Javier Gomez Monteiro

# Deep Learning aplicado al procesamiento de tomografías por microondas

2023

*Instituto de Ingeniería y Agronomía*

*Carrera: Ingeniería en Informática*

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Carrera de Ingeniería en Informática

PRÁCTICA PROFESIONAL SUPERVISADA
Informe final

# *Deep Learning aplicado al procesamiento de tomografías por microondas*

Estudiante: Javier Gómez Monteiro
javiergomezmonteiro@gmail.com

Tutor: Dr. Ing. Ramiro Miguel Irastorza
rirastorza@unaj.edu.ar

Tutor organizacional: Dr. Ing. Martín Morales
martin.morales@unaj.edu.ar

Florencio Varela, diciembre 2023

**Estudiante**
Javier Gomez Monteiro
javiergomezmonteiro@gmail.com

**Organización donde se realiza la Práctica Profesional Supervisada**
Universidad Nacional Arturo Jauretche
Av. Calchaquí N° 6200, Florencio Varela, (1888) Buenos Aires, Argentina
+54 11 4275 6100
Sector: Programa TICAPPS (Tecnologías de la Información y la Comunicación en Aplicaciones de Interés Social), Instituto de Ingeniería y Agronomía

**Tutor**
Dr. Ing. Ramiro Irastorza
rirastorza@unaj.edu.ar

**Tutor Organizacional**
Dr. Ing. Martín Morales
martin.morales@unaj.edu.ar

# Resumen

El objetivo de este trabajo fue el del desarrollo de un software en Python utilizando Inteligencia Artificial de Redes Neuronales Convolucionales (CNN por sus siglas en inglés) de la librería de TensorFlow capaz de realizar estimaciones y reconstrucciones de las propiedades dieléctricas y geométricas cilindros dispersores medidos utilizando la técnica de imágenes por microondas sin información de fase. Con este fin, para el entrenamiento de la red, se generaron imágenes mediante un software de simulación en Python implementando la librería Meep con distintos tipos de arreglos geométricos y dieléctricos en los que se utilizaron como referencia dos cilindros homogéneos (uno dentro de otro) que representan un modelo de juguete de un corte transversal de tobillo. Las imágenes son representaciones directas de la magnitud del campo eléctrico medido en las antenas receptoras y generado por estas estructuras. Al no contar con mediciones experimentales, las mismas fueron simuladas utilizando la técnica de FDTD (Finite-Difference Time-Domain por sus siglas en inglés), que resuelve numéricamente las ecuaciones de Maxwell.

A partir de esta premisa, se evaluaron diferentes topologías de redes neuronales variando los hiperparámetros de la red con el fin de encontrar la combinación óptima. Basados en estos resultados se logró conformar y compilar un modelo cuya tasa de predicción es aceptable en términos aproximados pero no del todo certera para aplicaciones donde se requieran resultados más precisos. Este comportamiento se puede deber a que, en esencia, el método de reconstrucción tomográfica sin información de fase pertenece a una categoría de problemas inversos mal planteados o mal condicionados donde no se conoce nada del objeto dispersor, se busca es reconstruir una imagen de las propiedades dieléctricas del dominio de investigación y sólo se tiene información en el dominio de medición (antenas receptoras).

# Agradecimientos

Agradezco a la Universidad Nacional Arturo Jauretche por darme la oportunidad de estudiar y convertirme en un profesional. A mi tutor organizacional Ramiro Irastroza y a Jesus Fajardo por acompañarme y guiarme en este proyecto.

Dedico este trabajo y mi esfuerzo a mi familia, pareja, amigos, compañeros y profesores que me acompañaron estos años y que me dieron la fuerza cuando más la necesitaba. Muchas gracias.

# Lista de Términos

**CNN**: Convolutional Neural Network.

**FDTD**: Finite-Difference Time-Domain.

**FEM**: Finite Element Moment.

**MOM**: Method of Moments.

**AI/IA**: Artificial Intelligence.

**Ez**: Campo eléctrico.

**TMO**: Tomografía por microondas.

**PLN**: Procesamiento de Lenguaje Natural.

# Índice General

# Capítulo 1

# Introducción

La presente Práctica Profesional Supervisada (PPS) se desarrolló en el marco del Proyecto de Investigación de la Universidad Nacional Arturo Jauretche UNAJ INVESTIGA 2020 (Código del Proyecto: 80020200100030UJ y Resolución Rectoral N° 183/21 de fecha 08/08/2021), cuyo título es "Algoritmos de Machine Learning para procesamiento de imágenes en aplicaciones biomédicas, agronómicas y ambientales", bajo la Dirección del Dr. Ing. Martín Morales y Co-Dirección de Dr. Ing. Ramiro Irastorza en el marco del Programa Tecnologías de la información y la comunicación (TICs) en aplicaciones de interés social (TICAPS).

Las técnicas de imágenes por microondas se han vuelto muy populares en esta última década como alternativa a los métodos tradicionales de imágenes, como por ejemplo, la resonancia magnética o la tomografía (utilizando rayos X). El método de tomografía por microondas consiste en iluminar el objeto que se desea detectar con frecuencias que van desde los 100 MHz a las decenas de GHz y a partir de un arreglo de antenas receptoras se analiza cómo la muestra afecta las señales y se reconstruye una imagen de la distribución interna de propiedades dieléctricas, (la permitividad y la conductividad) [1,2]. Si bien el método es relativamente conocido y estándar, una problemática actual yace en la reconstrucción de las propiedades geométricas y dieléctricas del objeto original a partir de mediciones de campo eléctrico sin información de fase. Una de las soluciones propuestas en este proyecto es utilizar y entrenar inteligencia artificial con las mediciones en las antenas receptoras en dispersores a partir de medios con las propiedades conocidas. En particular, en este trabajo nos centraremos en objetos cilíndricos (un cilindro dentro de otro) en dos dimensiones como una extensión del trabajo que se viene desarrollando en el grupo [3]. De esta manera, la red podría predecir y reconstruir valores geométricos, como el radio y el centro, y dieléctricos (la permitividad y la conductividad).

Otra de las tecnologías que hoy en día cada vez gana más popularidad son las redes neuronales, que pertenecen a un tipo específico de inteligencia artificial. En concreto, para este proyecto se usa un tipo de red neuronal convolucional (Convolutional Neural Network o CNN) enfocada al reconocimiento de patrones, clasificaciones, detección de objetos y segmentación semántica a través del procesamiento de imágenes. Este tipo de redes se inspiran en las neuronas biológicas que funcionan mediante la descomposición de la información en diferentes características, como bordes, texturas y formas para luego unirlas y comprender la imagen.

En el capítulo 2 será descrito el problema directo e inverso de las imágenes por microondas y la física involucrada junto con el funcionamiento del simulador responsable del cálculo numérico del campo eléctrico $E_z$ generado por la dispersión de dos cilindros (uno dentro de otro) aleatoriamente creados. Para tal fin, utilizamos la librería Meep que implementa el Método de Diferencias Finitas en Dominio de Tiempo (FDTD) para resolver las ecuaciones de Maxwell, donde el espacio es dividido en una grilla discreta y los campos electromagnéticos evolucionan utilizando pasos de tiempo discretos [4].

En el capítulo 3 será descrito el desarrollo de la red neuronal junto con todo su funcionamiento, desde la forma que consume los datos de entrada (imágenes y parámetros) hasta la normalización que se aplica a los mismos antes de alimentar la red. También se detallará el tipo de topología de la red, sus secciones y características, y por último la sección de salida de la red conformada por el modelo entrenado compilado una vez terminada la sesión de entrenamiento y validación.

El capítulo 4 estará dedicado a la presentación analitica de los resultados en dos secciones: por un lado las propiedades geométricas y por el otro las propiedades dieléctricas, exponiendo sobre estos, índices informativos como porcentajes de la tasa de acierto, desviaciones, errores relativos, histogramas y diagramas de dispersión a partir de un set de datos de prueba aleatorios.

Finalmente el capítulo 5 estará destinado a presentar una discusión general que contiene las conclusiones junto con los resultados más relevantes para un análisis objetivo de la efectividad en la resolución del problema planteado descrito por este trabajo.

# Capítulo 2

# El método de imágenes por microondas

La tomografía por microondas TMO es una técnica de imagen que utiliza ondas electromagnéticas con frecuencias de entre aproximadamente 100 MHz a 20 GHz para obtener información detallada sobre la estructura interna de un objeto o una muestra. A diferencia de las técnicas de tomografía médica convencionales que utilizan rayos X, la tomografía por microondas se basa en su capacidad de estas frecuencias para penetrar ciertos materiales y proporcionar información sobre las propiedades dieléctricas de los objetos.

En un sistema típico de tomografía por microondas, se utilizan antenas para generar y transmitir microondas a través del objeto o la muestra que se va a estudiar. Las microondas se propagan a través del material y experimentan cambios en su velocidad y amplitud según las propiedades dieléctricas del material.

Las antenas también se utilizan para recoger las señales de microondas que se transmiten a través del objeto y que son dispersadas por éste. La atenuación y la fase de estas señales proporcionan información sobre las características eléctricas del material en el camino de las microondas. Para obtener una imagen tridimensional, se realizan mediciones desde diferentes ángulos alrededor del objeto. Esto se logra moviendo las antenas o el objeto mismo, o mediante el uso de antenas distribuidas alrededor del objeto.

Las señales medidas se utilizan para reconstruir una imagen tridimensional de la distribución de propiedades dieléctricas dentro del objeto. Esto se hace mediante técnicas de procesamiento de imágenes y algoritmos de reconstrucción tomográfica. La información clave que se extrae a través de la tomografía por microondas está relacionada con las propiedades dieléctricas del material, como la permitividad y la pérdida dieléctrica. Estas propiedades pueden variar según la composición y la estructura interna del objeto, lo que permite obtener información sobre su contenido.

La tomografía por microondas se utiliza en diversas aplicaciones, como la inspección no destructiva de materiales, la caracterización de muestras biológicas, la detección de objetos

ocultos y la evaluación de la calidad de productos alimenticios, entre otros.[12] Aunque presenta desafíos, como la resolución y la complejidad de los algoritmos de reconstrucción, la tomografía por microondas ofrece ventajas en términos de capacidad para penetrar materiales y obtener información sobre propiedades dieléctricas.

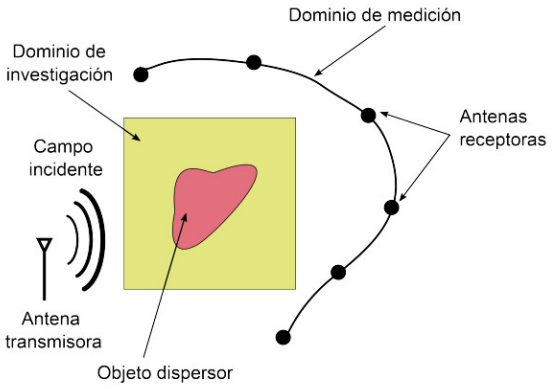



*Figura 1. Esquema de medición en tomografía por microondas.*

## 2.1 Algoritmos de TMO: El problema directo

Dentro de la mecánica presentada en la forma de realizar TMO existen diferentes algoritmos para resolver dos tipos de problemas: el directo y el inverso. Para el primer tipo de problema, las propiedades del objeto dispersor (forma, estructura interna, posición y propiedades dieléctricas) son conocidas y el método consiste en aplicar un campo eléctrico en las antenas transmisoras (campo incidente Ei) y se busca conocer el campo eléctrico (Ez) en todo el dominio de investigación y también en las antenas receptoras. Para el segundo tipo de problema, en el que no se conoce nada del objeto dispersor pero si se conoce en el dominio de medición (ver antenas receptoras en Fig. 1). En esta PPS nos centraremos en el problema inverso y se aplicará inteligencia artificial para reconstruir estos parámetros, y dado que no contamos con mediciones experimentales, nos basaremos en la salida del primero.

Para resolver el problema directo se puede emplear una forma analítica (en algunos casos) o numérica. Los enfoques numéricos más utilizados son: El Método de Elementos Finitos

(FEM), el Método de los Momentos (MoM) y el de Diferencias Finitas en Dominio de Tiempo (en sus siglas en inglés FDTD) en donde este último es nuestro caso de estudio y algoritmo de aplicación para la generación de imágenes por simulación.
El método FDTD es una técnica de discretización en el tiempo y el espacio de las ecuaciones de Maxwell, que describen el comportamiento de los campos electromagnéticos, para lograr esto se trabaja en un dominio de tiempo utilizando la celda de Yee, discretizando directamente las ecuaciones de Maxwell.

### 2.1.1 Teoría

Si suponemos que la onda se propaga en el modo transversal magnético en dos dimensiones (campo eléctrico en la dirección $z$), el problema de la dispersión (*scattering*) de una onda electromagnética para una densidad de corriente arbitraria $J\,(r)$ se representa por la ecuación de Helmholtz en su forma escalar:

$$\nabla^2 E_z\,(r) - k(r)^2 E_z\,(r) = j\omega\mu 0\, J(r) \qquad (1)$$

donde $E_z\,(r)$ es el campo eléctrico total para una posición r(vector que representa la posición en dos dimensiones) y μ0 es la permeabilidad magnética del vacío. El número de onda se puede calcular con $k = \omega\sqrt{\mu 0\, \varepsilon_r\,(r)\epsilon_0}$, donde ω, $\epsilon_0$ , $\varepsilon_r$ son la frecuencia de propagación de la onda, la permitividad del vacío y la permitividad relativa (que es función de la posición), respectivamente. Esta es una ecuación en derivadas parciales con solución compleja (tanto εr como $E_z$ pueden ser complejos) que se debe resolver en un dominio Ω con bordes ∂Ω. Para la discretización por FDTD se trabaja en dominio de tiempo utilizando la conocida celda de Yee, discretizando directamente las ecuaciones de Maxwell, aunque también se deben considerar condiciones de borde del tipo capa perfectamente adaptada (PML, por sus siglas en inglés).

De lo descrito hasta ahora se desprende que en el problema directo debemos conocer en todo el dominio Ω, la permitividad $\varepsilon_r(r)$ y la densidad de corriente $J(r)$ (en la representación de la Ec. 1) para poder calcular $E_z(r)$.

### 2.1.2 El método FDTD (Finite-Difference Time-Domain)

El método de Diferencias Finitas en el Dominio del Tiempo (FDTD, por sus siglas en inglés) se utiliza para resolver ecuaciones diferenciales en el tiempo y el espacio. En el contexto del electromagnetismo y las ondas, las ecuaciones de Maxwell son las que se discretizan y resuelven mediante el método FDTD. Las ecuaciones de Maxwell describen cómo se comportan los campos eléctricos y magnéticos en función del tiempo y el espacio.

Las ecuaciones de Maxwell en forma diferencial son las siguientes:

1. Ecuaciones de Gauss para el campo eléctrico (*E*):

   $\nabla \cdot E = \frac{\rho}{\varepsilon_0}$

2. Ecuaciones de Gauss para el campo magnético (*B*):

   $\nabla \cdot B = 0$

3. Ley de Faraday para la inducción electromagnética (*E* y *B*):

   $\nabla \times E = -\frac{\partial B}{\partial t}$

4. Ley de Ampère con corrección de Maxwell (*E* y *B*):

   $\nabla \times B = \mu_0 (J + \varepsilon_0 \frac{\partial E}{\partial t})$

En el método FDTD, estas ecuaciones se discretizan en una malla espacial y temporal. La discretización en el tiempo y el espacio permite resolver numéricamente las ecuaciones de Maxwell en pasos discretos. Las ecuaciones discretizadas para el método FDTD se derivan utilizando aproximaciones en diferencias finitas para las derivadas parciales. Las ecuaciones resultantes son específicas para la discretización en el tiempo y el espacio elegido y el resultado que vamos a obtener en nuestro caso particular será el campo eléctrico en la dirección z, debido a la condición cilíndrica de nuestro problema en dos dimensiones.

## 2.2 Inteligencia Artificial: El problema inverso

Si bien el método directo consiste en la resolución analítica del problema basado en la premisa del conocimiento de las propiedades del objeto dispersor, el problema inverso es justamente lo contrario, no se conoce nada acerca del objeto dispersor y lo que se busca es que a partir de los valores del dominio de medición (antenas receptoras) buscar la reconstrucción (basada en la imagen del campo eléctrico resultante de la medición) de las propiedades dieléctricas y geométricas del dominio de investigación.

Existen varias formas de resolver el problema inverso y se pueden clasificar en dos: determinísticos y estocásticos. Dentro de los primeros se encuentran varios métodos y conceptos utilizados en el campo de la inversión de fuentes para la reconstrucción de parámetros en problemas de propagación de ondas, especialmente en el contexto de ondas electromagnéticas, entre ellos se destacan: Born, Born Extendido, Backpropagation (Retropropagación), Contrast Source Inversion (Inversión de Fuente de Contraste) [1,2]. Estos conceptos y métodos son capaces de reconstruir información sobre la estructura interna de un medio a partir de mediciones en las antenas receptoras pero al ser un problema inverso no lineal y por lo tanto inestable en varios de estos métodos se suele utilizar la linealización del problema para resolverlos y suelen ser aproximaciones.

En contraste a los métodos clásicos de resolución de problema inverso, el objetivo de este trabajo es la de presentar un método de resolución inversa basado en el aprendizaje de una inteligencia artificial estructurado sobre redes neuronales convolucionales capaces de observar estas mediciones y poder discernir y predecir las propiedades originales del objeto dispersor.

## 2.3 Descripción general de la problemática

Cómo se describió previamente, la técnica para una TMO se realiza con un arreglo de antenas alrededor del objeto dispersor que se desea medir y se lo ilumina con una frecuencia dada. De esta manera, en un problema de dos dimensiones, la onda incidente se dispersa debido al objeto y modifica el campo total resultante ($E_z$). En este caso se creó un arreglo de 16 (dieciséis) antenas receptoras de forma circular en un radio de 7,5 cm que también funciona como antenas transmisoras. Nos ocuparemos de problemas simples, en particular dos cilindros uno dentro del otro (Fig. 2 A3).

El principal objetivo de este proyecto consiste en el desarrollo de dos tipos de aplicaciones de software: Por una lado una herramienta encargada de simular todo el arreglo descrito para la simulación tanto de las antenas como del objeto dispersor basados en el métodos FDTD el cual generará como salida imágenes de resolución 16x16 del campo eléctrico medido en cada una de las antenas receptoras, y por otro lado el desarrollo de una aplicación basada en redes neuronales artificiales capaz de interpretar la información de la primera, aprender los valores de los cilindros y luego predecir sus propiedades.

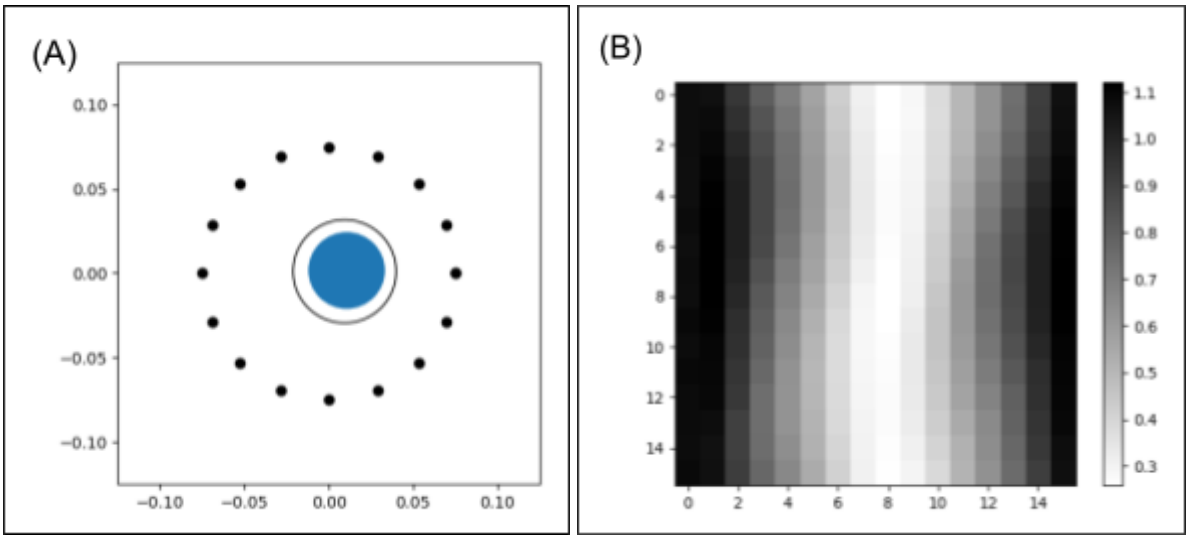



*Figura 2: Ejemplo de geometría simulada (A) y el módulo del campo eléctrico total simulado (B).*

## 2.4 Simulador de images de campo eléctrico

El objetivo principal del simulador es el de crear de forma aleatoria cilindros concéntricos de propiedades conocidas y en base al método FDTD generar imágenes del campo eléctrico resultante mediante dieciséis antenas receptoras.

Para resolver este escenario se desarrolló una aplicación en Python con las siguientes características:

- Se cuenta con una interfaz de usuario por consola en la que puede controlar la cantidad de simulaciones a generar y si desea también, generar datos a partir de un punto de partida previo (un "seed" o semilla para comenzar el contador pseudo-random de la librería *Meep*).
- La lógica principal el módulo *main.py* se basa en $n$ iteraciones las cuales generan en simultáneo implementando la librería de *concurrent.futures*, *threading* y *queue* cada una de las imágenes de salida con el objetivo de lograr una total utilización de los recursos del dispositivo.
- Para la simulación y creación de la imagen $E_z$ resultante se utiliza la librería de *Meep* contenida en el módulo *generate_data.py* y *fordward_fdtd2.py* los cuales reciben los parámetros generados de manera aleatoria y contenidas en el objeto *Cylinder* dentro del módulo *classes.py*.
- Finalmente se guarda toda esta información respectivamente en dos tipos de archivo: por un lado un los parámetros geométricos y dieléctricos de los cilindros en un archivo *.csv* llamado *output* y por otro lado un archivo que contiene una matriz de tipo 16x16 en formato de escala de gris de entre 0 a 1 en un archivo *npy* llamado *input*. El mismo archivo puede ejecutarse de manera paralela en instancias corriendo en simultáneo, debido a esto, cada una de las instancias puede generar $n$ resultados los cuales se podrán fusionar en uno solo siempre y cuando se respeten su orden.

### 2.4.1 Herramientas utilizadas

Lenguaje de programación:

- Python 3.10

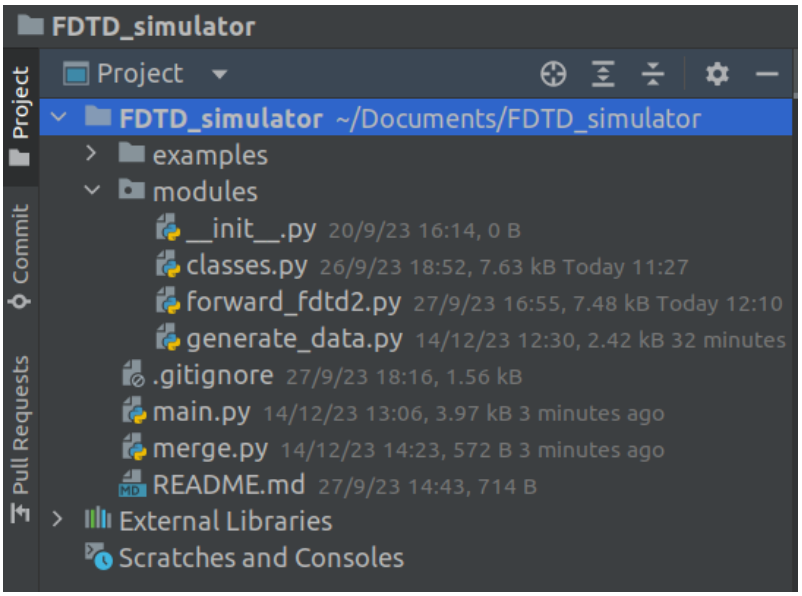
Librerías principales:

- PyMeep 1.26.0
- Matplotlib 3.7.1
- Numpy 1.24.3

Herramientas auxiliares:

- Miniconda
- PyCharm IDE
- Jupyter Notebook

### 2.4.2 Jerarquía del proyecto



### 2.4.3 Descripción general

- examples: paquete que contiene imágenes en formato .png de $E_z$ y ejemplos de los datos de entrada y salida generados (input.npy y output.csv).
- modules: paquete que contiene la mayoría de los módulos que contiene la lógica de simulación, dentro se encuentra:
    - classes.py: contiene todos los objetos de tipo Class entre los cuales se encuentran:
        - Antenna: clase que contiene los valores simulados de las antenas como la permitividad, conductividad, frecuencia, etc.
        - Cylinder: clase que contiene los valores de los cilindros concéntricos como los radios, las posiciones (x,y) las permitividades y conductividades.
    - fordward_fdtd2.py: contiene los valores del transmisor y acoplantes utilizados por el objeto Antenna como también la función RunMeep2 encargada de ejecutar la simulación FDTD.
        - ACOPLANTE_parameters: valores del material utilizado como acoplante.
        - TRANSMISOR_parameters: valores de la antena transmisora.
        - RunMeep2: función principal encargada de generar la geometría, la fuente, definir el medio y de ejecutar la simulación de Meep utilizando como argumentos los cilindros, el transmisor, el acoplante, el índice Tx de la antena emisora y los valores x,y de la celda de Yee.
        - Simulate: función de simulación de prueba con parámetros fijos.
    - generate_data.py: módulo que integra classes.py and fordward_fdtd2.py para ejecutar una simulación mediante la función generate_models que recibe como parámetro opcional un cola (queue) de tipo Queue para el almacenamiento en forma de lista (input, output) de los resultados de la simulación dado los parámetros de los cilindros.

- main.py: archivo principal para la ejecución de la simulación que cuenta con una interfaz de usuario por consola.
- merge.py: archivo auxiliar para combinar los datos de salida en caso de que se hayan ejecutado múltiples instancias de main.py en paralelo.

### 2.4.4 Funcionamiento

A continuación se dará un descripción general del funcionamiento del código correspondiente a la función RunMeep2, generate_models y el módulo main.py para un mejor entendimiento del simulador y cómo es que la información se genera, se simula y finalmente se guarda.

La Función RunMeep2 es la responsable de realizar la simulación a partir de una fuente y un objeto dispersor que dará como resultado dos matrices: el campo eléctrico y la permitividad eléctrica, eso se repetirá para cada una de las antenas en las que en cada iteración una trabajará como emisora y las demás quince trabajaran como receptoras hasta completar el ciclo completo y generar la imagen de 16x16.

A continuación se definen las constantes:

```
#
# - Constantes
#

pi = S.pi
eps0 = S.epsilon_0
c = S.c
mu0 = S.mu_0

a = 0.005  # Meep unit
```

Se define la función RunMeep2 con los parámetros correspondientes

```

#
# - Definición de funciones
#


#
# - Función numérica con FDTD utilizando software meep
#
def RunMeep2(cilindro1, cilindro2, acoplante, trans, Tx, caja, RES=5, calibration=False, unit=None):
```

Se inicializan las variables para la simulación.

```python
    res = RES  # pixels/a
    dpml = 1

    sx = caja[0] / a
    sy = caja[1] / a

    # print('sxa: ', sx, 'sxa: ', sy)

    # rhoS = tran1.5*c/trans.f

    fcen = trans.f * (a / c)  # pulse center frequency
    sigmaBackgroundMeep = acoplante.sigma * a / (c * acoplante.epsr * eps0)
    sigmaCylinderMeep = cilindro1.sigma * a / (c * cilindro1.epsr * eps0)
    sigmaCylinderMeep2 = cilindro2.sigma * a / (c * cilindro2.epsr * eps0)

    materialBackground = mp.Medium(epsilon=acoplante.epsr,
                                   D_conductivity=sigmaBackgroundMeep)  # Background dielectric properties at operation frequency
    materialCilindro = mp.Medium(epsilon=cilindro1.epsr,
                                 D_conductivity=sigmaCylinderMeep)  # Cylinder dielectric properties at operation frequency
    materialCilindro2 = mp.Medium(epsilon=cilindro2.epsr,
                                  D_conductivity=sigmaCylinderMeep2)  # Cylinder dielectric properties at operation frequency

    default_material = materialBackground

    # Simulation box and elements
    cell = mp.Vector3(sx, sy, z: 0)
    pml_layers = [mp.PML(dpml)]
```

Se destaca como importante la definición de la frecuencia central fcen, la definición del material de fondo o acoplante materialBackground y la definición del material de los cilindros materialCilindro1 y materialCilindro2 por medio de la clase Medium la cual representa la geometría de los objetos y el material por el cual están conformados.

Utilizando los materiales anteriores se instancia e inicia la simulación:

```python
    if calibration:  # el cilindro1 del centro es Background
        geometry = [mp.Cylinder(material=materialBackground, radius=cilindro1.radio / a, height=mp.inf,
                                center=mp.Vector3(cilindro1.xc / a, cilindro1.yc / a, z: 0))]
    else:  # el cilindro1 del centro es la muestra
        geometry = [mp.Cylinder(material=materialCilindro, radius=cilindro1.radio / a, height=mp.inf,
                                center=mp.Vector3(cilindro1.xc / a, cilindro1.yc / a, z: 0)),
                    mp.Cylinder(material=materialCilindro2, radius=cilindro2.radio / a, height=mp.inf,
                                center=mp.Vector3(cilindro2.xc / a, cilindro2.yc / a, z: 0))]

    xt = (trans.rhoS) * np.cos(Tx * 2 * pi / trans.S)  # Coordenada x antena transmisora
    yt = (trans.rhoS) * np.sin(Tx * 2 * pi / trans.S)  # Coordenada y antena transmisora

    sources = [mp.Source(mp.ContinuousSource(frequency=fcen), component=mp.Ez, center=mp.Vector3(xt / a, yt / a, z: 0.0),
                         amplitude=trans.amp, size=mp.Vector3( x: 0.0, y: 0.0, mp.inf))]

    sim = mp.Simulation(cell_size=cell, sources=sources, resolution=res, default_material=default_material,
                        eps_averaging=False, geometry=geometry, boundary_layers=pml_layers, force_complex_fields=True)

    nt = 600

    sim.run(until=nt)

    eps_data = sim.get_array(center=mp.Vector3(), size=cell, component=mp.Dielectric)
    ez_data = sim.get_array(center=mp.Vector3(), size=cell, component=mp.Ez)

    return ez_data, eps_data
```

La variable calibration es utilizada para saber si en esta simulación se está midiendo el objeto dispersor (False) o sólo el medio de acoplamiento (True). En general, en la técnica de TMO se utilizan mediciones de calibración solo con el medio de acoplamiento.

Aquí se definen dos clases más, por un lado Cylinder la cual define la geometría que necesitamos y Source que es la fuente donde se definirá la señal que iluminará el objeto dispersor, se utiliza la clase ContinuousSource. Por último se da instancia a la clase Simulation que va a recibir todo lo necesario para la simulación del campo eléctrico: el material de fondo, los cilindros, las fuentes y la caja de simulación. El método run recibe como parámetro el valor de unidad meep que determinará el tiempo en el cual correrá dicha simulación, cuanto mayor el valor más precisa la simulación pero tardará más en ser ejecutada.

Por último se recolectan los valores resultantes del campo eléctrico resultante ez_data y el mapa de la permitividad relativa compleja eps_data.

Luego en la función generate_models del módulo generate_data se generarán cada una de las imágenes del campo eléctrico dada la cantidad de iteraciones deseadas, basándose en la función RunMeep2 descrita anteriormente.

A continuación de define la función:

```
9   def generate_models(queue=None):
10      """
11      Función principal encargada de generar todos los datos de la simulación, estos son:
12      - Los valores de entrada Queue para almacenar los valores temporalmente.
13      - Los valores de salida que representan los valores geométricos y dieléctricos de los cilindors: radios, centros(x,y)
14      permitividad y conductividad
15
16      :return:
17          List()
18          - EzTr: NumpyArray() Matriz 16x16 que representa la imagen del campo electrico
19          - cil_data: Dict() Diccionario que contiene la clave valor de cada elemento descrito anteriormente.
20      """
```

generate_models recibe un solo argumento como parámetro llamado queue que se utilizará en el caso de que se ejecuten múltiples instancias de la misma función para poder ir guardando los resultados de cada uno de los threads.

Luego se definen los valores de los actores en la simulación, instanciando clases de tipo Cylinder y Antenna y de manera aleatoria asignando cada uno de sus atributos.

```
    # Graficos
    sx = 0.25
    sy = 0.25
    box = [sx, sy]

    antenna = Antenna()

    r = np.random.uniform( low: 0.01,  high: 0.040)
    anglescat = np.random.uniform( low: 0.0, 2 * pi)
    dext = np.random.uniform( low: 0.0, antenna.TRANSMISOR_parameters.rhoS - 1e-2 - r)
    Xc = dext * np.cos(anglescat)
    Yc = dext * np.sin(anglescat)

    rin = np.random.uniform( low: 0.005, r * 0.95)
    anglescatin = np.random.uniform( low: 0.0, 2 * pi)
    d = np.random.uniform( low: 0.0, r - rin)
    Xcin = Xc + d * np.cos(anglescatin)
    Ycin = Yc + d * np.sin(anglescatin)
    cilindros = Cylinder(antenna, r, Xc, Yc, rin, Xcin, Ycin)

    resolucion = 5
    n = resolucion * sx / a
    Tx = np.arange(16)
    Tr = np.arange(16)
    EzTr = np.zeros((16, 16))
```

Finalmente se realizan dos iteraciones sobre Tx y Tr (arreglos de 16 valores) los cuales representan a las antenas receptoras y emisoras. En la primera iteración tx actúa como la antena emisora y tr actúan como receptoras, junto con los resultados del campo eléctrico incidente se calcula el campo eléctrico resultante. Por último la función guarda en una lista de forma ordenada los resultados con sus respectivos valores de los cilindros utilizados para dicha imagen.

```
    for tx in Tx:
        Ezfdtd, eps_data = RunMeep2(cilindros[0], cilindros[1], antenna.ACOPLANTE_parameters,
                                    antenna.TRANSMISOR_parameters, tx, box, calibration=False, unit=0.005)
        Ezfdtdinc, eps_data_no = RunMeep2(cilindros[0], cilindros[1], antenna.ACOPLANTE_parameters,
                                          antenna.TRANSMISOR_parameters, tx, box, calibration=True, unit=0.005)
        for tr in Tr:
            xSint = int(resolucion * ((0.15 / 2) * np.cos(tr * 2 * pi / 16.)) / a) + int(
                n / 2)  # Coordenada x antena receptora
            ySint = int(resolucion * ((0.15 / 2) * np.sin(tr * 2 * pi / 16.)) / a) + int(n / 2)
            EzTr[tx, tr] = abs(Ezfdtd)[xSint, ySint] / abs(Ezfdtdinc)[xSint, ySint]

        EzTr[tx, :] = np.roll(EzTr[tx, :], -tx)

    cil_data = cilindros.as_dict()
    if queue:
        queue.put((EzTr, cil_data))
    return EzTr, cil_data
```

Para terminar, el módulo main.py o el módulo principal se encargará de toda la sección dedicada a la interfaz de usuario, por un lado un thread que hará seguimiento del progreso para que el usuario tenga en cuenta un estimado de finalización.

progress utilizará un while loop e irá chequeando el estado de cada uno de los threads de la lista para saber su estado y con este dato estimará cuando finalizará la simulación.

```
27  def progress(thread_list, iterations):
28      print("Comenzando simulación...")
29      print("Estimando tiempo de espera...")
30      single = 54
31      total_estimated_time = single * iterations
32      print("Tiempo estimado de: {} minutos".format(round(total_estimated_time / 60, 2)))
33      start = time.time()
34      while True:
35          now = time.time()
36          percentage = (int(now - start) * 100) / total_estimated_time
37          if percentage < 100:
38              print("Porcentaje completado: {}          \r".format(round(percentage, 2)))
39          if "RUNNING" not in [t._state for t in thread_list] and percentage >= 50:
40              finished = int(time.time() - start)
41              print("Simulación terminada con {} segundos".format(finished))
42              break
43          else:
44              if percentage >= 100:
45                  print("Tiempo estimado completo, terminando ultimas simulaciones...")
46          time.sleep(5)
```

Por otro lado se instancia la cola queue y se crea una lista de threads. El script luego pregunta la cantidad de iteraciones a realizar y se le dá la opción al usuario de poder guardar las imágenes en formato .png

```
49  if __name__ == '__main__':
50
51      threads = []
52      # results = []
53      result_queue = queue.Queue()
54
55      print("Ingrese la cantidad de simulaciones a realizar: ")
56      simulations = int(input())
57      print("¿Desea guardar las imagenes (formato .png) (s/n)?:")
58      save_images = str(input())
59
```

Luego con un ThreadPoolExecutor se lanzan cada uno de los thread que tiene como objetivo la función genrate_models en un total de n veces (simulations) junto con el thread de control que utiliza la función de progress descrita anteriormente.

```python
    #     results.append(result_queue.get())

    with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
        # Submit the function calls with different parameters
        futures = [executor.submit(generate_models, result_queue) for _ in range(simulations)]  # sim times
        control_thread = threading.Thread(target=progress, args=(futures, simulations,))
        control_thread.start()

    # Retrieve the results when they are ready
    results = [future.result() for future in futures]

```

Por último se espera a la finalización de todos los threads con el método result(). Una vez obtenidos los resultados el script realiza el proceso de almacenado en disco de los resultados, por un lado la creación del archivo .csv para los parámetros geométricos y dieléctricos de los cilindros y por otro lado se guarda en un archivo numpy .npy las matrices que representan al campo eléctrico resultante para cada uno de los valores del .csv. Por último, si el usuario eligió guardar dichas imágenes en formato .png se comprueba la variable save_images y se itera por cada uno de los resultados de .npy para convertir los valores de las matrices en imágenes de resolución 16x16 utilizando la librería de matplotlib pyplot.

```python
    # Retrieve the results when they are ready
    results = [future.result() for future in futures]

    print("Guardando resultados...")
    file_directory = base_dir + '/output.csv'
    csvfile = open(file_directory, 'w', newline='')
    fieldnames = ['radius', 'xc', 'yc', 'iradius', 'ixc', 'iyc', 'epsilon', 'sigma', 'iepsilon', 'isigma']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    cyl_params = [d[1] for d in results]
    writer.writeheader()
    writer.writerows(cyl_params)
    csvfile.close()
    data_input = [d[0] for d in results]
    np.save(base_dir + '/input.npy', np.array(data_input, dtype=object), allow_pickle=True)
    print("input y output han sido guardados")

    if save_images.lower() == 's':
        print("Guardando imagenes..")
        input_data = np.load(base_dir + '/input.npy', allow_pickle=True)
        input_data = np.asarray(input_data).astype('float32')
        # print(results)
        for idx, idata in enumerate(input_data):
            figure, axes = plt.subplots()
            plt.xlim( *args: -0.25 / 2, 0.25 / 2)
            plt.ylim( *args: -0.25 / 2, 0.25 / 2)
            plt.figure()
            plt.imshow(idata, cmap='binary')
            plt.colorbar()
            plt.savefig('EzTr {0}.png'.format(idx))
            plt.close()
        print("Imagenes guardadas")
```

```
javier@javier-desktop  ~/Documents/FDTD_simulator  main ● ?  python main.py
Ingrese la cantidad de simulaciones a realizar:
10
¿Desea guardar las imagenes (formato .png) (s/n)?:
n
Comenzando simulación...
Estimando tiempo de espera...
Tiempo estimado de: 9.0 minutos
Porcentaje completado: 0.0
Porcentaje completado: 0.93
Porcentaje completado: 1.85
```

*Figura 3: Imagen de la salida por consola del script.*

### 2.4.5 Salida

Como ya se ha descrito en la parte de funcionamiento, el script dará como resultado dos archivos, necesarios para el entrenamiento de la red, por un lado output.csv representará las etiquetas o valores de los cilindros y por otro lado input.npy representará las imágenes que la red deberá utilizar para poder entrenarse.

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | radius | xc | yc | iradius | ixc | iyc | epsilon | sigma | iepsilon | isigma |
| 2 | 0.019364695562354 | 0.0220256046114928 | -0.0219060292348494 | 0.0071151192131165 | 0.0139115212140631 | -0.0218788018863505 | 68.901845377301 | 1.07858235610174 | 34.1969357212723 | 1.08646345847095 |
| 3 | 0.028420312119917 | -0.00027018016641292 | -0.0203877115912657 | 0.0090905193424751 | 0.00076312946436659 | -0.0207914946363687 | 17.1207121524008 | 1.46744642726358 | 75.665007073266 | 0.459112570544113 |
| 4 | 0.039648226629407 | 0.00896203235314495 | -2.299676008299E-05 | 0.0370968221336097 | 0.00760859848020141 | -0.000582549662585 78 | 61.3365317739067 | 0.960317789219338 | 74.8227187095145 | 1.35768925642878 |
| 5 | 0.01576256985824 | 0.00901646509577909 | 0.0197199260231437 | 0.0077506188207439 | 0.00807786592556523 | 0.0148444843752678 | 68.4819442387155 | 1.28369480119529 | 48.9136859159136 | 0.618307033694786 |
| 6 | 0.01102046102805 | 0.0348723434844856 | -0.0199986024084906 | 0.00878630715212 2 | 0.0352099588123814 | -0.0194860427607018 | 33.9300605975371 | 1.17935008685856 | 41.0560019695517 | 0.650010970716233 |
| 7 | 0.032474615872205 | -0.0016613583096398 4 | -0.00071312005809685 | 0.0225921569881379 | -0.00023653787603469 | -3.66276557104789E-05 | 78.2317835135107 | 0.558037190966541 | 54.593192697369 | 1.56486795704663 |
| 8 | 0.011049592268827 | 0.0351532553320848 | -0.0111570619709362 | 0.0066292575587493 | 0.0352204529741951 | -0.0110009031958254 | 40.4704733123366 | 1.25356245038988 | 40.2594448939563 | 1.45497687476584 |
| 9 | 0.03289980887107 | -0.0092832473741554 | 0.0141318289658309 | 0.0308978797364685 | -0.00862484958860174 | 0.0138247028115354 | 43.7294864334422 | 0.577074340944178 | 10.3879583026038 | 0.55771034162559 |
| 10 | 0.037604825659207 | 0.0108906918826164 | -0.020783633941403 | 0.0127095474538285 | 0.0159040452012904 | -0.00682230176187468 | 11.1246760047027 | 0.468942736899641 | 25.5482629718632 | 0.814450483135098 |
| 11 | 0.015518500333016 | 0.0193887748064498 | 0.0110032641197575 | 0.0134998180229025 | 0.0192742884795001 | 0.012193758440779 | 73.9191352984608 | 1.4585493988208 | 21.3606999096664 | 0.905644073205032 |
| 12 | | | | | | | | | | |

*Figura 4: Archivo resultado de output.csv. Ejemplo de los valores de las etiquetas para los valores geométricos y dieléctricos.*

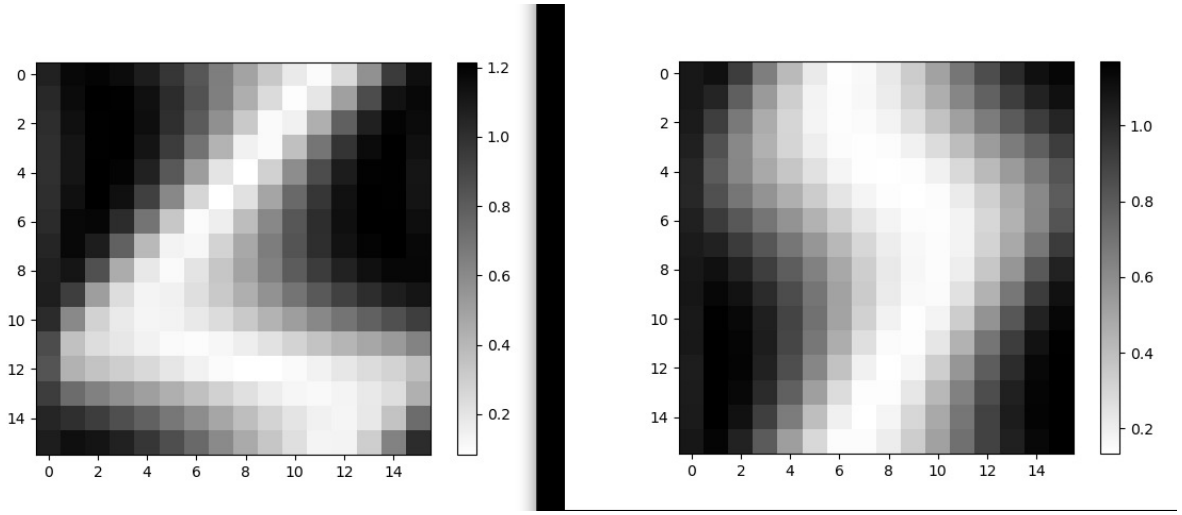



*Figura 5: Archivo input.npy. Ejemplos de imágenes generadas , se utilizó escala de grises para representar el valor de cada píxel entre cero y uno dado, que es el resultado entre la relación del campo eléctrico incidente y el campo eléctrico resultante, ambos calculados con FDTD.*

# Capítulo 3

# Redes Neuronales

En esta sección explicaremos el concepto fundamental de una red neuronal dentro del concepto de inteligencia artificial, que hoy en día pertenece a una tecnología de vanguardia.

Las redes neuronales son un componente fundamental de la inteligencia artificial (IA) y del aprendizaje profundo (deep learning en inglés). Se inspiran en el funcionamiento del cerebro humano y están diseñadas para aprender patrones y realizar tareas específicas. Consisten en capas de nodos o "neuronas" interconectadas, donde cada conexión tiene un peso que se ajusta durante el entrenamiento.

Las aplicaciones de las redes neuronales son diversas y abarcan una amplia gama de campos. Algunas de sus aplicaciones más comunes incluyen por ejemplo: el reconocimiento de patrones ya que son eficaces en la identificación de los mismos en grandes conjuntos de datos. Esto se aplica a la clasificación de imágenes, reconocimiento de voz, y otras tareas de reconocimiento de patrones; Procesamiento de Lenguaje Natural (PLN) que se utilizan para tareas relacionadas con el lenguaje, como la traducción automática, generación de texto, análisis de sentimientos y chatbots; la visión por computadora para la identificación y clasificación de objetos en imágenes y videos, así como en tareas de seguimiento y detección de objetos; en juegos y estrategia ya que han demostrado habilidades en juegos estratégicos, como Go y ajedrez, a través de algoritmos como AlphaGo y AlphaZero; en la conducción autónoma y en el desarrollo de sistemas para vehículos autónomos, utilizando redes neuronales para el reconocimiento de señales, detección de obstáculos y toma de decisiones; en la medicina y diagnóstico para la interpretación de imágenes médica, el cuál abarca nuestro caso de estudio, en el diagnóstico de enfermedades y predicción de resultados basados en datos clínicos y finalmente en las finanzas para poder realizar predicciones de tendencias del mercado, detección de fraudes y gestión de riesgos [9,10].

El concepto de redes neuronales tiene sus raíces en los modelos matemáticos de las neuronas y en la idea de simular el proceso de aprendizaje del cerebro. A lo largo del tiempo, ha habido varios hitos importantes. *McCulloch y Pitts (1943)*[7] propusieron el primer modelo formal de una neurona artificial, inspirada en la estructura y función de las neuronas biológicas. El psicólogo y científico de la computación Frank Rosenblatt en 1957 introdujo el perceptrón, un tipo básico de red neuronal de una sola capa [8] (ver Fig. 6). En 1969 *se* demostró que los perceptrones de una sola capa tenían limitaciones para resolver problemas más complejos, como la función XOR, lo que llevó al declive del interés en las redes neuronales durante algunos años. En los años 2000 los avances en algoritmos de entrenamiento, aumento en la cantidad de datos y poder de computación contribuyeron al resurgimiento de las redes neuronales, especialmente con el desarrollo de arquitecturas de aprendizaje profundo, como las redes neuronales profundas (deep neural networks) y las redes neuronales convolucionales (CNN) que demostraron capacidades impresionantes en diversas tareas [9].

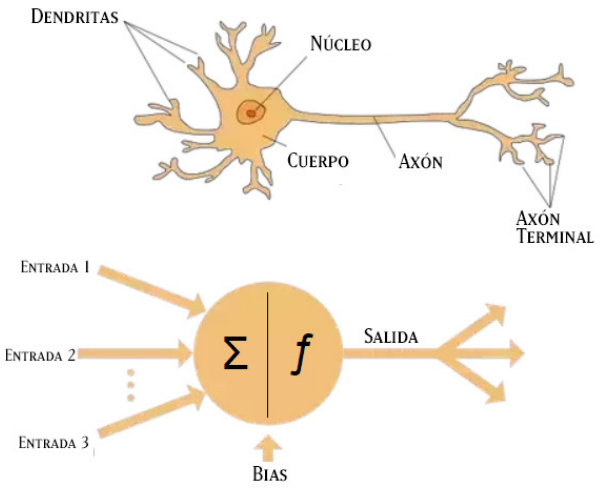



*Figura 6. Comparación entre una red neuronal biológica y una artificial* [13].

## 3.1 Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales (CNN), a diferencia de las redes neuronales clásicas, son un tipo específico de arquitectura de redes neuronales diseñadas para procesar y analizar datos de malla, como imágenes. Están especialmente diseñadas para capturar patrones

espaciales y jerarquías de características en datos multidimensionales. A continuación, se explica su propósito y se destacan algunas diferencias clave con respecto a las redes neuronales clásicas.

Las CNN fueron diseñadas para abordar eficientemente problemas de visión por computadora, donde la información está dispuesta en una estructura de cuadrícula o malla (por ejemplo, píxeles en una imagen). Su capacidad para reconocer patrones espaciales y jerarquías de características las hace ideales para tareas como la clasificación de imágenes, detección de objetos, segmentación semántica y más [11].

### 3.1.1 Características Principales

Las CNN logra un trabajo de reconocimiento preciso de imágenes efectivo a través de varias capas que son fundamentales y se dividen dependiendo su responsabilidad:

- *Capa de entrada*: usualmente responsable de recibir los datos (la imágenes) que utilizará la red para su entrenamiento, para esto debe estar bien definido el tipo de datos a utilizar (la resolución de la imagen) y la cantidad de canales (RGB, Greyscale, etc.).

- *Capas convolucionales*: estas capas son fundamentales para la red ya que a través de los denominados filtros o kernels se realizan operaciones de convolución o filtrado a la información de entrada, extrayendo así características locales y patrones necesarios para el entrenamiento. Cada filtro o kennel tiene una medida (usualmente menor que la imagen original) en particular que se va desplazando como una ventana sobre toda la imagen. Es necesario definir la cantidad y la medida de los kernels para obtener resultados óptimos, cada uno trabaja de manera tal que al desplazarse por la entrada produce un mapa de características. A este proceso se lo llama convolución.

- *Función de activación*: el propósito de esta función está dedicado a crear una relación no lineal entre la entrada y la salida y así ayudar a la red a no crear linealidades o aprendizaje lineal. Este objetivo se logra a través de la asignación de la suma de los pesos (weights) de la entrada que a su vez se envía a la siguiente capa mediante la salida. Usualmente la función de activación se aplica a cada capa de la red neuronal, los algoritmos más utilizados son: Rectified linear unit (ReLU), Sigmoid y Hyperbolic tangent (tanh). ReLu, por sobre las demás, es el más popular al ser uno de los más eficientes y por generar representaciones dispersas de la información de entrada. La función de activación se podría decir que actúa como el "cerebro" de la CNN donde la información de entrada se transforma a representaciones significativas de la misma para lograr el aprendizaje.

- *Capas de pooling*: Las capas de pooling se utilizan para reducir la dimensión espacial de las representaciones intermedias, disminuyendo la cantidad de parámetros y computación en la red y de esta forma creando abstracciones de la información. El pooling se logra a través de operaciones como el max pooling, que conservan las características más importantes y elimina la mayoría del ruido de la entrada.

- *Capa de normalización por bloques*: la capa de normalización por bloques o The Batch Normalization (BN) se utiliza comúnmente para mejorar el aprendizaje de la red al tener la información en un rango finito de valores. Usualmente este proceso se puede aplicar antes de que la información entre a la red o durante, en el primer caso la normalización deberá ser ajustada en función de los valores máximos y mínimos de cada uno de los parámetros de entrada, en el segundo caso esta normalización se hace de forma automática por la red y ayuda regularizar el modelo y prevenir los overfitting (sobre aprendizajes).

- *Capas de Dropout*: la capa de dropout es otra capa utilizada para mejorar el aprendizaje y prevenir overfitting. A diferencia de la capa de normalización, la capa de dropout trabaja de forma tal que elimina parte del aprendizaje de la red durante cada una de las iteraciones de entrenamiento. Esto previene que la red no aprenda linealmente o "de memoria" y así lograr mejores valores de precisión (accuracy).

- *Capa de salida o capa de clasificación:* la capa de clasificación es la capa de la CNN que produce los valores de salida final de la red, que puede ser un vector de probabilidades para la clasificación en problemas de clasificación o un valor para problemas de regresión.

- *Función de pérdida*: La función de pérdida es la encargada de evaluar las discrepancias entre las predicciones de la red y las etiquetas reales del conjunto de entrenamiento. El objetivo durante el entrenamiento es minimizar esta función, ajustando los pesos de la red.

- Optimizador: El optimizador es responsable de ajustar los pesos de la red durante el entrenamiento para minimizar la función de pérdida. Algoritmos comunes de optimización incluyen el descenso de gradiente estocástico (SGD), Adam, RMSprop, entre otros.

### 3.1.2 Diagrama de Swimlane

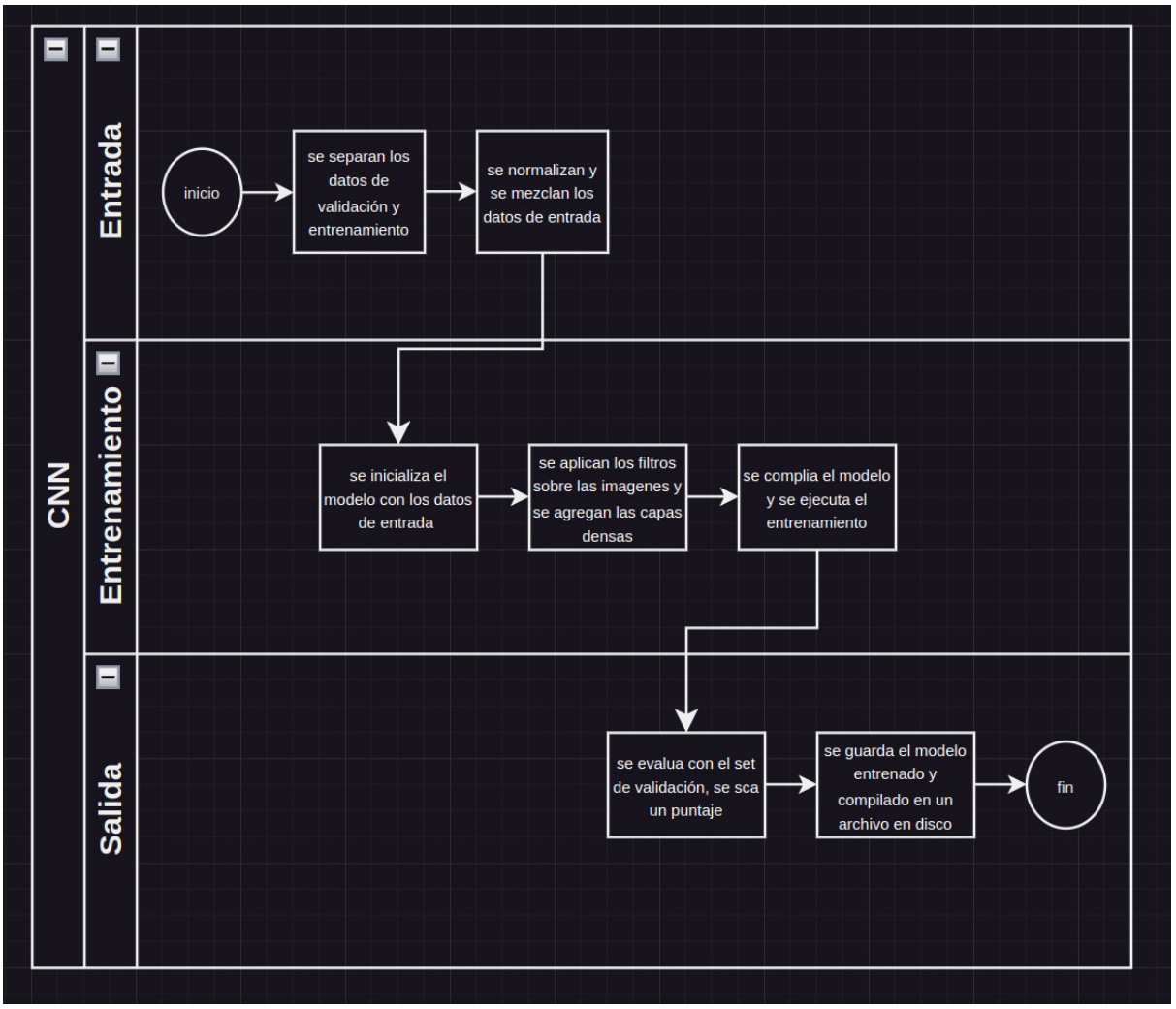



Figura 7: Diagrama de Swinlane para entender el proceso de entrenamiento de una CNN.

Aclaraciones:

- Los datos de entrada hacen referencia tanto a las etiquetas como a las imágenes.
- Los datos de validación y entrenamiento usualmente pertenecen al set de datos original que se subdivide y se utiliza para que la red valide su entrenamiento.
- La normalización y aleatoriedad de los datos, como se explicó previamente, es para que se logre un mejor aprendizaje de la red.
- El puntaje de evaluación de desempeño de la red se basa en los valores de Training accuracy y Validation accuracy.
- El modelo se guarda en un archivo binario compilado de formato .h5.

## 3.2 Desarrollo de una CNN en Python con Tensorflow

Con el fin de reconstruir los valores originales de los cilindros dieléctricos se desarrolló una herramienta de software utilizando el lenguaje de Python e implementando las librerías de Tensorflow para solucionar el problema inverso planteado en este trabajo. Este software es capaz de realizar todo lo descrito anteriormente y como resultado compilar una red neuronal que deberá ser capaz de poder resolver el problema de predicción.

El software de CNN se divide en dos grandes scripts o módulos el cual el trabajo de cada uno es por un lado entrenar la red dados los datos de entrada y realizar una evaluación de desempeño y el segundo script será el encargado de cargar el modelo de red compilado, desnormalizar la información y traducir la misma en datos útiles para el usuario.

### 3.2.1 Herramientas utilizadas

Lenguaje de programación:

- Python 3.10

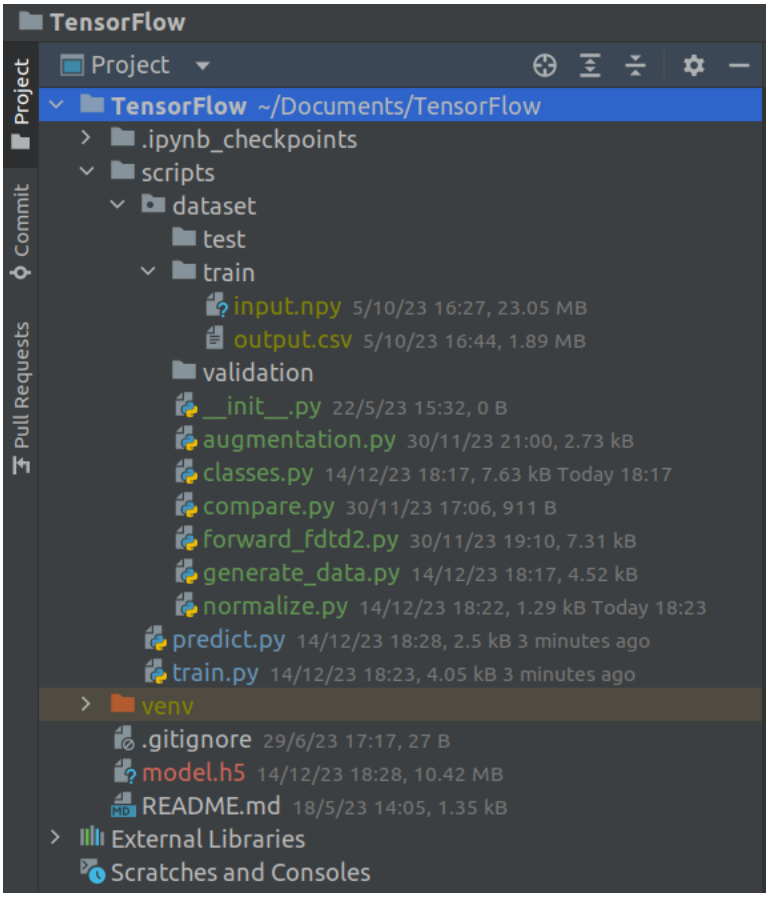
Librerías principales:

- PyMeep 1.26.0
- TensorFlow 2.12.0
- Keras 2.12.0
- Matplotlib 3.7.1
- Numpy 1.24.3
- Pandas 2.1.2

Herramientas auxiliares:

- Virtualenv
- PyCharm IDE
- Jupyter Notebook

### 3.2.2 Jerarquía del proyecto




### 3.2.3 Descripción general

- Scripts: paquete que contiene los principales archivos del proyecto junto con la data de entrenamiento generada por el simulador, también contiene módulos del simulador para ejecutar una predicción y evaluar el desempeño
    - dataset: directorio que contiene toda la información relacionada a los datos de entrenamiento, validación y prueba junto con unos scripts para realizar data augmentation, comparaciones y módulos del simulador para generar nuevos datos de prueba.
        - normalize.py: módulo encargado de contener las funciones de normalización y desnormalización para las coordenadas (x,y) y los valores dieléctricos.

- predict.py: módulo encargado a partir del modelo compilado de la red, realizar una predicción, desnormalización y comparación de la información predicha por la red neuronal.
- train.py: módulo encargado de entrenar la red y que contiene las funciones para la carga de datos, la normalización, la definición del modelo, la compilación y los resultados de la misma.

- Model.h5: binario compilado generado por el entrenamiento de la CNN.

### 3.2.4 Módulo de entrenamiento de la red: train.py

A continuación se explicará el funcionamiento dentro del código de train.py responsable del entrenamiento de la red neuronal el cual contiene las definiciones topológicas de la misma.

Se importan todas las librerías a utilizar y se define el valor del directorio base.

```
import os
import numpy as np
import pandas as pd
import tensorflow
from tensorflow.keras import layers, models
import tensorflow as tf
from matplotlib import pyplot as plt
from dataset.normalize import norm_value

base_dir = os.getcwd()
```

Se cargan los datos de entrada (la salida del simulador) y se preparan los datos.

```
# Load data from CSV and NPY files
csv_data = pd.read_csv(base_dir + '/scripts/dataset/train/output.csv')
input_data = np.load(base_dir + '/scripts/dataset/train/input.npy', allow_pickle=True)

# Reshape the images
input_data = np.asarray(input_data).astype('float32')
images_reshaped = input_data.reshape(input_data.shape[0], input_data.shape[1], input_data.shape[2], 1)


# Convert the data to a NumPy array
data_array = csv_data.values
```

Se normalizan los valores utilizando norm_value del módulo normalize.py.

```
# Normalize the data value by value

for i in range(10):
    normalized_data[:, i] = norm_value(data_array[:, i])
```

**Normalize.py**

Este módulo contiene las funciones encargadas de la normalización y desnormalización de la información de entrada. La función norm value devuelve un valor entre 0 y 1 dependiendo de min y max de el universo de valores de entrada para ese tipo.

Para el radio, las coordenadas, epsilon y sigma se usa el mismo algoritmo que normaliza todo el conjunto de datos raw_points al mismo tiempo.

```
import numpy as np


def norm_value(raw_points):
    low = np.min(raw_points)
    high = np.max(raw_points)
    result = (raw_points - low) / (high - low)
    return result
```

que utiliza la siguiente fórmula:

$$zi = (xi - min(x)) / (max(x) - min(x))$$

Donde:

- *zi*: el valor normalizado del set de datos.
- *xi*: el valor en el índice i del set de datos.
- *min(x)*: el menor valor del set de datos.
- *max(x)*: el mayor valor del set de datos.

Finalizada la normalización se realiza una separación entre el set de datos utilizado para entrenar la red y el set de datos utilizado para validar la red, esto sirve para que durante el

entrenamiento la red pueda ir corrigiendo posibles desvíos entre los datos predichos y los datos reales.

```
# Data splinting for training and validation

X = tensorflow.random.shuffle(images_reshaped[0:5000, :], seed=8)
XVal = tensorflow.random.shuffle(images_reshaped[5000:10000, :], seed=3)
Y = tensorflow.random.shuffle(normalized_data[0:5000, 0:10], seed=8)
YVal = tensorflow.random.shuffle(normalized_data[5000:10000, 0:10], seed=3)
```

A continuación se pasa a instanciar el modelo de la red

```
# Initialize the model

model = models.Sequential()  model: <keras.engine.sequential.Sequential object at 0x7f742a9d1540>
model.add(layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu', input_shape=(16, 16, 1)))
model.add(layers.MaxPooling2D(pool_size=(2, 2)))
model.add(layers.Dropout(0.25))
model.add(layers.Conv2D(filters=128, kernel_size=(3, 3), activation='relu'))
model.add(layers.MaxPooling2D(pool_size=(2, 2)))
model.add(layers.Dropout(0.5))
model.add(layers.Flatten())
model.add(layers.Dense( units: 2304, activation='relu'))
model.add(layers.Dense( units: 100, activation='relu'))
model.add(layers.Dense( units: 100, activation='relu'))
model.add(layers.Dense( units: 100, activation='relu'))
model.add(layers.Dense( units: 10, activation='linear'))
```

Donde:

- *Sequencial*: tipo de modelo lineal en el que las capas se van a apilando una encima de otras y donde la salida de la siguiente capa es la entrada de la predecesora. Debido a su simplicidad es una de las más recomendables para usar en el entrenamiento de CNNs.
- *Conv2D*: capas de convolución en 2D de la entrada para aplicar la convolución de la información, la salida de la capa está dada según la cantidad de filtros y el tamaño del kernel.
- *MaxPooling2D*: se utiliza para realizar el submuestreo (downsampling) espacial en las dimensiones de ancho y alto de la entrada. Es comúnmente utilizada para reducir la resolución espacial de las representaciones intermedias y disminuir la cantidad de parámetros en la red.

- *Dropout*: esta capa se utiliza para que la red olvide cierta parte de la información destruyendo relaciones entre las redes neuronales y de esta forma evitar el overfitting y el aprendizaje memorizado.
- *Flatten*: se utiliza para convertir las representaciones bidimensionales o tridimensionales de las características en una única dimensión "aplanando" la entrada para que pueda ser conectada directamente a capas completamente conectadas (fully connected) que se utilizan típicamente en la parte final de una red neuronal.
- *Dense*: Representan las capas completamente conectadas donde cada neurona en la capa está conectada a cada neurona de la capa siguiente. Estas son fundamentales en la construcción de modelos de aprendizaje profundo y se utilizan para realizar tareas como clasificación, regresión y generación de secuencias.

Sumario topológico de la red

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 14, 14, 64)        640

 max_pooling2d (MaxPooling2D  (None, 7, 7, 64)         0
 )

 dropout (Dropout)           (None, 7, 7, 64)          0

 conv2d_1 (Conv2D)           (None, 5, 5, 128)         73856

 max_pooling2d_1 (MaxPooling  (None, 2, 2, 128)        0
 2D)

 dropout_1 (Dropout)         (None, 2, 2, 128)         0

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 2304)              1181952

 dense_1 (Dense)             (None, 100)               230500

 dense_2 (Dense)             (None, 100)               10100

 dense_3 (Dense)             (None, 100)               10100
```

Finalmente se compila el modelo y se ejecuta el entrenamiento

```
# Compile model

opt = tf.keras.optimizers.Adam(learning_rate=0.0001)

model.compile(loss='mean_squared_error', optimizer=opt, metrics=['accuracy'])

# Summary

model.summary()

# Fit the model
model_trained = model.fit(X, Y, validation_data=(XVal, YVal), epochs=300, batch_size=30, verbose=1)
```

El optimizador utilizado en este caso es Adam (Adaptive Moment Estimation)[5] el cual es un optimizador que combina conceptos de dos otros optimizadores: el descenso de gradiente estocástico (SGD) con momento y el RMSprop (Root Mean Square Propagation). Es adaptable, eficaz y suele ser una elección sólida para entrenar modelos de aprendizaje profundo más aún en las redes de detección de imágenes.

Se define un learning_rate bajo para asegurar un entrenamiento seguro y conciso.

X e Y hacen referencia a los valores de entrada (imágenes y etiquetas).

Xval e Yval hacen referencia a los valores de entrada que la red no va a utilizar para aprender sino para validar el aprendizaje realizado en cada época.

El parámetro epochs o épocas indica cuántas veces se desea que el modelo recorra todo el conjunto de entrenamiento, en otras palabras, un “epoch” se completa cuando el modelo ha visto y ha sido ajustado a cada muestra en el conjunto de entrenamiento una vez.

El parámetro batch_size o lote es el tamaño de muestra que refiere a la cantidad de datos que van a ser utilizados en cada iteración.

### 3.2.5 Salida y resultados

El entrenamiento finaliza por un lado con dos gráficos representando los valores del training_acurracy y training_loss y por otro lado almacenando en la raíz del proyecto el modelo compilado resultante del entrenamiento por la CNN.

Se guarda el modelo en disco.

```
# Save the model
model.save('model.h5')
```

```
.gitignore 29/6/23 17:17, 27 B
model.h5 14/12/23 22:15, 18.16 MB
README.md 18/5/23 14:05, 1.35 kB
```

Resultados finales.

```
Epoch 297/300
167/167 [==============================] - 1s 9ms/step - loss: 0.0213 - accuracy: 0.4834 - val_loss: 0.0258 - val_accuracy: 0.4414
Epoch 298/300
167/167 [==============================] - 1s 9ms/step - loss: 0.0214 - accuracy: 0.4846 - val_loss: 0.0273 - val_accuracy: 0.4186
Epoch 299/300
167/167 [==============================] - 1s 9ms/step - loss: 0.0214 - accuracy: 0.4904 - val_loss: 0.0265 - val_accuracy: 0.4348
Epoch 300/300
167/167 [==============================] - 1s 9ms/step - loss: 0.0212 - accuracy: 0.4908 - val_loss: 0.0262 - val_accuracy: 0.4432
157/157 [==============================] - 0s 998us/step - loss: 0.0262 - accuracy: 0.4432
Test loss: 0.0261650700122118
Test accuracy: 0.4431999921798706
```

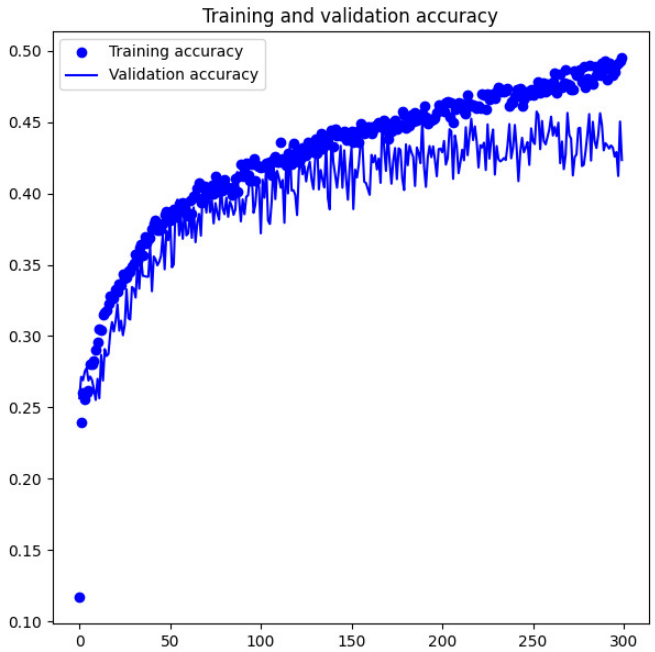



Figura 8: Resultado final comparativo entre la exactitud del entrenamiento y la validación

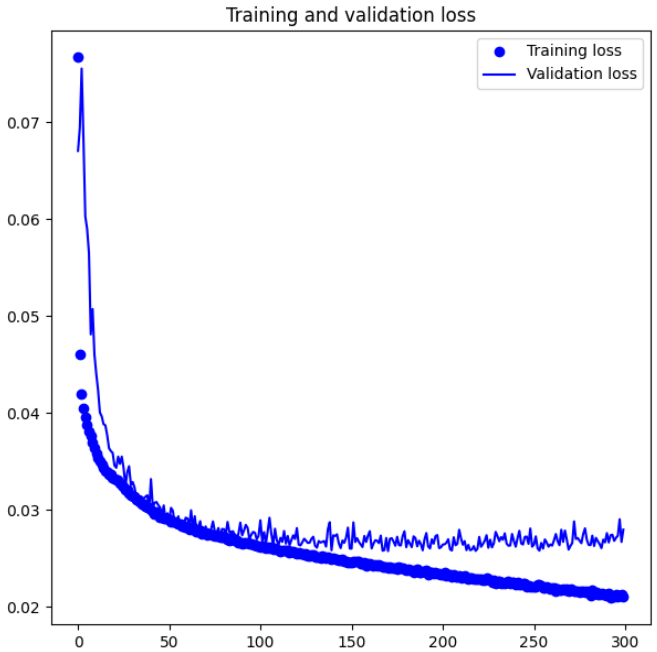



Figura 9: Resultado final comparativo entre la pérdida del entrenamiento y la validación

### 3.2.6 Módulo de predicción de valores: predict.py

Finalmente, en esta sección, se dará el detalle del funcionamiento del script de predicción de valores predict.py encargado de cargar el modelo compilado model.h5 del módulo de entrenamiento y realizar una predicción de los valores geométrico/dieléctricos de los cilindros dada una imagen del campo eléctrico.

Se importan las librerías, la función generate_models del modulo de generate_data.py y la función de desnormalización denorm_epsilon, denorm_coord y denorm_sigma del módulo de normalize.

```
from scripts.dataset.generate_data import generate_models
from scripts.dataset.classes import Cylinder
from keras.models import load_model
from dataset.normalize import denorm_r, denorm_coord, denorm_epsilon, denorm_sigma
import numpy as np
import os

base_dir = os.getcwd()
```

Funciones de desnormalización

```
def denorm_r(value, low=0.005, high=0.04):
    result = (value * (high - low)) + low
    return result


def denorm_coord(value, low=-0.06, high=0.06, ext_r=None, int_r=None, coord=None):
    if int_r and ext_r and coord:
        high = coord + (ext_r - int_r)
        low = coord - (ext_r - int_r)
        result = (value * (high - low)) + low
    else:
        result = (value * (high - low)) + low
    return result


def denorm_epsilon(value, low=10, high=80):
    result = (value * (high - low)) + low
    return result


def denorm_sigma(value, low=0.4, high=1.6):
    result = (value * (high - low)) + low
    return result
```

Estas funciones aplican la función inversa a la de norm_value, para lograrlo utilizan valores fijos conocidos para los máximos y mínimos dado el conjunto de datos de entrada.

Se define la función make_random_prediction cuyo proceso será el de cargar el modelo entrenado, llamar a la función de generate_models y luego realizar un predicción con el método de predict sobre el model.h5

```
11  def make_random_prediction():
12      model = load_model(base_dir + '/model.h5')
13
14      img, lbl = generate_models()
15
16      img = np.asarray([img]).astype('float32')
17
18      img = img.reshape(img.shape[0], img.shape[1], img.shape[2], 1)
19
20      raw_data = model.predict(img)
```

Luego de generar los valores para la predicción se deben desnormalizar utilizando las funciones vistas anteriormente, una vez desnormalizados se toman ambos valores, de la predicción y del objeto original para crear dos instancias de Cylinder y de esta forma poder comparar los resultados y sus representaciones gráficas.

```
22  pred_dict = dict()
23
24  pred_dict['radius'] = denorm_r(raw_data[0][0])
25  pred_dict['xc'] = denorm_coord(raw_data[0][1])
26  pred_dict['yc'] = denorm_coord(raw_data[0][2])
27  pred_dict['iradius'] = denorm_r(raw_data[0][3])
28  pred_dict['ixc'] = denorm_coord(raw_data[0][4])
29  pred_dict['iyc'] = denorm_coord(raw_data[0][5])
30  pred_dict['epsilon'] = denorm_epsilon(raw_data[0][6])
31  pred_dict['sigma'] = denorm_sigma(raw_data[0][7])
32  pred_dict['iepsilon'] = denorm_epsilon(raw_data[0][8])
33  pred_dict['isigma'] = denorm_sigma(raw_data[0][9])
34
35  print(lbl)
36  print(pred_dict)
37
38  original = Cylinder( antenna: None, lbl['radius'], lbl['xc'], lbl['yc'], lbl['iradius'], lbl['ixc'], lbl['iyc'])
39  original.epsilon = lbl['epsilon']
40  original.iepsilon = lbl['iepsilon']
41  original.sigma = lbl['sigma']
42  original.isigma = lbl['isigma']
43  original.draw()
44
45  predicted = Cylinder( antenna: None, pred_dict['radius'], pred_dict['xc'], pred_dict['yc'], pred_dict['iradius'], pred_dict['ixc'], pred_dict['iyc'])
46  predicted.epsilon = pred_dict['epsilon']
47  predicted.iepsilon = pred_dict['iepsilon']
48  predicted.sigma = pred_dict['sigma']
49  predicted.isigma = pred_dict['isigma']
50  predicted.draw()
```

Al finalizar el script los valores tanto reales como predichos se almacenan en el disco para su posterior análisis. En la carpeta prediction_simulations se guardarán por un lado las imágenes en .png de la geometría del cilindro original/predicho y los valores numéricos correspondientes a las propiedades geométricas y dieléctricas en formato .json

A continuación se muestran unos ejemplos de dichos archivos:

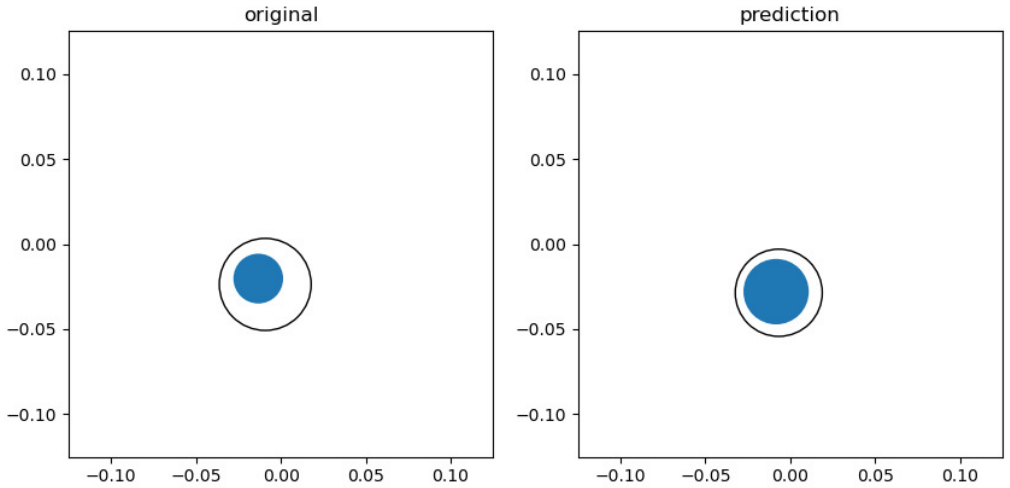



*Figura 10: Imagenes generadas por el módulo de predicción de la CNN.*

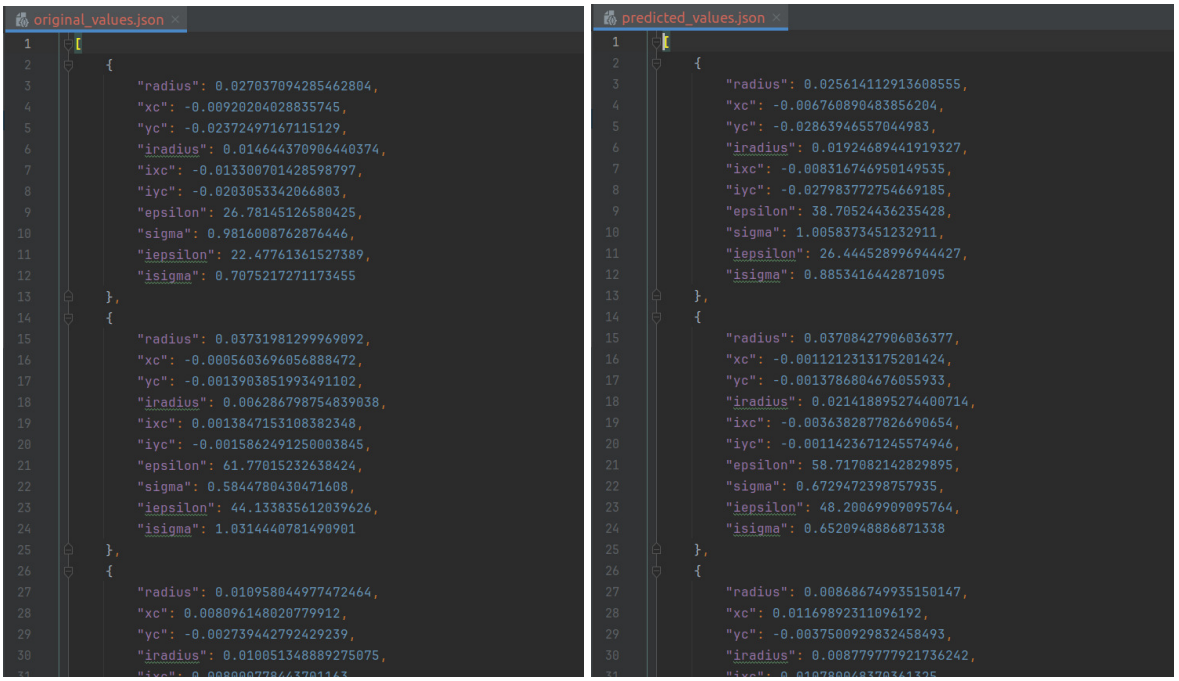



*Figura 11: Valores correspondientes a los archivos .json almacenados en disco.*

# Capítulo 4

# Análisis de los resultados

En esta sección se presentará un análisis sobre los resultados obtenidos por la red neuronal en contraste a los valores que fueron simulados. Cabe destacar que como vimos en el capítulo anterior, la red finalizó su entrenamiento con valores bajos en términos de training_accuracy y training_vaidation, lo que conlleva inevitablemente a trabajar con valores prácticos aproximados.

A continuación se expone la información que fue relevada a un total de cincuenta muestras simuladas. Con el objetivo de simplificar se mostrarán únicamente diez valores aleatorios de cada tabla. Se utilizará una tasa de acierto al milímetro contemplando dos dígitos decimales para su comparación.

## 4.1 Resultados: Propiedades Geométricas

| Campo | Valor real (m) | Valor predicho (m) | Error relativo |
|---|---|---|---|
| Radio | 0.027 | 0.025 | 0.052 |
| | 0.037 | 0.037 | 0.006 |
| | 0.010 | 0.008 | 0.207 |
| | 0.034 | 0.036 | 0.079 |
| | 0.033 | 0.033 | 0.011 |
| | 0.021 | 0.021 | 0.002 |
| | 0.012 | 0.008 | 0.295 |
| | 0.031 | 0.032 | 0.032 |
| | 0.037 | 0.036 | 0.014 |
| | 0.011 | 0.008 | 0.261 |

Tasa de acierto: 82%

Error relativo promedio: 0.002

| Campo | Valor real (m) | Valor predicho (m) | Error relativo |
|---|---|---|---|
| Radio interno | 0.014 | 0.019 | 0.314 |
| | 0.006 | 0.021 | 2.406 |
| | 0.010 | 0.008 | 0.126 |
| | 0.027 | 0.023 | 0.173 |
| | 0.021 | 0.021 | 0.036 |
| | 0.007 | 0.013 | 0.705 |
| | 0.010 | 0.008 | 0.189 |
| | 0.021 | 0.019 | 0.060 |
| | 0.006 | 0.020 | 2.180 |
| | 0.008 | 0.008 | 0.027 |

Tasa de acierto: 64%
Error relativo promedio: 0.004

| Campo | Valor real (m) | Valor predicho (m) | Error relativo |
|---|---|---|---|
| Centro X cilindro ext. | -0.009 | -0.006 | 0.265 |
| | -0.0005 | -0.001 | 1.000 |
| | 0.008 | 0.011 | 0.444 |
| | 0.0006 | 0.005 | 6.379 |
| | 0.008 | 0.011 | 0.369 |
| | 0.007 | 0.009 | 0.274 |
| | -0.007 | -0.005 | 0.267 |
| | 0.006 | 0.009 | 0.415 |
| | -0.008 | -0.008 | 0.022 |
| | -0.012 | -0.010 | 0.159 |

Tasa de acierto: 82%
Error relativo promedio: 0.002

| Campo | Valor real (m) | Valor predicho (m) | Error relativo |
|---|---|---|---|
| Centro Y cilindro ext. | -0.023 | -0.028 | 0.207 |
| | -0.001 | -0.001 | 0.008 |
| | -0.002 | -0.003 | 0.368 |
| | -0.001 | -0.003 | 1.526 |
| | -0.004 | -0.005 | 0.054 |
| | 0.0103 | 0.011 | 0.080 |
| | -0.018 | -0.021 | 0.177 |
| | 0 | 0 | 3.909 |
| | -0.006 | -0.004 | 0.293 |
| | 0.0132 | 0.014 | 0.087 |

Tasa de acierto: 66%
Error relativo promedio: 0.002

| Campo | Valor real (m) | Valor predicho (m) | Error relativo |
|---|---|---|---|
| Centro X cilindro int. | -0.013 | -0.008 | 0.374 |
| | 0.001 | -0.003 | 3.627 |
| | 0.008 | 0.0107 | 0.347 |
| | 0.003 | 0.003 | 0.060 |
| | 0.007 | 0.010 | 0.366 |
| | 0.005 | 0.008 | 0.451 |
| | -0.007 | -0.005 | 0.197 |
| | 0.002 | 0.009 | 2.796 |
| | 0 | -0.009 | 12.09 |
| | 0.011 | -0.012 | 0.166 |

Tasa de acierto: 74%
Error relativo promedio: 0.003

| Campo | Valor real (m) | Valor predicho (m) | Error relativo |
|---|---|---|---|
| Centro Y cilindro int. | -0.020 | -0.027 | 0.378 |
| | -0.001 | -0.001 | 0.279 |
| | -0.002 | 0 | 0.257 |
| | 0 | -0.005 | 0.574 |
| | -0.011 | 0.013 | 0.007 |
| | 0.013 | 0 | 0.545 |
| | -0.018 | -0.019 | 0.057 |
| | -0.002 | 0 | -0.572 |
| | 0.009 | 0.003 | 1.338 |
| | 0.013 | 0.014 | 0.146 |

Tasa de acierto: 68%
Error relativo promedio: 0.003

A continuación se presentan los histogramas y diagramas de dispersión de la diferencia a partir de las cincuenta muestras utilizadas para la prueba.

**Histogramas**

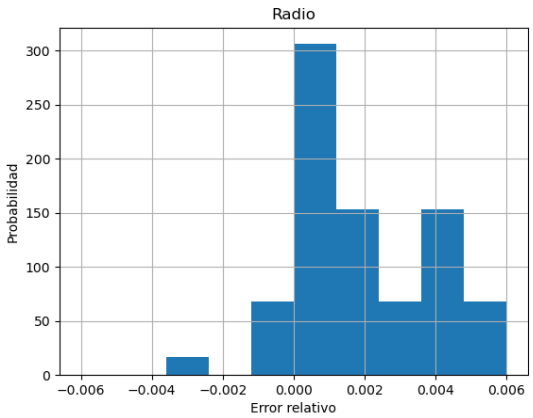



*Figura 12 a: Histograma error relativo radio.*

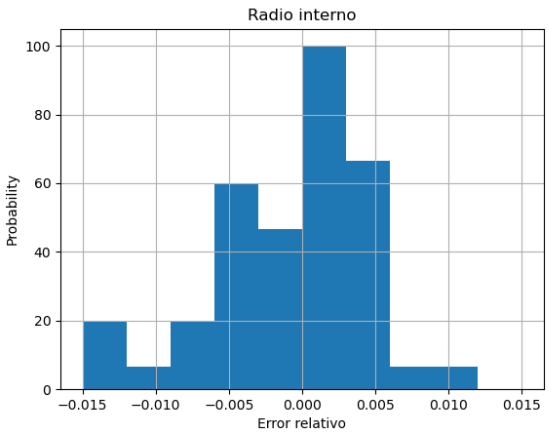



*Figura 12 b: Histograma error relativo radio interno.*

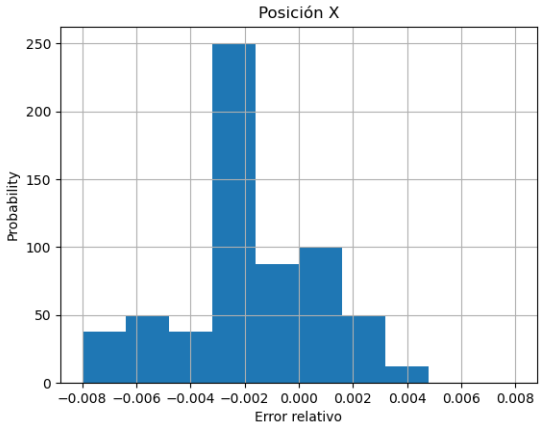



*Figura 14 a: Histograma error relativo posición eje X.*

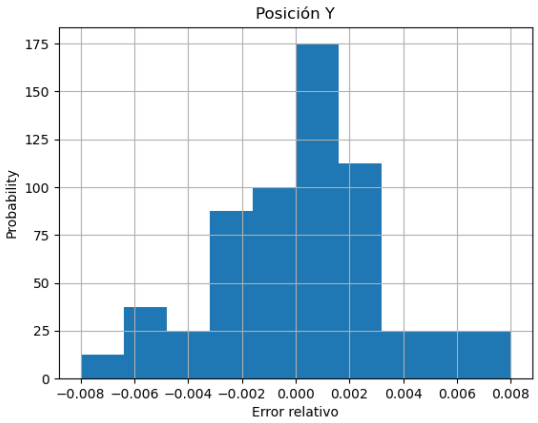



*Figura 14 b: Histograma error relativo posición eje X interno.*

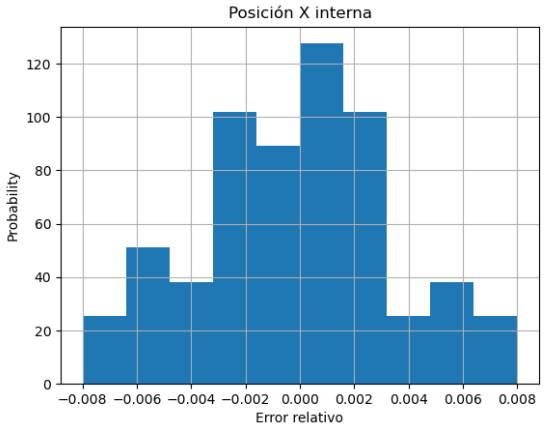



*Figura 14 c: Histograma error relativo posición eje Y.*

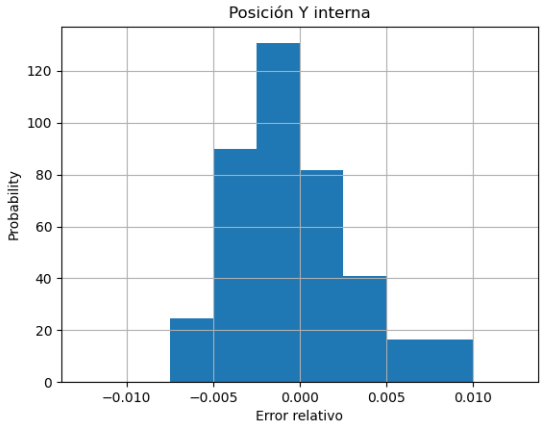



*Figura 14 d: Histograma error relativo posición eje Y interno.*

## Diagramas de dispersión

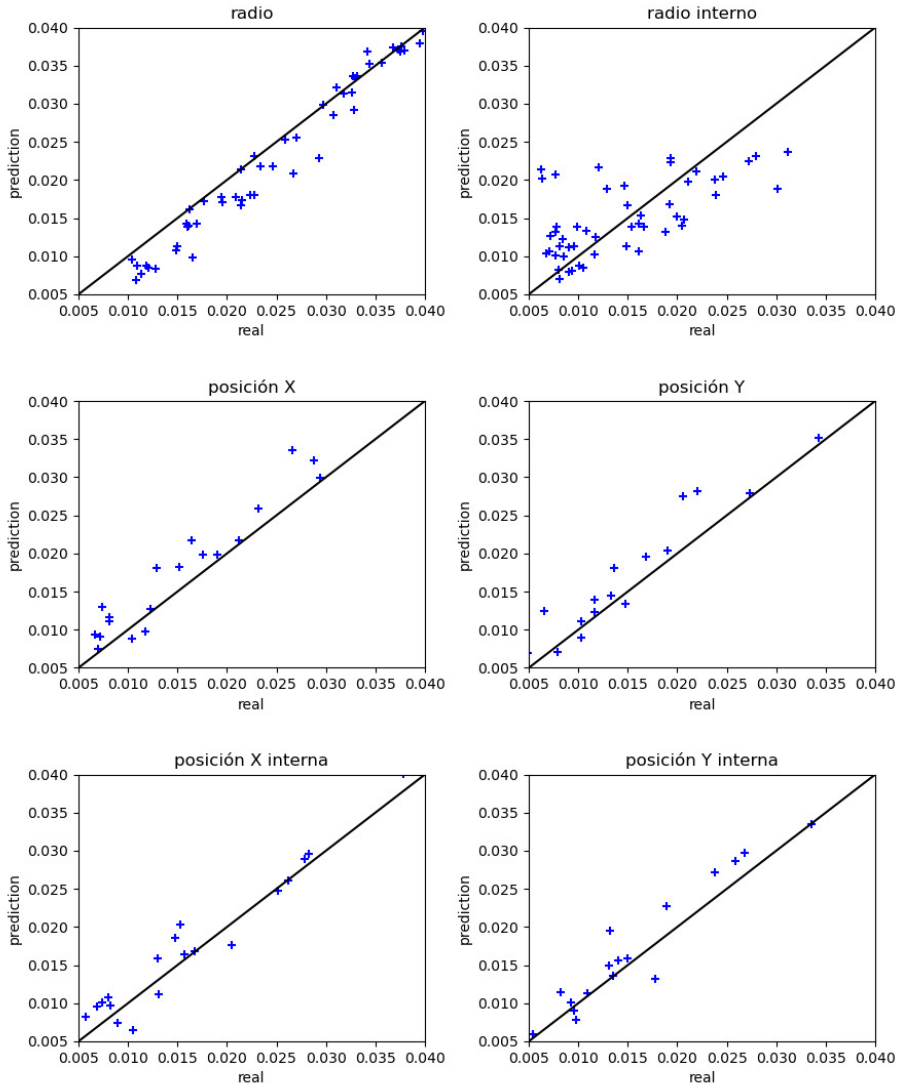



*Figura 15: Diagramas de dispersión para el radio, radio interno, posición X, posición X interna, posición Y y posición Y interna para las propiedades geométricas.*

## Muestra de la representación gráfica para los cilindros

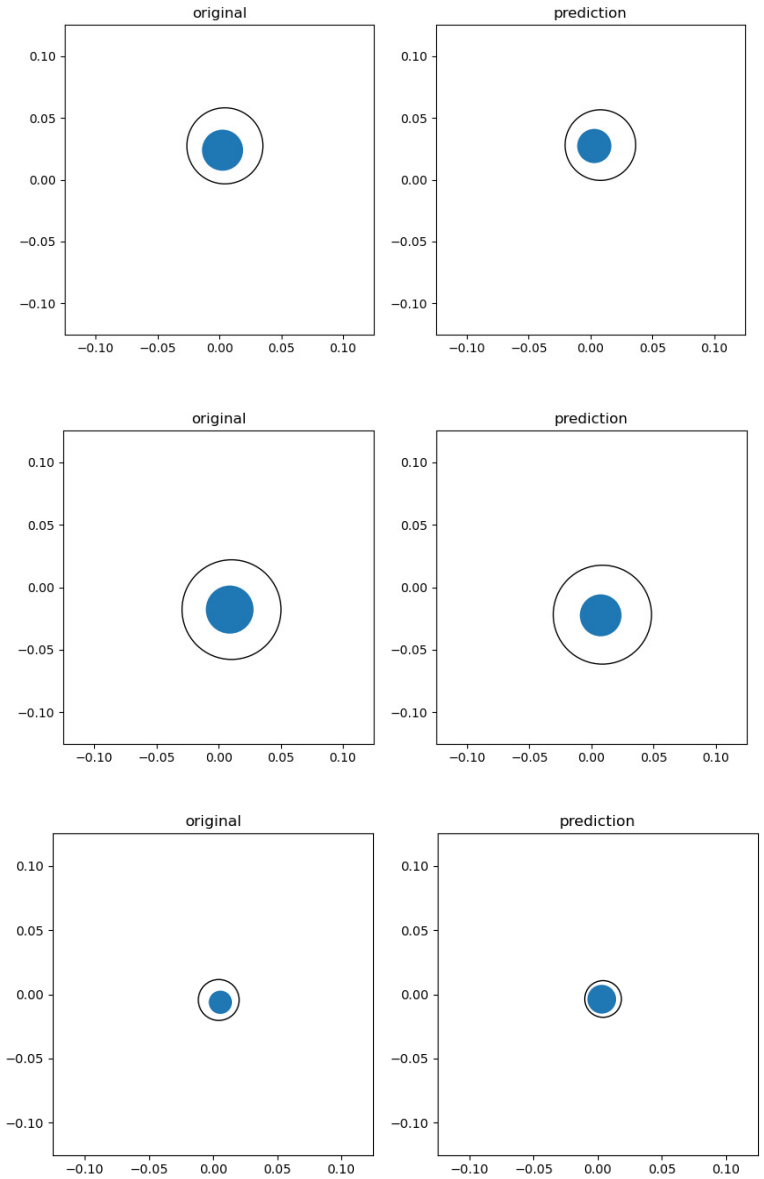



*Figura 16: Tres representaciones visuales entre geometría de los cilindros reales y predichos por la red neuronal.*

## 4.2 Resultados: Propiedades dieléctricas

| Campo | Valor real (S/m) | Valor predicho(S/m) | Error relativo |
|---|---|---|---|
| Conductividad | 0.98 | 1 | 0.44 |
| | 0.58 | 0.67 | 0.04 |
| | 1.54 | 1.10 | 0.37 |
| | 0.95 | 1.25 | 0.08 |
| | 1.15 | 1.22 | 0.12 |
| | 0.59 | 0.70 | 0.03 |
| | 0.50 | 0.86 | 0.08 |
| | 1.23 | 1.08 | 0.15 |
| | 1.31 | 1.34 | 0.12 |
| | 1.39 | 1.01 | 0.33 |

Tasa de acierto: 12% *1
Error relativo promedio: 0.23

| Campo | Valor real (S/m) | Valor predicho (S/m) | Error relativo |
|---|---|---|---|
| Permitividad relativa | 26.78 | 38.70 | 0.02 |
| | 61.77 | 58.71 | 0.15 |
| | 12.88 | 17.70 | 0.28 |
| | 78.84 | 72.48 | 0.15 |
| | 49.67 | 43.38 | 0.05 |
| | 42.12 | 40.64 | 0.19 |
| | 29.23 | 31.75 | 0.72 |
| | 33.37 | 28.04 | 0.12 |
| | 65.51 | 57.37 | 0.02 |
| | 68.99 | 46.05 | 0.27 |

Tasa de acierto: 0% *2
Error relativo promedio: 13.54

| Campo | Valor real (S/m) | Valor predicho (S/m) | Error relativo |
|---|---|---|---|
| Conductividad cilindro Int. | 0.70 | 0.88 | 0.25 |
| | 1.03 | 0.65 | 0.36 |
| | 1.12 | 0.98 | 0.13 |
| | 0.80 | 1.11 | 0.38 |
| | 1.50 | 1.19 | 0.20 |
| | 1.37 | 0.98 | 0.28 |
| | 0.96 | 1.03 | 0.06 |
| | 0.91 | 1.04 | 0.13 |
| | 0.50 | 0.93 | 0.82 |
| | 0.74 | 1 | 0.35 |

Tasa de acierto: 12% *1
Error relativo promedio: 0.24

| Campo | Valor real (S/m) | Valor predicho (S/m) | Error relativo |
|---|---|---|---|
| Permitividad cilindro Int. | 22.47 | 26.44 | 0.17 |
| | 44.13 | 48.20 | 0.09 |
| | 20.75 | 11.71 | 0.43 |
| | 10.67 | 36.27 | 2.39 |
| | 58.76 | 55.77 | 0.05 |
| | 21.57 | 34.20 | 0.58 |
| | 29.89 | 23.11 | 0.22 |
| | 14.12 | 24.31 | 0.72 |
| | 51.10 | 60.35 | 0.18 |
| | 47.46 | 60.35 | 0.06 |

Tasa de acierto: 6% *2
Error relativo promedio: 8.81

***1**: La conductividad al tener un rango de valores chicos entre 0 y 1,6 se decidió utilizar una tasa de asertividad con un solo decimal.
***2**: La permitividad al tener un rango de valores grandes entre 10 y 80 se decidió utilizar una tasa de asertividad sin decimales.

## Histogramas

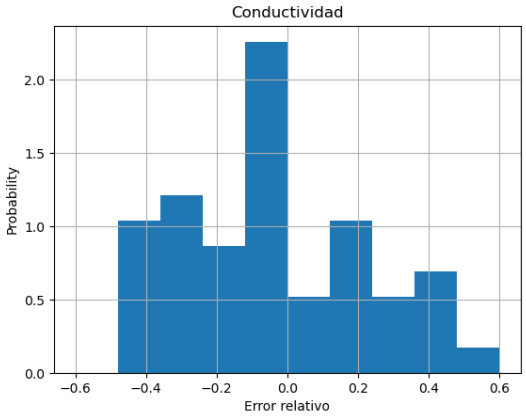



*Figura 17 a: Histograma error relativo conductividad.*

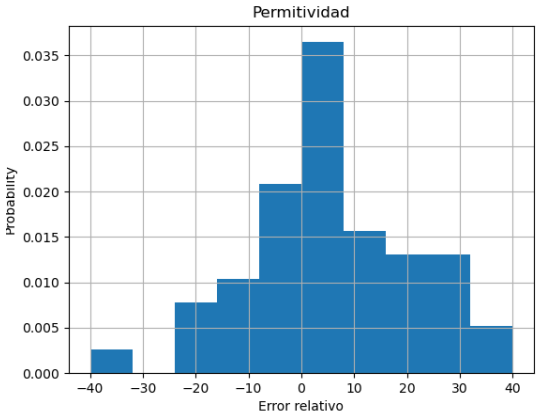



*Figura 17 b: Histograma error relativo permitividad.*

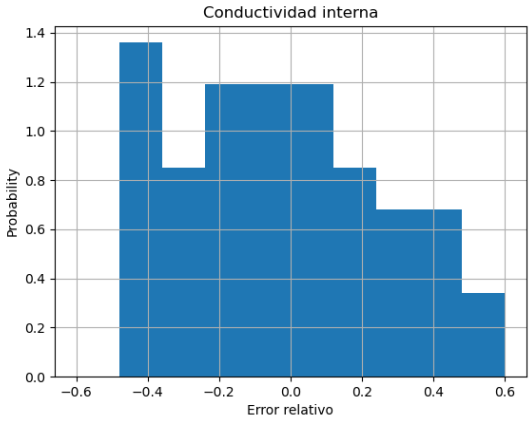



*Figura 18 a: Histograma error relativo conductividad interna.*

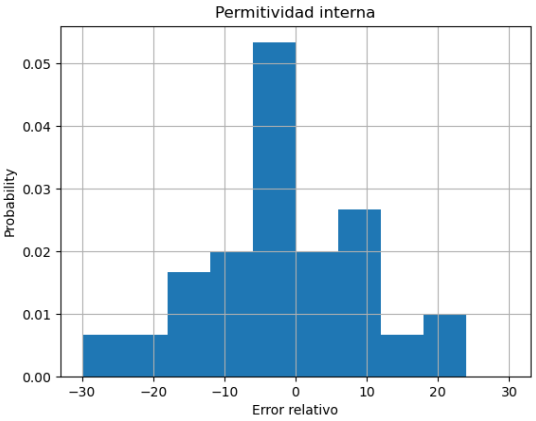



*Figura 18 b: Histograma error relativo permitividad interna.*

## Diagramas de dispersión

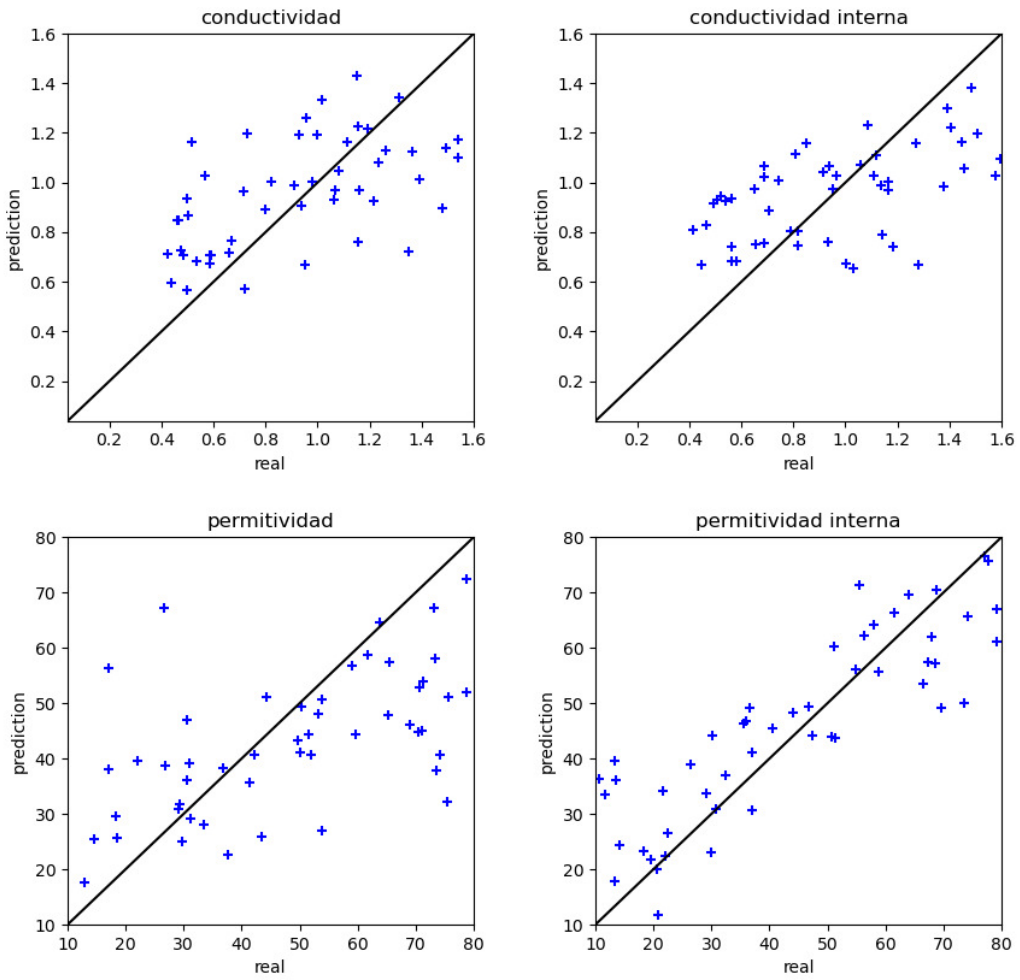



*Figura 19: Diagramas de dispersión para la conductividad y permitividad de los cilindros externos e internos.*

# Capítulo 5

# Discusión general y conclusiones

A lo largo de este trabajo, el foco de estudio ha sido las redes neuronales, que a su vez son parte integral de la tecnología de Inteligencia Artificial (IA). Comprender su desarrollo y evolución, tanto desde una perspectiva histórica como analítica, revela que aún hay muchos aspectos por explorar y mejorar. Aunque las redes neuronales son herramientas poderosas, su naturaleza en constante evolución indica que hay un continuo espacio para el crecimiento y la mejora.

Si bien las redes neuronales forman parte de la IA, es fundamental reconocer que abarca un espectro más amplio de tecnologías y enfoques. Las redes neuronales son solo una de las muchas herramientas dentro del vasto campo de la inteligencia artificial, diseñadas para simular ciertos aspectos del funcionamiento del cerebro humano.

La tecnología subyacente a las redes neuronales sigue evolucionando, y a pesar de ser una herramienta valiosa, todavía está en desarrollo. Como refleja el propio concepto de Inteligencia Artificial, esta disciplina está intrínsecamente vinculada a nuestra comprensión de la información y la búsqueda de patrones, reflejando la lógica subyacente detrás de los fenómenos físicos. A pesar de los avances significativos, la Inteligencia Artificial continúa siendo una exploración en curso, donde la capacidad de entender y simular la inteligencia humana es un desafío constante.

Parte de este capítulo está dedicado una breve sección conclusiones en el análisis de los resultados obtenidos y las mejoras a nivel lógico que pueden ser aplicadas al software con el fin de conseguir resultados más precisos y mejorar el proceso de entrenamiento y aprendizaje de la CNN.

En el Capítulo 2 se introdujo la problemática que atañe a este trabajo la cual era la predicción de valores físicos, de materiales reales de los cuales la única información con la cual se podía trabajar eran sus mediciones externas en forma de, si se quiere decir, interferencias generadas en el espectro del campo eléctrico. Si bien hay procesos físicos que están directamente relacionados a lo que reflejan las mediciones, como interferencias, perturbaciones, errores de medición, etc. A ciencia cierta es imposible que la medición refleje todos y cada uno de los resultados en relación al aspecto original del objeto, o al menos captar por completo todas las variables en juego para una recreación exacta, entonces, desde un punto de vista teórico de este trabajo el objetivo siempre fue una aproximación a los valores iniciales como el punto de partida a trabajar sobre ellos, incrementando por encima capas más complejas de análisis de datos implementando técnicas que no son ni están necesariamente relacionadas a la IA sino más bien a otra clase de ciencia analitica que puedan ayudar a una reconstrucción más exacta.

El Capítulo 3 explica la implementación integral de la red neuronal detallando cada una de las piezas fundamentales de su arquitectura, si bien no fue parte del trabajo, a lo largo de este proyecto se realizó una extensa prueba de hiperparámetros variando tanto el optimizador, la función de pérdida y el activador en cada capa de convolución como se muestra en la Fig.12. El valor utilizado para cada uno de los hiperparámetros no fue arbitrario y las pruebas demostraron que Adam junto con ReLu y mean_squared_error resultaron ser los mejores algoritmos en la resolución de este problema.
Adicionalmente, hubo correcciones dentro del simulador que fueron parte fundamental en el afinamiento impactando directamente en los resultados del entrenamiento. Otros arreglos también tuvieron lugar en la normalización de la información correspondiente a los valores de los cilindros en la cual se decidió de pasar a un normalización general a una particular para cada dato, resultando en predicciones más precisas.

En el Capítulo 4 se expone toda la información relevada a través del análisis de los resultados de la red neuronal, en los mismo se puede hacer una primera apreciación de la tasa de aprendizaje para cada tipo de característica, por un lado, los datos geométricos (radio, posición interna y externa) tuvieron mejores resultados que los valores dieléctricos

(permitividad y conductividad). Una de las conjeturas posibles es que el método de simulación del campo eléctrico sea más sensible y haya en cierta forma una relación más lineal entre el resultado y las propiedades geométricas por encima de las dieléctricas y que en dicho caso la red no pueda construir un peso adecuado a nivel neuronal para predecir dichos valores de forma correcta.

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | Plan de pruebas CNN: tabla de hiperparámetros | | | | | | |
| 2 | | Combinación 1 | | | | | |
| 3 | Funcion de activación | Softmax | | | | | |
| 4 | Optimizador | Adam | | SGD | | RMSprop | |
| 5 | Funcion de perdida | Mean Absolute Error | Mean Squared Error | Mean Absolute Error | Mean Squared Error | Mean Absolute Error | Mean Squared Error |
| 6 | Tasa de aprendizaje | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |
| 7 | Dropout | 25/40 | 25/40 | 25/40 | 25/40 | 25/40 | 25/40 |
| 8 | Relación Datos de prueba/validación | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 |
| 9 | Resultados | | | | | | |
| 10 | Validation loss: | 0.04 | 0.01 | 0.04 | 0.01 | 0.05 | 0.01 |
| 11 | Training Validation: | 0.49 | 0.50 | 0.49 | 0.49 | 0.49 | 0.49 |
| 12 | | Combinación 2 | | | | | |
| 13 | Funcion de activación | ReLu | | | | | |
| 14 | Optimizador | Adam | | SGD | | RMSprop | |
| 15 | Funcion de perdida | Mean Absolute Error | Mean Squared Error | Mean Absolute Error | Mean Squared Error | Mean Absolute Error | Mean Squared Error |
| 16 | Tasa de aprendizaje | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |
| 17 | Dropout | 25/40 | 25/40 | 25/40 | 25/40 | 25/40 | 25/40 |
| 18 | Relación Datos de prueba/validación | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 |
| 19 | Resultados | | | | | | |
| 20 | Validation loss: | 0.03 | 0.01 | 0.04 | 0.01 | 0.03 | 0.009 |
| 21 | Training Validation: | 0.62 | 0.61 | 0.49 | 0.56 | 0.62 | 0.62 |
| 22 | | Combinación 3 | | | | | |
| 23 | Funcion de activación | LeakyReLu | | | | | |
| 24 | Optimizador | Adam | | SGD | | RMSprop | |
| 25 | Funcion de perdida | Mean Absolute Error | Mean Squared Error | Mean Absolute Error | Mean Squared Error | Mean Absolute Error | Mean Squared Error |
| 26 | Tasa de aprendizaje | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |
| 27 | Dropout | 25/40 | 25/40 | 25/40 | 25/40 | 25/40 | 25/40 |
| 28 | Relación Datos de prueba/validación | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 | 5000/5000 |
| 29 | Resultados | | | | | | |
| 30 | Validation loss: | 0.03 | | | | | |
| 31 | Training Validation: | 0.60 | | | | | |

*Figura 20. Planilla de plan de prueba para cada combinación de hiperpáramtros evaluados de forma empírica*

## 5.1 Posibles Mejoras

Esta sección está dedicada a la discusión de posibles mejoras que pueden mejorar el desempeño general de la red tanto a nivel topológico como a nivel procedimental en el tratado y transformación los datos de entrada asi como también técnicas preexistentes para ahorrar procesamiento en la red neuronal.

- **Data augmentation:** Una de las posibles causas puede estar atada a la cantidad de datos que se utilizaron de entrenamiento y validación. Por un tema de tiempos se decidió utilizar un set de prueba discreto de diez mil datos, los cuales fueron divididos en dos conjuntos de cinco mil cada uno correspondientemente. Una solución viable a esto sería aplicar técnicas de data augmentation sobre la entrada para generar un mayor volumen artificial de datos para garantizar cierto grado de uniformidad.
- **Función de pérdida personalizada:** El desarrollo de una función de pérdida personalizada también puede ayudar a mejorar el desempeño de la red neuronal ya que se pueden atribuir pesos específicos para cada valor haciendo que la red tenga más énfasis en ciertos valores por sobre otros, en este caso, podría aplicarse una función de pérdida que penalice la desviación de errores en los valores dieléctricos por sobre los geométricos.
- **Mejoras en la normalización**: Otra posible mejora se da en el campo de la normalización de la información, en este caso, se podría aplicar al algoritmo de desnormalización de forma tal que los valores predichos tomen en cuenta las restricciones geométricas de nuestro modelo original y que ayuden a una mejor predicción, como puede ser el caso de un argumento en dicha función que considere un valor máximo restrictivo para el radio del cilindro interno con respecto al cilindro externo o de tener cierta desviacioń máxima en la diferencia para los centros en X e Y con respecto a un cilindro del otro.
- **Transfer learning**: Finalmente, una posible técnica que impactaría en el nivel predictivo de la red es la de tomar de línea base modelos pre entrenados correspondientes a problemáticas similares, en este caso se debería poder consumir modelos que tenga afinidad a la hora de identificar formas geométricas en imágenes de escala de grises en un formato de resolución baja para que la red pueda, sobre esta base, realizar un entrenamiento con recursos neuronales destinados específicamente al problema.

# Bibliografía

[1] M. Pastorino, Microwave imaging. John Wiley & Sons, 2010.

[2] X. Chen, Computational methods for electromagnetic inverse scattering. John Wiley & Sons, 2018.

[3] J. E. Fajardo, J. Galvan, F. Vericat, C. M. Carlevaro, and R. M. Irastorza, "Phaseless microwave imaging of dielectric cylinders: An artificial neural networks-based approach." Progress In Electromagnetics Research, vol. 166, pp. 95–106, 2019.

[4] A. F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J.D. Joannopoulos, S. G. Johnson, "Meep: A flexible free-software package for electromagnetic simulations by the fdtd method" Computer Physics Communications, vol. 181, no. 3, pp. 687–702, 2010.

https://blog.roboflow.com/what-is-a-convolutional-neural-network/#:~:text=The%20activation%20function%20is%20typically,Rectified%20linear%20unit%20(ReLU)

[5] Diederik P. Kingma, Jimmy Ba, "Adam: A Method for Stochastic Optimization", Machine Learning (cs.LG). Cap. 6.3, 2015 https://arxiv.org/abs/1412.6980

[6] McCulloch, Warren; Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics, 1943.

[7] Cornell Aeronautical Laboratory. The Perceptron: A Perceiving and Recognizing Automaton (Project PARA), 1957.

https://blogs.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf

[8] Li Deng, Artificial Intelligence in the Rising Wave of Deep Learning: The Historical Path and Future Outlook [Perspectives], Artificial Intelligence, Citadel, United States, 2017.

[9] Chris M. Bishop, Neural networks and their applications, Volume 65, Issue 6, 1994.

[10] Nikhil Bhargava, student of Master of Technology, IIT Delhi, Manik Gupta, student of Master of Technology, IIT Delhi, Application of Artificial Neural Networks in Business Applications, 2008. https://www.geocities.ws/nikhil_bhargav/papers/lahore.pdf

[11] GMG - UTN FRLP, Redes neuronales convolucionales en Python con Keras, 2020.

https://github.com/gmg-utn/machine-learning/blob/master/encuentro_03/Convolucional_encuentro_3.ipynb

[12] Serguei Semenov, Microwave tomography: review of the progress towards clinical applications, 2009. https://royalsocietypublishing.org/doi/full/10.1098/rsta.2009.0092

[13] Richard Nagyfi, The differences between Artificial and Biological Neural Networks, 2018.
https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks-a8b46db828b7