UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE MÁSTER

Simulación de emisiones anisotrópicas de rayos gamma en campos magnéticos y redes neuronales

Simulation of gamma ray anisotropic emissions in magnetic fields and neural networks

---

Autor: Iñaki GARCÍA DEL POZO

Supervisores: Dr. Joaquín LÓPEZ HERRÁIZ,
Dr. José Manuel UDIAS MOINELO

Un trabajo presentado en la convocatoria de junio, en cumplimiento de los requisitos para el grado de Máster interuniversitario en Física Nuclear

en el

Grupo de Física Nuclear
Departamento de Estructura de la Materia, Física Térmica y Electrónica

29 de junio de 2021

UNIVERSIDAD
COMPLUTENSE
MADRID

ANEXO I: DECLARACIÓN DE NO PLAGIO

D./Dña. Iñaki García del Pozo con NIF 53984680T, estudiante de Máster en la Facultad de Ciencias Físicas de la Universidad Complutense de Madrid en el curso 2020-2021, como autor/a del trabajo de fin de máster titulado Simulación de emisiones anisotrópicas de rayos gamma en campos magnéticos y redes neuronales y presentado para la obtención del título correspondiente, cuyo/s tutor/ es/son:
Joaquín López Herraiz

DECLARO QUE:

El trabajo de fin de máster que presento está elaborado por mí y es original. No copio, ni utilizo ideas, formulaciones, citas integrales e ilustraciones de cualquier obra, artículo, memoria, o documento (en versión impresa o electrónica), sin mencionar de forma clara y estricta su origen, tanto en el cuerpo del texto como en la bibliografía. Así mismo declaro que los datos son veraces y que no he hecho uso de información no autorizada de cualquier fuente escrita de otra persona o de cualquier otra fuente.
De igual manera, soy plenamente consciente de que el hecho de no respetar estos extremos es objeto de sanciones universitarias y/o de otro orden.

En Madrid, a 17 de Noviembre de 2020

Fdo.: IÑAKI GARCIA

Esta DECLARACIÓN debe ser insertada en primera página de todos los trabajos fin de máster conducentes a la obtención del Título.

«Si el hombre occidental arroja su máscara, si renuncia a ser personaje en la historia, quedará disponible para elegirse como persona. Y no es posible elegirse a sí mismo como persona sin elegir, al mismo tiempo, a los demás. Y los demás son todos los hombres.»

«Con ello no se acaba el camino; más bien empieza.»

María Zambrano

Agradecimientos

En primer lugar, quiero agradecer una vez más a Joaquín por aceptarme de nuevo como su tutorando. Gracias por todo el tiempo que me has dedicado, por todas las dudas que me has resuelto y por todas las cosas que me has enseñado, a pesar del poco tiempo del que dispones.

Muchas gracias a mi familia, sobre todo a mis padres, por apoyarme siempre. Gracias por todo el esfuerzo que habéis hecho, para que hoy pueda presentar este trabajo. Gracias también a mis amigos, por hacer que este complicado año sea más llevadero, y por haberme servido como lugar en el que desconectar.

Por último, gracias al Instituto de Física de Partículas y del Cosmos, IPARCOS, por la beca concedida para la realización de este Trabajo de Fin de Máster.

UNIVERSIDAD COMPLUTENSE DE MADRID

Resumen

Facultad de Ciencias Físicas

Departamento de Estructura de la Materia, Física Térmica y Electrónica

Máster interuniversitario en Física Nuclear

Simulación de emisiones anisotrópicas de rayos gamma en campos magnéticos y redes neuronales

por Iñaki GARCÍA DEL POZO

En este trabajo se estudia una novedosa modalidad de imagen médica, Gamma-MRI, que combina la tradicional resonancia magnética con la medicina nuclear para obtener mejor sensibilidad en la detección de la señal que con técnicas tradicionales. Esta se basa en la emisión anisotrópica de rayos γ de núcleos excitados con espín $> 1/2$ hiperpolarizados en presencia de campos magnéticos. En el trabajo que inspiró esta nueva modalidad, toman un promedio temporal para el modelado de la precesión del espín bajo campos magnéticos. En este trabajo se estudia la posibilidad de prescindir el promedio temporal y así almacenar más información sobre la señal medida. Para ello se han realizado simulaciones de la emisión anisotrópica de rayos γ por núcleos con espín $> 1/2$. También se ha realizado con éxito la reconstrucción de 2 sencillas distribuciones de actividad con el algoritmo iterativo ML-EM, a partir de los datos proporcionados por la simulación. Por último, se estudia la posibilidad de realizar dicha reconstrucción de la imagen con una red neuronal. Los resultados de este trabajo pueden servir de guía para el diseño de la adquisición y codificación de datos en el proyecto Gamma-MRI.

Palabras clave—PNI, MRI, rayos gamma, polarización, anisotropía, simulación, reconstrucción, red neuronal, actividad

UNIVERSIDAD COMPLUTENSE DE MADRID

Abstract

Facultad de Ciencias Físicas

Departamento de Estructura de la Materia, Física Térmica y Electrónica

Máster interuniversitario en Física Nuclear

Simulation of gamma ray anisotropic emissions in magnetic fields and neural networks

by Iñaki GARCÍA DEL POZO

For this project, a novel medical imaging modality is explored. This new method combines traditional magnetic resonance with nuclear medicine to obtain a higher sensitivity in signal detection respect traditional techniques. It is based on the anisotropic γ ray emission of excited hyperpolarized nuclei with spin $> 1/2$ in the presence of magnetic fields. In the paper that inspired this new modality, they use time averaged data to model the spin precession in a magnetic field. In this project, we study the viability of storing all of the signal's data without calculating the average values over time. For this purpose, simulations of the anisotropic emission of γ rays by nuclei with spin $> 1/2$ have been carried out. The reconstruction of 2 simple activity distributions has also been successfully achieved with the iterative ML-EM algorithm, from the data provided by the simulation. Finally, the possibility of carrying out the image reconstruction with a neural network is explored and discussed. The results of this work can serve as a guide for the design of data acquisition and codification in the Gamma-MRI project.

Keywords—PNI, MRI, gamma rays, polarization, anisotropy, simulation, reconstruction, neural network, activity

Índice general

Índice de figuras

Siglas

API Interfaz de Programación de Aplicaciones (*Application Programming Interface*). 15

CNN Red Neuronal Convolucional (*Convolutional Neural Network*). XII, XIII, XVI, 18, 19, 20, 21, 23, 36, 37, 39, 40, 41, 43, 52, 53, 55, 58, 67, 81, 82, 83, 84, 85, 86

CPU Unidad Central de Procesamiento (*Central Processing Unit*). 18, 58

CT Tomografía Axial Computerizada (*Computed Tomography*). 2, 3, 4

GPU Unidad Gráfica de Procesamiento (*Graphics Proccesing Unit*). 14, 15, 18, 37, 41, 53, 58, 59

HP Hiperpolarizado (*Hyperpolarized*). 9, 10

MRI Imagen por Resonancia Magnética (*Magnetic Resonance Imaging*). VII, IX, XV, 1, 2, 3, 4, 5, 8, 9, 10, 12, 59

NMR Resonancia Magnética Nuclear (*Nuclear Magnetic Resonance*). XI, 4, 5, 8, 10, 30

PDF Función de Densidad de Probabilidad (*Probability Density Funcion*). XVI, 27, 28, 44, 45, 46

PET Tomografía por Emisión de Positrones (*Positron Emission Tomography*). 2, 3, 4, 37, 57

PNI Imagen por Polarización Nuclear (*Polarized Nuclear Imaging*). VII, IX, XI, 1, 10, 11, 12, 23

RF Radiofrecuencia (*Radiofrequency*). 8, 10, 59

SEOP Bombeo Óptico de Intercambio de Espín (*Spin Exchange Optical Pumping*). 9, 58

Capítulo 1

Introducción

Este proyecto está inspirado en un artículo publicado en Nature por un equipo de la Universidad de Virginia en 2016 (Zheng y col., 2016), que sigue el trabajo hecho en una tesis doctoral en el año 2014 por el mismo autor (Zheng, 2014). En este artículo se describe una nueva técnica de obtención de imagen médica, y le dan el nombre de Imagen por Polarización Nuclear (*Polarized Nuclear Imaging*) (PNI). La idea principal de esta nueva técnica de imagen es la de combinar la tradicional Imagen por Resonancia Magnética (*Magnetic Resonance Imaging*) (MRI) con la medicina nuclear, para así obtener una herramienta de obtención de imágenes todavía mejor. La MRI proporciona buena resolución espacial, sensibilidad espectral y una abundante variedad de mecanismos de contraste para el diagnóstico médico (Brown y col., 2014). La imagen nuclear usando cámaras de rayos γ tiene el beneficio del uso de pequeñas cantidades de trazadores radiactivos que buscan objetivos de interés específico dentro del cuerpo (Cherry, Sorenson y Phelps, 2012). En el artículo mencionado se describe una nueva modalidad de imagen y de espectroscopía que combina aspectos favorables de ambos enfoques. La ventaja de usar radiación (rayos γ en este caso) es que su detección es relativamente más sencilla en comparación con la señal de MRI. Por lo tanto, combinando ambas técnicas, podemos obtener mayor sensibilidad en el origen de la señal. La imagen que se muestra en la figura 1.1 es la primera imagen publicada usando PNI.

El presente trabajo se engloba dentro del proyecto Gamma-MRI (sección 1.5), que busca diseñar y construir el primer prototipo preclínico de un escáner que utilice la revolucionaria técnica de PNI. En el trabajo se busca generar simulaciones realistas y métodos avanzados de reconstrucción de la señal (tanto con algoritmos clásicos como con redes neuronales) para esta nueva técnica de imagen. Las simulaciones trataran de modelizar la emisión anisotrópica de rayos γ del Xenón metaestable y su interacción con campos magnéticos externos. Se pretende sentar las bases de estas simulaciones, para que en un futuro permitan guiar los montajes experimentales del

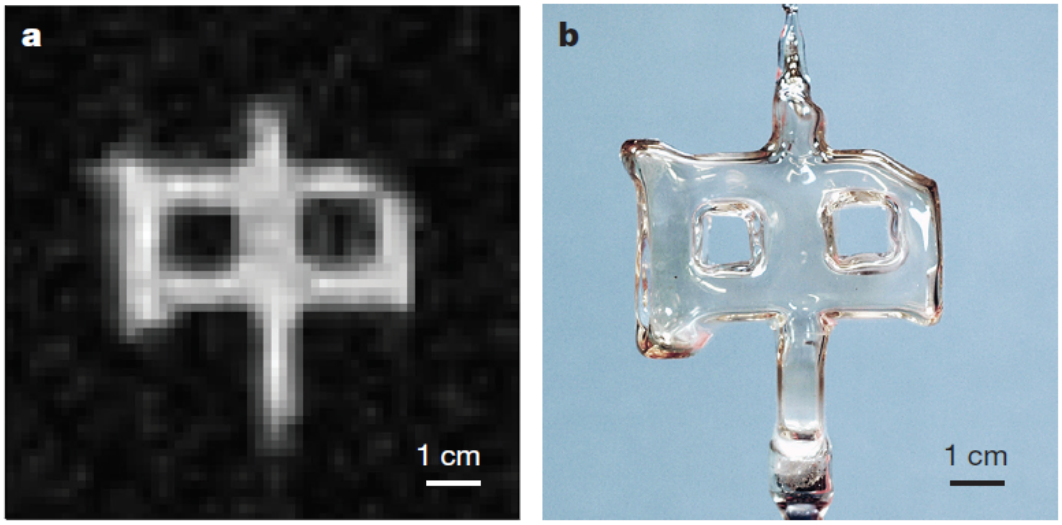


FIGURA 1.1: **Izquierda:** Imagen de aproximadamente 1 mCi de ^{131m}Xe obtenida combinando técnicas de resonancia magnética con la detección de rayos γ. **Derecha:** Fotografía de la celda de vidrio en la que estaba contenida la muestra de ^{131m}Xe (Zheng y col., 2016).

proyecto Gamma-MRI. También, una parte importante del trabajo es utilizar estas simulaciones para entrenar una red neuronal que sea capaz de reconstruir la imagen a partir de las medidas.

1.1. Principales técnicas de imagen

En el campo de la imagen médica, las máquinas de rayos-X, la Tomografía Axial Computerizada (*Computed Tomography*) (CT), la Tomografía por Emisión de Positrones (*Positron Emission Tomography*) (PET), los ultrasonidos y la MRI son las tecnologías más utilizadas para el diagnóstico (Dowsett, Kenny y Johnston, 2006). En esta sección haremos una breve explicación de las técnicas mencionadas, excepto de MRI, que le dedicaremos la sección 1.2 completa, para explicarla con mayor detalle.

Las radiografías por rayos-X, obtenidas mediante la transmisión de radiación ionizante, son adquisiciones bidimensionales de la anatomía tridimensional del paciente. Esto implica una compresión de la información, lo cual puede suponer una desventaja, ya que no se conoce la posición a lo largo de la trayectoria del haz de rayos X. Una sola radiografía de tórax puede revelar información importante para el diagnóstico sobre los pulmones, la columna, las costillas y el corazón (Dowsett, Kenny y Johnston, 2006). Para obtener una radiografía, en primer lugar, los rayos-X salen de un tubo de rayos-X colocado a un lado del cuerpo del paciente. Estos rayos-X atraviesan e interactúan con el cuerpo del paciente para ser detectados al otro lado del cuerpo. Los huesos y otros tejidos densos frenarán la mayor parte de la

radiación, y por tanto, aparecen de color blanquecino, mientras que la radiación que atraviesa tejidos blandos resulta en una apariencia grisácea (Chen, Rogalski y Anker, 2012). La imagen por rayos-X es una tecnología rápida y barata que suele utilizarse para observar posibles fracturas en los huesos.

La tomografía computarizada (CT) es una técnica de imagen no invasiva que usa también radiación ionizante. A diferencia de un escáner de rayos-X, un escáner CT proporciona una imagen volumétrica (3D) del cuerpo del paciente. Esto es debido a que produce las imágenes como secciones axiales del cuerpo, sin interferencia entre secciones ni emborronamiento (Kissane, Neutze y Singh, 2020). La imagen se obtiene utilizando un tubo de rayos-X giratorio y una serie de detectores que servirán para medir las atenuaciones de los rayos-X en diferentes tejidos del interior del cuerpo del paciente. Estás medidas de rayos-X tomadas desde diferentes ángulos se procesan en una computadora usando algoritmos de reconstrucción para producir imágenes tomográficas transversales, es decir, "cortes" virtuales del cuerpo. Este tipo de escáner es muy común, debido a que son rápidos y ofrecen alta precisión anatómica. Uno de los mayores problemas a los que se enfrenta la técnica de escáner CT es la dosis de radiación ionizante que recibe el paciente durante la toma de una imagen, la cual es proporcional al área de la parte del cuerpo que está siendo escaneada. Es por eso que en los últimos años se están haciendo grandes esfuerzos para reducir la cantidad de dosis que recibe el paciente, utilizando por ejemplo, técnicas de inteligencia artificial para la reconstrucción de la imagen (Willemink y Noël, 2019) a partir de una dosis de radiación menor.

PET es una técnica que proporciona información cuantitativa de la distribución de radiotrazadores suministrados al paciente, lo que nos permite obtener información de procesos fisiológicos en el cuerpo. Un escáner PET implica la inyección intravenosa de una molécula biológicamente activa, marcada con un radionucleido emisor de positrones. El positrón emitido dentro del cuerpo del paciente recorre una corta distancia en el tejido antes de aniquilarse con un electrón del cuerpo del paciente. Esta aniquilación produce dos fotones de 511 keV, los cuales forman un ángulo de 180 grados. Estos fotones son los que llegan a los detectores (dispuestos en forma de anillo alrededor del paciente), que tiene una unidad de procesamiento para seleccionar solo los pares de fotones "en coincidencia", es decir, detectados al mismo tiempo, en la misma dirección y en sentidos opuestos. Como los escáner PET proporcionan, principalmente, información fisiológica, se suelen combinar, ya sea con MRI o con CT para así obtener también información anatómica (Vallabhajosula, 2009).

La imagen por ultrasonidos, más conocida con el nombre de ecografía en España, es la tecnología de imagen médica más común, después de las radiografías por rayos-X (Wells y Liang, 2011). Tiene algunas ventajas respecto a otras técnicas de imagen médica, como por ejemplo, que son imágenes generadas a tiempo real, que las exposiciones utilizadas en la práctica se consideran seguras y que el equipo es generalmente menos costoso que el de otras tecnologías de obtención de imagen. Los ultrasonidos son ondas sonoras con frecuencias superiores a las audibles para los humanos ($>$ 20000 Hz). Las imágenes se obtienen enviando pulsos de ultrasonido al cuerpo del paciente, los cuales hacen eco en los tejidos del cuerpo con diferentes propiedades de reflexión y posteriormente son registrados para reconstruir la imagen. La técnica de imagen por ultrasonidos es ampliamente usada en el campo de la medicina debido a que son dispositivos relativamente baratos y que pueden ser portátiles debido a su pequeño tamaño.

En resumen, los ultrasonidos y la CT no dan información funcional. Las técnicas de medicina nuclear presentan el reto de la dosis, que para que la dosis sea segura y la mínima necesaria, exige tener la máxima sensibilidad posible y detectar la mayor cantidad de radiación emitida posible. En el caso del PET se logra hacer imagen aprovechando que los 2 fotones salen a 180 grados. En Gamma-MRI se busca un nuevo mecanismo para obtener imagen sin sacrificar sensibilidad.

1.2. Imagen por resonancia magnética (MRI)

La MRI es una técnica de obtención de imagen no invasiva y se basa en el fenómeno de la Resonancia Magnética Nuclear (*Nuclear Magnetic Resonance*) (NMR) que se beneficia de las propiedades de los núcleos atómicos (especialmente del ^{1}H presente en las moléculas de agua que hay en el interior del cuerpo humano) en un campo magnético. Fue introducida en los años 70 (Lauterbur, 1973), y es de vital importancia en la medicina actual. Las técnicas actuales de MRI pueden mostrar (Dowsett, Kenny y Johnston, 2006):

- **Diferencias químicas** entre tejidos, que dan lugar a cambios en una imagen en escala de grises (patología tumoral).
- **Flujo sanguíneo** como una imagen de alta intensidad de los vasos sanguíneos, tanto en pequeñas rodajas como en imágenes 3D.
- Imágenes **axiales**, **coronales**, **sagitales** y **oblicuas** de un volumen 3D (por ejemplo, la cabeza).

- **Cortes largos**, especialmente en vista sagital (utilizado, por ejemplo, para obtener imágenes de la columna vertebral).

Una parte fundamental en un escáner de MRI es el imán principal, el cual, produciendo un campo magnético permanente y estable, alinea los momentos magnéticos de espín de un número significativo de núcleos de hidrógeno de los tejidos del cuerpo. Después, se aplican campos magnéticos más débiles para inducir un cambio en la precesión de los núcleos, que da lugar a una señal electromagnética medible.

Recordemos que la NMR es un fenómeno físico en el cual los núcleos atómicos sometidos a un campo magnético fuerte y constante, son perturbados por un campo magnético más débil. Esta perturbación hace que los núcleos produzcan una señal electromagnética con una frecuencia característica que depende del campo magnético en el núcleo.

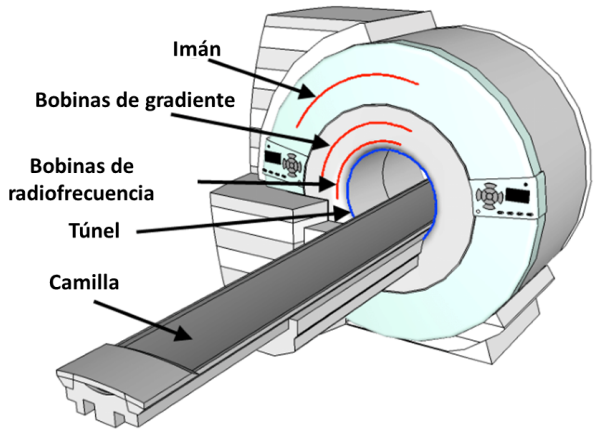


FIGURA 1.2: Esquema de un escáner de MRI estándar (modificado de Haynes, 2013).

Lo que hace especial a la técnica de MRI es que consigue localizar con alta precisión la fuente de la señal de la NMR gracias a la aplicación de campos magnéticos no uniformes, llamados gradientes, los cuales generan frecuencias de precesión ligeramente distintas dependiendo de la posición de los núcleos atómicos en estos gradientes. En la figura 1.2 se muestra un esquema de un escáner de MRI estándar.

1.2.1. Resonancia Magnética Nuclear (NMR)

La teoría de esta sección se ha extraído del libro de Levitt (Levitt, 2008) y de la tesis de Wiström (Wiström, 2020). El espín y el magnetismo están estrechamente

relacionados. Un teorema de simetría muy fundamental establece que el momento angular de espín y el momento magnético son proporcionales entre si:

$$\hat{\mu} = \gamma \hat{S} \tag{1.1}$$

donde los "gorros" encima de los símbolos indican que se tratan de operadores cuánticos. Para núcleos atómicos, la constante de proporcionalidad γ recibe el nombre de *relación giromagnética*, y se suele dar en unidades de $\mathrm{rad\,s^{-1}\,T^{-1}}$. La relación giromagnética puede tener signo tanto positivo como negativo. Si el valor es positivo, indica que el momento magnético es paralelo al momento angular de espín, en cambio, si es negativo el momento magnético tiene sentido opuesto al momento angular de espín.

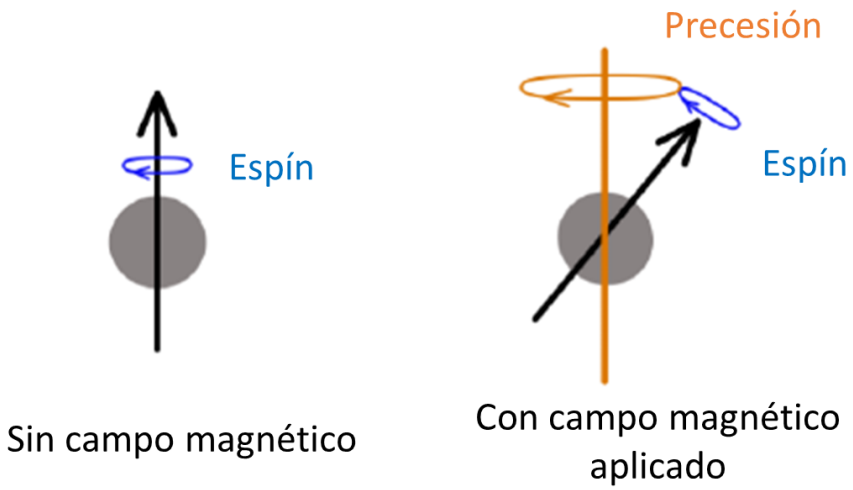


FIGURA 1.3: Precesión de un núcleo con espín no nulo en un campo magnético externo (modificado de Wiström, 2020).

Un núcleo con momento angular de espín bajo un campo magnético externo sufre el fenómeno de la precesión de Larmor, donde el vector del momento angular precesa alrededor del eje del campo magnético externo, tal y como se ilustra en la figura 1.3, con una frecuencia angular que recibe el nombre de *frecuencia de Larmor*, ω_0:

$$\omega_0 = \gamma \cdot B_0 \tag{1.2}$$

donde B_0 es el campo magnético externo que hay en el lugar de la partícula y γ es la relación giromagnética. La frecuencia de Larmor es proporcional al campo magnético externo. Este fenómeno se puede explicar desde el punto de vista microscópico, ya que colocar un núcleo, que tenga espín nuclear no nulo, en un campo magnético externo divide los niveles de energía de forma proporcional a la fuerza del campo magnético, tal y como se ilustra en la figura 1.4

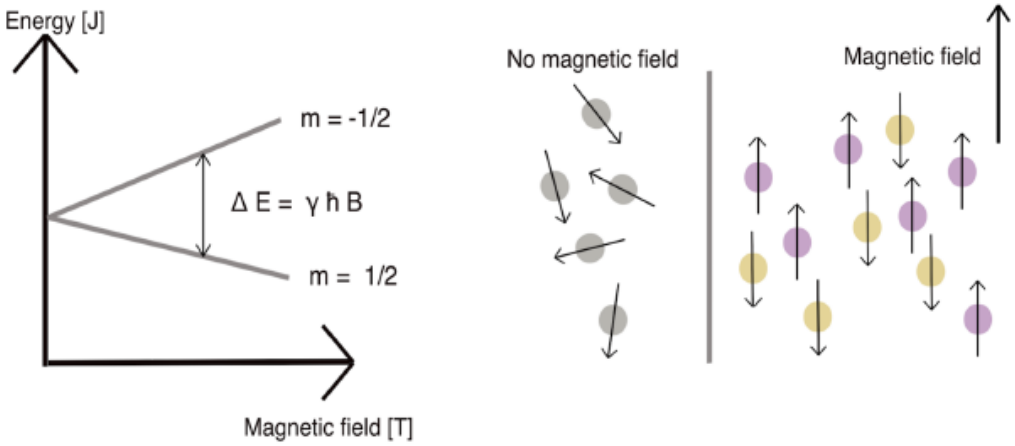


FIGURA 1.4: Efecto de la aplicación de un campo magnético estático a un núcleo con espín $1/2\ \hbar$ (con sus dos estados de espín m) y un momento magnético positivo. **Izquierda:** Separación de energía definida por ΔE de la ecuación 1.3 con la relación giromagnética γ, la constante de Planck $\hbar$, y el campo magnético B_0. **Derecha:** Núcleos alineados paralela y antiparalelamente con el campo magnético externo (Wiström, 2020).

Un espín de $1/2\ \hbar$ tiene dos estados, el *espín arriba* (o *spin up* en inglés) con $m = 1/2\ \hbar$ en la figura 1.4, y el *espín abajo* (o *spin down* en inglés) con $m = -1/2\ \hbar$ en la figura 1.4, los cuales se encuentran alineados paralela y antiparalelamente al eje del campo magnético como se muestra en la figura 1.4. La diferencia de energía ΔE de los estados nucleares de espín viene dada por la expresión:

$$\Delta E = \gamma \hbar B_0 \tag{1.3}$$

donde γ es la relación giromagnética, $\hbar$ es la constante de Planck, y B_0 es el campo magnético estático. La ocupación de los estados espín arriba y espín abajo en equilibrio térmico viene dado, de acuerdo con la distribución de Boltzmann, por:

$$\frac{N_m}{N_0} = \frac{e^{-E_m/(k_B T)}}{\sum_{n=-I}^{n=I} e^{-E_n/(k_B T)}} \tag{1.4}$$

donde m es el subnivel nuclear, el número de espines ocupando cada estado es N_m, N_0 es el número total de espines, E_m es la energía de cada estado, I es el espín nuclear total, T es la temperatura absoluta y k_B es la constante de Boltzmann. La suma del momento magnético μ de todos los espines nucleares recibe el nombre de magnetización total, M. La dirección de la magnetización total será la misma que la del campo magnético, que estableceremos en el eje z sin pérdida de generalidad. Así

pues, aplicando la aproximación $e^x \approx 1 + x$ en la ecuación 1.4 obtenemos la siguiente expresión para M_z:

$$M_z = \frac{N_0 \gamma^2 \hbar^2 I(I+1)}{3 k_B T} B_0 \tag{1.5}$$

donde N_0 es el número total de espines, γ es la relación giromagnética, I es el espín nuclear total, $\hbar$ es la constante de Planck, k_B es la constante de Boltzmann, T es la temperatura absoluta, y B_0 es el campo magnético. La magnetización total M_z, rotará alrededor del eje del campo magnético debido a las propiedades de los núcleos individuales en un campo magnético externo, tal y como se ilustra en la figura 1.3. Es este el fenómeno físico que se conoce con el nombre de precesión de Larmor. La magnetización neta puede salirse de su precesión si un campo magnético secundario gira con una frecuencia cercana a la frecuencia de Larmor, ω_0 (de la ecuación 1.2), de los núcleos. La manipulación de ω_0 es un proceso rudimentario en NMR y MRI.

1.2.2. Gradientes magnéticos y pulsos de radiofrecuencia

En un escáner de MRI los núcleos de hidrógeno de las moléculas de agua del cuerpo humano se ven afectados por el campo magnético estático, por los gradientes magnéticos aplicados y por pulsos de Radiofrecuencia (*Radiofrequency*) (RF). El imán principal genera un campo magnético constante y uniforme, por lo que todos los núcleos que posean el mismo momento magnético (por ejemplo, todos los núcleos de hidrógeno) tendrán la misma frecuencia de resonancia. En consecuencia, la señal electromagnética, ocasionada por la NMR, será detectada con el mismo valor desde todas las partes del cuerpo, por lo que, haciendo uso solo del imán principal no hay manera de obtener información espacial de la resonancia. Esto se soluciona introduciendo las llamadas bobinas de gradiente, las cuales generan cada una un gradiente de campo magnético de diferente intensidad en el espacio, modificando el campo magnético ya presente y haciendo que varíe con la posición. Esto quiere decir que todos los núcleos con espín no nulo precesan con frecuencias ligeramente distintas, de acuerdo con la ecuación 1.2. Además de los gradientes magnéticos, se aplican pulsos de RF con un rango determinado de frecuencias. Dicho rango de frecuencias determina la anchura de la rodaja excitada. Los pulsos de RF son producidos por campos magnéticos de corta vida inducidos por los cables de transmitancia de la figura 1.2, y se utilizan para cambiar la dirección del vector de magnetización neta $\boldsymbol{M}$. Para la reconstrucción de la imagen, la localización de la señal se define por cuatro factores: la fase, la frecuencia, la distancia y el tiempo de la señal.

1.3. Gases polarizados

Una nueva rama de MRI haciendo uso de gas Hiperpolarizado (*Hyperpolarized*) (HP) fue desarrollada en el año 1994 (Albert y col., 1994), y ha demostrado ser particularmente útil para obtener imágenes de los pulmones. A diferencia de la MRI convencional, que utiliza la polarización térmica de protones, esta nueva técnica hace uso de gases nobles previamente polarizados. Los gases nobles que se suelen emplear suelen ser ^{3}He o ^{129}Xe, y son polarizados por la técnica de Bombeo Óptico de Intercambio de Espín (*Spin Exchange Optical Pumping*) (SEOP), utilizada ya durante varias décadas (Walker y Happer, 1997; Appelt y col., 1998). Una vez que se han inhalado por el paciente, la magnetización de los gases nobles, debida a la HP, se puede utilizar para adquirir la imagen. A pesar de que, a diferencia del hidrógeno utilizado en la MRI convencional, la densidad del gas en el cuerpo es muy pequeña se puede obtener una buena visualización de la imagen, gracias a que la polarización de los gases nobles polarizados es 5 órdenes de magnitud mayor que la polarización térmica del agua.

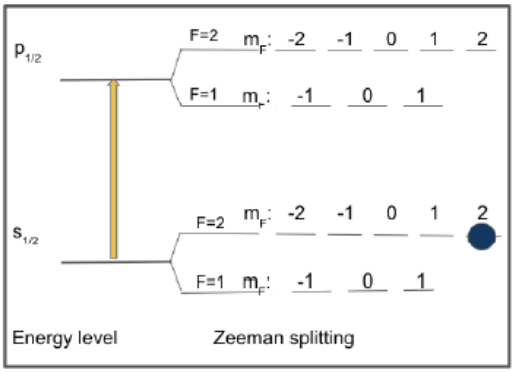


FIGURA 1.5: El electrón más externo de ^{85}Rb con espín nuclear $3/2\ \hbar$. El nivel $s_{1/2}$ es el estado fundamental, y el nivel $p_{1/2}$ es el estado excitado. El electrón se encuentra en el nivel de mayor energía del estado fundamental (Wiström, 2020).

La técnica de SEOP consiste en polarizar un electrón de un átomo de un metal alcalino y transferir esta polarización al núcleo de un átomo de gas noble. El SEOP se realiza bajo un campo magnético, por lo que los niveles de Zeeman del metal alcalino se encuentran disponibles. El electrón más externo del átomo del metal alcalino se excita al valor de espín más alto posible en su estado fundamental, tal y como se ilustra en la figura 1.5.

Una de las ventajas de la técnica de MRI con gases HPs recae en que, en principio, es posible sustituir los imanes superconductores masivos que se suelen usar en los escáneres comerciales con simples solenoides. Esto es debido a que la polarización de los gases nobles es independiente de la fuerza del campo magnético

aplicado B_0, lo que permite llevar a cabo un escáner MRI con un campo magnético mucho más pequeño (mT en vez de T). El uso de ^{129}Xe es particularmente interesante ya que el Xenón es una sustancia lipofílica y es bien conocido por poder disolverse en la sangre. Por lo tanto, en principio, el ^{129}Xe HP puede llegar a ser utilizado para obtener imágenes de partes del cuerpo distintas a los pulmones.

1.4. Imagen por Polarización Nuclear (PNI)

En esta sección describiremos el concepto de la PNI, una novedosa modalidad de obtención de imagen en la que se busca combinar principios relacionados con la NMR con la detección de radiación. Básicamente, algunos núcleos radiactivos muestran una asimetría de emisión de radiación cuando se encuentran polarizados. Se manipulan los espines nucleares polarizados usando técnicas de NMR, como se ha hecho tradicionalmente en MRI, pero se detecta la dinámica del espín mediante rayos γ, en lugar de mediante ondas electromagnéticas de RF. La cantidad de núcleos radiactivos involucrados es suficientemente pequeña como para que puedan polarizarse fácilmente con un grado alto de polarización. Además, debido a las pequeñas cantidades involucradas, los isótopos radiactivos pueden introducirse fácilmente en cualquier tejido y usarse para sondear una variedad de procesos fisiológicos (Zheng, 2014). La motivación de esta nueva técnica radica en que un escáner MRI ofrece muy buena resolución espacial, pero está condicionado por una baja sensibilidad en la detección de la señal. Es por eso por lo que se combina con la detección de rayos γ, ya que ofrece buena sensibilidad en la detección de la señal, ayudando a paliar los defectos que tiene la MRI.

Se sabe desde hace muchos años que ciertos núcleos radiactivos muestran una anisotropía espacial en la emisión de radiación. La dependencia angular de las emisiones gamma de núcleos polarizados se puede describir mediante la siguiente expresión (Steenberg, 1952):

$$W(\theta) = a_0 + a_2 \cos(2\theta) + a_4 \cos(4\theta) + a_8 \cos(8\theta) \tag{1.6}$$

donde $W(\theta)$ es la probabilidad relativa de que se emita un rayo γ dado en un ángulo θ con respecto a la dirección de orientación. Los valores de los coeficientes a_n dependen del isótopo, de la transición γ y del grado de polarización nuclear. En la figura 1.6 se muestra, en coordenadas polares, la ecuación 1.6 para la transición γ de 164 keV del ^{131m}Xe, para grados de polarización del 0 %, del 70 %, y del 100 %.

Hay que señalar que solamente los núcleos radiactivos con espín $> 1/2$ muestran la emisión anisotrópica de rayos γ.

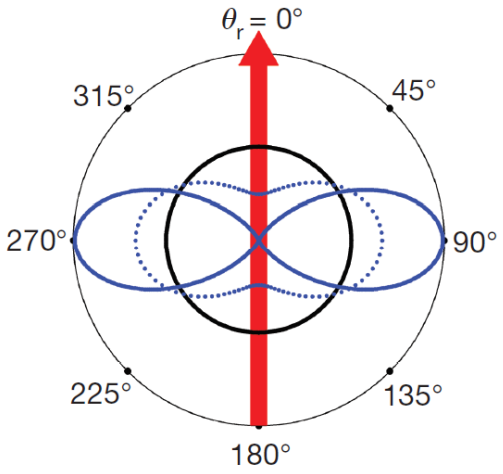


FIGURA 1.6: Probabilidad de emisión direccional (ecuación 1.6) en coordenadas polares como función del ángulo θ con respecto a la dirección de orientación (flecha roja), para un núcleo de ^{131m}Xe con grados de polarización del 0 % (línea negra), del 70 % (línea azul a puntos), y del 100 % (línea azul sólida) (Zheng y col., 2016).

Cuando el núcleo de ^{131m}Xe está totalmente despolarizado, el único coeficiente de la ecuación 1.6 distinto de cero es a_0, lo que da lugar a una distribución de emisión γ que no depende del ángulo, como cabría esperar. A medida que aumenta la polarización, aumenta también el coeficiente a_2, lo que dará lugar a cierto grado de anisotropía. Para una polarización mayor del 70 %, a_4 también comienza a aumentar, lo que complica la distribución angular. Por suerte, a_6 y a_8 se mantienen pequeños para todas las polarizaciones y, por lo general, pueden ser ignorados. Para la ecuación 1.6 podemos tomar la siguiente aproximación:

$$W(\theta) = a_0 - a_2 \cos(2\theta), \quad \text{donde: } a_2 > 0 \tag{1.7}$$

1.5. Proyecto Gamma-MRI

Gamma-MRI es un proyecto de investigación europeo, que cuenta con la colaboración de varias universidades y que está financiado por el programa *FET Open* de la unión europea. El proyecto busca realizar, por primera vez, un prototipo de escáner preclínico que utilice la novedosa técnica de PNI (sección 1.4). Con esta revolucionaria modalidad de imagen, se pretende formar la base para los estudios del cerebro humano en el futuro. Se pretende crear las últimas novedades en varios aspectos tecnológicos. Los objetivos principales del proyecto son:

- Producir e hiperpolarizar de manera eficiente varios isótopos radiactivos del Xenón con propiedades optimizadas.
- Utilizar estrategias de encapsulación biocompatibles para mantener la polarización del Xenón *in vivo* durante el orden de varios segundos, para que la muestra de Xenón pueda alcanzar el órgano objetivo desde el lugar de administración.
- Desarrollar técnicas de reconstrucción de imagen basadas en MRI rápidas y con estrategias emergentes basadas en inteligencia artificial, para poder obtener la imagen del escáner en minutos.
- Desarrollar detectores y componentes electrónicos compactos, rápidos, de alto rendimiento, y compatibles con campos magnéticos.
- Construir un prototipo preclínico para la demostración *in vitro* e *in vivo* de la técnica de PNI.
- Obtener la primera imagen de PNI *in vivo* del cerebro de un roedor utilizando el prototipo diseñado y el Xenón hiperpolarizado.
- Explorar otros isótopos, además del Xenón, que puedan ser médicamente útiles y factibles para esta técnica.

Para alcanzar estos objetivos hace falta superar ciertas barreras tecnológicas:

- Hiperpolarizar el Xenón (y más tarde otros radiotrazadores) usando bombeo óptico de intercambio directo o de espín, pero también estudiar la técnica de Polarización Nuclear Dinámica (Abragam y Goldman, 1978).
- Almacenar y transportar los radiotrazadores, emisores de rayos γ, hiperpolarizados.
- Disponer de detectores de radiación gamma rápidos, compactos, de alto contraste y compatibles con campos magnéticos.
- Diseñar una estrategia de adquisición de datos eficiente para limitar el tiempo de adquisición a un máximo del orden de varios minutos.
- Encapsular los radiotrazadores en construcciones supramoleculares biocompatibles para preservar la hiperpolarización por más tiempo, para permitir llegar a un órgano objetivo distante, y para, eventualmente, incluir la capacidad de servir como biosensor.

1.6. Redes neuronales

1.6.1. Machine Learning

A pesar de que hoy en día la inteligencia artificial (*artificial intelligence* o AI en inglés) es un tema muy actual, es importante señalar que no es un concepto nuevo. Las ideas más básicas relacionadas con la inteligencia artificial se remontan al siglo IV a.C., cuando Aristóteles explicó un conjunto de reglas que describían una parte del funcionamiento de la mente para obtener conclusiones racionales. En el año 1950, Alan Turing propuso el conocido "Test de Turing", y ese mismo año Isaac Asimov propuso las 3 leyes de la robótica (Asimov, 1941). En 1951 se desarrolló el primer programa informático basado en inteligencia artificial, y 8 años después, en 1959, se fundó el laboratorio de IA del MIT (*MIT AI Lab*). En el año 1961, en la línea de montaje de la empresa estadounidense GM (*General Motors*), se introdujo el primer robot. En el año 1965, se inventó "ELIZA" el primer *chat-bot*, y casi diez años después, en 1974, se creó el primer vehículo autónomo en el laboratorio de inteligencia artificial de la Universidad de Standford. En 1989, se creó el primer vehículo autónomo que hacía uso de una red neuronal. Uno de los hitos más importantes que se han conseguido en la historia de la inteligencia artificial, se dio en el año 1997, cuando el algoritmo *Deep Blue* desarrollado por la empresa IBM venció al campeón mundial Garri Kaspárov en una partida de ajedrez. El 15 de marzo de 2016, ocurrió otro gran y muy mediático hito, cuando el algoritmo *Deep Mind* de Google venció a Lee Sedol, campeón Coreano del tradicional juego oriental Go, en una partida de este mismo juego. Este hecho fue muy importante, ya que el Go es un juego mucho más complejo y con muchas más variables que el ajedrez, y el desarrollo de un algoritmo capaz de entender y desenvolverse en un entorno como éste demostró la increíble capacidad que tienen los algoritmos de *Deep Learning* para resolver problemas con un alto nivel de complejidad.

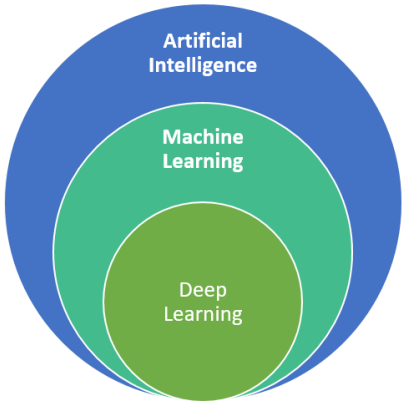


FIGURA 1.7: Esquema ilustrativo de la relación entre las disciplinas de la Inteligencia Artificial, el *Machine Learning* y el *Deep Learning* (Berchane, 2018).

Después de esta breve introducción histórica (Buchanan, 2005), conviene que aclaremos la diferencia que el término *Machine Learning* tiene con otros como *Inteligencia Arificial* o *Deep Learning*, que con frecuencia son usados indistintamente. Definir qué es la *Inteligencia Artificial* es algo complicado, ya que depende del significado

de inteligencia, que tiene múltiples interpretaciones hoy en día. A pesar de esto, simplificando y generalizando, podríamos definirla cómo la disciplina que busca la creación de máquinas que puedan imitar comportamientos inteligentes. El *Machine Learning* o *Aprendizaje Automático*, es entonces, la rama de la inteligencia artificial que busca dotar a las máquinas de capacidad de aprendizaje. Dentro de la disciplina de *Machine Learning* existen técnicas como los árboles de decisión, modelos de regresión, *clusterización* y muchas otras más. De todas estas, hay que destacar las redes neuronales, las cuales son capaces de aprender de forma jerarquizada. El *Deep Learning* o *Aprendizaje Profundo*, entra en juego cuando hablamos de redes neuronales con una gran cantidad de capas y complejidad. La diferente relación que tienen estas tres disciplinas de las que hemos hablado se puede entender de manera intuitiva a partir del esquema que se muestra en la figura 1.7. Estos algoritmos necesitan una gran cantidad de datos para ser entrenados, y es por eso que están teniendo tanto éxito en los últimos años (Lecun, Bengio e Hinton, 2015), mejorando drásticamente los últimos avances en campos como el reconocimiento de voz, la detección de objetos, el descubrimiento de fármacos, y más. A pesar de los grandes avances en el campo del *Deep Learning* en los últimos años, esta disciplina se enfrenta a varios desafíos (Dhillon y Verma, 2020), que incluyen:

- Gran cantidad de datos. Tener una gran base de datos es esencial para obtener la precisión deseada en los algoritmos de *Deep Learning*.
- Reconocimiento de expresiones faciales.
- Técnicas energéticamente eficientes y hardware de alto rendimiento. Para garantizar una alta precisión y disminuir el consumo de tiempo, es necesario trabajar con GPUs de alto rendimiento.
- Análisis de Big Data usando *Deep Learning*. Abordar diversos criterios como la variedad, el volumen y la velocidad de los datos.
- Falta de flexibilidad y falta de aprendizaje multitarea. Desarrollar algoritmos capaces de resolver múltiples tareas en varios dominios, sin la necesidad de reelaborar toda la arquitectura.

Estas aplicaciones inteligentes usan cada vez mejor la información, teniendo impacto prácticamente en todas las áreas de la industria y la investigación. En el corazón de toda esta revolución, yacen las herramientas y los métodos que la impulsan, desde el procesamiento de las grandes cantidades de datos que se generan cada día, hasta el aprendizaje y la toma de medidas útiles. Las redes neuronales profundas, junto con los avances en técnicas de *Machine Learning*, y la computación a través

de GPUs escalable y de uso global, se han convertido en los ingredientes críticos de la inteligencia artificial.

Python sigue siendo el lenguaje de programación preferido (Raschka, Patterson y Nolet, 2020) para la computación científica y la ciencia de datos (*Data Science* en inglés), impulsando tanto el rendimiento como la productividad, al permitir el uso de librerías fáciles de aprender y APIs limpias de alto nivel, como *Jupyter Notebook* o *Google Colaboratory*.

En el campo del *Machine Learning* existen distintos tipos de aprendizajes, es decir, distintas maneras en las que se puede hacer que un modelo aprenda a resolver ciertas tareas. Los tipos de aprendizaje se pueden separar en tres grupos principalmente: *aprendizaje supervisado*, *aprendizaje no-supervisado* y *aprendizaje reforzado*. Hacemos una breve explicación de estas técnicas en el apéndice A.1 . En este trabajo, haremos uso del aprendizaje supervisado.

1.6.2. Perceptrón

Las redes neuronales, como modelo computacional, existen desde mediados del siglo pasado (Rosenblatt, 1957), pero no ha sido hasta hace unos años, gracias a la mejora de la técnica y la tecnología, que su uso ha crecido exponencialmente. Estos algoritmos han sido aplicados en distintas y muy variadas áreas, como la clasificación de imágenes, el reconocimiento de voz, la generación de texto, la traducción de idiomas, la conducción autónoma, el análisis genético, la predicción de enfermedades e incluso en investigaciones recientes se ha desarrollado un modelo capaz de traducir código de un lenguaje de programación a otro (Lachaux y col., 2020). Como en la mayoría de los comportamientos y estructuras avanzadas, la complejidad de estos sistemas emerge de la interacción de unidades mucho más simples, que trabajan de forma conjunta. En el caso de las redes neuronales, la unidad básica de procesamiento recibe el nombre de *neurona* o *perceptrón*, que fue inventada en el año 1958 (Rosenblatt, 1957).

De forma similar a una neurona biológica, el perceptrón tiene conexiones de entrada a través de las que recibe los estímulos externos, los valores de entrada, que serán utilizados por la neurona para realizar un cálculo interno y dar un valor de salida. Por lo que *neurona*, no deja de ser un término un tanto fantasioso para referirse a una función matemática. Un esquema sencillo del funcionamiento de un perceptrón se muestra en la figura 1.8. Internamente, la neurona utiliza todos los valores de entrada x_i para realizar una suma ponderada de ellos. La ponderación de cada una de las entradas viene dada por el peso, ω_i, que se le asigna a cada una de las

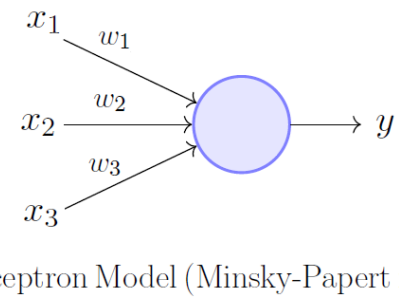


FIGURA 1.8: Esquema del funcionamiento de un perceptrón (Khapra, 2018)

conexiones de entrada. Este valor, ω_i, servirá para definir con qué intensidad cada variable de entrada afecta a la neurona. Estos pesos, ω_i, son los parámetros de nuestro modelo, y los valores que debemos ajustar para que el modelo pueda aprender. Los modelos más recientes pueden llegar a tener hasta más de un billón de parámetros (Fedus, Zoph y Shazeer, 2021). Además de la suma ponderada, para calcular el valor de salida y, hace falta sumar un término de sesgo (o *bias* en inglés), b. Por lo tanto, el funcionamiento de una neurona quedaría descrito matemáticamente por la ecuación 1.8, donde N es el número de valores de entrada que recibe la neurona.

$$y = \sum_{i=1}^{N} \omega_i x_i + b \tag{1.8}$$

Una red neuronal construida a partir de perceptrones recibe el nombre de *perceptrón multicapa*, y se muestra un esquema de su estructura en la figura 1.9. Para construir la red neuronal a partir de la unidad básica de procesamiento, el perceptrón, tenemos dos opciones a la hora de organizar las neuronas. Si colocamos las neuronas en la misma columna, o como comúnmente se denomina, en la misma capa, éstas recibirán la misma información de entrada de la capa anterior, y los cálculos que realizan los pasarán a la capa siguiente. A la primera capa de una red neuronal, en la cual están las variables de entrada, se le denomina *capa de entrada*, a la última *capa de salida*, y a las capas intermedias *capas ocultas* (o *input layer*, *output layer*, y *hidden layers* en inglés respectivamente). Al colocar dos neuronas de forma secuencial (en distintas capas), una de ellas recibe información procesada por la neurona de la capa anterior. Esto nos aporta una ventaja muy importante, ya que la red puede aprender conocimiento jerarquizado y aprender representaciones de datos con múltiples niveles de abstracción (Lecun, Bengio e Hinton, 2015). Cuantas más capas añadimos, más complejo puede ser el conocimiento que elaboremos. Esta profundidad en la cantidad de capas es lo que da nombre al *aprendizaje profundo*.

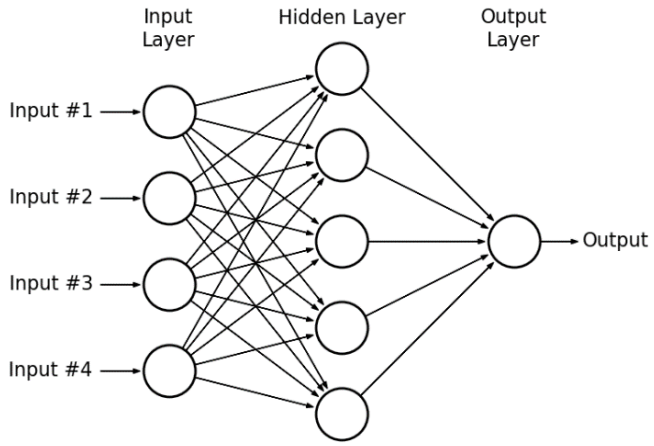


FIGURA 1.9: Estructura de un perceptrón multicapa (Zahran, 2015).

La estructura descrita, de la cual se muestra un esquema en la figura 1.9, tiene un problema grande, y es que, si aplicamos distintas combinaciones lineales, por muchas que sean, equivale a hacer una única combinación lineal. Es decir, combinar muchas neuronas es como utilizar una solar. Por lo que el perceptrón multicapa, tal y como lo hemos explicado hasta ahora, sería capaz de resolver los mismos problemas que una única neurona, es decir, problemas con solución lineal. Aquí es donde entran en juego las *funciones de activación*, que servirán para introducir manipulaciones no lineales en las operaciones internas de cada neurona. Así pues, el comportamiento matemático de una neurona descrito en la ecuación 1.8 quedaría descrito de la forma:

$$y = f(\sum_{i=1}^{N} \omega_i x_i + b) \tag{1.9}$$

donde $f(x)$ es la ya mencionada función de activación. Algunas de las funciones de activación más utilizadas son la función *ReLU*, o *Unidad Rectificada Lineal* en español (ecuación 1.10), la función *sigmoide* (ecuación 1.11) o la función tangente hiperbólica (ecuación 1.12). Cuál es la mejor función de activación dependerá del problema a resolver.

$$ReLU(x) = \begin{cases} 0 & si \quad x < 0 \\ x & si \quad x \geq 0 \end{cases} \tag{1.10}$$

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1.11}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.12}$$

El entrenamiento de una red neuronal es el proceso mediante el cual la red neuronal aprende a resolver la tarea que se le ha asignado. Esto se suele hacer combinando el algoritmo del descenso del gradiente con el algoritmo de *Backpropagation*. El proceso de aprendizaje y entrenamiento de una red neuronal se explica en detalle en el apéndice A.2

1.6.3. Redes neuronales convolucionales

Una Red Neuronal Convolucional (*Convolutional Neural Network*) (CNN) es un tipo de red neuronal profunda. Este tipo de redes fueron introducidas por primera vez en 1980 (Fukushima, 1980) y en los últimos años han logrado un rendimiento sobrehumano en algunas tareas complejas de visión, con aplicaciones en campos como la conducción autónoma, la clasificación y detección de objetos, o el análisis de imagen médica, entre otros (Geron, 2017; Suzuki, 2017). Las redes convolucionales fueron inspiradas por procesos biológicos que ocurren en el cerebro (Fukushima, 1980; Hubel y Wiesel, 1968) en el hecho de que el patrón de conectividad entre neuronas artificiales se asemeja a la organización de la corteza visual animal. A pesar de que las CNNs fueron inventadas en los años 80, no fue hasta los años 2000 que, con la implementación de las GPUs para el entrenamiento de la red, empezaron a ganar popularidad y se vieron potenciadas sus aplicaciones. En 2004, Oh y Jung mostraron que la implementación de una red neuronal estándar con GPU era 20 veces más rápida que una implementación equivalente en CPU (Oh y Jung, 2004).

Para ser capaz de entender bien el funcionamiento de una CNN, es necesario comprender los principios básicos que hay detrás del proceso de visión, y no del proceso de visión artificial, sino del animal. Por poner un ejemplo, cuando nosotros vemos una imagen de un rostro humano, somos capaces de reconocer que se trata de un rostro humano debido a que detectamos algunos elementos que, en virtud de nuestro aprendizaje, sabemos que conforman un rostro: como detectar dos ojos, una nariz, una boca, etc. Ahora, ¿cómo somos capaces de detectar lo qué es un ojo? Pues porque hemos detectado otros elementos que normalmente conforman un ojo: una pupila negra, pestañas, una superficie blanca (la esclerótica), etc. Y hemos sido capaces de reconocer todos estos elementos porque también somos capaces de detectar patrones circulares, cambios de contraste, texturas, etc. Y así sucesivamente. Si reproducimos los pasos que realiza nuestro córtex visual, nos encontraremos con

un procesamiento en cascada, donde primero se identifican aquellos elementos básicos y generales, y donde en posteriores capas esto se combina para generar patrones cada vez más complejos (Hubel y Wiesel, 1968).

Las CNN son un tipo de red neuronal cuyo diseño ha sido pensado para sacar partido a algo muy evidente que encontramos en una imagen, su estructura espacial. Si a la hora de trabajar con imágenes utilizamos una red neuronal estándar (perceptrón multicapa), lo que estaríamos haciendo es introducir el valor de cada píxel como si de una variable independiente se tratara, es decir, como si fuera un vector plano. En ningún momento le estaríamos dando la importancia que tiene la posición del píxel dentro de la imagen. Normalmente, el valor de un píxel va a estar estrechamente ligado al de sus píxeles vecinos, y esto es lo que hace que surjan estructuras, formas y patrones que, analizadas correctamente, nos pueden ser de ayuda a la hora de entender qué estamos viendo. Y ésta es la idea tras la que surgen las redes neuronales convolucionales.

Una CNN es un tipo de red neuronal cuya principal característica es la de aplicar un tipo de capa donde se realiza una operación matemática conocida como convolución. Una convolución, aplicada sobre una imagen, no es más que una operación que, jugando con los valores de los píxeles, es capaz de producir otra imagen, que llamaremos mapa de características. En concreto, cada nuevo píxel se calculará colocando una matriz de números, que llamaremos filtro o *kernel*, sobre la imagen original y donde multiplicaremos y sumaremos los valores de cada píxel vecino para obtener así el valor del píxel del mapa de características. Este proceso se ilustra en la figura 1.10 (los filtros pueden ser de tantas dimensiones como queramos, se ha ilustrado un filtro de 3 dimensiones, matriz 3x3, como ejemplo). El mapa de características completo lo obtendremos desplazando el filtro, es decir realizando la operación de convolución para cada píxel de la imagen original. El mapa de características dependerá, por tanto, de cómo configuremos los parámetros del filtro. Aunque en realidad, no seremos nosotros quienes configuremos los valores de estos parámetros, si no que será la red neuronal la que irá aprendiendo a ajustar estos valores para ser capaz de cumplir su tarea. Aprender los filtros para detectar distintos patrones será el principal trabajo de la CNN. Es por eso por lo que, a la imagen generada tras la aplicación de un filtro a la imagen original, le llamaremos mapa de características, ya que actúa como un mapa donde se nos indica en qué parte de la imagen se ha detectado la característica buscada por dicho filtro.

Resumiendo lo dicho hasta ahora: una imagen entra en la red, se aplican una serie de convoluciones, y se genera un conjunto de mapas de características. La clave de las CNN se encuentra en que este tipo de operación se realiza de forma secuencial,

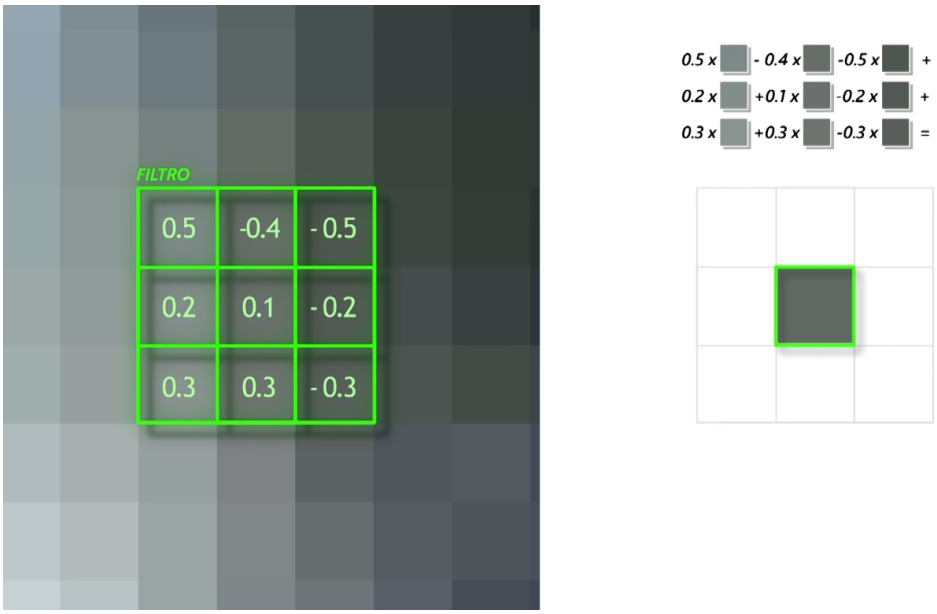


FIGURA 1.10: Esquema ilustrativo de la aplicación de una convolución a una imagen (DotCSV, 2020).

de manera que la salida (u *output*) de una de las capas se convierte en la entrada (o *input*) de la siguiente. El tamaño común de los filtros suele ser de 3x3, 5x5, o incluso hasta 7x7. En primera instancia, podemos llegar a pensar que un filtro tan pequeño no va a ser capaz de detectar patrones tan complejos como lo puede ser, una señal de tráfico, un ojo o una nariz. Sin embargo, un filtro, a pesar de su pequeño tamaño, sí que es capaz de detectar patrones complejos cuyo tamaño es mayor que el suyo. Esto es debido a la sucesión de capas dentro de la CNN y a la profundidad de esta. La operación de convolución, a medida que avanzamos en las capas de la red, se va haciendo más potente, porque donde en la imagen original se tenía una región de 9 píxeles (suponiendo un filtro 3x3), la primera convolución convierte está región en un único píxel de información en el mapa de características. Por lo tanto, al aplicar una nueva convolución sobre el mapa de características generado, se estará accediendo a cada vez más información espacial de la imagen original. Además, este proceso se estimula reduciendo cada cierto número de capas la resolución de los mapas de características. Para la reducción de la resolución se suele utilizar una operación conocida como *Max pooling* ya que es la que mejor suele funcionar en la práctica (Scherer, Müller y Behnke, 2010). Dentro de la red, esta operación recibe el nombre de capa de reducción de muestreo o *pooling layer*. Normalmente se suele insertar dentro de la red de forma periódica una *pooling layer* (típicamente seguida de una función de activación, como lo puede ser una *ReLU*) entre capas convolucionales sucesivas (Geron, 2017).

En definitiva, las convoluciones no son más que unas operaciones capaces de detectar cambios de contraste, texturas, etc. La clave está en que en una CNN

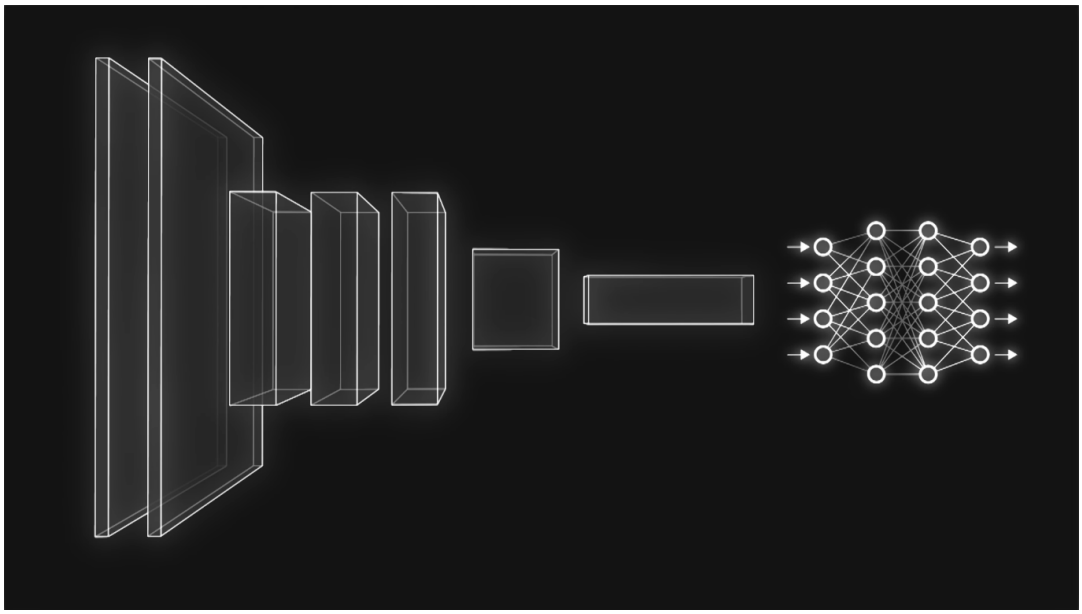


FIGURA 1.11: Esquema ilustrativo de la estructura de una CNN tal y como la hemos explicado (DotCSV, 2020).

se hacen detecciones sobre las detecciones (los mapas de características) recibidas de las capas anteriores. Es decir, aplicar convoluciones sucesivas sobre mapas de características generados anteriormente nos permite componer cada vez patrones más complejos. Es por eso que el diseño de una arquitectura convolucional se suele presentar en artículos como una especie de embudo, donde la imagen inicial se va comprimiendo espacialmente (es decir, su resolución va disminuyendo) a la vez que su grosor va aumentando (es decir, el número de mapas de características va aumentando). Una vez se llega al final del embudo, la red ya habrá detectado todos los patrones necesarios. Es decir, la red cuenta con muchos mapas de características que, ahora sí, se pueden utilizar como *inputs* independientes dentro de una red neuronal multicapa, que se encargará de tomar la decisión sobre qué es lo que hay en la imagen. Un esquema de toda esta estructura recién explicada se ilustra en la figura 1.11.

Capítulo 2

Objetivos

El principal objetivo de este trabajo es el de implementar simulaciones realistas de la emisión anisotrópica de rayos γ por parte de núcleos excitados de espín $> 1/2$ hiperpolarizados en presencia de campos magnéticos. A diferencia de lo que se hace en el artículo de Nature (Zheng y col., 2016), se pretende modelar la interacción de la precesión del espín con el campo magnético y con los gradientes magnéticos en función del tiempo. Manteniendo la información temporal se espera que estas simulaciones sirvan como prueba de concepto para el proyecto Gamma-MRI (sección 1.5).

También se pretende confirmar la eficacia de algoritmos clásicos (algoritmo ML-EM, sección 3.5) para la reconstrucción de la imagen. Estos algoritmos se conocen bien, y han demostrado su eficacia en otras modalidades de imagen, por lo que nos conviene estudiar su eficacia para esta nueva modalidad de imagen (PNI, sección 1.4).

Además, exploraremos otras maneras de hacer la reconstrucción de la imagen, en concreto, con redes neuronales. Se ha demostrado en los últimos años que los algoritmos de *Deep Learning* son capaces de realizar la reconstrucción en distintas modalidades de imagen médica (Ahishakiye y col., 2021). Se pretende estudiar la información que hace falta darle a la red neuronal para que ésta sea capaz de aprender a reconstruir la imagen. Por lo que, otro objetivo importante del trabajo es realizar la implementación y el entrenamiento de una red neuronal en Python.

Los objetivos más generales desde el punto de vista didáctico de este trabajo fueron tres principalmente: familiarizarse con las principales técnicas de imagen y comprender sus limitaciones; entender los fundamentos y las técnicas de la nueva modalidad de obtención de imagen descrita en el artículo de Nature (Zheng y col., 2016); y familiarizarse con los fundamentos teóricos y las técnicas del campo del *Deep Learning*, en especial con las CNN. Por último, también se pretende familiarizar con

la perspectiva investigadora a la hora de enfrentarse a un problema.

Capítulo 3

Métodos

Este capítulo se divide en tres principales bloques. Las simulaciones (secciones 3.1, 3.2, 3.3 y 3.4), la reconstrucción de la imagen con algoritmo iterativo (sección 3.5) y la reconstrucción de la imagen con una red neuronal (secciones 3.6 y 3.7).

3.1. Generación de rayos de forma anisotrópica, caso estático

En primer lugar, comenzaremos simulando la generación de rayos γ de forma anisotrópica. Para simular un número N de eventos, lo que queremos es saber el ángulo de emisión de cada uno de los eventos. Este ángulo será un valor comprendido entre $-\pi$ y π. Para esto, no hay más que generar números de forma aleatoria haciendo que éstos sigan una cierta distribución de probabilidad, donde la distribución de probabilidad será la dependencia angular de las emisiones gamma de núcleos polarizados, ecuación 1.7:

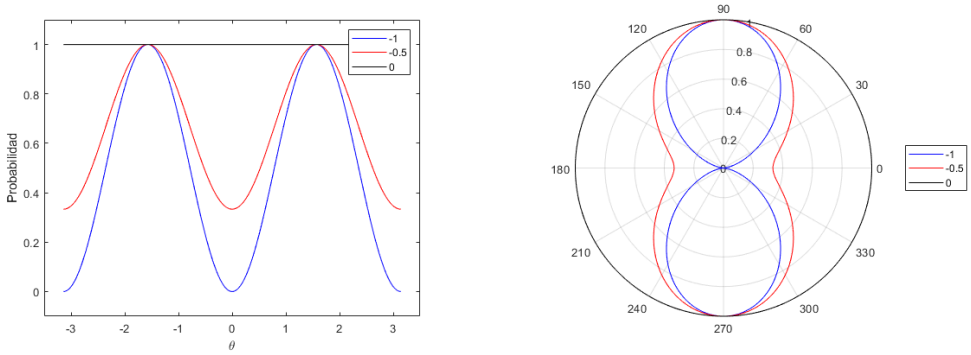
$$W(\theta) = a_0 - a_2 \cos(2\theta) \tag{3.1}$$

donde tomaremos $a_0 = 1$ y a_2 será el valor que irá variando, dependiendo del grado de polarización, donde $a_2 = 0$ implica ausencia de polarización, y $a_2 = 1$ implica máxima polarización. Por lo tanto, lo que queremos hacer es generar números aleatorios entre $-\pi$ y π, que serán los valores de los ángulos de cada emisión. Para que $W(\theta)$ de la ecuación 3.1 se pueda interpretar como una probabilidad, es necesario normalizarla de manera que su valor máximo sea 1. Es por ello que la expresión que utilizaremos en la simulación es la siguiente:

$$P(\theta) = \frac{a_0 - a_2 \cos(2\theta)}{a_0 + a_2} \tag{3.2}$$

Podemos dibujar en Matlab la ecuación 3.2, tanto en coordenadas cartesianas como en coordenadas polares. Dichas representaciones se muestran en la figura

3.1 para valores del parámetro a_2 de 0, 0.5 y 1. Como se puede apreciar, el caso de $a_2 = 0$ es el caso de nula polarización, lo que implica emisión isótropa, es decir, no hay ningún ángulo favorecido para la emisión (misma probabilidad para todos los ángulos).



(a) Coordenadas cartesianas, con $\theta \in [-\pi, \pi)$.

(b) Coordenadas polares.

FIGURA 3.1: Probabilidad de emisión de rayos γ en función del ángulo (ecuación 3.2), representada tanto en coordenadas cartesianas como en coordenadas polares, para valores del parámetro a_2 de 0, 0.5 y 1.

Para generar números aleatorios que sigan la distribución de la ecuación 3.2 usaremos un algoritmo conocido con el nombre de *Rejection sampling* (Bishop, 2006) en inglés, que se podría traducir como *Muestreo de rechazo*. El algoritmo consiste en tres sencillos pasos:

1. Generar un valor de θ aleatorio entre $-\pi$ y π con la distribución uniforme, lo llamaremos θ_0. Obtener el valor de $P(\theta_0)$ de acuerdo con la ecuación 3.2, que tendrá un valor entre 0 y 1.

2. Generar un número aleatorio s entre 0 y 1 con la distribución uniforme. Comprobar si $P(\theta_0) \geq s$:

   - Si se cumple $P(\theta_0) \geq s$, se acepta la emisión y se guardan los valores de θ_0 y de $P(\theta_0)$.
   - Si no se cumple $P(\theta_0) \geq s$, se rechaza la emisión (de ahí el nombre del algoritmo), y se vuelve a probar con un nuevo valor de θ_0 (se vuelve al paso 1).

3. Repetir hasta conseguir un cierto número N de emisiones.

Para la cantidad de emisiones generadas establecemos un valor para la actividad de la fuente, A_f, y un tiempo de medida t_m. Por lo tanto, la cantidad de

emisiones, N, que se deberían detectar en un tiempo de medida t_m será:

$$N = A_f \cdot t_m \tag{3.3}$$

donde A_f tiene que estar dada en desintegraciones por segundo, es decir, en becquereles (Bq) y t_m en segundos (s). En nuestro caso, para utilizar el algoritmo de *Rejection sampling*, generaremos $5 \cdot N$ valores de θ_0, ya que no todos los valores de θ_0 generados con el algoritmo acabarán siendo emisiones aceptadas. Después seleccionaremos solamente las N primeras emisiones aceptadas, descartando las demás, para así tener un total de N emisiones. La implementación que hemos hecho del algoritmo en Matlab la podemos encontrar en el código 3.1.

```
n_max = 5 * n_em;    % WE CREATE MORE THAN NEEDED AND THEN WE PRUNE (
    THINNING)
theta_value = 2 * pi * rand(n_max,1) - pi;%GENERATE n_max VALUES
    BETWEEN -pi y pi
P0 = (a0 - a2 * cos(2 * theta_value)) / (a0 + a2);%P(theta_value)
s = rand(n_max, 1); %generate a random number s between 0 and 1
theta_true = theta_value(s <= P0);%we select the values of theta that
    satisfy s <= P0(theta)
theta_true = theta_true(1:n_em);%we discard the excess to have the
    requested number of emissions(n_em)
P0_true = P0(s <= P0);%Same for P0 values
P0_true = P0_true(1:n_em);
```

CÓDIGO 3.1: Algoritmo de *Rejection Sampling*, en Matlab.

```
    histogram(X, nbins, 'Normalization', 'pdf')
```

CÓDIGO 3.2: Histograma normalizado como una PDF, en Matlab.

Por último, para comprobar si los ángulos generados para las emisiones aceptadas siguen la probabilidad de la ecuación 3.2, dibujaremos el histograma de dichos ángulos generados normalizado como una Función de Densidad de Probabilidad (*Probability Density Funcion*) (PDF). Para ello, utilizamos el código 3.2 en Matlab, donde lo importante son los 2 últimos argumentos de la función `histogram`, ya que el argumento `X` es el vector del cuál queremos hacer el histograma, y el argumento `nbins` es el número de bins. Encima de dicho histograma dibujaremos la PDF de la ecuación 3.2. Para ello no hay más que integrar dicha ecuación multiplicada por una constante de normalización A desde $-\pi$ hasta π, y forzar que esta integral sea igual a 1:

$$\int_{-\pi}^{\pi} A \cdot P(\theta) = 1 \tag{3.4}$$

de esta forma obtenemos el valor para la constante de normalización A:

$$\int_{-\pi}^{\pi} A \cdot \frac{a_0 - a_2 \cos(2\theta)}{a_0 + a_2} = \frac{2\pi a_0 A}{a_0 + a_2} = 1 \Rightarrow A = \frac{a_0 + a_2}{2\pi a_0} \tag{3.5}$$

Así pues, podemos escribir la ecuación de la PDF de $P(\theta)$ como:

$$P_{PDF}(\theta) = A \cdot P(\theta) = \frac{a_0 - a_2 \cos(2\theta)}{2\pi a_0} \tag{3.6}$$

El código utilizado en Matlab para la simulación de esta sección se muestra en el apéndice C.1. Los resultados se muestran y se discuten en la sección 4.1.

3.2. Tiempo entre medidas, caso dinámico

El siguiente paso para hacer la simulación más realista es tener en cuenta el tiempo entre medidas (o *interarrival time* en inglés), es decir, tener en cuenta que los rayos γ se emiten en un tiempo dado. La actividad nos dice el promedio de desintegraciones por segundo que tiene la fuente. Para simplificar la simulación, consideraremos que la actividad es constante en el tiempo y no cambia. El proceso de emisión radiactiva de rayos γ sigue la estadística de un proceso homogéneo de Poisson (*Homogeneous Poisson point process*) (Cannizzaro y col., 1978), el cual se explica más adelante, en la sección 3.2.1.

Para realizar la simulación es algo bastante sencillo, ya que solo hay que tener en cuenta que entre una emisión y la siguiente, el tiempo (*interarrival time*) sigue la distribución exponencial:

$$P_t(t) = \lambda e^{-\lambda t} \tag{3.7}$$

donde, en nuestro caso, λ sería la actividad de la fuente, $\lambda = A_f$. Por lo tanto, para la simulación, generaremos una serie de eventos temporales que sigan dicha distribución exponencial. Esto sería el tiempo entre medidas de cada una de las emisiones de rayos γ de nuestra simulación. Para obtener el tiempo de llegada de la emisión número n, simplemente se calcula la suma acumulativa de los tiempos entre medidas de los primeros n eventos. El código para esta parte de la simulación es muy sencillo (debido a que Matlab incorpora una función, `exprand`, para generar números aleatorios que sigan una distribución exponencial) y se muestra en el código 3.3.

```
1 p_at0 = exprnd(1.0 / act, [n_em,1]);%We generate numbers with the
    interarrival time distribution
```

```
2 tiempos = cumsum(p_at0);%We get each event's time with the cumulative
    sum
```

CÓDIGO 3.3: Simulación del tiempo entre medidas de las emisiones, en Matlab.

3.2.1. Proceso homogéneo de Poisson

La teoría de esta sección se ha extraído del libro de Kingman (Kingman, 1992). Un proceso de Poisson se caracteriza por seguir la distribución de Poisson. La distribución de Poisson es la distribución de probabilidad de una variable aleatoria N, de forma que la probabilidad de que N sea igual a n viene dada por:

$$P\{N = n\} = \frac{\Lambda^n}{n!} e^{-\Lambda} \tag{3.8}$$

donde el parámetro Λ define la forma de la distribución, de hecho, Λ es el valor esperado de N. Por definición, un proceso de Poisson tiene la propiedad de que el número de puntos en una región acotada del espacio subyacente del proceso es una variable aleatoria cuya distribución viene dada por la distribución de Poisson.

Si en un proceso de Poisson tiene un parámetro de la forma $\Lambda = \nu\lambda$, donde ν es una medida de Lebesgue y λ es una constante, el proceso recibe el nombre de Proceso homogéneo de Poisson. λ se puede interpretar como el número promedio de puntos por alguna unidad, como longitud, área, volumen o tiempo (tiempo en nuestro caso), también recibe el nombre de *tasa media*.

Un proceso homogéneo de Poisson se puede interpretar como un proceso de conteo, que se puede denotar por $\{N(t), t \geq 0\}$. Un proceso de conteo representa el número total de eventos que han sucedido hasta el momento t (t incluido). Un proceso de conteo será un proceso homogéneo de Poisson si cumple las siguientes propiedades:

- $N(0) = 0$.
- El número de eventos en un intervalo de longitud t es una variable aleatoria de Poisson con media λt.

Por lo tanto, la probabilidad de que la variable aleatoria $N(t)$ sea igual a n viene dada por:

$$P\{N(t) = n\} = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{3.9}$$

El proceso de conteo también se puede definir de forma que las diferencias de tiempo entre los eventos del proceso de conteo son variables exponenciales, con media $1/\lambda$. La diferencia de tiempos entre eventos recibe el nombre de tiempo entre llegadas (*interarrival time* en inglés).

3.3. Precesión del emisor

Es en esta sección donde encontramos la parte novedosa del trabajo, con respecto al método publicado en el artículo de Nature (Zheng y col., 2016), en el que este proyecto está inspirado. En el modelo que describen en dicho artículo, la probabilidad de emisión es independiente de la precesión del emisor, ya que, como explican en las ecuaciones 7-9, alinean el detector en la dirección del campo magnético (eje z). Por lo tanto, la probabilidad de emisión dependerá únicamente de la relajación, según el espín se vaya orientando con el eje z, el ángulo θ_{eff} entre la dirección del detector y el ángulo de precesión del espín tenderá a cero. Posteriormente, en las ecuaciones 20-23, para el caso en el que el detector se encuentra en dirección perpendicular al eje z, toman un promedio temporal para el ángulo θ_{eff}. Como explican en el artículo, este promedio temporal hace que se pierda mucha información sobre la señal. En este trabajo buscamos, teniendo una buena señal temporal, ser capaces de obtener más información sobre la señal, omitiendo el promedio temporal. Para hacer esto, nos basamos en nos basamos en que el periodo de precesión es del orden o superior al microsegundo, y que para la resolución temporal de los detectores de radiación usados comúnmente se trabaja con resoluciones del orden de nanosegundos o mejores aún.

Para tener en cuenta la precesión del emisor cuando estamos en NMR podemos añadir una fase, θ_0 al ángulo de emisión, θ, de la ecuación 3.2. Esta fase θ_0 va cambiando con el tiempo, debido a la precesión del espín. Así pues, la probabilidad de emisión (ecuación 3.2) nos quedaría de la forma:

$$P(\theta) = \frac{a_0 - a_2 \cos\left(2(\theta + \theta_0)\right)}{a_0 + a_2} \tag{3.10}$$

Para calcular θ_0 podemos utilizar la ecuación 7 del artículo de Nature (Zheng y col., 2016), ligeramente modificada:

$$\theta_0 = \theta_{00} + \gamma_{Xe} B_0 t \tag{3.11}$$

donde γ_{Xe} es el ratio giromagnético del Xenón (para ^{131m}Xe, $|\gamma_{Xe}/2\pi| = 1{,}37$ MHz T^{-1},

Calaprice y col., 1985), y t es el tiempo de cada emisión. La modificación que hemos realizado a la ecuación 7 del artículo de Nature ha sido convertir la dependencia con el gradiente aplicado y la posición en la constante θ_{00}, ya que, de momento, no tenemos dependencia con la posición debido a que estamos simulando un único *vóxel* de actividad. Para realizar las simulaciones utilizaremos un valor para el campo magnético estático de $B_0 = 0.1$ T.

3.4. Caso dinámico con múltiples fuentes

Todas las simulaciones realizadas hasta ahora partían del hecho de que había una única fuente de actividad, un único vóxel. El siguiente paso para tener una simulación más realista es añadir a dicha simulación múltiples fuentes que emitan rayos γ. Es decir, realizar la simulación con múltiples vóxeles en vez de con uno único.

Para ello, asumimos que, mientras el espín esta precesando en el plano transversal (plano x-y), la muestra está sometida a un gradiente de campo magnético lineal descrito por el vector:

$$\boldsymbol{G} = \frac{\partial B_z}{\partial x}\hat{\boldsymbol{x}} + \frac{\partial B_z}{\partial y}\hat{\boldsymbol{y}} \tag{3.12}$$

Tras aplicar dicho gradiente, un espín localizado en la posición $\boldsymbol{r}$ habrá precesado en el plano x-y un ángulo θ_0 dado por (ecuación 7 del artículo de Nature, Zheng y col., 2016):

$$\theta_0 = \gamma_{Xe}\left(\boldsymbol{G}\cdot\boldsymbol{r} + B_0 t\right) \tag{3.13}$$

Esta será la expresión que utilizaremos, en lugar de la 3.11, para simular la precesión del espín teniendo en cuenta el gradiente aplicado y la posición. Para simular la posición, construiremos un mallado bidimensional, con dimensiones de 7×7. Para los gradientes, tanto en x ($\frac{\partial B_z}{\partial x}$), como en y ($\frac{\partial B_z}{\partial y}$), los valores irán desde -2π hasta 2π. Habrá tantos gradientes en x cómo vóxeles en el eje x, y tantos gradientes en y cómo vóxeles en el eje y. Podemos decir que las unidades, tanto de la posición, como de los gradientes, son arbitrarias, ya que, si la posición está dada en metros (m), los gradientes estarán dados en tesla por metro (T m^{-1}), o si la posición está dada en milímetros (mm), los gradientes estarán dados en tesla por milímetro (T mm^{-1}).

El código completo de la simulación se muestra en el código C.1 en el apéndice C.1. Vamos ahora a analizar y explicar parte del código implementado para la

simulación. En primer lugar, en el código 3.4 se definen los parámetros básicos (número de vóxeles en el eje x, `Nx`, y número de vóxeles en el eje y, `Ny`). Después se definen los gradientes, tanto en x como en y. Como hemos dicho anteriormente, dicho gradientes van desde -2π hasta 2π, tal y como podemos ver en las líneas 7 y 8 del código 3.4. Por último, se define el mallado con la posición de cada uno de los vóxeles, que irán desde -2 hasta 2, tanto en x como en y. Por lo tanto, el mallado tendrá dimensiones de 7 $\times$ 7, tal y como se dijo anteriormente.

```
1  Nx=7;
2  Ny=7;
3  Nvoxels = Nx*Ny;%Number of voxels (in this case only in x)
4  Ngradx=Nx;
5  Ngrady=Ny;
6  Ngrads=Nvoxels;%Number of gradiens applied (typically equal to Nvoxels)
7  gradsx=Nx*(2*pi/Ngradx)*linspace(-1,1,Ngradx);%Value of the gradient k
       in each case (k_x)
8  gradsy = Ny*(2*pi/Ngrady)*linspace(-1,1,Ngrady);%Value of the gradient
       k in each case (k_x)
9
10 [posx,posy] = meshgrid(linspace(-2,2,Nx),linspace(-2,2,Ny)); %voxels'
       position from x=-2 to x=2, and y=-2 to y=2
```

CÓDIGO 3.4: Implementación del mallado bidimensional para simular la posición y de los gradientes, en Matlab.

La parte más importante de la simulación se muestra en el código 3.5. Dicho código comienza realizando dos bucles, uno para los gradientes en x, y otro para los gradientes en y. Después, se realiza un tercer bucle para realizar la simulación de las emisiones para cada uno de los vóxeles. Dentro ya de este tercer bucle, en primer lugar, en las líneas 10-13 del código simulamos el tiempo entre medidas (sección 3.2). Después, en las líneas 15-17, calculamos el ángulo de precesión del espín para cada evento (ecuación 3.13) y calculamos el ángulo de emisión de cada evento de acuerdo con la distribución de probabilidad de las emisiones de rayos γ (ecuación 3.10). Por último, almacenamos en la variable `detections` la información que utilizaremos para la reconstrucción de la imagen. Dicha información es el tiempo, el ángulo de emisión y el vóxel de cada uno de los eventos.

```
1  nev_all = 0;
2  for kx=1:Ngradx  %Loop over the different gradients
3  for ky=1:Ngrady  %Loop over the different gradients
4      gradx = gradsx(kx);
5      grady = gradsy(ky);
6      for j=1:Nvoxels%Using a loop for now (Each voxel is like a new
       simulation)
7          nev_j = floor(1.5*Activity(j)*time);%Aditional factor 1.5 just
       in case we dont get enough
```

```
8         %nev_j: number of events for j simulation (integer, thats why
    floor is used)
9         mu_j = 1.0/Activity(j);%Exponential distribution's parameter
10        pt_j = exprnd(mu_j, [nev_j,1]);%We generate numbers (nev_j#
    numbers) with the interarrival time distribution
11        t_j = cumsum(pt_j);%We get each event's time with the
    cumulative sum
12        t_j = t_j(t_j<time);%We just select events up to acquisition
    time
13        nev_j = length(t_j);%Set the number of emissions only up to
    acquisition time
14        theta_offsets = (gradx*posx(j) + grady*posy(j) + gyro * B) *
    t_j;%Polar angle of the Spin over time
15        ithetas = 1+floor(n_em*rand(nev_j,1));%Getting random angles
    from angular distribution
16        theta_j = theta_true(ithetas) + theta_offsets;%Module 2pi can
    be used to simplify it (make that are angles belong to [0, 2pi)
17        detections(nev_all+1:nev_all+nev_j,1)=t_j;%Storage of the
    result
18        detections(nev_all+1:nev_all+nev_j,2)=theta_j;
19        detections(nev_all+1:nev_all+nev_j,3)=j;%This is the
    information we want to reconstruct
20        nev_all = nev_all+nev_j;
21    end
22
23    % We can sort the events by arrival time (as it happens in the
    detection)
24    [~,idx] = sort(detections(:,1));% sort just the first column
25    %~ is used so we dont store the sorted values, we just want the
    indexes
26    detections_sorted{kx,ky} = detections(idx,:);% sort the whole
    matrix using the sort indices
27 end
28 end
29 Total_events = nev_all
```

CÓDIGO 3.5: Bucle en el que realizamos la simulación para cada gradiente y cada uno de los vóxeles.

3.5. Reconstrucción con el algoritmo ML-EM

La reconstrucción de la imagen a partir de los datos obtenidos en la simulación de la sección 3.4 la realizaremos con un algoritmo de reconstrucción iterativo. En concreto, utilizaremos el algoritmo ML-EM, que recibe su nombre por sus siglas

en inglés, *Maximum Likelihood* (ML), *Expectation Maximization* (EM). La principal ventaja de los algoritmos iterativos recae en que se obtiene una buena calidad de imagen. Sin embargo, son métodos muy costosos computacionalmente (Lange y Carson, 1984).

Los algoritmos iterativos tienen un funcionamiento bastante intuitivo. Esencialmente, comienzas haciendo una suposición en la distribución de actividad de cada vóxel de la imagen. A partir de esta distribución de actividad, y utilizando el modelo del escáner (en nuestro caso, el modelo de la simulación) realizas una estimación de los datos medidos por el escáner (en nuestro caso, una estimación de los datos obtenidos en la simulación). Después, comparas la estimación de los datos con los datos reales (en nuestro caso, con los datos obtenidos a partir de la simulación), por ejemplo, como en el caso del algoritmo ML-EM, realizando el cociente entre los datos reales y la estimación (o proyección). Este cociente, que estará en el espacio de los datos, se puede retroproyectar, para obtener así una corrección para aplicar a la imagen de manera multiplicativa. Básicamente, el algoritmo ML-EM que vamos a utilizar se podría describir como la aplicación de 4 sencillos pasos hasta que se logre la convergencia:

1. **Proyectar.** Estimar, a partir de la suposición de la distribución de actividad, los datos que se obtienen en la simulación.
2. **Comparar.** Comparar los datos reales obtenidos en la simulación (sección 3.4) con la proyección del paso anterior. En concreto, la comparación la hacemos calculando el cociente entre los datos reales y la proyección.
3. **Retroproyectar.** Retroproyectamos el cociente del paso anterior, que se encuentra en el espacio de los datos, para obtener la corrección en el espacio de la imagen.
4. **Actualizar.** Actualizar la suposición de la distribución de actividad, multiplicándola por la corrección obtenida en el paso anterior.

La implementación del algoritmo ML-EM que hemos realizado en Matlab se muestra en el código C.2 del apéndice C.2. Aquí haremos una breve explicación de la parte más importante, que se muestra en el código 3.6. En primer lugar, se empieza haciendo una suposición de la distribución de actividad en la imagen. En nuestro caso, suponemos un valor para la actividad de 1 para todos los vóxeles. Se define también el número de veces (número de iteraciones) que queremos aplicar el algoritmo. Después, se realizan dos bucles, uno para los gradientes en x y otro para los gradientes en y. Para cada par de gradientes, se cargan los datos de la simulación (en

la variable `detect`, línea 11). Posteriormente, en las líneas 14-18, se realiza un bucle para cada uno de los vóxeles de la imagen para realizar la proyección estimar los datos a partir de la suposición de la distribución de actividad. Con esto, ya podemos calcular la proyección (línea 19), y así utilizarla para calcular el término de corrección. Por último, solo queda actualizar la imagen multiplicando por la corrección (línea 25).

```
1  X = ones(Nvoxels,1);% Reconstruction goal: Activity in each voxel
2  sensit = ones(Nvoxels,1)*Ngrads*time; %Improve this (for now, it's
       constant)
3
4  Niter=50
5
6  for iter=1:Niter% Loop over the iterations of the recons
7   iteration = iter
8
9   for kx=1:Ngradx%Loop over the different gradients
10  for ky=1:Ngrady%Loop over the different gradients
11   sumlor = zeros(1,Nvoxels);%Initialization of the corrections array
12   gradx = gradsx(kx);%Value of this gradient
13   grady = gradsy(ky);%Value of this gradient
14   detect = detections_sorted{kx,ky};%Getting the detections with this
       gradient
15   nevs = length(detect);%Number of detections in this particular
       gradient
16   c = zeros(nevs,Nvoxels);%Probability Matrix
17   for j=1:Nvoxels
18    theta_offsets = mod((gradx*posx(j) + grady*posy(j) + gyro * B) *
      detect(:,1),2*pi);%Polar angle of the Spin over time
19    dtheta = detect(:,2) - theta_offsets;%Detection angle respect to the
        Spin for each voxel
20    c(:,j)= a0 - a2*cos(2*dtheta);%Probability for each event and voxel
21   end
22   proj = c*X;%Projection (over all detected events)
23   ratio = c./proj;%Backward Projection (list-mode)
24   ratio(proj<eps)=0.;%Avoids NaNs
25   sumlor = sumlor + sum(ratio,1);%Acumulate Corrections from each
       gradient
26
27   update = sumlor'./sensit;%Normalize Corrections by Sensitivity
28   X = X.*update;%Update Voxel Activity
29  end
30  end
31 end
```

CÓDIGO 3.6: Parte más importante de la implementación del algoritmo ML-EM en Matlab (implementación completa en el código C.2).

3.6. Reconstrucción con una CNN

La principal limitación de algoritmos clásicos de reconstrucción de imagen, como el algoritmo ML-EM (sección 3.5), recae, principalmente, en su tiempo de cómputo. Además, en estos métodos de reconstrucción, cada uno de los píxeles de la imagen que se va a reconstruir está relacionado con cada una de las medidas, por lo tanto, la matriz del sistema es significativa. Si se da el caso de que queremos reconstruir una imagen con una buena resolución (alto número de píxeles), esto puede presentar problemas de memoria para el ordenador, debido a un tamaño excesivo de la matriz del sistema. Sin embargo, una red neuronal, con el entrenamiento adecuado, podría lograr que este proceso de reconstrucción fuese mucho más rápido y eficiente, como ya se ha demostrado en los últimos años (Ahishakiye y col., 2021).

Para realizar la reconstrucción de imágenes con una red neuronal, lo primero que necesitamos es el conjunto de datos para entrenar a la red, ya que vamos a hacer uso de aprendizaje supervisado. El tipo de red que vamos a utilizar recibe el nombre de U-NET (Ronneberger, Fischer y Brox, 2015), y es bien conocida por sus éxitos en el campo de la imagen médica. Este tipo de red recibe como entrada una imagen, y da como salida otra imagen también. La estructura y el funcionamiento de este tipo de red se explica con detalle en el apéndice B.

3.6.1. Conjunto de datos

Recordemos que, en el campo del aprendizaje supervisado, el conjunto de datos utilizado se divide en dos principales grupos: conjunto de datos para el entrenamiento, el cual se utiliza para entrenar a la red; y el conjunto de datos de validación, que se utilizará, una vez entrenada la red, para comprobar su rendimiento en datos que no ha visto antes. En concreto, nosotros destinaremos el 80 % del conjunto de datos al entrenamiento de la red, y el 20 % restante formará parte del conjunto de datos para la validación.

El conjunto de datos que vamos a utilizar para entrenar a la red consiste en dos tipos de imágenes: el *input* de la red, que serán las imágenes que la red reciba como entrada para reconstruir la imagen; y el *output*, que serán las imágenes que la red, en caso de que funcione correctamente, deberá darnos como salida, es decir, la imagen ya reconstruida.

Para el conjunto de imágenes que corresponde al *output*, queremos utilizar perfiles de actividad lo más realistas posibles. Para ello utilizaremos imágenes reales

de un escáner de PET, el cual se encuentra en formato .raw, y podemos descargar de `https://tomografia.es/data/F18_V2_ALL.raw` (Herraiz, Bembibre y López-Montes, 2021). Este fichero contiene 8 imágenes tridimensionales obtenidas mediante un escáner PET con la molécula radiotrazadora F18. Las dimensiones de este archivo son de 154 $\times$ 154 píxeles, conteniendo 80 rodajas y 8 casos distintos, es decir, 154 $\times$ 154 $\times$ 80 $\times$ 8. Cada vóxel cúbico de estas imágenes tiene dimensiones de 0.28 $\times$ 0.28 $\times$ 0.28 mm. Podemos visualizar dicha imagen con un programa de visualización de imagen médica como lo es Amide. También podemos abrir dicho archivo en Matlab, y ahí visualizar rodajas bidimensionales de cada una de las imágenes tridimensionales. Lo que nos interesa para la red son estas rodajas bidimensionales, en la figura 3.2 se muestran dos de estas rodajas correspondientes al archivo `F18_V2_ALL.raw`, elegidas aleatoriamente.

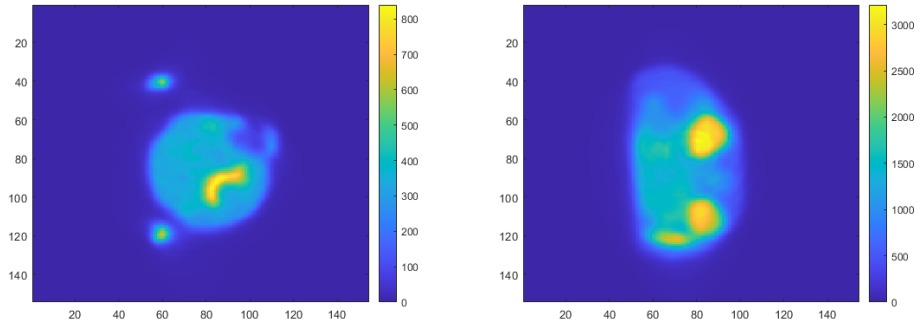


FIGURA 3.2: Dos rodajas bidimensionales, del primer caso (imagen tridimensional) del archivo F18_V2_ALL.raw, elegidas aleatoriamente.

Para la parte del *output* de nuestro conjunto de datos elegiremos aleatoriamente 25 rodajas de cada uno de los 8 casos, es decir, un total de 200 imágenes bidimensionales. Sin embargo, estas imágenes necesitan un procesamiento previo. En primer lugar, hace falta normalizar dichas imágenes entre 0 y 1, así el trabajo para la red será más sencillo. Esto lo hacemos dividiendo la matriz de 4 dimensiones (154 $\times$ 154 $\times$ 80 $\times$ 8) por su valor máximo. Así pues, todas las rodajas que extraigamos estarán normalizadas entre 0 y 1. Como en las imágenes la mayoría de los vóxeles tienen actividad nula, seleccionaremos únicamente la región de interés, es decir, la región en la que está contenida la actividad. Será una región de 16x16 píxeles. Elegimos una región tan pequeña, además, para disminuir el tiempo de cómputo, ya que no disponemos de GPUs para la ejecución del código. En la figura 3.3 se muestran dos ejemplos de imágenes ya procesadas que, ahora sí, ya forman parte del conjunto de datos que se va a utilizar. Estas imágenes las guardamos en forma matricial en un archivo con el nombre de `images.mat`. El código para la selección y el procesamiento de estas imágenes se muestra en el apéndice C.3 (código C.3).

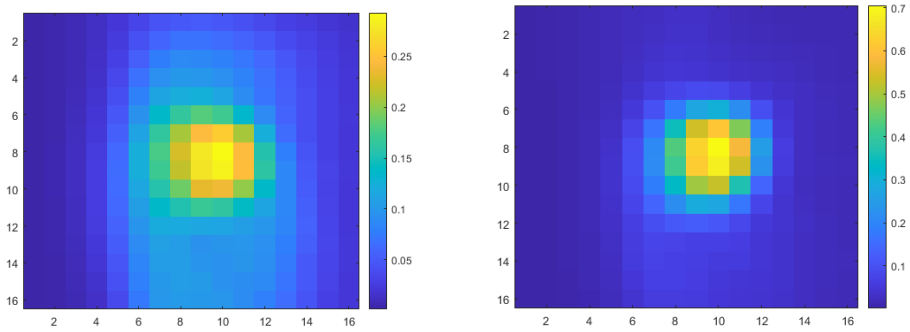


FIGURA 3.3: Dos imágenes de baja resolución, elegidas aleatoriamente, del archivo `images.mat` (la parte del *output* de nuestro conjunto de datos).

Para la parte del conjunto de datos que constituye el *input* utilizaremos la retroproyección de las imágenes del archivo `images.mat`, es decir, la primera iteración del algoritmo de reconstrucción descrito en la sección 3.5, ya que el objetivo de la red es evitar el proceso iterativo del algoritmo de reconstrucción, debido a su costo tanto computacional como temporal. Para ello, en primer lugar, introduciremos cada una de las imágenes del archivo `images.mat` (la parte del *output* de nuestro conjunto de datos) como el perfil de actividad de nuestra simulación (sección 3.4). Esta simulación, genera los datos necesarios con los que realizar la reconstrucción. Para realizar la simulación se ha elegido una actividad de $A_f = 1 \times 10^3$ Bq y un tiempo de medida de $t_m = 0.1$ s. Una vez se ha realizado la simulación, solo falta utilizar los datos proporcionados por ésta y utilizarlos para generar la primera iteración del algoritmo de reconstrucción, descrito en la sección 3.5. Este es un proceso costoso (temporalmente), ya que es necesario realizarlo para las 200 imágenes del archivo `images.mat` y al algoritmo de reconstrucción se caracteriza por ser lento. En concreto, para nuestro conjunto de datos, el código tardó unas 35 horas en ejecutarse. Estas imágenes retroproyectadas se guardan en forma matricial en un archivo con el nombre de `back_images.mat`. El código utilizado para generar estas imágenes se muestra en el apéndice C.3 (código C.4). La retroproyección de cada una de las imágenes de la figura 3.3 se muestra en la figura 3.4. Nótese que estas imágenes no se encuentran normalizadas entre 0 y 1, cosa que tendremos que hacer antes de utilizarlas para entrenar la red. Esta normalización la haremos ya en Python utilizando el código 3.7.

```
data_backimages = h5py.File('back_images.mat','r')
input = data_backimages.get('back_images')
input = np.array(input)

input /= input.max()
```

CÓDIGO 3.7: Normalización del *input* de la red.

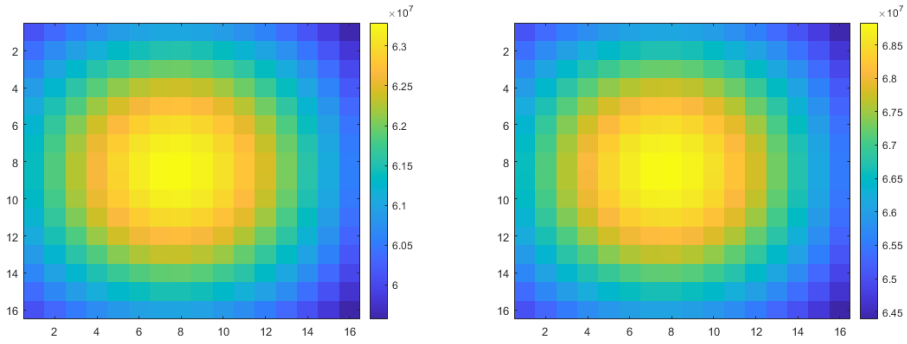


FIGURA 3.4: Retroproyección de las imágenes de la figura 3.3 (imágenes correspondientes al archivo `back_images.mat`.)

Las primeras 3 líneas del código 3.7 cargan el archivo `back_images.mat` como un array de Numpy. En la línea 5 se aplica la normalización, dividiendo cada uno de los píxeles de todas las imágenes por el valor máximo de todos los píxeles de las 200 imágenes retroproyectadas.

Como se puede apreciar en la figura 3.4, las imágenes retroproyectadas son muy similares entre sí. Esto parece indicar que la retroproyección no contiene la información suficiente que la red necesita para realizar la reconstrucción de la imagen correctamente.

Aumento de datos

En el campo del *Deep Learning*, un conjunto de datos con 200 imágenes para entrenar a la red es un conjunto relativamente pequeño. Cuando dispones de una cantidad de datos limitados, es común utilizar una técnica conocida con el nombre de Aumento de datos (o *Data augmentation* en inglés), la cual ha demostrado una mejora en los resultados obtenidos tras su aplicación, en especial para CNNs (Taylor y Nitschke, 2019). La técnica de Aumento de datos infla el conjunto de datos de entrenamiento, haciendo algún tipo de transformación a las imágenes originales. En concreto, para nuestro caso, el tipo de Aumento de datos que utilizaremos serán reflexiones tanto verticales como horizontales. La aplicación de Aumento de datos a nuestro conjunto de datos de entrenamiento se muestra en el código 3.8. Donde, en la primera línea de código, se importa la función `get_augmented` de la librería Keras, la cual utilizaremos para realizar el Aumento de datos. Después, en las líneas 3 y 4, se especifican los parámetros de entrada de dicha función, para generar reflexiones tanto horizontales como verticales.

```
1 from keras_unet.utils import get_augmented
2
3 train_gen = get_augmented(x_train_tf, y_train_tf, batch_size=32,
```

```
4    data_gen_args = dict(rotation_range=0.0, height_shift_range=0.,
     shear_range=0, horizontal_flip=True,vertical_flip=True))
```

CÓDIGO 3.8: Aumento de datos de nuestro conjunto de entrenamiento, con reflexiones verticales y horizontales, en Python.

3.7. Google Colab

El código para la CNN de la sección 3.6 lo escribiremos y ejecutaremos en *Google Colaboratory*. *Google Colaboratory* o *Google Colab*, es un documento ejecutable que permite escribir, ejecutar y compartir código de Python mediante *Google Drive*. Se podría decir que es un *Jupyter Notebook* almacenado en *Google Drive*. Un cuaderno (o *notebook* en inglés) de *Google Colab* está compuesto por celdas, que pueden contener código, texto, imágenes y más. Esto hace que esta herramienta sea muy adecuada para códigos explicativos y/o didácticos, o que vayan a ser compartidos, ya que utilizar celdas de texto (en las que se pueden añadir títulos, listas e incluso expresiones matemáticas) resulta de gran ayuda para proporcionar una narrativa alrededor del código. Existe también la posibilidad de exportar directamente el *notebook* a *GitHub*. Los *notebooks* se almacenan en el formato estándar de *Jupyter Notebook*, por lo que éstos se pueden ver y ejecutar en *Jupyter Notebook*, *JupyterLab*, y otros sistemas compatibles.

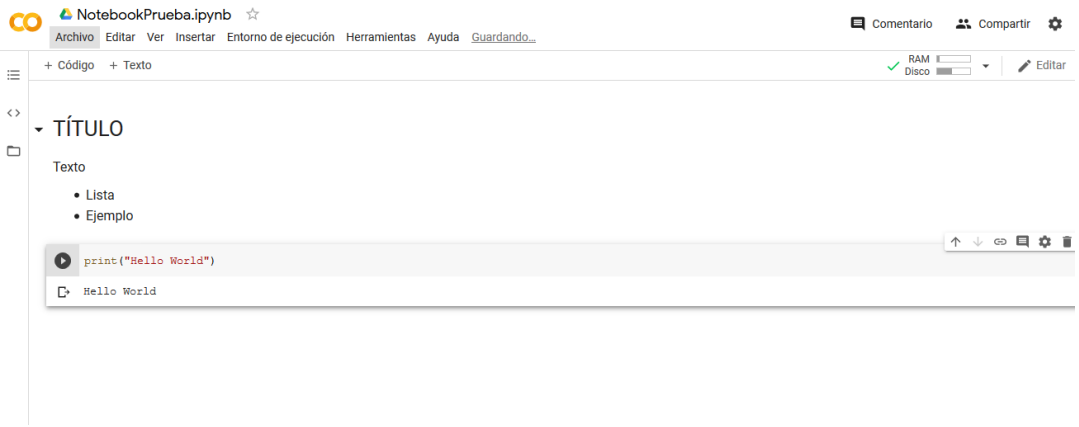


FIGURA 3.5: Interfaz de *Google Colab*

Una de las grandes ventajas que proporciona esta plataforma, es la posibilidad de configurar el entorno de ejecución de manera que el código sea ejecutado a través de la red, en máquinas que el propio *Google* pone a nuestra disposición. Para ello, hay que ir al desplegable «Entorno de ejecución» (que se muestra en la barra de herramientas de la figura 3.5) y seleccionar «Cambiar tipo de entorno de ejecución». Esto permite mejoras significativas en los tiempos de ejecución, sobre todo en el caso

de los algoritmos de *Deep Learning*, gracias a la opción de ejecutar el código en una GPU.

3.7.1. Código de la U-NET

En el apéndice B se explica la estructura y el funcionamiento de una red tipo U-NET. Para implementar la U-NET en Python es muy sencillo, ya que contamos con la función `custom_unet` de la librería Keras que nos permite, especificando ciertos parámetros, definir una red U-NET personalizada. La implementación de la red en *Google Colab* se muestra en el código 3.9.

```
1  from keras_unet.models import custom_unet
2
3  model = custom_unet(
4      input_shape=(Nx, Ny, x_train_tf.shape[3]),
5      use_batch_norm=True,
6      activation='swish',   #SWISH PROVIDES BETTER RESULTS THAN RELU
7      filters=16,
8      num_layers=2,
9      use_attention=False,
10     dropout=0.05,
11     output_activation='relu')  #RELU IN THE OUTPUT (POSITIVE BUT NOT
       LIMITED TO [0..1])
```

CÓDIGO 3.9: Construcción de la red U-NET personalizada, en Python.

En la primera línea de código se carga la función `custom_unet` de la librería Keras para después, en la línea 3, definir el modelo llamando a dicha función. A la hora de llamar a la función `custom_unet` se definen una serie de parámetros de la red, como el tamaño de nuestro *input*, las funciones de activación a utilizar, y el tamaño de la red, es decir, número de filtros y número de capas.

La función de activación utilizada en nuestra red, como se puede ver en el código 3.9, es la función *Swish*, que matemáticamente tiene la siguiente expresión:

$$swish(x) = \frac{x}{1 + e^{-\beta x}} \tag{3.14}$$

donde β es, o una constante, o un parámetro entrenable. Aunque la función de activación *ReLU* es la más usada comúnmente, hemos elegido la *Swish* debido a que se ha demostrado que proporciona mejores resultados en CNNs (Ramachandran, Zoph y Le, 2017). Para el *output* de la red, en cambio, utilizamos como función de activación la función *ReLU* (ecuación 1.10), tal y como se puede ver en el código 3.9.

Capítulo 4

Resultados

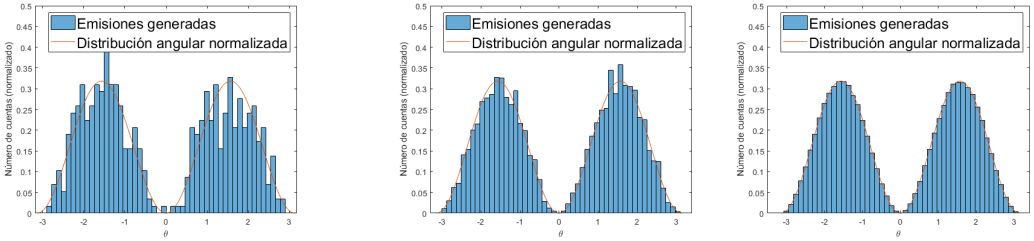
En este capítulo se muestran los resultados de una serie de distintos casos relevantes para el modelado de la emisión anisotrópica de rayos γ. En primer lugar, se simula, únicamente, el ángulo de emisión de estos rayos γ para una única fuente. En el siguiente caso, se toma en cuenta que los rayos γ se emiten en un tiempo dado y modelamos el tiempo entre medidas, que sigue la estadística de un proceso de Poisson. Después, se simula también la interacción de la precesión del emisor con el campo magnético y con los gradientes aplicados. Por último, se realiza la simulación con más de una fuente, es decir, más de un emisor.

Además de analizar las simulaciones, en este capítulo, se analiza también la reconstrucción con el algoritmo ML-EM de dos distribuciones de actividad distintas, una unidimensional y una bidimensional. Por último, se entrena a una CNN tipo U-NET (sección B) para a realizar la reconstrucción de la imagen, y analizaremos los resultados obtenidos con dicha red.

4.1. Generación de rayos de forma anisotrópica, caso estático

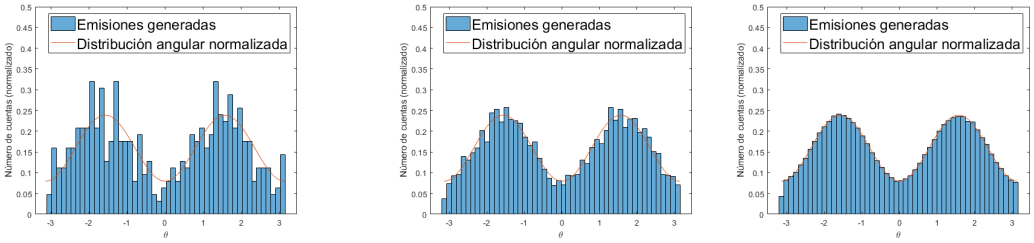
En esta sección presentaremos y discutiremos los resultados obtenidos para distintas simulaciones realizadas con los métodos presentados en la sección 3.1, utilizando el código del apéndice C.1. Analizaremos distintos casos, para distintos valores del parámetro a_2 de la ecuación 3.2, y para distintos valores de la actividad de la fuente, A_f, de la ecuación 3.3. Los valores para el parámetro que mide el grado de polarización, a_2, que analizaremos serán 0, 0.5 y 1. Para la actividad, estudiaremos también 3 valores distintos (en Bq): 500, 5×10^3 y 5×10^5. El tiempo de medida, t_m, será para todos los casos de 1 s.

Los resultados obtenidos para las distintas simulaciones se muestran en las



(a) $A_f = 500$ Bq. (b) $A_f = 5 \times 10^3$ Bq. (c) $A_f = 5 \times 10^5$ Bq.

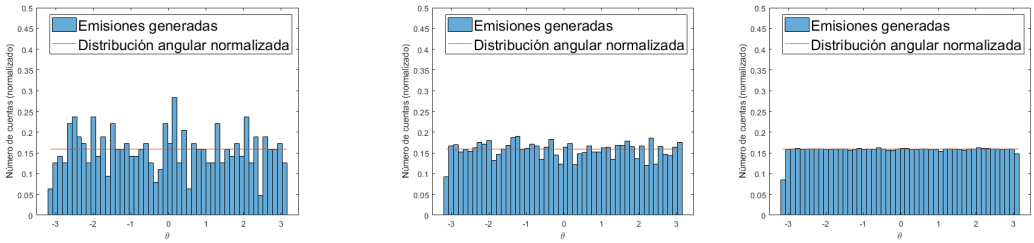
FIGURA 4.1: Histograma de los ángulos de las emisiones simuladas, para un valor del parámetro $a_2 = 1$ y para distintos valores de la actividad de la fuente, A_f, junto a la PDF (ecuación 3.6)



(a) $A_f = 500$ Bq. (b) $A_f = 5 \times 10^3$ Bq. (c) $A_f = 5 \times 10^5$ Bq.

FIGURA 4.2: Histograma de los ángulos de las emisiones simuladas, para un valor del parámetro $a_2 = 0{,}5$ y para distintos valores de la actividad de la fuente, A_f, junto a la PDF (ecuación 3.6)

figuras 4.1, 4.2 y 4.3. Como podemos observar en las figuras 4.1a, 4.2a y 4.3a, para una actividad baja ($A_f = 500$) el histograma presenta fluctuaciones estadísticas, ya que un número pequeño de emisiones implica una estadística pequeña. Para estos casos no podemos confirmar que la distribución de las emisiones generadas siga la ecuación 3.6, aunque si podemos ver que sigue su tendencia. Sin embargo, a medida que aumenta la actividad, aumenta también el número de emisiones detectadas, lo que implica una mejor estadística y debería dar lugar a menos fluctuaciones estadísticas. Esto se ve claramente para una actividad de $A_f = 5 \times 10^5$ Bq, en las figuras 4.1c, 4.2c y 4.3c, donde podemos apreciar que el histograma de los ángulos de las emisiones simuladas encaja de manera perfecta con la PDF de la ecuación 3.6. Esto nos confirma que el método utilizado para generar números aleatorios que sigan una cierta distribución de probabilidad, *Rejection sampling* (expuesto en la sección 3.1), funciona perfectamente.



(a) $A_f = 500$ Bq. (b) $A_f = 5 \times 10^3$ Bq. (c) $A_f = 5 \times 10^5$ Bq.

FIGURA 4.3: Histograma de los ángulos de las emisiones simuladas, para un valor del parámetro $a_2 = 0$ (emisión isótropa) y para distintos valores de la actividad de la fuente, A_f, junto a la PDF (ecuación 3.6)

4.2. Caso dinámico

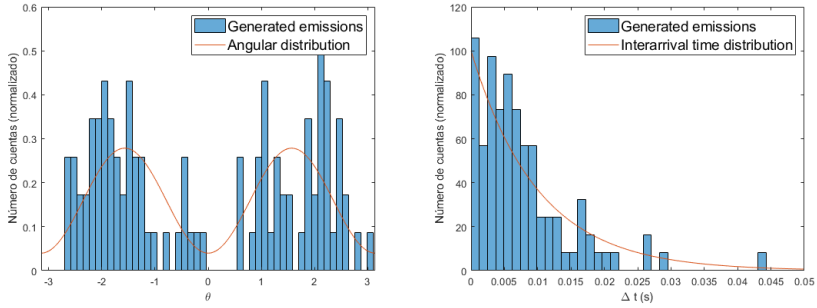
En esta sección presentaremos los resultados obtenidos para distintas simulaciones realizadas con los métodos presentados en la sección 3.2. En este caso, a diferencia de lo hecho en la sección 4.1, analizaremos un solo valor para el parámetro a_2, ya que, en los resultados de la sección 4.1, hemos visto que la simulación es consistente para distintos valores de dicho parámetro. Utilizaremos un valor de $a_2 = 0{,}75$ de aquí en adelante, para el resto de los resultados. Para la actividad estudiaremos, en este caso, 4 valores distintos (en Bq): 100, 500, 5×10^3 y 5×10^5.

Como podemos ver en las figuras 4.4a, y 4.4b (especialmente en la primera) para valores bajos de actividad, la estadística es demasiado baja, y no podemos asegurar que la distribución tanto de las emisiones generadas como de los tiempos entre medidas sigan las ecuaciones 3.6 y 3.7, respectivamente. Sin embargo, a medida que aumentamos la actividad de la fuente, se tiene una mejor estadística, y como podemos ver en la figura 4.4d dichas distribuciones siguen de manera perfecta las ecuaciones correspondientes.

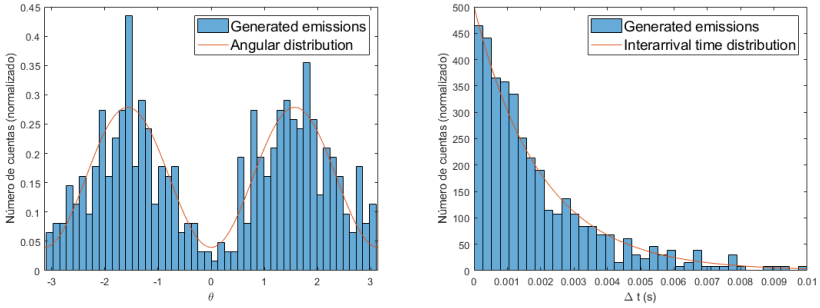
4.3. Efecto del ruido de Poisson en la distribución angular

En esta sección, vamos a estudiar, viendo los resultados obtenidos de la simulación realizada con los métodos de la sección 3.3, cómo afecta el ruido de Poisson generado al simular el tiempo de cada evento (sección 3.2) al ángulo de precesión, θ_0, del emisor, y por ende, al ángulo de emisión de cada evento.

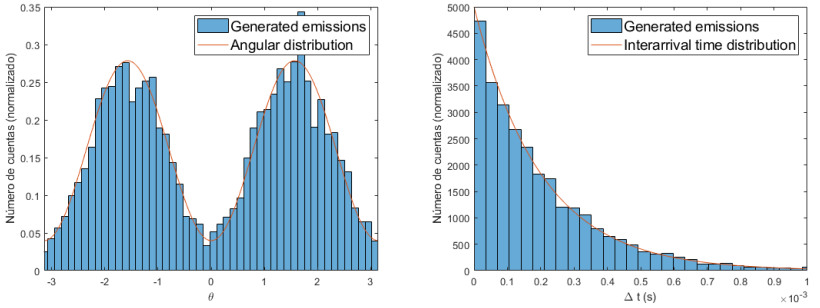
Para ello realizaremos dos simulaciones distintas: una con baja estadística para tener ruido, utilizando una baja actividad, de $A_f = 1 \times 10^3$ Bq y un tiempo de adquisición de $t_m = 1$ s; y otra con buena estadística para tener la menor cantidad



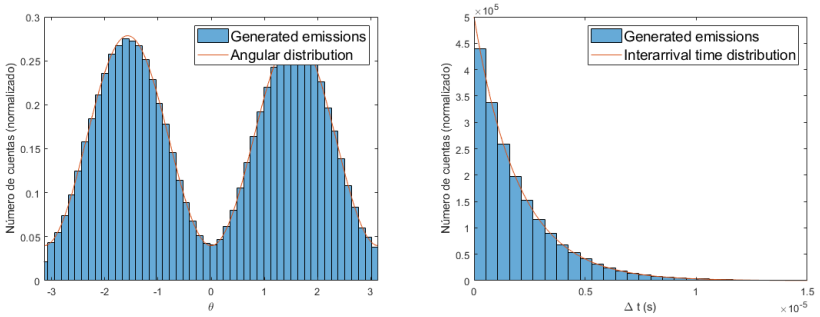
(a) $A_f = 100$ Bq.



(b) $A_f = 500$ Bq.



(c) $A_f = 5 \times 10^3$ Bq.



(d) $A_f = 5 \times 10^5$ Bq.

FIGURA 4.4: Resultados de la sección 4.1 para un valor del parámetro $a_2 = 0{,}75$, y para distintos valores de la actividad de la fuente, A_f. **Izquierda:** Histograma de los ángulos de las emisiones simuladas, junto a la PDF (ecuación 3.6). **Derecha:** Histograma de los tiempos entre medidas generados, junto a la distribución exponencial (ecuación 3.7).

de ruido posible, utilizando una actividad alta, de $A_f = 5 \times 10^6$ Bq y un tiempo de adquisición de $t_m = 1$ s. Después, haremos dos histogramas con los ángulos θ_0 obtenidos en ambas simulaciones, y así poder comparar el caso ideal (sin ruido) y el caso con ruido. Ambos histogramas estarán normalizados, utilizando el código 3.2, para que los resultados se puedan comparar más fácilmente. Dichos histogramas se muestran en la figura 4.5.

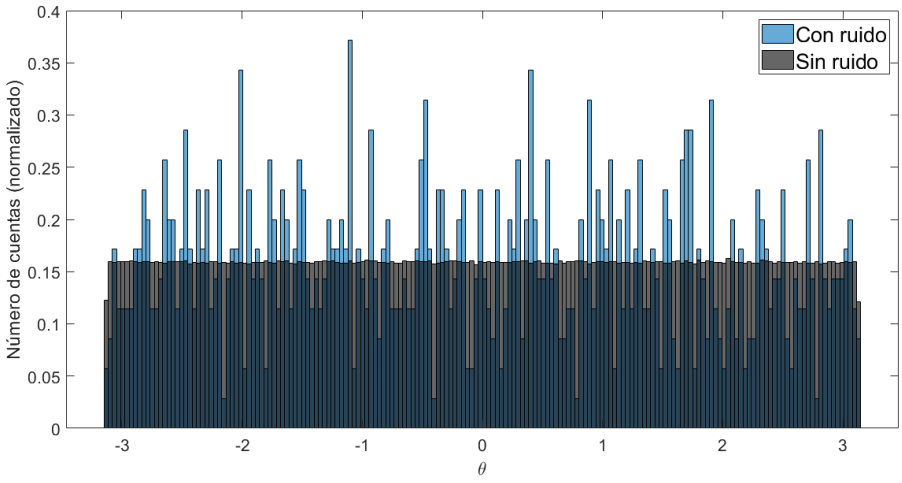


FIGURA 4.5: Histograma de los ángulos generados, teniendo en cuenta la precesión del emisor, para una actividad baja ($A_f = 1 \times 10^3$ Bq) en azul, y para una actividad alta $A_f = 5 \times 10^6$ Bq en negro.

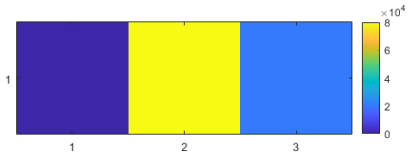
Para realizar los histogramas, hemos tomado un paso de 2 grados (cada compartimento del histograma abarca 2 grados). Podemos suponer que disponemos de un escáner con unos 300 o 360 cristales por anillo, lo que daría lugar a un paso de $\approx$ 1 grado en el histograma. Sin embargo, para que el histograma se pueda visualizar correctamente, hemos elegido un paso de 2 grados. Como podemos ver en la figura 4.5 el ruido de Poisson tiene un efecto considerable sobre la distribución angular, en el caso de tener una baja actividad.

4.4. Reconstrucción de la imagen con el algoritmo ML-EM

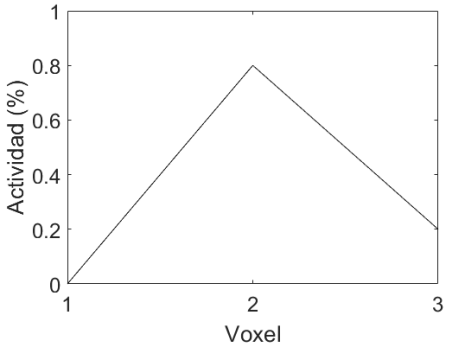
Realizaremos la reconstrucción de la distribución de actividad haciendo uso del algoritmo ML-EM (sección 3.5) a partir de los datos proporcionados por la simulación, para dos casos distintos: un caso unidimensional en el que tenemos 3 únicos vóxeles, la imagen por lo tanto será de 3×1 píxeles; y un segundo caso bidimensional en el que se tienen 7×7 vóxeles, es decir, la imagen tendrá un total de 49 píxeles.

4.4.1. Caso unidimensional

Empezaremos primero analizando un caso sencillo, unidimensional, de únicamente 3 vóxeles en el eje x. La simulación para esta sección se ha realizado con una actividad de $A_f = 1 \times 10^5$ Bq y un tiempo de adquisición de $t_m = 0.1$ s. Al primer vóxel le hemos asignado una actividad del 0 % sobre la actividad total ($0 \cdot A_f$), al segundo vóxel una actividad del 80 % ($0{,}8 \cdot A_f$), y al tercer y último vóxel una actividad del 20 % ($0{,}2 \cdot A_f$). La distribución de actividad recién descrita se muestra en la figura 4.6. Al tratarse de una simulación unidimensional, la distribución de actividad la podemos representar como una imagen (figura 4.6a) o como el perfil de intensidad de la actividad (figura 4.6b). Nótese que en la figura 4.6b, la actividad se encuentra normalizada a la actividad total de la simulación, por lo tanto, lo que se ha dibujado en dicha gráfica son los porcentajes de actividad en cada vóxel respecto a la actividad total.



(a) Actividad vista como una imagen.



(b) Actividad vista como un perfil de intensidad, donde la actividad en cada vóxel está dada como porcentaje sobre la actividad total.

FIGURA 4.6: Distribución de actividad para el caso unidimensional.

En la simulación realizada se han medido un total de 10 185 eventos, lo cual concuerda con la actividad y el tiempo de adquisición indicados anteriormente. Las fluctuaciones se deben al ruido de Poisson generado por el tiempo entre medidas (sección 3.2).

Una vez se ha realizado la simulación, podemos utilizar los datos obtenidos para realizar la reconstrucción de la imagen con el algoritmo ML-EM (sección 3.5). El número de iteraciones elegido para ejecutar el algoritmo ha sido de 200 iteraciones. Las imágenes reconstruidas para distintos números de iteraciones se muestran en la figura 4.7. Como podemos ver, el algoritmo logra reconstruir la imagen de manera satisfactoria. A simple vista, podríamos decir que el algoritmo converge para las 100 iteraciones (la imagen reconstruida con 100 iteraciones es indistinguible a la figura

4.6a a simple vista).

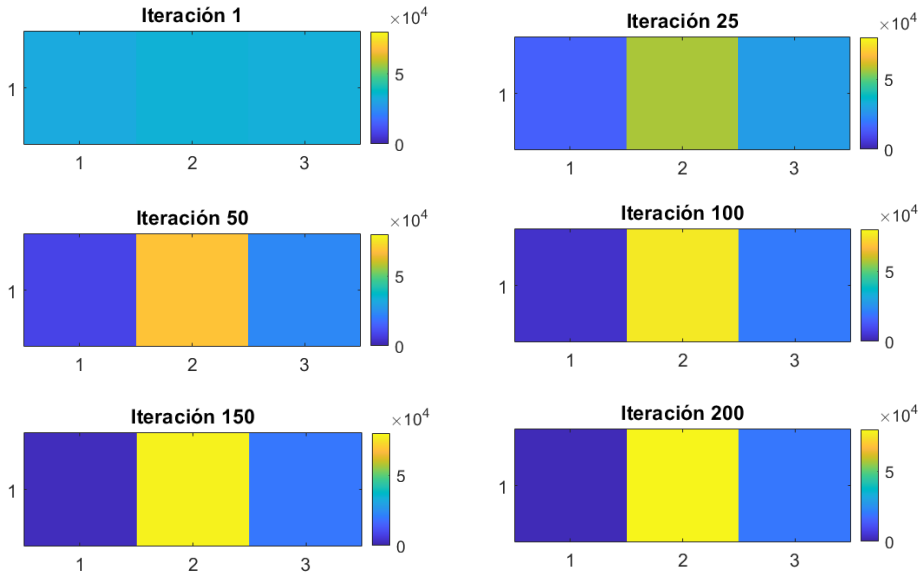


FIGURA 4.7: Imagen del caso unidimensional reconstruida con el algoritmo ML-EM (sección 3.5) para distintos números de iteraciones.

En vez de ver la convergencia "a simple vista", podemos estudiarla más al detalle. Para ello, comparamos la imagen reconstruida en cada iteración con la imagen original (figura 4.6a). Para comparar las imágenes, utilizaremos el error cuadrático medio (ECM), que viene dado por la expresión:

$$ECM = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2 \tag{4.1}$$

En la figura 4.8 se muestra representado el error cuadrático medio obtenido al comparar la imagen reconstruida con la imagen original para cada iteración del algoritmo de reconstrucción. Mirando a dicha gráfica, podemos confirmar lo que habíamos visto a simple vista, y es que el algoritmo converge para 100 iteraciones, donde el error es prácticamente nulo. Si queremos ser más detallistas, podríamos llegar a decir que la convergencia se obtiene para 150 iteraciones, ya que mirando con detalle la figura 4.8 podemos ver que para 100 iteraciones el error no se encuentra del todo sobre el eje de abscisas (eje x).

4.4.2. Caso bidimensional

El segundo caso que analizaremos se trata de una simulación de 7x7 vóxeles, es decir, una simulación bidimensional, en la que toda la actividad está concentrada en el vóxel central, tal y como se puede ver en la figura 4.9. La simulación de

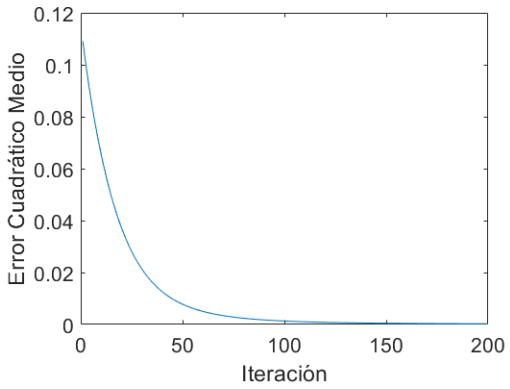


FIGURA 4.8: Error cuadrático medio de la imagen reconstruida respecto a la imagen original, en función del número de iteraciones (caso unidimensional).

esta sección se ha realizado con una actividad de $A_f = 1 \times 10^3$ Bq y un tiempo de adquisición de $t_m = 0.1$ s. Se han simulado un total de 4915 eventos.

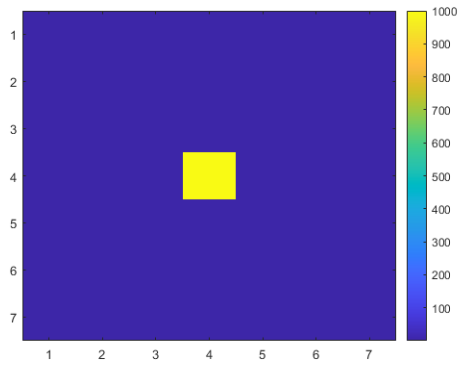


FIGURA 4.9: Distribución de actividad para el caso bidimensional.

Con los datos obtenidos a partir de la simulación, hemos realizado la reconstrucción de la imagen, haciendo uso del algoritmo ML-EM (sección 3.5), donde hemos elegido aplicar 150 iteraciones. Las imágenes reconstruidas para distintos números de iteraciones se muestran en la figura 4.10. Como podemos ver el algoritmo logra reconstruir la imagen de manera satisfactoria. A simple vista, podríamos decir que el algoritmo alcanza la convergencia para 25 iteraciones.

Para estudiar la convergencia más al detalle, hemos pintado, al igual que para el caso unidimensional, el error cuadrático medio (ecuación 4.1) en función del número de iteraciones. Esto se muestra en la figura 4.11, tanto en escala lineal como en escala logarítmica, para las primeras 50 iteraciones. Mirando al error cuadrático medio en escala lineal, podemos ver que nuestra estimación de 25 iteraciones para la convergencia es correcta. De hecho, mirando a dicha gráfica se podría decir que la convergencia se alcanza para 15 iteraciones. Sin embargo, mirando a la figura 4.10,

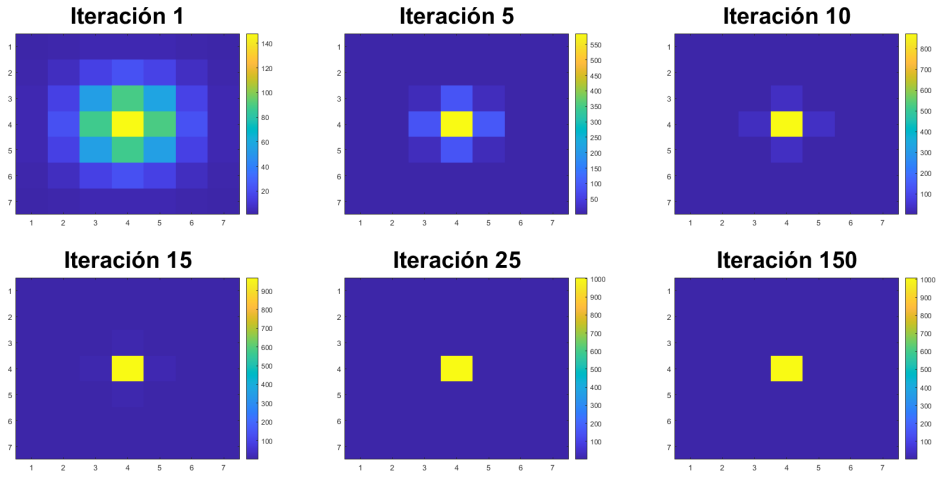


FIGURA 4.10: Imagen del caso bidimensional reconstruida por el algoritmo ML-EM (sección 3.5) para distintos números de iteraciones.

se puede ver, a simple vista, que hay diferencias entre la imagen reconstruida a 15 iteraciones y la imagen original (figura 4.9) en los cuatro vóxeles contiguos al vóxel central. Es por eso por lo que hemos representado también el error cuadrático medio en escala logarítmica, ya que ahí podemos ver que, realmente, la convergencia no se alcanza hasta las 30 iteraciones.

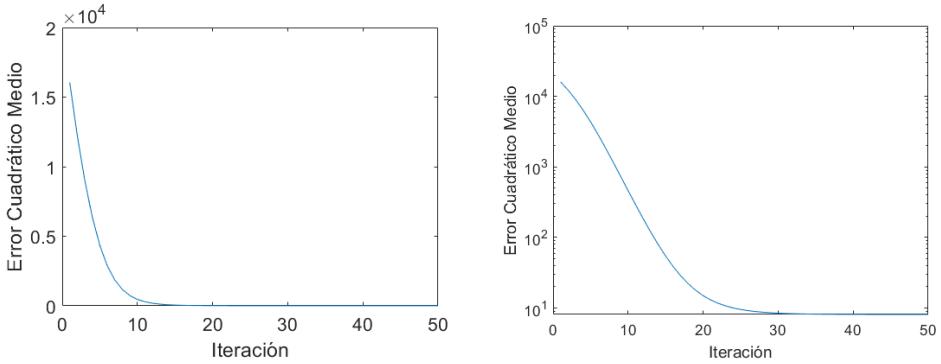


FIGURA 4.11: Error cuadrático medio de la imagen reconstruida respecto a la imagen original, en función del número de iteraciones (caso bidimensional). **Derecha:** en escala lineal. **Izquierda:** en escala logarítmica.

La reconstrucción que se muestra en esta sección, realizada ejecutando el código C.2, tardó un total de 55 segundos. Para ser un caso muy simple, y una resolución muy baja, 7×7, es un tiempo de reconstrucción considerablemente alto, debido a que, en esta modalidad de Gamma-MRI, todos los píxeles están conectados con una cierta probabilidad con cada una de las detecciones.

4.5. Reconstrucción con una CNN

Con el objetivo de disponer de un método más rápido para realizar la reconstrucción de imagen a partir de medidas con la técnica Gamma-MRI, se exploró el posible uso de redes neuronales. El código completo de la CNN utilizado para obtener los resultados de esta sección se muestra en el código D.1 del apéndice D.

Una vez hemos definido el modelo (código 3.9), podemos obtener un resumen de la estructura de nuestro modelo, ejecutando el código 4.1. El resumen de nuestro modelo se muestra en la figura E.1, del Apéndice E. De este resumen podemos destacar que nuestro modelo consta de 117 713 parámetros, de los cuales 117 073 son parámetros entrenables, y 640 son parámetros no entrenables.

```
1 model.summary()
```

CÓDIGO 4.1: Código para obtener un resumen del modelo previamente definido.

Antes de entrenar el modelo, mostramos como ejemplo dos casos distintos del conjunto de datos, para hacernos una idea de la tarea que la red debe aprender. Para cada caso mostraremos la imagen original, es decir, lo que la red utilizará para comparar con su predicción a la hora de entrenarse, y la imagen retroproyectada, es decir, el *input* de la red a partir del cual tiene que realizar la predicción. Ambos casos se muestran en la figura 4.12. Como podemos observar las imágenes retroproyectadas son muy similares entre sí, por lo que la tarea que debe resolver la red no es trivial, y podríamos decir que es sobrehumana, ya que a simple vista no podemos decir cuál de las imágenes retroproyectadas corresponde a cada imagen original. Que las imágenes retroproyectadas sean tan similares entre sí puede deberse, entre otras cosas, a lo comentado en la sección 3.6.1. Es decir, una baja actividad y un bajo tiempo de adquisición a la hora de realizar la simulación, debido a límites en el poder computacional.

Para entrenar a la red hace falta elegir una función de error, que la red utilizará para comparar sus predicciones con las imágenes de referencia. En nuestro caso, la métrica elegida es el error absoluto medio, que viene dado por la siguiente expresión:

$$EAM = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i| \tag{4.2}$$

Para realizar el entrenamiento de la red, no hay más que ejecutar el código 4.2:

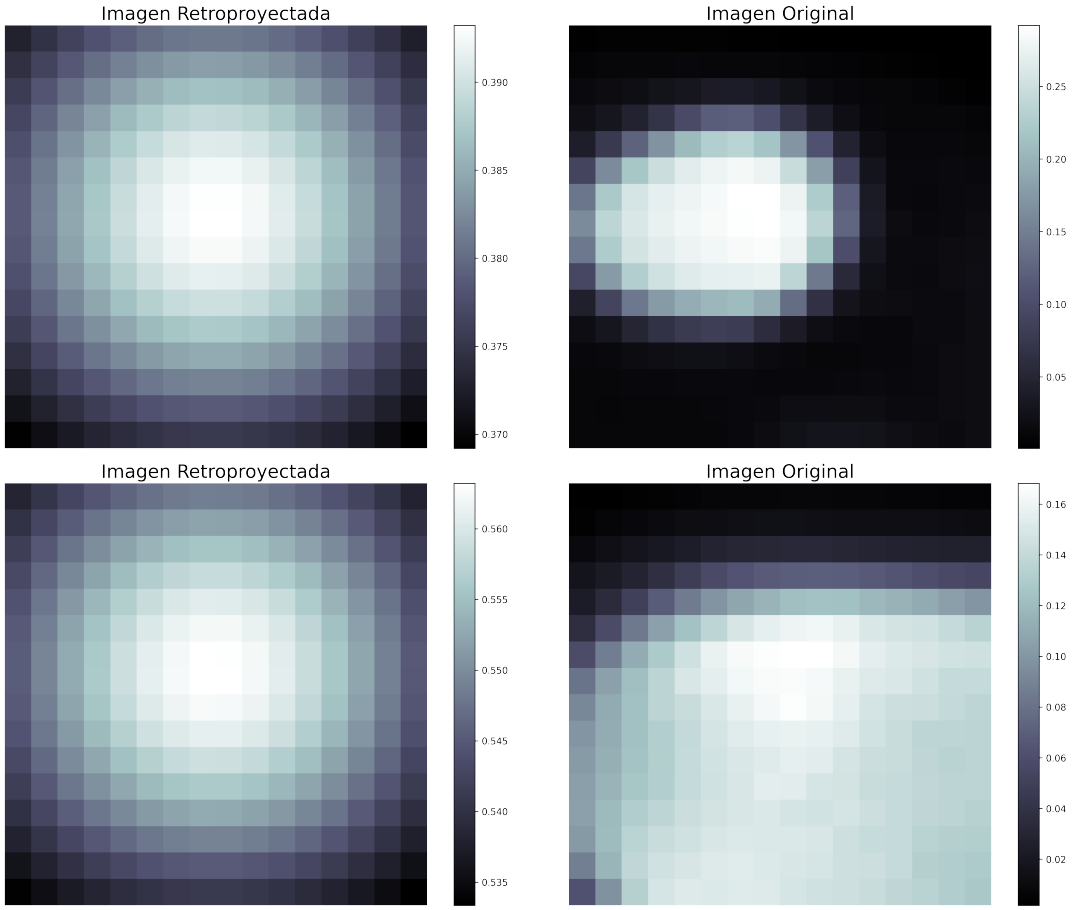


FIGURA 4.12: Dos casos distintos, elegidos aleatoriamente, de nuestro conjunto de datos. **Izquierda:** Imagen retroproyectada, es decir, el *input* de nuestra red. **Derecha:** Imagen original, es decir, la imagen con la que la red comparará su predicción a la hora de realizar el entrenamiento.

```
history = model.fit(train_gen,steps_per_epoch=100, epochs=50,
    validation_data=(x_val_tf, y_val_tf))
```

CÓDIGO 4.2: Entrenamiento de la red.

donde, hemos seleccionado 50 épocas de entrenamiento, y 100 pasos o *steps* por época. Una época significa un ciclo completo por el conjunto de datos de entrenamiento. Un paso de entrenamiento significa una actualización del algoritmo del descenso del gradiente (sección A.2.1). El entrenamiento es relativamente rápido para nuestra red, debido a que el conjunto de datos es relativamente pequeño, 200 imágenes, y a que utilizamos como entorno de ejecución una GPU proporcionada por *Google Colab*, tal y como mencionamos en la sección 3.7. En nuestro caso, el entrenamiento, tarda entre uno y dos minutos en completarse.

Una vez se ha completado el entrenamiento, ya podemos utilizar nuestro modelo para realizar predicciones a partir del conjunto de datos de validación (casos

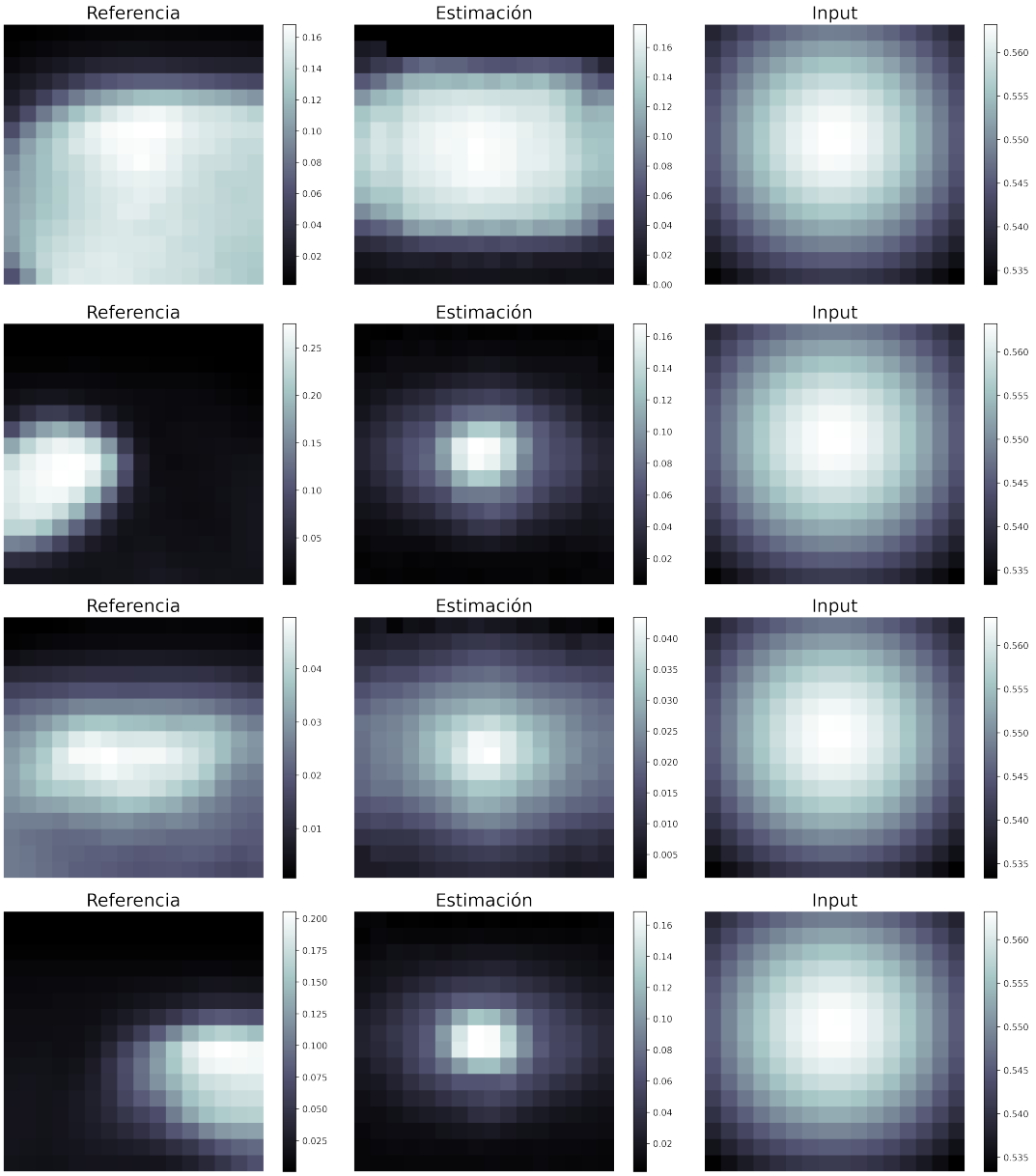


FIGURA 4.13: Resultados de nuestro modelo para 4 casos distintos. **Izquierda:** Imagen original o de referencia. **Centro:** Predicción que nuestro modelo ha realizado. **Derecha:** Imagen retroproyectada, es decir, el *input* de la red.

que no ha visto antes y con los que no se ha entrenado) y ver qué tal se desenvuelve realizando la reconstrucción de la imagen. Los resultados obtenidos se muestran en la figura 4.13, es decir, se muestra la predicción de la red para distintos casos, junto a la imagen retroproyectada y a la imagen original o de referencia para cada caso. Como podemos observar en dicha figura, los resultados no son ideales. La red no es capaz de realizar la reconstrucción de la imagen de forma satisfactoria. Aun así, podemos decir que la predicción de la red no es aleatoria y que ha aprendido algo, debido a que al realizar la predicción la red es capaz de reconocer si la actividad está más localizada o extendida. Este resultado se podría mejorar disponiendo de un

mayor poder computacional, ya que se podría generar un conjunto de datos mayor y una mejor calidad (mayor resolución).

Con la red ya entrenada, podemos ver también cómo ha ido evolucionando el error en cada época de entrenamiento. Esto se muestra en la figura 4.14, en la cual se han pintado tanto el error de entrenamiento (el error en el conjunto de datos de entrenamiento) como el error de validación (el error en el conjunto de datos de validación). Se puede ver claramente que ambos errores van de la mano, es decir, disminuyen de forma pareja, ambos a la vez. Esto nos indica que la red ha aprendido a generalizar el conocimiento aprendido, y no está haciendo un sobreajuste, o *overfitting* en inglés. El concepto de *overfitting* hace referencia al hecho de que la red se aprenda los casos del conjunto de entrenamiento de "memoria", es decir, que solo sepa resolver los casos del conjunto de entrenamiento y no sea capaz de hacer predicciones con un error bajo para el conjunto de validación. Es por esto, que decimos que la red ha sido capaz de generalizar el conocimiento aprendido. También se puede ver que ambos errores disminuyen considerablemente al principio del entrenamiento. Posteriormente, la disminución del error es bastante lenta, por lo que podemos afirmar que el modelo ha llegado a un mínimo local haciendo uso del algoritmo del descenso del gradiente.

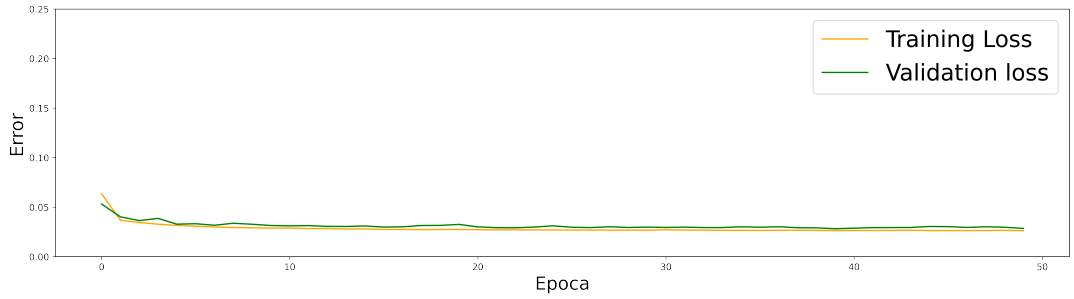


FIGURA 4.14: Evolución de tanto el error de entrenamiento, como el error de validación en función de la época de entrenamiento.

Capítulo 5

Discusión

Los resultados obtenidos de la simulación y de la reconstrucción (tanto con el algoritmo ML-EM, como con la red neuronal) de señales de Gamma-MRI son prometedores. Existe una serie de aspectos para tener en cuenta sobre ellos.

Respecto a los datos utilizados para entrenar la red neuronal (sección 3.6.1) hay que destacar la contribución que se ha hecho al trabajo con su aportación. Se trata de una adquisición PET de un animal pequeño haciendo uso del radioisótopo ^{18}F y de la molécula Fluordesoxiglucosa (^{18}F-FDG). Los datos se han obtenido de un artículo sobre la corrección del rango de los positrones en imágenes PET haciendo uso de técnicas de *Deep Learning* (Herraiz, Bembibre y López-Montes, 2021).

Como podemos ver en la sección 4.4, los resultados obtenidos para la reconstrucción de la imagen con el algoritmo ML-EM son satisfactorios. Se ha conseguido reconstruir con éxito dos distribuciones de actividad, una unidimensional de únicamente 3 vóxeles, y una bidimensional, con un total de 49 vóxeles. Para ambos casos, la convergencia se alcanza para un número de iteraciones razonable. Por lo tanto, se ha demostrado que el algoritmo ML-EM funciona y converge para esta nueva modalidad de imagen. Sin embargo, hay que recordar que, como dijimos en el capítulo 2, estas simulaciones no son más que una prueba de concepto, y sirven como punto de partida para elaborar simulaciones más realistas y complejas, por lo que el trabajo realizado tiene todavía mucho margen de mejora si los resultados se quieren utilizar para guiar los montajes experimentales de Gamma-MRI.

Para realizar las simulaciones se ha utilizado el lenguaje de programación Matlab. Se ha elegido Matlab debido a que es un lenguaje de programación sencillo y con el que ya estamos familiarizados, debido a su uso a lo largo de la carrera. No obstante, utilizar Matlab, da lugar a algunas limitaciones, debido a que hay lenguajes de programación más rápidos. Este sería uno de los puntos a mejorar, implementar las simulaciones en otro lenguaje de programación más eficiente. Otra manera

de mejorar la eficiencia computacional de las simulaciones sería haciendo uso de la computación paralela usando GPUs. Una de las principales limitaciones de nuestro código recae en que la matriz de probabilidades del sistema se encuentra almacenada en su totalidad en memoria, lo que implica que, para uno número alto de vóxeles, Matlab se queda sin memoria. Por lo tanto, es esencial para el proyecto Gamma-MRI buscar esquemas para aplicar este tipo de simulación con mayor eficiencia computacional.

Respecto a los resultados de la reconstrucción con una CNN, se ha demostrado que una simple retroproyección comprime demasiado la información de la señal y no sirve para que la red neuronal realice la reconstrucción con éxito. Por lo tanto, habría que buscar nuevos esquemas para organizar la señal y así dar a la red neuronal mayor cantidad de información. También, una de las principales limitaciones en los resultados de la red neuronal, recae en el poder de cálculo computacional. Es decir, con mayor poder computacional, y un código más eficiente para las simulaciones, se podría tener un conjunto de datos mucho mayor y de mejor calidad, además de tener más resolución en las imágenes. Como vimos en la sección 1.6.1, disponer de técnicas eficientes y de una gran cantidad de datos son algunos de los retos a los que se enfrenta la disciplina del *Deep Learning*.

Es importante destacar el uso de las GPUs que *Google Colab* pone a nuestra disposición para el entrenamiento de la red. El entrenamiento con GPUs de *Google Colab* tardaba alrededor de 1 a 2 minutos. Sin embargo, haciendo uso de la CPU del ordenador, el tiempo de entrenamiento se iba a las decenas de minutos. Esto nos ha permitido analizar con rapidez distintas arquitecturas para la U-NET. Se han probado casos con un mayor número de capas y un mayor número de filtros, sin embargo, aumentar estos parámetros no producía una mejora significativa en los resultados.

Como núcleo radiactivo emisor de rayos γ se ha analizado el caso del ^{131m}Xe, principalmente, porque es un isótopo fácilmente hiperpolarizable haciendo uso de la técnica de SEOP y porque es un gas que los humanos podemos inhalar. Se podrían utilizar otros nucleidos, sin embargo, es un campo que todavía requiere de estudio. Los requisitos para la búsqueda de nuevos nucleidos serían varios: que sea un núcleo radiactivo con estados metaestables y una vida media razonable; que tenga momento magnético de espín $> 1/2$ (para que la emisión de rayos γ sea anisótropa); y que sean fácilmente polarizables. La polarización de gases se puede hacer fácilmente con la técnica de SEOP, en cambio, si no se trata de un gas, hace falta utilizar la técnica de Polarización Dinámica Nuclear (Abragam y Goldman, 1978).

De cara a futuros trabajos, como ya hemos dicho, una de las principales tareas sería mejorar la eficiencia de las simulaciones. Tanto implementando el código en GPUs, como utilizando un lenguaje de programación más rápido. Para realizar la reconstrucción de la imagen con una red neuronal, hace falta buscar nuevas formas de suministrar la información de la señal a la red, ya que se ha demostrado que una retroproyección no es suficiente. También se podría ampliar la arquitectura del modelo, añadiendo una red generativa adversaria (GAN) que se encargarse de competir con nuestro modelo generador, la U-NET. Estas redes (las GAN) compiten con modelos generadores aprendiendo a distinguir una imagen generada por el modelo generador de una imagen real. Así, se consigue que el modelo generador produzca imágenes más parecidas a las reales.

Como principal contribución de este proyecto, se puede destacar el haber sentado las bases para realizar simulaciones realistas que guíen los montajes experimentales del proyecto Gamma-MRI. Se ha demostrado que el algoritmo clásico ML-EM funciona y converge. También, se ha aprendido que hacen falta nuevas formas de comprimir la información a la hora de realizar la reconstrucción con una red neuronal. Por tanto, se puede afirmar que los resultados obtenidos han cumplido los objetivos marcados inicialmente para este trabajo.

5.1. Conclusiones

En este trabajo, se han mostrado los principales fundamentos de Gamma-MRI. Se han realizado simples simulaciones que permiten hacer una primera estimación de cómo serán las señales de Gamma-MRI y como se podrían reconstruir. Estos eran los objetivos principales del trabajo, tal y como se ha contado en el capítulo 2.

Como se ha indicado, la simulación implementada es bastante sencilla, sin embargo, refleja las ecuaciones de emisión anisotrópica y de la precesión. De cara a futuros trabajos, para hacer las simulaciones más realistas, habría que implementar métodos Monte Carlo para simular el transporte y la detección de los rayos γ. También habría que implementar un simulador de MRI para tener en cuenta los tiempos de relajación tanto longitudinal, como transversal, T_1 y T_2, respectivamente, además de simular secuencias más realistas de pulsos de RF.

Se ha visto que la reconstrucción con el algoritmo ML-EM funciona, sin embargo, es un proceso lento y costoso desde el punto de vista computacional. En cambio, la red neuronal tiene potencial para ser capaz de realizar la reconstrucción de

forma mucho más rápida, aunque se ha demostrado que almacenar los datos de la señal en una imagen retroproyectada no es suficiente para entrenar a la red, debido a que, con una simple retroproyección, se pierde demasiada información sobre la señal.

Los resultados presentados en este trabajo abren caminos para evaluar nuevos métodos que logren superar las dificultades observadas.

Apéndice A

Tipos de aprendizaje y entrenamiento de una red neuronal

A.1. Tipos de aprendizaje

El aprendizaje en *Machine Learning* lo podemos separar en tres principales grupos: *aprendizaje supervisado*, *aprendizaje no-supervisado* y *aprendizaje reforzado*:

- El aprendizaje supervisado, se basa, fundamentalmente, en descubrir la relación existente entre unas variables de entrada y unas de salida. En otras palabras, el aprendizaje surge en este caso de enseñarle al algoritmo cuál es el resultado que quieres obtener para un determinado valor de entrada. Mostrando muchos ejemplos y con las condiciones necesarias, el algoritmo podrá ser capaz de dar un resultado correcto, incluso para valores de entrada que no había visto antes. Por ejemplo, con los siguientes pares de valores [1,5], [3,7], [10,14], [15,19], gracias a los procesos biológicos de aprendizaje que hay en nuestro cerebro, somos capaces de deducir la pareja que debería tener un nuevo valor que no hemos visto, como por ejemplo el 20, del cual su pareja será el 24. Ésta es una analogía que simplifica lo qué es el aprendizaje supervisado, pero es capaz de dar una intuición de su funcionamiento. Se le da la denominación de *supervisado* debido a que al mostrarle los resultados que queremos al algoritmo, participamos en la supervisión de su aprendizaje.
- El aprendizaje no supervisado, consiste en producir conocimiento a partir de únicamente los datos que se proporcionan como entrada, sin especificar en ningún momento al algoritmo cuál es el resultado que queremos obtener. Esto se consigue buscando patrones de similitud en los datos de entrada. Este método de aprendizaje tiene una ventaja respecto al anterior, y es que los conjuntos de datos para entrenar el algoritmo son más sencillos de conseguir. Esto se debe a

que en aprendizaje supervisado cada elemento del conjunto de datos debe ser etiquetado, típicamente por un humano, de forma manual y de uno en uno, con el resultado deseado. Si, además, tenemos en cuenta que estos conjuntos de datos suelen ser, generalmente, de miles de elementos, esto da lugar a una reducción del trabajo considerable.

- El aprendizaje reforzado, está relacionado con la forma en la que los agentes (los algoritmos) deben tomar decisiones en un entorno, para maximizar alguna noción de recompensa o premio acumulativo previamente definidos. Esta área del *Machine Learning* ha tenido grandes avances en los últimos años. Un claro ejemplo es el diseño de *biobots* (Kriegman y col., 2020), un organismo biológico creado célula a célula en el laboratorio, cuya estructura le permite desarrollar tareas sencillas. Lo interesante de esto es que esta estructura ha sido diseñada por un algoritmo de *Deep Learning* haciendo uso del aprendizaje reforzado. En concreto, por un algoritmo evolutivo (Sloss y Gustafson, 2019) ejecutándose en un superordenador.

Estos son los 3 tipos de aprendizaje más destacados y conocidos, pero existen más métodos como, por ejemplo, el *aprendizaje semi-supervisado*.

A.2. Aprendizaje y entrenamiento

En esta sección hablaremos del aprendizaje y del entrenamiento de una red neuronal multicapa desde el punto de vista del aprendizaje supervisado (explicado en el apéndice A.1), ya que es el tipo de aprendizaje que utilizamos a lo largo de este trabajo. Así pues, el aprendizaje lo podemos definir como la capacidad de adaptación de la red para realizar mejor la tarea propuesta a partir de considerar casos a modo de ejemplo. Es decir, el aprendizaje consiste en modificar los pesos y umbrales de la red (los parámetros) para mejorar la precisión del resultado optimizando los errores observados. Esto se hace definiendo una función de coste (o función de error) que se evalúa periódicamente durante el aprendizaje, y que sirve para evaluar la calidad de una predicción de la red. Es decir, la función de coste nos da el error en la predicción. La elección de la función de coste dependerá del problema a resolver, y actualmente es un amplio campo de investigación. Por lo tanto, un error bajo implica que la diferencia entre el *output* de la red y el resultado que queremos, el valor correcto, es pequeña. La mayoría de los modelos de aprendizaje pueden verse como una aplicación sencilla de la teoría de la optimización y la estimación estadística. En esta sección haremos una breve explicación de uno de los métodos más

usados (Geron, 2017) en el campo del *Deep Learning* para el aprendizaje, el descenso del gradiente combinado con la retropropagación (o *backpropagation* en inglés).

A.2.1. Descenso del gradiente

El descenso del gradiente es un algoritmo de optimización muy genérico, capaz de encontrar soluciones óptimas para una amplia variedad de problemas (Geron, 2017). La idea principal del método del descenso del gradiente es ajustar los parámetros de forma iterativa para minimizar una función de coste dada. El fundamento del método se puede explicar con un breve experimento mental. Suponga que está perdido en las montañas con una densa niebla, y no puede ver, solo puede sentir la pendiente del suelo debajo de sus pies. Una buena estrategia para llegar rápidamente a la parte baja de las montañas podría ser la siguiente: evaluar la pendiente para distintas direcciones, elegir la dirección en la que la pendiente sea mayor, y dar un pequeño paso en esa dirección. Repitiendo iterativamente estos pasos sería capaz de llegar a la parte baja de las montañas. Esto es exactamente lo que hace el algoritmo del descenso del gradiente: mide el gradiente local de la función de coste respecto al vector de parámetros y va en la dirección descendiente del gradiente. Una vez el gradiente vale cero, quiere decir que se ha alcanzado un mínimo de la función de coste.

Concretamente, el método comienza inicializando el vector de parámetros de forma aleatoria y luego trata de mejorarlo gradualmente, dando pequeños pasos, cada vez intentando disminuir la función de coste, hasta que el algoritmo converja a un mínimo. El parámetro más importante del algoritmo recibe el nombre de ratio de aprendizaje (o *learning rate* en inglés). El ratio de aprendizaje hace referencia al tamaño de los pasos que se dan a la hora de descender la montaña en la metáfora antes presentada. Si el ratio de aprendizaje es muy pequeño el algoritmo tendrá que realizar muchas iteraciones para converger. En cambio, si el ratio de aprendizaje es muy alto los pasos que da el algoritmo son tan largos que es incapaz de introducirse en la zona de mínimo coste. El algoritmo del descenso del gradiente, matemáticamente, no es más que aplicar la ecuación A.1 de forma iterativa hasta la convergencia. Donde θ es el vector de parámetros, α es el ratio de aprendizaje, y ∇f es el gradiente de la función de coste.

$$\theta := \theta - \alpha \nabla f \tag{A.1}$$

Algoritmo 1: Descenso del gradiente

while *La función de coste no esté en un mínimo* **do**
  $\theta := \theta - \alpha \nabla f$
end

A.2.2. *Backpropagation*

Durante muchos años, los investigadores, buscaron formas de entrenar los perceptrones multicapa, sin éxito alguno. Hasta 1986, año en el que vio a la luz un artículo innovador y rompedor (Rumelhart, Hinton y Williams, 1986), en el que se presentó el algoritmo de entrenamiento de retropropagación (o *backpropagation* en inglés). El algoritmo de *backpropagation* es utilizado como un método para calcular las derivadas parciales de la función de coste con respecto a cada uno de los parámetros de la red. Estas derivadas parciales son importantes, ya que componen el gradiente de la función de coste (∇f en la ecuación A.1), el cual es utilizado por el algoritmo del descenso del gradiente para optimizar la red.

Para cada instancia de entrenamiento, el algoritmo, en primer lugar, realiza una predicción. Después, calcula el error de dicha predicción (es decir, la diferencia entre el *output* deseado y el *output* real de la red), y computa cuánto ha contribuido cada neurona de la última capa oculta en el error de cada una de las neuronas de salida (neuronas de la capa de salida). Posteriormente, procede a computar cuánto de estas contribuciones al error provienen de cada neurona en la capa oculta anterior, y así sucesivamente, hasta que el algoritmo llegue a la capa de entrada. De esta forma, el algoritmo mide el gradiente de la función de coste en todos los parámetros de la red mediante la propagación del gradiente hacia atrás en la red (de ahí el nombre del algoritmo).

Matemáticamente, el algoritmo de *backpropagation* se puede definir como la aplicación de estos 3 pasos para cada una de las capas L de la red:

1. Cómputo del error de la última capa:

$$\delta^L = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z_i^L} \tag{A.2}$$

   Donde C es la función de coste, a^L es la activación de las neuronas de la capa L (el *output* de la capa L), y z_i^L es la suma ponderada de la neurona i de la capa L, ecuación 1.8. El término $\frac{\partial C}{\partial a^L}$ nos está diciendo cómo varía el coste de la red cuando se varía un poco el *output* de la neurona. Y el término $\frac{\partial a^L}{\partial z^L}$ nos dice como varía el *output* de la neurona cuando se varía un poco la suma ponderada de la

neurona. Combinando, δ^L nos dice en qué grado se modifica el coste de la red cuando se produce un pequeño cambio en la suma ponderada de la neurona. Es decir, δ^L nos dice qué responsabilidad tiene la neurona en el resultado final, y, por tanto, en el error. A este término se le suele llamar error imputado en la neurona i de la capa L.

2. Retropropagación del error a la capa anterior:

$$\delta^{L-1} = W^L \delta^L \cdot \frac{\partial a^{L-1}}{\partial z_i^{L-1}} \tag{A.3}$$

Donde W^L es la matriz de parámetros que conecta ambas capas y nos habla de cómo varía la suma ponderada de una capa cuando se varía el *output* de una neurona en la capa previa, y $\frac{\partial a^{L-1}}{\partial z_i^{L-1}}$ no es más que la derivada de la función de activación.

3. Cálculo de las derivadas de la capa usando el error:

$$\frac{\partial C}{\partial b^{L-1}} = \delta^{L-1}, \qquad \frac{\partial C}{\partial w^{L-1}} = \delta^{L-1} a^{L-2} \tag{A.4}$$

El error respecto al umbral b_i^L no es más que el error imputado en la neurona, ya que la suma ponderada z_i^L no varía con respecto al umbral. Es decir, $\frac{\partial z_i^L}{\partial b_i^L} = 1$. El error respecto a los pesos ω^L no es más que el error imputado en la neurona multiplicado por cómo varía la suma ponderada z_i^L con respecto a los pesos. Esta variación viene dada por la derivada parcial de la suma ponderada con respecto a los pesos, y no es más que el output de las neuronas de la capa anterior: $\frac{\partial z_i^L}{\partial w^L} = a^{L-1}$.

Apéndice B

Red neuronal tipo U-NET

Lo que nosotros queremos de nuestra red es que aprenda a generar la imagen real, es decir, la imagen reconstruida, a partir de la información contenida en la retroproyección de dicha imagen, es decir, generar, a partir de las imágenes del archivo `back_images.mat`, las imágenes del archivo `images.mat`. En el campo del *Deep Learning* suele usarse para generar imágenes un tipo de red conocida con el nombre de Red Neuronal Deconvolucional (Isola y col., 2017). Este tipo de red es muy similar a una CNN, sin embargo, su estructura es completamente inversa. Genera, a partir de un vector de números, una imagen, en vez de generar, a partir de una imagen, un vector de números. Esto lo hace haciendo uso de una operación conocida con el nombre de *upsampling* o *unpooling*, la operación inversa al *pooling*, descrita en la sección 1.6.3. Esta operación aumenta la resolución de cada píxel de información, justo la tarea inversa de la operación de *pooling*. Las diferencias que tienen tanto la convolución y la deconvolución, como el *pooling* y el *unpooling* se ilustra de manera esquemática en la figura B.1.

Por lo tanto, una Red Neuronal Deconvolucional tiene una estructura de cuello de botella inverso, justo al contrario que una CNN. Para generar imágenes a partir de otras imágenes se combinan ambas redes (que es lo que nosotros queremos), se combina una CNN con una Red Neuronal Deconvolucional. Dicha estructura recibe el nombre de *enconder-decoder* o codificador-descodificador, ya que la CNN se encarga de codificar la información de la imagen que utilizamos como *input*, y la Red Neuronal Deconvolucional se encargar de decodificar la información generada por la CNN para generar la imagen deseada. Un esquema de la arquitectura de una red *enconder-decoder* se muestra en la figura B.2.

Para crear una red U-NET a partir de un *enconder-decoder* solo falta añadir un elemento más la arquitectura ya descrita. Este elemento recibe el nombre de conexiones de salto (o *skip connections* en inglés). Lo que hacen las conexiones de salto es concatenar el *output* de cada nivel de procesamiento del *enconder* en el mismo nivel

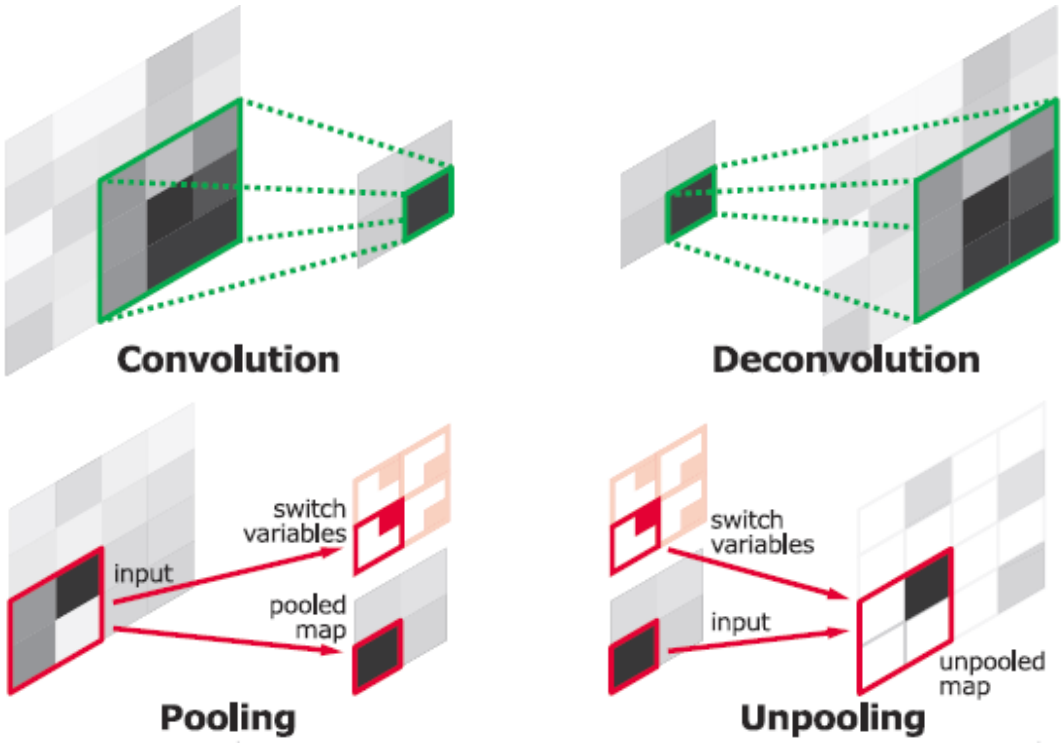


FIGURA B.1: **Arriba:** Operación de convolución y operación de deconvolución. **Abajo:** Operación de *pooling* y operación de *unpooling* (Tsang, 2018)

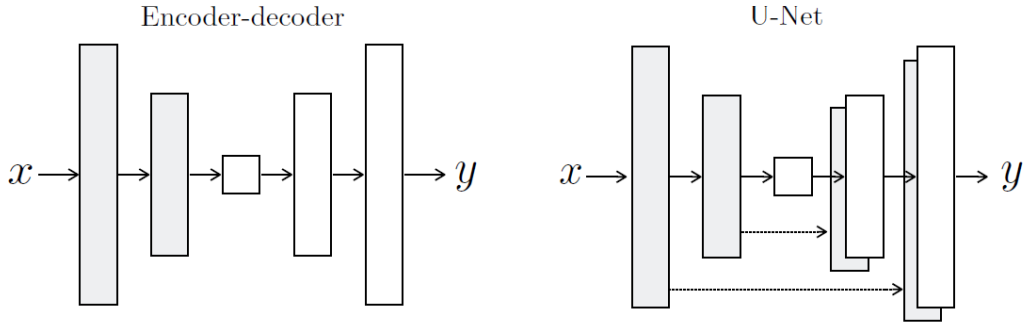


FIGURA B.2: Comparación entre las arquitecturas de una red tipo *enconder-decoder* y una U-NET. La U-NET es un *enconder-decoder* con conexiones de salto entre capas reflejadas del codificador y del descodificador (Isola y col., 2017).

equivalente del *decoder*, tal y como se ilustra en la figuras B.2 y B.3. Esto se hace para evitar la pérdida de información debido a la compresión al pasar por el cuello de botella, ya que comprimir demasiado la información puede dar lugar a la pérdida de información relevante que el *decoder* necesita para reconstruir la imagen. Así pues, ya tenemos todos los elementos que conforman una red tipo U-NET. Esta red fue desarrollada para aplicaciones en el campo de la segmentación de imagen biomédica en el Departamento de Ciencias de la Computación de la Universidad de Friburgo en el año 2015 (Ronneberger, Fischer y Brox, 2015). Un esquema más elaborado de la

arquitectura de este tipo de red se ilustra en la figura B.3.

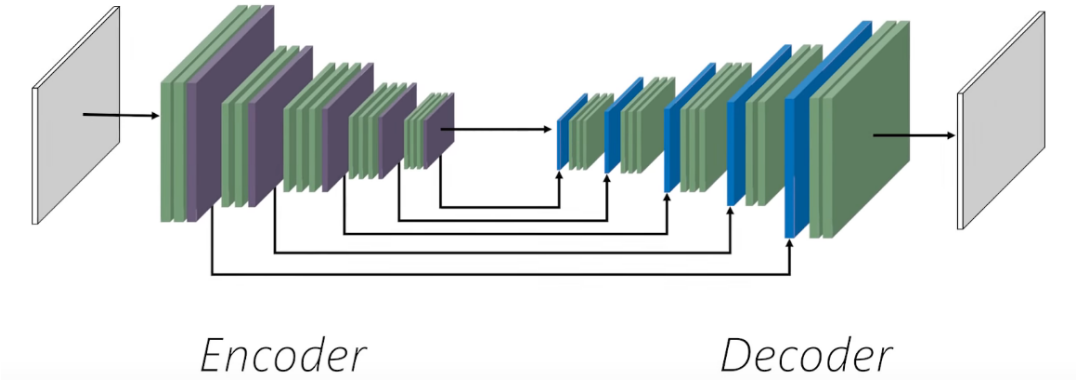


FIGURA B.3: Arquitectura de una red tipo U-NET (DotCSV, 2019).

Apéndice C

Código de Matlab

C.1. Simulación (sección 3.4)

```
%Anisotropic gamma ray emissions. More than one source.
%We aim to generate random numbers given some angular probability
%distribution, in this case, de probability distribution is
%p=a0+a2*cos(2*theta), where theta=-pi,...,pi; a0 =1; and a2 will depend on
%the degree of polarization (a2=0 -> no polarization, a2=1 -> maximun
%polarization)
%We also seek that the interarrivaltime (time elapsed between one emission
%and the following one) follows the distribution
%p(t)=lambda*exp(-lambda*t), where lambda is source's activity (5e6 Bq)
%https://en.wikipedia.org/wiki/Poisson_point_process
clear all, close all,

tic

act = 1e4;            %actividad de la fuente (Bq)
sens = 0.01;          %sensitivity (1%) (Geometrical and Detector)
time = 0.1;           %Acquisition Time (seconds) for each gradient
n_em = act*time;      %NUMBER OF EMISSIONS SIMULATED

%Polar plot of the probability distribution

a0 = 1;
a2 = 0.75;%This parameter defines the degree of polarization
          %(a2=0 -> no polarization, a2=1 -> maximun polarization)

%RANDOM NUMERS GENERATION

n_max = 5 * n_em;%WE CREATE MORE THAN NEEDED AND THEN WE PRUNE (THINNING)
n_bins = 50;%NUMBER OF BINS FOR HISTOGRAMS
```

```
30
31 %WE GENERATE EACH EVENT'S EMISSION ANGLE FOLLOWING GIVEN ANGULAR
       DISTRIBUTION
32 theta_value = 2 * pi * rand(n_max,1) - pi;%GENERATE n_em VALUES BETWEEN
        -pi y pi
33 P0 = (a0 - a2 * cos(2 * theta_value)) / (a0 + a2); %P(theta_value)
34 s = rand(n_max, 1);%generate a random number s between 0 and 1
35 theta_true = theta_value(s <= P0);%we select the values of theta that
       satisfy
36                                     %s <= P0(theta)
37 theta_true = theta_true(1:n_em);%we discard the excess to have the
       requested
38                                   %number of emissions(n_em)
39 P0_true = P0(s <= P0);%Same for P0 values
40
41 %MORE PARAMETERS
42
43 B = 0.1;%in Teslas (applied magnetic field)
44 gyro = 2 * pi * 1.37e6;%gyromagnetic ratio in radians per second per
       tesla for 131mXe
45
46
47 %% GRADIENT (applied all the time, not only for a fixed amount tau)
48 % The phase due to the gradient will continue changing over time
49
50 % PARAMTERS OF THE SIMULATION
51 Nx = 7;
52 Ny = 7;
53 Nvoxels = Nx*Ny;%Number of voxels (in this case only in x)
54 Ngradx = Nx;
55 Ngrady = Ny;
56 Ngrads = Nvoxels;%Number of gradiens applied (typically equal to
       Nvoxels)
57 gradsx = Nx*(2*pi/Ngradx)*linspace(-1,1,Ngradx);%Value of the gradient
       k in each case (k_x)
58 gradsy = Ny*(2*pi/Ngrady)*linspace(-1,1,Ngrady);%Value of the gradient
       k in each case (k_x)
59 %lineally spaced vector with Ngrads points, from -2pi to 2pi
60 sig = 0.2;%Sigma of the gaussian activity distribution
61
62 [posx,posy] = meshgrid(linspace(-2,2,Nx),linspace(-2,2,Ny));
63 %voxels from x=-2 to x=2, and y=-2 to y=2
64
65 Activity=act*exp(-0.5*((posx/sig).^2+(posy/sig).^2));%Amount of
       activity in each voxel
66 figure,imagesc(Activity),title('Activity vs Voxel Position'),colorbar
67
68 nev_all = 0;
```

```
for kx=1:Ngradx%Loop over the different gradients in x axis
for ky=1:Ngrady%Loop over the different gradients in y axis
    gradx = gradsx(kx);
    grady = gradsy(ky);
    for j=1:Nvoxels%Using a loop for now (Each voxel is like a new
    simulation)
        nev_j = floor(1.5*Activity(j)*time);%Aditional factor 1.5 just
    in case we dont get enough
        %nev_j: number of events for j simulation (integer, thats why
    floor is used)
        mu_j = 1.0/Activity(j);%Exponential distribution's parameter
        pt_j = exprnd(mu_j, [nev_j,1]);%We generate numbers (nev_j#
    numbers) with the interarrival time distribution
        t_j = cumsum(pt_j);%We get each event's time with the
    cumulative sum
        t_j = t_j(t_j<time);%We just select events up to acquisition
    time
        nev_j = length(t_j);%Set the number of emissions only up to
    acquisition time
        theta_offsets = (gradx*posx(j) + grady*posy(j) + gyro * B) *
    t_j;%Polar angle of the Spin over time
        ithetas = 1+floor(n_em*rand(nev_j,1));%Getting random angles
    from angular distribution
        theta_j = theta_true(ithetas) + theta_offsets;
        detections(nev_all+1:nev_all+nev_j,1)=t_j;%Storage of the
    result
        detections(nev_all+1:nev_all+nev_j,2)=theta_j;
        detections(nev_all+1:nev_all+nev_j,3)=j;%This is the
    information we want to reconstruct
        nev_all = nev_all+nev_j;
    end

    % We can sort the events by arrival time (as it happens in the
    detection)
    [~,idx] = sort(detections(:,1));% ort just the first column
    %~ is used so we dont store the sorted values, we just want the
    indixes
    detections_sorted{kx,ky} = detections(idx,:);%sort the whole matrix
     using the sort indices
end
end
Total_events = nev_all
%File needed for Inference (Reconstruction)
save('detections2D.mat','detections_sorted','gyro','B','time','posx','
    posy','gradsx','gradsy','a0','a2','-v7.3');
%The file includes all the information needed and known "a-priori"
%The reconstruction process seeks to obtain the j of the events from
    the
```

```
%time and angle information.
```

CÓDIGO C.1: Código de la simulación para el caso real (sección 3.4).

C.2. Reconstrucción iterativa (sección 3.5)

```
% Gamma-MRI project
% Reconstruction code (MLEM algorithm) - 2D case
clear all, close all,
load('detections2D.mat');%INFO %detections_sorted{gradx, grady}(time,
    angle, voxel)

Ngradx = length(gradsx);
Ngrady = length(gradsy);
Ngrads = Ngradx*Ngrady;
Nx = length(posx);
Ny = length(posy);
Nvoxels = Nx*Ny;

nev_tot = 0;%Total number of detections
for kx=1:Ngradx
for ky=1:Ngrady
 nev_tot = nev_tot + length(detections_sorted{kx,ky}(:,1));
end
end

X = ones(Nvoxels,1);% Reconstruction goal: Activity in each voxel
sensit = ones(Nvoxels,1)*Ngrads*time; %Improve this (for now, it's
    constant)
%with this sensitivity we obtain X in Bq units.

figure,
gif('reconstruction.gif')

Niter=50

for iter=1:Niter% Loop over the iterations of the recons
 iteration = iter

 for kx=1:Ngradx%Loop over the different gradients
 for ky=1:Ngrady%Loop over the different gradients
  sumlor = zeros(1,Nvoxels);%Initialization of the corrections array
  gradx = gradsx(kx);%Value of this gradient
  grady = gradsy(ky);%Value of this gradient
  detect = detections_sorted{kx,ky};%Getting the detections with this
    gradient
  nevs = length(detect);%Number of detections in this particular
    gradient
```

```
  c = zeros(nevs,Nvoxels);%Probability Matrix
  for j=1:Nvoxels
   theta_offsets = mod((gradx*posx(j) + grady*posy(j) + gyro * B) *
    detect(:,1),2*pi);%Polar angle of the Spin over time
   dtheta = detect(:,2) - theta_offsets;%Detection angle respect to the
     Spin for each voxel
   c(:,j)= a0 - a2*cos(2*dtheta);%Probability for each event and voxel
  end
  proj = c*X;%Projection (over all detected events)
  ratio = c./proj;%Backward Projection (list-mode)
  ratio(proj<eps)=0.;%Avoids NaNs
  sumlor = sumlor + sum(ratio,1);%Acumulate Corrections from each
    gradient

  update = sumlor'./sensit;%Normalize Corrections by Sensitivity
  X = X.*update;%Update Voxel Activity
 end
 end
 %update = sumlor'./sensit;%Normalize Corrections by Sensitivity
 %X = X.*update;%Update Voxel Activity

 %XT(iter,:) = X;
 iternumber = sprintf("Reconstructed Activity - Iteration = %i",iter);
 imagesc(reshape(X,Nx,Ny)),title(iternumber),colorbar
 drawnow
 gif
end

web('reconstruction.gif')
```

CÓDIGO C.2: Código de la reconstrucción iterativa de la imagen para el caso real (sección 3.5).

C.3. Conjunto de datos (sección 3.6.1)

```
fid=fopen('F18_V2_ALL.raw');%open file
X=fread(fid, 'float');
fclose(fid);
X=reshape(X, [154, 154, 80, 8]);%reshape data

max_value = max(X, [], 'all');
X= X/max_value;%normalize data between 0 and 1

 n = 25;%# of images per case
 n_img = 8*n ;%# of saved images (divisible by 8)

 images = [];%we will store the images here

```

```
 for i = 1:8
     random = round(rand(25, 1)*79) + 1;%generate random numbers
    between 1 and 80
     for j = 1:n
         img = X(:,:,random(j),i);%select random slice
         img = imresize(img, 64/154);%reshape slice
         img = img(23:38, 20:35);%select region of interest
         images = cat(3, images, img);%store the image
     end
 end


save('images.mat','images','-v7.3');
```

CÓDIGO C.3: Generación del archivo `images.mat`, sección 3.6.1.

```
clear all, close all,

tic

act = 1e3;%actividad de la fuente (Bq)
sens = 0.01;%sensitivity (1%) (Geometrical and Detector)
time = 0.1;%Acquisition Time (seconds) for each gradient
n_em = act*time;%NUMBER OF EMISSIONS SIMULATED

a0 = 1;
a2 = 0.75; %This parameter defines the degree of polarization
           %(a2=0 -> no polarization, a2=1 -> maximun polarization)

%RANDOM NUMERS GENERATION

n_max = 5 * n_em;% WE CREATE MORE THAN NEEDED AND THEN WE PRUNE (
    THINNING)
n_bins = 50;%NUMBER OF BINS FOR HISTOGRAMS

%WE GENERATE EACH EVENT'S EMISSION ANGLE FOLLOWING GIVEN ANGULAR
    DISTRIBUTION
theta_value = 2 * pi * rand(n_max,1) - pi;%GENERATE n_em VALUES BETWEEN
     -pi y pi
P0 = (a0 - a2 * cos(2 * theta_value)) / (a0 + a2);%P(theta_value)
s = rand(n_max, 1);%generate a random number s between 0 and 1
theta_true = theta_value(s <= P0);%we select the values of theta that
    satisfy
                                  %s <= P0(theta)
theta_true = theta_true(1:n_em);%we discard the excess to have the
    requested
                                %number of emissions(n_em)
P0_true = P0(s <= P0);%Same for P0 values

```

```
%MORE PARAMETERS

B = 0.1;%in Teslas (applied magnetic field)
gyro = 2 * pi * 1.37e6;%gyromagnetic ratio in radians per second per
    tesla for 131mXe

%% GRADIENT (applied all the time, not only for a fixed amount tau)
% The phase due to the gradient will continue changing over time
% PARAMTERS OF THE SIMULATION
Nx = 16;
Ny = 16;
Nvoxels = Nx*Ny;%Number of voxels (in this case only in x)
Ngradx = Nx;
Ngrady = Ny;
Ngrads = Nvoxels;%Number of gradiens applied (typically equal to
    Nvoxels)
gradsx = Nx*(2*pi/Ngradx)*linspace(-1,1,Ngradx);%Value of the gradient
    k in each case (k_x)
gradsy = Ny*(2*pi/Ngrady)*linspace(-1,1,Ngrady);%Value of the gradient
    k in each case (k_x)

[posx,posy] = meshgrid(linspace(-2,2,Nx),linspace(-2,2,Ny));%voxels
    from x=-2 to x=2, and y=-2 to y=2

back_images = [];
load('images.mat')
%n_img = 4;
n_img = length(images);

for i = 1:n_img
clearvars -except images back_images n_img i posx posy gradsx gradsy
    Ngrads Ngrady Ngradx Nvoxels Nx Ny gyro B theta_true act n_em time
    sens a0 a2

Activity = images(:,:,i);
Activity = im2double(Activity);
Activity = act*Activity;

nev_all = 0;
for kx=1:Ngradx%Loop over the different gradients
for ky=1:Ngrady%Loop over the different gradients
    fprintf("simulation: kx = %i ky= %i n_img = %i \n", kx, ky, i)
    gradx = gradsx(kx);
    grady = gradsy(ky);
    for j=1:Nvoxels%Using a loop for now (Each voxel is like a new
    simulation)
        nev_j = floor(1.5*Activity(j)*time);%Aditional factor 1.5 just
    in case we dont get enough
```

```
68          %nev_j: number of events for j simulation (integer, thats why
    floor is used)
69          mu_j = 1.0/Activity(j);%Exponential distribution's parameter
70          pt_j = exprnd(mu_j, [nev_j,1]);%We generate numbers (nev_j#
    numbers) with the interarrival time distribution
71          t_j = cumsum(pt_j);%We get each event's time with the
    cumulative sum
72          t_j = t_j(t_j<time);%We just select events up to acquisition
    time
73          nev_j = length(t_j);%Set the number of emissions only up to
    acquisition time
74          theta_offsets = (gradx*posx(j) + grady*posy(j) + gyro * B) *
    t_j;%Polar angle of the Spin over time
75          ithetas = 1+floor(n_em*rand(nev_j,1));%Getting random angles
    from angular distribution
76          theta_j = theta_true(ithetas) + theta_offsets;%Module 2pi can
    be used to simplify it (make that are angles belong to [0, 2pi)
77          detections(nev_all+1:nev_all+nev_j,1)=t_j;%Storage of the
    result
78          detections(nev_all+1:nev_all+nev_j,2)=theta_j;
79          detections(nev_all+1:nev_all+nev_j,3)=j;%This is the
    information we want to reconstruct
80          nev_all = nev_all+nev_j;
81      end
82
83      % We can sort the events by arrival time (as it happens in the
    detection)
84      [~,idx] = sort(detections(:,1));            % sort just the first
    column
85      %~ is used so we dont store the sorted values, we just want the
    indixes
86      detections_sorted{kx,ky} = detections(idx,:);    % sort the whole
    matrix using the sort indices
87  end
88  end
89
90  %The reconstruction process seeks to obtain the voxel j of the events
        from the
91  %time and angle information.
92  nev_tot = 0;    %Total number of detections
93  for kx=1:Ngradx
94  for ky=1:Ngrady
95   nev_tot = nev_tot + length(detections_sorted{kx,ky}(:,1));
96  end
97  end
98
99  X = ones(Nvoxels,1);    % Backprojected Image
100
```

```
for kx=1:Ngradx%Loop over the different gradients
for ky=1:Ngrady%Loop over the different gradients
  fprintf("backprojection: kx = %i ky= %i n_img = %i \n", kx, ky, i)
  sumlor = zeros(1,Nvoxels);%Initialization of the corrections array
  gradx = gradsx(kx);%Value of this gradient
  grady = gradsy(ky);%Value of this gradient
  detect = detections_sorted{kx,ky};%Getting all detected events with
    this gradient
  nevs = length(detect);%Number of detections in this particular
    gradient
  for j=1:Nvoxels
   theta_offsets = mod((gradx*posx(j) + grady*posy(j) + gyro * B) *
    detect(:,1),2*pi);%Polar angle of the Spin over time
   dtheta = detect(:,2) - theta_offsets;%Detection angle respect to the
     Spin for each voxel
   X(j) = X(j) + sum(a0 - a2*cos(2*dtheta),'all');%Probability for each
     event and voxel
  end
 end
end
back_img = reshape(X,Nx,Ny);

back_images = cat(3, back_images, back_img);

end

timeElapsed = toc

save('back_images.mat','back_images','-v7.3');
```

CÓDIGO C.4: Generación del archivo `back_images.mat`, sección 3.6.1.

Apéndice D

Código de la CNN en Python

```
# -*- coding: utf-8 -*-
"""Copia de Copia de TFM_inaki.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
    rH9G13RgyJlUTmvD6V4ABY1hcbg0hGT3

## U-NET for GAMMA-MRI simulations' image reconstruction

## STEP 0 ) Install all required libraries
"""

!pip install -q opencv-python
!pip install -q keras-unet

"""## STEP 1 ) Load Libraries"""

import tensorflow as tf
tf.version.VERSION
import numpy as np
import matplotlib.pyplot as plt
import h5py
import sklearn.model_selection as sk

"""## CHECK GPU"""

tf.config.list_physical_devices('GPU')

"""## IMAGE PARAMETERS"""

Nx = 16
Ny = 16

```

```
dx = 0.0280
dy = 0.0280
dz = 0.0280    #VOXEL SIZE (cm) ALONG X,Y,Z

"""## LOAD IMAGES"""

data_images = h5py.File('images.mat','r')
output = data_images.get('images')
output = np.array(output)

data_backimages = h5py.File('back_images.mat','r')
input = data_backimages.get('back_images')
input = np.array(input)

"""## INPUT NORMALIZATION [0..1]

Aqu  es donde tengo dudas, no se que tipo de normalizaci n hacer.
"""

#output already normalized between 0 and 1
#we normalize the input dividing by the global maximun (as we did in
    matlab for output)
input /= input.max()

"""## INPUT  /  OUTPUT

3-D Array with shape [batch, height, width]
"""

inp_np = np.expand_dims(input,axis=-1)
out_np = np.expand_dims(output,axis=-1)

"""## SPLIT DATA INTO TRAIN AND VALIDATION"""

x_train, x_val, y_train, y_val = sk.train_test_split(inp_np, out_np,
    test_size=0.2, shuffle=True)

"""## NUMPY TO TENSOR"""

x_train_tf = tf.convert_to_tensor(x_train, tf.float32)
x_val_tf = tf.convert_to_tensor(x_val, tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, tf.float32)
y_val_tf = tf.convert_to_tensor(y_val, tf.float32)

"""## SHOW EXAMPLE"""

from mpl_toolkits.axes_grid1 import make_axes_locatable

```

```
k=8   # SLICE SELECTED
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))

im = ax1.imshow(x_val_tf[k,:,:,0], cmap=plt.cm.bone)
ax1.set_title('Imagen Retroproyectada')
ax1.axis('off')
divider = make_axes_locatable(ax1)
cax1 = divider.append_axes("right", size="5%", pad=0.4)
fig.colorbar(im,cax=cax1)

im2 = ax2.imshow(y_val_tf[k,:,:,0], cmap=plt.cm.bone)
ax2.set_title('Imagen Original')
ax2.axis('off')
divider = make_axes_locatable(ax2)
cax2 = divider.append_axes("right", size="5%", pad=0.4)
fig.colorbar(im2,cax=cax2)

plt.show()

"""## U-NET"""

from keras_unet.models import custom_unet
from keras_unet.utils import get_augmented

"""### DATA AUGMENTATION"""

train_gen = get_augmented(x_train_tf, y_train_tf, batch_size=32,
    data_gen_args = dict(rotation_range=0.0, height_shift_range=0.,
    shear_range=0, horizontal_flip=True,vertical_flip=True))  #
    rotation_range=360,

"""### MODEL DEFINITION"""

model = custom_unet(
    input_shape=(Nx, Ny, x_train_tf.shape[3]),
    use_batch_norm=True,
    activation='swish',   #SWISH PROVIDES BETTER RESULTS THAN RELU
    filters=16,
    num_layers=4,
    use_attention=False,
    dropout=0.05,
    output_activation='relu')  #RELU IN THE OUTPUT (POSITIVE BUT NOT
    LIMITED TO [0..1])

model.summary()

"""### NEW OPTIMIZERS

```

```
Tambi n tengo dudas aqu  ya que no se que son los optimizers ni para que sirven.
"""

!pip install -q tfa-nightly
import tensorflow_addons as tfa
#opt = tfa.optimizers.RectifiedAdam(learning_rate=1e-3)
#opt = tfa.optimizers.Lookahead(opt)

#(Alternative option if the previous section does not work):
opt = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, amsgrad=False, name='Adam')

model.compile(optimizer=opt,loss='MeanAbsoluteError')   #l1 NORM

history = model.fit(train_gen,steps_per_epoch=100, epochs=50, validation_data=(x_val_tf, y_val_tf))

"""## DISPLAY OF RESULTS"""

img_index = 3
test = np.expand_dims(x_val_tf[img_index,:,:,:],axis=0)
estim = model.predict(test)
inp_img = np.squeeze(test[:,:,:,0])
estim_img = np.squeeze(estim[:,:,:,0])
out_img = np.squeeze(y_val_tf[img_index,:,:,:])

plt.figure(figsize=(20, 10))
plt.subplot(1,3,1)
plt.imshow(inp_img, cmap=plt.cm.bone)
#plt.imshow(inp_img, cmap='gray', vmin=0, vmax=1)
plt.axis('off')
plt.title('Input')
plt.colorbar()

plt.subplot(1,3,2)
#plt.imshow(estim_img, cmap=plt.cm.bone)
plt.imshow(estim_img, cmap='gray') #, vmin=0, vmax=1)
plt.axis('off')
plt.title('Estimated')
plt.colorbar()

plt.subplot(1,3,3)
#plt.imshow(out_img, cmap=plt.cm.bone)
plt.imshow(out_img, cmap='gray') #, vmin=0, vmax=1)
plt.axis('off')
plt.title('Reference')
plt.colorbar()
```

```
171
172 plt.show;
173
174 img_index = 8
175 test = np.expand_dims(x_val_tf[img_index,:,:,:],axis=0)
176 estim = model.predict(test)
177 inp_img = np.squeeze(test[:,:,:,0])
178 estim_img = np.squeeze(estim[:,:,:,0])
179 out_img = np.squeeze(y_val_tf[img_index,:,:,:])
180
181 fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 10))
182
183 im1 = ax1.imshow(inp_img, cmap=plt.cm.bone)
184 ax1.set_title('Input')
185 ax1.axis('off')
186 divider = make_axes_locatable(ax1)
187 cax1 = divider.append_axes("right", size="5%", pad=0.3)
188 fig.colorbar(im,cax=cax1)
189
190 im2 = ax2.imshow(estim_img, cmap='gray')
191 ax2.set_title('Estimaci n')
192 ax2.axis('off')
193 divider = make_axes_locatable(ax2)
194 cax2 = divider.append_axes("right", size="5%", pad=0.3)
195 fig.colorbar(im2,cax=cax2)
196
197 im3 = ax3.imshow(out_img, cmap='gray')
198 ax3.set_title('Referencia')
199 ax3.axis('off')
200 divider = make_axes_locatable(ax3)
201 cax3 = divider.append_axes("right", size="5%", pad=0.3)
202 fig.colorbar(im3,cax=cax3)
203
204 plt.show()
205
206 """## PLOT LOSS HISTORY"""
207
208 fig, ax = plt.subplots(figsize=(20,5))
209
210 loss = np.array(history.history['loss'])
211 var_loss = np.array(history.history['val_loss'])
212 ax.plot(loss, 'orange', label='Training Loss')
213 ax.plot(var_loss, 'green', label='Validation loss')
214 ax.set_ylim([0,0.25])
215 ax.legend()
216 fig.show()
217
218 """## SAVE LOSS HISTORY"""
```

```
219
220 loss = np.array(history.history['loss'])
221 var_loss = np.array(history.history['val_loss'])
222 loss_info = np.transpose(100*np.array([loss,var_loss]))
223 np.savetxt( "LOSS.csv", loss_info, fmt='%.3f', delimiter='\t')
224
225 """## SAVE MODEL"""
226
227 # Save the entire model as a HDF5 file.
228 model.save('deep_gammaMRI.h5')
229
230 """## LOAD MODEL"""
231
232 new_model = tf.keras.models.load_model('deep_gammaMRI.h5',compile=False
    )   # LOADING KERAS MODEL
```

CÓDIGO D.1: Código completo, en Python, de la CNN, de donde se han obtenido los resultados de la sección 4.5.

Apéndice E

Resumen del modelo

A continuación, en la figura E.1, se muestra el resumen del modelo (definido por el código 3.9) obtenido al ejecutar el código 4.1.

```
Model: "model"
```

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 16, 16, 1)]	0	
conv2d (Conv2D)	(None, 16, 16, 16)	144	input_1[0][0]
batch_normalization (BatchNorma	(None, 16, 16, 16)	64	conv2d[0][0]
spatial_dropout2d (SpatialDropo	(None, 16, 16, 16)	0	batch_normalization[0][0]
conv2d_1 (Conv2D)	(None, 16, 16, 16)	2304	spatial_dropout2d[0][0]
batch_normalization_1 (BatchNor	(None, 16, 16, 16)	64	conv2d_1[0][0]
max_pooling2d (MaxPooling2D)	(None, 8, 8, 16)	0	batch_normalization_1[0][0]
conv2d_2 (Conv2D)	(None, 8, 8, 32)	4608	max_pooling2d[0][0]
batch_normalization_2 (BatchNor	(None, 8, 8, 32)	128	conv2d_2[0][0]
spatial_dropout2d_1 (SpatialDro	(None, 8, 8, 32)	0	batch_normalization_2[0][0]
conv2d_3 (Conv2D)	(None, 8, 8, 32)	9216	spatial_dropout2d_1[0][0]
batch_normalization_3 (BatchNor	(None, 8, 8, 32)	128	conv2d_3[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 4, 4, 32)	0	batch_normalization_3[0][0]
conv2d_4 (Conv2D)	(None, 4, 4, 64)	18432	max_pooling2d_1[0][0]
batch_normalization_4 (BatchNor	(None, 4, 4, 64)	256	conv2d_4[0][0]
spatial_dropout2d_2 (SpatialDro	(None, 4, 4, 64)	0	batch_normalization_4[0][0]
conv2d_5 (Conv2D)	(None, 4, 4, 64)	36864	spatial_dropout2d_2[0][0]
batch_normalization_5 (BatchNor	(None, 4, 4, 64)	256	conv2d_5[0][0]
conv2d_transpose (Conv2DTranspo	(None, 8, 8, 32)	8224	batch_normalization_5[0][0]
concatenate (Concatenate)	(None, 8, 8, 64)	0	conv2d_transpose[0][0] batch_normalization_3[0][0]
conv2d_6 (Conv2D)	(None, 8, 8, 32)	18432	concatenate[0][0]
batch_normalization_6 (BatchNor	(None, 8, 8, 32)	128	conv2d_6[0][0]
conv2d_7 (Conv2D)	(None, 8, 8, 32)	9216	batch_normalization_6[0][0]
batch_normalization_7 (BatchNor	(None, 8, 8, 32)	128	conv2d_7[0][0]
conv2d_transpose_1 (Conv2DTrans	(None, 16, 16, 16)	2064	batch_normalization_7[0][0]
concatenate_1 (Concatenate)	(None, 16, 16, 32)	0	conv2d_transpose_1[0][0] batch_normalization_1[0][0]
conv2d_8 (Conv2D)	(None, 16, 16, 16)	4608	concatenate_1[0][0]
batch_normalization_8 (BatchNor	(None, 16, 16, 16)	64	conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 16, 16, 16)	2304	batch_normalization_8[0][0]
batch_normalization_9 (BatchNor	(None, 16, 16, 16)	64	conv2d_9[0][0]
conv2d_10 (Conv2D)	(None, 16, 16, 1)	17	batch_normalization_9[0][0]

```
Total params: 117,713
Trainable params: 117,073
Non-trainable params: 640
```

FIGURA E.1: Resumen del modelo, obtenido al ejecutar el código 4.1.

Bibliografía

Abragam, A. y M. Goldman (1978). «Principles of dynamic nuclear polarisation». En: *Reports on Progress in Physics* 41 (3). ISSN: 00344885. DOI: 10.1088/0034-4885/41/3/002.

Ahishakiye, Emmanuel y col. (2021). «A survey on deep learning in medical image reconstruction». En: *Intelligent Medicine*. ISSN: 26671026. DOI: 10.1016/j.imed.2021.03.003.

Albert, M. S. y col. (1994). «Biological magnetic resonance imaging using laser-polarized ^{129}Xe». En: *Nature* 370 (6486). ISSN: 00280836. DOI: 10.1038/370199a0.

Appelt, S. y col. (1998). «Theory of spin-exchange optical pumping of ^{3}He and ^{129}Xe». En: *Physical Review A - Atomic, Molecular, and Optical Physics* 58 (2). ISSN: 10502947. DOI: 10.1103/physreva.58.1412.

Asimov, I. (1941). *«Círculo vicioso». Los robots*. ISBN: 84-270-0906-2.

Berchane, N. (2018). *Artificial Intelligence, Machine Learning, and Deep Learning: Same context, Different concepts - Master Intelligence Economique et Stratégies Compétitives*. URL: https://master-iesc-angers.com/artificial-intelligence-machine-learning-and-deep-learning-same-context-different-concepts/.

Bishop, C.M. (2006). *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer. ISBN: 9780387310732. URL: https://books.google.es/books?id=qWPwnQEACAAJ.

Brown, Robert W. y col., eds. (abr. de 2014). *Magnetic Resonance Imaging: Physical Principles and Sequence Design*. John Wiley & Sons Ltd. DOI: 10.1002/9781118633953. URL: https://doi.org/10.1002/9781118633953.

Buchanan, Bruce G. (2005). «A (very) brief history of artificial intelligence». En: *AI Magazine* 26 (4). ISSN: 07384602.

Calaprice, F. P. y col. (1985). «Nuclear alignment and magnetic moments of ^{133}Xe, ^{133m}Xe, and ^{131m}Xe by spin exchange with optically pumped ^{87}Rb». En: *Physical Review Letters* 54 (3). ISSN: 00319007. DOI: 10.1103/PhysRevLett.54.174.

Cannizzaro, F. y col. (1978). «Results of the measurements carried out in order to verify the validity of the poisson-exponential distribution in radioactive decay events». En: *The International Journal Of Applied Radiation And Isotopes* 29 (11). ISSN: 0020708X. DOI: 10.1016/0020-708X(78)90101-1.

Chen, Hongyu, Melissa M. Rogalski y Jeffrey N. Anker (2012). *Advances in functional X-ray imaging techniques and contrast agents*. DOI: 10.1039/c2cp41858d.

Cherry, S.R., J.A. Sorenson y M.E. Phelps (2012). *Physics in Nuclear Medicine*. ClinicalKey 2012. Elsevier Health Sciences. ISBN: 9781416051985. URL: https://books.google.es/books?id=i794wmV6YQkC.

Dhillon, Anamika y Gyanendra K. Verma (2020). *Convolutional neural network: a review of models, methodologies and applications to object detection*. DOI: 10.1007/s13748-019-00203-0.

DotCSV (2019). *DeepNude, la IA que TE DESNUDA + (cGANs y Pix2Pix) - Data Coffee 11*. Youtube. URL: https://youtu.be/ysEjAqnHp64?t=1027.

— (2020). *¡APRENDE Qué son las Redes Neuronales CONVOLUCIONALES!* Youtube. URL: https://www.youtube.com/watch?v=V8j1oENVz00.

Dowsett, D., P.A. Kenny y R.E. Johnston (2006). *The Physics of Diagnostic Imaging Second Edition*. A Hodder Arnold Publication. Taylor & Francis. ISBN: 9780340808917. URL: https://books.google.es/books?id=Yewc5mchX34C.

Fedus, William, Barret Zoph y Noam Shazeer (2021). *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. arXiv: 2101.03961 [cs.LG].

Fukushima, Kunihiko (1980). «Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position». En: *Biological Cybernetics* 36 (4). ISSN: 03401200. DOI: 10.1007/BF00344251.

Geron, Aurelien (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media. ISBN: 978-1491962299.

Haynes, H. (2013). *Schematic diagram of an MRI machine illustrating the concentric arrangement of coils (360°) and magnet*. URL: https://www.researchgate.net/figure/Schematic-diagram-of-an-MRI-machine-illustrating-the-concentric-arrangement-of-coils_fig1_266266309.

Herraiz, Joaquín L., Adrián Bembibre y Alejandro López-Montes (2021). «Deep-Learning Based Positron Range Correction of PET Images». En: *Applied Sciences* 11.1. ISSN: 2076-3417. DOI: 10.3390/app11010266. URL: https://www.mdpi.com/2076-3417/11/1/266.

Hubel, D. H. y T. N. Wiesel (1968). «Receptive fields and functional architecture of monkey striate cortex». En: *The Journal of Physiology* 195 (1). ISSN: 14697793. DOI: 10.1113/jphysiol.1968.sp008455.

Isola, Phillip y col. (2017). «Image-to-image translation with conditional adversarial networks». En: vol. 2017-January. DOI: 10.1109/CVPR.2017.632.

Khapra, Mitesh M. (2018). *Perceptron: The Artificial Neuron*. URL: https://laptrinhx.com/perceptron-the-artificial-neuron-3673461102/.

Kingman, J.F.C. (1992). *Poisson Processes*. Oxford Studies in Probability. Clarendon Press. ISBN: 9780191591242. URL: https://books.google.es/books?id=VEiM-OtwDHkC.

Kissane, Jennifer, Janet A. Neutze y Harjit Singh, eds. (2020). *Radiology Fundamentals: Introduction to Imaging Technology*. Springer International Publishing. DOI: 10.1007/978-3-030-22173-7. URL: https://doi.org/10.1007/978-3-030-22173-7.

Kriegman, Sam y col. (2020). «A scalable pipeline for designing reconfigurable organisms». En: *Proceedings of the National Academy of Sciences of the United States of America* 117 (4). ISSN: 10916490. DOI: 10.1073/pnas.1910837117.

Lachaux, Marie Anne y col. (2020). *Unsupervised Translation of Programming Languages*.

Lange, K. y R. Carson (1984). «EM reconstruction algorithms for emission and transmission tomography». En: *Journal of Computer Assisted Tomography* 8 (2). ISSN: 03638715.

Lauterbur, P. C. (1973). «Image formation by induced local interactions: Examples employing nuclear magnetic resonance». En: *Nature* 242 (5394). ISSN: 00280836. DOI: 10.1038/242190a0.

Lecun, Yann, Yoshua Bengio y Geoffrey Hinton (2015). *Deep learning*. DOI: 10.1038/nature14539.

Levitt, M.H. (2008). *Spin Dynamics: Basics of Nuclear Magnetic Resonance*. 2.ª ed. Wiley. ISBN: 978-0-470-51117-6. URL: https://books.google.es/books?id=_1wZXxz1TIQC.

Oh, Kyoung Su y Keechul Jung (2004). «GPU implementation of neural networks». En: *Pattern Recognition* 37 (6). ISSN: 00313203. DOI: 10.1016/j.patcog.2004.01.013.

Ramachandran, Prajit, Barret Zoph y Quoc V. Le (2017). «Searching for Activation Functions». En: *CoRR* abs/1710.05941. arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941.

Raschka, Sebastian, Joshua Patterson y Corey Nolet (2020). *Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence*. DOI: 10.3390/info11040193.

Ronneberger, Olaf, Philipp Fischer y Thomas Brox (2015). «U-net: Convolutional networks for biomedical image segmentation». En: vol. 9351. DOI: 10.1007/978-3-319-24574-4_28.

Rosenblatt, F. (1957). *The Perceptron: A Perceiving and Recognizing Automaton (Project PARA). Report No. 85-460-1*. Cornell Aeronautical Laboratory.

Rumelhart, D.E, G.E Hinton y R.J Williams (1986). *Learning Internal Representations By Error Propagation (original)*.

Scherer, Dominik, Andreas Müller y Sven Behnke (2010). «Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition». En: *Artificial Neural Networks – ICANN 2010*. Ed. por Konstantinos Diamantaras, Wlodek Duch y Lazaros S. Iliadis. Berlin, Heidelberg: Springer Berlin Heidelberg, págs. 92-101. ISBN: 978-3-642-15825-4.

Sloss, Andrew N. y Steven Gustafson (2019). *2019 Evolutionary Algorithms Review*. DOI: 10.1007/978-3-030-39958-0_16.

Steenberg, N. R. (1952). «The angular distribution of -radiation from aligned nuclei». En: vol. 65. DOI: 10.1088/0370-1298/65/10/302.

Suzuki, Kenji (2017). *Overview of deep learning in medical imaging*. DOI: 10.1007/s12194-017-0406-5.

Taylor, Luke y Geoff Nitschke (2019). «Improving Deep Learning with Generic Data Augmentation». En: DOI: 10.1109/SSCI.2018.8628742.

Tsang, Sik-Ho (oct. de 2018). *Review: DeconvNet — Unpooling Layer (Semantic Segmentation)*. URL: https://towardsdatascience.com/review-deconvnet-unpooling-layer-semantic-segmentation-55cf8a6e380e.

Vallabhajosula, Shankar (2009). *Molecular Imaging - Radiopharmaceuticals for PET and SPECT*. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-76735-0. URL: https://doi.org/10.1007/978-3-540-76735-0.

Walker, Thad G. y William Happer (1997). «Spin-exchange optical pumping of noble-gas nuclei». En: *Reviews of Modern Physics* 69 (2). ISSN: 00346861. DOI: 10.1103/revmodphys.69.629.

Wells, Peter N.T. y Hai Dong Liang (2011). *Medical ultrasound: Imaging of soft tissue strain and elasticity*. DOI: 10.1098/rsif.2011.0054.

Willemink, Martin J. y Peter B. Noël (2019). «The evolution of image reconstruction for CT—from filtered back projection to artificial intelligence». En: *European Radiology* 29 (5). ISSN: 14321084. DOI: 10.1007/s00330-018-5810-7.

Wiström, Emma Linnea (sep. de 2020). «Developing gamma-MRI with the Hyperpolarization of ^{129}Xe and ^{131}Xe by Spin Exchange Optical Pumping». Tesis de mtría. University of Oslo.

Zahran, M. (2015). *A hypothetical example of Multilayer Perceptron Network*. URL: https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065.

Zheng, Yuan (sep. de 2014). «Low Field MRI and the Development of Polarized Nuclear Imaging (PNI) | A New Imaging Modality». Tesis doct. University of Virginia.

Zheng, Yuan y col. (2016). «A method for imaging and spectroscopy using γ-rays and magnetic resonance». En: *Nature* 537 (7622). ISSN: 14764687. DOI: 10.1038/nature19775.