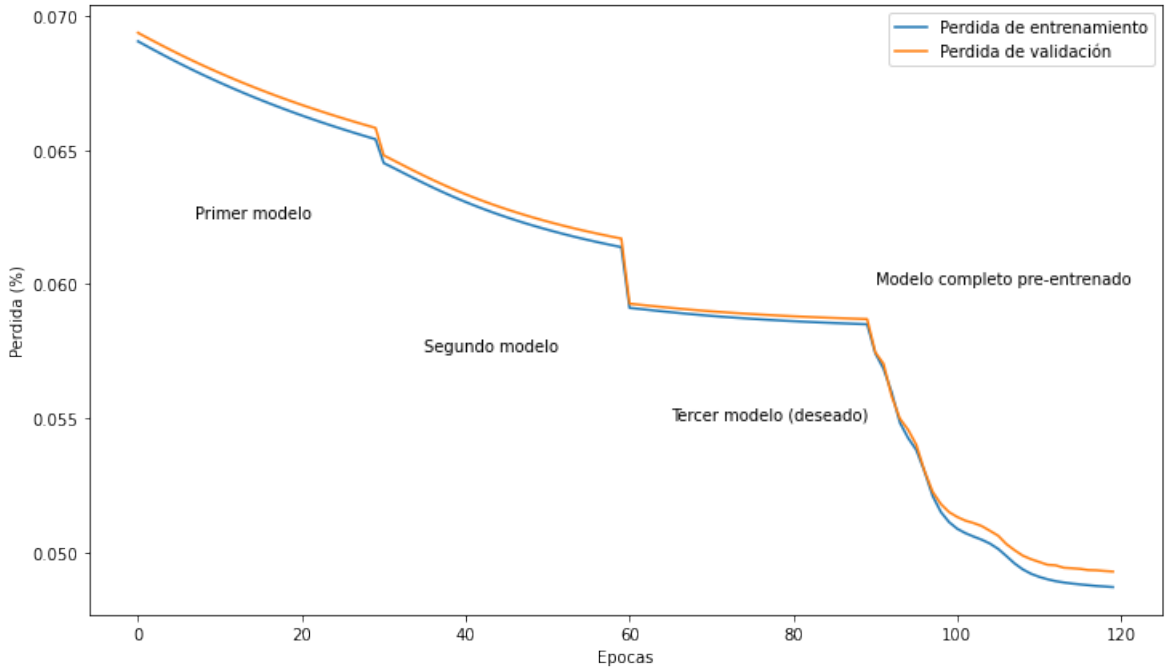

Figura 5.13: Curva de entrenamiento de las 4 etapas.

El proceso de entrenamiento se realizó bajo las condiciones descritas en la Tabla 5.3.

Entrenamiento	Número de capas del modelo	Número de capas entrenables	Método de optimización	Índice de aprendizaje
1	3	3	*Adadelta*	0.001
2	5	2	*Adadelta*	0.001
3	7	2	*Adadelta*	0.001
4	7	7	*Adam*	$5e-4$

Tabla 5.3: Tabla de tipos de capa.

Evaluación del desempeño del autoencoder mediante reconstrucción y *clustering*

Uno de los criterios que ayudan a identificar el grado de generalización de una arquitectura no supervisada, suele ser la reconstrucción y el análisis de la información de las estructuras internas. En la tabla 5.4 se observan los archivos con los que fue entrenado el modelo, los cuales corresponden a muestras saludables y con fallas de 0.007", así como los valores finales de pérdida en entrenamiento y validación.

Archivos empleados	Pérdida en entrenamiento y validación del modelo
`Normal_0`,`B007_0`, `IR007_0`,`OR007@6_0`	Entrenamiento: `0.0476`, Validación: `0.0487`

Tabla 5.4: Condiciones experimentales del entrenamiento para datos saludable y con fallas de 0.007".

La reconstrucción proporcionada por el modelo se observa en la Figura 5.14; se muestran dos gráficas, la roja corresponde a la señal original y la azul a su reconstrucción.

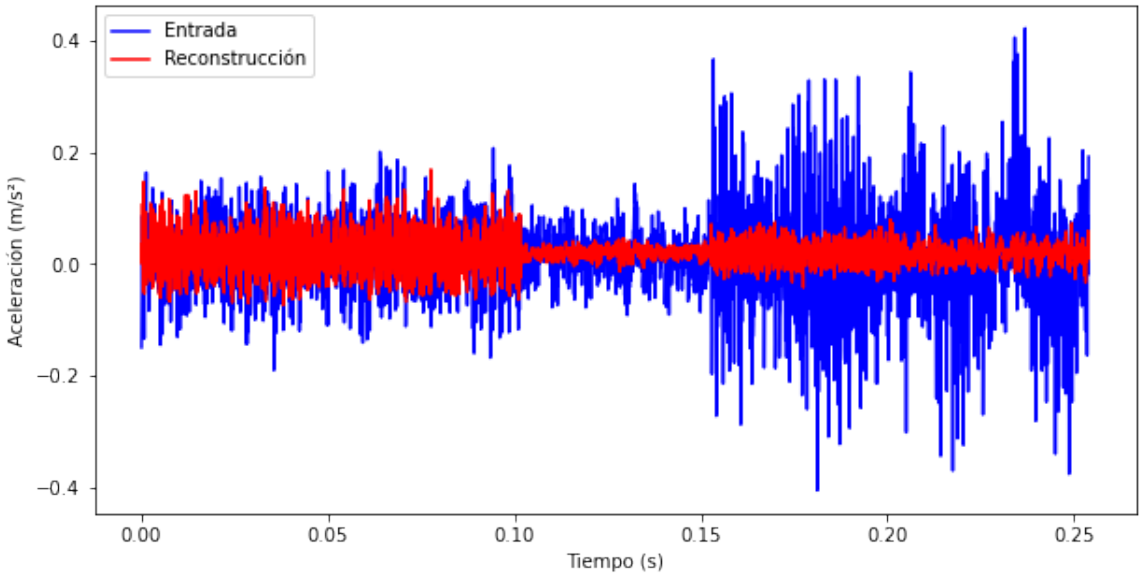

Figura 5.14: Gráficas de señal original (**gráfica azul**) y reconstrucción (**gráfica roja**).

En la Figura 5.14 se observa una reconstrucción poco fiel, pero los resultados resultan razonables al presentar la reconstrucción un grado de discernimiento entre conjuntos[7].

[7]La señal original está compuesta de datos saludables y con falla; esto se puede apreciar visualmente a través de las amplitudes en la señal original, cuya reconstrucción también sigue el patrón de atenuación y amplificación.

La visualización de los conjuntos, también conocida como *Clustering*, consiste en asignar etiquetas de clase a la información de entrada y asociar la representación reducida proporcionada por la estructura interna del autoencoder con dichas etiquetas. En la Figura 5.15 se observa un ejemplo de lo ya mencionado mediante la representación en el espacio de la salida numérica de las tres neuronas que se encuentran en la capa intermedia de la arquitectura. Los vectores color rojo corresponden a muestras saludables mientras que los vectores azules son muestras que tienen algún tipo de falla.

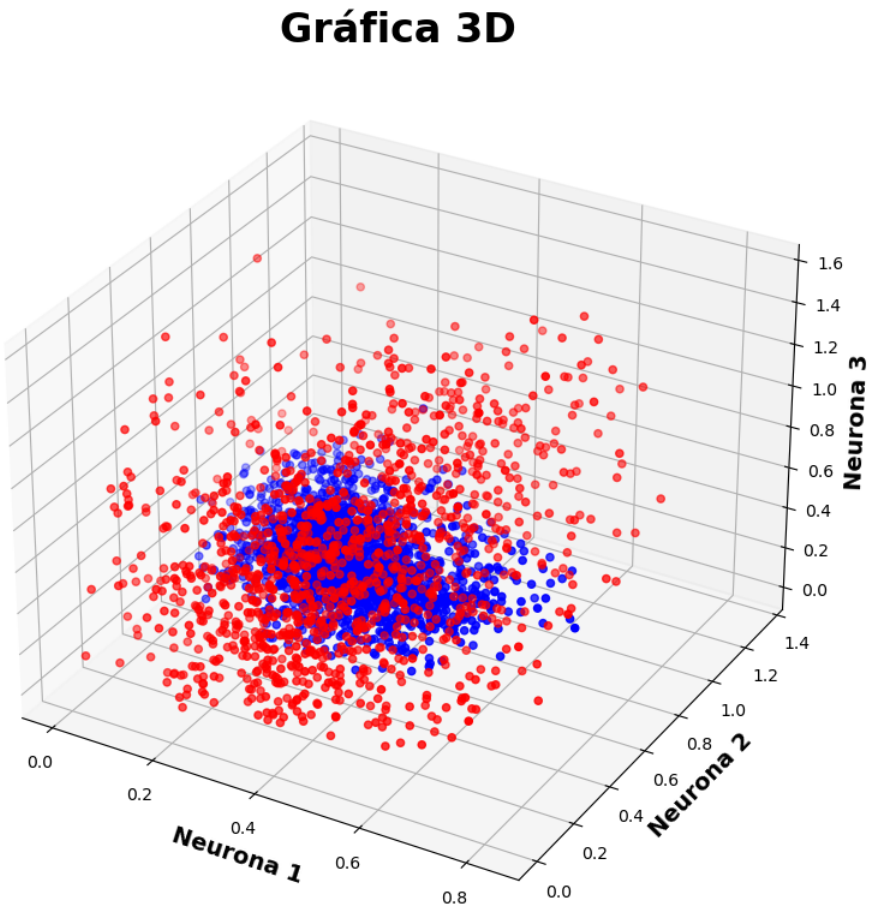

Figura 5.15: Gráfico tridimensional de los conjuntos discernidos por el modelo. Los vectores color **rojo** corresponden a muestras saludables y los **azules** a muestras con algún tipo de falla.

En la Figura 5.15 se observa un agrupamiento bastante notorio y consistente con los resultados esperados; cabe mencionar que las etiquetas agregadas no son consideradas en el entrenamiento y por tanto la agrupación visualizada es generada por el proceso de extracción de características de la arquitectura autoencoder.

El análisis de reconstrucción y *clustering* realizados para los conjuntos especificados en la Tabla 5.4, se realizará de manera análoga para los datos saludables y con falla de 0.014", cuyas condiciones experimentales se aprecian en la Tabla 5.5.

Archivos empleados	Pérdida en entrenamiento y validación de la compresión
`Normal_0`,`B014_0`, `IR014_0`,`OR014@6_0`	Entrenamiento: `0.0337`, Validación: `0.0339`

Tabla 5.5: Condiciones experimentales del entrenamiento para datos saludable y con fallas de 0.014".

En la Figura 5.16 se observa la reconstrucción del modelo, generada a partir de la información proporcionada en la Tabla 5.5.

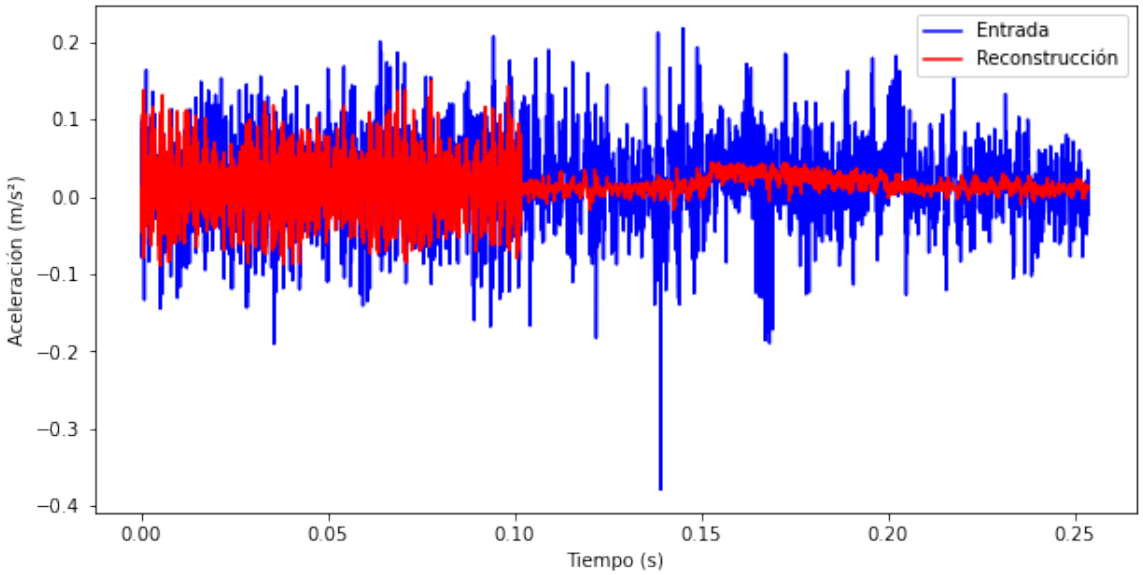

Figura 5.16: Gráficas de señal original (**gráfica azul**) y reconstrucción (**gráfica roja**).

En la Figura 5.16 se observa una reconstrucción aceptable para las muestras saludables, mientras que en los datos que presentan fallas, se aprecia una reconstrucción pobre desde el punto de vista de amplitud. Por otra parte, tiene una correcta interpretación del centro de la señal original.

En la Figura 5.17 se aprecia la agrupación generada por el modelo; se observa que el grado de separabilidad generado por el modelo es muy bueno, lo cual indica un correcto aprendizaje de la red.

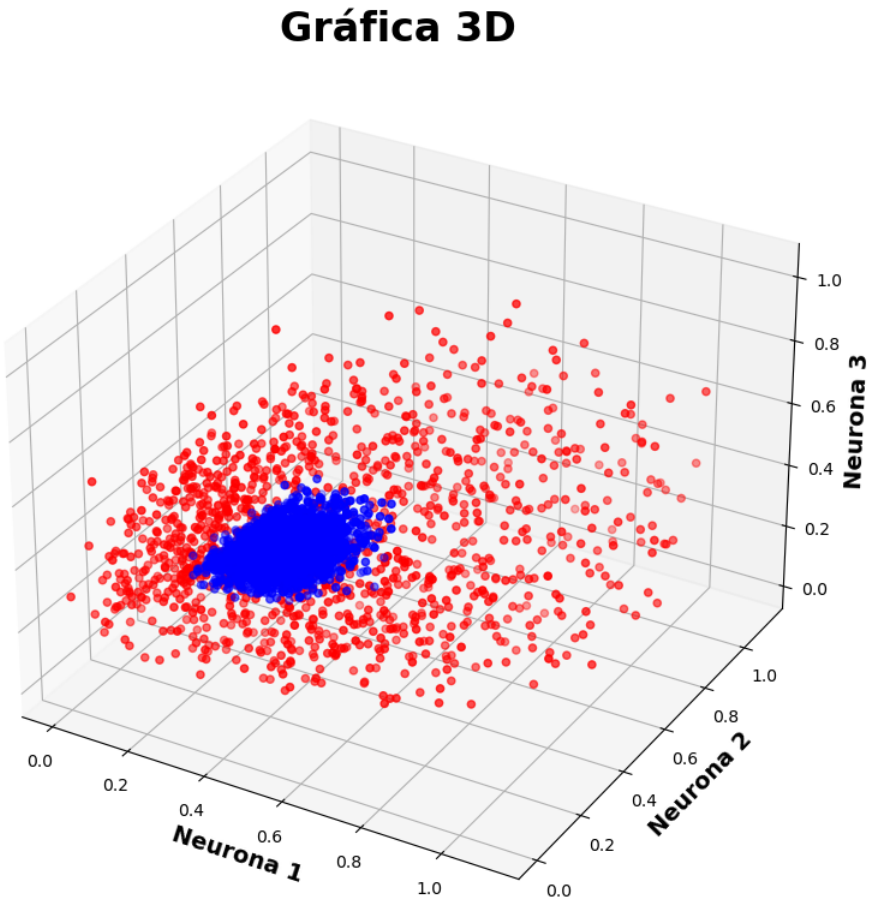

Figura 5.17: Gráfico tridimensional de los conjuntos discernidos por el modelo. Los vectores color **rojo** corresponden a muestras saludables y los **azules** a muestras con algún tipo de falla.

La evaluación del desempeño para los datos saludables y de fallas de 0.021" se realizó con la información de la Tabla 5.6.

Archivos empleados	Pérdida en entrenamiento y validación de la compresión
`Normal_0,B021_0, IR021_0,OR021@6_0`	Entrenamiento: `0.0511`, Validación: `0.0526`

Tabla 5.6: Condiciones experimentales del entrenamiento para datos saludable y con fallas de 0.021".

En la Figura 5.18 se aprecian las gráficas de las muestras saludables y con falla de 0.021". La

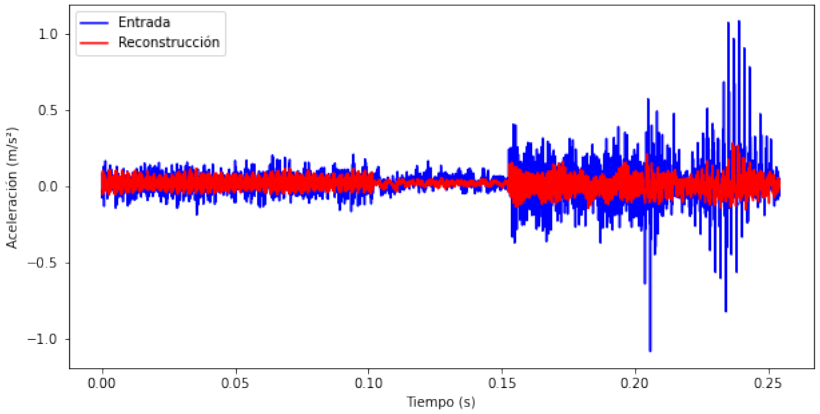

Figura 5.18: Gráficas de señal original (**gráfica azul**) y reconstrucción (**gráfica roja**).

agrupación generada por la red se observa en la Figura 5.19.

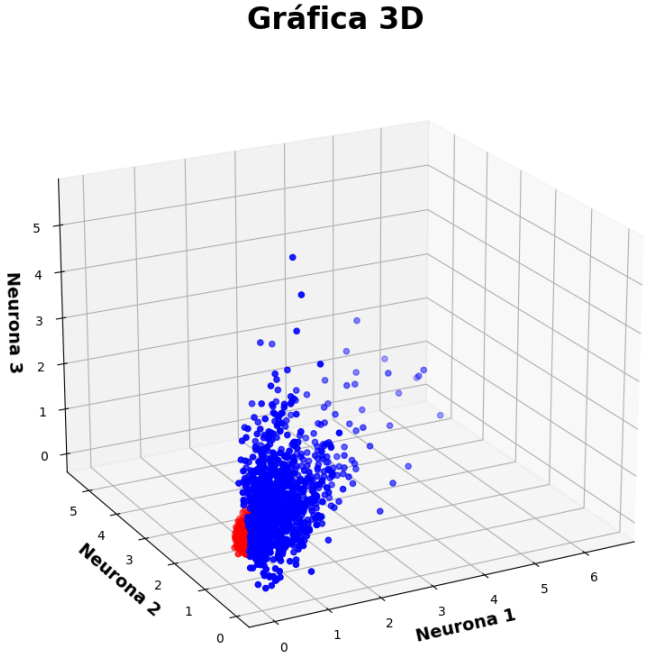

Figura 5.19: Gráfico tridimensional de los conjuntos discernidos por el modelo. Los vectores color **rojo** corresponden a muestras saludables y los **azules** a muestras con algún tipo de falla.

5.1.5. Construcción del clasificador *ELM*

La construcción del clasificador se llevará a cabo contemplando que la información de entrada consta un conjunto de vectores de tres elementos provenientes de la extracción de información interna del autoencoder, como los que fueron ilustrados gráficamente en las Figuras 5.15, 5.17 y 5.19. La arquitectura de aprendizaje supervisado *ELM* está constituida de una capa intermedia y capa de salida, como se ilustra en la Figura 5.20.

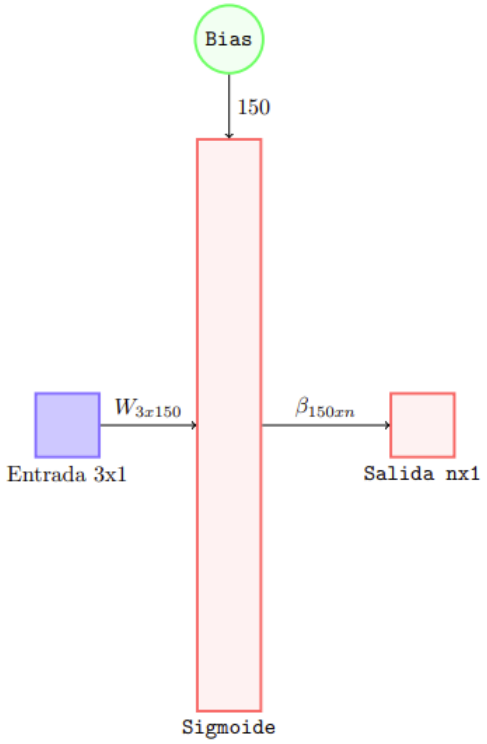

Figura 5.20: Esquema de una arquitectura *ELM*.

En la Tabla 5.7 se observan los parámetros de la arquitectura a construir.

Capa	Número de neuronas	Función de activación	Polarización
Entrada	3	Lineal	No
Capa oculta	150	*Sigmoide*	Si
Capa de salida 2	n	Lineal	No

Tabla 5.7: Tabla de tipos de capa.

5.1.6. Integración del encoder a la arquitectura *ELM*

Para poder crear una metodología integral de medición, se integró la sección encoder al clasificador con el propósito de hacer predicciones en "un solo paso".

El la Figura 5.21 se observa la metodología integral propuesta, la cual consta del encoder el cual se encarga de comprimir la información y construir una representación de la entrada a tres características; posteriormente, la arquitectura *ELM* toma la información de salida de las tres neuronas del encoder y predice su respectiva clase.

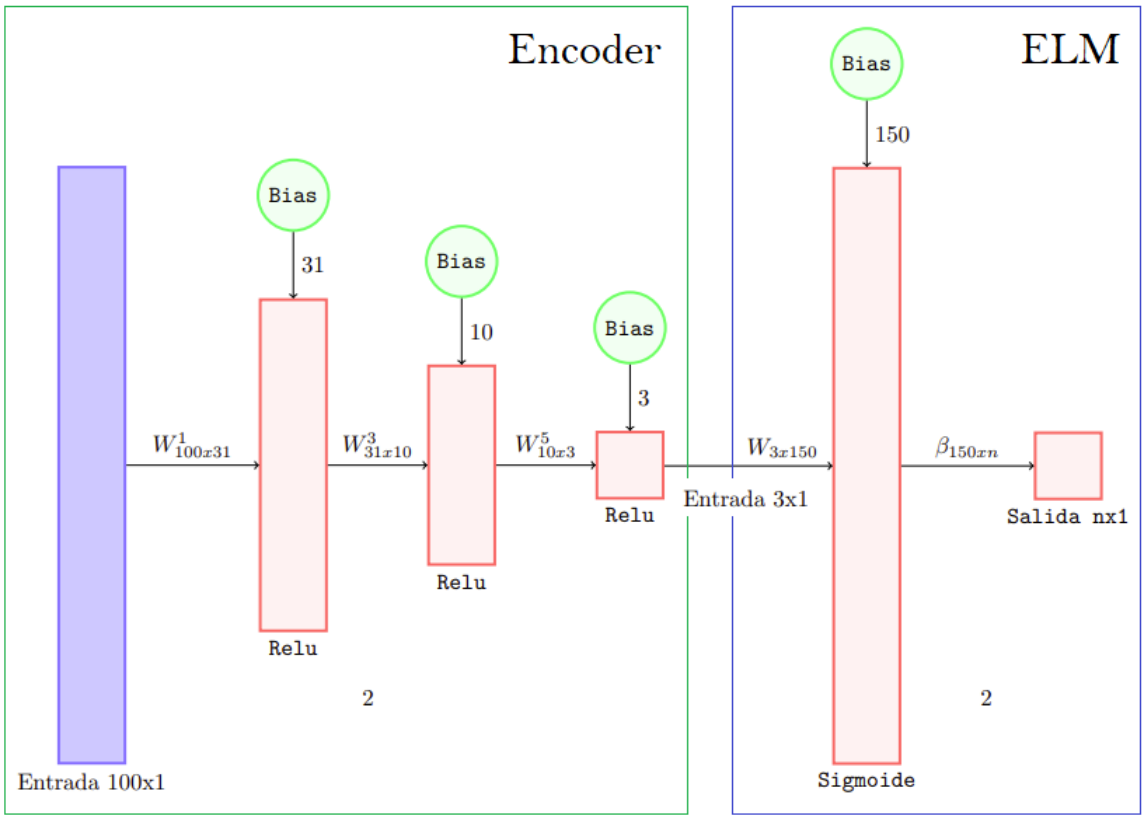

Figura 5.21: Arquitectura integral.

La arquitectura propuesta en la Figura 5.21, se considera que puede ser implementada en dispositivos *FPGA*. Por otra parte, se podría proponer su implementación en un *ASIC*, como una arquitectura neuromórfica, haciendo uso de las capacidades reportadas por los novedosos dispositivos activos de almacenaje de niveles analógicos de potencial, llamados *memristores*.

Evaluación del desempeño mediante matrices de confusión

Para evaluar el desempeño de la arquitecturas pertenecientes al aprendizaje supervisado, existe una herramienta llamada matriz de confusión, la cual está constituida por columnas que representan la predicción de la clase, y las filas corresponden a las clases reales.

En la Figura 5.22 se observa una matriz de confusión para 4 clases, en la cual las clases están ordenadas como **Normal** para las muestras que no presentan fallas y **FB**, **FCI**, **FCE** para las muestras con falla inducidas en el balín, canal interno y canal externo respectivamente, cuyos archivos fueron descritos en la Tabla 1.1 de la Sección 1.3.1. En el extremo derecho de la matriz de confusión se agregó una columna la cual calcula la exactitud de la predicción por clase. En una matriz de confusión con un buen desempeño, se observará una diagonal principal que tiene los valores más altos de cada fila, haciendo referencia a la correspondencia entre clase verdadera y predicha.

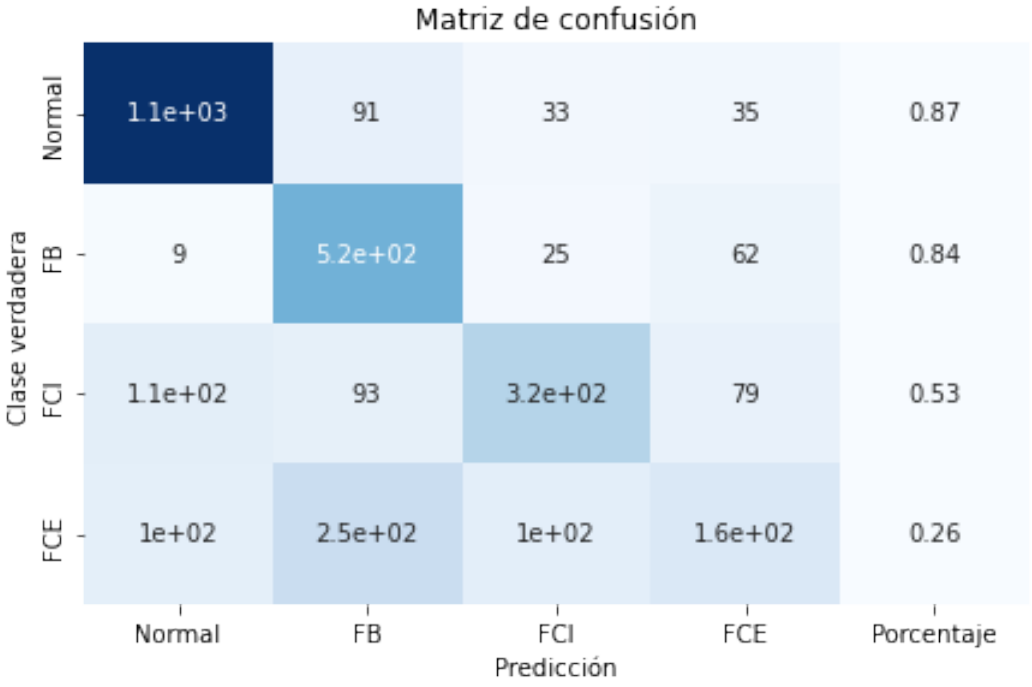

Figura 5.22: Matriz de confusión para cuatro clases.

En la Matriz de confusión de la Figura 5.22, se observa una exactitud para la clase **Normal** de 87 %, para la clase **FB** de 84 %, para la clase **FCI** de 53 y para la clase **FCE** de 26 %, resultando en una exactitud general de la arquitectura de 67.47 %. Los vectores de entrenamiento ingresados corresponden a los archivos normal y con fallas de diámetro de 0.007"[8], comprimidos a 3 características mediante el *autoencoder* descrito en la sección 5.1.4, con errores de pérdida y validación de `0.048` y `0.049`, respectivamente. Como el grado de incertidumbre en la predicción de la arquitectura para 4 clases es alto, se optará por trabajar a dos clases: con falla y sin falla; cabe mencionar que una de las características de la arquitectura *ELM* es su velocidad de respuesta, por lo que una predicción binaria es aceptable para los propósitos de este trabajo.

[8] Véase la Sección 1.3.1, Tabla 1.1.

En las Figuras 5.23,5.24 y 5.25 se observan las matrices de confusión generadas por la arquitectura *ELM* para datos saludables y con falla de 0.007", 0.014" y 0.021", respectivamente con los datos mostrados en la Tabla 5.8.

Archivos empleados	Pérdida en entrenamiento y validación de la compresión	Exactitud (%)
`Normal_0,B007_0, IR007_0,OR007@6_0`	`0.0486479178071022, 0.0492450259625911`	84
`Normal_0,B014_0, IR014_0,OR014@6_0`	`0.0352826975286006, 0.0349741317331790`	95
`Normal_0,B021_0, IR021_0,OR021@6_0`	`0.0572242401540279, 0.0581989511847496`	84

Tabla 5.8: Parámetros de entrada y resultados del proceso de clasificación.

Para la obtención de las Figuras 5.23,5.24 y 5.25 se realizó un proceso iterativo similar al visto en la Sección 5.1.4, en el cual se entrena el modelo 10 veces de manera sucesiva con la misma información de entrada. Los resultados obtenidos tienen la particularidad de proporcionar un grado de dispersión bajo entre cada iteración, el cual raramente sobrepasa el 1 % de diferencia.

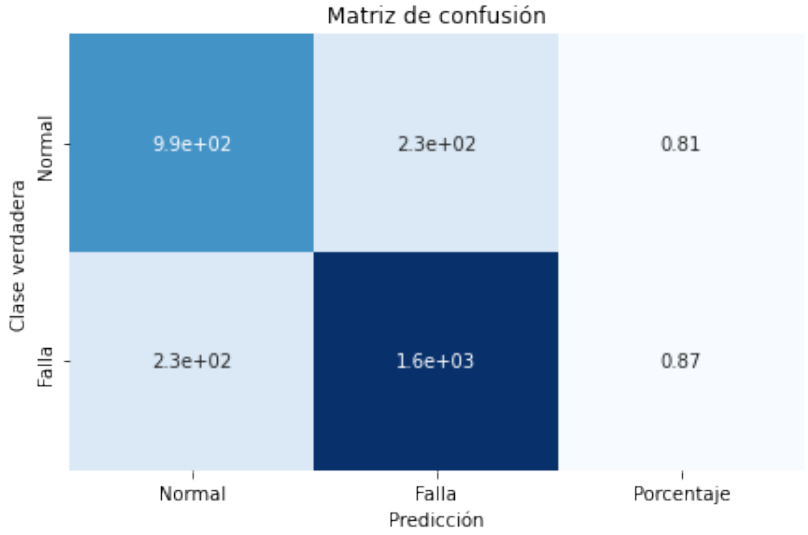

Figura 5.23: Matriz de confusión para dos clases, para muestras saludables y con falla de 0.007".

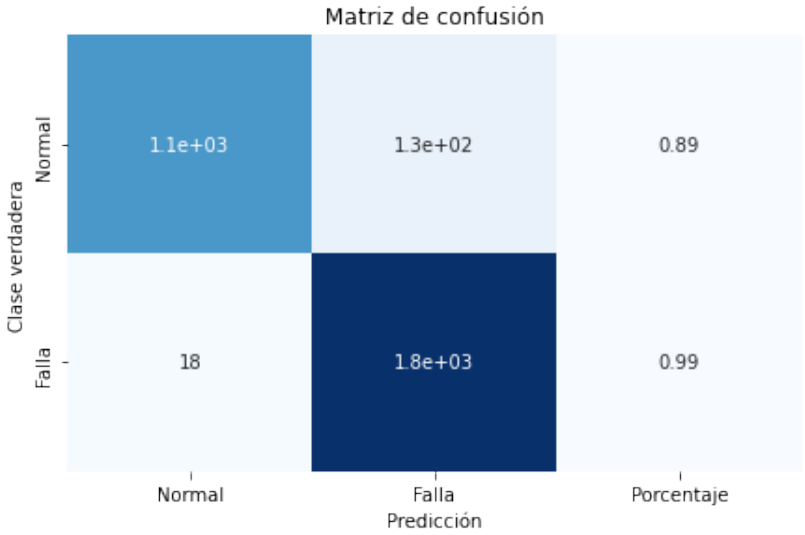

Figura 5.24: Matriz de confusión para dos clases, para muestras saludables y con falla de 0.014".

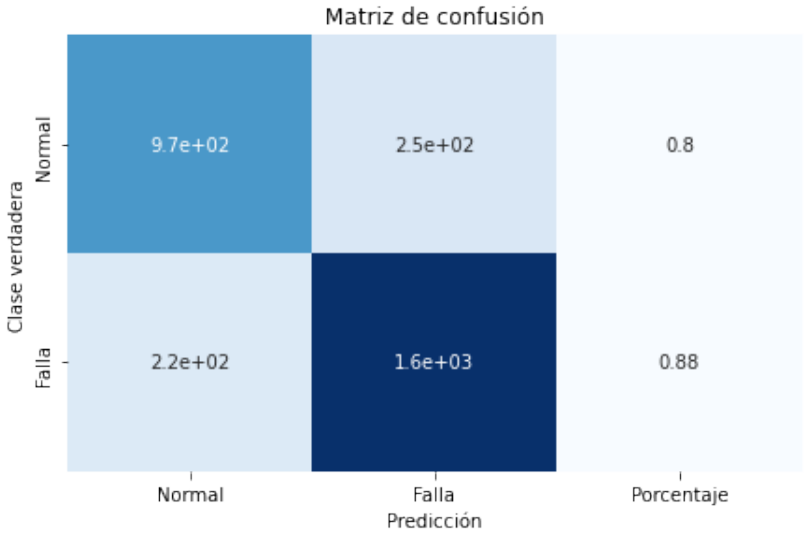

Figura 5.25: Matriz de confusión para dos clases, para muestras saludables y con falla de 0.021".

5.2. Conclusiones del capítulo

En este capítulo se construyó y entreno la arquitectura autoencoder haciendo uso de la base de datos de la Universidad *Case Western Reserve*, mediante una descripción en lenguaje *Python* empleando el entorno de desarrollo *Jupyter* y los paquetes de aprendizaje en máquina *Tensor Flow*. En la fase experimental, el modelo inicialmente propuesto lo constituyeron 3 capas, una de entrada, una oculta y una de salida; el desempeño obtenido en dicha arquitectura resulto ser poco factible, por lo que se procedió a construir un modelo multicapa, también conocido como profundo, agregando cuatro capas adicionales al modelo inicial, cuyas dimensiones fueron determinadas siguiendo una relación exponencial entre las dimensiones de las capas. En este modelo se obtuvieron mejores valores de reconstrucción, con la particularidad de presentar un grado de variación alto en los valores de perdida de los diferentes entrenamientos, motivo por el cual, se desarrolló una metodología de entrenamiento por etapas, conocida como transferencia de aprendizaje y ajuste fino, la cual trata de facilitar el trabajo de entrenamiento al algoritmo *Back Propagation*, obteniendo un grado más equitativo de aprendizaje en todas las capas del modelo, así como una menor variación de los resultados en el entrenamiento. De esta manera se concluye que en las arquitecturas autoencoder los modelos profundos presentan un mejor grado de aprendizaje en comparación con los de una sola capa, pero su metodología de entrenamiento es más elaborada.

En este capítulo se construyó y entreno el clasificador *ELM* con la representación reducida de la base de datos, obteniendo resultados satisfactorios para 2 clases, pero no siendo así para las 4 clases con las que está constituida dicha base de datos. Con lo cual se concluye que a arquitectura *ELM* es eficiente para datos con cierto grado de pre-procesamiento, y resulta poco adecuada para tareas de clasificación de más de dos clases.

En este capítulo se fusionaron la etapa del decoder y el clasificador *ELM*, creando una sola arquitectura integral de medición, la cual fue simulada y cuyos resultados de clasificación en la simulación de modelos numéricos, fueron aceptables, con lo cual se concluye que su implementación en dispositivos lógicos programables es viable.

CAPÍTULO 6

Conclusiones y trabajo a futuro

En base a los resultados obtenidos en este trabajo y al aprendizaje adquirido en su desarrollo, las conclusiones se pueden resumir en los siguientes cinco puntos:

• Se concluye que las redes neuronales artificiales pueden ser empleadas para la solución de problemas de ingeniería, como es el caso del problema abordado en este trabajo, referente a la detección de fallas en rodamientos, con un grado aceptable de exactitud en su diagnostico.

• Durante el proceso de experimentación se llego a la conclusión que las arquitectura autoencoder profundas son capaces de extraer mas información de los conjuntos de datos, en comparación con las arquitecturas autoencoder elementales, pues en ambos casos es posible reducir la dimensión, pero la calidad de la información comprimida es mayor en las arquitecturas profundas.

• Como contribución de este trabajo se concluye que las técnicas de *Machine Learning* son adecuadas para el tratamiento de información en grandes volúmenes, por lo que en la rama del *Big Data*, estas técnicas representan herramientas de extracción de información importantes.

• Se concluye que es posible implementar una red neuronal en sistemas electrónicos mediante el diseño de una neurona digital, la cual puede ser replicada con el fin de crear estructuras neuronales organizadas en dispositivos lógicos programables.

• Se concluye que este trabajo puede ser retomado en el futuro para enriquecer la idea fundamental, explorando arquitecturas más avanzadas, como es el caso de los autoencoders basados en maquinas restringidas de Boltzmann, la cual es una red *GAN*, cuya estructura es similar a la construida en este trabajo pero su capacidad de generalización es mayor.

CAPÍTULO 7

Perspectivas

Existen varias propuestas de trabajos a futuro, derivadas de esta tesis, las cuales se pueden organizar en los siguientes 3 puntos:

• En este trabajo se planteó la posibilidad de implementar el sistema neuronal en un dispositivo *FPGA*, lo cual conlleva muchas consideraciones que pueden ser abordadas en futuras investigaciones de posgrado.

• Se abre la posibilidad de realizar el entrenamiento de la arquitectura autoencoder, empleando métodos de optimización novedosos como lo son los algoritmos genéticos y de inteligencia de colonia, con el propósito de mejorar la exactitud de diagnóstico de la red, conservando la arquitectura optima en recursos.

•Los resultados de este trabajo, confirman la factibilidad de implementar la arquitectura construida en un posible *ASIC*, haciendo uso de herramientas mas avanzadas de diseño electrónico.

CAPÍTULO 8

Apéndice

8.1. Apéndice A: Descripciones en *Python*

8.1.1. Descripción del pre procesamiento

8.1.2. Descripción del autoencoder

8.1.3. Descripción del clasificador

8.2. Apéndice B: Descripción del modelo en *VHDL*

8.2.1. Descripción de la maquina de estados (entidad de nivel superior)

8.2.2. Descripción del proceso aritmético (modulo)

8.2.3. Descripción de la simulación funcional

8.3. Apéndice C: Descripción de la simulación de punto fijo y flotante en *Simulink*©

Los archivos de las descripciones mencionadas, se pueden encontrar el el siguiente enlace:
https://drive.google.com/drive/folders/1f7J8Q1bOAGRrHJ6snL8gPdoFDkmPzGs3?usp=sharing

Bibliografía

[1] E.K. Blum y A.V. Aho. *Computer Science: The Hardware, Software and Heart of It*. SpringerLink : Bücher. Springer New York, 2011. ISBN: 9781461411680. URL: https://books.google.com.mx/books?id=S7QU9RRLYIYC.

[2] A. Deshpande y M. Kumar. *Artificial Intelligence for Big Data: Complete guide to automating Big Data solutions using Artificial Intelligence techniques*. Packt Publishing, 2018. ISBN: 9781788476010. URL: https://books.google.com.mx/books?id=pF9dDwAAQBAJ.

[3] Emerson. *AMS 2140 Machinery Health Analyzer*. Technical report +1 865 675 2400. 835 Innovation Drive Knoxville, TN 37932 USA: Emerson Reliability Solutions, dic. de 2017.

[4] Forbes. *The future of AI is unsupervised*. Disponible en https://www.forbes.com/sites/insights-ibmai/2020/06/01/the-future-of-ai-is-unsupervised/.

[5] Quentin Fournier y Daniel Aloise. "Empirical Comparison between Autoencoders and Traditional Dimensionality Reduction Methods". En: *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE, jun. de 2019. DOI: 10.1109/aike.2019.00044. URL: https://doi.org/10.1109/aike.2019.00044.

[6] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] P. Grohs, M. Holler y A. Weinmann. *Handbook of Variational Methods for Nonlinear Geometric Data*. Springer International Publishing, 2020. ISBN: 9783030313517.

[8] D.G. Lowe. "Object recognition from local scale-invariant features". En: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150-1157 vol.2. DOI: 10.1109/ICCV.1999.790410.

[9] F. Preparata. *Theoretical Computer Sciences*. Springer Berlin, Heidelberg, 1975. DOI: https://doi.org/10.1007/978-3-642-11120-4.

[10] Universidad Case Western Reserve. *Bearing Data Center*. Disponible en https://engineering.case.edu/bearingdatacenter/12k-drive-end-bearing-fault-data.

[11] Sebastian Roldan, David Sanchez-Londono y Giacomo Barbieri. "Thermographic Indicators for the State Assessment of Rolling Bearings". En: *IFAC-PapersOnLine* 54.1 (2021). 17th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2021, págs. 1218-1223. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2021.08.208. URL: https://www.sciencedirect.com/science/article/pii/S2405896321009794.

[12] Abhishek Singh, Ashish Payal y Sourabh Bharti. "A walkthrough of the emerging IoT paradigm: Visualizing inside functionalities, key features, and open issues". En: *Journal of Network and Computer Applications* 143 (2019), págs. 111-151. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2019.06.013. URL: https://www.sciencedirect.com/science/article/pii/S1084804519302188.

[13] Wang SY, Yang DX y Hu HF. "Evaluation for Bearing Wear States Based on Online Oil Multi-Parameters Monitoring". En: *Sensors (Basel)*. PubMed Central, abr. de 2018. DOI: 10.3390/s18041111.

[14] TensorFlow. *Datasets*. Disponible en https://www.tensorflow.org/datasets/catalog/mnist.

[15] TensorFlow. *Guia inicial de TensorFlow 2.0 para principiantes*. Disponible en https://www.tensorflow.org/tutorials/quickstart/beginner?hl=es-419.

[16] TensorFlow. *Página de inicio de TensorFlow en español*. Disponible en https://www.tensorflow.org/?hl=es-419.

[17] Wikipedia. *Neuron*. Disponible en https://lfn.wikipedia.org/wiki/Neuron.

[18] Xin-She Yang. "3 - Optimization algorithms". En: *Introduction to Algorithms for Data Mining and Machine Learning*. Ed. por Xin-She Yang. Academic Press, 2019, págs. 45-65. ISBN: 978-0-12-817216-2. DOI: https://doi.org/10.1016/B978-0-12-817216-2.00010-7. URL: https://www.sciencedirect.com/science/article/pii/B9780128172162000107.

[19] D.S. Yeung et al. *Sensitivity Analysis for Neural Networks*. Natural Computing Series. Springer Berlin Heidelberg, 2009. ISBN: 9783642025327. URL: https://books.google.com.mx/books?id=NK5tuMHaO6sC.

[20] Aston Zhang et al. "Dive into Deep Learning". En: *arXiv preprint arXiv:2106.11342* (2021).

[21] Xian-Da Zhang. *A Matrix Algebra Approach to Artificial Intelligence*. Springer Singapore, 2020. DOI: 10.1007/978-981-15-2770-8. URL: https://doi.org/10.1007/978-981-15-2770-8.